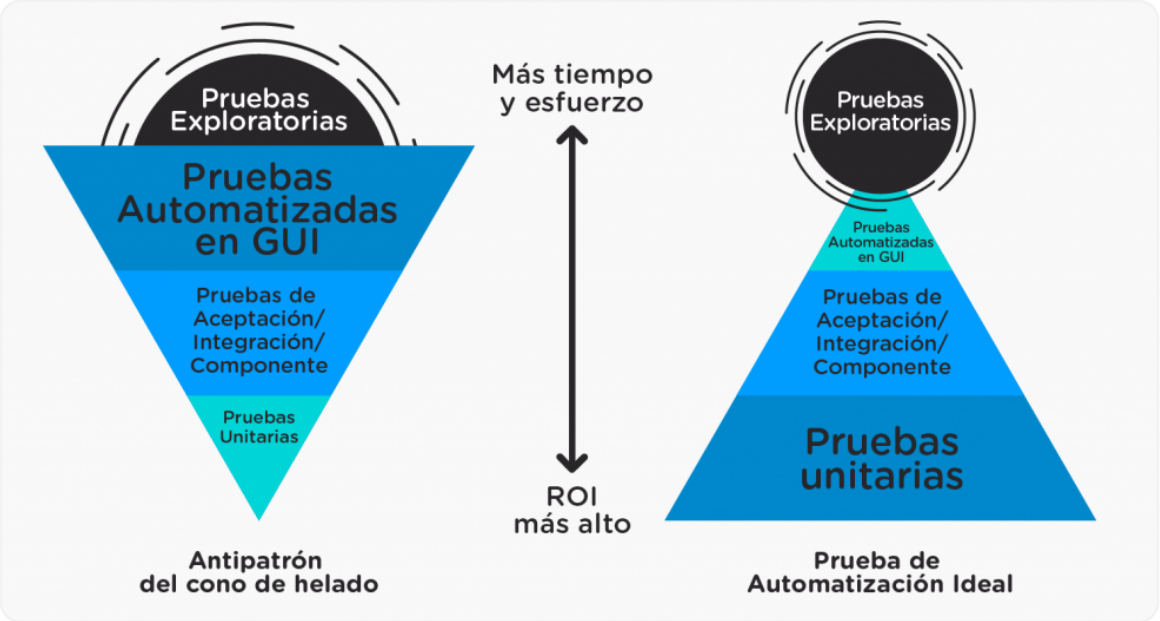
# Iniciación en pruebas de aceptación




## Setup inicial

Otra forma de instalación sobre un proyecto nuevo sería usando el artefacto que proporciona Maven. Ejecutando el siguiente comando en un terminal:

```
mvn archetype:generate \
-DarchetypeGroupId=com.intuit.karate \
-DarchetypeArtifactId=karate-archetype \
-DarchetypeVersion=1.1.0 \
-DgroupId=co.com.sofka \
-DartifactId=myproject
```

## Casos de uso principales

Para aplicar los casos de test se usará la web de prueba https://reqres.in/, la cual proporciona diversos endpoints sobre sobre los que se pueden realizar peticiones GET, PUT, POST y DELETE.

Sobre ella se aplicarán los ejemplos más comunes de pruebas utilizados con este framework.

Para el diseño de los test se crearán tantos archivos ".feature" como se necesite. En estos archivos se escribe la colección de escenarios que tienen como objetivo probar una funcionalidad en concreto.

