Run Test Cases In Parallel & Generate Reports Using Karate Tool

Last Updated: June 28, 2021

This tutorial explains how to perform some basic operations on the API, run test cases in parallel & generate reports with Karate Framework:

We have learned how to create a basic test script in our previous tutorial, we can now move ahead with learning some basic operations that can be performed while working with the API and the Karate Framework. There are many such operations and we will discuss a few commonly used ones in this tutorial.

We shall also delve into the process of running test cases in parallel by following a step-by-step approach. We will also discuss the current report that gets generated automatically and compare it with the Cucumber report that we can generate by integrating a plugin.

What is System Testing

NOW PLAYING

What
Testin

Jest: S
Testin

File M
Unix -
Overv
#3 Pa

Introd
Wirem

What You Will Learn: [hide]

Working With API And Karate Testing Tool

As discussed in the previous tutorial, in the **.feature** file that we had created, we could use different keywords to perform different operations on the API. **Karate** framework provides

us with several keywords that can be used to perform various actions.

=>Recommended Reading: API Testing With Karate Framework

Performing Various Operations

#1) Printing the response in the console

Print is a keyword that is provided by the Karate Framework to print the result in the console or the file. One of the most common uses is to print the response of the API. This can be very useful for the user.

We can do this by using the following lines of code:

```
Feature: fetching User Details

Scenario: testing the get call for User Details

Given url 'https://reqres.in/api/users/2'
When method GET
Then status 200

#We are printing the Response of the API using the print keyword#
Then print response
```

The above lines of code will give the following output:

```
18:15:44.495 [main] INFO com.intuit.karate - [print] {
    "ad": {
        "company": "StatusCode Weekly",
        "text": "A weekly newsletter focusing on software development, infrastructure, the
        "url": "http://statuscode.org/"
    },
    "data": {
        "last_name": "Weaver",
        "id": 2,
        "avatar": "https://s3.amazonaws.com/uifaces/faces/twitter/josephstein/128.jpg",
        "first_name": "Janet",
        "email": "janet.weaver@reqres.in"
    }
}
```

This is how we can print the response of the API in the console for the reading purpose, which can be used at the time of debugging.

#2) Declaring the Variables

We can declare the variables using the keyword **def** in the Karate framework and then use the declared variables in the code wherever necessary.

In the below example, we have added a few more lines of code to the existing **userDetails.feature** file to help declare the variables in the script.

```
Feature: fetching User Details

Scenario: testing the get call for User Details

Given url 'https://reqres.in/api/users/2'
When method GET
Then status 200

#We are printing the Response of the API using the print keyword
Then print response


# Declaring and assigning a string value:
Given def varName = 'value'

# using a variable
Then print varName
```

#3) Asserting the Actual Response to the Expected Response

Karate Framework helps in performing the Assertion related operations using the **match** keyword. The **match** is smart because white-space does not matter to it and the order of the keys is not important.

For using **match keyword,** we need to make the use of the double-equal sign “==” that represents a comparison.

Now we will try to elaborate on some uses of **match** keyword.

a) When the entire expected response is mentioned in the .feature file itself.

At certain times we have some data that we would like to validate immediately in the file itself. Usually, such kinds of data are mentioned while debugging the code.

We could do the same in the .feature file itself as shown below:

```
Feature: fetching User Details

Scenario: testing the get call for User Details

Given url 'https://reqres.in/api/users/2'
When method GET
Then status 200
#Asserting the reponse
#response variable is holding the Actual response from API
#Right hand side value is holding the expected Response
And match response == {"ad":{"company":"StatusCode Weekly","text":"A weekly newsle
focusing on software development, infrastructure, the server, performance,
and the stack end of things.","url":"http:\/\/statuscode.org\/"},"data":{"last_nam
"https:\/\/s3.amazonaws.com\/uifaces\/faces\/twitter\/josephstein\/128.jpg","first
"email":"janet.weaver@reqres.in"}}
```

If you send a request to the URL **'https://reqres.in/api/users/2'** in the browser, then **you will get the following response:**


```
{
  "ad": {
    "company": "StatusCode Weekly",
    "text": "A weekly newsletter focusing on software development, infrastructure,
and the stack end of things.",
    "url": "http://statuscode.org/"
  },
  "data": {
    "last_name": "Weaver",
    "id": 2,
    "avatar": "https://s3.amazonaws.com/uifaces/faces/twitter/josephstein/128.jpg"
    "first_name": "Janet",
    "email": "janet.weaver@reqres.in"
  }
}
```


We are trying to validate the above-said response using the *.feature file.

