
Reqnroll

Reqnroll

Apr 10, 2024

CONTENTS

1	How to use the documentation	3
1.1	Quickstart	3
1.2	Installation & Setup	12
1.3	Guides	21
1.4	Gherkin	34
1.5	Automation Features	46
1.6	Execution Features	93
1.7	Extend Reqnroll	108
1.8	Integrations	116
1.9	IDE integrations	129
1.10	Troubleshooting	129
1.11	Frequently Asked Questions	130
1.12	Samples	130
1.13	Support	130

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](#) or [xUnit](#). On Reqnroll projects you can work using Visual Studio 2022, Visual Studio Core 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](#).

HOW TO USE THE DOCUMENTATION

Documentation is in progress

We are currently in progress of reviewing and restructuring the documentation ported from SpecFlow. If you find any glitches, please help with fixing it in our GitHub repository. 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 2022 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 2022 for Reqnroll, please make sure you install the Reqnroll for Visual Studio extension. See 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, double> _priceTable = new()
    {
        { "Electric guitar", 180.0 },
        { "Guitar pick", 1.5 }
    };

    public double 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
```

(continues on next page)

(continued from previous page)

```
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 by 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()
    {
        throw new PendingStepException();
    }

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

(continues on next page)

```
}

[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);
    }
}
```

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 ReqrollQuickstart.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 Reqroll 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 ReqrollQuickstart.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)  
    {  
        //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 phrasing 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
{
    [...]
```

(continues on next page)

(continued from previous page)

```
[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 Reqroll 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 adventurous, 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 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 <code>TestContext</code> class.
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 <code>Console.WriteLine</code> , 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.

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 Visual Studio integration*, 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 6.0

Listing 18: .NET CLI

```
dotnet new reqnroll-project -t mstest -f net6.0
```

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 2022](#)
- [Setup Visual Studio Code](#)
- [Setup Rider](#)

Setup Visual Studio 2022

In order to use Reqrroll with Visual Studio 2022, you need to install the [Reqrroll for Visual Studio 2022](#) extension.

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

1. Open Visual Studio 2022
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 `Reqrroll` to the *Search* text box on the right top corner.
4. Choose the *Reqrroll for Visual Studio 2022* from the list and click on the *Download* button.
5. Restart Visual Studio 2022.

For more details about the Reqrroll with Visual Studio extension, please check the [Reqrroll Visual Studio integration](#) page.

Hint: The Reqrroll 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 Reqrroll 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.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
    "MyReqrrollProject/**/*.cs",
```

(continues on next page)

(continued from previous page)

```
],
"cucumber.features": [ // sets the location of the feature files
  "MyReqnrollProject/**/*.*.feature",
]
}
```

Setup Rider

Rider plugin is not available yet

The Reqnroll Rider plugin has not yet been ported and released. Please, come back later or help contribute to it in our [open-source GitHub project](#).

We recommend using the Visual Studio 2022 or the Visual Studio Code integration until the plugin will be ready. The SpecFlow Rider plugin can also be used with limited capabilities.

Documentation is in progress

This documentation page is in progress. Please, come back later or help contribute to it in our [open-source GitHub project](#).

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 2022 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 `reqroll.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.: <code>en-US</code>) rather than generic (neutral) cultures (e.g.: <code>en</code>). <i>Default:</i> <code>en-US</code>
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. <i>Default:</i> not specified

generator

Use this section to define test generation options.

Setting	Value	Description
allowDebugGeneratedFiles	<code>true/:</code>	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 <code>true</code> . <i>Default:</i> <code>false</code>
allowRowTests	<code>true/:</code>	Determines whether “row tests” should be generated for <i>scenario outlines</i> . This setting is ignored if the <i>test execution framework</i> does not support row based testing. <i>Default:</i> <code>true</code>
addNonParallelizableMarkerForTags	List of tags	Defines a set of tags, any of which specify that a feature should be excluded from running in parallel with any other feature. See Parallel Execution . <i>Default:</i> empty

runtime

Use this section to specify various test execution options.

Setting	Value	Description
missingOr-Pending-StepsOutcome	Pending / Inconclusive / Ignore / Error	Determines how Reqnroll behaves if a step binding is not implemented or pending. See <i>Test Results</i> . <i>Default: Pending</i>
obsoleteBehavior	None / Warn / Pending / Error	Determines how Reqnroll behaves if a step binding is marked with [Obsolete] attribute. <i>Default: Warn</i>
stopAt-FirstError	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). <i>Default: false</i>

trace

Use this section to determine the Reqnroll trace output.

Setting	Value	Description
stepDefinition-Snippet-Style	CucumberExpression / RegexAttribute	Specifies the default <i>step definition style</i> . <i>Default: CucumberExpressionAttribute</i>
colored-Output	true/false	Determine whether Reqnroll should color the test result output. See <i>Color Test Result Output</i> for more details. You can override this setting to disable color (e.g. on build servers), with the environment variable <code>NO_COLOR=1</code> <i>Default: false</i>

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"
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```
]
}
```