La especificación básica de la pruebas se basa en el **lenguaje Gherkin**:

```
Scenario:
Given
When
And
Then
```

Para ir viendo las principales funcionalidades de Karate se irán recorriendo los casos de uso más habituales:

**GET**

La prueba consiste en obtener un listado de los usuarios del sistema y comprobar que la respuesta es correcta.

```
Feature: Get Tests on reqres.in
Scenario: Get users list
Given url 'https://reqres.in/api/users?page=2'
When method GET
Then status 200
```

Como se puede observar, existen definiciones constantes como por ejemplo las url que se van a volver a usar en más de una ocasión y que se pueden sacar y definirlas para su uso en toda la feature en el apartado “Background”. Para ello se usa la palabra clave en karate “def”.

```
Feature: Get Tests on reqres.in
Background:
* def urlBase = 'https://reqres.in'
* def usersPath = '/api/users'
Scenario: Get users list
Given url urlBase + usersPath + '?page=2'
When method GET
Then status 200
```

Por ejemplo, conocer si uno de los usuarios de la lista se llama “Emma” y que los identificadores de todos los usuarios no sean “nulos”. Se puede ampliar esta prueba añadiendo además comprobaciones adicionales sobre el contenido de la respuesta obtenida.

```
Feature: Get Tests on reqres.in
Background:
* def urlBase = 'https://reqres.in'
* def usersPath = '/api/users'
Scenario: Get users list and check value in field
Given url urlBase + usersPath
When method GET
Then status 200
```

```
And match $..first_name contains 'Emma'
And match $..id contains '#notnull'
```

Los principales operadores que se usan en este tipo de expresiones son: Para realizar los asertos se usa la palabra clave “match”. Con ella se comprueba que una expresión se evalúa como true, utilizando el motor de JavaScript incorporado.

- match ==
- match !=
- match contains, match contains only, match contains any y match !contains

La expresión utilizada “$” equivale al cuerpo de la respuesta de la llamada que se haya realizado. “$” sería el equivalente a “response”. Cuando se añaden los dos puntos, como muestra el ejemplo, se buscará ese atributo en cualquier nivel de profundidad del cuerpo de la respuesta.

Para las comprobaciones genéricas se pueden utilizar distintos marcadores como son:

#ignore: Ignora el campo.
#null: El valor debe ser nulo.
#notnull: El valor no debe ser nulo.
#array: El valor debe ser una matriz JSON.
#object: El valor debe ser un objeto JSON.
#boolean: El valor debe ser verdadero o falso.
#number: El valor debe ser un número.
#string: El valor debe ser una cadena.
#uuid: El valor debe coincidir con el formato UUID.
#regex: El valor coincide con una expresión regular.
#? EX: La expresión de Javascript EX debe evaluarse como verdadera.

**POST**

Este test verifica la creación de un usuario, por lo que debemos añadir un “BODY” a la petición.
En este caso, veremos cómo conseguirlo haciendo uso además del concepto de “Scenario Outline” para parametrizar tanto la petición como los resultados esperados en cada caso.
Esta forma de realizar pruebas resulta muy útil cuando deseamos ejercitar una misma funcionalidad bajo diferentes casuísticas.

```
Feature: Login and register Tests on reqres.in
Background:
* def urlBase = 'https://reqres.in'
* def loginPath = '/api/login'
Scenario Outline: As a <description>, I want to get the corresponding response_code
<status_code>
```

```
Given url urlBase + loginPath
And request { 'email': <username> , 'password': <password> }
When method POST
* print response
Then response.status == <status_code>
Examples:
|username |password | status_code | description |
|'eve.holt@reqres.in' |'cityslicka' | 200 | valid user |
|'eve.holt@reqres.in' |null | 400 | invalid user|
```

En esta definición de test se pueden ver varios elementos a destacar como son:

- parámetros de sustitución o placeholders delimitados por < >.
- tabla de ejemplos de prueba o Examples.

