

# Testing APIs using Karate DSL



## What you'll learn

- [Prerequisites](#)
- [Integrating](#)
- [Defining](#) tests using Karate DSL
  - [Run](#) the test and push the test report to Xray
  - [Validate](#) in Jira that test results are available
  - [Jira UI](#)
- [Tips](#)
- [References](#)

## Source-code for this tutorial

- code is available in [GitHub](#)

## Overview

Karate is an open-source tool to combine API test-automation, mocks, performance and UI automation in one framework. The BDD syntax popularized by Cucumber is language-neutral, and accessible for non-programmers. Assertions and HTML reports are built-in, and you can run tests in parallel.

## Prerequisites

For this example we will use Karate DSL, that has available a Maven archetype that will build the skeleton of the project.

The Karate Maven archetype will create the `pom.xml`, recommended directory structure, sample test and [JUnit 5](#) runner.

We will need:

- Access to a [demo site](#) that we aim to test
- Maven environment with [JUnit 5](#)

To start using Karate DSL please follow the [Get Started](#) documentation.

The test consists in validating the listing operation of the API from the [demo site](#) and a second one to create and fetch the created user to validate the success.

By default we see 5 files being created, one that will hold the logging configurations, called *logback-test.xml*

#### logback-test.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %
msg%n</pattern>
        </encoder>
    </appender>

    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>target/karate.log</file>
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %
msg%n</pattern>
        </encoder>
    </appender>

    <logger name="com.intuit" level="DEBUG"/>

    <root level="info">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
    </root>

</configuration>
```

A second file that will have the Karate configurations (*karate-config.js*) regarding the environments, it will allow the definitions of variables per environment or to define actions to be executed in different environments:

#### karate-config.js

```
function fn() {
    var env = karate.env; // get system property 'karate.env'
    karate.log('karate.env system property was:', env);
    if (!env) {
        env = 'dev';
    }
    var config = {
        env: env,
        myVarName: 'someValue'
    }
    if (env == 'dev') {
        // customize
        // e.g. config.foo = 'bar';
    } else if (env == 'e2e') {
        // customize
    }
    return config;
}
```

For this example we will not change the above files. We still have 3 other files that were created, one called *ExamplesTest.java* that is a special Java class that will allow the execution in parallel of the tests defined in Karate (you can find more information [here](#)). In this class we have added a method `.outputJUnitxml(true)` in the runner to enable the Junit report to be generated in the output.

The final version of the class is below.

### ExamplesTest.java

```
package examples;

import com.intuit.karate.Results;
import com.intuit.karate.Runner;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class ExamplesTest {

    @Test
    void testParallel() {
        Results results = Runner.path("classpath:examples")
            .outputJunitXml(true)
            .parallel(2);
        assertEquals(0, results.getFailCount(), results.
            getErrorMessages());
    }

}
```

Karate supports JUnit 5 and the advantage is that you can have multiple methods in a test-class. Notice that in the below class we use the `@Karate.Test` tag that will identify this method as a test.

In here we are defining what is the test case we want to execute, in this case we are saying that we want to execute the *"DummyUsers"* feature.

### DummyUsersRunner.java

```
package examples.users;

import com.intuit.karate.junit5.Karate;

class DummyUsersRunner {

    @Karate.Test
    Karate testDummyUsers() {
        return Karate.run("DummyUsers").relativeTo(getClass());
    }

}
```

The final file is the feature file where the tests are defined, although it has similarities with Cucumber, you will see that there is a staggering difference, in this case there is no code behind that you need to define, the notation defined here will be handled directly by Karate. Notice that Json is supported by default and there are some keywords that will trigger actions, check the Karate documentation for more information.

For our example we have defined two scenarios, one to get all dummy users and then fetch the first user by id and another that will create a user and fetch it to validate its creation.

## dummyusers.feature

Feature: sample karate test script

Background:

```
* url 'http://dummy.restapiexample.com/api/v1/'
```

Scenario: get all dummy users and then get the first user by id

Given path 'employees'

When method get

Then status 200

```
* def first = response.data[0]
```

Given path 'employee', first.id

When method get

Then status 200

Scenario: create a dummy user and then get it by id

```
* def user =
```

```
  """
```

```
  {
```

```
    "name": "Karate Test User",
```

```
    "salary": "3000",
```

```
    "age": "35",
```

```
  }
```

```
  """
```

Given path 'create'

And request user

When method post

Then status 200

