KARATE FRAMEWORK

Actualización con servicios SOAP y REST

Contenido

En este documento encontraras una guía práctica para poder desarrollar unas destrezas iniciales en la aplicación de Karate a pruebas de aceptación, tanto en servicios SOAP, como en servicios REST.

¿Qué es Karate?

Karate es una herramienta de código abierto que combina la automatización de pruebas de API, mocks, pruebas de rendimiento e incluso la automatización de la interfaz de usuario en un marco único y unificado.

(Karate)

Algunas características

- Basa sus pruebas en sintaxis Gherkin de BDD (Cucumber).
- No depende de ningún lenguaje de programación de propósito general.
- Utiliza DSL para describir las pruebas de API basadas en HTTP.
- Reportes entendibles por cualquier actor que intervenga en las pruebas, tanto de negocio, técnico y líder.
- Karate se ejecuta sobre la JVM
- No requiere implementar definiciones de pasos adicionales, ni código extra.
- JSON está embebido en la sintaxis.

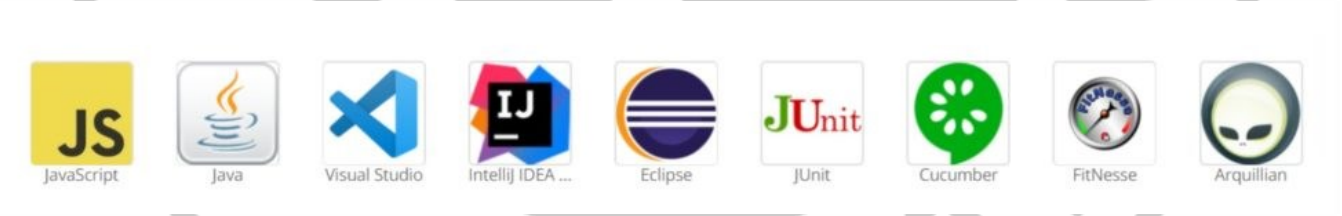


Figura 1 Lenguajes y herramientas con las que se integra Karate

¿Dónde lo puedes aplicar?

Casos de uso habituales

Karate es una buena opción cuando se desee disponer de una suite automatizada de pruebas sobre servicios REST o SOAP, y no se precisen acciones excesivamente complejas para realizar dichas comprobaciones.

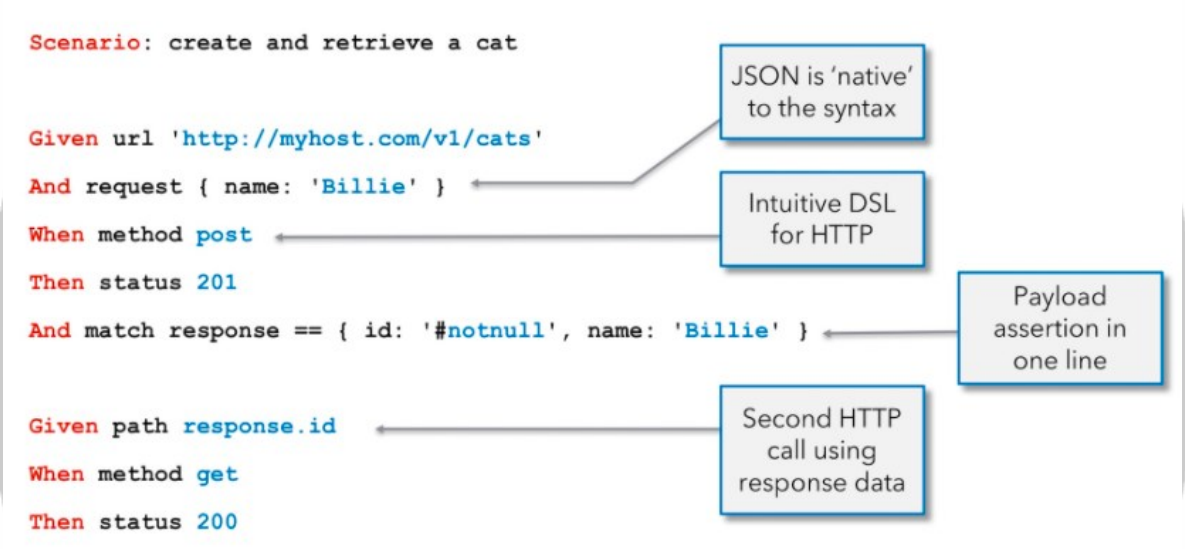
Es decir, por su sencillez y facilidad de uso, resulta ideal para disponer rápidamente de una batería completa de pruebas sin apenas conocimientos de programación y utilizando un

lenguaje común que puede ser comprendido tanto por la parte técnica como la de negocio. Esto hace que las pruebas sirvan a su vez como documentación funcional del API sobre el que se lanzan las pruebas (concepto de Living documentation).