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 <code>.dll</code>).

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 6.0
- .NET 7.0
- .NET 8.0

Note: Reqnroll can also be installed on any .NET frameworks that supports .NET Standard 2.0, including .NET Core 3.1 and .NET 5.0, but please note that these frameworks are out of support already.

Visual Studio

- Visual Studio 2022 (Workloads: ASP.NET and web development **or** .NET Desktop environment **or** .NET Core cross- platform development)

Test Execution Frameworks

- NUnit
- MsTest
- xUnit

1.3 Guides

This part contains details of the following topics.

1.3.1 How to change the test execution framework used by Reqnroll

Documentation is in progress

This documentation page is in progress. Please come back later or help contributing to it in our [open-source GitHub project](#).

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 *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.

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 27: C# Project File (.csproj)

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </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="1.0.0" />
    <PackageReference Include="Reqnroll.SpecFlowCompatibility" Version="1.0.0" />
  </ItemGroup>
```

(continues on next page)

(continued from previous page)

```
[...]  
</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 28: App.config

```
<?xml version="1.0" encoding="utf-8" ?>  
<configuration>  
  <configSections>  
    <section name="specFlow" type="Reqnroll.SpecFlowCompatibility.ReqnrollPlugin.  
↪ConfigurationSectionHandler, Reqnroll.SpecFlowCompatibility.ReqnrollPlugin" />  
  </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 29: C# Project File (.csproj)

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </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" />
  </ItemGroup>
</Project>
```

(continues on next page)

(continued from previous page)

```
<!-- Reqnroll -->
<PackageReference Include="Reqnroll.MsTest" Version="1.0.0" />
</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 30: reqnroll.json

```
{
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",
  "language": {
    "feature": "hu-HU"
  },
}
```

(continues on next page)

(continued from previous page)

```
"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")]
```

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

Reqroll 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
```

Reqroll 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 Reqroll to generate skeletons with regular expressions. You can achieve this with the following setting in the `reqroll.json` file:

Listing 31: reqroll.json

```
{
  "$schema": "https://schemas.reqroll.net/reqroll-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 Reqroll (has been removed in SpecFlow v4):

Listing 32: 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

Complete changelog of SpecFlow v4

Breaking Changes:

- Removed the ability to call steps from steps via string
- Removed .NET Core 2.1 support (min .NET Core version: 3.1)
- 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 easy 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

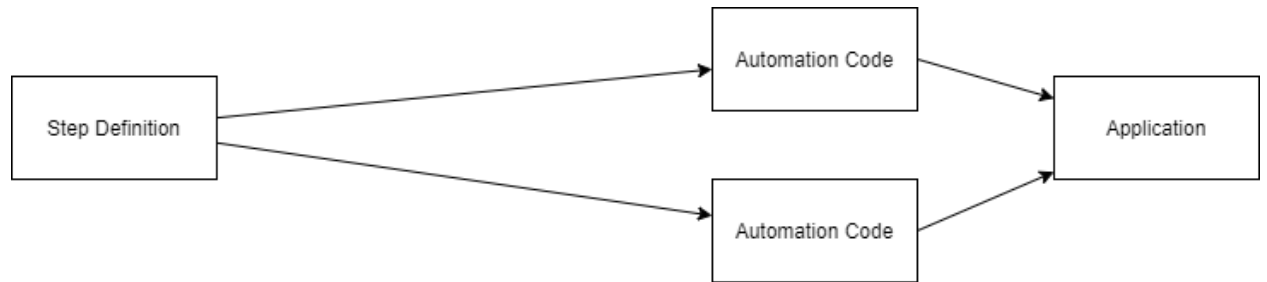
In this example you see how the code looks before and after refactoring with the Driver pattern.

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().NotNull();
            submissionPageObject.UrlLabel.Should().Be(expectedLabel);
            break;
        case "TYPE":
            submissionPageObject.TypeWebElement.Should().NotNull();
            submissionPageObject.TypeLabel.Should().Be(expectedLabel);
            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;
    }
}
```