```
* def id = response.data.id
```

```
* print 'created id is: ', id
```

Given path 'employee',id

When method get

Then status 200

And match response contains {status:success}

Let us go over some specificities of the above code to make it more clear.

First notice that we are using Gherkin language with extra definitions, we have a *Feature* with two *Scenarios*, one *Background* common to both *Scenarios*, where we have defined the default url to be used.

In the scenarios we are using Gherkin language (using the Given-When-Then keywords) and, as Gherkin supports [catch-all symbol '\\*'](#), each time you want to use a script inline prefix it with '\*'.

In the first scenario we are performing a *GET* from the default url (defined in the background) plus what is defined in the *path* and validating that we receive a HTTP 200. Then we extract from the response the first entry of the data element and save it in a variable *first*.

Still in the same test we are performing the last *HTTP GET*, now to the url plus '*employee*' adding the value of the variable *first* in the query string and validating that we get an *HTTP 200*.

The second scenario is a little more complicated as we are performing a *POST* request with a user object in the *BODY* and then extracting the user id to perform a *GET* with it and check if the user was created with success.

Once the code is implemented it can be executed with the following command, that will execute all tests present:

```
mvn test
```

```
mvn test "-Dkarate.options=--tags ~@skipme classpath:examples/DummyUsers
/dummyusers.feature" -Dtest=ExamplesTest
```


```
feature) classpath:examples/dummyUsers/dummyUsers.feature
scenario: 2 passed: 2 failed: 0 | time: 3.75ms

11:10:11.924 [main] INFO com.intuit.karate.Suite - <os>os: feature 1 of 1 (0 remaining) classpath:examples/dummyUsers/dummyUsers.feature
Karate version: 1.1.0

elapsed: 5.05 | threads: 3 | thread time: 3.75
features: 1 skipped: 0 | efficiency: 8.74
scenarios: 2 passed: 2 failed: 0

HTML report: (paste into browser to view) | Karate version: 1.1.0
https://crystalcandor.com/projects/karate62/KarateTutorial/target/karate-reports/karate-summary.html

[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.256 s - in examples.ExampleTest
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 7.347 s
[INFO] Finished at: 2022-01-28T11:40:11Z
[INFO]
```



2

0

[Home](#) / [Tags](#) / [Feedback](#) / [sample4@cs.cmu.edu/monitors/monitors/Answers](#) / [sample4@cs.cmu.edu/monitors/monitors/monitors](#)

### Statistics

2022-05-20 10:31 AM

2022 create a dummy user and then get the first user by id

id	Background	id	Background
1	When method call	10	When method call
2	When status 200	11	When method call
3	When status 200	12	When method call
4	When status 200	13	When method call
5	When status 200	14	When method call
6	When status 200	15	When method call
7	When status 200	16	When method call
8	When status 200	17	When method call
9	When status 200	18	When method call

### Statistics

2022 create a dummy user and then get the first user by id

id	Background	id	Background
1	When method call	10	When method call
2	When status 200	11	When method call
3	When status 200	12	When method call
4	When status 200	13	When method call
5	When status 200	14	When method call
6	When status 200	15	When method call
7	When status 200	16	When method call
8	When status 200	17	When method call
9	When status 200	18	When method call

In this example the correspondent Junit report is as below:

## JUnit Report

```
<testsuite failures="0" name="examples/DummyUsers/dummyusers.feature"
skipped="0" tests="2" time="4.30768"><testcase classname="examples.
DummyUsers.dummyusers" name="[1:6] get all dummy users and then get the
first user by id" time="3.102637"><system-out>* url 'http://dummy.
restapiexample.com/api/v1/' ..... passed
Given path 'employees'
..... passed
When method get
..... passed
Then status 200
..... passed
* def first = response.data[0]
..... passed
Given path 'employee', first.id
..... passed
When method get
..... passed
Then status 200
..... passed
</system-out></testcase>
<testcase classname="examples.DummyUsers.dummyusers" name="[2:17] create a
dummy user and then get it by id" time="1.205043"><system-out>* url
'http://dummy.restapiexample.com/api/v1/' .....
passed
* def user =
..... passed
Given url 'http://dummy.restapiexample.com/api/v1/create'
..... passed
And request user
..... passed
When method post
..... passed
Then status 200
..... passed
* def id = response.data.id
..... passed
* print 'created id is: ', id
..... passed
Given path id
..... passed
</system-out></testcase>
</testsuite>
```

If you have more than one feature file there will be one JUnit report per feature file.



A new version of Karate is about to be released where the testcase name will not have the order of the scenario and line.

A release candidate with the change is already available for you to experiment: <https://search.maven.org/artifact/com.intuit.karate/karate-core/1.2.0.RC4/jar>.

With the next official release the next step will not be needed and can be skipped.

Notice that the JUnit report generated with Karate joins, in the name of the testcase, the order of the scenario and line: "[1:6]" concatenated with the testcase name. In Xray we are using the testcase path+name to uniquely identify the test each time the result is uploaded, in this case if the line changes (due to some edition of the file thus changing the line of the code) Xray will create a new test (with this new name) instead of uploading the results to the previously created one.

We advise you to use the tool available in <https://github.com/bitcoder/junit-processor> to remove the characters from the testcase name, this tool have a patch exactly to remove that from the JUnit report generated. To use it you just have to run the following command:

```
junit-processor -p 1 examples.DummyUsers.dummyusers.xml
```

This will produce a new file called "junit-new.xml" that you can use to upload to Xray.

## Integrating with Xray

As we saw in the above example, where we are producing a JUnit report with the result of the tests, we need to import those results to your Jira instance, this can be done by simply submitting automation results to Xray through the REST API, by using one of the available CI/CD plugins (e.g. for Jenkins) or using the Jira interface to do so.

## API

### API

Once you have the report file available you can upload it to Xray through a request to the [REST API endpoint for JUnit](#).

### JUnit XML results

We will do a request to the API with the definition of some common fields on the Test Execution, such as the target project, test plan, etc.

```
curl -H "Content-Type: multipart/form-data" -u USERNAME:USER_PASSWORD -F "file=@junit-new.xml" http://yourserver/rest/raven/1.0/import/execution/junit?projectKey=COM&testPlanKey=COM-104
```

With this command we are creating a new Test Execution in the referred Test Plan with a generic summary and two tests with a summary based on the test name.

The screenshot shows the Jira Xray interface for a Test Plan named 'tutorial-java-karate' (ID: COM-104). The interface includes tabs for Details, Description, and Tests. The Tests tab is active, showing a summary of 2 tests passed. Below the summary, there is a table listing the tests.

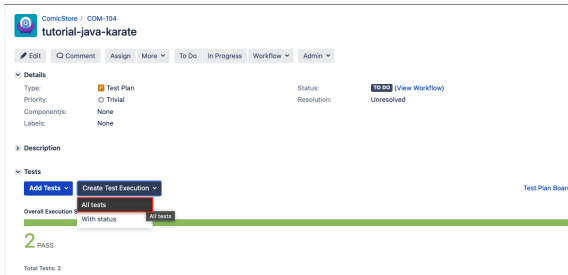
Summary	Requirements	#Test Executions	Issue Assignee	Component(s)	Begin Date	End Date	Test Plan	File Version(s)	Latest Status
get all dummy users and then get the first user by id		1	Administrator				None		PASS
create a dummy user and then get it by id		1	Administrator				None		PASS

## Jira UI

## Jira UI

1

Create a Test Execution for the tests that you have



2

Fill in the necessary fields and press "Create"

Create new test execution for tests in test plan COM-104

Project\* **ComicStore**

Summary\* **Test Execution for Test Plan COM-104**

Assignee **Administrator**

Priority **Blocker**

Fix Version/s

Sprint

Test Environments

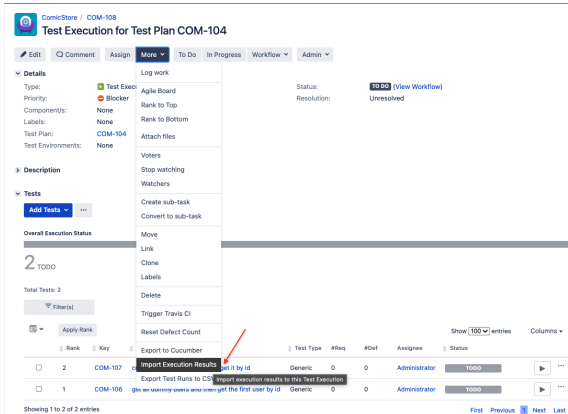
Revision

☒ Redirect to Test Execution

**Create** **Cancel**

3

Open the Test Execution and import the JUnit report





4

Choose the results file and press "Import"

### Import Execution Results

No file chosen

The file with the execution results for the Test Execution.

5

The Test Execution is now updated with the test results imported

ComicStore / COM-108

### Test Execution for Test Plan COM-104

[Edit](#)
[Comment](#)
[Assign](#)
[More](#)
[To Do](#)
[In Progress](#)
[Workflow](#)
[Admin](#)

**Details**

Test Execution

Blocker

Component(s):

None

Labels:

None

Test Plan:

COM-104

Test Environments:

None

Status:

Unresolved

Resolution:

Unresolved

Description

Tests

Overall Execution Status

2

PASS

Total Tests: 2

Rank	Key	Summary	Test Type	#Req	#Def	Assignee	Status
2	COM-107	create a dummy user and then get it by id	Generic	0	0	Administrator	PASS
1	COM-106	get all dummy users and then get the first user by id	Generic	0	0	Administrator	PASS

Showing 1 to 2 of 2 entries

[First](#)
[Previous](#)
[Next](#)
[Last](#)

Tests implemented will have a corresponding Test entity in Xray. Once results are uploaded, Test issues corresponding to the tests are auto-provisioned, unless they already exist.

ComicStore / COM-107

### create a dummy user and then get it by id

[Edit](#)
[Comment](#)
[Assign](#)
[More](#)
[To Do](#)
[In Progress](#)
[Workflow](#)
[Admin](#)

**Details**

Test

Trivial

Priority:

None

Component(s):

None

Labels:

None

Status:

Unresolved

Resolution:

Unresolved

Description

Test Details

Type:

Generic