Otra expresión muy útil que favorece las tareas de depuración es “print”. Se usa para imprimir por consola el valor de las variables en medio de un script. En este caso, si se necesita saber la respuesta completa para conocer los atributos que se pueden después comprobar, se incluye en la definición y tendría la siguiente salida por consola:

```
08:12:19.298 [ForkJoinPool-1-worker-1] INFO com.intuit.karate - [print] {
"token": "QpwL5tke4Pnpja7X4"
}
```

**PUT**

El siguiente test actualiza el valor de un atributo de un usuario y se comprueba que efectivamente se ha modificado.

```
Feature: Post/Put/Patch/Delete Tests on reqres.in
Background:
* def urlBase = 'https://reqres.in'
* def usersPath = '/api/users'
Scenario: Put user
Given url urlBase + usersPath + '/2'
And request { name: 'morpheus updated',job: 'leader updated' }
When method PUT
Then status 200
And match $.name == 'morpheus updated'
```

En esta definición se comprueba que en Karate el lenguaje JSON es un tipo nativo. Ésto quiere decir que se puede insertar en cualquier parte de la definición sin tener que preocuparse por encerrarlos en cadenas o “escapar” con comillas. Además, se puede observar que ni siquiera hay que poner comillas dobles a las claves ya que el analizador se encarga de procesarlas.
Al igual que se usan como datos de entrada las estructuras JSON, también se puede usar XML.

**REUTILIZACIÓN DE FEATURES**

Imaginemos que tras una prueba de creación de entidades (POST) se necesita que el objeto sea eliminado y esta funcionalidad va a ser requerida en más de una ocasión. En este caso resulta conveniente la reutilización de features.

Para ello, se crearán dos features reutilizables, una de creación y otra de eliminación, que al no ser una prueba en sí misma se debe anotar con @ignore de manera que cuando se realice la ejecución de los tests, éstos sean ignorados.

```
@ignore
Feature: Reusable Post Tests on reqres.in [post2.feature]
Background:
* def urlBase = 'https://reqres.in'
* def usersPath = '/api/users'
Scenario: Post user Data-Driven
Given url urlBase + usersPath
And request { name: '#(name)',job: '#(job)' }
When method POST
Then status 201
@ignore
Feature: Reusable Delete Tests on reqres.in [delete.feature]
Background:
* def urlBase = 'https://reqres.in'
* def usersPath = '/api/users'
Scenario: Delete user
* def path = urlBase + usersPath +'/' + id
Given  url path
When method DELETE
Then status 204
```

Ahora veamos el ejemplo donde se utilizan ambas mediante la instrucción “call”.

```
Feature: Reusable Tests on reqres.in
Scenario: call post and delete with reusable features and delete with conditional operation
* table users
|name    |job      |
|'max'   |'tester1' |
* def result = call read('post2.feature') users
* def id = result[0].response.id
* table ids
|id|
|id|
* def res = result.responseStatus == 201 ? {} : karate.call('delete.feature', ids)
```

Otra funcionalidad que se puede observar, además de la reutilización de features es el de las operaciones condicionales. En este ejemplo, sólo se realizará la eliminación si el código de respuesta de la operación anterior, el POST, es un código de éxito.

**EJECUCIÓN DE LAS PRUEBAS**

Para la ejecución de los tests se define una clase Java en la que se configura su rutina así como la ubicación de los features.

```
package examples;
import com.intuit.karate.KarateOptions;
import com.intuit.karate.junit4.Karate;
import org.junit.runner.RunWith;
@RunWith(Karate.class)
@KarateOptions(features = "classpath:examples", tags = "~@ignore")
class ExamplesRunner {
}
```

