Reqnroll

Reqnroll

May 21, 2026

CONTENTS

Reqnroll is an open-source .NET test automation tool to practice Behavior Driven Development (BDD).

Reqnroll is a .NET port of Cucumber and it is based on the SpecFlow framework and code base. You can find more information about the goal of the Reqnroll project and the motivations to create it on the Reqnroll website.

Reqnroll enables writing executable specifications for BDD using Gherkin, the widely-accepted *feature file* specification format. With that you can define the requirements using *Given-When-Then* style *scenarios* and turn them to automated tests in order to verify their implementation.

Reqnroll works on all major operating systems (Windows, Linux, macOS), on all commonly used .NET implementations (including .NET Framework 4.6.2+ and .NET 8.0). For executing the automated scenarios, Reqnroll can use MsTest, NUnit, TUnit or xUnit. On Reqnroll projects you can work using Visual Studio 2022/2026, Visual Studio Code and Rider, but you can also use Reqnroll without any IDE.

Since Reqnroll has been based on SpecFlow, you can use your SpecFlow knowledge to work with Reqnroll and it is also very easy to port an existing SpecFlow project to Reqnroll. You can check out our detailed *migration guide*.

This documentation provides a comprehensive source of information about how to use Reqnroll. We also recommend you to follow the news and the blog on the Reqnroll website.

CHAPTER

ONE

HOW TO USE THE DOCUMENTATION

Documentation suggestions

Feel free to suggest changes to our documentation! Each page contains a small (*edit*) icon to perform quick edits.

The documentation is structured in a way that you can find all relevant information quickly.

- In order to understand the concept of Reqnroll, we recommend checking out our *Quickstart Guide*.
- For setting up Reqnroll for your own project from scratch, you can find all details in the *Installation & Setup* section.
- To migrate an existing SpecFlow project to Reqnroll, please follow our *SpecFlow Migration Guide*.
- The *FEATURES* block covers all the details of Reqnroll features. There are separate sections about
  - the *Gherkin format*,
  - features for *writing automation code*,
  - features related to *test execution*, and
  - *extending the capabilities* of Reqnroll.

1.1 Quickstart

This guide gives a quick introduction to Reqnroll.

Note

The guide uses Visual Studio as an IDE, but you can also follow it with other tools.

In this tutorial we demonstrate the usage of Reqnroll by implementing the *price calculation* module of an *online instrument & accessories shop*.

1.1.1 Setup environment & get starting point

When you use Visual Studio for Reqnroll, please make sure you install the Reqnroll for Visual Studio extension. See *Setup Visual Studio* for details. Please make sure that the SpecFlow extension is disabled or removed for this Quickstart.

We prepared a simple starting point for this tutorial that you can find on GitHub: https://github.com/reqnroll/Quickstart. Clone this project to your machine or download it as zip and extract it to a local folder.

Open the solution file (`ReqnrollQuickstart.sln`) in Visual Studio 2022 and let's have a look at the content:

- The solution contains two projects: `ReqnrollQuickstart.App` is our application that we build, `ReqnrollQuickstart.Specs` contains the automated specification for it, so basically the Reqnroll acceptance tests. We will refer to this as Reqnroll project in this guide.
- The application contains one important class for now, the `PriceCalculator`, this is a very simple class, with an unfinished method for calculating the price.
- The Reqnroll project has a single specification file (called *feature file*), the `PriceCalculation.feature` in the `Features` folder, with our first scenario for the pricing module.

The Reqnroll project has been configured for using Reqnroll with MsTest. You can check the *Setup Reqnroll Project* guide for more details about the installation and setup options.

Make sure you build your solution, otherwise the feature file editor might behave incorrectly in Visual Studio.

Once you have done this, you should see these files.

Listing 1: PriceCalculator.cs

```
namespace ReqnrollQuickstart.App;

public class PriceCalculator
{
    // the item prices are hard coded for now
    private readonly Dictionary<string, decimal> _priceTable = new()
    {
        { "Electric guitar", 180.0 },
        { "Guitar pick", 1.5 }
    };

    public decimal CalculatePrice(Dictionary<string, int> basket)
    {
        throw new NotImplementedException();
    }
}
```

Listing 2: PriceCalculation.feature

```
Feature: Price calculation

This feature is about calculating the basket price.

We work with fixed item prices for now:
* Electric guitar: $180
* Guitar pick: $1.5

Rule: The price for a basket with items can be calculated based on the item prices

Scenario: Client has a simple basket
    Given the client started shopping
    And the client added 1 pcs of "Electric guitar" to the basket
    When the basket is prepared
    Then the basket price should be $180
```

1.1.2 Automating the first scenario

Run the tests from the Reqnroll project by opening the *Test Explorer* window (using the *Test / Test Explorer* menu command) and run all tests. You can find more details about running the tests in the *Executing Reqnroll Scenarios* guide.

The test execution reports a so called "undefined" state for our scenario. That means that Reqnroll has detected the scenario, but we did not *define* how the scenario steps should be automated. We will do this now.

In order to define the steps, we need to create a *step definition class*. This can be done be copying the code snippet from the test result output, but with Visual Studio we can also use the *Define Steps* dialog. You can access it by invoking the "Define Steps…" command from the feature file editor context menu or with the *Ctrl+B, D* keyboard shortcut from the editor.

1.1.3 Generate step definition snippets

For now, we can simply accept the suggestion provided by the *Define Steps* dialog by clicking to the *Create* button.

This will create a new class `PriceCalculationStepDefinitions` in the `StepDefinitions` folder.

The class contains suggestions provided by the Visual Studio extension. In many cases the suggestions are just perfect and you don't need to change them. In some other cases, like in ours, you need to make some small corrections on the generated names and types.

In our case we can provide more meaningful parameter names (instead of `p0` and `p1`). You can see the updated parameter names in the emphasized lines below.

After converting it to file-scoped namespace, the generated snippet looks like this.

Listing 3: PriceCalculationStepDefinitions.cs

```
namespace ReqnrollQuickstart.Specs.StepDefinitions;

[Binding]
public class PriceCalculationStepDefinitions
{
    [Given("the client started shopping")]
    public void GivenTheClientStartedShopping()
```

(continues on next page)

(continued from previous page)

```
    {
        throw new PendingStepException();
    }

    [Given("the client added {int} pcs of {string} to the basket")]
    public void GivenTheClientAddedPcsOfToTheBasket(int quantity, string product)
    {
        throw new PendingStepException();
    }

    [When("the basket is prepared")]
    public void WhenTheBasketIsPrepared()
    {
        throw new PendingStepException();
    }

    [Then("the basket price should be ${float}")]
    public void ThenTheBasketPriceShouldBe(decimal expectedPrice)
    {
        throw new PendingStepException();
    }
}
```

As you can see, each step in our scenario has a corresponding method, called a *step definition method*. These are currently unfinished, but guide us to provide the necessary automation code to verify our application. You can find more information about step definitions in the *Step Definitions* guide.

> **Note**
>
> After adding step definitions or changing their expressions, you have to build the project in Visual Studio, so that the changes are shown in the feature file editor.

1.1.4 Prepare fields for the step definitions

Let's provide the automation code. First, let's declare a few class-level fields.

Listing 4: PriceCalculationStepDefinitions.cs

```
namespace ReqnrollQuickstart.Specs.StepDefinitions;

[Binding]
public class PriceCalculationStepDefinitions
{
    private readonly PriceCalculator _priceCalculator = new();
    private readonly Dictionary<string, int> _basket = new();
    private decimal _calculatedPrice;

    [...]
}
```

These fields will be used for different purposes:

- The field `_priceCalculator` contains the module class that we would like to test.

- The `_basket` field will be used to collect the item/s the client puts in the basket, an item is a pair of product and quantity.
- The `_calculatedPrice` field will contain the price calculated by the application, so that we can make assertions for it.

These fields will provide data (or with other word *state*) for the step definitions. For now, all our step definition methods were in the same class, therefore declaring them as simple class-level fields was enough. For learning more about sharing data between step definition methods please check the *Sharing Data between Bindings* guide.

1.1.5 Automate steps

Now let's provide the automation code for the steps. Our plan is the following:

- In the "Given" steps we will prepare the items in the basket,
- in the "When" step we invoke the `CalculatePrice` method of our price calculator class and save the result, and
- in the "Then" step we make sure that the saved price is the same as what we expected using an assertion.

After adding all these, our code looks like this (changes emphasized):

Listing 5: PriceCalculationStepDefinitions.cs

```
namespace ReqnrollQuickstart.Specs.StepDefinitions;

[Binding]
public class PriceCalculationStepDefinitions
{
    private readonly PriceCalculator _priceCalculator = new();
    private readonly Dictionary<string, int> _basket = new();
    private decimal _calculatedPrice;

    [Given("the client started shopping")]
    public void GivenTheClientStartedShopping()
    {
        _basket.Clear();
        _calculatedPrice = 0.0m;
    }

    [Given("the client added {int} pcs of {string} to the basket")]
    public void GivenTheClientAddedPcsOfToTheBasket(int quantity, string product)
    {
        _basket.Add(product, quantity);
    }

    [When("the basket is prepared")]
    public void WhenTheBasketIsPrepared()
    {
        _calculatedPrice = _priceCalculator.CalculatePrice(_basket);
    }

    [Then("the basket price should be ${float}")]
    public void ThenTheBasketPriceShouldBe(decimal expectedPrice)
    {
        Assert.AreEqual(expectedPrice, _calculatedPrice);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

> **Note**
>
> In our example we call methods of our application code from the step definitions. In other projects, you might need to invoke REST HTTP requests there or interact with a web browser in the step definitions. Reqnroll does not prescribe any particular automation model.

1.1.6 Run tests and implement application code

We seem to have completed our automation code, still if we run our tests it shows an error.

Listing 6: Test Output

```
Test method ReqnrollQuickstart.Specs.Features.PriceCalculationFeature.
↪ClientHasASimpleBasket threw exception:
System.NotImplementedException: The method or operation is not implemented.
```

> **Hint**
>
> What we have done so far was a *test-first development* that might be known to you from Test-Driven Development (TDD). We automated the scenario (the "test") before implementing the production code. You can use Reqnroll for "test-after" development as well, but we encourage you to try test-first, because the automated tests can help to shape the implementation and can help to avoid unnecessary or unused code.

Our test fails, because we haven't implemented the price calculation module yet.

In our case it would be easy to implement the "final" version of the calculation module immediately, but currently our scenario illustrates a very simple case, when we only add a single item of a product to the basket. For complex or complicated system the "final" solution that you have in your mind might not be the best one, so it is better to make it iteratively. Let's imagine that we have a complex system, and therefore we will start with a temporary, basic implementation for now.

Open the `PriceCalculator` class and add the emphasized lines from the code below.

Listing 7: PriceCalculator.cs

```
namespace ReqnrollQuickstart.App;

public class PriceCalculator
{
    // the item prices are hard coded for now
    private readonly Dictionary<string, decimal> _priceTable = new()
    {
        { "Electric guitar", 180.0m },
        { "Guitar pick", 1.5m }
    };

    public decimal CalculatePrice(Dictionary<string, int> basket)
    {
```

(continues on next page)

(continued from previous page)

```
        //TODO: complete the price calculation once we defined more scenarios
        var item = basket.First();
        return _priceTable[item.Key];
    }
}
```

1.1.7 Add a new scenario and extend code

So it is time to add a new scenario where the client has multiple items in the basket. The scenario can be drafted as:

Listing 8: New scenario

```
Scenario: Client has multiple items in their basket
    Given the client started shopping
    And the client added
        | product         | quantity |
        | Electric guitar | 1        |
        | Guitar pick     | 10       |
    When the basket is prepared
    Then the basket price should be $195.0
```

Where should we document this scenario?

This scenario is also related to price calculation, so we should include it to our `PriceCalculation.feature` file, but let's look at the current structure of the file. You can notice that it also contains a `Rule` block. Rules are optional in Gherkin but they are very useful to group the scenarios by acceptance criteria. You can learn more about the `Rule` keyword in the *Rule* page.

Currently we have a single rule: "The price for a basket with items can be calculated based on the item prices" and it is clear that the new scenario also belongs to that, so we can just include it to the end of the rule block (that is in our case the end of the file). Later we might need to introduce additional rules, like applying discounts.

Listing 9: PriceCalculation.feature

```
Feature: Price calculation
[...]
Rule: The price for a basket with items can be calculated based on the item prices

Scenario: Client has a simple basket
[...]

Scenario: Client has multiple items in their basket
    Given the client started shopping
    And the client added
        | product         | quantity |
        | Electric guitar | 1        |
        | Guitar pick     | 10       |
    When the basket is prepared
    Then the basket price should be $195.0
```

Visual Studio shows most of the steps of the new scenario with default font color, except the "And the client added" step. This is because all other steps have been already used in our other scenario as well, so we can automatically reuse the automation we provided for them. Great! But the "And the client added" step is still *undefined*. This is a special step as it contains an attached tabular parameter with the products and the quantities to be added to the basket. This

parameter is called *Data Table* in Gherkin and you can read more about it in the *Data Tables* section of our Gherkin page.

> **Tip**
>
> You can easily find the step definition method of a *defined* step by invoking the *Go To Definition* command from the context menu of the step. And once you are at the step definition, the *Find Step Definition Usages* command shows where it was used.

Actually even undefined step could have been rephrased in a way that we use only existing steps. We could have written:

Listing 10: Building basket with multiple items using existing steps

```
And the client added 1 pcs of "Electric guitar" to the basket
And the client added 10 pcs of "Guitar pick" to the basket
```

This way of phasing becomes cumbersome with multiple items. The one with the data table is nicer. But we need to define it still.

We can use the *Define Steps* dialog as before, but to extend an existing step definition class with a new snippet, you need to click on the *Copy to clipboard* button on the dialog and paste the snippet to our step definition class, for example right after the other "Given" step dealing with basket addition.

The content of the data table is provided as a parameter of type `DataTable`. We can rename the parameter to `itemsTable`.

Listing 11: PriceCalculationStepDefinitions.cs

```
[...]
public class PriceCalculationStepDefinitions
{
    [...]

    [Given("the client added {int} pcs of {string} to the basket")]
    public void GivenTheClientAddedPcsOfToTheBasket(int quantity, string product)
    {
        _basket.Add(product, quantity);
    }

    [Given("the client added")]
    public void GivenTheClientAdded(DataTable itemsTable)
    {
        throw new PendingStepException();
    }

    [When("the basket is prepared")]
    public void WhenTheBasketIsPrepared()
    {
        _calculatedPrice = _priceCalculator.CalculatePrice(_basket);
    }

    [...]
}
```

For handling data tables you can find more information in the *Data Table or DocString Arguments* section of the step

definition guide. In this Quickstart we use one of the *DataTable Helpers* method to convert the table structure to a strongly typed structure (a list of tuples).

Listing 12: PriceCalculationStepDefinitions.cs

```
[...]
public class PriceCalculationStepDefinitions
{
    [...]

    [Given("the client added")]
    public void GivenTheClientAdded(DataTable itemsTable)
    {
        var items = itemsTable.CreateSet<(string Product, int Quantity)>();
        foreach (var item in items)
        {
            _basket.Add(item.Product, item.Quantity);
        }
    }

    [...]
}
```

Let's run the tests now. As we expected, the first scenario still passes, but the new one fails, because our basic implementation of the calculator does not support this case yet.

Listing 13: Test Output

```
Assert.AreEqual failed. Expected:<195.0>. Actual:<180.0>.
```

Now based on this example we can complete the calculation method.

Listing 14: PriceCalculator.cs

```
public class PriceCalculator
{
    [...]

    public decimal CalculatePrice(Dictionary<string, int> basket)
    {
        decimal price = 0;
        foreach (var item in basket)
        {
            price += _priceTable[item.Key] * item.Value;
        }
        return price;
    }
}
```

Now both of our tests pass!

1.1.8 Next Steps

Congratulations! You have completed our Quickstart tutorial and now you have a working Reqnroll automation solution that you can experiment with.

If you get lost, you can check out our sample result in the `completed` branch of our Quickstart repository.

If you need inspirations how to extend the solution, here are a few ideas:

- Consider introducing a `Currency` class and create a argument transformation that recognizes currencies like `$195.0` and converts them to a currency value. You can find more about step argument transformations in *Step Argument Conversions*.
- You can replace the hard-coded product prices with "Given" steps that describe the available products and their prices. You can also use the *Background* section for that.
- You can consider implementing a new rule that provides 10% discount when the basket value is over $200. Separate their scenarios with the *Rule keyword*.
- If you are really adventureous, you can turn the app into a backed service that provides the price calculation as a REST HTTP service. In this case that step definitions can make HTTP requests to test the service. In that case you can use the `BeforeScenario` and `AfterScenario` *hooks* to start and stop the application.

Share your results at our Reqnroll discussion topic!

1.2 Installation & Setup

Reqnroll is distributed as a set of NuGet packages that you need to configure for your project, for most of the cases there is no additional configuration required. It is also recommended to configure your Integrated Development Environment (IDE, e.g. Visual Studio 2022) to work conveniently with Reqnroll.

1.2.1 Setup Reqnroll Project

This page guides you through setting up your Reqnroll project. It is also recommended to configure your Integrated Development Environment (IDE, e.g. Visual Studio 2022) to work conveniently with Reqnroll. To set up your IDE, please follow the *Setup an IDE for Reqnroll* guide first.

Choosing your test execution framework

Reqnroll uses *test execution frameworks* (MsTest, NUnit, TUnit or xUnit) to run the tests. So first of all, you need to decide, which one you would like to use. Reqnroll does not have a favorite one, so you should better choose the one you have the most experience with. If you don't have any preference, choose NUnit. The following table gives you a quick comparison of the different supported execution frameworks.

Framework	NuGet package	Description
NUnit	`Reqnroll.NUnit`	Easy to use testing framework with respectful history. Supports test attachments and comes with an extensive assertion library.
MsTest	`Reqnroll.MsTest`	A widely supported framework by Microsoft. Supports test attachments and input parameters through the `TestContext` class.
TUnit	`Reqnroll.TUnit`	A modern testing framework for .NET built with performance in mind. Leveraging source generation and AOT compilation for efficient test execution.
xUnit	`Reqnroll.xUnit`	Simple and modern testing framework that reports the original names of the scenarios during execution. It does not support test attachments and writing test execution output cannot be done with `Console.WriteLine`, so it is less practical for integration tests.

> **Note**
>
> If you changed your mind and you would like to switch to another test execution framework, check out the *How to change the test execution framework used by Reqnroll* guide. Using independent assertion frameworks, like Fluent Assertions makes the change much easier.

> **Note**
>
> Reqnroll converts the Gherkin feature files to test classes via code-behind (`*.feature.cs`) files. These code-behind files are generated by default to the project folder structure, next to the feature files, but since these are generated files, it is not recommended to include them to source-control. From v3.3 you can configure these files to be generated to the intermediate output folder (e.g. `obj/Debug/net8.0`), by setting the `ReqnrollUseIntermediateOutputPathForCodeBehind` MsBuild property to `true` in the project file. According to the plans, this will be the default behavior from v4.
>
> Including linked feature files (files used from outside of the project folder) is also supported from v3.3, but only when the code-behind files are generated to the intermediate output folder using the setting from above.

Setting up a Reqnroll project

Once you have chosen your test execution framework, you need to setup the Reqnroll project. This can be done by creating a new Reqnroll project or setup an existing .NET test project.

Depending on your situation, you can find the necessary setup instructions in one of the following sub-sections.

- *Creating a new Reqnroll project from Visual Studio*
- *Creating a new Reqnroll project from console*
- *Setup an existing test project*

Creating a new Reqnroll project from Visual Studio

If you have installed the *Reqnroll for Visual Studio*, you can easily create a new Reqnroll project using the *Add new project* wizard, by performing the following steps:

1. From the context menu of your solution in the *Solution Explorer* window select *Add / New Project...*
2. In the *Add a new project* dialog enter `Reqnroll` to the *Search for templates* text box.
3. Choose *Reqnroll Project" from the list and click on *Next* button.
4. Follow the wizard by choosing the name, the target framework and the test framework for the project.

As a result, a new Reqnroll project is created with a sample *feature file* and *step definition class*.

Build your project and *execute the sample scenarios*.

Creating a new Reqnroll project from console

Reqnroll projects can also be installed using the .NET template infrastructure and the `dotnet new` command.

First, you need to make sure that the Reqnroll templates are installed on your computer by running the `dotnet new install Reqnroll.Templates.DotNet`:

Listing 15: .NET CLI

```
dotnet new install Reqnroll.Templates.DotNet
```

Once the templates have been installed, you can create a new project using the `dotnet new reqnroll-project` command in a new directory.

Listing 16: Terminal

```
> mkdir MyReqnrollProject
> cd MyReqnrollProject
> dotnet new reqnroll-project
The template "Reqnroll Project Template" has been created successfully.
```

This command creates a Reqnroll project with NUnit for the latest .NET framework. In order to use other test execution framework or .NET version, you can use the `-t` and the `-f` option. For the possible values and all options, you can invoke `dotnet new reqnroll-project --help`.

Listing 17: .NET CLI

```
dotnet new reqnroll-project --help
```

The following command creates a Reqnroll project with MsTest using .NET 8.0

Listing 18: .NET CLI

```
dotnet new reqnroll-project -t mstest -f net8.0
```

As a result, a new Reqnroll project is created with a sample *feature file* and *step definition class*.

You can go ahead and *execute the sample scenarios* from console.

To add further feature file or a *Reqnroll configuration file*, you can also use the `reqnroll-feature` and `reqnroll-config` item templates. The following command adds a new feature file to the project named `MyFeature.feature`.

Listing 19: .NET CLI

```
dotnet new reqnroll-feature -n MyFeature
```

Setup an existing test project

Reqnroll can also be configured for an existing test project. For that you need to add the NuGet package of your chosen test execution framework to your project. (Check the NuGet package names *above*). The chosen test execution framework has to match the framework used in your existing test project.

The following example adds the Reqnroll NuGet package for an MsTest project.

Listing 20: .NET CLI

```
dotnet add package Reqnroll.MsTest
```

Although the Reqnroll tests can be mixed with normal unit tests in the same .NET project, for the sake of clarity it is recommended to create a separate project for your Reqnroll BDD scenarios.

1.2.2 Setup an IDE for Reqnroll

> **Tip**
>
> Reqnroll can be used without any IDE integration as well, so setting up the IDE is optional.

Setting up the Integrated Development Environment (IDE) integration for Reqnroll can add convenience and productivity features like:

- Adding new project elements, like feature files based on templates
- Syntax coloring of feature files
- Showing suggestions (completions) for Gherkin syntax keywords
- Navigating between steps and step definitions
- Adding step definition snippets to your codebase for undefined steps

This guide describes the setup steps for the following IDEs:

- *Setup Visual Studio*
- *Setup Visual Studio Code*
- *Setup Rider*

Setup Visual Studio

In order to use Reqnroll with Visual Studio 2022 or Visual Studio 2026, you need to install the Reqnroll for Visual Studio extension.

> **Warning**
>
> The *Reqnroll with Visual Studio* extension cannot work together with the *SpecFlow for Visual Studio 2022* extension, as they both process feature files. As the Reqnroll extension also supports SpecFlow projects, you can remove the SpecFlow extension if you install the Reqnroll extension. Alternatively, you can disable the SpecFlow extension for the time you work with Reqnroll.

1. Open Visual Studio
2. From the *Extensions* menu, choose the *Manage Extensions...* command.
3. On the dialog, make sure that *Online* is selected from the list on the left and type `Reqnroll` to the *Search* text box on the right top corner.
4. Choose the *Reqnroll for Visual Studio* from the list and click on the *Download* button.
5. Restart Visual Studio.

For more details about the Reqnroll with Visual Studio extension, please check the *Reqnroll for Visual Studio* page.

> **Hint**
>
> The Reqnroll Visual Studio extension cannot be used for Visual Studio for Mac. On macOS we recommend using *Visual Studio Code*.

Setup Visual Studio Code

For using Reqnroll with Visual Studio Code, you can choose from multiple available extensions. We recommend using the Cucumber extension.

In order to use the navigation features of the extension, you should configure the location of your feature files and step definition classes within your repository.

The following Visual Studio configuration shows a typical configuration.

Listing 21: .vscode/settings.json

```
{
  "explorer.fileNesting.enabled": true,
  "explorer.fileNesting.patterns": {  // shows *.feature.cs files as nested items
    "*.feature": "${capture}.feature.cs"
  },
  "files.exclude": { // excludes compilation result
    "**/obj/": true,
    "**/bin/": true,
  },
  "cucumber.glue": [ // sets the location of the step definition classes
    "MyReqnrollProject/**/*.cs",
  ],
  "cucumber.features": [ // sets the location of the feature files
    "MyReqnrollProject/**/*.feature",
  ]
}
```

Setup Rider

In order to use Reqnroll with Rider, you need to install the Reqnroll for Rider extension.

> **Warning**
>
> The *Reqnroll with Rider* extension cannot work together with the *SpecFlow for Rider* extension, as they both process feature files. As the Reqnroll extension also supports SpecFlow projects, you can remove the SpecFlow extension if you install the Reqnroll extension. Alternatively, you can disable the SpecFlow extension for the time you work with Reqnroll.

1. Launch Rider and ensure you are using a compatible version. The following versions have been verified to work with Reqnroll:
   - Rider compatibility
2. Top right of Rider click the gear icon and press plugins.
3. Click `Marketplace`.
4. Enter `Reqnroll` in the search box and install.
5. Open csproj and verify your project contains

   ```
   <ItemGroup>
     <Content Include="**/*.feature"/>
   </ItemGroup>
   ```

   - This is a work around for a known issue found here Content Include .feature files in .csproj

6. Restart Rider.
7. *(optional)*: If your `.feature` files aren't recognized by the plugin:
    - Right click on a `.feature` file in the explorer then `Associate with File Type...`
    - Make sure `Open matching files in JetBrains Rider` is selected
    - Select `Reqnroll file` in the list and click `OK`
    - This will add a new File name pattern under `Settings -> Editor -> File Types -> Reqnroll file`

1.2.3 Configuration

Reqnroll can be *setup* simply by adding a NuGet package to your project and in the most of the cases there is no additional configuration required.

The default configuration can be altered by adding a `reqnroll.json` configuration file to your project. An empty configuration file can be added using the *Add / New Item...* command of Visual Studio or using the Reqnroll .NET item template. The following example downloads the Reqnroll templates and adds a configuration file to the project.

Listing 22: .NET CLI

```
dotnet new install Reqnroll.Templates.DotNet
dotnet new reqnroll-config
```

You can also start by adding the following empty configuration file to your project.

Listing 23: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json"
}
```

Tip

There is a JSON schema file available for `reqnroll.json`. By specifying the schema reference like in the example above, most IDE (including Visual Studio and Visual Studio Code) provides auto completion and documentation hints for the configuration file.

In this guide we show examples for the most common situations when you need to modify the config file followed by a full configuration reference.

- *Use bindings from external projects*
- *Set the default feature file language*
- *Configuration file reference*

Use bindings from external projects

In order to use bindings (step definitions, hooks or step argument transformations) from other projects (called *external projects*) it is not enough to add a project reference to the Reqnroll project, but you need to also configure Reqnroll to search bindings in these projects. See *Bindings from External Assemblies* for further details.

This can be achieved by listing the *assembly name* of the external project to the `bindingAssemblies` section of the configuration file.

The following example registers the project `SharedStepDefinitions` as an external binding assembly.

Listing 24: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

  "bindingAssemblies": [
    {
      "assembly": "SharedStepDefinitions"
    }
  ]
}
```

Set the default feature file language

The keywords in the *feature files* are available in many many natural languages matching the language your business is using.

In order to use the keywords in a language other than English, you can either use the *Gerkin `#language` directive* in every feature file or specify a default language in the Reqnroll configuration.

The following example sets the default feature language to Hungarian:

Listing 25: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

  "language": {
    "feature": "hu-HU"
  }
}
```

Configuration file reference

The following configuration sections are available for `reqnroll.json`.

`language`

Use this section to define the default language for feature files and other language-related settings. For more details on language settings, see *Feature Language*.

Setting	Value	Description
feature	culture name (`en-US`)	The default language of feature files added to the project. We recommend using specific culture names (e.g.: `en-US`) rather than generic (neutral) cultures (e.g.: `en`). *Default:* `en-US`
binding	culture name (`en-US`)	Specifies the culture to be used to execute binding methods and convert step arguments. If not specified, the feature language is used. *Default:* not specified

generator

Use this section to define test generation options.

Setting	Valu	Description
allowDebugGeneratedFiles	`true`	By default, the debugger is configured to step through the generated code. This helps you debug your feature files and bindings (see *Debugging Tests*). Disabled this option by setting this attribute to `true`. *Default:* `false`
allowRowTe	`true`	Determines whether "row tests" should be generated for *scenario outlines*. This setting is ignored if the *test execution framework* does not support row based testing. *Default:* `true`
addNonParallelizableMarkerForTags	List of tags	Defines a set of tags that mark tests as exclusive (non-parallelizable). If the tag appears on a feature, the whole generated test class is marked non-parallelizable. If the tag appears only on a scenario (and not on the feature), the generated test method is marked non-parallelizable on frameworks that support scenario/method level isolation (currently NUnit, MsTest V2 and TUnit). See *Parallel Execution*. *Default:* empty
disableFriendlyTestNames	`true`	Option available in Reqnroll versions **greater than v3.0.3**. Determines whether generated tests will contain a `"DisplayName"` in their test attributes with the name of the Scenario. Friendly names are easier to read in test reports, but may cause issues with some test runners or CI systems. Friendly display names are currently only supported by MSTest and xUnit. *Default:* `false`

runtime

Use this section to specify various test execution options.

Setting	Value	Description
missingOrPendingStepsOutcome	`Pending` / `Inconclusive` / `Ignore` / `Error`	Determines how Reqnroll behaves if a step binding is not implemented or pending. See *Test Results*. *Default:* `Pending`
obsoleteBehavior	`None` / `Warn` / `Pending` / `Error`	Determines how Reqnroll behaves if a step binding is marked with `[Obsolete]` attribute. *Default:* `Warn`
stopAtFirstError	`true/false`	Determines whether the execution of the scenario should stop when encountering the first error, or whether it should attempt to try and match subsequent steps (in order to detect missing steps). *Default:* `false`

trace

Use this section to determine the Reqnroll trace output.

Setting	Value	Description
stepDefinitionSkeletonStyle	`CucumberExpressionAttribute` / `RegexAttribute` / `AsyncCucumberExpressionAttribute` / `AsyncRegexAttribute`	Specifies the default *step definition style*. *Default:* `CucumberExpressionAttribute`
coloredOutput	`true`/`false`	Determine whether Reqnroll should color the test result output. See *Color Test Result Output* for more details. You can override this setting to disable color (e.g. on build servers), with the environment variable `NO_COLOR=1` *Default:* `false`

`bindingAssemblies`

This section can be used to configure additional assemblies that contain bindings (step definitions, hooks or step argument transformations). See *Bindings from External Assemblies* for further details.

The assembly of the Reqnroll project (the project containing the feature files) is automatically included. The binding assemblies must be placed in the output folder (e.g. bin/Debug) of the Reqnroll project, for example by adding a reference to the assembly from the project.

The following example registers an additional binding assembly (`SharedStepDefinitions.dll`).

Listing 26: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

  "bindingAssemblies": [
    {
      "assembly": "SharedStepDefinitions"
    }
  ]
}
```

The `bindingAssemblies` section can contain multiple JSON objects (one for each assembly), with the following settings.

Setting	Value	Description
assembly	assembly name	The name of the assembly containing bindings (without `.dll`).

`formatters`

This section can be used to configure *Reqnroll Formatters*. See *Formatter Configuration* for further details.

Formatter Configuration

> **Note**
>
> Reqnroll formatters are only available in Reqnroll v3.0 or later.

There are two ways to configure *Reqnroll Formatters*.

- *Configuration File*
- *Environment Variables*

Defaults

Unless overwritten by using the Reqnroll configuration file and/or environment variable, Reqnroll will use the following defaults to configure formatters.

Setting	Value	Description
HTML Formatter outputFilePath	`reqnroll_report.html`	Default output file path for the HTML formatter (relative to project output folder)
Message Formatter outputFilePath	`reqnroll_report.ndjson`	Default output file path for the Cucumber Messages formatter (relative to project output folder)

Configuration File

The `formatters` section of the *`reqnroll.json` configuration file* can be used to configure formatters. Each section within `formatters` enables and configures a built-in or custom formatter. You can enable multiple formatters.

The following example enables both the *HTML* and the *Cucumber Message* formatter with custom output file paths.

Listing 27: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

    "bindingAssemblies": [
    ],
    "formatters": {
        "html" : { "outputFilePath" : "report\\living_doc.html" },
        "message" : { "outputFilePath" : "report\\cucumber_messages.ndjson" }
    }
}
```

If the formatter section is omitted, the report for that particular formatter is not enabled.

Built-in Formatter Configuration Examples

Enabling Formatters with Default Settings

To enable a formatter with its default settings, simply include it in the `formatters` section without any configuration:

Listing 28: reqnroll.json - Enable HTML formatter with defaults

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",
```

(continues on next page)

(continued from previous page)

```
  "formatters": {
    "html": {}
  }
}
```

Listing 29: reqnroll.json - Enable Message formatter with defaults

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",
  "formatters": {
    "message": {}
  }
}
```

Listing 30: reqnroll.json - Enable both formatters with defaults

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",
  "formatters": {
    "html": {},
    "message": {}
  }
}
```

Overriding Output File Path Components

You can override different parts of the output file path:

Override directory only (keep default filename):

Listing 31: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",
  "formatters": {
    "html": { "outputFilePath": "reports\\" },
    "message": { "outputFilePath": "reports\\" }
  }
}
```

Override filename only (use current directory):

Listing 32: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",
  "formatters": {
    "html": { "outputFilePath": "my_report.html" },
    "message": { "outputFilePath": "my_messages.ndjson" }
  }
}
```

Override both directory and filename:

Listing 33: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",
  "formatters": {
    "html": { "outputFilePath": "output\\detailed_report.html" },
    "message": { "outputFilePath": "output\\test_messages.ndjson" }
  }
}
```

Overriding path components with substitution variables:

> **Note**
>
> Introduced in Reqnroll v3.3

You can use substitution variables in the `outputFilePath` to dynamically generate file or directory names based on runtime information. This is useful for organizing reports by build, branch, or other environment-specific data.

Available Substitution Variables

- `{timestamp}`: Replaced with the current date and time in `yyyy-MM-dd_hh_mm_ss` format.
- `{buildNumber}`: Replaced with the build number, if available.
- `{revision}`: Replaced with the current revision or commit hash, if available.
- `{branch}`: Replaced with the current branch name, if available.
- `{tag}`: Replaced with the current tag, if available.
- `{env:environmentVariableName}`: Replaced with the value of the specified environment variable.

Example Usage

Listing 34: reqnroll.json

```
{
  "formatters": {
    "html": { "outputFilePath": "reports/{branch}/report_{timestamp}.html" },
    "message": { "outputFilePath": "reports/{env:TEST_ENV}/messages_{buildNumber}.ndjson
↪" }
  }
}
```

In this example:

- The HTML report will be saved in a subdirectory named after the current branch, with a timestamped filename.
- The Message formatter report will be saved in a directory named after the value of the `TEST_ENV` environment variable, and the filename will include the build number.

  Note: If a variable cannot be resolved at runtime, it will be replaced with an empty string.

Environment Variables

The settings discussed above can be overridden by setting an environment variable. When an environment variable is set, it takes precedence over the same configuration setting in the configuration file. If a setting is not overridden by an environment variable, the value will be taken from the configuration file (if set), otherwise a default (as shown above) will be used. The formatter specific environment variables override the general `REQNROLL_FORMATTERS` environment variable settings.

Available Environment Variables

REQNROLL_FORMATTERS_DISABLED

Description: Disables the entire formatter subsystem when set to `true`. When disabled, no formatters will run regardless of configuration file settings.

Default Value: `false` (formatters are enabled by default)

Behavior:

- When set to `true`: All formatters are disabled, no report files will be generated
- When set to `false` or not set: Formatters operate according to configuration file settings

Usage Examples:

```
# Disable all formatters
export REQNROLL_FORMATTERS_DISABLED=true

# Enable formatters (default behavior)
export REQNROLL_FORMATTERS_DISABLED=false
```

REQNROLL_FORMATTERS_*formatter*

Description: Overrides the configuration of a specific formatter using key-value pair settings. For example the `REQNROLL_FORMATTERS_HTML` environment variable can be used to configure the *html formatter*.

Default Value: Not set (uses configuration file settings)

Behavior:

- When set to `true` it enables the formatter with default settings
- When set to `false` it disables the formatter (if it was configured in the configuration file)
- When set to `setting1=value1;setting2=value2` it configures the formatter with the specified settings and values. For example the value `outputFilePath=result.html` sets the output file to `result.html`.

REQNROLL_FORMATTERS

Description: Overrides the `formatters` section of the `reqnroll.json` configuration file using JSON format.

Default Value: Not set (uses configuration file settings)

Behavior: When set, replaces the named `formatters` sub-section(s) from the configuration file.

> **Note**
>
> When using an environment variable to override a `formatters` section, the value of the environment variable must be properly escaped (appropriate to your shell) to remain a valid json representation of the configuration setting.

Environment Variable Configuration Examples

Enable HTML formatter with default settings:

```
export REQNROLL_FORMATTERS_HTML='true'
```

Enable HTML formatter with custom output path:

```
export REQNROLL_FORMATTERS_HTML='outputFilePath=result.html'
```

Enable HTML formatter with custom directory only:

```
export REQNROLL_FORMATTERS_HTML='outputFilePath=test-results/'
```

This setting will generate the HTML report in the specified folder with the default file name (`test-results/reqnroll_report.html`).