(continues on next page)

(continued from previous page)

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

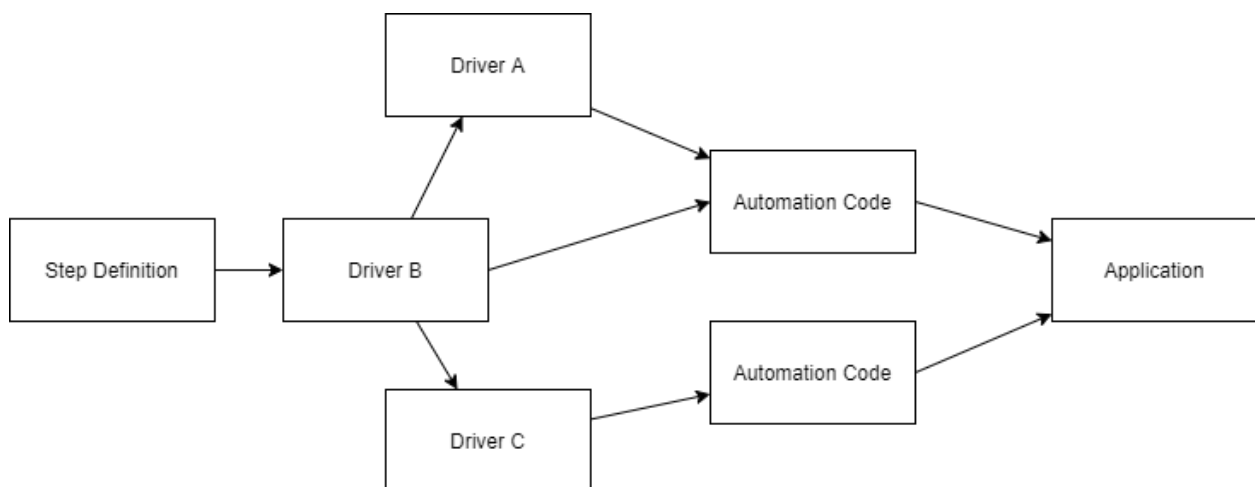
// ...
```

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

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

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

(continues on next page)

(continued from previous page)

```
    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>

1.4 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.

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 33: 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 34: 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 35: 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 36: 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 37: 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 38: 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 39: 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 40: 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 41: 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 42: 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 43: 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 44: 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 45: 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 what's 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 46: 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 47: Feature File

```
Scenario Outline: eating
  Given there are <start> cucumbers
```

(continues on next page)

(continued from previous page)

```
When I eat <eat> cucumbers
Then I should have <left> cucumbers
```

Examples:

start	eat	left
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 48: 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 49: 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 50: 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 were 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 Reqroll. 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 51: Step Definition File

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

Note: Bindings (step definitions, hooks, step argument transformations) are global for the entire Reqroll 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 52: 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 53: 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 54: 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 55: 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.
- Should return `void` or `Task`.

Step Matching Styles & Rules

There are multiple options for step definition matching:

- Use attributes with *cucumber expressions*
- Use attributes with regular expressions
- Use the method name as cucumber or regular expressions (for F#)

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 56: 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 57: 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 58: Hook File

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

(continues on next page)

(continued from previous page)

```
}  
    //...  
}
```

Listing 59: 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 60: Hook File

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

Listing 61: 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] [Before] [AfterScenario] or [After]	or possible	Automation logic that has to run before/after executing each scenario or scenario outline example
[BeforeScenarioBlock] [AfterScenarioBlock]	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 62: Hook File

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

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

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

Listing 63: 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 Reqroll. For example you can get the `ScenarioContext` injected as parameter in the `BeforeScenario` hook.