Donde no es recomendable

En casos donde para la correcta ejecución de las pruebas se requiera un setup complejo de datos o del entorno puede no resultar una buena opción. No tanto porque desde Karate no se puedan realizar dichas acciones, pero a costa de perder su potencial de claridad y sencillez.

Estructura de una feature con Karate



Estructura de paquetes recomendada →

```
src/test/java
    |
    +-- karate-config.js
    +-- logback-test.xml
    +-- some-reusable.feature
    +-- some-classpath-function.js
    +-- some-classpath-payload.json
    |
    \-- animals
        |
        +-- AnimalsTest.java
        |
        +-- cats
        |   |
        |   +-- cats-post.feature
        |   +-- cats-get.feature
        |   +-- cat.json
        |   \-- CatsRunner.java
        |
        \-- dogs
            |
            +-- dog-crud.feature
            +-- dog.json
            +-- some-helper-function.js
            \-- DogsRunner.java
```

Creando la estructura básica

En nuestro proyecto y antes de iniciar con los casos prácticos debemos tener una configuración preliminar, como la modificación de nuestro archivo build.gradle, nuestro runner y la ruta de paquetes o directorios del proyecto

- El archivo build.gradle debemos de adecuarlo para que pueda ejecutar pruebas con Karate, de esta forma la configuración inicial recomendada es la siguiente:

```
plugins {
    id 'java'
}

ext {
    karateVersion = '1.2.0.RC3'
}

compileJava {
    sourceCompatibility = 1.8
    targetCompatibility = 1.8
}

dependencies {
    implementation "com.intuit.karate:karate-junit5:${karateVersion}"
    implementation 'net.masterthought:cucumber-reporting:5.6.1'
}

sourceSets {
    test {
        resources {
            srcDir file('src/test/java')
            exclude '**/*.java'
        }
    }
}

test {
    useJUnitPlatform() // junit5
    systemProperty "karate.env", System.properties.getProperty("karate.env")
    outputs.upToDateWhen { false }
}

repositories {
    mavenCentral()
}
```

En la sección de “repositories” usted puede agregar la url de su preferencia, por ejemplo “repositories” debería quedar así:

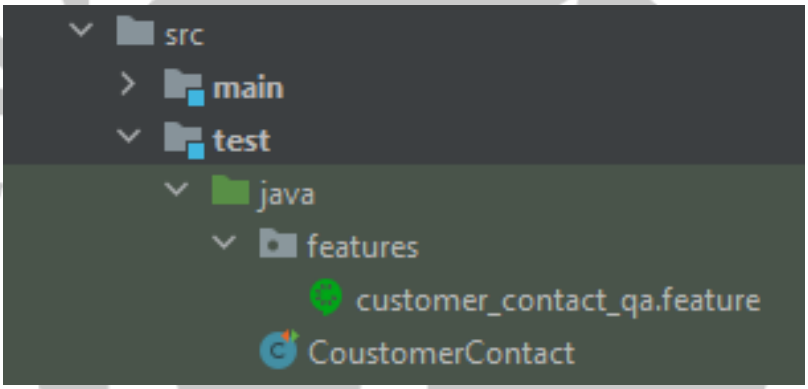
```
repositories {
    mavenCentral()
```

```
    maven {
        url 'https://artifactory.apps.bancolombia.com/maven-bancolombia'
    }
}
```

La versión recomendada para karate la encontramos en:

```
ext {
   karateVersion = '1.2.0.RC3'
}
```

- Comenzamos con una ruta de directorios y paquetes lo más sencilla posible, como se ve en la imagen. La complejidad de los directorios dependerá de cada proyecto y el fin que le queramos dar. Luego de crear el proyecto en **IntelliJ**, en la ruta src/test/java vamos a crear nuestra .feature y nuestro runner