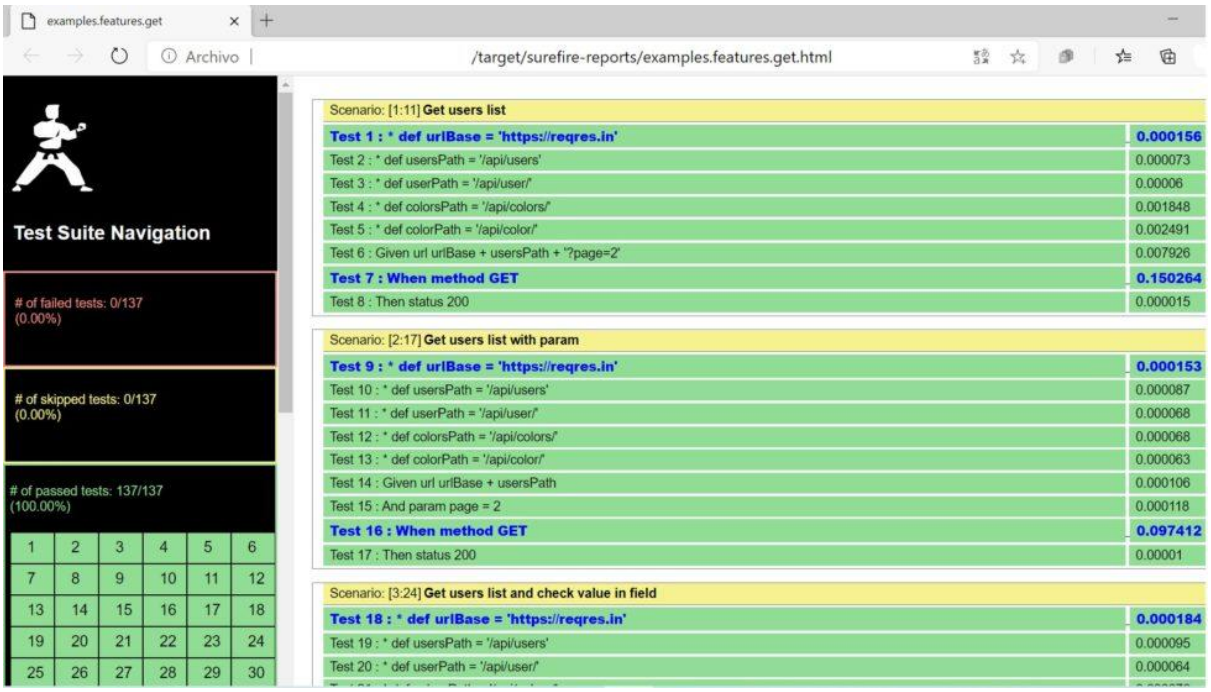
En dicha ejecución, Karate espera que exista un archivo “karate-config.js” con las variables de configuración. En el que se pueden definir variables como el environment o funciones JavaScript reutilizables.

Para lanzar la ejecución se puede hacer de dos formas: desde nuestro IDE o por línea de comandos. Para el segundo caso se podrá ejecutar el siguiente comando Maven:

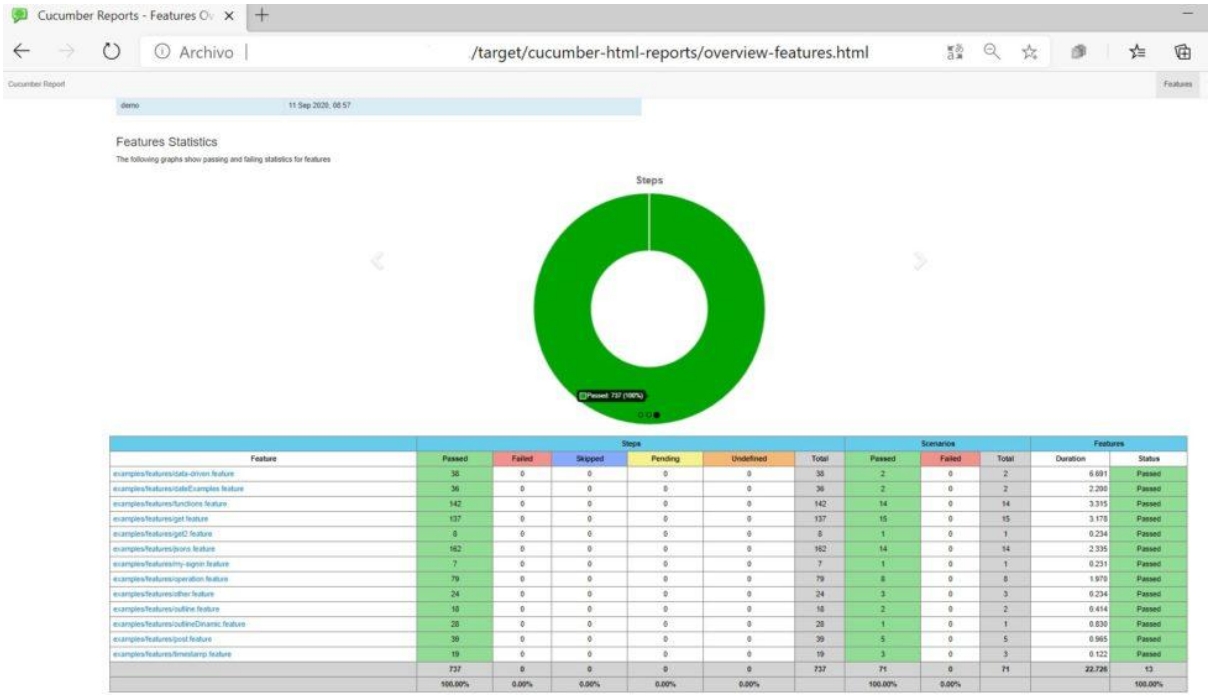
```
mvn clean test -Dtest=ExamplesRunner
```

