REST API Testing with Cucumber

Last modified: October 2, 2020

by baeldung

REST Testing

1. Overview

This tutorial gives an introduction to Cucumber, a commonly used tool for user acceptance testing, and how to use it in REST API tests.

In addition, to make the article self-contained and independent of any external REST services, we will use WireMock, a stubbing and mocking web service library. If you want to know more about this library, please refer to the introduction to WireMock.

2. Gherkin – the Language of Cucumber

Cucumber is a testing framework that supports Behavior Driven Development (BDD) allowing users to define application operations in plain text. It works based on the Gherkin Domain Specific Language (DSL). This simple but powerful syntax of Gherkin lets developers and testers write complex tests while keeping it comprehensible to even non-technical users.

2.1. Introduction to Gherkin

Gherkin is a line-oriented language using line endings, indentations and keywords to define documents. Each non-blank line usually starts with a Gherkin keyword, followed by an arbitrary text, which is usually a description of the keyword.

The whole structure must be written into a file with the *feature* extension to be recognized by Cucumber.

Here is a simple Gherkin document example:

```
Feature: A short description of the desired functionality

  Scenario: A business situation
    Given an initial context
    And another precondition
    When an event happens
    But there's an unwanted side effect
    Then a testable outcome is achieved
    And something else is also completed
```

In the following sections, we'll describe a couple of the most important elements in a Gherkin structure.

2.2. Feature

We use a *Gherkin file* to describe an application feature that needs to be tested. The file contains the *feature* keyword at the very beginning, followed up by the feature name on the same line and an optional description that may span multiple lines underneath.

Cucumber parser skips all the text, except for the *feature* keyword, and includes it for the purpose of documentation only.

2.3. Scenarios and Steps

A Gherkin structure may consist of one or more scenarios, recognized by the *Scenario* keyword. A scenario is basically a test allowing users to validate a capability of the application. It should describe an initial context, events that may happen and expected outcomes crossed by those events.

These things are done using steps, identified by one of the five keywords: *Given*, *When*, *Then*, *And* and *But*.

- *Given*: This step is to put the system into a well-defined state before users start interacting with the application. A *Given* clause can by considered a precondition for the use case.
- *When*: A *When* step is used to describe an event that happens to the application. This can be an action taken by the user or a trigger from an external system.
- *Then*: This step is to specify an expected outcome of the test. The outcome should be related to business values of the feature under test.
- *And* and *But*: These keywords can be used to replace the above step keywords when there are multiple steps of the same type.

Cucumber does not actually distinguish these keywords, however they are still there to make the feature more readable and consistent with the BDD structure.

3. Cucumber-JVM Implementation

Cucumber was originally written in Ruby and has been ported into Java with *Cucumber-JVM implementation*, which is the subject of this section.

3.1. Maven Dependencies

In order to make use of Cucumber-JVM in a Maven project, the following dependency needs to be included in the POM:

```
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>5.6.0</version>
    <scope>test</scope>
</dependency>
```

To facilitate JUnit testing with Cucumber, we need to have one more dependency:

```
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>5.6.0</version>
    <scope>test</scope>
</dependency>
```

Alternatively, we can use another artifact to take advantage of lambda expressions in Java 8, which won't be covered in this tutorial.

3.2. Step Definitions

Gherkin scenarios would be useless if they were not translated into actions and this is where step definitions come into play. Basically, a step definition is an annotated Java method with an attached pattern whose job is to convert Gherkin steps in plain text to executable code. After parsing a feature document, Cucumber will search for step definitions that match predefined Gherkin steps to execute.

In order to make it clearer, let's take a look at the following step:

```
Given I have registered a course in Baeldung
```

And a step definition:

```
@Given("I have registered a course in Baeldung")
public void setToCourse() {
    // write implementation
}
```

When Cucumber reads the given step, it will be looking for step definitions whose annotating patterns match the Gherkin text.

4. Creating and Running Tests

4.1. Writing a Feature File

Let's start with declaring scenarios and steps in a file with the name ending in the *.feature* extension:

```
Feature: Testing a REST API
  As a developer, I want to submit GET and POST requests to a web service,
    represented by WireMock

  Scenario: Data upload to a web service
    When users upload data on a project
    Then the server should handle it and return a success status

  Scenario: Data retrieval from a web service
    When users want to get information on the 'Cucumber' project
    Then the requested data is returned
```

We now save this file in a directory named *features*, on the condition that the directory will be loaded into the classpath at runtime, e.g. *src/main/resources*.

4.2. Configuring JUnit to Work With Cucumber

In order for JUnit to be aware of Cucumber and read feature files when running, the *Cucumber* class must be declared as the *Runner*. We also need to tell JUnit the place to search for feature files and step definitions:

```
@RunWith(Cucumber.class)
@CucumberOptions(features = "src/main/resources/feature")
public class CucumberIntegrationTest {
}
```

As you can see, the *features* element of *CucumberOption* locates the feature file created before. Another important element, called *glue*, provides paths to step definitions. However, if the test case and step definitions are in the same package as in this tutorial, that element may be dropped.

4.3. Writing Step Definitions

When Cucumber parses steps, it will search for methods annotated with Gherkin keywords to locate the matching step definitions.

A step definition's expression can either be a Regular Expression or a Cucumber Expression. In this tutorial, we'll use Cucumber Expressions.