We have used the **match** keyword that is provided by the Karate Framework, which helps in performing different kinds of **Assertions** in the API response.

Note: We would need to transform the API response in one line to perform the above step. You could use any of the tools available online.

b) When the expected output is kept in an external JSON file.

In the above example, we discussed a scenario where we were having limited data and the same response that was easy to handle, but in the real scenarios, we will have gigantic sets of JSON responses that we might have to evaluate.

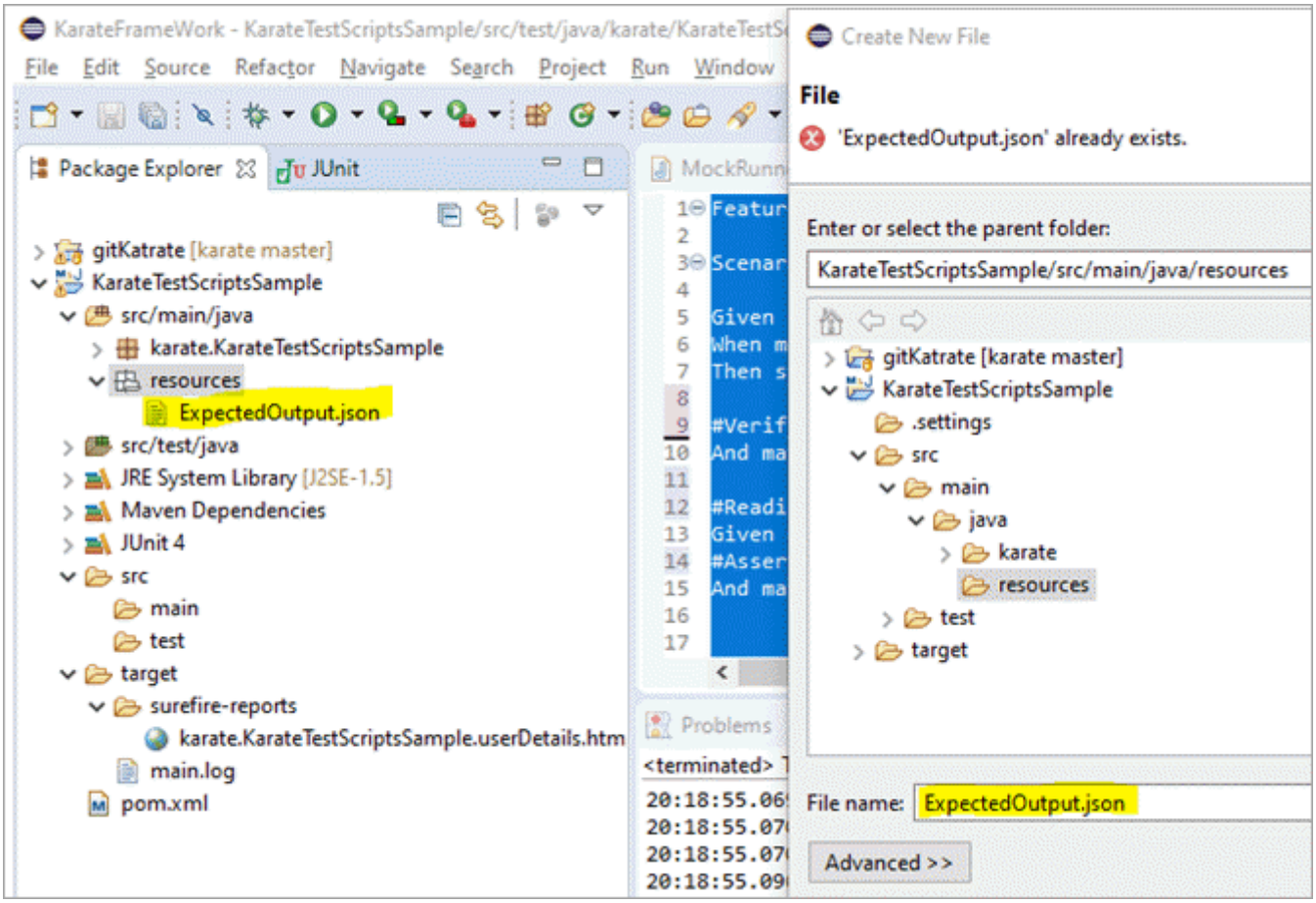
So, in those cases, it is better to keep the response in the external file and then verify the same.