**REPORTES**

Cuando se ejecutan los tests se genera un informe por cada feature. El informe propuesto por Karate está en formato html y su estructura es la siguiente:




Existe también la posibilidad de introducir una dependencia Maven con la que se pueden generar informes del mismo modo en el que lo hace Cucumber.




Estas son las principales funcionalidades de Karate pero no son las únicas ya que posee numerosas características que aplicar. Dentro del repositorio de la demo se encuentran algunas features con más ejemplos a las que no se han hecho referencia en este artículo pero que sería útil conocer.

## Ventajas

Una de las principales ventajas de esta herramientas es que es opensource, está en continuo desarrollo y tiene una amplia gama de ejemplos y documentación para multitud de casos de tests.
Además, dada su integración con Java es posible ampliar sus funcionalidad básicas.
Otra ventaja es la posibilidad de paralelizar la ejecución de los test ya que permite reducir enormemente el tiempo de pruebas.
Por otro lado, se trata de una herramientas que ya de por sí ofrece unos informes claros y legibles, no obstante también existen plugins disponibles en caso de que queramos disponer de informes con el mismo aspecto que los generados por Cucumber. Esto resultaría interesante en caso de que quedáramos buscando uniformidad en los informes de pruebas generados.
Otra ventaja importante es la posibilidad de crear features reutilizables entre escenarios, es la solución ideal para flujos de inicio de sesión o clases de utilidad.

## Inconvenientes

Tal y como se ha comentado anteriormente, para situación donde se requiera una preparación compleja del entornos o datos de pruebas antes de ser lanzadas puede no ser una buena opción. Puesto que, si bien es cierto que permite la integración con otras clases

Java y tiene un motor JavaScript incorporado para implementar funciones sencillas, desde nuestro punto de vista, esta manera de proceder no favorece la mantenibilidad de la suite de pruebas.

## Conclusiones

Una de las características que hace que utilizar BDD sea enriquecedor para los proyectos es que la parte de negocio escriba el Gherkin o que colaboren con el equipo de desarrollo para definirlo. En el caso de que la parte de negocio esté dispuesta a implicarse en dichas tareas, esta herramienta favorece la comunicación al utilizar un lenguaje de definición de alto nivel.
Precisamente, utilizar un lenguaje de alto nivel para definir las pruebas hace que estas sirvan a su vez como documentación funcional del propio API. Por tanto se obtiene un doble beneficio.
Por ello, consideramos que, desde el punto de vista de QA, la mayoría de las necesidades quedan cubiertas con el uso de Karate. Siendo una excelente, simple y eficaz herramienta para la realización de los test de integración de servicios web.
Bibliografía