Enable Message formatter with default settings:

```
export REQNROLL_FORMATTERS_MESSAGE='true'
```

Enable both formatters with custom output paths using JSON:

```
export REQNROLL_FORMATTERS='{"formatters": {"html": {"outputFilePath": "reports/test_
↪report.html"}, "message": {"outputFilePath": "reports/test_messages.ndjson"}}}'
```

Set JSON value in different shells with correct escaping:

```
# Windows Command Prompt
set REQNROLL_FORMATTERS={"formatters": {"html": {"outputFilePath": "output\\report.html"}
↪, "message": {"outputFilePath": "output\\messages.ndjson"}}}

# PowerShell
$env:REQNROLL_FORMATTERS='{"formatters": {"html": {"outputFilePath": "output/report.html
↪"}, "message": {"outputFilePath": "output/messages.ndjson"}}}'

# Linux/macOS Bash
export REQNROLL_FORMATTERS='{"formatters": {"html": {"outputFilePath": "output/report.
↪html"}, "message": {"outputFilePath": "output/messages.ndjson"}}}'
```

> **Note**
>
> When using an environment variable to override a `formatters` section, the `outputFilePath` may also use substitution variables.

Configuring Build

The main use case of Reqnroll involves adding one of the Reqnroll unit test NuGet packages to the test project in question, e.g.:

- `Reqnroll.MSTest`
- `Reqnroll.NUnit`
- `Reqnroll.TUnit`
- `Reqnroll.xUnit`

All of them depend on the package `Reqnroll.Tools.MsBuild.Generation`, including which modifies the project's build process. The obvious effect is generation of the *.cs files from the respective *.feature files and inclusion of the former in the compilation.

Embedding feature files as resource

The `Reqnroll.Tools.MsBuild.Generation` package is essentially identical to its previous incarnation - `SpecFlow.Tools.MsBuild.Generation`. However, there is one original behavior which is now opt-in instead of unconditional - embedding the *.feature files as resources in the generated assembly. This embedding can be enabled with the dedicated build variable - `ReqnrollEmbedFeatureFiles`.

Suppose we have a test project using Reqnroll:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    ...
  </PropertyGroup>
  ...
</Project>
```

One way to set the `ReqnrollEmbedFeatureFiles` build variable is by adding `<ReqnrollEmbedFeatureFiles>true</ReqnrollEmbedFeatureFiles>` to the project file:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <ReqnrollEmbedFeatureFiles>true</ReqnrollEmbedFeatureFiles>
    ...
  </PropertyGroup>
  ...
</Project>
```

If many test projects are involved and you want to enable this behavior for all of them, a convenient way to do so would be creating a file `Directory.Build.props` at the parent level such that all the projects are below it:

```
<Project>
  <PropertyGroup>
```

(continues on next page)

(continued from previous page)

```
    <ReqnrollEmbedFeatureFiles>true</ReqnrollEmbedFeatureFiles>
    <UpstreamDirectoryBuildProps>$([MSBuild]::GetPathOfFileAbove('Directory.Build.props',
↪ '$(MSBuildThisFileDirectory)\..\'))</UpstreamDirectoryBuildProps>
  </PropertyGroup>
  <Import Project="$(UpstreamDirectoryBuildProps)" Condition="
↪$(UpstreamDirectoryBuildProps) != '' " />
</Project>
```

The `Import` statement is there to make sure any `Directory.Build.props` files at the higher levels (if any) are not cut off.

You can find more details about build customization at https://learn.microsoft.com/en-us/visualstudio/msbuild/customize-by-directory?view=vs-2022

You are welcome to consult Microsoft documentation (or your favorite AI) for other ways to pass build variables to the build.

1.2.4 Compatibility

Supported operating systems

SpecSync is supported on all common operating systems that support .NET, including

- Windows
- Linux
- MacOS

.NET Versions

- .NET Framework 4.6.2
- .NET Framework 4.7.2
- .NET Framework 4.8.1
- .NET 8.0
- .NET 9.0
- .NET 10.0

> **Note**
>
> Reqnroll can also be installed on any .NET frameworks that supports .NET Standard 2.0, including end-of-life (EOL) frameworks, but please note that Reqnroll does not support EOL frameworks and frameworks before .NET 8 are out of support by Microsoft.

Visual Studio

- Visual Studio 2022 (Workloads: ASP.NET and web development **or** .NET desktop development)
- Visual Studio 2026 (Workloads: ASP.NET and web development **or** .NET desktop development)

Test Execution Frameworks

- NUnit
- MsTest
- TUnit
- xUnit, xUnit v3

> **Note**
>
> MsTest v4 is supported in v3.2 or later.

Versioning Policy

The core Reqnroll framework uses semantic versioning, that means that only major version changes will introduce breaking changes. Minor version number increase represent new features or backwards compatible improvements and patch number increase represents bug fixes. This means that generally you should be able to upgrade to the latest package within the major number range without problems.

The backwards compatibility applies for the specified behavior and the API used by the end-users, but also for interfaces commonly used by plugins. This means that if a plugin for example has been created for v2.0.0 generally suppose to work with v2.3.1 as well.

> **Note**
>
> The versioning policy is slightly different for the maintained Reqnroll integration plugins (e.g. `Reqnroll.Autofac`). See the detailed policy for these in the section below.

Versioning policy of Reqnroll plugins

We maintain a couple of external integration plugins (e.g. `Reqnroll.Autofac`, or `Reqnroll.Verify`) that typically work as an adapter for some other tools or libraries. These plugins are generally versioned together with the Reqnroll core package, to make the versioning and the release process simpler. This means that we publish a new version of these plugins with each new Reqnroll version even if the plugin itself did not change.

The Reqnroll plugins also follow semantic versioning, with the exception of the version changes of the integrated product. To understand the implication of this, please follow the description below.

In some cases there is a breaking change in the tool or library the plugin integrates with. E.g. `Reqnroll.Verify` v2.0.3 supports `Verify` v23, but there is a breaking change between v23 and v24 of the `Verify` package. In order to support v24, we update the plugin and include the change in a new *minor* release of Reqnroll (as now we *added* support for v24). So the new plugin will be included in Reqnroll v2.1.0.

If you upgrade to the latest minor release, you might observe a compatibility issue unless you also upgrade the integrated package (`Verify` in this case). You can keep using the older version of the package by using the latest Reqnroll version (v2.1.0), but use the previous version (v2.0.3) of the `Reqnroll.Verify` plugin. As plugins within the same main version are cross-compatible, the v2.0.3 version of the plugin will work with Reqnroll v2.1.0.

1.3 Guides

This part contains details of the following topics.

1.3.1 How to change the test execution framework used by Reqnroll

To change the test execution framework, reference the relevant NuGet Reqnroll package. For details see:

- *xUnit*
- *NUnit*
- *MSTest*

Only one test execution framework can be referenced by a Reqnroll project at a given time.

1.3.2 Migrating from SpecFlow

Reqnroll has been created based on the open-source codebase of SpecFlow, therefore it provides a high level of compatibility with SpecFlow. We can generally say that everything that has worked with SpecFlow also works with Reqnroll, but some names and the namespaces have been modified.

The key differences between SpecFlow and Reqnroll are the following:

- All packages have been renamed from `SpecFlow.*` to `Reqnroll.*`. E.g. `Reqnroll.MsTest`.
- The namespace of the classes has been changed from `TechTalk.SpecFlow` to `Reqnroll` and some classes that had SpecFlow in their name (e.g. `ISpecFlowOutputHelper`) have been renamed accordingly. An optional *SpecFlow Compatibility Package* has been created to migrate without changing all namespaces, see below.
- There is a new `DataTable` alias for the `Table` class to better match Gherkin terminology. The `Table` class can still be used.
- The main extension methods of the *Assist helpers* have been moved to the `Reqnroll` namespace, so that they can be used without an additional namespace using statement. The helpers are now referred to as *DataTable Helpers*.
- The namespace of `IObjectContainer` has been changed from `BoDi` to `Reqnroll.BoDi`. If you have used special customizations that required to access the dependency injection container directly, you might need to update the namespace usings.
- The *Reqnroll Visual Studio extension* has been reworked in a way that it can handle both SpecFlow and Reqnroll projects (also for .NET 8.0).
- The integration plugins that have been managed by SpecFlow have been also ported to work with Reqnroll (e.g. `Reqnroll.Autofac`). See *Available Plugins*.
- The "SpecFlow.Actions" plugins that provide a ready-to-use support for different automation technologies (e.g. Selenium) are ported as `Reqnroll.SpecFlowCompatibility.Actions.*` packages (e.g. `Reqnroll.SpecFlowCompatibility.Actions.Selenium`).

This article provides you a step-by-step guidance to migrate SpecFlow projects to Reqnroll. There are two migration paths you can choose from:

1. *Migrate with the Reqnroll SpecFlow Compatibility Package*: requires minimal change in the codebase.
2. *Migrate with namespace changes*: requires simple changes, mostly doable with search-and-replace.

It is also worth mentioning that Reqnroll is based on the SpecFlow v4 codebase, so if you migrate from SpecFlow v3, you should consider the *Breaking changes since SpecFlow v3* section as well.

> **Tip**
>
> It is recommended to upgrade directly to the latest version of Reqnroll (e.g. v2.0), as we keep improving the migration experience and fix issues that might block migration.

Attention

Living Documentation Support The SpecFlow+ LivingDoc was part of the closed source implementation of SpecFlow and therefore we could not take it over for Reqnroll. We are currently in a process of re-building the tool (or something similar to that), but with a workaround you can also use Reqnroll with the SpecFlow Living Doc Generator CLI tool. See more information about the plans and the workaround at the Living Documentation discussion topic.

Attention

MsTest Scenario Outline Handling Reqnroll generates data-driven tests from scenario outlines with MsTest. With some tooling (`VSTest` pipeline task, `VSTest.Console.exe` filtering) this might cause compatibility issues. Please check the *MsTest Scenario Outline Handling* section below about the details and how to switch back to the SpecFlow compatible behavior.

Migrate with the Reqnroll SpecFlow Compatibility Package

Reqnroll contains a *SpecFlow Compatibility Package* (`Reqnroll.SpecFlowCompatibility`) that allows to use the Reqnroll classes using the SpecFlow namespace (`TechTalk.SpecFlow`). This allows a quick migration of SpecFlow project that requires minimal code changes. Later the migrated project can be incrementally transformed to use the Reqnroll namespaces.

In order to migrate a SpecFlow project using the SpecFlow compatibility package, you need to perform the following steps.

Step 1 - Change NuGet packages

You need to remove the SpecFlow NuGet package references from the project and replace them with the Reqnroll ones. This can be done with the Visual Studio NuGet package manager or by modifying the project file in an editor.

- Packages to be removed:
  - any package where the name starts with SpecFlow, e.g. `SpecFlow` or `SpecFlow.MsTest`
  - the `CucumberExpressions.SpecFlow.*` packages (Reqnroll has built-in *Cucumber Expression* support)
- Packages to add:
  - The Reqnroll package according to the test execution framework you use: `Reqnroll.NUnit`, `Reqnroll.MsTest` or `Reqnroll.xUnit`
  - The SpecFlow Compatibility package: `Reqnroll.SpecFlowCompatibility`
  - If you have used any of the `SpecFlow.Actions.*` package (e.g. `SpecFlow.Actions.Selenium`), you need to add the matching `Reqnroll.SpecFlowCompatibility.Actions.*` package (`Reqnroll.SpecFlowCompatibility.Actions.Selenium`).

After the change, your project file might look like this:

Listing 35: C# Project File (.csproj)

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>
```

(continues on next page)

(continued from previous page)

```
  <ItemGroup>
    <!-- test project dependencies (MsTest) -->
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.8.0" />
    <PackageReference Include="MSTest.TestAdapter" Version="3.2.0" />
    <PackageReference Include="MSTest.TestFramework" Version="3.2.0" />

    <!-- Reqnroll -->
    <PackageReference Include="Reqnroll.MsTest" Version="2.0.0" />
    <PackageReference Include="Reqnroll.SpecFlowCompatibility" Version="2.0.0" />
  </ItemGroup>
  [...]
</Project>
```

Tip

For most of the SpecFlow projects this is the only change you need to do and your project is ready to run with Reqnroll.

Step 2 - Review code compatibility

Build the project with the changed package references. If the project builds successfully, you can move on to the next step.

If you see build errors, they might belong to one of the following categories.

1. If the C# compiler complains of a missing `TechTalk.SpecFlow.<component>` namespace or a missing class, it means that the code has used some infrastructural elements of SpecFlow. For these files, simply add a namespace using for the related Reqnroll namespace: `using Reqnroll.<component>`. This might happen for special hook classes or step argument transformations.
2. Any other compilation error might be caused by the breaking changes between SpecFlow v3 and v4. Please check the section *Breaking changes since SpecFlow v3* below for the fixes.

After fixing these issues, your project should compile successfully.

Step 3 - Review SpecFlow App.config settings (if applicable)

Reqnroll uses a JSON configuration file named `reqnroll.json`, but it is also compatible with the `specflow.json` configuration files. So if you have used `specflow.json` or have not used custom SpecFlow configuration, you can move on to the next step.

If you have used the legacy `App.config` file to configure SpecFlow, your configuration is also handled by the SpecFlow Compatibility package, except the configuration section declaration. So you need to change only one line in your configuration file as highlighted below.

Listing 36: App.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="specFlow" type="Reqnroll.SpecFlowCompatibility.ReqnrollPlugin.
↪ConfigurationSectionHandler, Reqnroll.SpecFlowCompatibility.ReqnrollPlugin" />
```

(continues on next page)

(continued from previous page)

```
  </configSections>
  <specFlow>
    <language feature="hu-HU" />
    <stepAssemblies>
      <stepAssembly assembly="ExternalStepDefs" />
    </stepAssemblies>
  </specFlow>
</configuration>
```

Step 4 - Review execution compatibility

Now it is time to run your tests. If the tests were passing before, they should be still passing, there is no reported compatibility issue.

If you run into any problems, it might be caused by the breaking changes between SpecFlow v3 and v4. Please check the section *Breaking changes since SpecFlow v3* below for the fixes.

Congratulations you are done!

Our more complex sample application, ReqOverflow has been also migrated from SpecFlow. You can check what changes we had to do in order to get it working with Reqnroll using the SpecFlow compatibility package. See the changes on GitHub.

Migrate with namespace changes

Thanks to the high level of compatibility, it is also easy to perform a full migration from SpecFlow projects that requires simple changes, it is mostly doable with search-and-replace.

In order to migrate a SpecFlow project, you need to perform the following steps.

Step 1 - Change NuGet packages

You need to remove the SpecFlow NuGet package references from the project and replace them with the Reqnroll ones. This can be done using the Visual Studio NuGet package manager or by modifying the project file in an editor.

- Packages to be removed:
  - any package where the name starts with SpecFlow, e.g. `SpecFlow` or `SpecFlow.MsTest`
  - the `CucumberExpressions.SpecFlow.*` packages (Reqnroll has built-in *Cucumber Expression* support)
- Packages to add:
  - The Reqnroll package according to the test execution framework you use: `Reqnroll.NUnit`, `Reqnroll.MsTest` or `Reqnroll.xUnit`
  - If you have used any of the `SpecFlow.Actions.*` package (e.g. `SpecFlow.Actions.Selenium`), you need to add the matching `Reqnroll.SpecFlowCompatibility.Actions.*` package (`Reqnroll.SpecFlowCompatibility.Actions.Selenium`).

After the change, your project file might look like this:

Listing 37: C# Project File (.csproj)

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
```

(continues on next page)

(continued from previous page)

```
  </PropertyGroup>

  <ItemGroup>
    <!-- test project dependencies (MsTest) -->
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.8.0" />
    <PackageReference Include="MSTest.TestAdapter" Version="3.2.0" />
    <PackageReference Include="MSTest.TestFramework" Version="3.2.0" />

    <!-- Reqnroll -->
    <PackageReference Include="Reqnroll.MsTest" Version="2.0.0" />
  </ItemGroup>
  [...]
</Project>
```

Step 2 - Replace namespaces

Now open the project in Visual Studio or in a code editor and replace all usages of the `TechTalk.SpecFlow` namespace to `Reqnroll`. This can be done with a search-and-replace operation in your editor. Make sure you perform the replace in all files (usual shortcut: *Shift-Ctrl-H*).

- Search for: `TechTalk.SpecFlow`, enable *Match case* and *Match whole word*
- Replace with: `Reqnroll`

This will replace the namespace in namespace usings (e.g. `using TechTalk.SpecFlow;`) or fully qualified class names (e.g. `TechTalk.SpecFlow.ScenarioContext`).

Step 3 - Review code compatibility

Build the project with the changed package references. If the project builds successfully, you can move on to the next step.

If you see build errors, they might belong to one of the following categories.

1. You might have used a SpecFlow class that had SpecFlow in the name. The most common example is `ISpecFlowOutputHelper`. Replace these accordingly (e.g. `IReqnrollOutputHelper`). If you use them extensively, you can also use a "replace in all files" function.
2. Any other compilation error might be caused by the breaking changes between SpecFlow v3 and v4. Please check the section *Breaking changes since SpecFlow v3* below for the fixes.

After fixing these issues, your project should compile successfully.

Step 4 - Migrate config settings

If you have not used custom SpecFlow configuration, you can move on to the next step.

Reqnroll uses a JSON configuration file named `reqnroll.json`. The format is compatible with the `specflow.json` configuration file format, so the migration is simple, you just need to rename the file to `reqnroll.json`. It is recommended to set the JSON schema reference, so that your editor offers completion for the settings. The official schema reference is `https://schemas.reqnroll.net/reqnroll-config-latest.json`, which you can use like the example shows below.

Listing 38: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

  "language": {
    "feature": "hu-HU"
  },
  "bindingAssemblies": [
    {
      "assembly": "ExternalStepDefs"
    }
  ]
}
```

There are two settings in the `reqnroll.json` that have different name, although the names used in SpecFlow are also accepted. It is recommended though to update these as well:

- The `stepAssemblies` section has been renamed to `bindingAssemblies`. See *bindingAssemblies*.
- The `bindingCulture/name` setting has been moved to the language section as `language/binding`. See *language*.

If you have used the legacy `App.config` file to configure SpecFlow, you need to migrate the settings to `reqnroll.json` based on our *Configuration* reference.

Step 5 - Review execution compatibility

Now it is time to run your tests. If the tests were passing before, they should be still passing, there is no reported compatibility issue.

If you run into any problems that might be caused by the breaking changes between SpecFlow v3 and v4. Please check the section *Breaking changes since SpecFlow v3* below for the fixes.

Congratulations you are done!

Our more complex sample application, ReqOverflow has been also migrated from SpecFlow. You can check what changes we had to do in order to get it working with Reqnroll with a complete migration. See the changes on GitHub.

Breaking changes since SpecFlow v3

As Reqnroll is based on SpecFlow v4, if you migrate from SpecFlow v3, you might encounter problems that are caused by the breaking changes between SpecFlow v3 and v4. The following list contains the most important breaking changes and the suggestions to resolve them.

Cucumber Expressions support, compatibility of existing expressions

Reqnroll supports *Cucumber Expressions* natively for *step definitions*. This means that whenever you define a step using the `[Given]`, `[When]` or `[Then]` attribute, you can either provide a regular expression for it as a parameter or a cucumber expression.

Most of your existing regex step definitions will be compatible, because they are either properly recognized as regex or the expression works the same way with both expression types (e.g. simple text without parameters).

In case your regular expression is wrongly detected as cucumber expression, you can always force to use regular expression by specifying the regex start/end markers (`^/$`).

```
[When(@"^this expression is treated as a regex$")]
```

There are a few special cases listed below.

Invalid expressions after upgrade

In some cases you may see an error after upgrading to the Reqnroll. For example if you had a step definition with an attribute like:

```
[When(@"I \$ something")]
```

```
This Cucumber Expression has a problem ...
```

In this case the problem is that Reqnroll wrongly identified your expression as a cucumber expression.

Solution 1: Force the expression to be a regular expression by specifying the regex start/end markers (`^`/`$`):

```
[When(@"^I \$ something$")]
```

If you have many of such step definitions, you can force all of them to be treated as regex by including the start/end markers using Visual Studio "Find and Replace in Files" (Ctrl+Alt+H) option:

- Set search text to `\[(Given|When|Then)\((@?)"(.*?)"\)\]`
- Set replacement text to `[$1($2"^$3$$")]`
- Check "Use regular expressions" setting
- Click on "Replace All"

This will add the markers to all step definition attributes.

Solution 2: Change the expression to be a valid cucumber expression. For the example above, you need to remove the masking character (\), because the `$` sign does not have to be masked in cucumber expressions:

```
[When("I $ something")]
```

Solution 3: If none of the above solutions are possible, you can also consider changing the cucumber expression detection strategy as described in *How to configure Cucumber Expression behavior for large legacy projects*.

Expression matching problems during test execution

In some very special cases it can happen that the expression is wrongly identified as cucumber expression, but you only get the step binding error during test execution (usually `No matching step definition found` error), because the expression is valid as regex and as cucumber expression as well, but with different meaning.

For example if you had a step definition that matches the step `When I a/b something`, it will be considered as a cucumber expression, but in cucumber expressions, the `/` is used for alternation (so it matches either `When I a something` or `When I b something`).

```
[When(@"I a/b something")]
```

Solutions: You can apply the same solutions as above: either force it to be a regular expression by specifying the regex start/end markers or make it a valid cucumber expression.

For the latter case, you would need to mask the `/` character:

```
[When(@"I a\/b something")]
```

Cucumber Expression step definition skeletons

Reqnroll will by default generate step definition skeletons (snippets) for the new steps. So in case you write a new step as

```
When I have 42 cucumbers in my belly
```

Reqnroll will suggest the step definition to be:

```
[When("I have {int} cucumbers in my belly")]
public void WhenIHaveCucumbersInMyBelly(int p0)
...
```

If you would like to use only regular expressions in your project, you either have to fix the expression manually, or you can configure Reqnroll to generate skeletons with regular expressions. You can achieve this with the following setting in the `reqnroll.json` file:

Listing 39: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",
  "trace": {
    "stepDefinitionSkeletonStyle": "RegexAttribute"
  }
}
```

Removed calling other steps with string

The SpecFlow v3 functionality of calling a step from a step like this is not available in Reqnroll (has been removed in SpecFlow v4):

Listing 40: Step Definition Class

```
[Binding]
public class CallingStepsFromStepDefinitionSteps : Steps
{
  [Given(@"the user (.*) exists")]
  public void GivenTheUserExists(string name) { ... }

  [Given(@"I log in as (.*)")]
  public void GivenILogInAs(string name) { ... }

  [Given(@"(.*) is logged in")]
  public void GivenIsLoggedIn(string name)
  {
    Given(string.Format("the user {0} exists", name));
    Given(string.Format("I log in as {0}", name));
  }
```

This is not possible anymore, as the methods are now removed.

If you use this feature, you have two options:

- refactor to the *Driver Pattern*
- call the methods directly

MsTest Scenario Outline Handling

For scenario outline examples, SpecFlow has generated multiple methods, one for each example. In the newer versions of MsTest they have introduced support for data-driven tests using the MsTest `[DataTestMethod]` and `[DataRow]` attributes. Reqnroll has been changed to use this feature, so by default it will generate a single method for scenario outlines with multiple `[DataRow]` attributes. This allows the test runners to display the related scenario outline results in a hierarchical way.

There are some tooling that does not fully support this MsTest feature. For example you might experience problems with the VSTest task on Azure pipelines when you use distributed execution and the filtering options with VSTest.Console.exe might also have problems.

If you use such incompatible tooling, you can revert back to the original behavior by setting `generator/allowRowTests` to `false` in the `reqnroll.json` configuration file.

Listing 41: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

  "generator": {
    "allowRowTests": false
  }
}
```

Tip

If the problem is only on the pipeline, you can have a pipeline step that patches the `reqnroll.json` configuration *before the build of the project*, so that you can still enjoy the nice structured tests locally.

Complete changelog of SpecFlow v4

As SpecFlow v4 was never officially released and the original GitHub project has been deleted, here is a list of changes that were planned for SpecFlow v4 and Reqnroll has overtaken.

Breaking Changes:

- Removed the ability to call steps from steps via string
- Removed .NET Core 2.1 support
- Removed .NET Framework 4.6.1 support (min .NET Framework version: 4.6.2)
- Bindings declared as `async void` are not allowed. Use `async Task` instead.

Features:

- Add an option to colorize test result output
- Support for using Cucumber Expressions for step definitions.
- Support Rule tags (can be used for hook filters, scoping and access through `ScenarioInfo.CombinedTags`)
- Support for async step argument transformations.
- Support for ValueTask and ValueTask binding methods (step definitions, hooks, step argument transformations)

- Rules now support Background blocks
- Collect binding errors (type load, binding, step definition) and report them as exception when any of the tests are executed.

Changes:

- Existing step definition expressions detected to be either regular or cucumber expression.
- Default step definition skeletons are generating cucumber expressions.
- `ScenarioInfo.ScenarioAndFeatureTags` has been deprecated in favor of `ScenarioInfo.CombinedTags`. Now both contain rule tags as well.
- AggregateExceptions thrown by async StepDefinition methods are no longer consumed; but passed along to the test host.

1.3.3 Using the Driver Pattern

The Driver Pattern is simply an additional layer between your step definitions and your automation code.

Over the years, we noticed that a good practice to organize your bindings and automation code is to keep the code in the bindings very short (around 10 lines) and easily understandable.

This gives you following benefits:

- easier to maintain your test automation code
  As you split your code into multiple parts, it gets easier to maintain
- easy to reuse methods in different step definitions or combine multiple steps into a single step
  We often see, a group of steps that are in a lot of Scenarios. As you have now the automation code in separate classes, chaining the method calls is really easy.
- easier to read step definitions
  This makes it possible, that also non- technical people can understand what is happening in a step definition. This makes your life in bigger projects easier, because nobody will remember what every single step is doing.

The Driver pattern is heavily using *Context- Injection* to connect the multiple classes together.

Example

In this example you see how the code looks before and after refactoring with the Driver pattern.

Before:

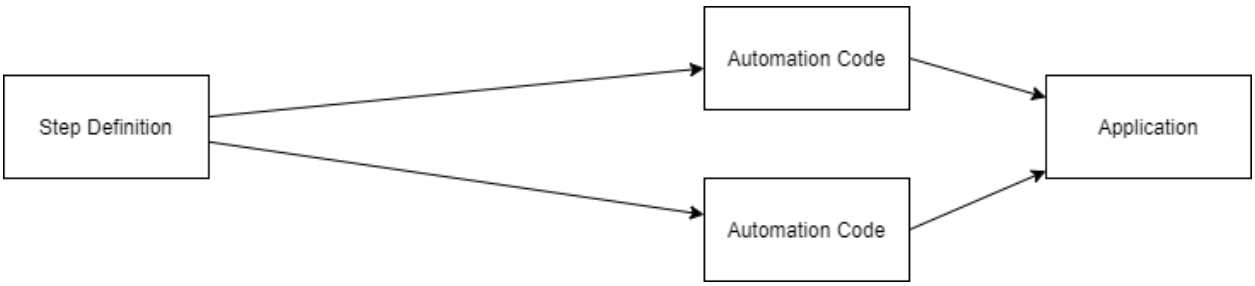
This is some automation code that uses the *Page Object Model* and checks if some WebElements are existing.

```
[Then(@"it is possible to enter a '(.*)' with label '(.*)'")]
public void ThenItIsPossibleToEnterAWithLabel(string inputType, string expectedLabel)
{
    var submissionPageObject = new SubmissionPageObject(webDriverDriver);

    switch (inputType.ToUpper())
    {
        case "URL":
            submissionPageObject.UrlWebElement.Should().NotBeNull();
            submissionPageObject.UrlLabel.Should().Be(expectedLabel);
            break;
        case "TYPE":
            submissionPageObject.TypeWebElement.Should().NotBeNull();
            submissionPageObject.TypeLabel.Should().Be(expectedLabel);
```

(continues on next page)

(continued from previous page)

```
            break;
        default:
            throw new NotImplementedException(inputType + " not implemented");
    }
}
```



After:

With moving the automation code into a driver class, we could reduce the number of lines in the step definition to one. Also we can now use a method-name (`CheckExistenceOfInputElement`), that is understandable by everybody in your team.

To get an instance of the driver class (`SubmissionSteps`), we are using the *Context- Injection* Feature of Reqnroll.

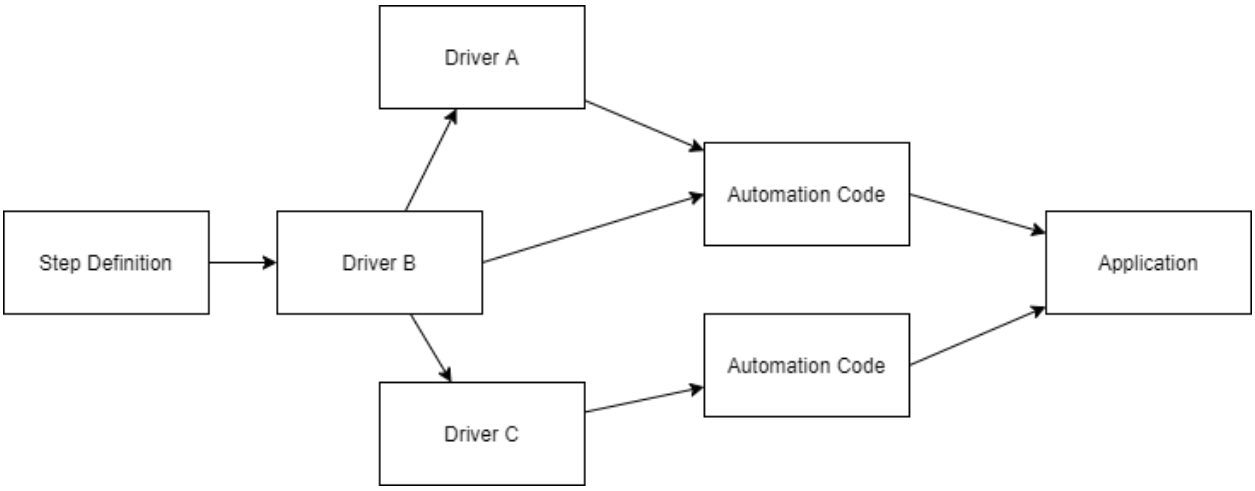
```
[Binding]
public class SubmissionSteps
{
    private readonly SubmissionPageDriver submissionPageDriver;

    public SubmissionSteps(SubmissionPageDriver submissionPageDriver)
    {
        this.submissionPageDriver = submissionPageDriver;
    }

    [Then(@"it is possible to enter a '(.*)' with label '(.*)'")]
    public void ThenItIsPossibleToEnterAWithLabel(string inputType, string expectedLabel)
    {
        submissionPageDriver.CheckExistenceOfInputElement(inputType, expectedLabel);
    }

    // ...
```

```
public class SubmissionPageDriver
{
    // ...

    public void CheckExistenceOfInputElement(string inputType, string expectedLabel)
    {
        var submissionPageObject = new SubmissionPageObject(webDriverDriver);

        switch (inputType.ToUpper())
        {
            case "URL":
                submissionPageObject.UrlWebElement.Should().NotBeNull();
```

(continues on next page)

(continued from previous page)

```
                submissionPageObject.UrlLabel.Should().Be(expectedLabel);
                break;
            case "TYPE":
                submissionPageObject.TypeWebElement.Should().NotBeNull();
                submissionPageObject.TypeLabel.Should().Be(expectedLabel);
                break;
            default:
                throw new NotImplementedException(inputType + " not implemented");
        }
    }

    // ...
```



Further Resources

- http://leitner.io/2015/11/14/driver-pattern-empowers-your-specflow-step-definitions

1.3.4 Using Page Object Model

The Page Object Model is a pattern, that is often used to abstract your Web UI with Selenium to easier automate it.

So to automate following HTML snippet

```
<input id="txtUrl" name="Url" type="text" value="">
```

you have following class to control it

```
public class PageObject
{
    public IWebElement TxtUrl {get;}
}
```

When you are working with Selenium, you are always working with WebElements to access the different elements on your Website. You can find them with the `FindElement` and `FindElements` methods on the `WebDriver` class.
If you are always using these methods directly in your automation code, you will get a lot of code duplication. This is the moment when you should start using the Page Object Model. You hide the calls to the `FindElement(s)` methods in a class.

This has following advantages:

- the classes are easier reusable
- if you need to change an `id` of your element, you need to change only one place
- your bindings are less dependent on your HTML structure

Simple Implementation

HTML:

```
<input id="txtUrl" name="Url" type="text" value="">
```

Code:

```
public class PageObject
{
    private IWebDriver _webDriver;

    public PageObject(IWebDriver webDriver)
    {
        _webDriver = webDriver;
    }

    public IWebElement txtUrl => _webDriver.FindElement(By.Id("txtUrl"));
}
```

You pass your `WebDriver` instance via constructor, and always when you access the `TxtUrl` property, the `WebDriver` searches on the whole page for an element with the id `txtUrl`. There is no caching involved.

Implementation with Caching

HTML:

```
<input id="txtUrl" name="Url" type="text" value="">
```

Code:

```
public class PageObject
{
    private IWebDriver _webDriver;
    private Lazy<IWebElement> _txtUrl;

    public PageObject(IWebDriver webDriver)
    {
        _webDriver = webDriver;
        _txtUrl = new Lazy<IWebElement>(() => _webDriver.FindElement(By.Id("txtUrl")));
    }

    public IWebElement txtUrl => _txtUrl.Value;
}
```

Again You pass your `WebDriver` instance via constructor. In this case we are using Lazy as a easy way to cache the result of the `FindElement` method.
Only the first call to the `txtUrl` property, triggers a call to the `FindElement` function. All subsequent calls, will return the same value as before. This will save you some time in execution of your automation code, as the WebDriver needs to do search less often for the same element.

If you use a caching strategy like that, be careful with your lifetime of your page objects and your page. Don't reuse an old instance of your page model, if the page changed in the meantime.

Implementation with Hierarchy

HTML:

```
<div class='A'>
    <div class='B'/>
</div>
<div class='B'>
</div>
```

Code:

```
public class ParentPageObject
{
    private IWebDriver _webDriver;

    public ParentPageObject(IWebDriver webDriver)
    {
        _webDriver = webDriver;
    }

    public IWebElement WebElement => _webDriver.FindElement(By.ClassName("A"));

    public ChildPageObject Child => new ChildPageObject(WebElement);
}

public class ChildPageObject
{
    private IWebElement _webElement;
    private Lazy<IWebElement> _txtUrl;

    public ChildPageObject(IWebElement webElement)
    {
        _webElement = webElement;
    }

    public IWebElement WebElement => _webElement.FindElement(By.ClassName("B"));
}
```

In this example we have a slightly adjusted HTML document to work with. There are two `div`- elements with the same class `B`, but we only want the PageObject for the `div`- element with the class `A` and the child.

If we would use the same `WebDriver.FindElement` method we would get the `div`- element that is on the same level as the `A` div.
But every WebElement has also the `FindElement(s)`- methods. This enable you to query the elements only in a part of your whole HTML DOM.
To do that we are passing this time the `parent`- WebElement to the `ChildPageObject` class to only search for the element with the class `B` within the `A`- div.

This concept enables you to structure your PageObjects in a similar way you have your HTML DOM structure.

Further resources

- https://www.browserstack.com/guide/page-object-model-in-selenium
- https://www.selenium.dev/documentation/en/guidelines_and_recommendations/page_object_models/
- https://martinfowler.com/bliki/PageObject.html

1.3.5 How to configure Cucumber Expression behavior for large legacy projects

Reqnroll uses Cucumber Expressions as the default method to connect step definitions to steps, while regular expressions are still supported. Cucumber expressions provide a nice and convenient way to define which step should match to a step definition, but if you migrated a large project that uses regular expressions extensively, you might run into some issues.

The main problem is that Reqnroll needs to decide based on an expression whether it is a regular expression or a cucumber expression. As SpecFlow introduced regular expression support without forcing the users to use the regex start (`^`) and end (`$`) markers, this is not an easy task to do. Reqnroll uses some heuristics to make the decision:

- If the expression has the regex start (`^`) and end (`$`) markers, it is treated as regex
- If the expression contains a cucumber expression parameter placeholder (e.g. `{string}`), it is treated as cucumber expression
- If the expression contains common regex patterns (`.*`, `\.`, `\d+`) it is treated as regex
- In all other cases it is treated as cucumber expression

These heuristics work for the most of the regular expressions used in legacy projects, but in a large project there can be a big number of mis-classification that can only be detected during test execution time.

To avoid that, our primary recommendation is to add the regex start (`^`) and end (`$`) markers for all of the existing step definitions. This can be done in Visual Studio with a solution-level search-and-replace:

- Invoke Visual Studio "Find and Replace in Files" (Ctrl+Alt+H) option
- Set search text to `\[(Given|When|Then)\((@?)"(.*?)"\)\]`
- Set replacement text to `[$1($2"^$3$$")]`
- Check "Use regular expressions" setting
- Click on "Replace All"

If for some reason this is not possible, you can also modify locally the heuristics that are used to decide whether an expression is a regular expression or a cucumber expression. Maybe there is a common regex pattern in your legacy project that you would like to handle or in an extreme case, you could override the logic (temporarily) to treat all expressions as regex.

Overriding Cucumber Expression detection strategy might be incompatible with IDE support

Changing the cucumber expression detection strategy might cause incompatibilities with different IDEs, resulting that the IDE will still mis-classify your expression and report a warning or an undefined step, although during test execution the scenarios can be executed without problems. (Visual Studio currently handles strategy overrides, but there might be features in the future that does not work correctly with overridden strategies.)

Because of that we recommend to apply this only temporarily or if the other solution (add the regex start and end markers) cannot be applied.

In order to override the detection strategy, you need to implement a simple Reqnroll runtime plugin. In the simplest case this is just adding a C# file to your Reqnroll project.

The plugin should look like the following (this is the file you need to add to your project):

Listing 42: ForceRegexPlugin.cs

```
using System.Text.RegularExpressions;
using Reqnroll.Bindings.CucumberExpressions;
using Reqnroll.Plugins;
using Reqnroll.UnitTestProvider;
using MyProject;

[assembly:RuntimePlugin(typeof(ForceRegexPlugin))]

namespace MyProject;

public class ForceRegexPlugin : IRuntimePlugin
{
    public class ForceRegexDetector : ICucumberExpressionDetector
    {
        public bool IsCucumberExpression(string cucumberExpressionCandidate)
        {
            // TODO: If cucumberExpressionCandidate contains a specific regex pattern␣
↪you use,
            // treat it as regex
            if (Regex.IsMatch(cucumberExpressionCandidate, @"some-pattern"))
                return false;
            // Otherwise fall back to the default logic
            // (you can also derive from CucumberExpressionDetector and use 'base')
            return new CucumberExpressionDetector().
↪IsCucumberExpression(cucumberExpressionCandidate);
            // In order to force all expressions to be regex, just return false.
        }
    }

    public void Initialize(
        RuntimePluginEvents runtimePluginEvents,
        RuntimePluginParameters runtimePluginParameters,
        UnitTestProviderConfiguration unitTestProviderConfiguration)
    {
        runtimePluginEvents.CustomizeGlobalDependencies += (_, args) =>
        {
            // register our class as ICucumberExpressionDetector
            args.ObjectContainer.RegisterTypeAs<ForceRegexDetector,␣
↪ICucumberExpressionDetector>();
        };
    }
}
```

You need to complete the logic marked with `TODO` and also correct the namespace `MyProject` when you apply this code.

