

RiverTech Automation Testing Test Guide

Installation of a BDD framework

Installed a BDD framework using NuGet packages windows. I am using LightBDD since in the test it states that RiverTech uses LightBDD framework.

Creating the API Testing Exercise Features & Steps

For the Api Testing Exercise, two Scenarios were created. One of the Scenario I am testing that the API returns a valid response code of **200** and in the other Scenario I am testing that the API returns the expected fields and values.

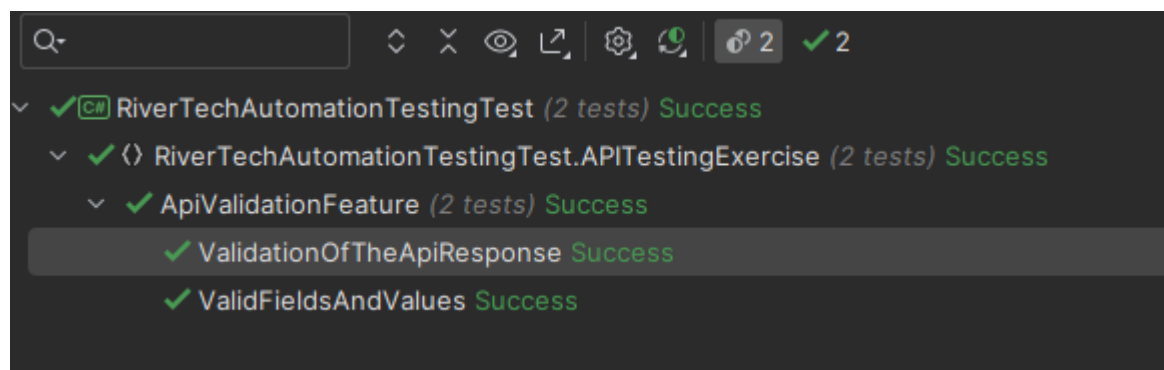
To make these test successfully I created another file that uses the same class name mentioned in the feature file **ApiValidationFeature** and created the steps for both scenarios. This was done to separate the features and steps.

Once the steps are created for both scenarios, they can be executed by clicking the Run button near the Scenario method or the class to execute all tests.

A screenshot of a code editor showing the implementation of the `ApiValidationFeature` class. The code is in C# and uses the LightBDD framework. It defines a `public partial class ApiValidationFeature` with a `[Scenario]` attribute. Inside the class, there is a `public void ValidationOfTheApiResponse()` method. This method calls `Runner.RunScenario()` with a lambda expression defining three steps: `Given_the_user_sets_a_get_request` (with a URL), `When_the_user_sends_the_http_request`, and `Then_the_response_returned_is_valid(200)`. Red arrows point to the `[Scenario]` attribute and the `ValidationOfTheApiResponse()` method.

```
13 public partial class ApiValidationFeature
14 {
15     [Scenario]
16     public void ValidationOfTheApiResponse()
17     {
18         Runner.RunScenario(
19             params steps: _ => Given_the_user_sets_a_get_request( apiUri: "https://jsonplaceholder.typicode.com/users/1"),
20             _ => When_the_user_sends_the_http_request(),
21             _ => Then_the_response_returned_is_valid(200));
22     }
23 }
```

Using the Unit Tests Explorer, the tester could identify if the test succeeded or not. In this case both tests are successful.

A screenshot of the Unit Tests Explorer in Visual Studio. The tree view shows a project named `RiverTechAutomationTestingTest` with a status of `(2 tests) Success`. Under this project, there is a test class `RiverTechAutomationTestingTest.APITestingExercise` with a status of `(2 tests) Success`. Under this class, there is a test feature `ApiValidationFeature` with a status of `(2 tests) Success`. The two individual tests are listed below: `ValidationOfTheApiResponse` and `ValidFieldsAndValues`, both with a status of `Success`. The top toolbar shows icons for search, expand, collapse, and other test-related actions, along with a summary of `2` tests passed.

```
✓ [C#] RiverTechAutomationTestingTest (2 tests) Success
  ✓ {} RiverTechAutomationTestingTest.APITestingExercise (2 tests) Success
    ✓ ApiValidationFeature (2 tests) Success
      ✓ ValidationOfTheApiResponse Success
      ✓ ValidFieldsAndValues Success
```

If a test fails, and the tester is comparing two values for example in the unit test console it will illustrate what the value is and what is the tester is expecting. This will easily inform the tester that there are some values that do not match in the GET HTTP request.

```

❌ ValidateFieldsAndValues [457ms] Expected string length 7 but was 8. Strings differ at index 7.
Expected string length 7 but was 8. Strings differ at index 7.
Expected: "-37.315"
But was:  "-37.3159"
-----^
at RTAutomationTestingTest.APITestingExercise.ApiValidationFeature.Then_the_api_response_return_the_fields_and_values(String expectedJson) in
C:\Users\Andre\RiderProjects\RTAutomationTestingTest\RTAutomationTestingTest\APITestingExercise\ApiValidationFeatureSteps.cs:line 49
at lambda_method26(Closure, NoContext, Object[])
at LightBDD.Core.Execution.Implementation.RunnableStep.RunStepAsync()
at System.Runtime.CompilerServices.AsyncMethodBuilderCore.Start[TStateMachine](TStateMachine& stateMachine)
at LightBDD.Core.Execution.Implementation.RunnableStep.RunStepAsync()

```

UI Automation Testin Exercise

To implement the UI automation test, the selenium webdriver needs to be installed from the NuGet packages. Once installed UI tests can be executed.

To execute a UI test, the tester should create a class and inside declares a method with a **[Test]** tag. This will mark the method as a test and can be executed from the unit test explorer.

First a driver should be created so that the tester can write tests against the Chrome browser.

```

[Test]
public void SauceDemo()
{
    IWebDriver driver = new ChromeDriver();
}

```

Then the tester should go through the Scenario manually and take note of the **ids** of the elements that are going to be tested. These id will be used in the test in various situations such as to write a string in a field (**SendKeys()**), to submit a form (**Submit()**), to click a button (**Click()**) etc etc... Once the implementation is completed, the tester can execute the test using the play button shown in the screen shot.

```

5
6 ▶ public class UISeleniumExercise
7 {
8     [Test]
9 ▶ public void SauceDemo()
10 {
11     IWebDriver driver = new ChromeDriver();
12     // Access the saucedemo website
13     driver.Navigate().GoToUrl("https://www.saucedemo.com/");
14
15     // Login to the website
16     driver.FindElement(By.Id("user-name")).SendKeys(text: "standard_user");
17     driver.FindElement(By.Id("password")).SendKeys(text: "secret_sauce");
18     driver.FindElement(By.Id("login-button")).Submit();
19

```

The above explanation was all used to implement the UI Automation Testing Exercise found inside the **UISeleniumExercise.cs**.