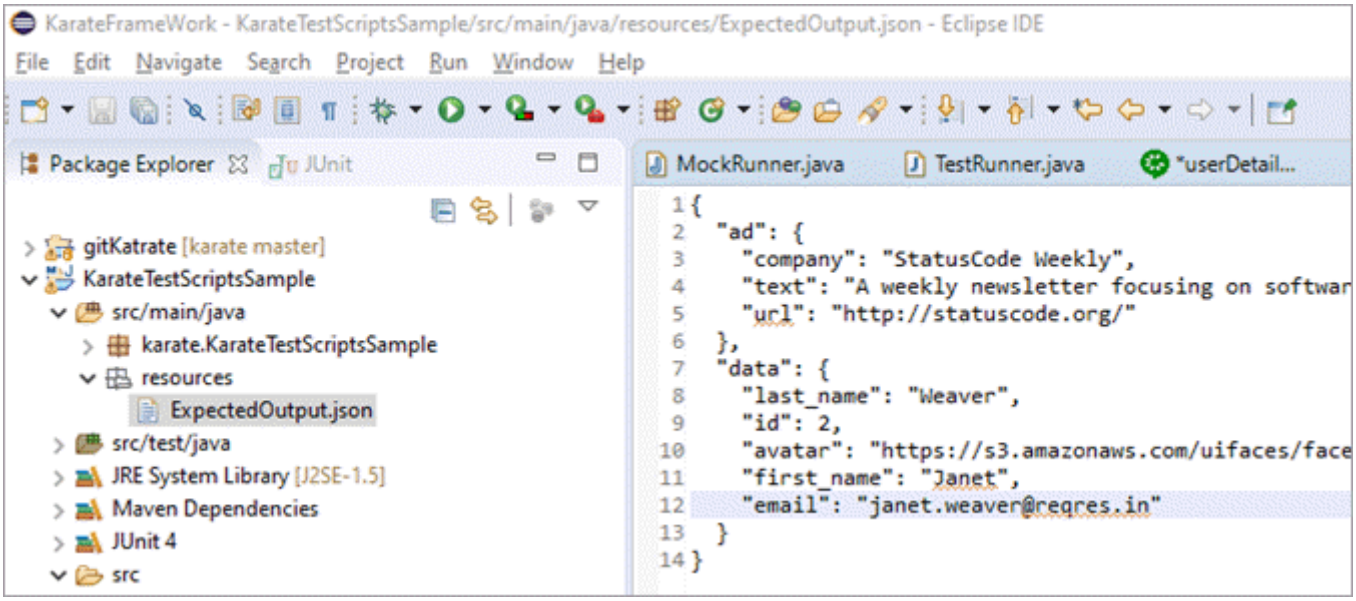
In the below example we will further discuss the same:

- Need to create an **ExpectedOutput.json** file in our Project folder as shown in the below image.

Create a new package resource -> Create a new file **ExpectedOutput.json**

And store the JSON response in this file and save it.