1.4 Gherkin

The feature files used by Reqnroll are in Gherkin format. This format is specified and maintained by the Cucumber project.

In this documentation we provide a few important details about the format and how they work with Reqnroll. For a full language reference please check the Cucumber documentation.

1.4.1 Feature Files

The feature files are the files that contain the BDD executable specification.

The feature files are plain text files with the `.feature` extension. You can put feature files in any folders within the Reqnroll project, but the convention is to have a `Features` folder in your project and put the feature files in that folder, optionally in sub-folders.

Feature Files Should Be Saved in UTF-8

For proper support of non-ASCII characters in feature files (such as currency symbols and accented characters, feature files must be encoded in UTF-8 (with or without BOM signature). You may wish to add a [*.feature] section to your .editorconfig file with a charset setting to enforce this.

The format of the feature files is called *Gherkin* that is specified and maintained by the Cucumber project. For a full language reference please check the Cucumber documentation.

The following example shows a feature file that describes the addition functionality of a calculator.

Listing 43: Calculator.feature

```
Feature: Calculator

Simple calculator for adding two numbers

Rule: Add should calculate the sum of the entered numbers

@mytag
Scenario: Add two numbers
        Given the first number is 50
        And the second number is 70
        When the two numbers are added
        Then the result should be 120
```

Please also check the *Gherkin Reference* section of the Reqnroll documentation for the details of the feature file syntax.

1.4.2 Feature Language

To avoid communication errors introduced by translations, it is recommended to keep the specification and the acceptance test descriptions in the language of the business. The Gherkin format supports many natural languages besides English, like German, Spanish or French. More details on the supported natural languages are available in the Cucumber documentation.

The language of the feature files can either be specified globally in your configuration (see *Set the default feature file language*, or in the feature file's header using the `#language` syntax. Specify the language using the ISO language names used by the `CultureInfo` class of the .NET Framework (e.g. `en-US`).

Listing 44: Feature File

```
#language: de-DE
Funktionalität: Addition
...
```

Reqnroll uses the feature file language to determine the set of keywords used to parse the file, but the language setting is also used as the default setting for converting parameters by the Reqnroll runtime. The culture for binding execution and parameter conversion can be specified explicitly, see *language element*.

As data conversion can only be done using a specific culture in the .NET Framework, we recommend using the specific culture name (e.g. `en-US`) instead of the neutral culture name (e.g. `en`). If a neutral culture is used, Reqnroll uses a specific default culture to convert data (e.g. `en-US` is used to convert data if the `en` language was used).

1.4.3 Gherkin Reference

Gherkin uses a set of special *keywords* to give structure and meaning to executable specifications. Each keyword is translated to many spoken languages; in this reference we'll use English.

Most lines in a Gherkin document start with one of the *keywords*.

Comments are only permitted at the start of a new line, anywhere in the feature file. They begin with zero or more spaces, followed by a hash sign (#) and some text.

Block comments are currently not supported by Gherkin.

Either spaces or tabs may be used for indentation. The recommended indentation level is two spaces. Here is an example:

Listing 45: GuessTheWord.feature

```
Feature: Guess the word

  # The first example has two steps
  Scenario: Maker starts a game
    When the Maker starts a game
    Then the Maker waits for a Breaker to join

  # The second example has three steps
  Scenario: Breaker joins a game
    Given the Maker has started a game with the word "silky"
    When the Breaker joins the Maker's game
    Then the Breaker must guess a word with 5 characters
```

The trailing portion (after the keyword) of each step is matched to a code block, called a *step definition*.

Keywords

Each line that isn't a blank line has to start with a Gherkin *keyword*, followed by any text you like. The only exceptions are the feature and scenario descriptions.

The primary keywords are:

- `Feature`
- `Rule`
- `Example` (or `Scenario`)

- `Given`, `When`, `Then`, `And`, `But` for steps (or `*`)
- `Background`
- `Scenario Outline` (or `Scenario Template`)
- `Examples`

There are a few secondary keywords as well:

- `"""` (Doc Strings)
- `|` (Data Tables)
- `@` (Tags)
- `#` (Comments)

Localization Gherkin is localized for many *spoken languages*; each has their own localized equivalent of these keywords.

Feature

The purpose of the `Feature` keyword is to provide a high-level description of a software feature, and to group related scenarios.

The first primary keyword in a Gherkin document must always be `Feature`, followed by a `:` and a short text that describes the feature.

You can add free-form text underneath `Feature` to add more description.

These description lines are ignored by Reqnroll at runtime, but are available for reporting (They are included by default in html reports).

Listing 46: GuessTheWord.feature

```
Feature: Guess the word

  The word guess game is a turn-based game for two players.
  The Maker makes a word for the Breaker to guess. The game
  is over when the Breaker guesses the Maker's word.

  Scenario: Maker starts a game
```

The name and the optional description have no special meaning to Reqnroll. Their purpose is to provide a place for you to document important aspects of the feature, such as a brief explanation and a list of business rules (general acceptance criteria).

The free format description for `Feature` ends when you start a line with the keyword `Background`, `Rule`, `Example` or `Scenario Outline` (or their alias keywords).

You can place tags above `Feature` to group related features, independent of your file and directory structure.

Tags

Tags are markers that can be assigned to features and scenarios. Assigning a tag to a feature is equivalent to assigning the tag to all scenarios in the feature file.

If supported by the *test execution framework*, Reqnroll generates categories from the tags. The generated category name is the same as the tag's name, but without the leading `@`. You can filter and group the tests to be executed using these unit test categories. For example, you can tag crucial tests with `@important`, and then execute these tests more frequently.

If your test execution framework does not support categories, you can still use tags to implement special logic for tagged scenarios in *bindings* by querying the `ScenarioContext.ScenarioInfo.Tags` property.

Scenario, Rule and Feature level tags are available by querying the `ScenarioInfo.CombinedTags` property.

Reqnroll treats the `@ignore` tag as a special tag. Reqnroll generates an *ignored test* method from scenarios with this tag.

Descriptions

Free-form descriptions (as described above for `Feature`) can also be placed underneath `Example/Scenario`, `Background`, `Scenario Outline` and `Rule`.

You can write anything you like, as long as no line starts with a keyword.

Rule

The purpose of the `Rule` keyword is to represent one *business rule* that should be implemented. It provides additional information for a feature. A `Rule` is used to group together several scenarios that belong to this *business rule*. A `Rule` should contain one or more scenarios that illustrate the particular rule.

You can also add tags on rules that will be inherited to all scenarios within that rule (like feature tags).

For example:

Listing 47: Highlander.feature

```
Feature: Highlander

  Rule: There can be only One

    Scenario: Only One -- More than one alive
      Given there are 3 ninjas
      And there are more than one ninja alive
      When 2 ninjas meet, they will fight
      Then one ninja dies (but not me)
      And there is one ninja less alive

    Scenario: Only One -- One alive
      Given there is only 1 ninja alive
      Then he (or she) will live forever ;-)

  @edge_case
  Rule: There can be Two (in some cases)

    Scenario: Two -- Dead and Reborn as Phoenix
      ...
```

Scenario

This is a *concrete example* that *illustrates* a business rule. It consists of a list of *steps*.

The keyword `Scenario` is a synonym of the keyword `Example`.

You can have as many steps as you like, but we recommend you keep the number at 3-5 per example. Having too many steps in an example, will cause it to lose its expressive power as specification and documentation.

In addition to being a specification and documentation, an example is also a *test*. As a whole, your examples are an *executable specification* of the system.

Examples follow this same pattern:

- Describe an initial context (`Given` steps)
- Describe an event (`When` steps)
- Describe an expected outcome (`Then` steps)

Steps

Each step starts with `Given`, `When`, `Then`, `And`, or `But`.

Reqnroll executes each step in a scenario one at a time, in the sequence you've written them in. When Reqnroll tries to execute a step, it looks for a matching step definition to execute.

Keywords are not taken into account when looking for a step definition. This means you cannot have a `Given`, `When`, `Then`, `And` or `But` step with the same text as another step.

Reqnroll considers the following steps duplicates:

Listing 48: Feature File

```
Given there is money in my account
Then there is money in my account
```

This might seem like a limitation, but it forces you to come up with a less ambiguous, more clear domain language:

Listing 49: Feature File

```
Given my account has a balance of £430
Then my account should have a balance of £430
```

Given

`Given` steps are used to describe the initial context of the system - the *scene* of the scenario. It is typically something that happened in the *past*.

When Reqnroll executes a `Given` step, it will configure the system to be in a well-defined state, such as creating and configuring objects or adding data to a test database.

The purpose of `Given` steps is to **put the system in a known state** before the user (or external system) starts interacting with the system (in the `When` steps). Avoid talking about user interaction in `Given`'s. If you were creating use cases, `Given`'s would be your preconditions.

It's okay to have several `Given` steps (use `And` or `But` for number 2 and upwards to make it more readable).

Examples:

- Mickey and Minnie have started a game
- I am logged in
- Joe has a balance of £42

When

`When` steps are used to describe an event, or an *action*. This can be a person interacting with the system, or it can be an event triggered by another system.

It's strongly recommended you only have a single `When` step per Scenario. If you feel compelled to add more, it's usually a sign that you should split the scenario up into multiple scenarios.

Examples:

- Guess a word
- Invite a friend
- Withdraw money

Imagine it's 1922. Most software does something people could do manually (just not as efficiently).

Try hard to come up with examples that don't make any assumptions about technology or user interface. Imagine it's 1922, when there were no computers.

Implementation details should be hidden in the *step definitions*.

Then

`Then` steps are used to describe an *expected* outcome, or result.

The *step definition* of a `Then` step should use an *assertion* to compare the *actual* outcome (what the system actually does) to the *expected* outcome (what the step says the system is supposed to do).

An outcome *should* be on an **observable** output. That is, something that comes *out* of the system (report, user interface, message), and not a behavior deeply buried inside the system (like a record in a database).

Examples:

- See that the guessed word was wrong
- Receive an invitation
- Card should be swallowed

While it might be tempting to implement `Then` steps to look in the database - resist that temptation!

You should only verify an outcome that is observable for the user (or external system), and changes to a database are usually not.

And, But

If you have successive `Given`'s, `When`'s, or `Then`'s, you *could* write:

Listing 50: Feature File

```
Scenario: Multiple Givens
  Given one thing
  Given another thing
  Given yet another thing
  When I open my eyes
  Then I should see something
  Then I shouldn't see something else
```

Or, you could make the example more fluidly structured by replacing the successive `Given`'s, `When`'s, or `Then`'s with `And`'s and `But`'s:

Listing 51: Feature File

```
Scenario: Multiple Givens
  Given one thing
  And another thing
  And yet another thing
  When I open my eyes
  Then I should see something
  But I shouldn't see something else
```

*

Gherkin also supports using an asterisk (*) in place of any of the normal step keywords. This can be helpful when you have some steps that are effectively a *list of things*, so you can express it more like bullet points where otherwise the natural language of `And` etc might not read so elegantly.

For example:

Listing 52: Feature File

```
Scenario: All done
  Given I am out shopping
  And I have eggs
  And I have milk
  And I have butter
  When I check my list
  Then I don't need anything
```

Could be expressed as:

Listing 53: Feature File

```
Scenario: All done
  Given I am out shopping
  * I have eggs
  * I have milk
  * I have butter
  When I check my list
  Then I don't need anything
```

Background

Occasionally you'll find yourself repeating the same `Given` steps in all of the scenarios in a `Feature`.

Since it is repeated in every scenario, this is an indication that those steps are not *essential* to describe the scenarios; they are *incidental details*. You can literally move such `Given` steps to the background, by grouping them under a `Background` section.

A `Background` allows you to add some context to the scenarios that follow it. It can contain one or more `Given` steps, which are run before *each* scenario, but after any *Before hooks*.

A `Background` is placed before the first `Scenario/Example`, at the same level of indentation.

For example:

Listing 54: MultipleSiteSupport.feature

```
Feature: Multiple site support
  Only blog owners can post to a blog, except administrators,
  who can post to all blogs.

  Background:
    Given a global administrator named "Greg"
    And a blog named "Greg's anti-tax rants"
    And a customer named "Dr. Bill"
    And a blog named "Expensive Therapy" owned by "Dr. Bill"

  Scenario: Dr. Bill posts to his own blog
    Given I am logged in as Dr. Bill
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."

  Scenario: Dr. Bill tries to post to somebody else's blog, and fails
    Given I am logged in as Dr. Bill
    When I try to post to "Greg's anti-tax rants"
    Then I should see "Hey! That's not your blog!"

  Scenario: Greg posts to a client's blog
    Given I am logged in as Greg
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."
```

`Background` is also supported at the `Rule` level, for example:

Listing 55: OverdueTasks.feature

```
Feature: Overdue tasks
  Let users know when tasks are overdue, even when using other
  features of the app

  Rule: Users are notified about overdue tasks on first use of the day
    Background:
      Given I have overdue tasks

    Scenario: First use of the day
      Given I last used the app yesterday
      When I use the app
      Then I am notified about overdue tasks

    Scenario: Already used today
      Given I last used the app earlier today
      When I use the app
      Then I am not notified about overdue tasks
  ...
```

You can only have one set of `Background` steps per `Feature` or `Rule`. If you need different `Background` steps for different scenarios, consider breaking up your set of scenarios into more `Rules` or more `Features`.

For a less explicit alternative to `Background`, check out *scoped step definitions*.

Tips for using Background

- Don't use `Background` to set up **complicated states**, unless that state is actually something the client needs to know.
  - For example, if the user and site names don't matter to the client, use a higher-level step such as `Given I am logged in as a site owner`.
- Keep your `Background` section **short**.
  - The client needs to actually remember this stuff when reading the scenarios. If the `Background` is more than 4 lines long, consider moving some of the irrelevant details into higher-level steps.
- Make your `Background` section **vivid**.
  - Use colorful names, and try to tell a story. The human brain keeps track of stories much better than it keeps track of names like `"User A"`, `"User B"`, `"Site 1"`, and so on.
- Keep your scenarios **short**, and don't have too many.
  - If the `Background` section has scrolled off the screen, the reader no longer has a full overview of whats happening. Think about using higher-level steps, or splitting the `*.feature` file.

Scenario Outline

The `Scenario Outline` keyword can be used to run the same `Scenario` multiple times, with different combinations of values.

The keyword `Scenario Template` is a synonym of the keyword `Scenario Outline`.

Copying and pasting scenarios to use different values quickly becomes tedious and repetitive:

Listing 56: Feature File

```
Scenario: eat 5 out of 12
  Given there are 12 cucumbers
  When I eat 5 cucumbers
  Then I should have 7 cucumbers

Scenario: eat 5 out of 20
  Given there are 20 cucumbers
  When I eat 5 cucumbers
  Then I should have 15 cucumbers
```

We can collapse these two similar scenarios into a `Scenario Outline`.

Scenario outlines allow us to more concisely express these scenarios through the use of a template with < >-delimited parameters:

Listing 57: Feature File

```
Scenario Outline: eating
  Given there are <start> cucumbers
  When I eat <eat> cucumbers
  Then I should have <left> cucumbers

  Examples:
    | start | eat | left |
```

(continues on next page)

(continued from previous page)

```
    |    12 |   5 |    7 |
    |    20 |   5 |   15 |
```

A `Scenario Outline` must contain an `Examples` (or `Scenarios`) section. Its steps are interpreted as a template which is never directly run. Instead, the `Scenario Outline` is run *once for each row* in the `Examples` section beneath it (not counting the first header row).

The steps can use <> delimited *parameters* that reference headers in the examples table. Reqnroll will replace these parameters with values from the table *before* it tries to match the step against a step definition.

Note

Tables used in `Examples` must have **unique headers**. Using duplicate headers will result in errors.

Hint

In certain cases, when generating method names using the regular expression method, Reqnroll is unable to generate the correct parameter signatures for unit test logic methods without a little help. Placing single quotation marks (`'`) around placeholders (eg. `'<placeholder>'`)improves Reqnroll's ability to parse the scenario outline and generate more accurate regular expressions and test method signatures.

You can also use parameters in *multiline step arguments*.

Step Arguments

In some cases you might want to pass more data to a step than fits on a single line. For this purpose Gherkin has `Doc Strings` and `Data Tables`.

Doc Strings

`Doc Strings` are handy for passing a larger piece of text to a step definition.

The text should be offset by delimiters consisting of three double-quote marks on lines of their own:

Listing 58: Feature File

```
Given a blog post named "Random" with Markdown body
  """
  Some Title, Eh?
  ===============
  Here is the first paragraph of my blog post. Lorem ipsum dolor sit amet,
  consectetur adipiscing elit.
  """
```

In your step definition, there's no need to find this text and match it in your pattern. It will automatically be passed as the last argument in the step definition.

Indentation of the opening """ is unimportant, although common practice is two spaces in from the enclosing step. The indentation inside the triple quotes, however, is significant. Each line of the Doc String will be dedented according to the opening """. Indentation beyond the column of the opening "" will therefore be preserved.

Data Tables

`Data Tables` are handy for passing a list of values to a step definition:

Listing 59: Feature File

```
Given the following users exist:
  | name   | email              | twitter         |
  | Aslak  | aslak@cucumber.io  | @aslak_hellesoy |
  | Julien | julien@cucumber.io | @jbpros         |
  | Matt   | matt@cucumber.io   | @mattwynne      |
```

Just like `Doc Strings`, `Data Tables` will be passed to the step definition as the first argument.

Reqnroll provides a rich API for manipulating tables from within step definitions. See the *DataTable Helpers* reference for more details.

Spoken Languages

The language you choose for Gherkin should be the same language your users and domain experts use when they talk about the domain. Translating between two languages should be avoided.

This is why Gherkin has been translated to over 70 languages.

Here is a Gherkin scenario written in Norwegian:

Listing 60: Feature File

```
# language: no
Funksjonalitet: Gjett et ord

  Eksempel: Ordmaker starter et spill
    Når Ordmaker starter et spill
    Så må Ordmaker vente på at Gjetter blir med

  Eksempel: Gjetter blir med
    Gitt at Ordmaker har startet et spill med ordet "bløtt"
    Når Gjetter blir med på Ordmakers spill
    Så må Gjetter gjette et ord på 5 bokstaver
```

A `# language:` header on the first line of a feature file tells Reqnroll what spoken language to use - for example `# language: fr` for French. If you omit this header, Reqnroll will default to English (`en`).

You can also define the language in the *configuration file*.

Gherkin Dialects

In order to allow Gherkin to be written in a number of languages, the keywords have been translated into multiple languages. To improve readability and flow, some languages may have more than one translation for any given keyword.

Overview

You can find all translation of Gherkin in the Cucumber documentation. This is also the place to add or update translations.

> **Note**
>
> Big parts of this page where taken over from Cucumber Gherkin Reference.

1.5 Automation Features

This part of the documentation describes the Reqnroll features that can be used to implement the automation code for the scenarios.

In order to automate the scenarios, you can create *step definitions*, *hooks* and *step argument transformations*. In Reqnroll these elements are called bindings.

This part contains details of the following topics.

1.5.1 Bindings

The *Gherkin feature files* are closer to free-text than to code – they cannot be executed as they are. The automation that connects the specification to the application interface has to be developed first. The automation that connects the Gherkin specifications to source code is called a *binding*. The binding classes and methods can be defined in the Reqnroll project or in *external binding assemblies*.

> **Note**
>
> Bindings (step definitions, hooks, step argument transformations) are global for the entire Reqnroll project.

There are several kinds of bindings in Reqnroll.

Step Definitions

This is the most important one. The *step definition* that automates the scenario at the step level. This means that instead of providing automation for the entire scenario, it has to be done for each separate step. The benefit of this model is that the step definitions can be reused in other scenarios, making it possible to (partly) construct further scenarios from existing steps with less (or no) automation effort.

It is required to add the `[Binding]` attribute to the classes where you define your step definitions.

See more details about step definitions in the *Step Definitions* page.

Hooks

Hooks can be used to perform additional automation logic on specific events, e.g. before executing a scenario. See more details about hooks in the *Hooks* page.

Step Argument Transformations

Step Argument Transformations can be used to extend the step definition parameter conversion system of Reqnroll. See more details about step argument conversions in the *Step Argument Conversions* page.

1.5.2 Step Definitions

The step definitions provide the connection between your feature files and application interfaces. You have to add the `[Binding]` attribute to the class where your step definitions are:

Listing 61: Step Definition File

```
[Binding]
public class StepDefinitions
{
        ...
}
```

> **Note**
>
> Bindings (step definitions, hooks, step argument transformations) are global for the entire Reqnroll project.

For better reusability, the step definitions can include parameters. This means that it is not necessary to define a new step definition for each step that just differs slightly. For example, the steps `When I perform a simple search on 'Domain'` and `When I perform a simple search on 'Communication'` can be automated with a single step definition, with 'Domain' and 'Communication' as parameters.

The following example shows a simple step definition that matches to the step `When I perform a simple search on 'Domain'`:

Listing 62: Step Definition File

```
[When("I perform a simple search on {string}")]
public void WhenIPerformASimpleSearchOn(string searchTerm)
{
    var controller = new CatalogController();
    actionResult = controller.Search(searchTerm);
}
```

Here the method is annotated with the `[When]` attribute, and includes the expression `I perform a simple search on {string}` used to match the step's text. This expression is called a *Cucumber expression*. The term `{string}` is used to define a (string) parameter for the method. For detailed description of the expression syntax, check the *Cucumber Expressions* page.

The matching can also be specified using regular expressions. The step definition above could be also written as:

Listing 63: Step Definition File

```
[When(@"^I perform a simple search on '(.*)'$")]
public void WhenIPerformASimpleSearchOn(string searchTerm)
{
    var controller = new CatalogController();
    actionResult = controller.Search(searchTerm);
}
```

When using regular expressions, the groups (e.g. `(.*)`) define the step definition parameters.

Supported Step Definition Attributes

- `[Given(expression)]` or `[Given]` - `Reqnroll.GivenAttribute`
- `[When(expression)]` or `[When]` - `Reqnroll.WhenAttribute`
- `[Then(expression)]` or `[Then]` - `Reqnroll.ThenAttribute`

- `[StepDefinition(expression)]` or `[StepDefinition]` - `Reqnroll.StepDefinitionAttribute`, matches for given, when or then attributes

The `expression` can be either a Cucumber Expression or a Regular Expression.

You can annotate a single method with multiple attributes in order to support different phrasings in the feature file for the same automation logic.

Listing 64: Step Definition File

```
[When("I perform a simple search on {string}")]
[When("I search for {string}")]
public void WhenIPerformASimpleSearchOn(string searchTerm)
{
  ...
}
```

Other Attributes

The `[Obsolete]` attribute from the system namespace is also supported, the *runtime section* of the configuration can be used to influence how Reqnroll behaves when an obsolete step definition is used.

Listing 65: Step Definition File

```
[Given("Stuff is done")]
[Obsolete]
public void GivenStuffIsDone()
{
    var x = 2+3;
}
```

Step Definition Methods Rules

- Must be in a public class, marked with the `[Binding]` attribute.
- Must be a public method.
- Can be either a static or an instance method. If it is an instance method, the containing class will be instantiated once for every scenario.
- Cannot have `out` or `ref` parameters.
- Cannot have optional parameters.
- Should return `void` or `Task`. Note `async` methods must return `Task`

Step Definition Styles

There are multiple options for step definition matching:

- Use attributes with *cucumber expressions*; *async* or normal
- Use attributes with regular expressions; *async* or normal

Parameter Matching Rules

- Step definitions can specify parameters. These will match to the parameters of the step definition method.
- The method parameter type can be `string` or other .NET type. In the later case a *configurable conversion* is applied.
- With cucumber expressions
  - The parameter placeholders (`{parameter-type}`) define the arguments for the method based on the order (the match result of the first group becomes the first argument, etc.).
  - For the exact parameter rules please check the *cucumber expressions* page.
- With regular expressions
  - The match groups (`(...)`) of the regular expression define the arguments for the method based on the order (the match result of the first group becomes the first argument, etc.).
  - You can use non-capturing groups `(?:regex)` in order to use groups without a method argument.
- With method name matching
  - You can refer to the method parameters with either the parameter name (ALL-CAPS) or the parameter index (zero-based) with the `P` prefix, e.g. `P0`.

Data Table or DocString Arguments

If the step definition method should match for steps having *Data Table or DocString text arguments*, additional `DataTable` or `string` parameters have to be defined in the method signature to be able to receive these arguments. You cannot have both of these arguments in a step definition.

Listing 66: Feature File

```
Given the following books
  |Author        |Title                          |
  |Martin Fowler |Analysis Patterns              |
  |Gojko Adzic   |Bridging the Communication Gap |
Given a blog post named "Random" with Markdown body
  """
  Some Title, Eh?
  ===============
  Here is the first paragraph of my blog post. Lorem ipsum dolor sit amet,
  consectetur adipiscing elit.
  """
```

Listing 67: Step Definition File

```
[Given("the following books")]
public void GivenTheFollowingBooks(DataTable table)
{
  ...
}

[Given("a blog post named {string} with Markdown body")]
public void GivenABlogPostWithMarkdownBody(string postName, string bodyText)
{
  ...
}
```

> **Note**
>
> For backwards compatibility with SpecFlow, you can also declare data table parameters with the `Reqnroll.Table` class. It is recommended to use the `DataTable` class whenever it is possible.

1.5.3 Hooks

Hooks (event bindings) can be used to perform additional automation logic at specific times, such as any setup required prior to executing a scenario. In order to use hooks, you need to add the `Binding` attribute to your class. Hooks can be synchronous or asynchronous, allowing them to perform operations that can benefit from async programming patterns:

Listing 68: Hook File

```
[Binding]
public class MyHooks
{
    [BeforeScenario]
    public void SetupTestUsers()
    {
        //...
    }
}
```

Listing 69: Hook File with async method

```
[Binding]
public class MyHooks
{
    [BeforeScenario]
    public async Task SetupTestUsersAsync()
    {
        // Asynchronous setup logic
        // Example async operation
        await Task.Delay(1000);
    }
}
```

Hooks are global, but can be restricted to run only for features or scenarios by defining a *scoped binding*, which can be filtered with *tags*. The execution order of hooks for the same type is undefined, unless *specified explicitly*.

The `[BeforeScenario]` in the following example will execute only for those scenarios that are (implicitly or explicitly) tagged with `@requiresUsers`.

Listing 70: Hook File

```
[Binding]
public class MyHooks
{
    [BeforeScenario("@requiresUsers")]
    public void SetupTestUsers()
    {
        //...
    }
}
```

Listing 71: Hook File with async method

```
[Binding]
public class MyHooks
{
    [BeforeScenario("@requiresUsers")]
    public async Task SetupTestUsersAsync()
    {
        //...
        // Asynchronous setup logic
        // Example async operation
        await Task.Delay(1000);
    }
}
```

Supported Hook Attributes

Attribute	Tag filtering*	Description
`[BeforeTestRun]` `[AfterTestRun]`	not possible	Automation logic that has to run before/after the entire test run (see note below). The method it is applied to must be static.
`[BeforeFeature]` `[AfterFeature]`	possible	Automation logic that has to run before/after executing each feature The method it is applied to must be static.
`[BeforeScenario]` or `[Before]` `[AfterScenario]` or `[After]`	possible	Automation logic that has to run before/after executing each scenario or scenario outline example
`[BeforeScenarioBlock]` `[AfterS`	possible	Automation logic that has to run before/after executing each scenario block (e.g. between the "givens" and the "whens")
`[BeforeStep]` `[AfterStep]`	possible	Automation logic that has to run before/after executing each scenario step

Note

As most of the unit test runners do not provide a hook for executing logic once the tests have been executed, the `[AfterTestRun]` event is triggered by the test assembly unload event.

The exact timing and thread of this execution may therefore differ for each test runner.

You can annotate a single method with multiple attributes, and both synchronous and asynchronous methods can be used as hooks, depending on the needs of your test setup and teardown logic.

Using Hooks with Constructor Injection

You can use *context injection* to access scenario level dependencies in your hook class using constructor injection. For example you can get the `ScenarioContext` injected in the constructor:

Listing 72: Hook File

```
[Binding]
public class MyHooks
{
    private ScenarioContext _scenarioContext;

    public MyHooks(ScenarioContext scenarioContext)
    {
        _scenarioContext = scenarioContext;
    }

    [BeforeScenario]
    public void SetupTestUsers()
    {
        //_scenarioContext...
    }
}
```

Listing 73: Hook File with async method

```
[Binding]
public class MyHooks
{
    private ScenarioContext _scenarioContext;

    public MyHooks(ScenarioContext scenarioContext)
    {
        _scenarioContext = scenarioContext;
    }

    [BeforeScenario]
    public async Task SetupTestUsersAsync()
    {
        //_scenarioContext...
        // Example async operation
        await Task.Delay(1000);
    }
}
```

Note

For static hook methods you can use parameter injection which can be combined with asynchronous execution to resolve dependencies and perform setup or teardown tasks asynchronously.

Using Hooks with Parameter Injection

You can add parameters to your hook method that will be automatically injected by Reqnroll. For example you can get the `ScenarioContext` injected as parameter in the BeforeScenario hook.

Listing 74: Hook File

```
[Binding]
public class MyHooks
{
    [BeforeScenario]
    public void SetupTestUsers(ScenarioContext scenarioContext)
    {
        //scenarioContext...
    }
}
```

Listing 75: Hook File with async method

```
[Binding]
public class MyHooks
{
    [BeforeScenario]
    public async Task SetupTestUsersAsync(ScenarioContext scenarioContext)
    {
        //scenarioContext...
        // Example async operation
        await Task.Delay(1000);
    }
}
```

Parameter injection is especially useful for hooks that must be implemented as static methods.

Listing 76: Hook File

```
[Binding]
public class Hooks
{
    [BeforeFeature]
    public static void SetupStuffForFeatures(FeatureContext featureContext)
    {
        Console.WriteLine("Starting " + featureContext.FeatureInfo.Title);
    }
}
```

Listing 77: Hook File with async method

```
[Binding]
public class Hooks
{
    [BeforeFeature]
    public static async Task SetupStuffForFeaturesAsync(FeatureContext featureContext)
    {
        // Example async operation
        await Task.Delay(1000);
        Console.WriteLine("Starting " + featureContext.FeatureInfo.Title);
    }
}
```

In the BeforeTestRun hook you can resolve test thread specific or global services/dependencies as parameters.

Listing 78: Hook File

```
[BeforeTestRun]
public static void BeforeTestRunInjection(ITestRunnerManager testRunnerManager)
{
    //All parameters are resolved from the test run (global) container automatically.
    var location = testRunnerManager.TestAssembly.Location;
}
```

Listing 79: Hook File with async method

```
[BeforeTestRun]
public static async Task BeforeTestRunInjectionAsync(ITestRunnerManager␣
↪testRunnerManager)
{
    var location = testRunnerManager.TestAssembly.Location;

    // Example async operation
    await Task.Delay(1000);
}
```

Depending on the type of the hook the parameters are resolved from a container with the corresponding lifecycle.

Attribute	Container
`[BeforeTestRun]``[AfterTestRun]`	TestRunContainer ("global" container)
`[BeforeFeature]``[AfterFeature]`	FeatureContainer
`[BeforeScenario]``[AfterScenario]``[BeforeScenarioBlock]``[AfterScenario`	ScenarioContainer `erStep]`

Hook Execution Order

By default the hooks of the same type (e.g. two `[BeforeScenario]` hook) are executed in an unpredictable order. If you need to ensure a specific execution order, you can specify the `Order` property in the hook's attributes.

Listing 80: Hook File

```
[BeforeScenario(Order = 0)]
public void CleanDatabase()
{
    // we need to run this first...
}

[BeforeScenario(Order = 100)]
public void LoginUser()
{
    // ...so we can log in to a clean database
}
```

Listing 81: Hook File with async method

```
[BeforeScenario(Order = 0)]
public async Task CleanDatabaseAsync()
{
    // we need to run this first...
    // Example async operation
    await Task.Delay(1000);
}

[BeforeScenario(Order = 100)]
public async Task LoginUserAsync()
{
    // ...so we can log in to a clean database
    // Example async operation
    await Task.Delay(1000);
}
```

The number indicates the order, not the priority, i.e. the hook with the lowest number is always executed first.

If no order is specified, the default value is 10000. However, we do not recommend on relying on the value to order your tests and recommend specifying the order explicitly for each hook.

Note

If a hook throws an unhandled exception, subsequent hooks of the same type are not executed. If you want to ensure that all hooks of the same types are executed, you need to handle your exceptions manually.

> **Note**
>
> If a `BeforeScenario` throws an unhandled exception then all the scenario steps will be marked as skipped and the `ScenarioContext.ScenarioExecutionStatus` will be set to `TestError`.

Tag Scoping

Most hooks support tag scoping. Use tag scoping to restrict hooks to only those features or scenarios that have *at least one* of the tags in the tag filter (tags are combined with OR). You can specify the tag in the attribute or using *scoped bindings*.

Throwing Exceptions from Hooks

The best practice is to avoid throwing exceptions in hooks, especially in the "after" hooks.

Rule 1: When you throw an exception in a hook, the remaining hooks of the same type (e.g. the other `[BeforeScenario]` hooks) will not be executed.

Rule 2: The "after" hooks are executed even if one of the related "before" hooks failed, therefore in the "after" hook you cannot assume that the resource you have initialized in a "before" hook has been executed.

To illustrate the exception handling behavior, consider the following example:

- `[BeforeScenario]` hook 1: Initializes service A
- `[BeforeScenario]` hook 2: Initializes service B
- `[AfterScenario]` hook 3: Cleans up service A
- `[AfterScenario]` hook 4: Cleans up service B

When "hook 1" fails, "hook 2" will not be executed (Rule 1), so "Service B" will not be initialized. However, the `[AfterScenario]` hooks ("hook 3" and "hook 4") will run, so "hook 4" will try to clean up "Service B", which was never initialized. To address this problem, it is recommended to check the status of the service in the "after" hook before performing any cleanup operations.

Note: The before/after feature hooks are handled by Reqnroll in a special way (see notes at *parallel execution*). As a result of that, the `[AfterFeature]` hooks might run "delayed" at the time of the next scenario execution in a different feature. That means that the related `[AfterFeature]` hook error will be reported at that scenario and will cause a failure of that scenario. Because of this, it is better to handle all exceptions within a `[AfterFeature]` hook.

1.5.4 Step Argument Conversions

Step definitions can use parameters to make them reusable for similar steps. The parameters are taken from either the step's text or from the values in additional examples. These arguments are provided as either strings or `Reqnroll.DataTable` instances.

To avoid cumbersome conversions in the step binding methods, Reqnroll can perform an automatic conversion from the arguments to the parameter type in the binding method. All conversions are performed using the culture of the feature file, unless the *binding setting of the language section* is defined in your `reqnroll.json` configuration file (see *Feature Language*). The following conversions can be performed by Reqnroll (in the following precedence):

- no conversion, if the argument is an instance of the parameter type (e.g. the parameter type is `object` or `string`)
- step argument transformation
- standard conversion

Step Argument Transformation

> **Note**
>
> Step argument transformations don't support *Cucumber Expressions*; use Regular Expressions (regex)

Step argument transformations can be used to apply a custom conversion step to the arguments in step definitions. The step argument transformation is a method that converts from text (specified by a regular expression) or a `DataTable` instance to an arbitrary .NET type.

A step argument transformation is used to convert an argument if:

- The return type of the transformation is the same as the parameter type
- The regular expression (if specified) matches the original (string) argument

> **Note**
>
> If multiple matching transformations are available, a warning is output in the trace and the first transformation is used.

The following example transforms a relative period of time (`in 3 days`) into a `DateTime` structure.

Listing 82: C# File

```
[Binding]
public class Transforms
{
   [StepArgumentTransformation(@"in (\d+) days?")]
   public DateTime InXDaysTransform(int days)
   {
      return DateTime.Today.AddDays(days);
   }
}
```

The following example transforms any string input (no regex provided) into an `XmlDocument`.

Listing 83: C# File

```
[Binding]
public class Transforms
{
    [StepArgumentTransformation]
    public XmlDocument XmlTransform(string xml)
    {
       XmlDocument result = new XmlDocument();
       result.LoadXml(xml);
       return result;
    }
}
```

The following example transforms a table argument into a list of `Book` entities (using the *Reqnroll Assist Helpers*).

Listing 84: C# File

```
[Binding]
public class Transforms
{
    [StepArgumentTransformation]
    public IEnumerable<Book> BooksTransform(DataTable booksTable)
    {
       return booksTable.CreateSet<Books>();
    }
}
```

By default, selection among matching step argument transformations is undeterministic. To specify selection order, use the `Order` property in the `StepArgumentTransformation` attribute, where the transformation with lower numbers takes precedence. If no order is specified, the default value is 10000.

The following example transforms a string argument to a Rating model. If regex matches the expression, the given rating score will be parsed. Otherwise, the default rating will be used.

Listing 85: C# File

```
[Binding]
public class Transforms
{
    [StepArgumentTransformation(@"with (\d+) score", Order = 1)]
    public Rating RatingTransformation(int score)
    {
      return new Rating(score);
    }

    [StepArgumentTransformation]
    public Rating GlobalRatingTransformation(string input)
    {
        return Rating.DefaultRating;
    }
}

public record Rating(int Value)
{
    public static Rating DefaultRating => new Rating(50);
}
```

Specifying the `Name` property in the `StepArgumentTransformation` attribute allows for *Parameters* to be specified with that name. This `Name` property does not enforce strict scoping to Cucumber expression parameters with that name, but is instead intended for clarity.