Listing 64: Hook File

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

Listing 65: 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 66: Hook File

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

Listing 67: Hook File with async method

```
[Binding]
public class Hooks
{
    [BeforeFeature]
    public static async Task SetupStuffForFeaturesAsync(FeatureContext featureContext)
    {
        // Example async operation
    }
}
```

(continues on next page)

(continued from previous page)

```

    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 68: 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 69: 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

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 70: Hook File

```

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

[BeforeScenario(Order = 100)]
public void LoginUser()

```

(continues on next page)

(continued from previous page)

```
{  
    // ...so we can log in to a clean database  
}
```

Listing 71: 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*.

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 `Reqroll.DataTable` instances.

To avoid cumbersome conversions in the step binding methods, Reqrroll 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 `reqroll.json` configuration file (see *Feature Language*). The following conversions can be performed by Reqrroll (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

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 transformation 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 72: 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 73: C# File

```
[Binding]
public class Transforms
{
    [StepArgumentTransformation]
    public XmlDocument XmlTransform(string xml)
    {
```

(continues on next page)

(continued from previous page)

```
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 74: C# File

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

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 75: 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 Reqroll 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# Reqroll project also in F# (As an extreme case, you can use your Reqroll 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 Reqroll project to ensure it is copied to the target folder and listed in the `reqroll.json` of the Reqroll 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 `reqroll.json` (the Reqroll 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 76: reqroll.json

```
{
  "$schema": "https://schemas.reqroll.net/reqroll-config-latest.json",
  "bindingAssemblies": [
    {
      "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 / Reqroll.

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

Listing 77: 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. Where `parameter-type` can be any of the following:

- A short name for some simple built-in types: `{int}`, `{long}`, `{byte}`, `{float}`, `{double}`, `{decimal}`
- `string` (`{string}`) that matches to quoted text wrapped with either `"` or `'`. E.g., `[Given("a user {string}")]` matches to `Given a user "Marvin"` or `Given a user 'Zaphod Beeblebrox'`.
- `word` (`{word}`) that matches to a single word without quotes. E.g., `[Given("a user {word}")]` matches to `Given a user Marvin`.
- Empty (`{}`) that matches to anything (like `(.*)` with regex).
- A type name without namespace that is supported by Reqroll as a parameter type (types with built-in support, enum types and types with *custom argument conversions*). 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.
- A custom type name you have specified in the `[StepArgumentTransformation]` attribute. 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`.

Optionals, alternatives

Cucumber expressions use the parentheses `((...))` for optionals and the `/` character to define alternatives. The step definition

Listing 78: 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 79: 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 Reqroll

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

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 (^/\$).

Listing 80: Step Definition File

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

1.5.8 Scoped Bindings

Bindings (step definitions, hooks, step argument transformations) are global for the entire Reqroll 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. Reqroll’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 81: 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 82: 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 83: 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 84: 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 85: 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 86: 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 Reqroll’s tracing will still display the steps, so you can work through the manual scenarios by following the steps in the report.

Listing 87: 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)
    {
    }
}
```

(continues on next page)

```
}  
}
```

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 88: 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.

Reqroll

- 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 90: 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 91: 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.

(continues on next page)

(continued from previous page)

↳to a collection of objects.

```
## `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:

```
```gherkin
```

```
Given these products exist
```

Sk	Na	Pr
BOOK1	Atlas Shrugged	25.04
BOOK2	The Fountainhead	20.15

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

Listing 92: 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 93: 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>`.