- Por ultimo debemos configurar el runner de nuestra automatización. Como vamos a generar reportes con cucumber, la configuración del runner debe quedar de la siguiente forma:

```java
public class CoustomerContact {
    @Test
    public void testParallel() {
        Results results = Runner.path( ...paths: "classpath:features/customer_contact_qa.feature")
                .outputCucumberJson(true)
                .tags("~@ignore")
                .parallel( threadCount: 4);
        generateReport(results.getReportDir());
        Assertions.assertTrue( condition: results.getFailCount() == 0, results.getErrorMessages());
    }

    public static void generateReport(String karateOutputPath) {
        Collection<File> jsonFiles = FileUtils.listFiles(new File(karateOutputPath), new String[]{"json"},  recursive: true);
        List<String> jsonPaths = new ArrayList<~>(jsonFiles.size());
        jsonFiles.forEach(file -> jsonPaths.add(file.getAbsolutePath()));
        Configuration config = new Configuration(new File( pathname: "target"),  projectName: "pruebas customer contact");
        ReportBuilder reportBuilder = new ReportBuilder(jsonPaths, config);
        reportBuilder.generateReports();
    }
}
```

Al método le podemos asignar cualquier nombre, las especificaciones siguientes te ayudaran a entender el propósito de las funciones **testParallel()** y **generateReport()**:

En **testParallel():**

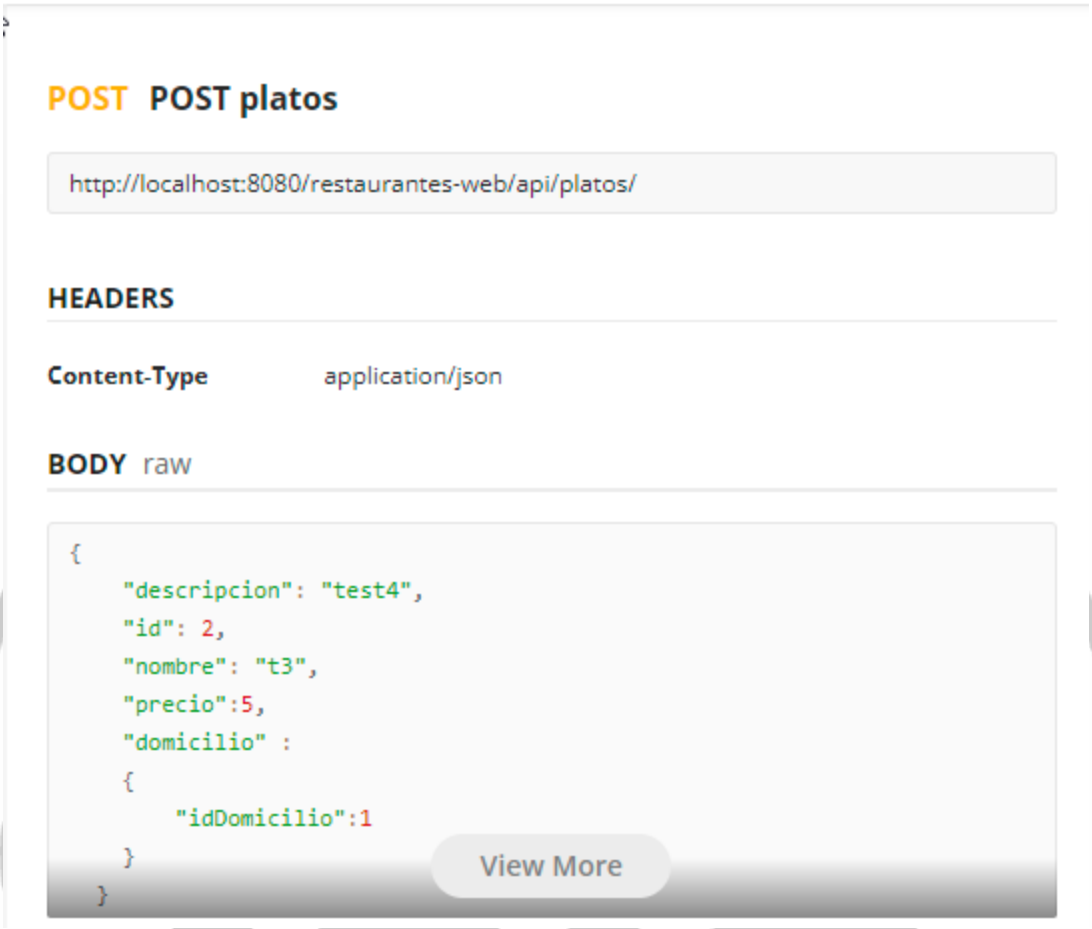
- **Runner.path**("<le especificamos el classpath o ruta donde tenemos nuestra **.feature**> ")
- **.tags**(“<agregamos el tag del escenario construido en nuestro **.feature**>”)
- **.parallel(4)** especifica cuantas pruebas van a ejecutarse en paralelo