Standard Conversion

A standard conversion is performed by Reqnroll in the following cases:

- The argument can be converted to the parameter type using `Convert.ChangeType()`
- The parameter type is an `enum` type and the (string) argument is an enum value
- The parameter type is `Guid` and the argument contains a full GUID string or a GUID string prefix. In the latter case, the value is filled with trailing zeroes.

1.5.5 Asynchronous Bindings

If you have code that executes an asynchronous task, you can define asynchronous bindings to execute the corresponding code using the `async` and `await` keywords.

The following example shows a step definition with an asynchronous When step:

Listing 86: Step Definition File

```
[When(@"I want to get the web page '(.*)'")]
public async Task WhenIWantToGetTheWebPage(string url)
{
    var message = await _httpClient.GetAsync(url);
    // ...
}
```

Hint

You can also use asynchronous *step argument transformations*.

Hint

It is also possible to use `ValueTask` and `ValueTask<T>` return types.

1.5.6 Bindings from External Assemblies

Bindings can be defined in the main Reqnroll project or in other assemblies (*external binding assemblies*). If the bindings are used from external binding assemblies, the following notes have to be considered:

- The external binding assembly can be another project in the solution or a compiled library (dll).
- The external binding assembly can also use a different .NET language, e.g. you can write bindings for your C# Reqnroll project also in F# (As an extreme case, you can use your Reqnroll project with the feature files only and with all the bindings defined in external binding assemblies).
- The external binding assembly has to be referenced from the Reqnroll project to ensure it is copied to the target folder and listed in the `reqnroll.json` of the Reqnroll project (see below).
- The external binding assemblies can contain all kind of bindings: *step definition*, *hooks* and also *step argument transformations*.

Configuration

In order to use bindings from an external binding assembly, you have to list it (with the assembly name) in the `reqnroll.json` (the Reqnroll project is always included implicitly). See *Use bindings from external projects* section of the documentation for details.

The following example registers the project `SharedStepDefinitions` as an external binding assembly.

Listing 87: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

  "bindingAssemblies": [
```

(continues on next page)

(continued from previous page)

```
    {
      "assembly": "SharedStepDefinitions"
    }
  ]
}
```

1.5.7 Cucumber Expressions

Cucumber Expression is an expression type to specify *step definitions*. Cucumber Expressions is an alternative to Regular Expressions with a more intuitive syntax.

You can find a detailed description about cucumber expressions on GitHub. In this page we only provide a short summary and the special handling in .NET / Reqnroll.

The following step definition that uses cucumber expression matches to the step `When I have 42 cucumbers in my belly`

Listing 88: Step Definition File

```
[When("I have {int} cucumbers in my belly")]
public void WhenIHaveCucumbersInMyBelly(int count) { ... }
```

Cucumber Expression basics

Simple text

To match for a simple text, just use the text as cucumber expression.

`[When("I do something")]` matches to `When I do something`

Parameters

Parameters can be defined using the `{parameter-type}` syntax. The type name is case-sensitive. The `parameter-type` can be any of the following:

- The types: `{int}`, `{long}`, `{byte}`, `{float}`, `{double}`, `{decimal}`, `{DateTime}`, `{Guid}`
- Quoted strings: `{string}`. The text should have single or double quotes. E.g., `[Given("a user {string}")]` matches to `Given a user "Marvin"` or `Given a user 'Zaphod Beeblebrox'`.
- A single word without quotes, `{word}`. E.g., `[Given("a user {word}")]` matches to `Given a user Marvin`.
- Any other type, `{}` like `(.*)` when using regex
- An enum type, with or without a namespace. E.g. `[When("I have {CustomColor} cucumbers in my belly")]` matches to `When I have green cucumbers in my belly` if `CustomColor` is an enum with `Green` as a value.
- *Step Argument Transformations*:
  - with a name property, E.g., With `[StepArgumentTransformation("v(.*)", Name = "my_version")]`, you can define a step as `[When("I download the release {my_version} of the application")]` that matches to `When I download the release v1.2.3 of the application`.

  - without a name property, use the typename of the return type without namespace. E.g. If the method `[StepArgumentTransformation("EUR (\d+)"]` has a return type of `MyCurrency` you could use `{MyCurrency}` in the expression.

No built-in support

These have no built-in support for:

- `{TimeSpan}`
- `{TimeOnly}`
- `{DateOnly}`

For `TimeSpan`, `TimeOnly` and `DateOnly` types, `{}` could be used.

Optional parameters and alternatives

Parameters that are optional should be wrapped with parentheses (`(...)`). The `/` character could be used to provide alternatives.

For example, this step definition:

Listing 89: Step Definition File

```
[When("I have {int} cucumber(s) in my belly/tummy")]
public void WhenIHaveCucumbersInMyBelly(int count)
```

will match to all of the following steps

Listing 90: Feature File

```
When I have 42 cucumbers in my belly
When I have 1 cucumber in my belly
When I have 8 cucumbers in my tummy
```

Using Cucumber Expressions with Reqnroll

You can use both cucumber expressions and regular expressions in your project. Reqnroll uses some heuristics to decide if your expression is a cucumber expression or a regular expression.

Expression Type Detection

Reqnroll automatically determines whether your step definition uses a Cucumber Expression or Regular Expression based on the expression string. The detection logic follows these rules:

1. **Force Regular Expression**: If the expression starts with `^` or ends with `$`, it's treated as a regular expression
2. **Force Cucumber Expression**: If the expression contains parameter placeholders like `{int}`, `{string}`, etc., it's treated as a cucumber expression
3. **Regular Expression patterns**: If the expression contains common regex patterns like `(.*)`, `(a+)`, `\d+`, or `\.`, it's treated as a regular expression
4. **Default**: Otherwise, it's treated as a cucumber expression

Explicit Expression Type Control

You can explicitly control the expression type by using the `ExpressionType` parameter in your step definition attributes:

Listing 91: Step Definition File - Explicit Cucumber Expression

```
[When("I have (one/two) cucumbers", ExpressionType = ExpressionType.CucumberExpression)]
public void WhenIHaveCucumbers() { ... }

[When("I have (one|two) cucumbers", ExpressionType = ExpressionType.RegularExpression)]
public void WhenIHaveCucumbersRegex(string countText) { ... }
```

The `ExpressionType` enum has three values:

- `ExpressionType.Unspecified` - Let Reqnroll auto-detect the expression type (default)
- `ExpressionType.CucumberExpression` - Force the expression to be treated as a Cucumber Expression
- `ExpressionType.RegularExpression` - Force the expression to be treated as a Regular Expression

1.5.8 Scoped Bindings

Bindings (step definitions, hooks, step argument transformations) are global for the entire Reqnroll project. This means that step definitions bound to a very generic step text (e.g. "When I save the changes") become challenging to implement. The general solution for this problem is to phrase the scenario steps in a way that the context is clear (e.g. "When I save the **book details**").

In some cases however, it is necessary to restrict when step definitions or hooks are executed based on certain conditions. Reqnroll's scoped bindings can be used for this purpose.

You can restrict the execution of scoped bindings by:

- tag
- feature (using the feature title)
- scenario (using the scenario title)

The following tags are taken into account for scenario, scenario block or step hooks:

- tags defined for the feature
- tags defined for the scenario
- tags defined for the scenario outline
- tags defined for the scenario outline example set (`Examples:`)

Danger

Be careful! Coupling your step definitions to features and scenarios is an anti-pattern. Read more about it on the Cucumber Wiki

Use the `[Scope]` attribute to define the scope:

Listing 92: Step Definition File

```
[Scope(Tag = "mytag", Feature = "feature title", Scenario = "scenario title")]
```

Scoping Rules

Scope can be defined at the method or class level.

If multiple criteria (e.g. both tag and feature) are specified in the same `[Scope]` attribute, they are combined with AND, i.e. all criteria need to match.

The following example combines the feature name and the tag scope with "AND":

Listing 93: Step Definition File

```
[Scope(Tag = "thisTag", Feature = "myFeature")]
```

If multiple `[Scope]` attributes are defined for the same method or class, the attributes are combined with OR, i.e. at least one of the `[Scope]` attributes needs to match.

The following example combines the tag scopes with "OR":

Listing 94: Step Definition File

```
[Scope(Tag = "thisTag")] [Scope(Tag = "OrThisTag")]
[Scope(Tag = "thisTag"), Scope(Tag = "OrThisTag")]
```

Note

Scopes on a different level (class and method) will be combined with OR: defining a `[Scope]` attribute on class level and defining another `[Scope]` at method level will cause the attributes to be combined with OR. If you want an AND combination, use a single Scope, e.g.:

Listing 95: Step Definition File

```
[Scope(Feature = "feature title", Scenario = "scenario title")]
```

If a step can be matched to both a step definition without a `[Scope]` attribute as well as a step definition with a `[Scope]` attribute, the step definition with the `[Scope]` attribute is used (no ambiguity).

If a step matches several scoped step definitions, the one with the most restrictions is used. For example, if the first step definition contains `[Scope(Tag = "myTag")]` and the second contains `[Scope(Tag = "myTag", Feature = "myFeature")]` the second step definition (the more specific one) is used if it matches the step.

If you have multiple scoped step definition with the same number of restrictions that match the step, you will get an ambiguous step binding error. For example, if you have a step definition containing `[Scope(Tag = "myTag1", Scenario = "myScenario")]` and another containing `[Scope(Tag = "myTag2", Scenario = "myScenario")]`, you will receive an ambiguous step binding error if the myScenario has **both** the "myTag1" and "myTag2" tags.

Scope Examples

Scoped BeforeScenario Hook

The following example starts Selenium for scenarios marked with the `@web` tag.

Listing 96: Hook File

```
[BeforeScenario("web")]
public static void BeforeWebScenario()
{
```

(continues on next page)

(continued from previous page)

```
    StartSelenium();
}
```

Different Steps for Different Tags

The following example defines a different scope for the same step depending on whether UI automation (`@web` tag) or controller automation (`@controller` tag) is required:

Listing 97: Step Definition File

```
[When(@"I perform a simple search on '(.*)'", Scope(Tag = "controller"))]
public void WhenIPerformASimpleSearchOn(string searchTerm)
{
    var controller = new CatalogController();
    actionResult = controller.Search(searchTerm);
}

[When(@"I perform a simple search on '(.*)'"), Scope(Tag = "web")]
public void PerformSimpleSearch(string title)
{
    selenium.GoToThePage("Home");
    selenium.Type("searchTerm", title);
    selenium.Click("searchButton");
}
```

Scoping Tips & Tricks

The following example shows a way to "ignore" executing the scenarios marked with `@manual`. However Reqnroll's tracing will still display the steps, so you can work through the manual scenarios by following the steps in the report.

Listing 98: Step Definition File

```
[Binding, Scope(Tag = "manual")]
public class ManualSteps
{
    [Given(".*"), When(".*"), Then(".*")]
    public void EmptyStep()
    {
    }

    [Given(".*"), When(".*"), Then(".*")]
    public void EmptyStep(string multiLineStringParam)
    {
    }

    [Given(".*"), When(".*"), Then(".*")]
    public void EmptyStep(DataTable tableParam)
    {
    }
}
```

Beyond Scope

You can define more complex filters using the *ScenarioContext* class. The following example starts selenium if the scenario is tagged with `@web` *and* `@automated`.

Listing 99: Step Definition File

```
[Binding]
public class Binding
{
    ScenarioContext _scenarioContext;

    public Binding(ScenarioContext scenarioContext)
    {
        _scenarioContext = scenarioContext;
    }

    [BeforeScenario("web")]
    public static void BeforeWebScenario()
    {
        if(_scenarioContext.ScenarioInfo.Tags.Contains("automated"))
            StartSelenium();
    }
}
```

1.5.9 DataTable Helpers

A number of helpers implemented as extension methods of the `DataTable` class make it easier to implement steps that accept a `DataTable` parameter.

When helper methods expect a generic type (usually denoted as <T> in the method signature), you can use:

- classes
- records (with C# 9)
- tuples

`CreateInstance<T>`

The `CreateInstance<T>` extension method of the `DataTable` class will convert a table in your scenario to a single instance of a class. The class used to convert the table is specified by the generic type `T` in the method signature `CreateInstance<T>`. You can use two different table layouts in your scenarios with the `CreateInstance<T>` method.

- **Vertical Tables**

  A vertical table consists of two columns where values in the first column match property names, and values of the second column are the values assigned to those properties. The header row of the table is ignored. Header cells may be named to suit your use case.

```
Given I entered the following data into the new account form:
    | Field                | Value             |
    | Name                 | John Galt         |
    | Birthdate            | 2/2/1902          |
    | Height In Inches     | 72                |
    | Bank Account Balance | 1234.56           |
```

(continues on next page)

(continued from previous page)

```
#     ^^^^^^^^^^^^^^^^^^^^    ^^^^^^^^^^^^^^^^^
#        property names        property values
```

This layout is desirable for tables containing many values making the vertical layout easier to read.

- **Horizontal Tables**

  A horizontal table consists of a header row where the header cells match property names, and subsequent data rows contain the values assigned to those properties. In order to convert a horizontal table to a single instance of a class, the table must only contain the header row and one data row.

```
Given I entered the following data into the new account form:
    | Name      | Birthdate | Height In Inches | Bank Account Balance | # Header␣
↪row (property names)
    | John Galt | 2/2/1902  | 72               | 1234.56              | # Data row␣
↪(property values)
```

  This layout is desirable when the table does not require too many values. This helps save vertical space by consuming more horizontal space in your feature file.

  Important: use the `CreateSet<T>` method described below to create a collection of objects if more than one data row is needed.

Deciding to use a vertical or horizontal table layout is subjective. Choose the layout that is easiest to read given the information in the table.

Reqnroll matches table values to property names regardless of letter case. To Reqnroll, "BankAccount", "Bank Account", "BANK ACCOUNT" and "bank account" will all map to a property named `BankAccount`. More information on column naming is below.

Using `CreateInstance` with a Class

You can map a table to a custom class you write. The following example will map the tables described above in the vertical or horizontal layouts. First, create the class:

Listing 100: C# File

```
// Class used to map table
class Account
{
    public string Name { get; set; }
    public int HeightInInches { get; set; }
    public decimal BankAccountBalance { get; set; }
}
```

Remember that property names should match values in the table in your scenario. Reqnroll requires properties to have both a public getter and a public setter. Most built-in .NET types are converted automatically. This includes the following types:

- `int` and `int?`
- `decimal` and `decimal?`
- `bool` and `bool?`
- `DateTime` and `DateTime?`

Plus many more.

The name of the class is put in place of the generic type `T` in the call to `CreateInstance<T>`. An example step definition is below.

Listing 101: Step Definition File

```
[Given(@"Given I entered the following data into the new account form:")]
public void GivenIEnteredTheFollowingDataIntoTheNewAccountForm(DataTable table)
{
    var account = table.CreateInstance<Account>();
    //                                 ^^^^^^^

    // account.Name is "John Galt"
    // account.HeightInInches is 72
    // account.BankAccountBalance is 1234.56
}
```

The `CreateInstance<T>` method will create the Account object and set properties according to what can be read from the table. Table cell values are strings by default. These strings are converted to the type specified for each property of the destination class. For example, the string `"1234.56"` in the table is converted to a decimal value before being assigned to the `BankAccountBalance` property.

Using `CreateInstance` with ValueTuple

Alternatively you can use ValueTuples and destructuring:

Listing 102: Step Definition File

```
[Given(@"Given I entered the following data into the new account form:")]
public void GivenIEnteredTheFollowingDataIntoTheNewAccountForm(DataTable table)
{
    var account = table.CreateInstance<(string name, DateTime birthDate, int␣
↪heightInInches, decimal bankAccountBalance)>();

    // account.name is "John Galt"
    // account.heightInInches is 72
    // account.bankAccountBalance is 1234.56
}
```

Important: In the case of tuples, ***you need to have the same number of parameters and types; parameter names do not matter***, as ValueTuples do not hold parameter names at runtime using reflection.

Scenarios with more than 7 properties are not currently supported when converting to ValueTuple, and you will receive an exception if you try to map more than 7 properties.

The next section describes how to convert a horizontal table with more than one data row to a collection of objects.

`FillInstance`

The `FillInstance` extension method of the `DataTable` class populates an existing object instance with values from a table. Unlike `CreateInstance<T>` which creates a new object, `FillInstance` takes an existing object and fills its properties with table data.

This method is useful when you:

- Already have an object instance that you want to populate
- Need to update an existing object with test data

- Want to reuse an object and modify its properties based on table values

Basic Usage

The `FillInstance` method accepts any object and populates its properties based on the table structure. Like `CreateInstance<T>`, it supports both vertical and horizontal table layouts:

Listing 103: Step Definition File

```
[Given(@"I update the account with the following data:")]
public void GivenIUpdateTheAccountWithTheFollowingData(DataTable table)
{
    // Assume we already have an account instance
    var existingAccount = GetExistingAccount();

    // Fill the existing instance with table data
    table.FillInstance(existingAccount);

    // existingAccount properties are now updated with table values
}
```

Using `FillInstance` with `InstanceCreationOptions`

The `FillInstance` method also supports `InstanceCreationOptions` to control how properties are populated:

Listing 104: Step Definition File

```
[Given(@"I update the account with the following data:")]
public void GivenIUpdateTheAccountWithTheFollowingData(DataTable table)
{
    var existingAccount = GetExistingAccount();

    var options = new InstanceCreationOptions
    {
        VerifyAllColumnsBound = true
    };

    table.FillInstance(existingAccount, options);
}
```

For more information about `InstanceCreationOptions`, see the *InstanceCreationOptions section* below.

Example with Vertical Table

Listing 105: Feature File

```
Given I update the account with the following data:
    | Property             | Value     |
    | Name                 | Jane Doe  |
    | Bank Account Balance | 2500.75   |
```

Example with Horizontal Table

Listing 106: Feature File

```
Given I update the account with the following data:
    | Name     | Bank Account Balance |
    | Jane Doe | 2500.75              |
```

Both table formats will update the existing object's `Name` and `BankAccountBalance` properties with the specified values.

`CreateSet<T>`

The `CreateSet<T>` extension method of the `DataTable` class converts the table into an enumerable set of objects. For example, assume you have the following step:

```
Given these products exist
    | Sku   | Name             | Price |
    | BOOK1 | Atlas Shrugged   | 25.04 |
    | BOOK2 | The Fountainhead | 20.15 |
```

And you want to map rows in that table to the following class:

Listing 107: C# File

```
public class Product
{
    public string Sku { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

You can convert the table to a collection of Product objects in your step definition using `CreateSet<Product>()`:

Listing 108: Step Definition File

```
[Given(@"Given these products exist")]
public void GivenTheseProductsExist(DataTable table)
{
    var products = table.CreateSet<Product>();
    // ...
}
```

The `CreateSet<T>` method returns an `IEnumerable<T>` based on the matching data in the table. It contains the values for each object, making appropriate type conversions from string to the related property. Column headers are matched to property names in the same way as `CreateInstance<T>`.

InstanceCreationOptions

The CreateInstance<T> and CreateSet<T> methods have an overload with InstanceCreationOptions to modify creation behavior.

Listing 109: InstanceCreationOptions

```
public class InstanceCreationOptions
{
    public bool VerifyAllColumnsBound { get; set; }
    public bool RequireTableToProvideAllConstructorParameters { get; set; }
}
```

VerifyAllColumnsBound

Default is false. If true, check if every column is bound. If not, it throws a ColumnCouldNotBeBoundException.

For example, given the class:

Listing 110: C# Test object

```
public class Person
{
    public string FirstName { get; set;}
    public string LastName { get; set; }
}
```

And we feed a table with 'Name' as 'Mike':

Listing 111: C# Call

```
table.CreateInstance<Person>(new InstanceCreationOptions { VerifyAllColumnsBound = true }
↪);
```

We get a ColumnCouldNotBeBoundException with the message "Member or field Name not found."

Note: It won't check the other way around, e.g., we don't get an exception because FirstName is not in the table.

VerifyCaseInsensitive

Default is false. If true, checking if columns are bound becomes case-insensitive. Active *only* if VerifyAllColumnsBound is also true.

For example, given the class:

Listing 112: C# Test object

```
public class Person
{
    public string FirstName { get; set;}
    public string LastName { get; set; }
}
```

And we feed a table with 'firstName' as 'Mike' and 'lastName' as 'Smith':

Listing 113: C# Call

```
table.CreateInstance<Person>(new InstanceCreationOptions { VerifyAllColumnsBound = true,␣
↪VerifyCaseInsensitive = true });
```

We do not get an exception because the table fields match the class members, ignoring case. Behaviour for invalid and missing members in the table remain unchanged when compared to `VerifyAllColumnsBound` alone.

`RequireTableToProvideAllConstructorParameters`

Default is `false`. If `true`, use the constructor that exactly matches the table. If a matching constructor cannot be found, the helper will throw a `MissingMethodException`. If there are multiple candidates that take all members, prefer the constructor with the fewest parameters.

If `false`, the constructor with the most parameters is preferred.

For example:

Listing 114: C# Test object

```
public record Person(string FirstName, string LastName, int Age);
```

If `RequireTableToProvideAllConstructorParameters` is `false`, `table.CreateInstance<Person>` will succeed and Age will be `0`. If `RequireTableToProvideAllConstructorParameters` is `true`, we get a `System.MissingMethodException` with the message "Unable to find a suitable constructor to create an instance of ..."

`CompareToInstance<T>`

The `CompareToInstance<T>` extension method of the `DataTable` class makes it easy to compare the properties of an object to the table in your scenario. For example, you have a class like this:

Listing 115: C# File

```
public class Person
{
    public string FirstName { get; set;}
    public string LastName { get; set; }
    public int YearsOld { get; set; }
}
```

You want to compare it to a table in a step like this:

Listing 116: Feature File

```
Then the person should have the following values
    | Field      | Value |
    | First Name | John  |
    | Last Name  | Galt  |
    | Years Old  | 54    |
```

You can assert that the properties match with this simple step definition:

Listing 117: Step Definition File

```
[Binding]
public class PersonSteps
{
    ScenarioContext _scenarioContext;

    public PersonSteps(ScenarioContext scenarioContext)
    {
        _scenarioContext = scenarioContext;
    }

    [Then("the person should have the following values")]
    public void ThenThePersonShouldHaveTheFollowingValues(DataTable table){
        // you don't have to get person this way, this is just for demonstration purposes
        var person = _scenarioContext.Get<Person>();

        table.CompareToInstance<Person>(person);
    }
}
```

If FirstName is not “John”, LastName is not “Galt”, or YearsOld is not 54, a descriptive error showing the differences is thrown.

If the values match, no exception is thrown, and Reqnroll continues to process your scenario.

`CompareToSet<T>`

The `CompareToSet<T>` extension method of the `DataTable` class works similarly to `CompareToInstance<T>`, except it compares a collection of objects. For example, you have a class like this:

Listing 118: C# File

```
public class Account
{
    public string Id { get; set;}
    public string FirstName { get; set;}
    public string LastName { get; set;}
    public string MiddleName { get; set;}
}
```

You want to test that your system returns a specific set of accounts, like this:

Listing 119: Feature File

```
Then I get back the following accounts
    | Id | First Name | Last Name |
    | 1  | John       | Galt      |
    | 2  | Howard     | Roark     |
```

You can test your results with one call to CompareToSet:

Listing 120: Step Definition File

```
[Binding]
```

(continues on next page)

(continued from previous page)

```
public class AccountSteps
{
    ScenarioContext _scenarioContext;

    public AccountSteps(ScenarioContext scenarioContext)
    {
        _scenarioContext = scenarioContext;
    }

    [Then("I get back the following accounts")]
    public void ThenIGetBackTheFollowingAccounts(DataTable table)
    {
        // (or get the accounts from the database or web service)
        var accounts = _scenarioContext.Get<IEnumerable<Account>>();

        table.CompareToSet<Account>(accounts);
    }
}
```

In this example, `CompareToSet<T>` checks that two accounts are returned, and only tests the properties you defined in the table. **It does not test the order of the objects, only that one was found that matches.** If no record matching the properties in your table is found, an exception is thrown that includes the row number(s) that do not match up.

Comparing Sets When Order Matters

In use cases where the order should match, pass `true` as the second argument to `CompareToSet`:

Listing 121: Step Definition File

```
table.CompareToSet<Account>(accounts, true);
//                                    ^^^^
```

In addition to throwing an exception if property values do not match, Reqnroll will throw an exception if the order of the accounts doesn't match your expectations. This is useful when the order of things is determined by business rules, or in use cases like search results.

Column naming

The Reqnroll DataTable helpers use the values in your table to determine what properties to set in your object. However, the names of the columns do not need to match exactly - whitespace and casing is ignored. For example, the following two tables are treated as identical:

Listing 122: Feature File

```
| FirstName | LastName | DateOfBirth | HappinessRating |
```

Listing 123: Feature File

```
| First name | Last name | Date of birth | HAPPINESS rating |
```

This allows you to make your tables more readable to others.

Aliasing

If you have properties in your objects that are known by different terms within the business domain, these can be Aliased in your model by applying the attribute `TableAliases`. This attribute takes a collection of aliases as regular expressions that can be used to refer to the property in question.

For example, if you have an object representing an Employee, you might want to alias the `Surname` property:

Listing 124: C# File

```
public class Employee
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }

    [TableAliases("Last[]?Name", "Family[]?Name")]
    public string Surname { get; set; }
}
```

Test writers can then refer to this property as “Surname”, “Last Name”, “Lastname”, “Family Name” or “FamilyName”, and it will still be mapped to the correct column in your scenario.

The `TableAliases` attribute can be applied to a field, a property as a single attribute with multiple regular expressions, or as multiple attributes, depending on your preference.

Extensions

Out-of-the-box, the Reqnroll table helpers knows how to handle most C# base types. Types like `String`, `Bool`, `Enum`, `Int`, `Decimal`, `DateTime`, etc. are all covered (see full list of supported times). If you want to cover more types, including your own custom types, you can do so by registering your own instances of `IValueRetriever` and `IValueComparer`.

For example, you have a complex object like this:

Listing 125: C# File

```
public class Shirt
{
    public string Name { get; set; }
    public Color Color { get; set; }
}
```

You have a table like this:

Listing 126: Feature File

```
| Name | Color |
| XL   | Blue  |
| L    | Red   |
```

If you want to map `Blue` and `Red` to the appropriate instance of the `Color` class, you need to create an instance of `IValueRetriever` that can convert the strings to the `Color` instance.

You can register your custom `IValueRetriever` (and/or an instance of `IValueComparer` if you want to compare colors) like this:

Listing 127: Step Definition File

```
[Binding]
public static class Hooks1
{
    [BeforeTestRun]
    public static void BeforeTestRun()
    {
        Service.Instance.ValueRetrievers.Register(new ColorValueRetriever());
        Service.Instance.ValueComparers.Register(new ColorValueComparer());
    }
}
```

Examples on implementing these interfaces can be found as follows:

- IValueRetriever
- IValueComparer

Configuration

Some built in classes support configuration to adjust the default behaviour.

- DateTimeValueRetriever and DateTimeOffsetValueRetriever have a static DateTimeStyles property to adjust the style used to parse date times.

Example of usage:

Listing 128: Hook File

```
[Binding]
public static class Hooks1
{
    [BeforeTestRun]
    public static void BeforeTestRun()
    {
        DateTimeValueRetriever.DateTimeStyles = DateTimeStyles.AdjustToUniversal |␣
↪DateTimeStyles.AssumeUniversal;
    }
}
```

`NullValueRetriever`

> **Note**
>
> If you are not looking to transform data from `DataTable` objects, but rather looking to transform values in your step definitions, you'll likely want to look at *Step Argument Conversions* instead.

By default, non-specified (empty string) values are considered:

- An empty string for `String` and `System.Uri` values
- A null value for `Nullable<>` primitive types
- An error for non-nullable primitive types

To specify null values explicitly, add a `NullValueRetriever` to the set of registered retrievers, specifying the text to be treated as a null value, e.g.:

Listing 129: Hook File

```
[Binding]
public static class Hooks1
{
    [BeforeTestRun]
    public static void BeforeTestRun()
    {
        Service.Instance.ValueRetrievers.Register(new NullValueRetriever("<null>"));
    }
}
```

Note

The comparison is case-insensitive.

Using LINQ-based instance and set comparison

The **CompareToSet** extension method only checks for equivalence of collections which is a reasonable default. The **Reqnroll.Assist** namespace also contains extension methods for With based operations.

Consider the following steps:

Listing 130: Feature File

```
Scenario: Matching music collections
    When I have a music collection
        | Artist     | Album       |
        | Beatles    | Rubber Soul |
        | Pink Floyd | Animals     |
        | Muse       | Absolution  |
    Then it should match
        | Artist     | Album       |
        | Beatles    | Rubber Soul |
        | Pink Floyd | Animals     |
        | Muse       | Absolution  |
    And it should match
        | Artist     | Album       |
        | Beatles    | Rubber Soul |
        | Muse       | Absolution  |
        | Pink Floyd | Animals     |
    And it should exactly match
        | Artist     | Album       |
        | Beatles    | Rubber Soul |
        | Pink Floyd | Animals     |
        | Muse       | Absolution  |
```

(continues on next page)

(continued from previous page)

```
    But it should not match
        | Artist  | Album       |
        | Beatles | Rubber Soul |
        | Queen   | Jazz        |
        | Muse    | Absolution  |
    And it should not match
        | Artist  | Album       |
        | Beatles | Rubber Soul |
        | Muse    | Absolution  |
    And it should not exactly match
        | Artist     | Album       |
        | Beatles    | Rubber Soul |
        | Muse       | Absolution  |
        | Pink Floyd | Animals     |
```

With LINQ-based operations each of the above comparisons can be expressed using a single line of code:

Listing 131: Step Definition File

```
[Binding]
public class MusicCollectionSteps
{
    ScenarioContext _scenarioContext;

    public MusicCollectionSteps(ScenarioContext scenarioContext)
    {
        _scenarioContext = scenarioContext;
    }

    [When(@"I have a music collection")]
    public void WhenIHaveAMusicCollection(DataTable table)
    {
        var collection = table.CreateSet<Item>();

        _scenarioContext.Add("Collection", collection);
    }

    [Then(@"it should match")]
    public void ThenItShouldMatch(DataTable table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsTrue(table.RowCount == collection.Count() && table.ToProjection<Item>().
↪Except(collection.ToProjection()).Count() == 0);
    }

    [Then(@"it should exactly match")]
    public void ThenItShouldExactlyMatch(DataTable table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsTrue(table.ToProjection<Item>().SequenceEqual(collection.
```

(continues on next page)

(continued from previous page)

```
↪ToProjection()));
    }

    [Then(@"it should not match")]
    public void ThenItShouldNotMatch(DataTable table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsFalse(table.RowCount == collection.Count() && table.ToProjection<Item>
↪().Except(collection.ToProjection()).Count() == 0);
    }

    [Then(@"it should not exactly match")]
    public void ThenItShouldNotExactlyMatch(DataTable table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsFalse(table.ToProjection<Item>().SequenceEqual(collection.
↪ToProjection()));
    }
}
```

In a similar way we can implement containment validation:

Listing 132: Feature File

```
Scenario: Containment
    When I have a music collection
        | Artist      | Album        |
        | Beatles     | Rubber Soul  |
        | Pink Floyd  | Animals      |
        | Muse        | Absolution   |
    Then it should contain all items
        | Artist  | Album        |
        | Beatles | Rubber Soul  |
        | Muse    | Absolution   |
    But it should not contain all items
        | Artist  | Album        |
        | Beatles | Rubber Soul  |
        | Muse    | Resistance   |
    And it should not contain any of items
        | Artist  | Album       |
        | Beatles | Abbey Road  |
        | Muse    | Resistance  |
```

Listing 133: Step Definition File

```
[Binding]
public class MusicCollectionSteps
{
    ScenarioContext _scenarioContext;
```

(continues on next page)

(continued from previous page)

```
    public MusicCollectionSteps(ScenarioContext scenarioContext)
    {
        _scenarioContext = scenarioContext;
    }

    [Then(@"it should contain all items")]
    public void ThenItShouldContainAllItems(DataTable table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsTrue(table.ToProjection<Item>().Except(collection.ToProjection()).
↪Count() == 0);
    }

    [Then(@"it should not contain all items")]
    public void ThenItShouldNotContainAllItems(DataTable table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsFalse(table.ToProjection<Item>().Except(collection.ToProjection()).
↪Count() == 0);
    }

    [Then(@"it should not contain any of items")]
    public void ThenItShouldNotContainAnyOfItems(DataTable table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsTrue(table.ToProjection<Item>().Except(collection.ToProjection()).
↪Count() == table.RowCount);
    }
}
```

What if Artist and Album are properties of different entities? Look at this piece of code:

Listing 134: C# File

```
var collection = from artist in ctx.Artists
                 where artist.Name == "Muse"
                 join album in ctx.Albums
                     on album.ArtistId equals artist.ArtistId
                 select new
                 {
                     Artist = artist.Name,
                     Album = album.Name
                 };
```

The **Reqnroll.Assist** namespace has a generic class named `EnumerableProjection<T>`. If a type `T` is known at compile time, the `ToProjection` method converts a table or a collection into an instance of `EnumerableProjection`:

Listing 135: C# File

```
table.ToProjection<Item>();
```

But if we need to compare a table with the collection of anonymous types from the example above, we need to express this type in some way so ToProjection will be able to build an instance of specialized `EnumerableProjection`. This is done by sending a collection as an argument to **ToProjection**. And to support both sets and instances and avoid naming ambiguity, corresponding methods are called **ToProjectionOfSet** and **ToProjectionOfInstance**:

Listing 136: C# File

```
table.ToProjectionOfSet(collection);
table.ToProjectionOfInstance(instance);
```

Here are the definitions of Reqnroll DataTable extensions methods that convert tables and collections of IEnumerables to EnumerableProjection:

Listing 137: C# File

```
public static IEnumerable<Projection<T>> ToProjection<T>(this IEnumerable<T> collection,␣
↪DataTable table = null)
{
    return new EnumerableProjection<T>(table, collection);
}

public static IEnumerable<Projection<T>> ToProjection<T>(this DataTable table)
{
    return new EnumerableProjection<T>(table);
}

public static IEnumerable<Projection<T>> ToProjectionOfSet<T>(this DataTable table,␣
↪IEnumerable<T> collection)
{
    return new EnumerableProjection<T>(table);
}

public static IEnumerable<Projection<T>> ToProjectionOfInstance<T>(this DataTable table,␣
↪T instance)
{
    return new EnumerableProjection<T>(table);
}
```

Note that last arguments of `ToProjectionOfSet` and `ToProjectionOfInstance` methods are not used in method implementation. Their only purpose is to bring information about `T`, so the `EnumerableProjection` adapter class can be built properly. Now we can perform the following comparisons with anonymous types collections and instances:

Listing 138: C# Test File

```
[Test]
public void Table_with_subset_of_columns_with_matching_values_should_match_collection()
{
    var table = CreateTableWithSubsetOfColumns();

    table.AddRow(1.ToString(), "a");
```

(continues on next page)

(continued from previous page)

```
    table.AddRow(2.ToString(), "b");

    var query = from x in testCollection
                select new { x.GuidProperty, x.IntProperty, x.StringProperty };

    Assert.AreEqual(0, table.ToProjectionOfSet(query).Except(query.ToProjection()).
↪Count());
}

[Test]
public void Table_with_subset_of_columns_should_be_equal_to_matching_instance()
{
    var table = CreateTableWithSubsetOfColumns();

    table.AddRow(1.ToString(), "a");

    var instance = new { IntProperty = testInstance.IntProperty, StringProperty =␣
↪testInstance.StringProperty };

    Assert.AreEqual(table.ToProjectionOfInstance(instance), instance);
}
```

1.5.10 Sharing Data between Bindings

In many cases, different bindings need to exchange data during execution. Reqnroll provides several ways of sharing data between bindings.

Instance Fields

If the binding is an instance method, Reqnroll creates a new instance of the containing class for every scenario execution. Following the entity-based step organization rule, defining instance fields in the binding classes is an efficient way of sharing data between different steps of the same scenario that are related to the same entity.

The following example saves the result of the MVC action to an instance field in order to make assertions for it in a "then" step.

Listing 139: Step Definition File

```
[Binding]
public class SearchSteps
{
    private ActionResult actionResult;

    [When(@"I perform a simple search on '(.*)'")]
    public void WhenIPerformASimpleSearchOn(string searchTerm)
    {
        var controller = new CatalogController();
        actionResult = controller.Search(searchTerm);
    }

    [Then(@"the book list should exactly contain book '(.*)'")]
    public void ThenTheBookListShouldExactlyContainBook(string title)
    {
```

(continues on next page)

(continued from previous page)

```
        var books = actionResult.Model<List<Book>>();
        CustomAssert.Any(books, b => b.Title == title);
    }
}
```

Context Injection

Reqnroll supports a very simple dependency framework that is able to instantiate and inject class instances for the scenarios. With this feature you can group the shared state to context-classes, and inject them into every binding class that is interested in that shared state.

See more about this feature in the *Context Injection* page.

The following example defines a context class to store referred books. The context class is injected to a binding class.

Listing 140: C# File

```
public class CatalogContext
{
    public CatalogContext()
    {
        ReferenceBooks = new ReferenceBookList();
    }

    public ReferenceBookList ReferenceBooks { get; set; }
}
```

Listing 141: Step Definition File

```
[Binding]
public class BookSteps
{
    private readonly CatalogContext _catalogContext;

    public BookSteps(CatalogContext catalogContext)
    {
        _catalogContext = catalogContext;
    }

    [Given(@"the following books")]
    public void GivenTheFollowingBooks(DataTable table)
    {
        foreach (var book in table.CreateSet<Book>())
        {
            SaveBook(book);
            _catalogContext.ReferenceBooks.Add(book.Id, book);
        }
    }
}
```

ScenarioContext and FeatureContext

Reqnroll provides two context instances.

The *ScenarioContext* is created for each individual scenario execution and it is disposed when the scenario execution has been finished.

The *FeatureContext* is created when the first scenario is executed from a feature and disposed when the execution of the feature's scenarios ends. In the rare case, when you need to preserve state in the context of a feature, the `FeatureContext` instance can be used as a property bag.

Static Fields

Generally, using static fields can cause synchronization and maintenance issues and makes the unit testability hard. As the Reqnroll tests are executed synchronously and people usually don't write unit tests for the tests itself, these arguments are just partly valid for binding codes.