### 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 94: 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 95: 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 96: 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 Reqroll 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 97: 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 98: 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 99: Step Definition File

```
[Binding]
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 100: Step Definition File

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

In addition to throwing an exception if property values do not match, Reqroll 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 Reqroll 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 101: Feature File

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

Listing 102: 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 103: 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 Reqrroll 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 104: C# File

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

You have a table like this:

Listing 105: 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 106: 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 107: 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 108: 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 109: 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 |
 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 110: Step Definition File

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

(continues on next page)

```
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.
↳ 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 111: Feature File

```

Scenario: Containmentment
 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 112: Step Definition File

```

[Binding]
public class MusicCollectionSteps
{
 ScenarioContext _scenarioContext;

 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")]

```

(continues on next page)

(continued from previous page)

```

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 113: 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 **Reqrroll.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 114: 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 115: C# File

```
table.ToProjectionOfSet(collection);
table.ToProjectionOfInstance(instance);
```

Here are the definitions of Reqrroll `DataTable` extensions methods that convert tables and collections of `IEnumerables` to `EnumerableProjection`:

Listing 116: 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)
{

```

(continues on next page)



(continued from previous page)

```

 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 117: 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");
 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. Reqroll provides several ways of sharing data between bindings.

### Instance Fields

If the binding is an instance method, Reqroll 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 118: 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)
 {
 var books = actionResult.Model<List<Book>>();
 CustomAssert.Any(books, b => b.Title == title);
 }
}
```

### Context Injection

Reqroll 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 119: C# File

```
public class CatalogContext
{
 public CatalogContext()
 {
 ReferenceBooks = new ReferenceBookList();
 }
}
```

(continues on next page)

(continued from previous page)

```
public ReferenceBookList ReferenceBooks { get; set; }
}
```

Listing 120: 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

Reqroll 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 Reqroll 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 121: C# File

```
public class PersonData // the POCO for sharing person data
{
 public string FirstName;
 public string LastName;
}
```

Listing 122: 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;
 }
}
```

(continues on next page)

(continued from previous page)

```

 //... 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 123: C# File

```

public class CatalogContext
{
 public CatalogContext()
 {
 ReferenceBooks = new ReferenceBookList();
 }

 public ReferenceBookList ReferenceBooks { get; set; }
}

```

Listing 124: 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)
 {

```

(continues on next page)

```
 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 125: 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 126: .NET CLI

```
dotnet add package Reqroll.Autofac
```

This tells Reqroll 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 127: 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);
 }

 [AfterScenario]
 public void AfterScenario()
 {
 var webDriver = _kernel.Get<IWebDriver>();

 // Output any screenshots or log dumps etc
 }
}
```

(continues on next page)

(continued from previous page)

```
 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 128: 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 129: 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 130: Step Definition File

```
[Binding]
public class Bindings
{
 private readonly ScenarioContext _scenarioContext;

 public Bindings(ScenarioContext scenarioContext)
 {
 _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 `Reqroll.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 131: Feature File

```
@feature_tag
Feature: My feature

@rule_tag
Rule: My rule
```

(continues on next page)

(continued from previous page)

```
@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 132: 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 133: Feature File

```
Scenario: Accessing the current example

When I use examples in my scenario
Then the examples are available in ScenarioInfo

 Examples:
 | Sport | TeamSize |
```

(continues on next page)

(continued from previous page)

Soccer	11	
Basketball	6	

Listing 134: Step Definition File

```
public class ScenarioExamplesDemo
{
 private ScenarioInfo _scenarioInfo;

 public ScenarioExamplesDemo(ScenarioInfo scenarioInfo)
 {
 _scenarioInfo = scenarioInfo;
 }

 [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 135: 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 136: Step Definition File

```
[When("an error occurs in a step")]
public void AnErrorOccurs()
{
 "not correct".Should().Equal("correct");
}

[AfterScenario("showingErrorHandling")]
```

(continues on next page)

(continued from previous page)

```

public void AfterScenarioHook()
{
 if(_scenarioContext.TestError != null)
 {
 var error = _scenarioContext.TestError;
 Console.WriteLine("An error occurred:" + error.Message);
 Console.WriteLine("It was of type:" + error.GetType().Name);
 }
}

```

This is another example, that might be more useful:

Listing 137: 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.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 138: 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 139: 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();
}

```

(continues on next page)

```
 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 140: 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 141: 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 142: 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 |
```

(continues on next page)

(continued from previous page)

TargetLanguage	CSharp	
Language	en-US	
Description	In order to	

... and in the step definition:

Listing 143: 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;

 // 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 Reqroll Scenarios

Reqroll 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 Reqroll project (where the `.csproj` file is located)
3. Invoke `dotnet test`

Listing 144: 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 -
↪MyReqrollProject.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 145: Developer Command Prompt