En **generateReport():**

- La línea de código: **Collection<File> jsonFiles = FileUtils.listFiles(new File(karateOutputPath), new String[]{"json"}, true);** debemos dejarla tal cual, dado que esta se encarga de transformar las salidas o resultados en formato JSON
- En la línea de código: **Configuration config = new Configuration(new File("target"), "pruebas customer contact");** nosotros establecemos el nombre del directorio de salida de los resultados y el título del reporte que contendrá esos resultados, en este caso el directorio se llama **target** y el resultado se titula **pruebas customer contact**

Caso práctico con servicios REST

Suponemos que vamos a automatizar una petición a una API, ese api trabaja con el método post y tiene unas ciertas características que hemos logrado mapear en postman, como se muestra en la imagen:



URL de la imagen: https://documenter.getpostman.com/view/342728/pruebas-postman-platos/2PMCaL#a1a0ca95-c8ac-b42f-8368-789e2c304cd4

Entonces, crearemos nuestro escenario en el archivo **.feature,** que ya especificamos en la ruta definida al comenzar este documento, donde nuestra **.feature** debería lucir similar a la siguiente, para eso la dividimos en dos partes: Background y Scenario

```
Feature: test the response of the API
  Background:
    * def urlBase = ' http://localhost:8080'
    * def usersPath = '/restaurantes-web/api/platos/'
    * def req = {"descripcion": "test4", "id": 2, "nombre": "t3", "precio":5,
"domicilio" : {"idDomicilio":1 } }

  @Scenario1
  Scenario: get the status service
    * configure ssl = true
    * configure headers = {'Content-Type': 'application/json' }
    Given url urlBase + usersPath
    And request req
    When method POST
    * print response
    Then match response.status == 200
```

Tal y como está constituido el escenario hemos realizado una migración de un consumo mediante postman a un consumo usando Karate.

Existen factores adicionales como lo son:

* configure ssl = true

* configure headers = { 'Content-Type': 'application/json' }

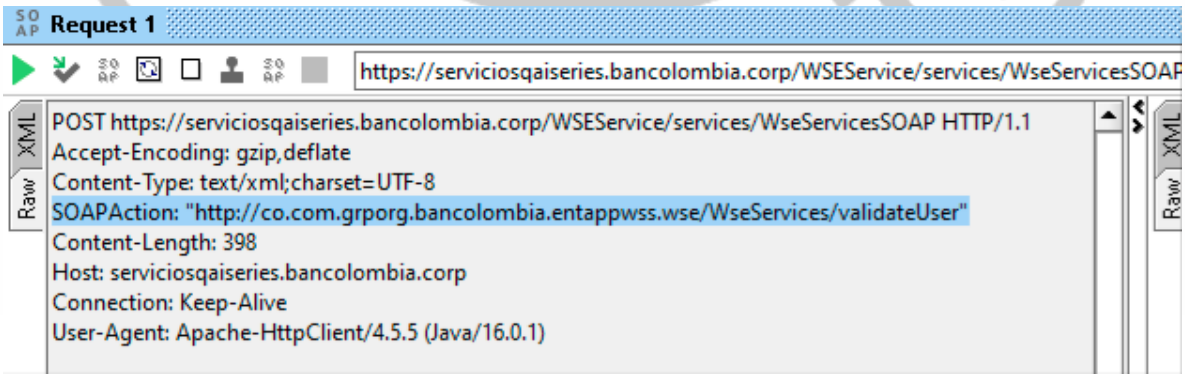
Estos factores habilitan la configuración para aceptar las características de certificados seguros y la parte de **headers** nos permite enviar los **headers** que tenemos mapeados para el funcionamiento del consumo. En headers podemos enviar las credenciales u otros elementos de conexión con el API o servicio REST.

Caso práctico con servicios SOAP

Para los servicios Soap, el escenario de la **.feature** no difiere mucho, una de las cosas que cambian es que tenemos la posibilidad de usar dos métodos de consumo: **POST** y **soap action.** Al igual que en el ejemplo de servicios REST, en esta parte tanto los headers y el aceptar certificados seguros deben de estar habilitados.