In some cases sharing a state through static fields can be an efficient solution.

1.5.11 Context Injection

Reqnroll supports a very simple dependency framework that is able to instantiate and inject class instances for scenarios. This feature allows you to group the shared state in context classes, and inject them into every binding class that needs access to that shared state.

To use context injection:

1. Create your POCOs (plain old CLR object), simple .NET classes, representing the shared data.
2. Define them as constructor parameters in every binding class that requires them.
3. Save the constructor argument to instance fields, so you can use them in the step definitions.

Rules:

- The life-time of these objects is limited to a scenario's execution.
- If the injected objects implement `IDisposable`, they will be disposed after the scenario is executed.
- The injection is resolved recursively, i.e. the injected class can also have dependencies.
- Resolution is done using public constructors only.
- If there are multiple public constructors, Reqnroll takes the first one.

The container used by Reqnroll can be customized, e.g. you can include object instances that have already been created, or modify the resolution rules. See the *Advanced options* section below for details.

Examples

In the first example we define a POCO for holding the data of a person and use it in a *given* and a *then* step that are placed in different binding classes.

Listing 142: C# File

```
public class PersonData // the POCO for sharing person data
{
  public string FirstName;
  public string LastName;
}
```

Listing 143: Step Definition File

```
[Binding]
public class MyStepDefs
{
  private readonly PersonData personData;
  public MyStepDefs(PersonData personData) // use it as ctor parameter
  {
    this.personData = personData;
  }

  [Given]
  public void The_person_FIRSTNAME_LASTNAME(string firstName, string lastName)
  {
    personData.FirstName = firstName; // write into the shared data
    personData.LastName = lastName;
    //... do other things you need
  }
}

[Binding]
public class OtherStepDefs // another binding class needing the person
{
  private readonly PersonData personData;
  public OtherStepDefs(PersonData personData) // ctor parameter here too
  {
    this.personData = personData;
  }

  [Then]
  public void The_person_data_is_properly_displayed()
  {
    var displayedData = ... // get the displayed data from the app
      // read from shared data, to perform assertions
    Assert.AreEqual(personData.FirstName + " " + personData.LastName,
      displayedData, "Person name was not displayed properly");
  }
}
```

The following example defines a context class to store referred books. The context class is injected into a binding class.

Listing 144: C# File

```
public class CatalogContext
{
    public CatalogContext()
    {
        ReferenceBooks = new ReferenceBookList();
    }

    public ReferenceBookList ReferenceBooks { get; set; }
}
```

Listing 145: Step Definition File

```
[Binding]
public class BookSteps
{
    private readonly CatalogContext _catalogContext;

    public BookSteps(CatalogContext catalogContext)
    {
        _catalogContext = catalogContext;
    }

    [Given(@"the following books")]
    public void GivenTheFollowingBooks(DataTable table)
    {
        foreach (var book in table.CreateSet<Book>())
        {
            SaveBook(book);
            _catalogContext.ReferenceBooks.Add(book.Id, book);
        }
    }
}
```

Advanced options

The container used by Reqnroll can be customized, e.g. you can include object instances that have already been created, or modify the resolution rules.

You can customize the container from a *plugin* or a before scenario *hook*. The class customizing the injection rules has to obtain an instance of the scenario execution container (an instance of `BoDi.IObjectContainer`). This can be done through constructor injection (see example below).

The following example adds the Selenium web driver to the container, so that binding classes can specify `IWebDriver` dependencies (a constructor argument of type `IWebDriver`).

Listing 146: Hook File

```
[Binding]
public class WebDriverSupport
{
  private readonly IObjectContainer objectContainer;

  public WebDriverSupport(IObjectContainer objectContainer)
  {
    this.objectContainer = objectContainer;
  }

  [BeforeScenario]
  public void InitializeWebDriver()
  {
    var webDriver = new FirefoxDriver();
    objectContainer.RegisterInstanceAs<IWebDriver>(webDriver);
  }
}
```

Custom Dependency Injection Frameworks

As mentioned above, the default Reqnroll container is `IObjectContainer` which is recommended for most scenarios. However, you may have situations where you need more control over the configuration of the dependency injection, or make use of an existing dependency injection configuration within the project you are testing, e.g. pulling in service layers for assisting with assertions in `Then` stages.

Consuming existing plugins

You can find the list of available plugins in the *Available Plugins* page.

To make use of these plugins, you need to add a reference to the plugin:

Listing 147: .NET CLI

```
dotnet add package Reqnroll.Autofac
```

This tells Reqnroll to load the runtime plugin and allows you to create an entry point to use this functionality. Once set up, your dependencies are injected into steps and bindings like they are with the `IObjectContainer`, but behind the scenes it will be pulling those dependencies from the DI container you added.

> **Note**
>
> One thing to note here is that each plugin has its own conventions for loading the entry point. This is often a static class with a static method containing an attribute that is marked by the specific plugin. You should check the requirements of the plugins you are using.

You can load all your dependencies within this handler section, or you can to inject the relevant IoC container into your binding sections like this:

Listing 148: Hook File

```
[Binding]
public class WebDriverPageHooks
{
    private readonly IKernel _kernel;

    // Inject in our container (using Ninject here)
    public WebDriverPageHooks(IKernel kernel)
    { _kernel = kernel; }

    private IWebDriver SetupWebDriver()
    {
        var options = new ChromeOptions();
        options.AddArgument("--start-maximized");
        options.AddArgument("--disable-notifications");
        return new ChromeDriver(options);
    }

    [BeforeScenario]
    public void BeforeScenario()
    {
        var webdriver = SetupWebDriver();
        _kernel.Bind<IWebDriver>().ToConstant(webdriver);
```

(continues on next page)

(continued from previous page)

```
    }

    [AfterScenario]
    public void AfterScenario()
    {
        var webDriver = _kernel.Get<IWebDriver>();

        // Output any screenshots or log dumps etc

        webDriver.Close();
        webDriver.Dispose();
    }
}
```

This gives you the option of either loading types up front or creating types within your binding sections so you can dispose of them as necessary.

Creating your own

We recommend looking at the autofac example and *plugins documentation* and following these conventions.

> **Note**
>
> Remember to adhere to the plugin documentation and have your assembly end in `.ReqnrollPlugin` e.g. `Reqnroll.AutoFac.ReqnrollPlugin`. Internal namespaces can be anything you want, but the assembly name must follow this naming convention or Reqnroll will be unable to locate it.

1.5.12 Scenario Context

`ScenarioContext` provides access to several functions, which are demonstrated using the following scenarios.

Accessing the ScenarioContext

In Bindings

To access the `ScenarioContext` you have to get it via *context injection*.

Example:

Listing 149: Step Definition File

```
[Binding]
public class Binding
{
    private ScenarioContext _scenarioContext;

    public Binding(ScenarioContext scenarioContext)
    {
        _scenarioContext = scenarioContext;
    }
}
```

Now you can access the ScenarioContext in all your step definitions with the `_scenarioContext` field.

In Hooks

Before/AfterTestRun

Accessing the `ScenarioContext` is not possible, as no `Scenario` is executed when the hook is called.

Before/AfterFeature

Accessing the `ScenarioContext` is not possible, as no `Scenario` is executed when the hook is called.

Before/AfterScenario

Accessing the `ScenarioContext` is done like in *normal bindings*

Before/AfterStep

Accessing the `ScenarioContext` is done like in *normal bindings*

Migrating from ScenarioContext.Current

The `ScenarioContext.Current` static accessor is obsolete, to make clear that you should avoid using these properties in future. The reason for moving away from these properties is that they do not work when running scenarios in parallel.

So how do you now access ScenarioContext?

Replace the code with `ScenarioContext.Current`...

Listing 150: Step Definition File

```
[Binding]
public class Bindings
{
    [Given(@"I have entered (.*) into the calculator")]
    public void GivenIHaveEnteredIntoTheCalculator(int number)
    {
        ScenarioContext.Current["Number1"] = number;
    }

    [BeforeScenario()]
    public void BeforeScenario()
    {
        Console.WriteLine("Starting " + ScenarioContext.Current.ScenarioInfo.Title);
    }
}
```

...to use *Context-Injection* to acquire an instance of ScenarioContext by requesting it via the constructor.

Listing 151: Step Definition File

```
[Binding]
public class Bindings
{
    private readonly ScenarioContext _scenarioContext;

    public Bindings(ScenarioContext scenarioContext)
```

(continues on next page)

(continued from previous page)

```
    {
        _scenarioContext = scenarioContext;
    }

    [Given(@"I have entered (.*) into the calculator")]
    public void GivenIHaveEnteredIntoTheCalculator(int number)
    {
        _scenarioContext["Number1"] = number;
    }

    [BeforeScenario()]
    public void BeforeScenario()
    {
        Console.WriteLine("Starting " + _scenarioContext.ScenarioInfo.Title);
    }
}
```

ScenarioContext.Pending

See *Mark Steps as Not Implemented*

Storing data in the ScenarioContext

ScenarioContext helps you store values in a dictionary between steps. This helps you to organize your step definitions better than using private variables in step definition classes.

There are some type-safe extension methods that help you to manage the contents of the dictionary in a safer way. To do so, you need to include the namespace Reqnroll.Assist, since these methods are extension methods of ScenarioContext.

ScenarioContext.ScenarioInfo

`ScenarioContext.ScenarioInfo` allows you to access information about the scenario currently being executed, such as its title and scenario and feature tags:

In the .feature file:

Listing 152: Feature File

```
@feature_tag
Feature: My feature

@rule_tag
Rule: My rule

@scenario_tag1 @scenario_tag2
Scenario: Showing information of the scenario

When I execute any scenario
Then the ScenarioInfo contains the following information
    | Field          | Value                                                |
    | Title          | Showing information of the scenario                  |
    | Tags           | scenario_tag1, scenario_tag2                         |
    | CombinedTags   | scenario_tag1, scenario_tag2, feature_tag, rule_tag  |
```

and in the step definition:

Listing 153: Step Definition File

```
private class ScenarioInformation
{
    public string Title { get; set; }
    public string[] Tags { get; set; }
    public string[] CombinedTags { get; set; }
}

[When(@"I execute any scenario")]
public void ExecuteAnyScenario(){}

[Then(@"the ScenarioInfo contains the following information")]
public void ScenarioInfoContainsInterestingInformation(DataTable table)
{
    // Create our small DTO for the info from the step
    var fromStep = table.CreateInstance<ScenarioInformation>();
    fromStep.Tags =  table.Rows[0]["Value"].Split(',');

    // Short-hand to the scenarioInfo
    var si = _scenarioContext.ScenarioInfo;

    // Assertions
    si.Title.Should().Equal(fromStep.Title);
    si.Tags.Should().BeEquivalentTo(fromStep.Tags);
    si.CombinedTags.Should().BeEquivalentTo(fromStep.CombinedTags);
}
```

`ScenarioContext.ScenarioInfo` also provides access to the current set of arguments from the scenario's examples in the form of an `IOrderedDictionary`:

Listing 154: Feature File

```
Scenario: Accessing the current example

When I use examples in my scenario
Then the examples are available in ScenarioInfo

    Examples:
    | Sport      | TeamSize |
    | Soccer     | 11       |
    | Basketball | 6        |
```

Listing 155: Step Definition File

```
public class ScenarioExamplesDemo
{
    private ScenarioInfo _scenarioInfo;

    public ScenarioExamplesDemo(ScenarioInfo scenarioInfo)
    {
        _scenarioInfo = scenarioInfo;
    }
```

(continues on next page)

(continued from previous page)

```
    [When(@"I use examples in my scenario")]
    public void IUseExamplesInMyScenario() {}

    [Then(@"the examples are available in ScenarioInfo")]
    public void TheExamplesAreAvailableInScenarioInfo()
    {
        var currentArguments = _scenarioInfo.Arguments;
        var currentSport = currentArguments["Sport"];
        var currentTeamSize = currentArguments["TeamSize"];
        Console.WriteLine($"The current sport is {currentSport}");
        Console.WriteLine($"The current sport allows teams of {currentTeamSize} players
↪");
    }
}
```

Another use is to check if an error has occurred, which is possible with the `ScenarioContext.TestError` property, which simply returns the exception.

You can use this information for "error handling". Here is an uninteresting example:

In the feature file…

Listing 156: Feature File

```
#This is not so easy to write a scenario for but I've created an AfterScenario-hook
@showingErrorHandling
Scenario: Display error information in AfterScenario
When an error occurs in a step
```

…and the step definition:

Listing 157: Step Definition File

```
[When("an error occurs in a step")]
public void AnErrorOccurs()
{
    "not correct".Should().Equal("correct");
}

[AfterScenario("showingErrorHandling")]
public void AfterScenarioHook()
{
    if(_scenarioContext.TestError != null)
    {
        var error = _scenarioContext.TestError;
        Console.WriteLine("An error ocurred:" + error.Message);
        Console.WriteLine("It was of type:" + error.GetType().Name);
    }
}
```

This is another example, that might be more useful:

Listing 158: Hook File

```
[AfterScenario]
public void AfterScenario()
{
    if(_scenarioContext.TestError != null)
    {
        WebBrowser.Driver.CaptureScreenShot(_scenarioContext.ScenarioInfo.Title);
    }
}
```

In this case, MvcContrib is used to capture a screenshot of the failing test and name the screenshot after the title of the scenario.

ScenarioContext.RuleInfo

`ScenarioContext.RuleInfo` allows you to access information about the rule for the scenario currently being executed, such as its title and tags:

In the .feature file:

Listing 159: Feature File

```
@feature_tag
Feature: My feature

@rule_tag
Rule: My rule

@scenario_tag1 @scenario_tag2
Scenario: Showing information of the scenario

When I execute any scenario
Then the RuleInfo contains the following information
    | Field         | Value    |
    | Title         | My rule  |
    | Tags          | rule_tag |
```

and in the step definition:

Listing 160: Step Definition File

```
private class RuleInformation
{
    public string Title { get; set; }
    public string[] Tags { get; set; }
}

[When(@"I execute any scenario")]
public void ExecuteAnyScenario(){}

[Then(@"the RuleInfo contains the following information")]
public void RuleInfoContainsInterestingInformation(DataTable table)
{
```

(continues on next page)

(continued from previous page)

```
    // Create our small DTO for the info from the step
    var fromStep = table.CreateInstance<RuleInformation>();
    fromStep.Tags =  table.Rows[0]["Value"].Split(',');

    // Short-hand to the ruleInfo
    var ri = _scenarioContext.RuleInfo;

    // Assertions
    ri.Title.Should().Equal(fromStep.Title);
    ri.Tags.Should().BeEquivalentTo(fromStep.Tags);
}
```

`ScenarioContext.RuleInfo` will be null if there is no valid `Rule` in the feature file.

ScenarioContext.CurrentScenarioBlock

Use `ScenarioContext.CurrentScenarioBlock` to query the "type" of step (Given, When or Then). This can be used to execute additional setup/cleanup code right before or after Given, When or Then blocks.

In the feature file:

Listing 161: Feature File

```
Scenario: Show the type of step we're currently on
    Given I have a Given step
    And I have another Given step
    When I have a When step
    Then I have a Then step
```

...and the step definition:

Listing 162: Step Definition File

```
[Given("I have a (.*) step")]
[Given("I have another (.*) step")]
[When("I have a (.*) step")]
[Then("I have a (.*) step")]
public void ReportStepTypeName(string expectedStepType)
{
    var stepType = _scenarioContext.CurrentScenarioBlock.ToString();
    stepType.Should().Equal(expectedStepType);
}
```

ScenarioContext.StepContext

Sometimes you need to access the currently executed step, e.g. to improve tracing. Use the `_scenarioContext.StepContext` property for this purpose.

1.5.13 Feature Context

Reqnroll provides access to the current test context using both `FeatureContext` and the more commonly used *ScenarioContext*. `FeatureContext` persists for the duration of the execution of an entire feature, whereas `ScenarioContext` only persists for the duration of a scenario.

Accessing the FeatureContext

in Bindings

To access the `FeatureContext` you have to get it via *Context-Injection*.

Example:

Listing 163: Step Definition File

```
[Binding]
public class Binding
{
    private FeatureContext _featureContext;

    public Binding(FeatureContext featureContext)
    {
        _featureContext = featureContext;
    }
}
```

Now you can access the FeatureContext in all your Bindings with the `_featureContext` field.

in Hooks

Before/AfterTestRun

Accessing the `FeatureContext` is not possible, as no `Feature` is executed, when the hook is called.

Before/AfterFeature

You can get the `FeatureContext` via parameter of the static method.

Example:

Listing 164: Hook File

```
[Binding]
public class Hooks
{
    [BeforeFeature]
    public static void BeforeFeature(FeatureContext featureContext)
    {
        Console.WriteLine("Starting " + featureContext.FeatureInfo.Title);
    }

    [AfterFeature]
    public static void AfterFeature(FeatureContext featureContext)
    {
        Console.WriteLine("Finished " + featureContext.FeatureInfo.Title);
    }
}
```

Before/AfterScenario

Accessing the `FeatureContext` is done like in *normal bindings*

Before/AfterStep

Accessing the `FeatureContext` is done like in *normal bindings*

Storing data in the FeatureContext

`FeatureContext` implements Dictionary<string, object>. So you can use the `FeatureContext` like a property bag.

FeatureContext.FeatureInfo

`FeatureInfo` provides more information than `ScenarioInfo`, but it works the same:

In the feature file:

Listing 165: Feature File

```
Scenario: Showing information of the feature

When I execute any scenario in the feature
Then the FeatureInfo contains the following information
    | Field          | Value                              |
    | Tags           | showUpInScenarioInfo, andThisToo   |
    | Title          | FeatureContext features            |
    | TargetLanguage | CSharp                             |
    | Language       | en-US                              |
    | Description    | In order to                        |
```

. . . and in the step definition:

Listing 166: Step Definition File

```
private class FeatureInformation
{
    public string Title { get; set; }
    public GenerationTargetLanguage TargetLanguage { get; set; }
    public string Description { get; set; }
    public string Language { get; set; }
    public string[] Tags { get; set; }
}

[When(@"I execute any scenario in the feature")]
public void ExecuteAnyScenario() { }

[Then(@"the FeatureInfo contains the following information")]
public void FeatureInfoContainsInterestingInformation(DataTable table)
{
    // Create our small DTO for the info from the step
    var fromStep =  table.CreateInstance<FeatureInformation>();
    fromStep.Tags = table.Rows[0]["Value"].Split(',');

    var fi = _featureContext.FeatureInfo;
```

(continues on next page)

(continued from previous page)

```
    // Assertions
    fi.Title.Should().Equal(fromStep.Title);
    fi.GenerationTargetLanguage.Should().Equal(fromStep.TargetLanguage);
    fi.Description.Should().StartWith(fromStep.Description);
    fi.Language.IetfLanguageTag.Should().Equal(fromStep.Language);
    for (var i = 0; i < fi.Tags.Length - 1; i++)
    {
        fi.Tags[i].Should().Equal(fromStep.Tags[i]);
    }
}
```

`FeatureContext` exposes a Binding Culture property that simply points to the culture the feature is written in (en-US in our example).

1.6 Execution Features

This part contains details of the following topics.

1.6.1 Executing Reqnroll Scenarios

Reqnroll generates executable tests from the scenarios defined in *feature files*. In order to execute these tests you can use your usual test execution tools.

Executing scenarios from console

From the console, you can use the `dotnet test` command.

1. Open a console
2. Change the current directory to the folder of your Reqnroll project (where the `.csproj` file is located)
3. Invoke `dotnet test`

Listing 167: Terminal

```
> dotnet test
  Determining projects to restore...
  All projects are up-to-date for restore.
[...]
Starting test execution, please wait...
[...]
Passed! - Failed: 0, Passed: 1, Skipped: 0, Total: 1, Duration: 76 ms -␣
↪MyReqnrollProject.dll
```

> **Note**
>
> Running the `dotnet test` command automatically restores the dependencies and builds your project by default.

On Windows the test execution is also possible using the `vstest.console.exe` tool. For that, make sure you use a Developer Command Prompt.

Listing 168: Developer Command Prompt

```
> vstest.console.exe .\bin\Debug\net8.0\MyReqnrollProject.dll
```

Executing scenarios from Visual Studio

Visual Studio contains a built-in test execution feature that can also be used for executing Reqnroll scenarios as well. In addition to that, other test execution tools, like ReSharper or NCrunch can also be used.

1. From the *Test* menu, choose the *Test Explorer* command. The *Test Explorer* tool window will open.
2. Wait until the tests are listed in the *Test Explorer* window. You might need to build your project first.
3. Locate the scenario you would like execute and invoke *Run* from the context menu. You can also use the *Run All Tests In View* button from the *Test Explorer* toolbar.

> **Note**
>
> Running the tests from the *Test Explorer* window will automatically save your files and build your project before executing the tests.

1.6.2 Executing Specific Scenarios

Executing a subset or only specific scenarios might be important locally and on the build pipeline.

Reqnroll converts the tags in your feature files to test case categories:

- NUnit: Category or TestCategory
- MSTest: TestCategory
- xUnit: Trait (similar functionality, Reqnroll will insert a Trait attribute with `Category` name)

This category can be used to filter the test execution in your build pipeline.

> **Note**
>
> Incorrect filter can lead to no test getting executed.

You don't have to include the `@` prefix in the filter expression.

Learn more about the filters in Microsoft's official documentation.

Examples

All the examples here are using `TestCategory`, but if you are using `xUnit` then you should use `Category` instead.

How to use the filters

Below are 2 scenarios where one of them has a tag: `@done`, and the other one does not have a tag.

Listing 169: Feature File

```
Feature: Breakfast

@done
```

(continues on next page)

(continued from previous page)

```
Scenario: Eating cucumbers
  Given there are 12 cucumbers
  When I eat 5 cucumbers
  Then I should have 7 cucumbers

Scenario: Use all the sugar
  Given there is some sugar in the cup
  When I put all the sugar to my coffee
  Then the cup is empty
```

If we would like to run only the scenario with `@done` tag, then the filter should look like:

```
TestCategory=done
```

Below are 2 scenarios where one of them has a tag: `@done`, and the other one has `@automated`.

Listing 170: Feature File

```
Feature: Breakfast

@done
Scenario: Eating cucumbers
  Given there are 12 cucumbers
  When I eat 5 cucumbers
  Then I should have 7 cucumbers

@automated
Scenario: Use all the sugar
  Given there is some sugar in the cup
  When I put all the sugar to my coffee
  Then the cup is empty
```

If we would like to run scenarios which have either `@done` or `@automated`:

```
TestCategory=done|TestCategory=automated
```

Below are 2 scenarios where one of them has a tag: `@done`, and the other one has `@automated`. There is also a `@US123` tag at Feature level.

Listing 171: Feature File

```
@US123
Feature: Breakfast

@done
Scenario: Eating cucumbers
  Given there are 12 cucumbers
  When I eat 5 cucumbers
  Then I should have 7 cucumbers
```

(continues on next page)

(continued from previous page)

```
@automated
Scenario: Use all the sugar
  Given there is some sugar in the cup
  When I put all the sugar to my coffee
  Then the cup is empty
```

If we would like to run only those scenarios, which have both `@US123` and `@done`:

```
TestCategory=US123&TestCategory=done
```

Below are 2 scenarios where one of them has two tags: `@done` and `@important`. There is another scenario, which has the `@automated` tag, and there is a `@us123` tag at Feature level.

Listing 172: Feature File

```
@US123
Feature: Breakfast

@done @important
Scenario: Eating cucumbers
  Given there are 12 cucumbers
  When I eat 5 cucumbers
  Then I should have 7 cucumbers

@automated
Scenario: Use all the sugar
  Given there is some sugar in the cup
  When I put all the sugar to my coffee
  Then the cup is empty
```

If we would like to run only those scenarios, which have both `@done` and `@important`:

```
TestCategory=done&TestCategory=important
```

dotnet test

Use the `--filter` command-line option:

```
dotnet test --filter TestCategory=done
```

```
dotnet test --filter "TestCategory=us123&TestCategory=done"
```

```
dotnet test --filter "TestCategory=done|TestCategory=automated"
```

vstest.console.exe

Use the `/TestCaseFilter` command-line option:

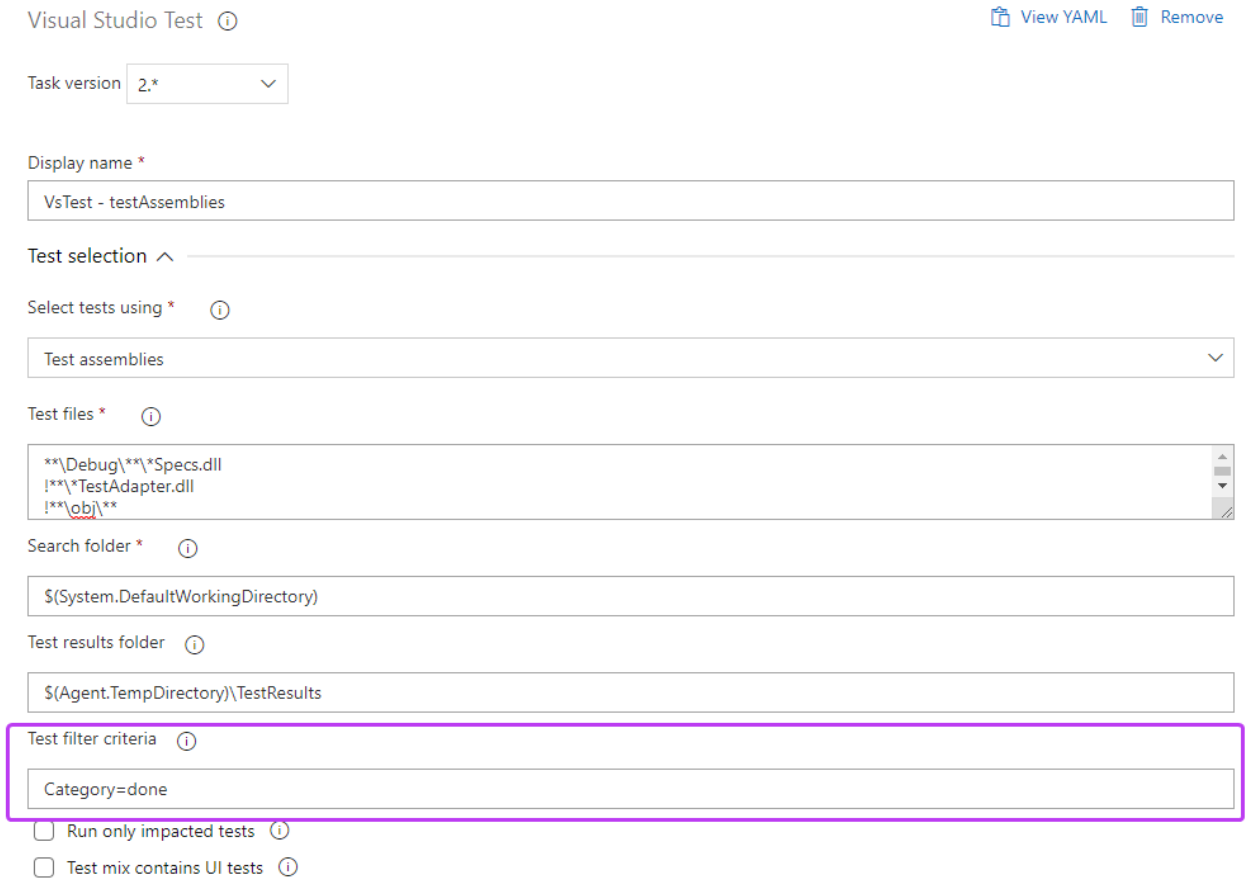
```
vstest.console.exe "C:\Temp\BookShop.AcceptanceTests.dll" /TestCaseFilter:
↪"TestCategory=done"
```

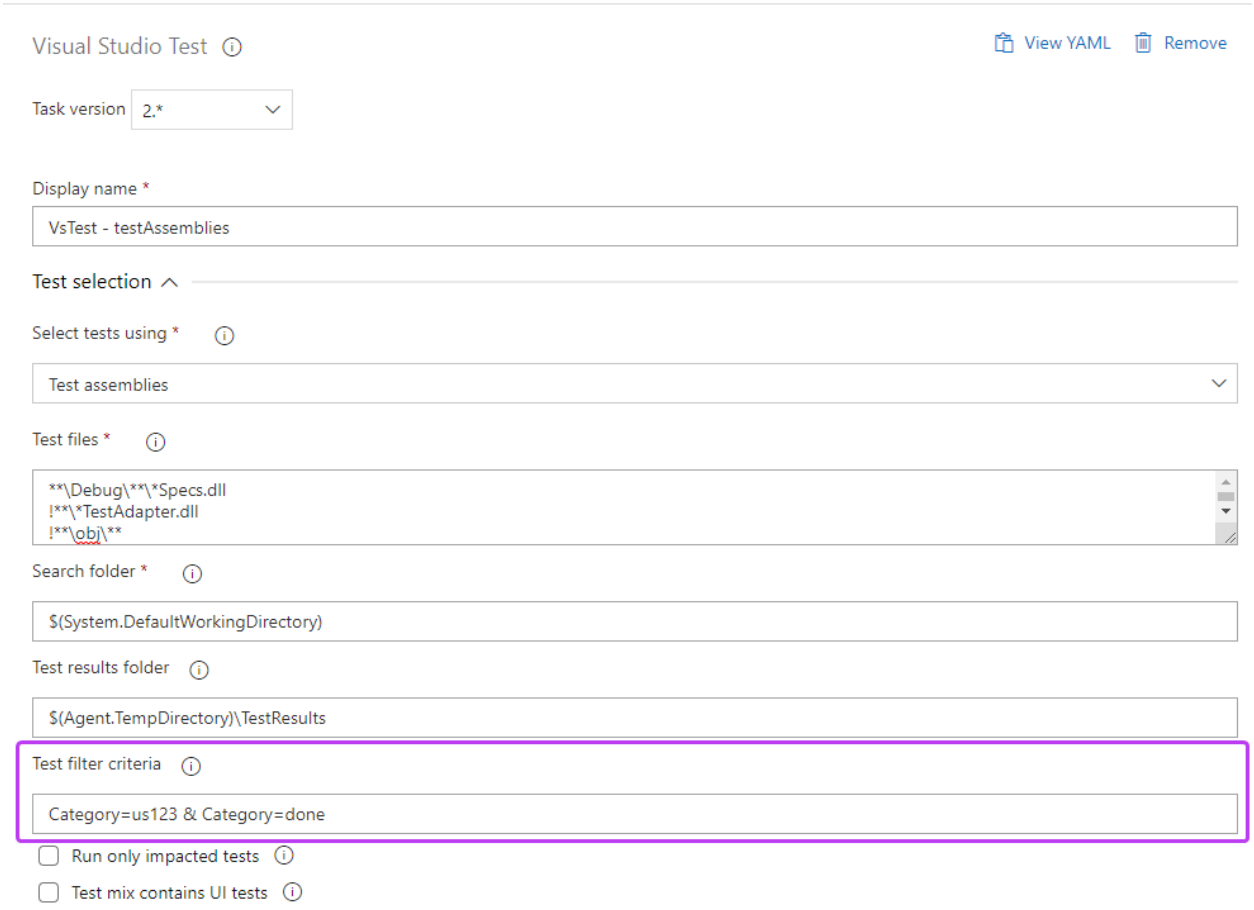
```
vstest.console.exe "C:\Temp\BookShop.AcceptanceTests.dll" /TestCaseFilter:
↪"TestCategory=us123&TestCategory=done"
```

```
vstest.console.exe "C:\Temp\BookShop.AcceptanceTests.dll" /TestCaseFilter:
↪"TestCategory=done|TestCategory=automated"
```

Azure DevOps - Visual Studio Test task

The filter expression should be provided in the "Test filter criteria" setting in the `Visual Studio Test` task:





Azure DevOps - .NET task

Alternatively you could use the dotnet task (DotNetCoreCLI) to run your tests. This works on all kinds of build agents:

```
- task: DotNetCoreCLI@2
  displayName: 'dotnet test'
  inputs:
    command: test
    projects: 'BookShop.AcceptanceTests'
    arguments: '--filter "TestCategory=done"'
```

```
- task: DotNetCoreCLI@2
  displayName: 'dotnet test'
  inputs:
    command: test
    projects: 'BookShop.AcceptanceTests'
    arguments: '--filter "TestCategory=us123&TestCategory=done"'
```

1.6.3 Mark Steps as Not Implemented

To mark a step as not implemented at runtime, you need to throw a `PendingStepException`. The Runtime of Reqnroll will detect this and will report the appropriate test result back to your test runner.

There are multiple ways to throw the exception.

Throwing the `PendingStepException`

You can throw the exception using a `throw` statement. In this case you have the possibility to provide a custom message.

Default Message

Listing 173: Step Definition File

```
[When("I set the current ScenarioContext to pending")]
public void WhenIHaveAPendingStep()
{
    throw new PendingStepException();
}
```

Custom Message

Listing 174: Step Definition File

```
[When("I set the current ScenarioContext to pending")]
public void WhenIHaveAPendingStep()
{
    throw new PendingStepException("custom pendingstep message");
}
```

Using ScenarioContext.Pending helper method

The `ScenarioContext` class has a static helper method to throw the default `PendingStepException`.

Listing 175: Step Definition File

```
[When("I set the current ScenarioContext to pending")]
public void WhenIHaveAPendingStep()
{
    ScenarioContext.StepIsPending();
}
```

1.6.4 Skipping Scenarios

You can skip programmatically scenarios using the `IUnitTestRuntimeProvider` interface.

Example Code

Listing 176: Step Definition File

```
[Binding]
public sealed class StepDefinitions
{
    private readonly IUnitTestRuntimeProvider _unitTestRuntimeProvider;

    public CalculatorStepDefinitions(IUnitTestRuntimeProvider unitTestRuntimeProvider)
    {
        _unitTestRuntimeProvider = unitTestRuntimeProvider;
    }
```

(continues on next page)

(continued from previous page)

```
    [When("your binding")]
    public void YourBindingMethod()
    {
        _unitTestRuntimeProvider.TestIgnore("This scenario is always skipped");
    }
}
```

Ignoring is like skipping the scenario. Be careful, as it behaves a little bit different for the different unit test runners (xUnit, NUnit, TUnit, MSTest).

1.6.5 Dry Run

> **Note**
>
> Introduced in Reqnroll v3

The Runtime of Reqnroll supports running tests in a "dry run" mode. This means that when each test is executed, the runtime will skip executing the code in your step handlers.

This can be useful for Pull Request/CI scenarios where you want to ensure all steps declared in the feature files match to a step handler in your C# code, but executing the test suite as normal is lengthy, expensive, or not possible. This feature is usually paired with the *runtime configuration* option `"missingOrPendingStepsOutcome": "Error"` to ensure any unbound steps are reported as errors. Another example use is for downstream reporting/analysis. By combining the use of Dry Run mode with *Formatters* you can obtain information for reports such as Feature names, lists of Scenarios, and lists of step definition bindings used by scenarios (and by inference, those not used).

Enabling Dry Run

To enable dry run mode, set the environment variable `REQNROLL_DRY_RUN=true` when executing your tests.

> **Note**
>
> It's usually not enough to simply set the environment variable in your session before invoking `dotnet test`. The .NET test runtime creates its own environment which may not inherit variables from the parent process. (May be different depending on the shell, test framework, OS, etc.)
>
> To ensure the environment variable is set correctly, use one of the options shown below.

Example with .NET CLI

Listing 177: .NET CLI

```
dotnet test -e "REQNROLL_DRY_RUN=true" MyReqnrollProject.csproj
```

Example with `.runsettings` file

Listing 178: MyRunSettings.runsettings

```
<?xml version="1.0" encoding="utf-8"?>
<RunSettings>
  <RunConfiguration>
    <EnvironmentVariables>
      <REQNROLL_DRY_RUN>true</REQNROLL_DRY_RUN>
    </EnvironmentVariables>
  </RunConfiguration>
</RunSettings>
```

Listing 179: Consuming the .runsettings file

```
dotnet test --settings MyRunSettings.runsettings
```

1.6.6 Test Results

When Reqnroll tests are executed, the execution engine processes the test steps, executing the necessary test logic and either finishing successfully or failing for various reasons.

Test Passes

While executing the tests, the engine outputs information about the execution to the test output. In some cases it makes sense to investigate the test output even if the test passes.

By default, the test output includes the executed test steps, the invoked test logic methods (*bindings*) and the execution time for longer operations. You can *configure* the information displayed in the test output.

Test Fails due to an Error

A test can fail because it causes an error. The test output contains more detailed information, e.g. a stack trace.

Test Fails due to step binding problems

A test can fail if the test logic (bindings) have not yet been implemented (or are configured improperly). By default, this is reported as an "inconclusive" result, although you can *configure* how Reqnroll behaves in this case.

Note

Some unit test frameworks do not support inconclusive result. In this case the problem is reported as an error instead.

The test output can be very useful if you are missing bindings, as it contain a step binding method skeleton you can copy to your project and extend with the test logic.

Ignored Tests

Just like with normal unit tests, you can also ignore Reqnroll tests. To do so, tag the feature or scenario with the `@ignore` tag.

Danger

Don't forget that ignoring a test will not solve any problems with your implementation…

1.6.7 Parallel Execution

Reqnroll scenarios are often automated as integration or system level tests. The system under test (SUT) might have several external dependencies and a more complex internal architecture. The key design question when running the tests in parallel is how the parallel test executions can be isolated from each other.

Test Isolation Levels

Determining the ideal level of isolation for your automated tests is a tradeoff. The higher the isolation of the parallel tests the smaller the likelihood of conflicts on shared state and dependencies, but at the same time the higher the execution time and amount of resources needed to maintain the isolated environments.