Definition:

examples.DummyUsers.dummyusers.create a dummy user and then get it by id

Pre-Conditions

This test is not associated with Pre-Conditions yet.

Test Sets

This test is not associated with Test Sets yet.

Xray uses a concatenation of the suite name and the test name as the the unique identifier for the test.

In Xray, results are stored in a Test Execution, usually a new one. The Test Execution contains a Test Run per each test that was executed using playwright-test runner.

ComicStore / COM-108

## Test Execution for Test Plan COM-104

Edit Comment Assign More To Do In Progress Workflow Admin

**Details**

Type: Test Execution Status: 100% (View Workflow)  
 Priority: Blocker Resolution: Unresolved  
 Component(s): None  
 Labels: None  
 Test Plan: COM-104  
 Test Environments: None

**Description**

**Tests**

Add Tests +

Overall Execution Status

2 PASS

Total Tests: 2

Filter(s)

Apply Rank

Rank Key Summary Test Type #Req #Def Assignee Status

2	COM-107	create a dummy user and then get it by id	Generic	0	0	Administrator	PASS
1	COM-108	get all dummy users and then get the first user by id	Generic	0	0	Administrator	PASS

Showing 1 to 2 of 2 entries First Previous Next Last

Detailed results, including logs and exceptions reported during execution of the test, can be seen on the execution screen details of each Test Run, accessible through the *Execution details*:

ComicStore / COM-108

## Test Execution for Test Plan COM-104

Edit Comment Assign More To Do In Progress Workflow Admin

**Details**

Type: Test Execution Status: 100% (View Workflow)  
 Priority: Blocker Resolution: Unresolved  
 Component(s): None  
 Labels: None  
 Test Plan: COM-104  
 Test Environments: None

**Description**

**Tests**

Add Tests +

Overall Execution Status

2 PASS

Total Tests: 2

Filter(s)

Apply Rank

Rank Key Summary Test Type #Req #Def Assignee Status

2	COM-107	create a dummy user and then get it by id	Generic	0	0	Administrator	PASS
1	COM-108	get all dummy users and then get the first user by id	Generic	0	0	Administrator	PASS

Showing 1 to 2 of 2 entries First Previous Next Last

**Execution Details**

EXECUTE WITH EXPANDED EXECUTION DETAILS

EXECUTE ONLINE

TODO

EXECUTING

As we can see here:

ComicStore / Test Plan COM-104 / Test Execution COM-108 / Test COM-107

create a dummy user and then get it by id

Report Test as Fail Add Notes to Test Execution Execute with Expansive Fail + Previous

Execution Status: PASS

Started On: 2024-02-02 03:03 PM Finished On: 2024-02-02 03:03 PM

Assignee: Administrator  
 Executed By: Administrator  
 Tests: 1  
 Environment: None

**Comment**

**Test Details**

Execution Fields

There are no Test Run Custom Fields defined

**Test Description**

Test Type: Generic  
 Subtype: example/DummyUsersAndUsers/create a dummy user and then get it by id

**Results**

Context	Output	Location	Status
Test Results: example/DummyUsersAndUsers/create a dummy user and then get it by id	-	Test	PASS

**Activity**

## Tips

- after results are imported, in Jira Tests can be linked to existing requirements/user stories, so you can track the impacts on their coverage.
- results from multiple builds can be linked to an existing Test Plan, to facilitate the analysis of test result trends across builds.
- results can be associated with a Test Environment, in case you want to analyze coverage and test results per environment later on. A Test Environment can be a testing stage (e.g. dev, staging, pre-prod, prod) or an identifier of the device/application used to interact with the system (e.g. browser, mobile OS).

## References

- <https://karatelabs.github.io/karate/>
- <https://github.com/karatelabs/karate>
- <http://dummy.restapiexample.com/>
- <https://github.com/bitcoder/junit-processor>