Antes de comenzar debemos mapear varias cosas, una de ellas es la URL de consumo, los headers a usar, el método que acepta y el request. Donde encontramos esos elementos:

- Request: lo podemos localizar en el body de nuestro consumo soap, si lo tenemos configurado en SoapUI
- Headers: aspecto importante, siempre vamos a configurar el header para que acepte un Content-Type = 'text/xml; charset=utf-8'
- Localizar el método de petición: por lo general los consumos a servicios soap agregan un soap action y lo podemos ver en el request de SoapUI



El siguiente ejemplo de .feature es referente a la anterior imagen, donde usamos un método soap action:

```
Feature: SOAP testing using Karate

  Background:
    * configure ssl = true
    * url 'https://serviciosqaiseries.bancolombia.corp/WSEService/services/WseServicesSOAP'

  @FirstRequest
  Scenario: Test SOAP request and assert the response
    Given request
    """
    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:wse="http://wse.entappwss.bancolombia.grporg.com.co">
     <soapenv:Header/>
     <soapenv:Body>
      <wse:validateUser>
     <login>KAAGOMEZ</login>
        <password>xxxxx</password>
        <application>WSE</application>
      </wse:validateUser>
    </soapenv:Body>
    </soapenv:Envelope>

    """
    And header Content-Type = 'text/xml; charset=utf-8'
    When soap action "http://co.com.grporg.bancolombia.entappwss.wse/WseServices/validateUser"
    Then status 200
```

Nótese en la anterior imagen el cambio en como definimos la URL del servicio a consumir, a diferencia del servicio REST, ya no definimos una urlBase, ahora bien, la configuración de un escenario con servicios SOAP requiere de un request tipo XML.

El siguiente ejemplo suma dos números y lo hemos adecuado para que reciba configuraciones desde un karate-config.js, este ejemplo utiliza el método de petición POST, puedes replicarlo configurando lo siguiente:

- Creamos en la ruta del runner (siguiendo la ruta básica src/test/java), el archivo de karate-config.js, en este archivo vamos a colocar lo siguiente:

```
function fn() {
    karate.configure('connectTimeout', 10000);
    karate.configure('readTimeout', 10000);
    karate.configure ('ssl', true);
    var env = karate.env; // get java system property 'karate.env' in
build.gradle
    var account = env == 'dev' ? '360735510274' : '278078741213';
    karate.log('karate.env system property was:', env);

    return {
            numbers: {
              numone: 2,
              numtwo: 3
            }
    };
}
```

La función anterior tiene dos números quemados, que bien, los podríamos enviar por comandos de ejecución a través de la terminal o como variables de entorno, en este

caso no vamos a explorar este archivo, pero encontraras información en el siguiente enlace: [karate configuration]

- Nuestro archivo **.feature** tendrá la siguiente estructura:

```
Feature: test soap end point

  Background:
    #* url demoBaseUrl + '/soap'
    # this live url should work if you want to try this on your own
    * url 'http://www.dneonline.com/calculator.asmx'

  @Soap11
  Scenario: soap 1.1
    * def one = numbers.numone
    * def two = numbers.numtwo
    Given request
    """
    <?xml version="1.0" encoding="utf-8"?>
    <soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
      <soap:Body>
        <Add xmlns="http://tempuri.org/">
          <intA>#(one)</intA>
          <intB>#(two)</intB>
        </Add>
      </soap:Body>
    </soap:Envelope>
    """
    And header Content-Type = 'text/xml; charset=utf-8'
    When method post
    Then status 200
    And def num = /Envelope/Body/AddResponse/AddResult
    And match /Envelope/Body/AddResponse/AddResult == '5'
    And print 'numero: ', num
```

En este ejemplo hemos agregado varias cosas, entre las cuales:

- Al utilizar `* def one = numbers.numone` estamos definiendo la variable tomando el valor que recibimos desde el archivo karate-config.js, recuerda ampliar la información en la página oficial.
- Cuando escribimos `#(one)` en la parte del request XML, estamos agregando el valor contenido en la variable definida. Es un ejemplo claro de cómo concatenar los valores.
- Hemos agregado un match, es la forma de hacer aserciones en karate. En el caso de soap, la aserción queda dad así: `And match /Envelope/Body/AddResponse/AddResult == '5'`

¿Qué diferencia existe respecto a un match a servicios REST?: en servicios rest nosotros capturamos la respuesta usando el valor propio “response”, y podemos obtener más cosas con solo usar punto (“**.**”), ejemplo: response.status