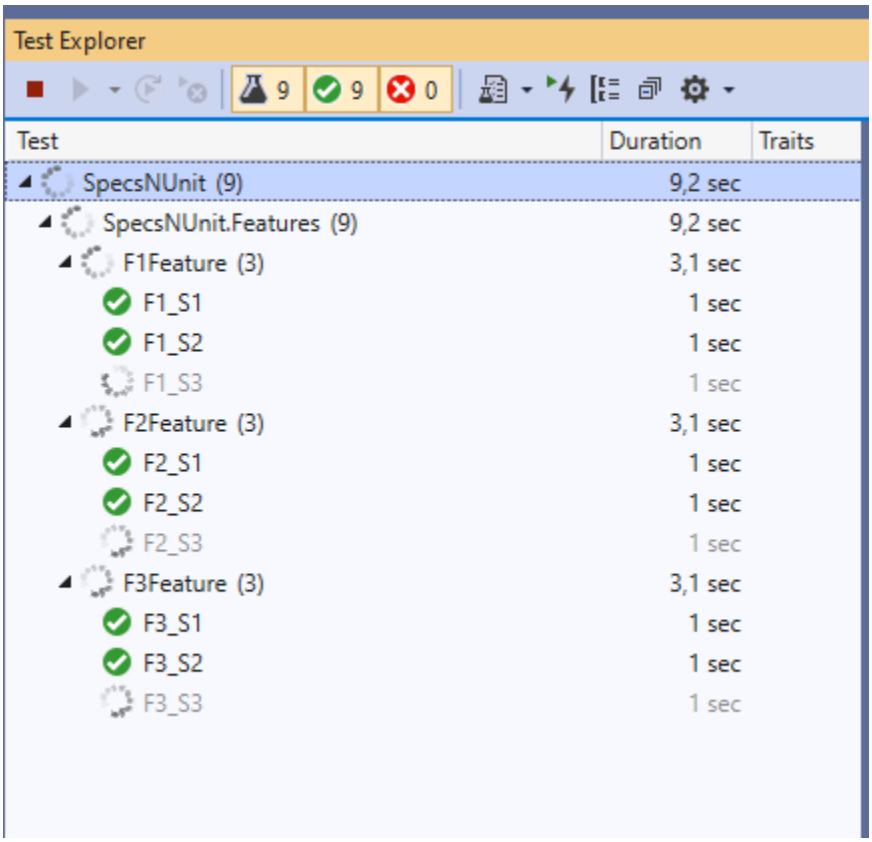
Isolation level	Description	Runner support
Thread	Test threads run as threads in the same process and application domain. Only the thread-local state is isolated.	NUnit, MsTest, xUnit
Process	Test threads run in separate processes.	VSTest per test assembly
Agent	Test threads run on multiple agents.	E.g. VSTest task

Parallel Scheduling Unit

Depending on the test isolation level and the used test runner tools you can consider different "units of scheduling" that can run in parallel with each other. When using Reqnroll we can consider the parallel scheduling on the level of scenarios, features and test assemblies.

Scheduling unit	Description	Runner support
Scenario	Scenarios can run in parallel with each other (also from different features)	NUnit, MsTest
Feature	Features can run in parallel with each other. Scenarios from the same feature are running on the same test thread.	NUnit, MsTest, xUnit
Test assembly	Different test assemblies can run in parallel with each other	e.g. VSTest

Running Reqnroll features in parallel with thread-level isolation



Properties

- Tests are running in multiple threads within the same process and the same application domain.
- Only the thread-local state is isolated.
- Smaller initialization footprint and lower memory requirements.
- The Reqnroll binding registry (step definitions, hooks, etc.) and some other core services are shared across test threads.

Requirements

- You have to use a test runner that supports in-process parallel execution (NUnit and MsTest supports scenario-level, xUnit supports feature-level)
- You have to ensure that your code does not conflict on static state.
- You must not use the static context properties of Reqnroll `ScenarioContext.Current`, `FeatureContext.Current` or `ScenarioStepContext.Current` (see further information below).
- You have to configure the test runner to execute the Reqnroll features in parallel with each other (see configuration details below).

Execution Behavior

- `[BeforeTestRun]` and `[AfterTestRun]` hooks (events) are executed only once on the first thread that initializes the framework. Executing tests in the other threads is blocked until the hooks have been fully executed on the first thread.
- As a general guideline, **we do not recommend using the `[BeforeFeature]` and `[AfterFeature]` hooks and the `FeatureContext` when running the tests with method-level parallelism**, because in this case there is no guarantee that these hooks will be executed only once per feature and that there will be only one instance of the `FeatureContext` per feature. The lifetime of the `FeatureContext` (that starts and finishes by invoking the `[BeforeFeature]` and `[AfterFeature]` hooks) is controlled by the test runner. So in the case of running scenarios with method-level parallelism, a feature's scenarios could be distributed across multiple workers and run in parallel. Therefore, each scenario could have its own dedicated `FeatureContext`, or some scenarios of a feature could share the same `FeatureContext`. It all depends on how the test runner (e.g. NUnit or MsTest) distributes the scenarios among the worker threads - which is not predictable or controllable. Because of this behavior of the test runner, Reqnroll can't share the `FeatureContext` between parallel threads. If you want to have a truly singleton `FeatureContext`, and `[BeforeFeature]` and `[AfterFeature]` hook execution, you must use either class-level parallelism or disable parallelism entirely so that scenarios of a feature are all executed on the **same worker thread**.
  - However, if you still want to use method-level parallelism and a `FeatureContext` in your test suite, then **the following things will be true**:
    * The `FeatureContext` and feature-level DI container will remain consistent **per feature, per test thread**. This means that anything you register in the feature container will be resolvable in the `[AfterFeature]` **per test thread**.
    * A given `[BeforeFeature]` or `[AfterFeature]` will only be executed once **per test thread** that runs a scenario of a feature.
    * Types you register in the feature-level DI container that implement `IDisposable` will still be disposed **per feature, per test thread**. (Keep this in mind if you try to work around this parallelism behavior to regain singleton-like behavior. E.g. by using static instances, `Lazy<>`, thread-safe collections, etc.)
- Scenarios and their related hooks (Before/After scenario, scenario block, step) are isolated in the different threads during execution and do not block each other. Each thread has a separate (and isolated) `ScenarioContext`.
- The test trace listener (that outputs the scenario execution trace to the console by default) is invoked asynchronously from the multiple threads and the trace messages are queued and passed to the listener in serialized form. If the test trace listener implements `Reqnroll.Tracing.IThreadSafeTraceListener`, the messages are sent directly from the threads.

NUnit Configuration

By default, NUnit does not run the tests in parallel. Parallelization must be configured by setting an assembly-level attribute in the Reqnroll project.

Listing 180: C# file for configuring feature-level parallelization

```
using NUnit.Framework;
[assembly: Parallelizable(ParallelScope.Fixtures)]
```

Listing 181: C# file for configuring scenario-level parallelization

```
using NUnit.Framework;
[assembly: Parallelizable(ParallelScope.Children)]
```

MSTest Configuration

By default, MsTest does not run the tests in parallel. Parallelisation must be configured by setting an assembly-level attribute in the Reqnroll project.

Listing 182: C# file for configuring feature-level parallelization

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

Listing 183: C# file for configuring scenario-level parallelization

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

xUnit Configuration

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

Thread-safe ScenarioContext, FeatureContext and ScenarioStepContext

When using parallel execution accessing the obsolete `ScenarioContext.Current`, `FeatureContext.Current` or `ScenarioStepContext.Current` static properties is not allowed. Accessing these static properties during parallel execution throws a `ReqnrollException`.

To access the context classes in a thread-safe way you can either use context injection or the instance properties of the `Steps` base class. For further details please see the *FeatureContext* and *ScenarioContext* documentation.

Excluding Reqnroll features from parallel execution

To exclude specific features from running in parallel with any other features, see the `addNonParallelizableMarkerForTags` *configuration* option.

Please note that xUnit requires additional configuration to ensure that non parallelizable features do not run in parallel with any other feature. This configuration is automatically provided for users via the xUnit plugin (so no additional effort is required). The following class will be defined within your test assembly for you:

Listing 184: C# File

```
[CollectionDefinition("ReqnrollNonParallelizableFeatures", DisableParallelization =␣
↪true)]
public class ReqnrollNonParallelizableFeaturesCollectionDefinition
{
}
```

Running Reqnroll scenarios in parallel with process isolation

If there are no external dependencies or they can be cloned for parallel execution, but the application architecture depends on static state (e.g. static caches etc.), the best way is to execute tests in parallel isolated by process. This ensures that every test execution thread is hosted in a separate process and hence static state is not accessed in parallel.

Properties

- Tests threads are separated by a process boundary.
- Also the static memory state is isolated. Conflicts might be expected on external dependencies only.
- Bigger initialization footprint and higher memory requirements.

Requirements

- You have to use VSTest task.

Execution Behavior

- `[BeforeTestRun]` and `[AfterTestRun]` hooks are executed for each individual test execution thread, so you can use them to initialize/reset shared memory.
- Each test thread manages its own enter/exit feature execution workflow. The `[BeforeFeature]` and `[AfterFeature]` hooks may be executed multiple times in different test threads if they run scenarios from the same feature file. The execution of these hooks do not block one another, but the Before/After feature hooks are called in pairs within a single thread (the `[BeforeFeature]` hook of the next scenario is only executed after the `[AfterFeature]` hook of the previous one). Each test thread has a separate (and isolated) `FeatureContext`.

1.6.8 Debugging

Reqnroll Visual Studio integration also supports debugging the execution of your tests. Just like in the source code files of your project, you can place breakpoints in the Reqnroll feature files. Whenever you execute the generated tests in debug mode, the execution will stop at the specified breakpoints and you can execute the steps one-by-one using "Step Over" (F10), or follow the detailed execution of the bindings using "Step Into" (F11).

If the execution of a Reqnroll test is stopped at a certain point of the binding (e.g. because of an exception), you can navigate to the current step in the feature file from the "Call Stack" tool window in Visual Studio.

By default, you cannot debug inside the generated .feature.cs files. You can enable debugging for these files by setting *allowDebugGeneratedFiles setting in generator section* to `true`.

1.6.9 Output API

The Reqnroll Output API allows you to display texts and attachments in your IDE's test explorer output window and also in the test result logs.

To use the Reqnroll output API interface you must inject the `IReqnrollOutputHelper` interface via *Context Injection*:

Listing 185: Step Definition File

```
private readonly IReqnrollOutputHelper _reqnrollOutputHelper;

public CalculatorStepDefinitions(IReqnrollOutputHelper outputHelper)
{
    _reqnrollOutputHelper = outputHelper;
}
```

There are two methods available:

`WriteLine(string text)`

This method adds text:

```
_reqnrollOutputHelper.WriteLine("TEXT");
```

`AddAttachment(string filePath)`

This method adds an attachment and requires the file path:

```
_reqnrollOutputHelper.AddAttachment("filePath");
```

> **Note**
>
> The attachment file can be stored anywhere. But it is important to keep mind that if a local file is added, it will only work on your machine and not accessible when shared with others.

> **Note**
>
> Handling of attachments depends on your runner. MStest and NUnit currently support this feature but xUnit do **not**.

1.6.10 Color Test Result Output

Configuration

To enable the colorization of the test result output, you can turn the `trace.coloredOutput` to true in the *configuration*

The color will only be visible in supported place, like in Rider test runner or in the console when running test using `dotnet test`.

You can turn off the color by setting `NO_COLOR=1` environment variable. This can be useful when you run the tests on a build server that does not support colors.

Customization

You can customize the colors by configuring a Hook and injecting `IColorOutputTheme` like in the following example.

Listing 186: Hook File

```
[Binding]
public class Hooks
{
    [BeforeTestRun]
    public static void ConfigureColor(IColorOutputTheme colorOutputTheme)
    {
        colorOutputTheme.Keyword = AnsiColor.Reset;
        colorOutputTheme.Error = AnsiColor.Composite(AnsiColor.Bold, AnsiColor.
↪Foreground(TerminalRgbColor.FromHex("FF8EF3")));
        colorOutputTheme.Done = AnsiColor.Foreground(TerminalRgbColor.FromHex("3A86FF"));
    }
}
```

1.7 Reporting

Reqnroll supports different reporting options.

1.7.1 Reqnroll formatters

> **Note**
>
> Reqnroll formatters are only available in Reqnroll v3.0 or later.

Reqnroll provides a *formatter* infrastructure, similar to Cucumber formatters. The formatters can be used to generate reports of the test execution. Reqnroll provides built-in formatters (*HTML*, *Cucumber Messages*) and can be extended with custom formatters. A list of publicly available custom formatter plugin can be found in the *Available Plugins* page.

In order to generate a report with a formatter, you need to enable it. You can enable multiple formatters as well. The easiest way to enable a formatter is to add a `formatters` section to the `reqnroll.json` configuration file or with environment variables.

The following example enables the HTML formatter and configures the output file as `reqnroll_report.html`.

Listing 187: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

    "formatters": {
        "html" : { "outputFilePath" : "reqnroll_report.html" }
    }
}
```

The same configuration can be achieved by setting an environment variable before running the tests.

```
$env:REQNROLL_FORMATTERS_HTML = 'outputFilePath=reqnroll_report.html'
dotnet test
```

See *Formatter Configuration* for further details about formatter configuration.

HTML formatter

The *HTML formatter* generates a stand-alone HTML file with all executed feature files and the test execution results.

The result can also be used as a Living Documentation and it is the default replacement of the `SpecFlow+ LivingDoc Generator`.

The HTML generator uses the Cucumber React Components and the Cucumber HTML Formatter to render the feature files and the test results to HTML.

> **Note**
>
> Currently the HTML formatter does not allow customizations or templating. We plan to add customization options in future releases.
>
> With this release, customizations can be done either by a custom formatter (based on the HTML formatter implementation) or by using the *Cucumber Message formatter* to generate a Cucumber Messages report and use custom tooling to generate a HTML report from it (see this sample project as an example).

The HTML formatter can be enabled with a `html` section in the configuration file within `formatters`. By default, it generates a `reqnroll_report.html` file to the project output folder (e.g., `bin/Debug/net8.0/`), but this can be configured using the `outputFilePath` setting.

The following configuration file enables the HTML formatter and generates the report to the `report/living_doc.html` file within the project output folder.

Listing 188: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

    "formatters": {
        "html" : { "outputFilePath" : "report/living_doc.html" }
    }
}
```

Cucumber Messages formatter

The *Cucumber Messages formatter* generates a Cucumber Messages (`.ndjson` or `.jsonl`) file. A Cucumber Messages file is a standardized file format for Cucumber-based test tooling (like Reqnroll) to feed test results to reporting systems.

> **Note**
>
> For more information about Cucumber Messages and the tooling that consumes them, please see the Cucumber Messages page on Github.

The Cucumber Messages formatter can be enabled with a `message` section in the configuration file within `formatters`. By default, it generates a `reqnroll_report.ndjson` file to the project output folder (e.g., `bin/Debug/net8.0/`), but this can be configured using the `outputFilePath` setting.

The following configuration file enables the Cucumber Messages formatter and generates the report to the `report/cucumber_messages.ndjson` file within the project output folder.

Listing 189: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

    "formatters": {
        "message" : { "outputFilePath" : "report/cucumber_messages.ndjson" }
    }
}
```

Creating a custom formatter

> **Warning**
>
> The formatter infrastructure is new, therefore the interface details and the implementation rules might change significantly even in minor version changes.

To create a custom formatter, you need to create a Reqnroll *Runtime plugin*, create a class that implements the `ICucumberMessageFormatter` interface and register it to the test run (global) DI container with a new name. This

name must be the same as returned by the `ICucumberMessageFormatter.Name` property of the formatter implementation.

The following example registers the `CustomFormatter` with the name `custom`.

```
using Reqnroll.Formatters;
using Reqnroll.Plugins;
using Reqnroll.UnitTestProvider;
using ReqnrollFormatters.Custom;

[assembly: RuntimePlugin(typeof(CustomFormatterPlugin))]

namespace ReqnrollFormatters.Custom;

public class CustomFormatterPlugin : IRuntimePlugin
{
    public void Initialize(RuntimePluginEvents runtimePluginEvents,␣
↪RuntimePluginParameters runtimePluginParameters, UnitTestProviderConfiguration␣
↪unitTestProviderConfiguration)
    {
        runtimePluginEvents.RegisterGlobalDependencies += (_, args) =>
        {
            args.ObjectContainer.RegisterTypeAs<CustomFormatter,␣
↪ICucumberMessageFormatter>("custom");
        };
    }
}
```

The created formatter can be enabled with a `custom` section in the configuration file within `formatters`.

Listing 190: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

    "formatters": {
        "custom" : { "outputFilePath" : "custom_report.txt" }
    }
}
```

For a complete example that contains a custom formatter, please check our Custom Formatter Test Project.

Troubleshooting formatter errors

In order to diagnose formatter errors you can enable formatter logging.

> **Warning**
>
> The formatter logging infrastructure is experimental. In later versions we will provide easier configuration. The method described here might also change even during minor version updates.

In order to enable formatter log, you need to add the following class to your project. The class is a simple *Reqnroll runtime plugin* that configures a formatter logger.

Listing 191: EnableFormatterLogPlugin.cs

```
using Reqnroll.Formatters.RuntimeSupport;
using Reqnroll.Plugins;
using Reqnroll.UnitTestProvider;

[assembly: RuntimePlugin(typeof(EnableFormatterLogPlugin))]

namespace Reqnroll.Formatters.RuntimeSupport;

public class EnableFormatterLogPlugin : IRuntimePlugin
{
    public void Initialize(RuntimePluginEvents runtimePluginEvents,␣
↪RuntimePluginParameters runtimePluginParameters, UnitTestProviderConfiguration␣
↪unitTestProviderConfiguration)
    {
        runtimePluginEvents.CustomizeGlobalDependencies += (_, args) =>
        {
            args.ObjectContainer.RegisterTypeAs<TraceListenerFormatterLog, IFormatterLog>
↪();
        };
    }
}
```

Once the formatter log is enabled, you can run the tests in with verbose console mode and investigate the result if you see any errors.

```
dotnet test --logger "console;verbosity=detailed"
```

Because the lengthy log, it is recommended to save the console output to a file.

```
dotnet test --logger "console;verbosity=detailed" > log.txt
```

Note: The built-in `TraceListenerFormatterLog` does not seem to produce visible results for NUnit (works with MsTest). As an alternative, you can implement a simple listener that saves the messages to a file (the file will be generated in the output folder, e.g. `bin\Debug\net8.0`).

```
public class FileFormatterLog : IFormatterLog
{
    private readonly List<string> _entries = new();

    public void WriteMessage(string message)
    {
        _entries.Add($"{DateTime.Now:HH:mm:ss.fff}: {message}");
    }

    public void DumpMessages()
    {
        File.WriteAllLines("formatter_log.txt", _entries);
    }
}
```

1.7.2 .NET test execution framework loggers

The tests generated by Reqnroll from the BDD scenarios integrate into the .NET test execution model, and therefore the generic loggers provided for .NET can also be used with Reqnroll.

The following example uses the TRX logger to produce a TRX test result file as the result of the execution.

Listing 192: Terminal

```
> dotnet test --logger trx
[...]
Results File: C:\MySolution\MyReqnrollProject\TestResults\gaspar_WORK_2025-07-29_15_38_
↪19.trx
[...]
```

The loggers can have additional parameters. For example, for the TRX logger you can specify the output file name. For further details please see the `--logger` option of the `dotnet test` command documentation.

Listing 193: Terminal

```
> dotnet test --logger "trx;logfilename=result.trx"
[...]
Results File: C:\MySolution\MyReqnrollProject\TestResults\result.trx
[...]
```

A few common logger use cases can be found in the following table. The complete list of the available loggers can be found in the VSTest documentation.

Logger	Option	Description
`trx`	`--logger "trx;logfilename=result.trx"`	Can be used to produce a generic TRX test result file that is supported by many tools and can also be opened with Visual Studio.
`conso`	`--logger "console;verbosity=detailed"`	Can be used to diagnose test execution problems.
`html`	`--logger "html;logfilename=result.html"`	Produces a basic HTML report.

1.7.3 Integrations to external reporting solutions

Other external or custom reporting solutions can also be used with Reqnroll. The following list contains a few options that can be considered.

> **Note**
>
> In addition to the listed options below, you can also check the *Available Plugins* page for other publicly available formatter plugins that can be used to generate reports in different formats.

- Allure Report, one of the most popular open-source reporting tool with an integration with Reqnroll, so you can use Allure as a reporting tool with Reqnroll. See the Allure Report Reqnroll integration documentation for details.

- Expressium LivingDoc, an open-source tool that generates a single HTML test report in a Living Documentation style for ReqnRoll projects.
- Johan Van Berkel (ICT Improve) posted about an enterprise reporting solution they have built from open-source components that you can also consider. See details in his post on LinkedIn.

1.8 Extend Reqnroll

This part contains details of the following topics.

1.8.1 Value Retrievers

Reqnroll can turn properties in a table like this:

```
Given I have the following people
| First Name | Last Name | Age | IsAdmin |
| John       | Guppy     | 40  | true    |
```

Into an object like this:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public bool IsAdmin { get; set; }
}
```

With commands like these:

```
    [Given(@"I have the following people")]
    public void x(DataTable table)
    {
        var person = table.CreateInstance<Person>();
        // OR
        var people = table.CreateSet<Person>();
    }
```

But how does Reqnroll match the values in the table with the values in the object? It does so with Value Retrievers. There are value retrievers defined for almost every C# base type, so mapping most basic POCOs can be done with Reqnroll without any modification.

Extending with your own value retrievers

Often you might have a more complicated POCO type, one that is not comprised solely of C# base types. Like this one:

```
    public class Shirt
    {
        public Guid Id { get; set; }
        public string Name { get; set; }
        public Color Color { get; set; }
    }
```

(continues on next page)

(continued from previous page)

```
    public class Color
    {
        public string Name { get; set; }
    }
```

Simple example how to process the human readable color 'red' to the Hex value:

```
| First Name | ShirtColor |
| Scott      | Red        |
```

The table will be processed, and the following code can be used to capture the table translation and customize it:

```
public class ShirtColorValueRetriever : IValueRetriever
    {
        public bool CanRetrieve(KeyValuePair<string, string> keyValuePair, Type␣
↪targetType, Type propertyType)
        {
           if (!keyValuePair.Key.Equals("ShirtColor"))
           {
               return false;
           }

           bool value;
           if (Color.TryParse(keyValuePair.Value, out value))
           {
              return true;
           }
        }

        public object Retrieve(KeyValuePair<string, string> keyValuePair, Type␣
↪targetType, Type propertyType)
        {
            return Color.Parse(keyValuePair.Value).HexCode;
        }
    }
```

Registering Custom ValueRetrievers

Before you can utilize a custom `ValueRetriever`, you'll need to register it. We recommend doing this prior to a test run using the `[BeforeTestRun]` attribute and `Service.Instance.ValueRetrievers.Register()`. For example:

```
[Binding]
public static class Hooks
{
    [BeforeTestRun]
    public static void BeforeTestRun()
    {
        Service.Instance.ValueRetrievers.Register(new MyCustomValueRetriever());
    }
}
```

1.8.2 Plugins

Reqnroll supports the following types of plugins:

- Runtime
- Generator

> **Note**
>
> The versioning and compatibility of the Reqnroll plugins is described in detail in the *Compatibility page*.

All types of plugins are created in a similar way.

Runtime plugins

Runtime plugins should target .NET Standard 2.0 to be compatible with all .NET versions supported by Reqnroll. Targetting a more specific version will limit the compatibility of your plugin.

Reqnroll searches for files that end with `.ReqnrollPlugin.dll` in the following locations:

- The folder containing your `Reqnroll.dll` file
- Your working directory

Reqnroll loads plugins in the order they are found in the folder.

Create a runtime plugin

You can create your `RuntimePlugin` in a separate project, or in the same project where your tests are.

Optional:

1. Create a new class library for your plugin.

Mandatory:

1. Add the Reqnroll NuGet package to your project.
2. Define a class that implements the `IRuntimePlugin` interface (defined in Reqnroll.Plugins).
3. Flag your assembly with the `RuntimePlugin` attribute for the plugin to be identified by Reqnroll plugin loader. The following example demonstrates a `MyNewPlugin` class that implements the `IRuntimePlugin` interface: `[assembly: RuntimePlugin(typeof(MyNewPlugin))]`
4. Implement the `Initialize` method of the `IRuntimePlugin` interface to access the `RuntimePluginEvents` and `RuntimePluginParameters`.

RuntimePluginsEvents

- `RegisterGlobalDependencies` - registers a new interface in the global container, see *Available Containers & Registrations*
- `CustomizeGlobalDependencies` - overrides registrations in the global container, see *Available Containers & Registrations*
- `ConfigurationDefaults` - adjust configuration values
- `CustomizeTestThreadDependencies` - overrides or registers a new interface in the test thread container, see *Available Containers & Registrations*

- `CustomizeFeatureDependencies` - overrides or registers a new interface in the feature container, see *Available Containers & Registrations*
- `CustomizeScenarioDependencies` - overrides or registers a new interface in the scenario container, see *Available Containers & Registrations*

Generator plugins

Runtime plugins should target .NET Standard 2.0 to be compatible with all scenarios supported by Reqnroll.

The generator plugins are invoked during build. They are usually invoked in a .NET environment according to your .NET SDK (e.g. .NET 8.0), but in some cases (when built using MSBuild or in Visual Studio) they might be invoked in a .NET 4.8 environment. Therefore, you have to make sure that your plugin works in both environments. If necessary, you can multi-target your plugin, but using the right compiled version of your plugin is the responsibility of the plugin itself.

The MSBuild task needs to know which generator plugins it should use. You therefore have to add your generator plugin to the `ReqnrollGeneratorPlugins` ItemGroup. This is passed to the MSBuild task as a parameter and later used to load the plugins.

Create a generator plugin

1. Create a new class library for your plugin.
2. Add the Reqnroll.CustomPlugin NuGet package to your project.
3. Define a class that implements the `IGeneratorPlugin` interface (defined in Reqnroll.Generator.Plugins namespace).
4. Flag your assembly with the `GeneratorPlugin` attribute for the plugin to be identified by Reqnroll plugin loader. The following example demonstrates a `MyNewPlugin` class that implements the `IGeneratorPlugin` interface: `[assembly: GeneratorPlugin(typeof(MyNewPlugin))]`
5. Implement the `Initialize` method of the `IGeneratorPlugin` interface to access `GeneratorPluginEvents` and `GeneratorPluginParameters` parameters.

GeneratorPluginsEvents

- `RegisterDependencies` - registers a new interface in the Generator container
- `CustomizeDependencies` - overrides registrations in the Generator container
- `ConfigurationDefaults` - adjust configuration values

Third-Party Dependencies

If your plugin uses third party assemblies, you need to make sure that the dependencies can be found.

Reqnroll will attempt to find your plugin's dependencies by

1. Loading assemblies from the same directory as your assembly.
2. Loading your plugin's .deps.json and loading the specified runtime assembly (e.g. from NuGet cache).

   Note: If you are using Nuget to publish your plugin, make sure your NuGet package contains the correct dependencies, otherwise the NuGet cache may be empty.
3. Check if the assembly is provided by Reqnroll itself (e.g. System.CodeDom).

   Note: The assemblies included by Reqnroll can change between versions, e.g. we switched to System.Text.Json at some point. So if you want to be on the safe side, do not rely on this approach.

Combined Package with both plugins

You can have a single NuGet package that contains both the runtime and generator plugins. We use this approach for the `Reqnroll.xUnit`, `Reqnroll.NUnit`, `Reqnroll.TUnit` and `Reqnroll.xUnit` packages for example.

The combined package can be built from a single project, or from two projects. The latter allows you to have different dependencies for the runtime and generator plugins.

If you use two projects for the combined package, the **NuSpec files should only be present in the generator project**. This is because the generators typically have more dependencies.

You can simply combine the contents of the `.targets` and `.props` file to a single one.

Tips & Tricks

Building Plugins on non-Windows machines

For building .NET 4.6.2 projects on non-Windows machines, the .NET Framework reference assemblies are needed.

You can add them with following PackageReference to your project:

```
<ItemGroup>
    <PackageReference Include="Microsoft.NETFramework.ReferenceAssemblies" Version="1.0.0
↪">
        <PrivateAssets>all</PrivateAssets>
        <IncludeAssets>runtime; build; native; contentfiles; analyzers</IncludeAssets>
    </PackageReference>
</ItemGroup>
```

1.8.3 Decorators

Reqnroll supports decorators which can be used in feature files. Decorators can be used to convert a tag in a feature file to an attribute in the generated code behind file.

Example decorator

Say we want to add an NUnit `Apartment` attribute to a test method in the generated code behind file (a file with extension `.feature.cs`) to specify that the test should be running in a particular apartment, either the STA or the MTA. For this, we can use a decorator which we need to register in a generator plugin so that the decorator can have its effect during the code behind file generation.

Steps to follow:

1. Create a Reqnroll project with test framework NUnit using the project template provided by the Reqnroll Visual Studio extension.
2. Create a GeneratorPlugin. You can follow the steps from the *Plugins* guide.
3. Create a Decorator (which is a class which implements interfaces like `ITestMethodTagDecorator`, `ITestMethodDecorator`, etc.):
    - `ITestMethodDecorator` is called always
    - `ITestMethodTagDecorator` is called only if the scenario has at least one tag

```
public class MyMethodTagDecorator : ITestMethodTagDecorator
{
    public static readonly string TAG_NAME = "myMethodTagDecorator";
    private readonly ITagFilterMatcher _tagFilterMatcher;
```

(continues on next page)

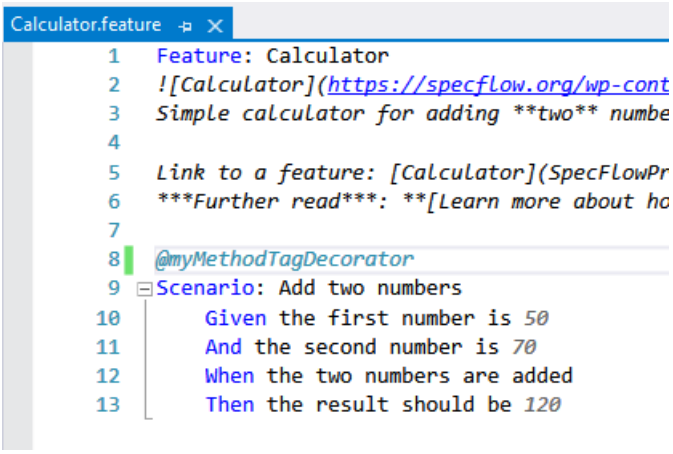
(continued from previous page)

```
    public MyMethodTagDecorator(ITagFilterMatcher tagFilterMatcher)
    {
        _tagFilterMatcher = tagFilterMatcher;
    }

    public bool CanDecorateFrom(string tagName, TestClassGenerationContext␣
↪generationContext, CodeMemberMethod testMethod)
    {
        return _tagFilterMatcher.Match(TAG_NAME, tagName);
    }

    public void DecorateFrom(string tagName, TestClassGenerationContext␣
↪generationContext, CodeMemberMethod testMethod)
    {
        var attribute = new CodeAttributeDeclaration(
            "NUnit.Framework.ApartmentAttribute",
            new CodeAttributeArgument(
                new CodeFieldReferenceExpression(
                    new CodeTypeReferenceExpression(typeof(System.Threading.
↪ApartmentState)),
                    "STA")));

        testMethod.CustomAttributes.Add(attribute);
    }

    public int Priority { get; }
    public bool RemoveProcessedTags { get; }
    public bool ApplyOtherDecoratorsForProcessedTags { get; }
}
```

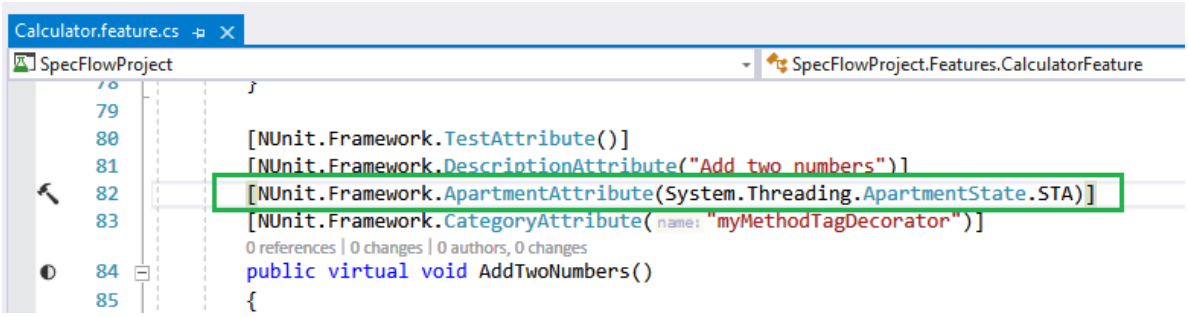
4. Register the Decorator in the `Initialize` method of the GeneratorPlugin:

```
public void Initialize(GeneratorPluginEvents generatorPluginEvents,␣
↪GeneratorPluginParameters generatorPluginParameters,
UnitTestProviderConfiguration unitTestProviderConfiguration)
{
    // Register the decorator
    generatorPluginEvents.RegisterDependencies += RegisterDependencies;
}

private void RegisterDependencies(object sender, RegisterDependenciesEventArgs␣
↪eventArgs)
{
    eventArgs.ObjectContainer.RegisterTypeAs<MyMethodTagDecorator,␣
↪ITestMethodTagDecorator>(MyMethodTagDecorator.TAG_NAME);
}
```

5. Install the GeneratorPlugin NuGet package to the Reqnroll project.
6. Add tag `@myMethodTagDecorator` to the feature file:



7. Build the solution
8. Check the generated code behind file (`.feature.cs`) if it contains the NUnit `Apartment` attribute:



Further read

- NUnit `Apartment` attribute: https://docs.nunit.org/articles/nunit/writing-tests/attributes/apartment.html
- Apartments: https://docs.microsoft.com/en-us/windows/win32/com/processes–threads–and-apartments

1.8.4 Available Containers

Global Container

The global container captures global services for test execution and the step definition, hook and transformation discovery result (i.e. what step definitions you have).

- IRuntimeConfigurationProvider
- ITestRunnerManager
- IStepFormatter
- ITestTracer
- ITraceListener
- ITraceListenerQueue
- IErrorProvider
- IRuntimeBindingSourceProcessor
- IRuntimeBindingRegistryBuilder

- IBindingRegistry
- IBindingFactory
- IStepDefinitionRegexCalculator
- IBindingInvoker
- IStepDefinitionSkeletonProvider
- ISkeletonTemplateProvider
- IStepTextAnalyzer
- IRuntimePluginLoader
- IBindingAssemblyLoader
- IBindingInstanceResolver
- RuntimePlugins
  - RegisterGlobalDependencies- Event
  - CustomizeGlobalDependencies- Event

Test Thread Container

> **Note**
>
> Parent Container is the Global Container

The test thread container captures the services and state for executing scenarios on a particular test thread. For parallel test execution, multiple test runner containers are created, one for each thread.

- ITestRunner
- IContextManager
- ITestExecutionEngine
- IStepArgumentTypeConverter
- IStepDefinitionMatchService
- ITraceListener
- ITestTracer
- RuntimePlugins
  - CustomizeTestThreadDependencies- Event

Feature Container

> **Note**
>
> Parent Container is the Test Thread Container

The feature container captures a feature's execution state. It is disposed after the feature is executed.

- FeatureContext (also available from the *test thread container* through `IContextManager`)

- RuntimePlugins
  - CustomizeFeatureDependencies- Event

Scenario Container

> **Note**
>
> Parent Container is the Feature Container

The scenario container captures the state of a scenario execution. It is disposed after the scenario is executed.

- (step definition classes)
- (dependencies of the step definition classes, aka context injection)
- ScenarioContext (also available from the *Test Thread Container* through `IContextManager`)
- RuntimePlugins
  - CustomizeScenarioDependencies- Event

1.8.5 Build Metadata Provider

This section covers the details of customizing Reqnroll's build metadata functionality for custom source control management (SCM) and build systems.

Overview

Reqnroll provides built-in support for extracting build metadata from various CI/CD environments including Azure Pipelines, TeamCity, Jenkins, GitHub Actions, GitLab CI, and many others. This metadata is captured through the `IBuildMetadataProvider` interface and stored in a `BuildMetadata` record.

When working with custom or unsupported build systems, you can create a runtime plugin that provides your own implementation of `IBuildMetadataProvider` to extract the relevant build information for your specific environment.

BuildMetadata Record Properties {#build-metadata-record}

The `BuildMetadata` record contains the following properties that capture essential build and source control information:

Core Properties

Property	Type	Description
`BuildUrl`	`string`	The URL to the build in the CI/CD system (e.g., link to build results page)
`BuildNumber`	`string`	The unique identifier or number assigned to the build by the CI/CD system
`Remote`	`string`	The URL of the source control repository (e.g., Git remote URL)
`Revision`	`string`	The specific commit hash, revision, or changeset identifier
`Branch`	`string`	The source control branch name from which the build was triggered
`Tag`	`string`	The source control tag name if the build was triggered from a tagged commit
`ProductName`	`string`	The name of the build server or CI/CD product (set automatically by the provider)

Property Details

BuildUrl: This should link directly to the build results page where developers can view build logs, artifacts, and status. Format varies by CI/CD system.

BuildNumber: Often an incrementing integer but can be any string format depending on your build system's numbering scheme.

Remote: Typically a Git repository URL, but can be any source control system URL. Should be in a format that allows cloning or accessing the repository.

Revision: For Git this would be the full commit SHA. For other SCM systems, use the equivalent unique identifier.

Branch: The branch name without any prefix (e.g., "main", "develop", "feature/new-functionality").

Tag: Only set if the build was triggered from a tagged commit. Should be the tag name without any prefix.

ProductName: Automatically set by the `GetBuildMetadata()` method to identify which build system provided the metadata.

Creating a Custom Build Metadata Provider {#custom-build-metadata-provider}

To support a custom SCM or build system, you need to create a Reqnroll runtime plugin that implements the `IBuildMetadataProvider` interface.

Step 1: Create the Plugin Project

Create a new .NET class library project that will contain your custom build metadata provider:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Reqnroll" Version="2.4.0" />
  </ItemGroup>
</Project>
```

Note: Target .NET Standard 2.0 for maximum compatibility with all .NET versions supported by Reqnroll.