You would need to write the following code in your **userDetails.feature** file:

```
Feature: fetching User Details

Scenario: testing the get call for User Details

Given url 'https://reqres.in/api/users/2'
When method GET
Then status 200

#Verifying the JSON response by providing same in feature file
And match response == {"ad":{"company":"StatusCode Weekly","text":"A weekly newsle
on software development, infrastructure, the server, performance, and the stack
 end of things.","url":"http:\/\/statuscode.org\/"},"data":{"last_name":"Weaver","
```

```
"https:\/\/s3.amazonaws.com\/uifaces\/faces\/twitter\/josephstein\/128.jpg","first
"Janet","email":"janet.weaver@reqres.in"}}

#Reading the file ExpectedOutput.json and storing same response in variable expect
Given expectedResult=read('./resources/ExpectedOutput.json')
#Asserting the Actual Response with the Expected Response
And match response == expectedResult
```

In the above example, we are first reading the file **ExpectedOutput.json** and storing the response of it in the variable **expectedResult** using the **following lines of code:**

```
Given expectedResult=read('./resources/ExpectedOutput.json')
```

Then, we are putting the Assertion using the following lines of code, where we are matching the **Actual response** with the **expectedResult** response with the "**==**" operator.

```
And match response == expectedResult
```

c) Matching/Verifying certain values from the Response

Till now we have verified the entire response of the API, but every time you wouldn't want to verify the whole response. Sometimes, you would like to evaluate a part of the response only. Usually, we do the same when we use the other tools for API testing or while creating a framework.

To understand it further, let's take the following JSON response as an example:

```
{
  "ad": {
        "company": "StatusCode Weekly"
             }
}
```

If we want to verify that the parameter **company** should have the value as **StatusCode Weekly,** then we will have to create a JSON Path. This can be done by traversing the JSON file and using the "." (Dot operator)

The JSON path for the above response will be:

ad.company == "StatusCode Weekly"

Below is the code snippet which will help us in evaluating the values for the particular parameter. This code belongs to the **.feature** file.

```
Feature: fetching User Details

  Scenario: testing the get call for User Details
    Given url 'https://reqres.in/api/users/2'
```

```
    When method GET
    Then status 200
   #Verifying the JSON response by providing same in feature file
    And match response == {"ad":{"company":"StatusCode Weekly",
"text":"A weekly newsletter focusing on software development,
 infrastructure, the server, performance, and the stack end of things.",
"url":"http:\/\/statuscode.org\/"},"data":{"last_name":"Weaver","id":2,"avatar":
"https:\/\/s3.amazonaws.com\/uifaces\/faces\/twitter\/josephstein\/128.jpg",
"first_name":"Janet","email":"janet.weaver@reqres.in"}}
   #Reading the file ExpectedOutput.json and storing same response in variable exp
   Given expectedResult=read('./resources/ExpectedOutput.json')
   #Asserting the Actual Response with the Expected Response
   And match response == expectedResult


   ##Creating JSON path to verify the values of particular parameters##
   And match response.ad.url == "http://statuscode.org/"
   And match response.data.first_name == "Janet"
```

Below is the line of code, which is performing the parametric assertions.

```
And match response.ad.url == "http://statuscode.org/"
And match response.data.first_name == "Janet"
```

Using the JSON Path, we are evaluating the values for the parameters.

Performing Post Operations

Till now we have covered the basic scenarios of testing an API when the method was “**GET”.** But when we are working in the real environment, we have to send a lot of information to the server, so in that case, we use the “**POST”** method**.**

This section will give you an insight into working with the basic POST request.

Let’s get some brief ideas about the parameters that we need for sending the POST request.

#1) Creating a POST request, when the JSON body is mentioned in *.feature file

- Create a userDetailsPost.feature using the similar steps mentioned in the previous tutorial.
- Write the following lines of code:

```
Feature: Posting User Details

   Scenario: testing the POST call for User Creation
      Given url 'https://reqres.in/api/users'
      And request '{"name": "morpheus","job": "leader"}'
     When method POST
     Then status 201
```