The following is a method that fully matches a Gherkin step. The method will be used to post data to a REST web service:

```
@When("users upload data on a project")
public void userUploadDataOnProject() throws IOException {
}
```

And here is a method matching a Gherkin step and takes an argument from the text, which will be used to get information from a REST web service:

```
@When("users want to get information on the {string} project")
public void userGetInformationOnAProject(String projectName) throws IOException {
}
```

As you can see, the *userGetInformationOnAProject* method takes a *String* argument, which is the project name. This argument is declared by *{string}* in the annotation and even here it corresponds to *Cucumber* in the step text.

Alternatively, we could use a regular expression:

```
@When("^users want to get information on the \"(.+)\" project$")
public void userGetInformationOnAProject(String projectName) throws IOException {
}
```

Note, the "^ and "$ which indicate the start and end of the regex accordingly. Whereas *(.+)* corresponds to the *String* parameter.

We'll provide the working code for both of the above methods in the next section.

4.4. Creating and Running Tests

First, we will begin with a JSON structure to illustrate the data uploaded to the server by a POST request, and downloadable to the client using a GET. This structure is saved in the *json*String field, and shown below:

```
{
    "Testing Framework": "Cucumber",
    "Supported Language":
    [
        "Ruby",
        "Java",
        "Groovy",
        "Python",
        "Kotlin",
        "JavaScript",
        "..."
    ],
    "WebSite": "cucumber.io"
}
```

To demonstrate a REST API, we use a WireMock server:

```
WireMockServer wireMockServer = new WireMockServer(options().dynamicPort());
```

In addition, we'll use Apache HttpClient API to represent the client used to connect to the server:

```
CloseableHttpClient httpClient = HttpClients.createDefault();
```

Now, let's move on to writing testing code within step definitions. We will do this for the *userUploadDataOnAProject* method first.

The server should be running before the client connects to it:

```
wireMockServer.start();
```

Using the WireMock API to stub the REST service:

```
configureFor("localhost", wireMockServer.port());
stubFor(post(urlEqualTo("/products/courses"))
  .withHeader("Content-type", equalTo("application/json"))
  .withRequestBody(equalToJson(jsonString,true,true))
  .willReturn(aResponse().withStatus(200)));
```

Now, send a POST request with the content taken from the *json*String field declared above to the server:

```
HttpPost request = new HttpPost("http://localhost:" + wireMockServer.port() + "/courses");
StringEntity entity = new StringEntity(jsonString);
request.addHeader("content-type", "application/json");
request.setEntity(entity);
HttpResponse response = httpClient.execute(request);
```

The following code asserts that the POST request has been successfully received and handled:

```
assertEquals(200, response.getStatusLine().getStatusCode());
verify(postRequestedFor(urlEqualTo("/courses"))
  .withHeader("content-type", equalTo("application/json")));
```

The server should stop after being used:

```
wireMockServer.stop();
```

The second method we will implement herein is *userGetInformationOnAProject*String *projectName*. Similar to the first test, we need to start the server and then stub the REST service:

```
wireMockServer.start();
configureFor("localhost", wireMockServer.port());
stubFor(get(urlEqualTo("/products/courses/1"))
  .withHeader("content-type", equalTo("application/json"))
  .willReturn(aResponse().withBody(jsonString)));
```

Submitting a GET request and receiving a response:

```
HttpGet request = new HttpGet("http://localhost:" + wireMockServer.port() + "/products/" + projectName.toLowerCase());
request.addHeader("content-type", "application/json");
HttpResponse httpResponse = httpClient.execute(request);
```

We will convert the *HttpResponse* variable to a *String* using a helper method:

```
String responseString = convertResponseToString(httpResponse);
```

Here is the implementation of that conversion helper method:

```
private String convertResponseToString(HttpResponse response) throws IOException {
    InputStream responseStream = response.getEntity().getContent();
    Scanner scanner = new Scanner(responseStream, "UTF-8");
    String responseString = scanner.useDelimiter("\\Z").next();
    scanner.close();
    return responseString;
}
```

The following verifies the whole process:

```
assertEquals(responseString, convertResponseToString(httpResponse));
assertThat(responseString, containsString("\"Testing Framework\": \"Cucumber\""));
verify(getRequestedFor(urlEqualTo("/products/courses/1"))
  .withHeader("content-type", equalTo("application/json")));
```

Finally, stop the server as described before.

5. Running Features in Parallel

Cucumber-JVM natively supports parallel test execution across multiple threads. We'll use JUnit together with Maven Failsafe plugin to execute the runners. Alternatively, we could use Maven Surefire.

JUnit runs the feature files in parallel, rather than scenarios, which means **all the scenarios in a feature file will be executed by the same thread**.

Let's now add the plugin configuration:

```
<plugin>
    <groupId>net.serenity-bdd.maven.plugins</groupId>
    <artifactId>serenity-failsafe-plugin</artifactId>
    <version>${serenity.plugin.version}</version>
    <configuration>
        <includes>
            <include>**/CucumberIntegrationTest.java</include>
        </includes>
        <systemPropertyVariables>
            <threadCount>${threadCount}</threadCount>
        </systemPropertyVariables>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Note that:

- *parallel* can be *classes*, *methods*, or *both* – in our case, *classes* will make each test class run in a separate thread
- *threadCount* indicates how many threads should be allocated for this execution

That's all we need to do to run the Cucumber features in parallel.

6. Conclusion

In this tutorial, we covered the basics of Cucumber and how this framework uses the Gherkin domain-specific language for testing a REST API.

As usual, all code samples shown in this tutorial are available *over on GitHub*.