```
> vstest.console.exe .\bin\Debug\net8.0\MyReqrollProject.dll
```

#### Executing scenarios from Visual Studio

Visual Studio contains a built-in test execution feature that can also be used for executing Reqroll 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.

Reqroll converts the tags in your feature files to test case categories:

- NUnit: Category or TestCategory
- MSTest: TestCategory
- xUnit: Trait (similar functionality, Reqroll 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 146: Feature File

```
Feature: Breakfast

@done
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 147: 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 148: 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

@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 149: Feature File

```

@US123
Feature: Breakfast

```

(continues on next page)

(continued from previous page)

```
@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:

Visual Studio Test ⓘ [View YAML](#) [Remove](#)

Task version  ▼

Display name \*

Test selection ^

Select tests using \* ⓘ  ▼

Test files \* ⓘ

Search folder \* ⓘ

Test results folder ⓘ


**Test filter criteria ⓘ**

Run only impacted tests ⓘ

Test mix contains UI tests ⓘ

## Reqroll

Visual Studio Test ⓘ

 View YAML  Remove

Task version  ▾

Display name \*

VsTest - testAssemblies

Test selection ^

Select tests using \* ⓘ

Test assemblies ▾

Test files \* ⓘ

```
\Debug*Specs.dll
!***TestAdapter.dll
!**\obj**
```

Search folder \* ⓘ

\$(System.DefaultWorkingDirectory)

Test results folder ⓘ

\$(Agent.TempDirectory)\TestResults

Test filter criteria ⓘ

Category=us123 & Category=done

Run only impacted tests ⓘ

Test mix contains UI tests ⓘ

## Azure DevOps - .NET Core 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 Reqroll 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 150: Step Definition File

```
[When("I set the current ScenarioContext to pending")]
public void WhenIHaveAPendingStep()
{
 throw new PendingStepException();
}
```

#### Custom Message

Listing 151: 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 152: Step Definition File

```
[When("I set the current ScenarioContext to pending")]
public void WhenIHaveAPendingStep()
{
 ScenarioContext.Pending();
}
```

## 1.6.4 Skipping Scenarios

You can skip programmatically scenarios using the `IUnitTestRuntimeProvider` interface.

### Example Code

Listing 153: Step Definition File

```
[Binding]
public sealed class StepDefinitions
{
 private readonly IUnitTestRuntimeProvider _unitTestRuntimeProvider;

 public CalculatorStepDefinitions(IUnitTestRuntimeProvider unitTestRuntimeProvider)
 {
 _unitTestRuntimeProvider = unitTestRuntimeProvider;
 }

 [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, MSTest).

### Limitations

Currently this works only in *step definitions*. It is not possible to use it in *hooks*.

## 1.6.5 Test Results

When Reqroll 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 Reqrroll 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 Reqrroll 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...

## 1.6.6 Parallel Execution

Reqrroll 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.

## Reqnroll

---

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)	N/A
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 Reqroll features in parallel with thread-level isolation

Test	Duration	Traits
SpecsNUnit (9)	9,2 sec	
SpecsNUnit.Features (9)	9,2 sec	
F1Feature (3)	3,1 sec	
F1_S1	1 sec	
F1_S2	1 sec	
F1_S3	1 sec	
F2Feature (3)	3,1 sec	
F2_S1	1 sec	
F2_S2	1 sec	
F2_S3	1 sec	
F3Feature (3)	3,1 sec	
F3_S1	1 sec	
F3_S2	1 sec	
F3_S3	1 sec	

### 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 Reqroll 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 (currently NUnit v3, xUnit v2 and MSTest)
- You have to ensure that your code does not conflict on static state.
- You must not use the static context properties of Reqroll `ScenarioContext.Current`, `FeatureContext.Current` or `ScenarioStepContext.Current` (see further information below).
- You have to configure the test runner to execute the Reqroll 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.
- All scenarios in a feature must be executed on the **same thread**. See the configuration of the test runners below. This ensures that the [BeforeFeature] and [AfterFeature] hooks are executed only once for each feature and that the thread has a separate (and isolated) `FeatureContext`.
- 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 `Reqroll.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 Reqroll project.