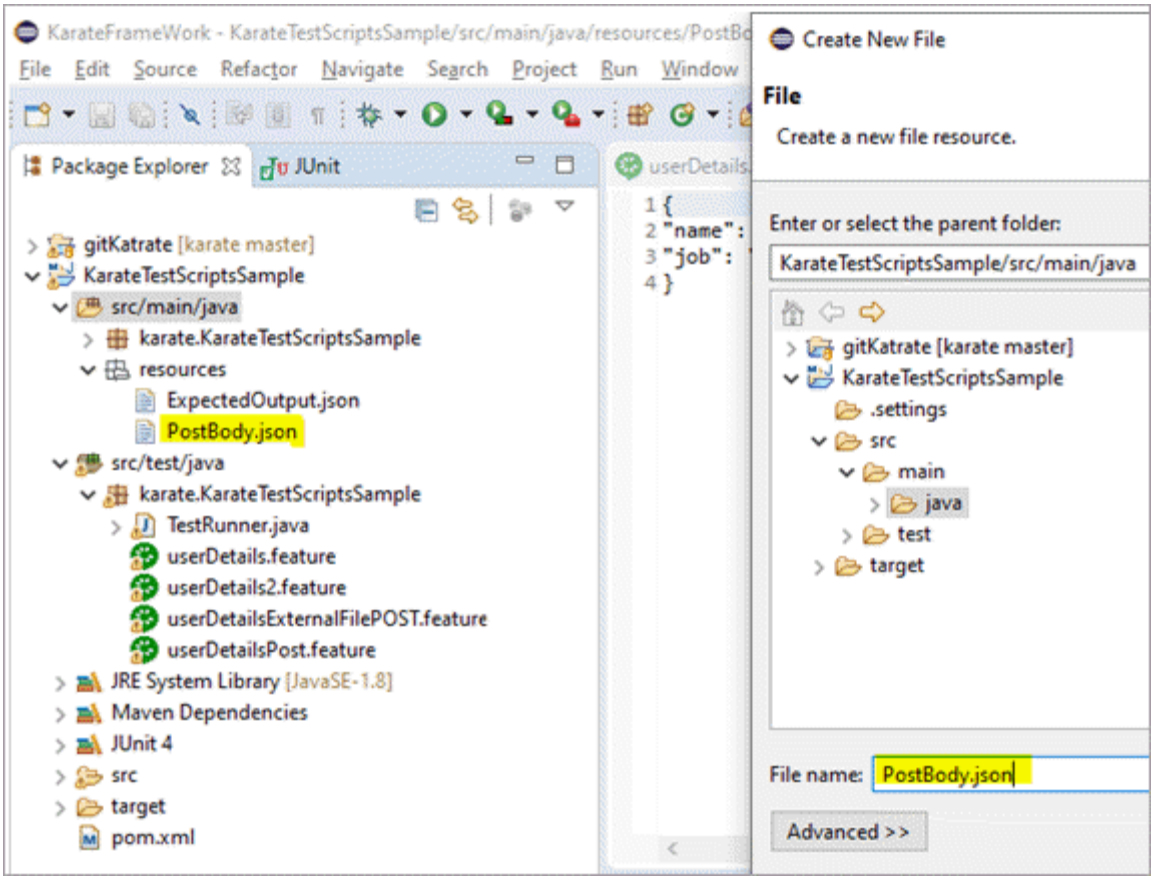
Since it is a POST request, which always needs to be accompanied with a body that needs to be sent to the server for a certain response, we have mentioned that under the following component:

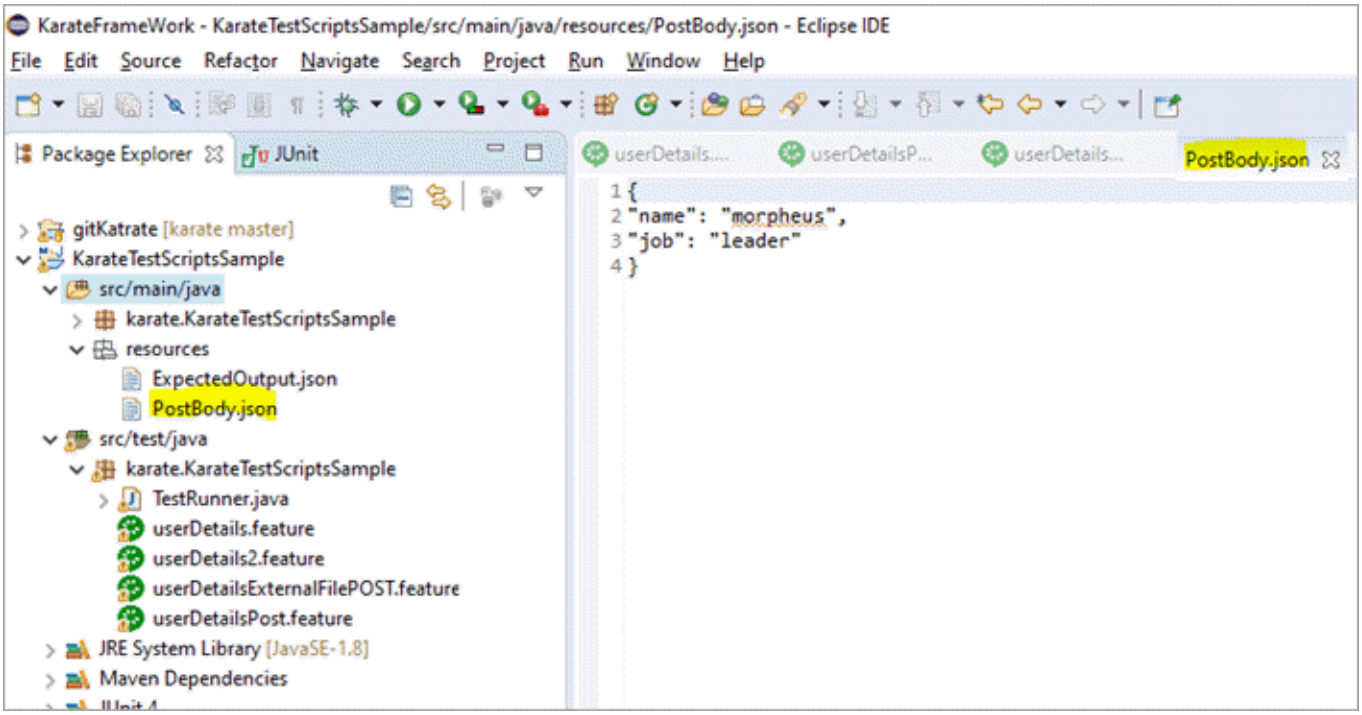
request: It takes a JSON body as the request which is required with the POST method.

#2) Creating a POST request, when the JSON body is mentioned in an external file

Usually, we will be having a huge request body, that would be difficult to mention in the ***.feature** file. So it is better to keep it in the external file.

- Need to create a PostBody.json file in our Project folder as shown below. Create a new package resource -> Create a new file PostBody.json and store the JSON Body in this file and save it.





Note: We have mentioned the Body of the POST method in the above JSON file.

- You would need to write the following code in your userDetailsPost**.feature** file:

```
Feature: Posting User Details

   Scenario: testing the POST call for User Creation using External File
       Given url 'https://reqres.in/api/users'
       Given postBody=read('./resources/PostBody.json')
       And request postBody
      When method POST
      Then status 201
```

We are reading the JSON Body from the PostBody.json using the following lines of code:

```
Given postBody=read('./resources/PostBody.json')
```

Note: All the **userDeatils.feature** files that we have created till now will require the basic **TestRunner.java** file to execute them, which we created in our Basic Test Script tutorial as shown below:

```
import org.junit.runner.RunWith;

import com.intuit.karate.junit4.Karate;

@RunWith(Karate.class)
public class TestRunner
{

}
```

Run Test Cases In Parallel

Now, since we have learned the steps to create a basic test script and performed some basic operations on the API, it is time we get started with working in the actual environment.

Usually, we have to run the Test Cases in parallel, to make the execution faster. Basically, the idea is to get more output in less time.

This is a core feature of the framework and does not depend on the JUnit, Maven, or Grade. It allows us to:

- Easily choose the features and the tags to run test suites in a simple manner.
- View the Parallel results under the surefire-plugin folder.
- We can even integrate the Cucumber JSON reports for better UI (which will be discussed shortly).

In Karate Framework, we do not need to perform many steps, to start the Parallel execution of our Test Cases. We just need to go through the following steps:

1) We now need to change the **TestRunner.java** file that we have been using till now. The code for the Parallel execution needs to be written in the above file.