Step 2: Implement IBuildMetadataProvider

Create your custom implementation of the `IBuildMetadataProvider` interface:

```
using Reqnroll.EnvironmentAccess;

namespace MyCompany.CustomScm.ReqnrollPlugin
{
    public class CustomBuildMetadataProvider : IBuildMetadataProvider
    {
        private readonly IEnvironmentWrapper _environment;

        public CustomBuildMetadataProvider(IEnvironmentWrapper environment)
        {
            _environment = environment;
        }
```

(continues on next page)

(continued from previous page)

```
        public BuildMetadata GetBuildMetadata()
        {
            // Check if we're running in your custom build environment
            var customBuildId = GetEnvironmentVariable("CUSTOM_BUILD_ID");
            if (string.IsNullOrEmpty(customBuildId))
                return null; // Not running in our custom environment

            // Extract metadata from your custom environment variables
            var buildUrl = GetEnvironmentVariable("CUSTOM_BUILD_URL");
            var buildNumber = GetEnvironmentVariable("CUSTOM_BUILD_NUMBER");
            var remote = GetEnvironmentVariable("CUSTOM_SCM_REMOTE");
            var revision = GetEnvironmentVariable("CUSTOM_SCM_REVISION");
            var branch = GetEnvironmentVariable("CUSTOM_SCM_BRANCH");
            var tag = GetEnvironmentVariable("CUSTOM_SCM_TAG");

            return new BuildMetadata(buildUrl, buildNumber, remote, revision, branch,␣
↪tag)
            {
                ProductName = "Custom Build System"
            };
        }

        private string GetEnvironmentVariable(string variable)
        {
            var result = _environment.GetEnvironmentVariable(variable);
            return result is ISuccess<string> success ? success.Result : null;
        }
    }
}
```

Step 3: Create the Runtime Plugin

Create a class that implements `IRuntimePlugin` to register your custom provider:

```
using Reqnroll.BoDi;
using Reqnroll.Plugins;
using Reqnroll.EnvironmentAccess;

[assembly: RuntimePlugin(typeof(MyCompany.CustomScm.ReqnrollPlugin.
↪CustomBuildMetadataPlugin))]

namespace MyCompany.CustomScm.ReqnrollPlugin
{
    public class CustomBuildMetadataPlugin : IRuntimePlugin
    {
        public void Initialize(RuntimePluginEvents runtimePluginEvents,
                               RuntimePluginParameters runtimePluginParameters,
                               UnitTestProviderConfiguration␣
↪unitTestProviderConfiguration)
        {
            // Register our custom implementation in the global container
```

(continues on next page)

(continued from previous page)

```
            runtimePluginEvents.CustomizeGlobalDependencies += (sender, args) =>
            {
                args.ObjectContainer.RegisterTypeAs<CustomBuildMetadataProvider,␣
↪IBuildMetadataProvider>();
            };
        }
    }
}
```

Step 4: Build and Deploy the Plugin

1. **Build your plugin project** to generate the assembly
2. **Name the output assembly** with the suffix `.ReqnrollPlugin.dll` (e.g., `MyCompany.CustomScm.ReqnrollPlugin.dll`)
3. **Deploy the plugin** by copying it to one of these locations:
   - The folder containing your `Reqnroll.dll` file
   - Your working directory

Reqnroll automatically discovers and loads plugins with the `.ReqnrollPlugin.dll` naming convention.

Step 5: Configure Environment Variables

Ensure your custom build system sets the appropriate environment variables that your provider expects:

```
# Example environment variables for your custom system
export CUSTOM_BUILD_ID="12345"
export CUSTOM_BUILD_URL="https://build.mycompany.com/builds/12345"
export CUSTOM_BUILD_NUMBER="1.2.3-build.12345"
export CUSTOM_SCM_REMOTE="https://scm.mycompany.com/repo/myproject.git"
export CUSTOM_SCM_REVISION="a1b2c3d4e5f67890123456789012345678901234567890abcd"
export CUSTOM_SCM_BRANCH="main"
export CUSTOM_SCM_TAG=""  # Empty if not a tag build
```

Advanced Scenarios

Supporting Multiple Build Systems

If you need to support multiple custom build systems, you can create a composite provider:

```
public class CompositeBuildMetadataProvider : IBuildMetadataProvider
{
    private readonly IEnvironmentWrapper _environment;
    private readonly IBuildMetadataProvider[] _providers;

    public CompositeBuildMetadataProvider(IEnvironmentWrapper environment)
    {
        _environment = environment;
        _providers = new IBuildMetadataProvider[]
        {
            new CustomSystemAProvider(environment),
```

(continues on next page)

(continued from previous page)

```
            new CustomSystemBProvider(environment),
            new LegacySystemProvider(environment)
        };
    }

    public BuildMetadata GetBuildMetadata()
    {
        foreach (var provider in _providers)
        {
            var metadata = provider.GetBuildMetadata();
            if (metadata != null)
                return metadata;
        }

        return null; // No custom system detected
    }
}
```

Fallback to Default Provider

To ensure compatibility with standard CI/CD systems while adding support for your custom system:

```
public class CustomWithFallbackBuildMetadataProvider : IBuildMetadataProvider
{
    private readonly IBuildMetadataProvider _customProvider;
    private readonly IBuildMetadataProvider _defaultProvider;

    public CustomWithFallbackBuildMetadataProvider(
        IEnvironmentWrapper environment,
        IEnvironmentInfoProvider environmentInfoProvider)
    {
        _customProvider = new CustomBuildMetadataProvider(environment);
        _defaultProvider = new BuildMetadataProvider(environmentInfoProvider,␣
↪environment);
    }

    public BuildMetadata GetBuildMetadata()
    {
        // Try custom provider first
        var customMetadata = _customProvider.GetBuildMetadata();
        if (customMetadata != null)
            return customMetadata;

        // Fallback to default provider for standard CI/CD systems
        return _defaultProvider.GetBuildMetadata();
    }
}
```

Handling Complex Environment Detection

For sophisticated environment detection logic:

```
public class SmartBuildMetadataProvider : IBuildMetadataProvider
{
    private readonly IEnvironmentWrapper _environment;

    public SmartBuildMetadataProvider(IEnvironmentWrapper environment)
    {
        _environment = environment;
    }

    public BuildMetadata GetBuildMetadata()
    {
        // Detect environment based on multiple indicators
        if (IsCustomSystemA())
            return GetCustomSystemAMetadata();

        if (IsCustomSystemB())
            return GetCustomSystemBMetadata();

        if (IsDockerEnvironment())
            return GetDockerEnvironmentMetadata();

        return null;
    }

    private bool IsCustomSystemA()
    {
        return !string.IsNullOrEmpty(GetVariable("CUSTOM_A_BUILD_ID")) &&
               !string.IsNullOrEmpty(GetVariable("CUSTOM_A_PROJECT"));
    }

    private bool IsCustomSystemB()
    {
        var buildTool = GetVariable("BUILD_TOOL");
        var buildId = GetVariable("BUILD_IDENTIFIER");
        return "CustomTool".Equals(buildTool, StringComparison.OrdinalIgnoreCase) &&
               !string.IsNullOrEmpty(buildId);
    }

    private bool IsDockerEnvironment()
    {
        return File.Exists("/.dockerenv") ||
               !string.IsNullOrEmpty(GetVariable("DOCKER_CONTAINER_ID"));
    }

    private string GetVariable(string name)
    {
        var result = _environment.GetEnvironmentVariable(name);
        return result is ISuccess<string> success ? success.Result : null;
    }
```

(continues on next page)

(continued from previous page)

```
    // Implementation methods for GetCustomSystemAMetadata(), etc.
}
```

Testing Your Custom Provider

Create unit tests to verify your custom provider works correctly:

```
[Test]
public void GetBuildMetadata_WithCustomEnvironmentVariables_ReturnsBuildMetadata()
{
    // Arrange
    var mockEnvironment = new Mock<IEnvironmentWrapper>();
    mockEnvironment.Setup(e => e.GetEnvironmentVariable("CUSTOM_BUILD_ID"))
                .Returns(new Success<string>("12345"));
    mockEnvironment.Setup(e => e.GetEnvironmentVariable("CUSTOM_BUILD_URL"))
                .Returns(new Success<string>("https://build.example.com/12345"));
    // ... setup other variables

    var provider = new CustomBuildMetadataProvider(mockEnvironment.Object);

    // Act
    var result = provider.GetBuildMetadata();

    // Assert
    Assert.That(result, Is.Not.Null);
    Assert.That(result.BuildNumber, Is.EqualTo("12345"));
    Assert.That(result.BuildUrl, Is.EqualTo("https://build.example.com/12345"));
    Assert.That(result.ProductName, Is.EqualTo("Custom Build System"));
}

[Test]
public void GetBuildMetadata_WithoutCustomEnvironment_ReturnsNull()
{
    // Arrange
    var mockEnvironment = new Mock<IEnvironmentWrapper>();
    mockEnvironment.Setup(e => e.GetEnvironmentVariable(It.IsAny<string>()))
                .Returns(new Success<string>(null));

    var provider = new CustomBuildMetadataProvider(mockEnvironment.Object);

    // Act
    var result = provider.GetBuildMetadata();

    // Assert
    Assert.That(result, Is.Null);
}
```

Container Registration Details

The plugin system uses Reqnroll's built-in BoDi dependency injection container. The key points for container registration:

Registration Methods

- `RegisterTypeAs<TImplementation, TInterface>()`: Registers a type to be instantiated when the interface is requested
- `RegisterInstanceAs<TInterface>(instance)`: Registers a pre-created instance
- `RegisterFactoryAs<TInterface>(factory)`: Registers a factory function

Container Hierarchy

Reqnroll uses a hierarchical container system:

- **Global Container**: For global services (where you register `IBuildMetadataProvider`)
- **Test Thread Container**: Per test thread
- **Feature Container**: Per feature execution
- **Scenario Container**: Per scenario execution

Registration Events

- `RegisterGlobalDependencies`: For new interface registrations
- `CustomizeGlobalDependencies`: For overriding existing registrations (recommended for `IBuildMetadataProvider`)

The `CustomizeGlobalDependencies` event is used because `IBuildMetadataProvider` is already registered by Reqnroll's default implementation, and you want to override it with your custom implementation.

Best Practices

1. **Environment Detection**: Always check if you're running in your target environment before returning metadata
2. **Error Handling**: Return `null` when your environment is not detected rather than throwing exceptions
3. **Fallback Support**: Consider supporting fallback to the default provider for standard CI/CD systems
4. **Testing**: Write comprehensive unit tests for your provider logic
5. **Documentation**: Document the environment variables your provider expects
6. **Versioning**: Use semantic versioning for your plugin to manage compatibility
7. **Performance**: Cache expensive operations if environment variable access is costly in your system

By following this guide, you can successfully extend Reqnroll to work with any custom SCM or build system while maintaining compatibility with existing functionality.

1.8.6 Custom Step Definition Attributes

Reqnroll provides the ability to create custom *step definition attributes*, enabling teams to express scenarios in local languages or to define grouped categories for step types beyond `Given`, `When`, and `Then`. This flexibility is commonly leveraged:

- To use attribute names and conventions in a localized (non-English) language, increasing naturalness for teams that work in other languages.
- To create broader or grouped step definition types (such as a general-purpose attribute that covers both "Given" and "When" semantics).

Key Requirements

To create a custom step definition attribute, the attribute class must:

- Inherit from `StepDefinitionBaseAttribute`.
- Derived constructors should use the same parameter names (expression, types) as the base class constructor.

Example: Custom Step Attribute

Below is a sample custom attribute that demonstrates the correct pattern. This example creates a `GivenWhenAttribute` attribute, which will match both `Given` and `When` steps.

```
using Reqnroll;
using System;

public class GivenWhenAttribute : StepDefinitionBaseAttribute
{
    readonly static StepDefinitionType[] types = [StepDefinitionType.Given,␣
↪StepDefinitionType.When];
    public GivenWhenAttribute(string expression) : base(expression, types) {
    }
}
```

Usage:

```
[Binding]
public class CalculatorSteps
{
    [GivenWhen("I add {int} and {int}")]
    public void MyAddStep(int x, int y)
    {
        // Step logic
    }
}
```

1.9 Integrations

This part contains details of the following topics.

1.9.1 Available Plugins

Below is a list of plugins for Reqnroll. Use the tag buttons to filter the list, or click "show all" to display every plugin again.

Important

Plugins marked with the official tag are maintained and verified by the Reqnroll team, and use the same open-source license as Reqnroll.

Plugins marked with the 3rd-party tag are maintained by third parties. **The Reqnroll team is not responsible for these plugins**. Please review each plugin's behavior and licensing terms before use, and submit issues to the plugin's own repository.

Name	Description	Tags	Download
Reqnroll.Autofac	Reqnroll plugin for using Autofac as a dependency injection framework for step definitions. *Read more...*	official di-container	
Reqnroll.Microsoft.Extensions.DependencyInjection	Reqnroll plugin for using Microsoft.Extensions.DependencyInjection as a dependency injection framework for step definitions. *Read more...*	official di-container	
Reqnroll.Windsor	Reqnroll plugin for using Castle Windsor as a dependency injection framework for step definitions. *Read more...*	official di-container	
Reqnroll.ExternalData	Package to use external data in Gherkin scenarios. Read more...	official	
Reqnroll.Verify	Reqnroll plugin for using Verify in scenarios. *Read more...*	official xunit	
Reqnroll.Assist.Dynamic	Reqnroll library for using dynamic types in bindings.	official	
DSL.Reqnroll	Reqnroll plugin that enables use of dynamic test data. Read more...	3rd-party	
Expressium.LivingDoc	LivingDoc Test Report plugin for Reqnroll Projects. Read more...	3rd-party formatter	

Note

If you would like your Reqnroll plugin to be listed here, please submit a pull request to update the documentation source page: `main/docs/integrations/available-plugins.md`.

1.9.2 Autofac

Introduction

Reqnroll plugin for using Autofac as a dependency injection framework for step definitions.

Note

Currently supports Autofac v4.0.0 or above

Step by step walkthrough of using Reqnroll.Autofac

1. Install plugin from NuGet into your Reqnroll project.

```
PM> Install-Package Reqnroll.Autofac
```

2. Create static methods somewhere in the Reqnroll project

Plugin supports both registration of dependencies globally and per scenario:

2.1 Optionally configure dependencies that need to be shared globally for all scenarios:

Create a static method somewhere in the Reqnroll project to register scenario dependencies: (Recommended to put it into the `Support` folder) that returns `void` and has one parameter of Autofac `ContainerBuilder`, tag it with the `[GlobalDependencies]` attribute.

When registering global dependencies, it is also a requirement to configure scenario dependencies as well in order to register classes marked with the `[Binding]` attribute as shown below.

Globally registered dependencies may be resolved in the `[BeforeTestRun]` and `[AfterTestRun]` methods.

2.2 Configure dependencies to be resolved each time for a scenario:

Create a static method somewhere in the Reqnroll project to register scenario dependencies: (Recommended to put it into the `Support` folder) that returns `void` and has one parameter of Autofac `ContainerBuilder`, tag it with the `[ScenarioDependencies]` attribute.

2.3 Configure your dependencies for the scenario execution within either the two methods `[GlobalDependencies]` and `[ScenarioDependencies]` or the single `[ScenarioDependencies]` method.

2.4 You also have to register the step definition classes in the `[ScenarioDependencies]` method, that you can do by either registering all public types from the Reqnroll project:

```
builder.RegisterAssemblyTypes(typeof(YourClassInTheReqnrollProject).Assembly).
↪SingleInstance();
```

2.5 or by registering all classes marked with the `[Binding]` attribute:

You may use a provided extension method to do this, but importing:

```
using Reqnroll.Autofac.ReqnrollPlugin;
```

Then

```
containerBuilder.AddReqnrollBindings<AnyClassInTheReqnrollProject>()
```

Or overload

```
containerBuilder.AddReqnrollBindings(Assembly.GetExecutingAssembly())
```

Or manually register like so:

```
builder
  .RegisterAssemblyTypes(typeof(AnyClassInTheReqnrollProject).Assembly)
  .Where(t => Attribute.IsDefined(t, typeof(BindingAttribute)))
  .SingleInstance();
```

3. A typical dependency builder method for `[GlobalDependencies]` with `[ScenarioDependencies]` probably looks like this:

```
public class SetupTestDependencies
{
  [GlobalDependencies]
  public static void SetupGlobalContainer(ContainerBuilder containerBuilder)
```

(continues on next page)

(continued from previous page)

```
  {
    // Register globally scoped runtime dependencies
    containerBuilder
      .RegisterType<MyGlobalService>()
      .As<IMyGlobalService>()
      .SingleInstance();
  }

  [ScenarioDependencies]
  public static void SetupScenarioContainer(ContainerBuilder containerBuilder)
  {
    // Register scenario scoped runtime dependencies
    containerBuilder
      .RegisterType<MyService>()
      .As<IMyService>()
      .SingleInstance();

    // register binding classes
    containerBuilder.AddReqnrollBindings<SetupTestDependencies>();
  }
}
```

4. It is also possible to continue to use the legacy method as well, however this method is not compatible with global dependency registration and can only be used on its own like so:

Create a static method somewhere in the Reqnroll project to register scenario dependencies: (Recommended to put it into the `Support` folder) that returns an Autofac `ContainerBuilder` and tag it with the `[ScenarioDependencies]` attribute.

5. If you have an existing container, built and owned by your application under test, you can use that instead of letting Reqnroll manage your container

Create a static method in your Reqnroll project to return a lifetime scope from your container. Note that Reqnroll creates a second scope under yours, so be sure to pair this use-case with the `CreateContainerBuilder` method above to add your step bindings.

```
[FeatureDependencies]
public static ILifetimeScope GetFeatureLifetimeScope()
{
    // TODO: Add any top-level dependencies here, though note that usually step bindings
        //       should be declared in the Configure method below, as this will ensure␣
↪they
        //       are in the correct scope to inject ScenarioContext etc.
    return containerScope.BeginLifetimeScope();
}

[ScenarioDependencies]
public static void ConfigureContainerBuilder(ContainerBuilder containerBuilder)
{
    //TODO: add customizations, stubs required for testing
    containerBuilder.AddReqnrollBindings<SetupTestDependencies>();
}
```

1.9.3 Microsoft.Extensions.DependencyInjection

Introduction

Reqnroll plugin for using Microsoft.Extensions.DependencyInjection as a dependency injection framework for step definitions.

> **Note**
>
> Currently supports Microsoft.Extensions.DependencyInjection v6.0.0 or above

Step by step walkthrough of using Reqnroll.Microsoft.Extensions.DependencyInjection

1. Install plugin from NuGet into your Reqnroll project.

```
PM> Install-Package Reqnroll.Microsoft.Extensions.DependencyInjection
```

2. Create static methods somewhere in the Reqnroll project

Create a static method in your SpecFlow project that returns a Microsoft.Extensions.DependencyInjection.IServiceCollection and tag it with the [ScenarioDependencies] attribute. Configure your test dependencies for the scenario execution within this method. Step definition classes (i.e. classes with the SpecFlow [Binding] attribute) are automatically added to the service collection.

3. A typical dependency builder method looks like this:

```
public class SetupTestDependencies
{
  [ScenarioDependencies]
  public static IServiceCollection CreateServices()
  {
    var services = new ServiceCollection();

    // TODO: add your test dependencies here
    services.AddSingleton<IMyService, MyService>();

    return services;
  }
}
```

1.9.4 External Data Plugin

You can easily apply standardized test cases across a wide range of features to significantly reduce redundant data for large test suites. By reusing execution flows, you can also speed up exploratory and approval testing for ranges of examples. Reqnroll makes all of this possible by introducing support for loading external data into scenarios easily.

The Reqnroll ExternalData plugin lets teams separate test data from test scenarios, and reuse examples across a large set of scenarios. This is particularly helpful when a common set of examples needs to be consistently verified in different scenarios.

Simply download the NuGet package and add it to your reqnroll projects to use it.

Supported Data Sources

- CSV files (format 'CSV', extension .csv)

> **Note**
>
> Standard RFC 4180 CSV format is supported with a header line (plugin uses CsvHelper to parse the files).

- Excel files (format Excel, extensions .xlsx, .xls, .xlsb)

> **Note**
>
> Both XLSX and XLS is supported (plugin uses ExcelDataReader to parse the files).

- JSON files (format 'JSON', extension .json)

> **Note**
>
> Object arrays and nested object arrays are supported (plugin uses JObject.Parse to parse the files).

Tags

The following tags can be used to specify the external source:

- `@DataSource:path-to-file` - This tag is the main tag that you can add to a scenario or a scenario outline to specify the data source you wish to use.

> **Caution**
>
> The path is a relative path to the folder of the **feature files**.

- `@DisableDataSource` - The `@DataSource` tag can be added to the feature node, turning all scenarios in the file to scenario outlines. This method is useful when the entire feature file uses the same data source. Use the `@DisableDataSource` If you want a select few scenarios in the feature file to **not** use the data source tagged at feature node level.
- `@DataFormat:format` - This tag only needs to be used if the format cannot be identified from the file extension.
- `@DataSet:data-set-name` - This tag is applicable to *Excel and Json files only*. For Excel it is used to select the worksheet of the Excel file you wish to use. By **default**, the first worksheet in an Excel file is targeted. For Json it is used to select the object array you wish to use. By **default**, the first object array in a Json file is targeted.
- `@DataField:name-in-feature-file=name-in-source-file` - This tag can be used to "rename" columns of the external data source.

General notes on tags:

- Tags can be added on feature, scenario, scenario outline or scenario outline examples.
- Tags can inherit from the feature node, but you can override them with another tag or disable them by using the `@DisableDataSource` tag on the scenario level.
- As tags cannot contain spaces, generally the underscore (_) character can be used to represent a space. It is currently not supported to access a file that contains spaces in the file name or in the relative path.

Examples

CSV files

The below examples all use the same ***products.csv*** file. The file contains three products and their corresponding prices:

	A
1	product,price
2	Chocolate,2.5
3	Apple,1.0
4	Orange,1.2

- This scenario will be treated as a scenario outline with the products from the CSV file replacing the **<product>** parameter in the given statement:

```
@DataSource:products.csv
Scenario: Valid product prices are calculated
        Given the customer has put 1 piece of <product> in the basket
        When the basket price is calculated
        Then the basket price should be greater than zero
```

- This scenario will be treated as a scenario outline similar to the above example but uses both **<product>** and **<price>** from the CSV file:

```
@DataSource:products.csv
Scenario: The basket price is calculated correctly
        Given the price of <product> is €<price>
        And the customer has put 1 pcs of <product> to the basket
        When the basket price is calculated
        Then the basket price should be €<price>
```

- This scenario shows how you can extend the product list using the example table with the ones from the CSV file. A total of 4 products will be added here, 3 from the CSV file plus "Cheesecake" from the example table:

```
@DataSource:products.csv
Scenario Outline: Valid product prices are calculated (Outline)
        Given the customer has put 1 pcs of <product> to the basket
        When the basket price is calculated
        Then the basket price should be greater than zero
Examples:
        | product    |
        | Cheesecake |
```

You may also add the `@DataSource` above the example table if you wish to:

```
Scenario Outline: Valid product prices are calculated (Outline, example annotation)
        Given the customer has put 1 pcs of <product> to the basket
        When the basket price is calculated
        Then the basket price should be greater than zero
@DataSource:products.csv
Examples:
        | product    |
        | Cheesecake |
```

- In this scenario the parameters names do not match the column names in the CSV file but we can address that by using the `@DataField:product-name=product` and `@DataField:price-in-EUR=price` tags:

```
@DataSource:products.csv @DataField:product-name=product @DataField:price-in-EUR=price
Scenario: The basket price is calculated correctly (renamed fields)
        Given the price of <product-name> is €<price-in-EUR>
        And the customer has put 1 piece of <product-name> in the basket
        When the basket price is calculated
        Then the basket price should be €<price-in-EUR>
```

- This scenario is similar to the above scenario with the renaming of the parameters, but the difference is the use of space in the parameter name. Spaces are **not** supported and must be replaced with underscore (_):

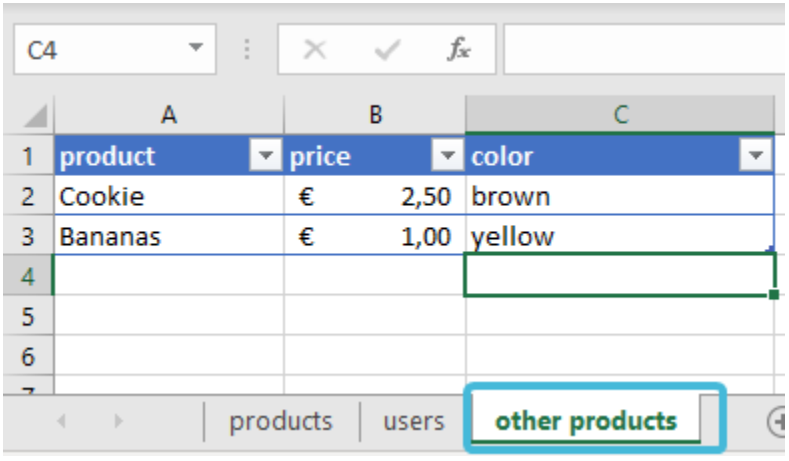
```
@DataSource:products.csv @DataField:product_name=product @DataField:price-in-EUR=price
Scenario: The basket price is calculated correctly

        Given the customer has put 1 piece of <product name> in the basket
        When the basket price is calculated
        Then the basket price should be greater than zero
Examples:
        | product name |
        | Cheesecake   |
```

Excel files

You can use Excel files the same way as you do with CSV files with some minor differences:

- Only simple worksheets are supported, where the **header is in the first row** and the data comes right below that. Excel files that contain tables, graphics, etc. are not supported.
- Excel files with multiple worksheets are supported, you can use the `@DataSet:sheet-name` to select the worksheets you wish to target. The plugin uses the **first** worksheet by **default**.
- Use underscores in the `@DataSet` tag instead of spaces if the worksheet name contains spaces.

The below example shows an Excel file with multiple worksheets and we wish to target the last worksheet labelled *"other products"*. We do this by using the `@DataSet:other_products` tag. Note the use of (_) instead of space:




```
@DataSource:products.xlsx @DataSet:other_products
Scenario: The basket price is calculated correctly for other products
```

(continues on next page)

(continued from previous page)

```
        Given the price of <product> is €<price>
        And the customer has put 1 piece of <product> in the basket
        When the basket price is calculated
        Then the basket price should be €<price>
```

Language Settings

The decimal and date values read from an Excel file will be exported using the language of the feature file (specified using the `#language` setting in the feature file or in the Reqnroll configuration file). This setting affects for example the decimal operator as in some countries comma (,) is used as decimal separator instead of dot (.). To specify not only the language but also the country use the `#language: language-country` tag, e.g. *#language: de-AT* for *Deutsch-Austria*.

Example: Hungarian uses comma (,) as decimal separator instead of dot (.), so Reqnroll will expect the prices in format 1,23:

This sample shows that the language settings are applied for the data that is being read by the external data plugin.

```
#language: hu-HU
Jellemző: External Data from Excel file (Hungarian)

@DataSource:products.xlsx
Forgatókönyv: The basket price is calculated correctly
        Amennyiben the price of <product> is €<price>
        És the customer has put 1 pcs of <product> to the basket
        Amikor the basket price is calculated
        Akkor the basket price should be €<price>
```

Json files

You can use Json files the same way as you do with Excel files with some minor differences:

- Only Object arrays and nested Object are supported. Arrays of simple types, empty arrays, etc. are not supported.
- Json files with multiple object arrays are supported, you can use the `@DataSet:array-property-name` to select the object array you wish to target. The plugin uses the **first** object array by **default**. Nested object arrays can be specified by appending property names using a '.'.
- Use underscores in the `@DataSet` tag instead of spaces if the array property name name contains spaces.

The below example shows a Json file with object arrays and we wish to target the last array labelled *"other products"*. We do this by using the `@DataSet:other_products` tag. Note the use of (_) instead of space:

```
    "other products":
    [
        {
            "product": "Cookie",
            "price": "2.5",
            "color": "brown"
        },
        {
            "product": "Bananas",
            "price": "1.0",
            "color": "yellow"
        }
     ]
```

```
@DataSource:products.json @DataSet:other_products
Scenario: The basket price is calculated correctly for other products
       Given the price of <product> is €<price>
       And the customer has put 1 pcs of <product> to the basket
       When the basket price is calculated
       Then the basket price should be €<price>
```

The below example shows a Json file with nested object arrays and we wish to target the inner array labelled *"varieties"*. We do this by using the `@DataSet:products.varieties` tag. Note the use of . to append nested property names:

```
"products":
    [
        {
            "product": "Chocolate",
            "color": "brown",
            "total price": "3.15",
            "varieties":
            [
                {
                    "name": "Dark Chocolate",
                    "price": "1.6"
                },
                {
                    "name": "Milk Chocolate",
                    "price": "1.55"
                }
            ]
        },
```

```
@DataSource:products-nested-dataset.json @DataSet:products.varieties
Scenario: The basket price is calculated correctly for products.varieties in nested␣
↪products json
       Given the price of <product> is €<price>
       And the customer has put 1 pcs of <product> to the basket
       When the basket price is calculated
```

(continues on next page)

(continued from previous page)

```
        Then the basket price should be €<price>
```

1.9.5 F# Support

Bindings for Reqnroll can be written also in F#. Doing so you can take the advantages of the F# language for writing step definitions: you can define regex-named F# functions for your steps. Simply put the regex between double backticks.

```
let [<Given>] ``I have entered (.*) into the calculator``(number:int) =
    Calculator.Push(number)
```

Although the regex method names are only important for *step definitions* you can also define *hooks* and *step argument conversions* in the F# binding projects.

Note: You need to create a C# or VB project for hosting the feature files and configure your F# project(s) as *external binding assemblies*:

Listing 194: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",

  "bindingAssemblies": [
    {
      "assembly": "MyFSharpBindings"
    }
  ]
}
```

1.9.6 MSTest

Reqnroll supports MsTest V2 or later (NuGet Version 2.2.8 or higher).

> **Note**
>
> MsTest V4 is supported from Reqnroll v3.2 onwards.

Documentation for MSTest can be found here.

Needed NuGet Packages

For Reqnroll: Reqnroll.MSTest

For MSTest: MSTest.TestFramework

For Test Discovery & Execution:

- MSTest.TestAdapter
- Microsoft.NET.Test.Sdk

Accessing TestContext

You can access the MsTest TestContext instance in your step definition or hook classes by constructor injection:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

[Binding]
public class MyStepDefs
{
    private readonly TestContext _testContext;
    public MyStepDefs(TestContext testContext) // use it as ctor parameter
    {
        _testContext = testContext;
    }

    [Given("a step")]
    public void GivenAStep()
    {
        //you can access the TestContext injected in the ctor
        _testContext.WriteLine(_testContext.TestRunDirectory);
    }


    [BeforeScenario()]
    public void BeforeScenario()
    {
        //you can access the TestContext injected in the ctor
        _testContext.WriteLine(_testContext.TestRunDirectory);
    }
}
```

In the static BeforeTestRun/AfterTestRun hooks you can use parameter injection:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

[Binding]
public class Hooks
{
    [BeforeTestRun]
    public static void BeforeTestRun(TestContext testContext)
    {
        //you can access the TestContext injected as parameter
        testContext.WriteLine(testContext.TestRunDirectory);
    }

    [AfterTestRun]
    public static void AfterTestRun(TestContext testContext)
    {
        //you can access the TestContext injected as parameter
        testContext.WriteLine(testContext.DeploymentDirectory);
    }
}
```

Tags on Examples - Workaround

The MsTest Generator MsTest does not support applying tags (categories) to specific entries of parameterized tests, see issues 4089 and issues 1043

In short, tags on Examples are *not* send to the test execution. So `@Test` and `@Acceptance` are not available for test filtering/reporting/etc.

```
Scenario: Sample Scenario
   Given sample step

@Test
Examples:
| User   |
| Tester |

@Acceptance
Examples:
| User   |
| Acc    |
```

The workaround for now is to disable the *row tests*. Note that this does impact how tests names are displayed:

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",
  // add the line below
  "generator": {"allowRowTests" :  false},

  "bindingAssemblies": [
  ]
}
```

Tags for TestClass Attributes

The MsTest Generator can generate test class attributes from tags specified on a **feature**.

Owner

Tag:

```
@Owner:John
```

Output:

```
[Microsoft.VisualStudio.TestTools.UnitTesting.OwnerAttribute("John")]
```

Priority

Tag:

```
@Priority:1
```

Output:

```
[Microsoft.VisualStudio.TestTools.UnitTesting.PriorityAttribute(1)]
```

Remarks:

The attribute is generated only when the value is a valid integer (valid means supported by `int.TryParse`)

WorkItem

Tag:

```
@WorkItem:123
```

Output:

```
[Microsoft.VisualStudio.TestTools.UnitTesting.WorkItemAttribute(123)]
```

DeploymentItem

Example 1 : Copy a file to the same directory as the deployed test assemblies

Tag:

```
@MsTest:DeploymentItem:test.txt
```

Output:

```
[Microsoft.VisualStudio.TestTools.UnitTesting.DeploymentItemAttribute("test.txt")]
```

Example 2 : Copy a file to a sub-directory relative to the deployment directory

Tag:

```
@MsTest:DeploymentItem:Resources\DeploymentItemTestFile.txt:Data
```

Output:

```
[Microsoft.VisualStudio.TestTools.UnitTesting.DeploymentItemAttribute("Resources\\
↪DeploymentItemTestFile.txt", "Data")]
```

1.9.7 NUnit

Reqnroll supports NUnit 3.13.1 or later.

Documentation for NUnit can be found here.

Needed NuGet Packages

For Reqnroll: Reqnroll.NUnit

For NUnit: NUnit

For Test Discovery & Execution:

- NUnit3TestAdapter
- Microsoft.NET.Test.Sdk

1.9.8 TUnit

Reqnroll supports TUnit 1.3.25 or later.

Documentation for TUnit can be found here.

Supported .NET Versions

TUnit with Reqnroll supports the following .NET versions:

- .NET 8.0 or later
- .NET Framework 4.6.2 or later

Needed NuGet Packages

For Reqnroll: Reqnroll.TUnit

For TUnit: TUnit

Access TestContext

The TUnit test context (`TUnit.Core.TestContext`) is registered in the scenario dependency scope. You can get access to it via *Context-Injection* when needed.

Parallel Execution

TUnit supports test-level (scenario-level) parallel test execution by default. The parallel execution can be disabled for the entire test project using the `[assembly: TUnit.Core.NotInParallel]` attribute or use *Excluding Reqnroll features from parallel execution*.

.NET 10 SDK Compatibility

TUnit uses Microsoft.Testing.Platform which dropped VSTest support in .NET 10 SDK. If you encounter the error "Testing with VSTest target is no longer supported by Microsoft.Testing.Platform on .NET 10 SDK and later", you need to enable the new dotnet test experience by adding the following property to your project file:

```
<PropertyGroup>
  <TestingPlatformDotnetTestSupport>true</TestingPlatformDotnetTestSupport>
</PropertyGroup>
```

For more information, see Microsoft's documentation on the Microsoft.Testing.Platform.

1.9.9 Castle Windsor

Introduction

Reqnroll plugin for using Castle Windsor as a dependency injection framework for step definitions.

> **Note**
>
> Currently supports Castle Windsor v5.0.1 or above

Step by step walkthrough of using Reqnroll.Windsor

1. Install plugin

- Install plugin from NuGet into your Reqnroll project.

```
PM> Install-Package Reqnroll.Windsor
```

2. Create static method

- Create a static method somewhere in the Reqnroll project

(Recommended to put it into the `Support` folder) that returns a Windsor `IWindsorContainer` and tag it with the `[ScenarioDependencies]` attribute.

- Configure your dependencies for the scenario execution within the method.

- All your binding classes are automatically registered, including ScenarioContext etc.

3. Sample dependency builder method

- A typical dependency builder method probably looks like this:

```
[ScenarioDependencies]
public static IWindsorContainer CreateContainer()
{
  var container = new WindsorContainer();

  //TODO: add customizations, stubs required for testing

  return container;
}
```

4. Reusing a container

- To re-use a container between scenarios, try the following:

Your shared services will be resolved from the root container, while scoped objects such as ScenarioContext will be resolved from the new container.

```
[ScenarioDependencies]
public static IWindsorContainer CreateContainer()
{
  var container = new WindsorContainer();
  container.Parent = sharedRootContainer;

  return container;
}
```

5. Customize binding behavior

- To customize binding behavior, use the following:

Default behavior is to auto-register bindings. To manually register these during `CreateContainer` you can use the following attribute:

```
[ScenarioDependencies(AutoRegisterBindings = false)]
public static IWindsorContainer CreateContainer()
{
    // Register your bindings here
}
```

1.9.10 xUnit

Reqnroll supports both xUnit v2 and xUnit v3.

- **xUnit v2**: Supported from xUnit v2 2.8 or later
- **xUnit v3**: Supported since Reqnroll 3.1

Documentation for xUnit can be found here.

xUnit v2

Needed NuGet Packages

For Reqnroll: Reqnroll.xUnit

For xUnit: xUnit

For Test Discovery & Execution:

- xunit.runner.visualstudio
- Microsoft.NET.Test.Sdk

xUnit v3

Needed NuGet Packages

For Reqnroll: Reqnroll.xunit.v3

For xUnit: xunit.v3

For Test Discovery & Execution:

- xunit.runner.visualstudio (version 3.0.2 or later)
- Microsoft.NET.Test.Sdk

Access ITestOutputHelper

The xUnit `ITestOutputHelper` is registered in the ScenarioContainer. You can get access to simply via getting it via *Context-Injection*.

Migrating from SpecFlow

Reqnroll generates Task based async code, which is different from the SpecFlow generated synchronous code. This change has a big effect on how xUnit runs the Tests.

xUnit since Version 2.8 has two modes for running tests in parallel: `conservative` (new and default since 2.8) and `aggressive` the default (older and default before 2.8). For more details about both algorithms and their configuration, see the xUnit Running Tests in Parallel. Reqnroll.xUnit also supports both modes since Version 2.4.0; on older versions, only the aggressive mode was possible (even with xUnit 2.8 or higher).

TLDR is:

- Conservative mode starts just as many tests, which are configured in max parallel threads.

- Aggressive mode starts all tests, and lets the Task Scheduler handle the maximum parallel threads.

Because of the async nature of Reqnroll generated test code, the aggressive mode together with resource intensive test (i.e. Browser-based tests) can lead to the impression that more tests are run in parallel then configured (more Browsers opened than expected). This is normally a wanted async/await behavior. To fully utilize the available resources.

If you have resource-intensive tests, use the default conservative mode and limit the parallel tests in xUnit. But you can opt in the aggressive mode if your tests are lighter on the resources, to optimize the time to finish the tests.