Listing 154: C# File

```
using NUnit.Framework;
[assembly: Parallelizable(ParallelScope.Fixtures)]
```

---

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

---

### MSTest Configuration

By default, MsTest does not run the tests in parallel. Parallelisation must be configured by setting an assembly-level attribute in the Reqroll project.

Listing 155: C# File

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

---

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

---

## xUnit Configuration

By default xUnit runs all Reqroll 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 `ReqrollException`.

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 Reqroll 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 156: C# File

```
[CollectionDefinition("ReqrollNonParallelizableFeatures", DisableParallelization =
→ true)]
public class ReqrollNonParallelizableFeaturesCollectionDefinition
{
}
```

### Running Reqroll 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.

### 1.6.7 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`.

### 1.6.8 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 157: Step Definition File

```
private readonly IReqnrollOutputHelper _reqnrollOutputHelper;

public CalculatorStepDefinitions(IReqnrollOutputHelper outputHelper)
{
 _outputHelper = 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**.

---

## 1.6.9 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 158: 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 Extend Reqroll

This part contains details of the following topics.

### 1.7.1 Value Retrievers

Reqroll 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; }
}

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());
 }
}
```

### 1.7.2 Plugins

Reqnroll supports the following types of plugins:

- Runtime
- Generator

All types of plugins are created in a similar way.

#### Runtime plugins

Runtime plugins need to target .NET Framework 4.6.2 and .NET Standard 2.0. 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

Generator plugins need to target .NET Framework 4.7.1 and .NET Core 3.1. 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

### Combined Package with both plugins

If you need to update generator and runtime plugins with a single NuGet package (as we are doing with the `Reqnroll.xUnit`, `Reqnroll.NUnit` and `Reqnroll.xUnit` packages), you can do so.

As with the separate plugins, you need two projects. One for the runtime plugin, and one for the generator plugin. As you only want one NuGet package, the **NuSpec files must only be present in the generator project**. This is because the generator plugin is built with a higher .NET Framework version (.NET 4.7.1), meaning you can add a dependency on the Runtime plugin (which is only .NET 4.6.1). This will not working the other way around.

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>
```

### 1.7.3 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;

 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 Reqroll project.

6. Add tag `@myMethodTagDecorator` to the feature file:

```
Calculator.feature -p X
1 Feature: Calculator
2 ![Calculator](https://specflow.org/wp-cont
3 Simple calculator for adding two numbe
4
5 Link to a feature: [Calculator](SpecFlowPr
6 Further read: [Learn more about hc
7
8 @myMethodTagDecorator
9 Scenario: Add two numbers
10 Given the first number is 50
11 And the second number is 70
12 When the two numbers are added
13 Then the result should be 120
```

7. Build the solution
8. Check the generated code behind file (`.feature.cs`) if it contains the NUnit Apartment attribute:

```
Calculator.feature.cs -p X
SpecFlowProject SpecFlowProject.Features.CalculatorFeature
78
79
80 [NUnit.Framework.TestAttribute()]
81 [NUnit.Framework.DescriptionAttribute("Add two numbers")]
82 [NUnit.Framework.ApartmentAttribute(System.Threading.ApartmentState.STA)]
83 [NUnit.Framework.CategoryAttribute(name: "myMethodTagDecorator")]
84 0 references | 0 changes | 0 authors, 0 changes
85 public virtual void AddTwoNumbers()
 {
```

## 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>

## 1.7.4 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 Integrations

This part contains details of the following topics.

### 1.8.1 Available Plugins

#### Plugins for DI Container

Name	Description	Download
Reqnroll.Autofac	Reqnroll plugin for using Autofac as a dependency injection framework for step definitions. <a href="#">Read more...</a>	
Reqnroll.Windsor	Reqnroll plugin for using Castle Windsor as a dependency injection framework for step definitions. <a href="#">Read more...</a>	



## Other Plugins

Name	Description	Download
<a href="#">Reqnroll.External Data</a>	Package to use external data in Gherkin scenarios. <a href="#">Read more...</a>	

## 1.8.2 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.

## Reqroll

---

**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 Reqroll project:**