Please keep in mind the following line, while executing your code in Parallel:

****We can't use @RunWith(Karate.class) annotation when we are trying to work in a parallel environment.**

Open the original **TestRunner.java** file and use the following code now:

```
import com.intuit.karate.Results;
import com.intuit.karate.Runner;
import org.junit.Test;

// important: do not use @RunWith(Karate.class) !
public class TestRunner {


    @Test
    public void testParallel() {

        Results results = Runner.parallel(getClass(),5);
    }
}
```

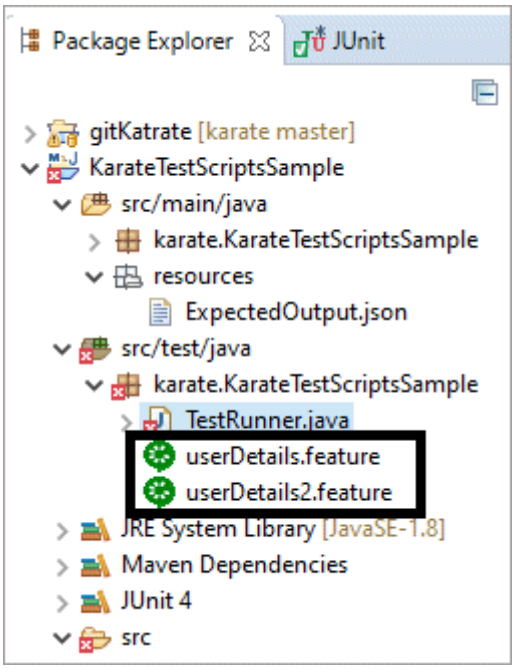
The following code will be applicable for **JUnit 4 Maven dependency

In the code snippet above, we have included the below line of code-

Results results = Runner.parallel(getClass(),5);

This line is instructing to run the Instance of the Test Cases in Parallel by dynamically fetching the classes at the Runtime.

2) Create a duplicate **userDetails.feature** file as mentioned below under the **src/test/java** folder.



Now we are all set for the Parallel execution with **2.features** file.

3) Go to **TestRunner.java** file created in the step above and **run it as JUnit Test.** With this, we will execute our test cases in parallel format.

For easy readability, some information is presented by the Karate Framework in the console, whenever the Test execution is completed.

The result looks something like the following:

With the parallel execution, all the features will execute in Parallel and the scenarios too will run in a parallel format.

Following the above steps, you will be able to start the very basic Parallel Execution of the API Test, using the Karate Framework.

You can study about parallel testing by traversing through the various filters on the page **Parallel Execution

Integrate Cucumber Plugin For Reporting

As we are using the **JUnit runner** for the execution of different scenarios that have been mentioned in the different ***.feature** files, it automatically creates a report for each of the feature files which are stored in the path **target/surefire-reports.**

It generates a **Basic UI formatted report** for presenting the test cases that have been executed.

But, the reports that are getting generated are not very pleasing in terms of the UI, and in order to share Reports with the stakeholders, we need something that is more user friendly and easily understandable.

In order to achieve such a reporting format, Karate Framework gives an option to integrate **Cucumber-reporting plugin** which will help us in generating a graphical formatted report, which will be more presentable.

Following are the steps to integrate the same:

#1) Add the following **Cucumber-Reporting** dependency to your POM.xml

```
<dependency>
        <groupId>net.masterthought</groupId>
```

```
        <artifactId>cucumber-reporting</artifactId>
        <version>3.8.0</version>
        <scope>test</scope>
</dependency>
```

#2) Edit the TestRunner.java file when there is only a single ***.feature** file in the project.

We need to update our TestRunner.java file, with the following generateReport() method for the Cucumber plugin.

```
public class TestRunner {


    @Test
    public void testParallel() {

        generateReport(results.getReportDir());
        assertTrue(results.getErrorMessages(), results.getFailCount() == 0);
    }

    public static void generateReport(String karateOutputPath) {
        Collection<File> jsonFiles = FileUtils.listFiles(new File(karateOutputPath
        final List<String> jsonPaths = new ArrayList(jsonFiles.size());
        jsonFiles.forEach(file -> jsonPaths.add(file.getAbsolutePath()));
        Configuration config = new Configuration(new File("target"), "demo");
        ReportBuilder reportBuilder = new ReportBuilder(jsonPaths, config);
        reportBuilder.generateReports();
    }

}
```

In the code mentioned above, we are performing the following actions:

- Creating a new instance of File
- Providing the path to store the files under the target folder
- Creating a ReportBuilder object which will create a new Cucumber report

Note: The above code works fine when we are having single ***. feature** file in our project.