We recommend sticking to the default conservative mode unless you have good reason to use the aggressive mode.

Example

```
using System;
using Reqnroll;

[Binding]
public class BindingClass
{
    private Xunit.Abstractions.ITestOutputHelper _testOutputHelper;
    public BindingClass(Xunit.Abstractions.ITestOutputHelper testOutputHelper)
    {
        _testOutputHelper = testOutputHelper;
    }

    [When(@"I do something")]
    public void WhenIDoSomething()
    {
        _testOutputHelper.WriteLine("EB7C1291-2C44-417F-ABB7-A5154843BC7B");
    }
}
```

1.9.11 Verify

Reqnroll supports Verify.Xunit 24.2.0 or later.

Documentation for Verify can be found here.

Needed NuGet Packages

- Reqnroll.xUnit and its *dependencies*
- Reqnroll.Verify

How it works

This plugin adds a VerifySettings instance to Reqnroll's scenario container, which can be used to set the correct path for the tests' verified files.

Example

```
Feature: Verify feature

    Scenario: Verify scenario
        When I calculate 1 + 2
        Then I expect the result is correct
```

```
[Binding]
internal class StepDefinitions
{
    private readonly VerifySettings _settings;
    private int _result;

    public StepDefinitions(VerifySettings settings)
    {
        _settings = settings;
    }

    [When("I calculate (\d+) + (\d+)")]
    public void WhenICalculate(int v1, int v2)
    {
        _result = v1 + v2; // simulate calling the SUT to get the result
    }

    [Then("I expect the result is correct")]
    public void ThenIExpectTheResultIsCorrect()
    {
        Verifier.Verify(_result, _settings);
    }
}
```

Legacy global VerifySettings support

For Verify versions prior to 29.0.0, the plugin also supported a legacy mode that uses a global `VerifySettings` instance (i.e. calling `Verifier.Verify()` without specifying the settings). This mode is not thread-safe and should only be used in single-threaded test execution.

From plugin version v3.1 this legacy support has been removed. It is recommended to always inject the `VerifySettings` instance into the step definition class like in the example above.

If this is not possible, the following workaround can be used to still support the legacy mode. The workaround works only in single-threaded test execution.

```
namespace Reqnroll.Verify.ReqnrollPlugin;

[Binding]
public class VerifyHooks
{
    [BeforeTestRun]
    public static void EnableGlobalVerifySettingsForCompatibility()
    {
        Verifier.DerivePathInfo(
            (_, projectDirectory, _, _) =>
            {
#pragma warning disable CS0618 // Type or member is obsolete
                var scenarioContext = Reqnroll.ScenarioContext.Current;
                var featureContext = Reqnroll.FeatureContext.Current;
#pragma warning restore CS0618 // Type or member is obsolete
                string scenarioInfoTitle = scenarioContext.ScenarioInfo.Title;

                foreach (System.Collections.DictionaryEntry scenarioInfoArgument in␣
```

(continues on next page)

(continued from previous page)

```
↪scenarioContext.ScenarioInfo.Arguments)
                {
                    scenarioInfoTitle += "_" + scenarioInfoArgument.Value;
                }

                return new PathInfo(
                    Path.Combine(projectDirectory, featureContext.FeatureInfo.
↪FolderPath),
                    featureContext.FeatureInfo.Title,
                    scenarioInfoTitle);
            });
    }
}
```

1.10 IDE integrations

This part contains details of the following topics.

1.10.1 Reqnroll for Visual Studio

This documentation covers the Visual Studio extension for Reqnroll. The extension supports Visual Studio 2022 and Visual Studio 2026.

- *Feature Overview* - Overview of the Visual Studio extension features
- *Installation* - How to install the Reqnroll Visual Studio extension
- *Editing Features* - Syntax highlighting, IntelliSense, and editing features
- *Navigation Features* - Navigate between steps and step definitions
- *Defining Steps* - Generate step definition code snippets
- *Visual Studio Extension Settings* - Configure the Visual Studio extension settings
- *Gherkin Formatting Settings with EditorConfig* - Define consistent Gherkin formatting styles with EditorConfig

Feature Overview

The Visual Studio extension includes a number of features that make it easier to edit Gherkin files and navigate to and from step definitions in Visual Studio. You can also generate code snippets for step definition methods from feature files.

You can install the extension from the Visual Studio Gallery (Marketplace) or directly in Visual Studio. Detailed instructions can be found *here*.

The extension provides the following features:

- *Editing Feature Files*
  - *Gherkin syntax highlighting* in feature files, highlighting unbound steps and parameters
  - *IntelliSense* (auto-completion) for keywords and steps
  - *Outlining* (folding) sections of the feature file
  - *Comment/uncomment* feature file lines
  - Automatic Gherkin *table formatting*

  - *Document formatting*
  - *Renaming steps*
- *Navigation*
  - Navigate from *steps in scenarios to step definitions and vice versa*
  - Navigate from a *scenario to hook methods*
  - Find Unused step definitions (those not yet bound to a step)
  - Detects step definitions within the Reqnroll project to enable navigation features and indication of defined status of a step
- Other
  - *Generate code snippets for step definition methods* from feature files
  - *Configurable settings*
  - *Configurable Gherkin formatting settings with EditorConfig*

Editing Features

The Visual Studio extension includes the following features to make it easier to edit feature files and identify which steps have already been bound.

> **Note**
>
> Many of the formatting behaviors can be controlled by an *EditorConfig file* and the `ide` section of the *`reqnroll.json` config file*.

Gherkin Syntax Highlighting

Various default styles have been defined for the Gherkin syntax. You can customize these colors in Visual Studio's settings (**Tools | Options | Environment | Fonts and Colors**). The names of the corresponding **Display items** in the list begin with "Reqnroll".

In addition to highlighting keywords, comments, tags etc., unbound steps and parameters in feature files are highlighted when editing the file in Visual Studio. The following syntax highlighting is used by default:

- Purple: unbound steps
- Black: bound steps
- Red: parameters in bound steps

You can thus tell immediately which steps in a feature file have been bound.

> **Note**
>
> A project must be **built** for syntax highlighting to update.

IntelliSense (auto-completion) for Keywords and Steps

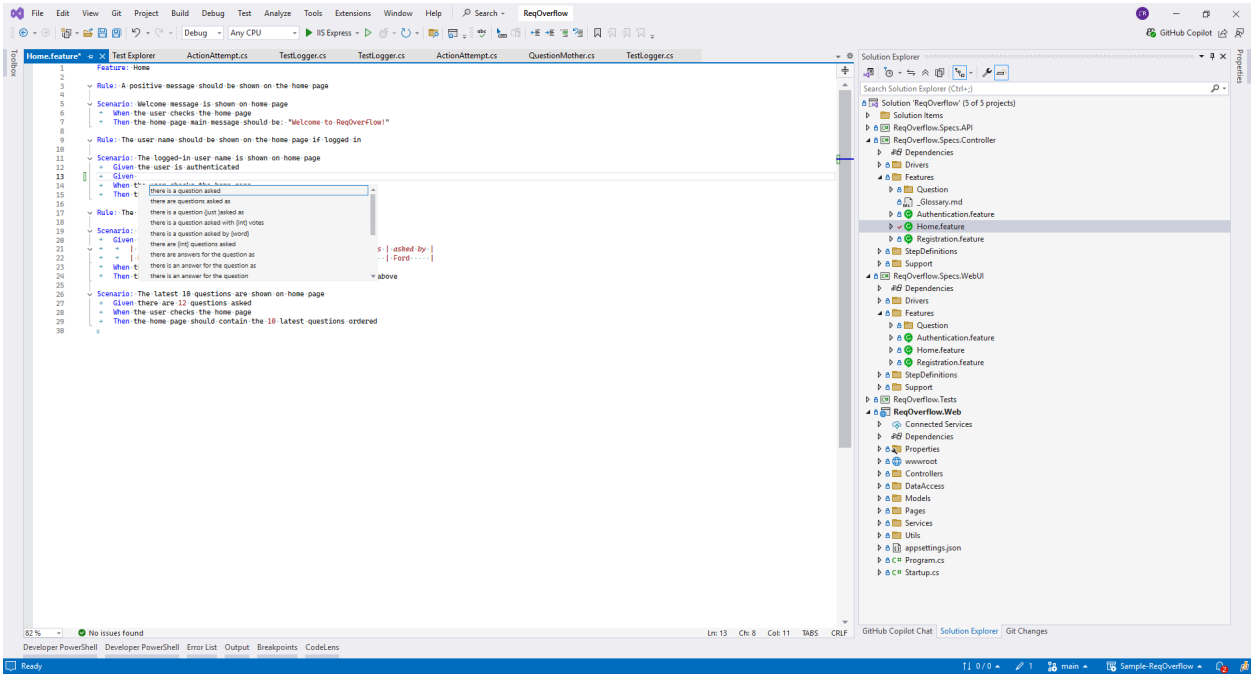
IntelliSense makes Reqnroll easy to use when integrated with Visual Studio. IntelliSense uses find-as-you-type to restrict the list of suggested entries.

Gherkin Files

IntelliSense is available in feature files for the following:

- Gherkin keywords (e.g. `Scenario`, `Given` etc.)
- Existing steps are listed after a `Given`, `When` or `Then` statement, providing quick access to your current steps definitions.

> **Note**
>
> Note that all the step definitions that match the current type (Given, When, Then) are displayed initially, but the list can be filtered by typing additional keywords.



Outlining and Comments in Feature Files

Most of the items in the **Edit** menu work well with Reqnroll feature files, for example:

- You can comment and uncomment selected lines (# character) with the default shortcut for comments (Ctrl+K Ctrl+C/Ctrl+K Ctrl+U) or from the menu
- You can use the options in the **Edit | Outlining** menu to expand and contract sections of your feature files

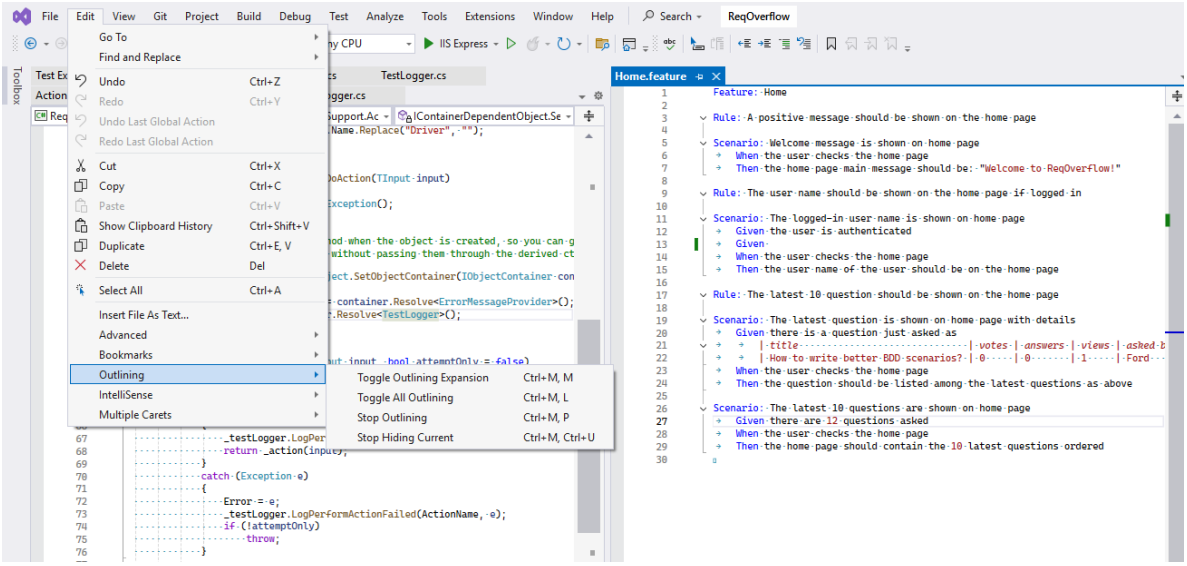


Table Formatting

Tables in Reqnroll are also expanded and formatted automatically as you enter column names and values:

Fig. 1: Formatted table

Document Formatting

Document formatting is also available. It automatically re-indents code and fixes blank lines, comments, etc.

You can find this option under *Edit->Advanced->Format document* or alternatively use the Ctrl+K, Ctrl+D shortcut:

Below is a feature file document which is not indented correctly:

After the `Format Document` command:

> **Note**
>
> The formatting behavior can be controlled by an *EditorConfig file* and the `ide` section of the *`reqnroll.json` config file*.

Renaming Steps

You can globally rename steps and update the associated step definitions automatically. To do so:

1. Open the feature file containing the step.
2. Right-click on the step you want to rename and select Rename from the context menu.
3. Enter the new text for the step in the dialog and confirm with OK.
4. Your step definitions and all feature files containing the step are updated.

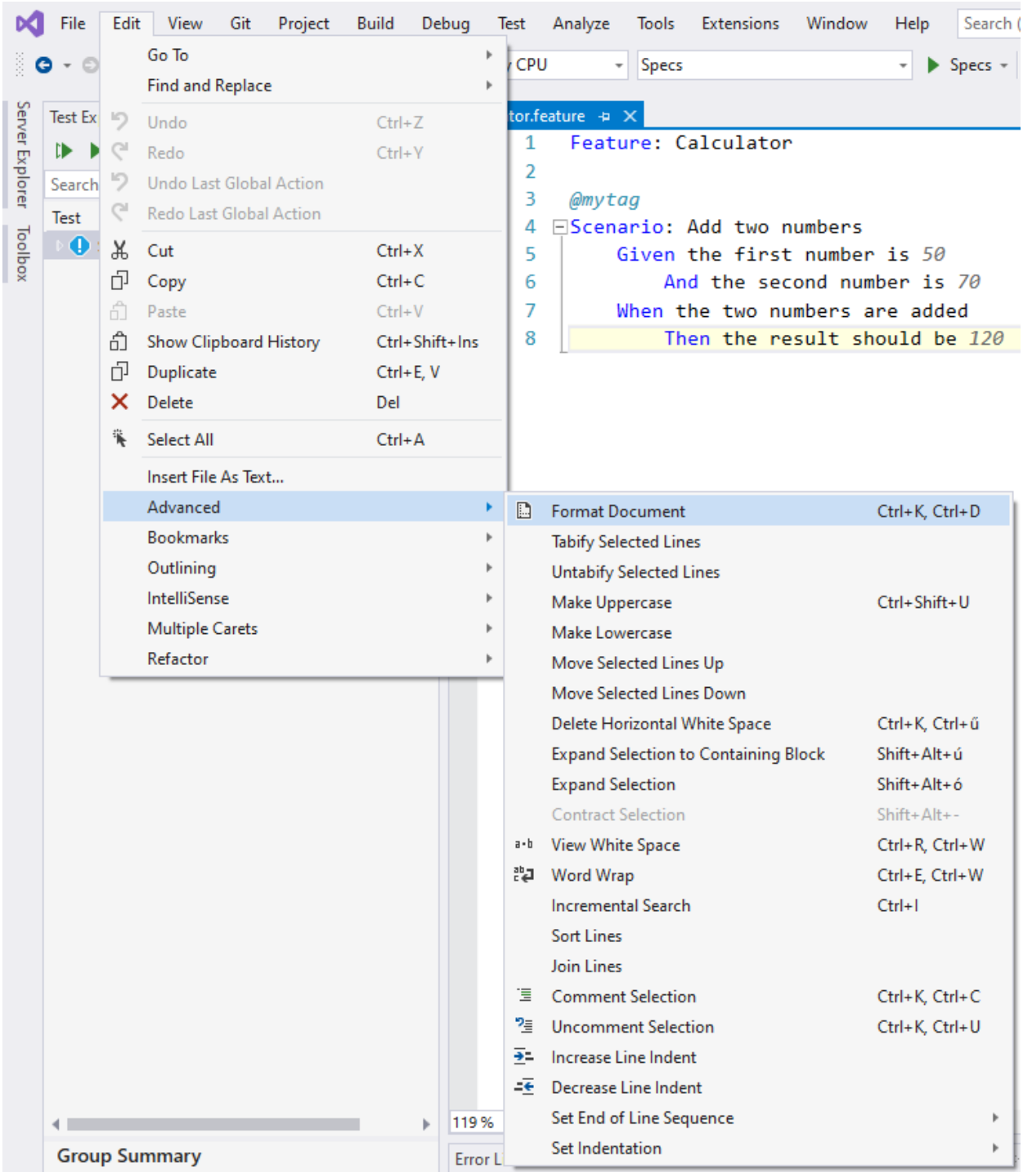


Fig. 2: Format document

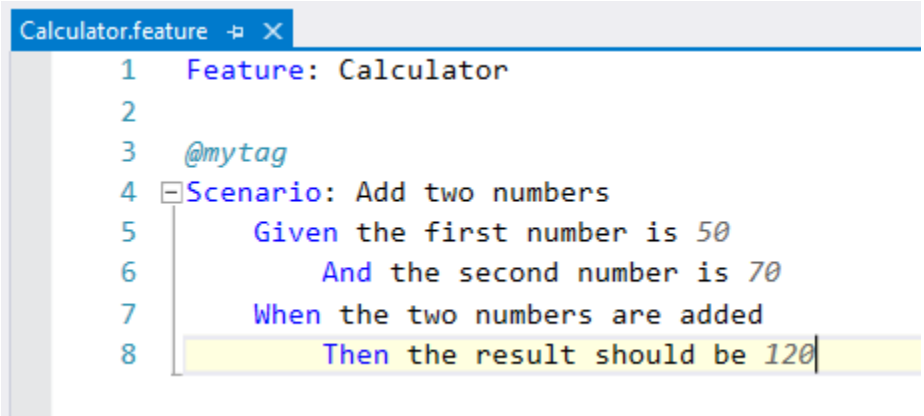


Fig. 3: Unformatted document

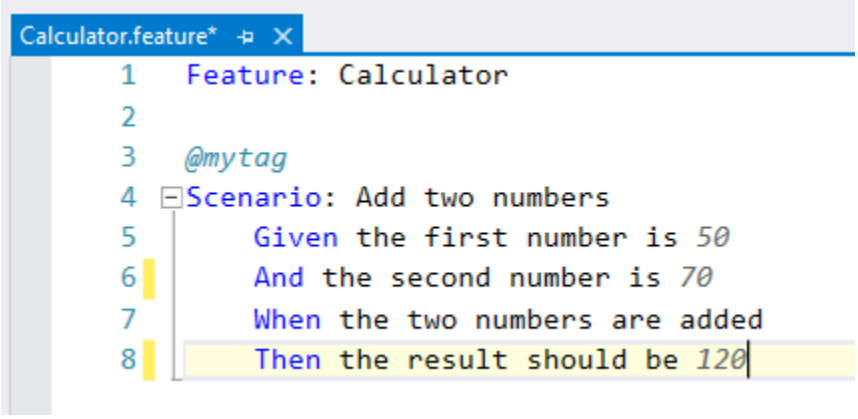
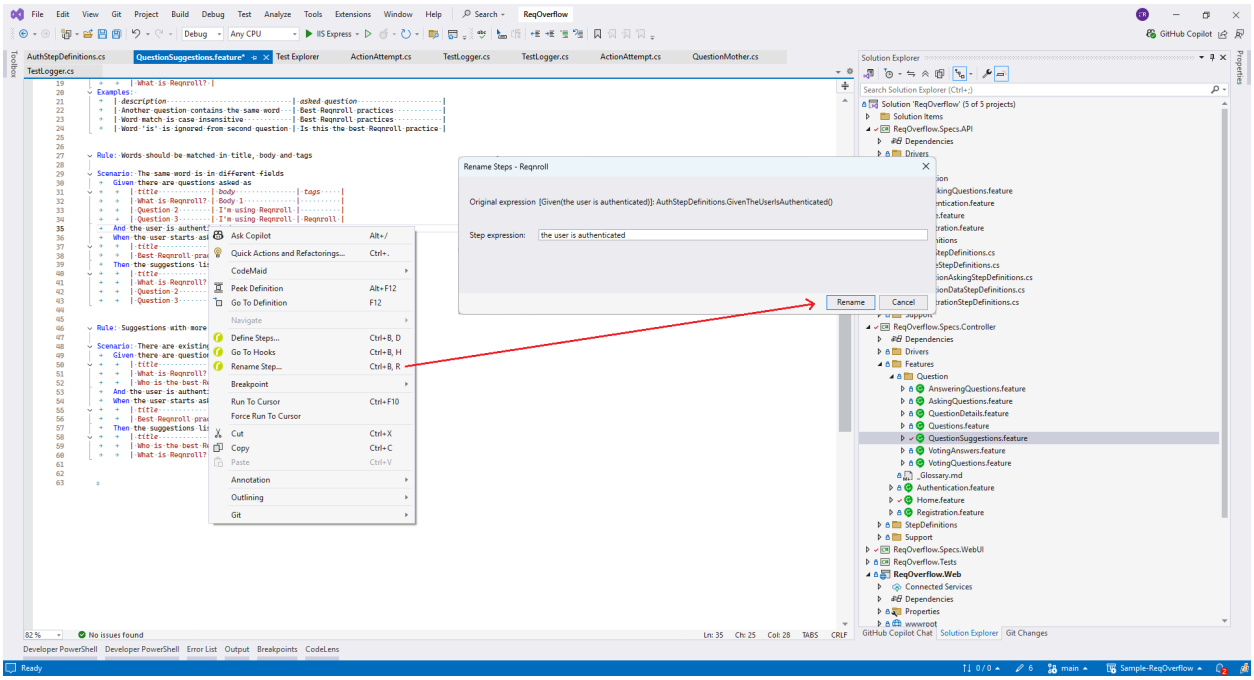


Fig. 4: Formatted document



> **Note**
>
> **Step Rename** is an experimental feature. It only works for those step methods that do not have parameters.

Navigation Features

You can navigate between the the step definition methods and the associated steps in your Gherkin feature files.

Navigating from a Scenario Step to a Step Definition

To navigate from a step in a feature file to the corresponding step definition method:

1. Place your cursor on the step in your feature file.
2. Right-click and select **Go To Step Definition** from the menu (F12).
3. The file containing the step definition is opened at the appropriate step definition method.

Navigating from a Step Definition to Steps in Feature Files

You can navigate from a step definition method to the matching step(s) in your feature file(s):

Right click the method and select *Find step definition usages*

Navigating from a Scenario to a Hook

To navigate from a scenario in a feature file to the hooks that are going to be involved during the execution of that scenario:

1. Place your cursor anywhere within a scenario in your feature file.

2. Right-click and select **Go To Hooks** from the menu
3. Select a hook from the pop-up menu
4. The file containing the hook is opened at the appropriate hook method.

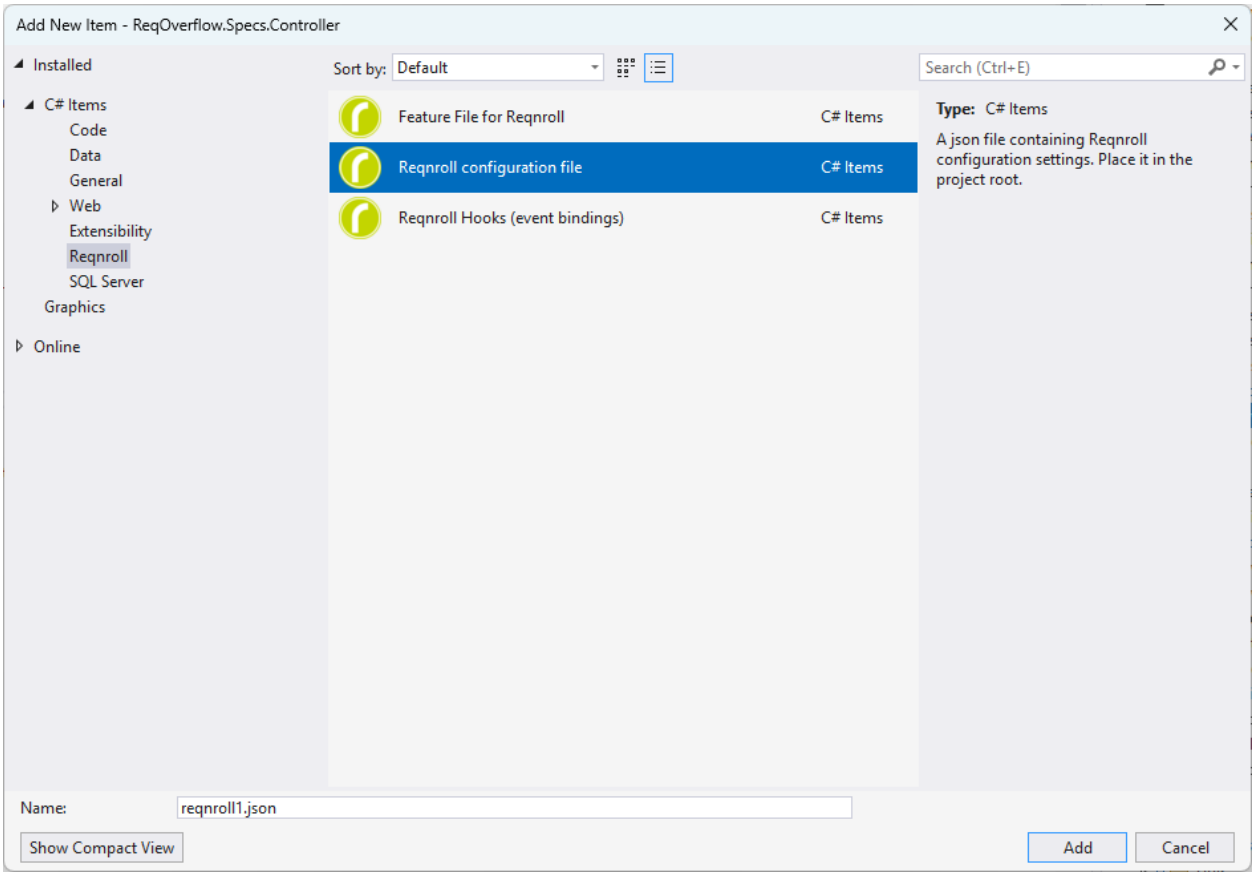
Defining Steps

You can automatically create a suitable class with step definition method snippets for a Gherkin feature file:

- Open your feature file.
- Right-click in the editor and select **Define steps…** from the menu.
- Enter a name for your class in the **Class name** field.
- Click on **Create** to generate a new step definition file or use the **Copy methods to clipboard** method to paste the snippet of code in an existing step definition file.

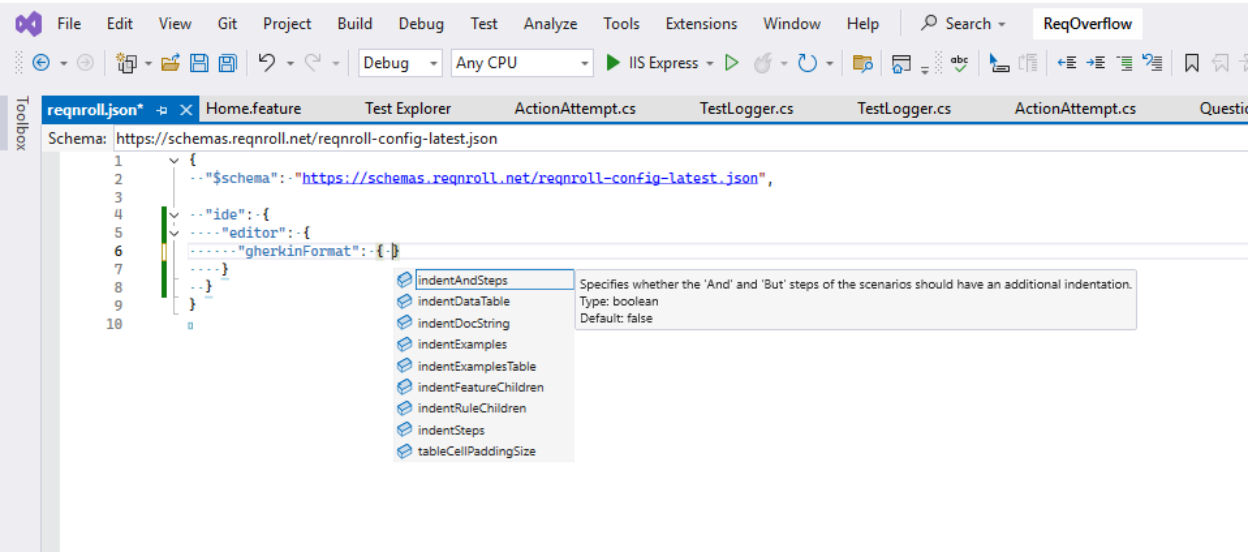
Visual Studio Extension Settings

To change the Visual Studio extension settings, edit the *`reqnroll.json`* config file. If you don't have the `reqnroll.json` file you can add it by right clicking on the *Reqnroll project -> Add -> New item… -> Add Reqnroll configuration file*.



> **Note**
>
> The formatting behavior can also be controlled by an *EditorConfig file*.

The configuration file has a JSON schema, therefore you will see all available properties as you start typing.



Configuring the Visual Studio extension in the configuration file

The `ide` section of the `reqnroll.json` file configures all settings related to the **Integrated Development Environment (IDE)** for Reqnroll projects. This section is extensible and allows fine-tuning of your development experience with Reqnroll. For other sections of the configuration file, please check the *Reqnroll Configuration Reference*.

The following sections are available within the `ide` section:

- *`editor` Section*
- *`traceability` Section*
- *`reqnroll` Section*
- *`bindingDiscovery` Section*

> **Note**
>
> You must build your project for the changes in `reqnroll.json` to take effect.

Example `ide` Configuration

```
"ide": {
  "editor": {
    "showStepCompletionAfterStepKeywords": true,
    "gherkinFormat": {
      "indentFeatureChildren": false,
      "indentSteps": true
    }
```

(continues on next page)

(continued from previous page)

```
  },
  "traceability": {
    "tagLinks": [
      {
        "tagPattern": "issue:(?<id>\\d+)",
        "urlTemplate": "https://github.com/org/repo/issues/{id}"
      }
    ]
  }
}
```

`editor` Section

- **Purpose**: Controls editor behaviors such as feature file formatting and code completion.
- **Settings**:
  - `showStepCompletionAfterStepKeywords` (boolean): Enables/disables step completions after keywords (`Given`, `When`, etc.). Default: `true`.
  - `gherkinFormat` (object): Controls the formatting of Gherkin feature files.
    - `indentFeatureChildren` (boolean): Indent children of `Feature` (`Background`, `Rule`, etc.). Default: `false`.
    - `indentRuleChildren` (boolean): Indent children of `Rule` elements. Default: `false`.
    - `indentSteps` (boolean): Indent steps in scenarios. Default: `true`.
    - `indentAndSteps` (boolean): Extra indent for "And"/"But" steps. Default: `false`.
    - `indentDataTable` (boolean): Indent `DataTable` arguments. Default: `true`.
    - `indentDocString` (boolean): Indent `DocString` arguments. Default: `true`.
    - `indentExamples` (boolean): Indent `Examples` blocks. Default: `false`.
    - `indentExamplesTable` (boolean): Indent `Examples` tables. Default: `true`.
    - `tableCellPaddingSize` (integer): Padding for table cells (spaces, default: 1).
    - `tableCellRightAlignNumericContent` (boolean): Specifies whether Table cells that contain digits should be right-aligned. Default: `true`.

Example

```
"ide": {
  "editor": {
    "showStepCompletionAfterStepKeywords": true,
    "gherkinFormat": {
      "indentFeatureChildren": false,
      "indentSteps": true,
      "indentAndSteps": false,
      "tableCellPaddingSize": 1
    }
  }
}
```

`traceability` Section

- **Purpose**: Enables traceability settings for scenarios, such as linking scenario tags to external issue trackers.
- **Settings**:
  - `tagLinks` (array): Defines patterns for tags and the corresponding external URLs.
    - Each entry:
      - `tagPattern` (string): Regex to match tag names (e.g., `"issue:(?<id>\\d+)"`).
      - `urlTemplate` (string): URL template using captured regex groups (e.g., `"https://github.com/org/repo/issues/{id}"`).

Example

The following example configures the extension to turn `@issue:1234` tags to clickable links to open the related GitHub issue.

```
"ide": {
  "traceability": {
    "tagLinks": [
      {
        "tagPattern": "issue:(?<id>\\d+)",
        "urlTemplate": "https://github.com/org/repo/issues/{id}"
      }
    ]
  }
}
```

`reqnroll` Section

> **Note**
>
> Specifying this section is only required for special cases when Reqnroll is not configured via NuGet packages.

- **Purpose**: Handles project-level settings related to Reqnroll itself.
- **Settings**:
  - `isReqnrollProject` (boolean): Enables the project as a Reqnroll project. Default: *(auto-detect)*.
  - `configFilePath` (string): Path to `App.config` or `reqnroll.json`. Default: *(auto-detect)*.
  - `version` (string): Specifies the Reqnroll version (e.g., `"2.3.1"`). Default: *(auto-detect)*.
  - `traits` (array): List of traits (e.g., `"XUnitAdapter"`, `"MsBuildGeneration"`, `"DesignTimeFeatureFileGeneration"`). Default: *(detected from NuGet packages)*.

`bindingDiscovery` Section

> **Note**
>
> Specifying this section is only required for special cases when the built-in binding discovery does not work.

- **Purpose**: Manages settings for discovering step bindings within the IDE.
- **Settings**:
  - `connectorPath` (string): File path to custom binding connector. Can reference environment variables (e.g., `%ENV_VAR%`). Relative paths use the default connector folder as base.

Gherkin Formatting Settings with EditorConfig

EditorConfig is a file format specification consistent coding styles maintained by EditorConfig.org and can be used by adding an `.editorconfig` file to your solution or project. Check the EditorConfig support documentation of Visual Studio for more details about EditorConfig files.

You can fine-tune the formatting for Gherkin files in your project by tweaking settings in your `.editorconfig` file. These options control indentation, table styling, and more for `*.feature` files recognized by the Reqnroll Visual Studio Extension.

Supported Settings

Setting Name	Description	Value Type	Default
`gherkin_indent_feature_children`	Indent child elements of Feature (Background, Rule, Scenario, Scenario Outline)	boolean	`false`
`gherkin_indent_rule_children`	Indent child elements of Rule (Background, Scenario, Scenario Outline)	boolean	`false`
`gherkin_indent_steps`	Indent steps in scenarios	boolean	`true`
`gherkin_indent_and_steps`	Apply additional indentation to `And`/`But` steps in scenarios	boolean	`false`
`gherkin_indent_datatable`	Indent DataTable arguments within steps	boolean	`true`
`gherkin_indent_docstring`	Indent DocString arguments within steps	boolean	`true`
`gherkin_indent_examples`	Indent the Examples block in Scenario Outlines	boolean	`false`
`gherkin_indent_examples_table`	Indent the Examples table within Examples blocks	boolean	`true`
`gherkin_table_cell_padding_size`	Number of spaces to pad table cells on each side	integer	1
`gherkin_table_cell_right_align_`	Right-align numeric values in table cells	boolean	`true`

Sample `.editorconfig` for Gherkin Files

Below is an example of an `.editorconfig` section for `*.feature` files.

> **Note**
>
> The Gherkin file format supports non-ASCII characters only with UTF-8 format. In order to ensure that the feature files are saved as UTF-8, you can specify the `charset` setting in the EditorConfig file as shown in the example below.

```
[*.feature]
charset = utf-8
```

(continues on next page)

(continued from previous page)

```
gherkin_indent_feature_children = true
gherkin_indent_steps = true
gherkin_indent_datatable = true
gherkin_indent_docstring = true
gherkin_indent_examples_table = true
gherkin_table_cell_padding_size = 2
gherkin_table_cell_right_align_numeric_content = true
```

Adjust the values above to match your project's formatting needs.

1.10.2 Reqnroll Rider integration

For setting up Reqnroll Rider integration, please check the related *Setup Rider* section of our IDE setup guide.

> **Documentation suggestions**
>
> This documentation page is in progress. Feel free to help by contributing to it in our open-source GitHub project. Each page contains a small (*edit*) icon to perform quick edits.

1.10.3 Reqnroll Visual Studio Code integration

For setting up Reqnroll Visual Studio Code integration using the VSCode Cucumber plugin, please check the related *Setup Visual Studio Code* section of our IDE setup guide.

> **Documentation suggestions**
>
> This documentation page is in progress. Feel free to help by contributing to it in our open-source GitHub project. Each page contains a small (*edit*) icon to perform quick edits.

1.11 Troubleshooting

1.11.1 Enabling Tracing

You can enable traces for Reqnroll. Once tracing is enabled, a new `Reqnroll` pane is added to the output window showing diagnostic messages.

To enable tracing, select **Tools | Options | Reqnroll** from the menu in Visual Studio and set **Enable Tracing** to 'True'.

1.11.2 Additional MSBuild logs

You can enable additional Reqnroll-specific MSBuild logs. Enable a higher level of detail in MSBuild.

The most important logs are already displayed with `detailed' output.

All Reqnroll log entries have a `[Reqnroll]` prefix.

1.12 Frequently Asked Questions

What is Reqnroll?

Reqnroll is an open-source .NET test automation tool to practice Behavior Driven Development (BDD).

Is Reqnroll a replacement of SpecFlow?

Yes. Reqnroll was forked from SpecFlow in January 2024 and it constantly being extended and fixed. The SpecFlow repository has been removed from GitHub by Tricentis in December 2024.

What license does Reqnroll use?

BSD 3-Clause License.

What is BDD?

BDD stands for Behavior Driven Development.

How can I contribute?

Reqnroll code is hosted in GitHub, constributors are welcome - see CONTRIBUTING.md for details.

Which .NET versions are supported?

All modern-ish versions of .NET and .NET Framework are *compatible*.

Where can I found Reqnroll NuGet packages?

In the official NuGet.

1.13 Samples

You can get a deeper understanding of Reqnroll by looking at sample and demo applications. The following list contains a few sample and demo applications from the community.

Sample	Reqnroll version	Contributors	Description
ReqOverflow	1.0.0	@gasparnagy	Shows different automation strategies for a realistic web application (Q&A site): controller, REST API, Web UI (Selenium).
ReqPlayWright	1.0.1	@Zsolt-Dunai	Sample PlayWright test project that shows how to setup Reqnroll and PlayWright with modern principles.

1.14 Support

For support options, please check the Support page of our website.