```
builder.RegisterAssemblyTypes(typeof(YourClassInTheReqrollProject).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 Reqroll.Autofac.ReqrollPlugin;
```

Then

```
containerBuilder.AddReqrollBindings(typeof(YourClassInTheReqrollProject))
```

Or overload

```
containerBuilder.AddReqrollBindings<YourClassInTheReqrollProject>()
```

Or manually register like so:

```
builder
 .RegisterAssemblyTypes(typeof(TestDependencies).Assembly)
 .Where(t => Attribute.IsDefined(t, typeof(BindingAttribute)))
 .SingleInstance();
```

**3. A typical dependency builder method for [GlobalDependencies] with [ScenarioDependencies] probably looks like this:**

```
[GlobalDependencies]
public static void CreateGlobalContainer(ContainerBuilder containerBuilder)
{
 // Register globally scoped runtime dependencies
 Dependencies.RegisterGlobalDependencies(containerBuilder);

 //TODO: add Services that are shared globally.
}

[ScenarioDependencies]
public static void CreateContainerBuilder(ContainerBuilder containerBuilder)
{
 // Register scenario scoped runtime dependencies
 Dependencies.RegisterScenarioDependencies(containerBuilder);

 //TODO: add customizations, stubs required for testing
```

(continues on next page)

(continued from previous page)

```

containerBuilder.AddReqrollBindings<TestDependencies>()
}

```

#### 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 it's own like so:

Create a static method somewhere in the Reqroll 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 Reqroll manage your container

Create a static method in your Reqroll project to return a lifetime scope from your container. Note that Reqroll 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.AddReqrollBindings<TestDependencies>();
}

```

### 1.8.3 F# Support

*Bindings* for Reqroll 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 159: reqnroll.json

```
{
 "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",
 "bindingAssemblies": [
 {
 "assembly": "MyFSharpBindings"
 }
]
}
```

### 1.8.4 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();

```

(continues on next page)

(continued from previous page)

```
//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
}
```

### 1.8.5 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).

---

### 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 files only*. 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.
- `@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)
```

(continues on next page)

(continued from previous page)

```

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:

	A	B	C
1	product	price	color
2	Cookie	€ 2,50	brown
3	Bananas	€ 1,00	yellow
4			
5			
6			
7			

The screenshot shows the Excel interface with the 'other products' worksheet selected in the bottom tab bar. The table above represents the data from the first worksheet.

```

@DataSource:products.xlsx @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 piece of <product> in the basket

```

(continues on next page)



(continued from previous page)

```
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>
```

## 1.8.6 MSTest

Reqnroll supports MsTest V2.

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 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")]
```

## 1.8.7 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`

## 1.8.8 xUnit

Reqnroll supports xUnit 2.4 or later.

Documentation for xUnit can be found [here](#).

### Needed NuGet Packages

For Reqnroll: `Reqnroll.xUnit`

For xUnit: `xUnit`

For Test Discovery & Execution:

- `xunit.runner.visualstudio`
- `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*.

### Example

```
using System;
using Reqrroll;

[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 IDE integrations

This part contains details of the following topics.

### 1.9.1 Reqrroll Visual Studio integration

---

#### Documentation is in progress

This documentation page is in progress. Please come back later or help contributing to it in our [open-source GitHub project](#).

---

## 1.10 Troubleshooting

### 1.10.1 Enabling Tracing

You can enable traces for Reqrroll. Once tracing is enabled, a new Reqrroll pane is added to the output window showing diagnostic messages.

To enable tracing, select **Tools | Options | Reqrroll** from the menu in Visual Studio and set **Enable Tracing** to 'True'.

### 1.11 Frequently Asked Questions

---

#### Documentation is in progress

This documentation page is in progress. Please come back later or help contributing to it in our [open-source GitHub project](#).

---

### 1.12 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
ReqOver-flow	1.0.0	@gasparnagy	Shows different automation strategies for a realistic web application (Q&A site): controller, REST API, Web UI (Selenium).
ReqPlay-Wright	1.0.1	@ZsoltDunai	Sample PlayWright test project that shows how to setup Reqnroll and PlayWright with modern principles.

### 1.13 Support

For support options, please check the [Support page](#) of our website.