#3) Edit the TestRunner.java file when there are **multiple *.feature** files in the Project.

We would need to add a line of code (highlighted in bold below) in order to ensure that parallel execution is taken care of, while the scenarios are being executed for the report generation.

```
public class TestRunner {


    @Test
    public void testParallel() {
        System.setProperty("karate.env", "demo"); // ensure reset if other tests (
        Results results = Runner.parallel(getClass(),5);

        generateReport(results.getReportDir());
        assertTrue(results.getErrorMessages(), results.getFailCount() == 0);
    }
```

```
    public static void generateReport(String karateOutputPath) {
        Collection<File> jsonFiles = FileUtils.listFiles(new File(karateOutputPath
        final List<String> jsonPaths = new ArrayList(jsonFiles.size());
        jsonFiles.forEach(file -> jsonPaths.add(file.getAbsolutePath()));
        Configuration config = new Configuration(new File("target"), "demo");
        ReportBuilder reportBuilder = new ReportBuilder(jsonPaths, config);
        reportBuilder.generateReports();
    }

}
```

After performing the above-mentioned steps, we will be able to successfully create a well-represented Graphical UI report using the **Cucumber – reporting** plugin.

We can find the report at the following path in our project as shown in the image below:

The following report was generated for our project, for all the operations we have performed until now in this Karate Framework Tutorial:

Conclusion

To summarize, in this tutorial we have discussed the basic operations that are useful on a day-to-day basis while working with the **Karate Framework** and how to execute **multiple *.feature files** in parallel. We also learned to create a graphical report for the users using the **Cucumber reporting** plugin.

First, we discussed the basic operations that can be performed on the API. We discussed how we can send the POST body/request to the server, either by mentioning the body in the *.feature file itself (which is usually not a recommended practice) or by using an external file (a recommended practice, in order to maintain a clean code).

Second, after following a few basic steps, we could successfully execute the test result for two ***.feature** files that were executed in parallel, just by adding a few lines of code in the **TestRunner.java** file enabling the initiation of the parallel run.

Further to this, we learned how to transform the native JUnit Test report to a Cucumber report by integrating the **Cucumber-reporting** plugin. The plugin allows us to generate reports that have a better UI, are much more comprehensible for the user, and hence provide a better user experience for the stakeholders with whom these reports are shared.

By now, you should be able to perform some basic operation, run the test cases parallelly, and generate an easy-to-read report for the users.

<<PREV

Recommended Reading

- Karate Framework Tutorial: Automated API Testing With Karate
- 10 Best API Testing Tools in 2021 (SOAP and REST API Testing Tools)
- How To Run Cucumber With Jenkins: Tutorial With Examples
- Guide to Generate Extent Reports in Selenium WebDriver
- Specflow Reporting: How to Generate Test Reports and Execute Selective Tests
- How to Manage Requirements, Execute Test Cases and Generate Reports Using TestLink – Tutorial #2
- Running Your Appium Tests in Parallel Using Appium Studio for Eclipse
- How to Run Large-scale Execution of Appium Tests in Parallel

About SoftwareTestingHelp

Helping our community since 2006! Most popular portal for Software professionals with **100 million+ visits and 300,000+ followers!** You will absolutely love our tutorials on QA Testing, Development, Software Tools and Services Reviews and more!

Recommended Reading

Karate Framework Tutorial: Automated API Testing With Karate

10 Best API Testing Tools in 2021 (SOAP and REST API Testing Tools)

How To Run Cucumber With Jenkins: Tutorial With Examples

Guide to Generate Extent Reports in Selenium WebDriver

Specflow Reporting: How to Generate Test Reports and Execute Selective Tests

How to Manage Requirements, Execute Test Cases and Generate Reports Using TestLink – Tutorial #2

Running Your Appium Tests in Parallel Using Appium Studio for Eclipse

How to Run Large-scale Execution of Appium Tests in Parallel

Join Our Team!