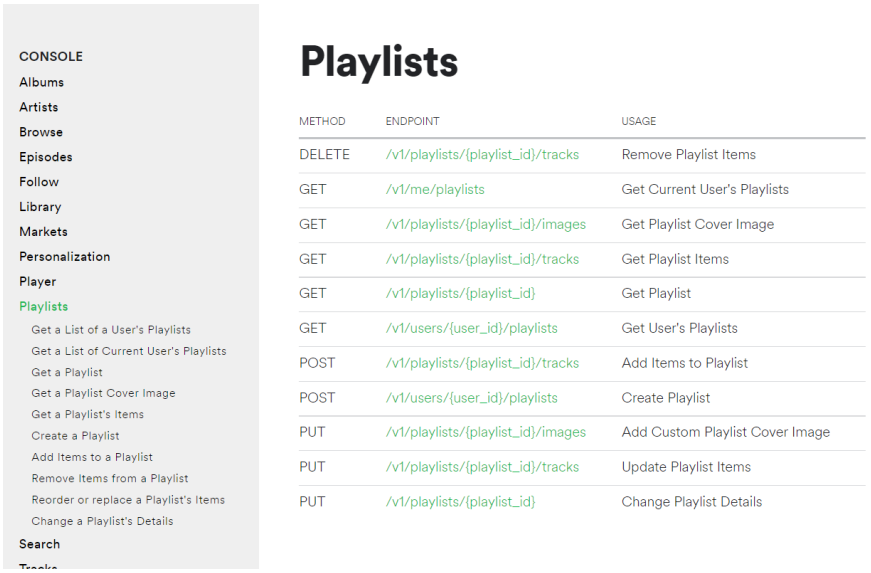
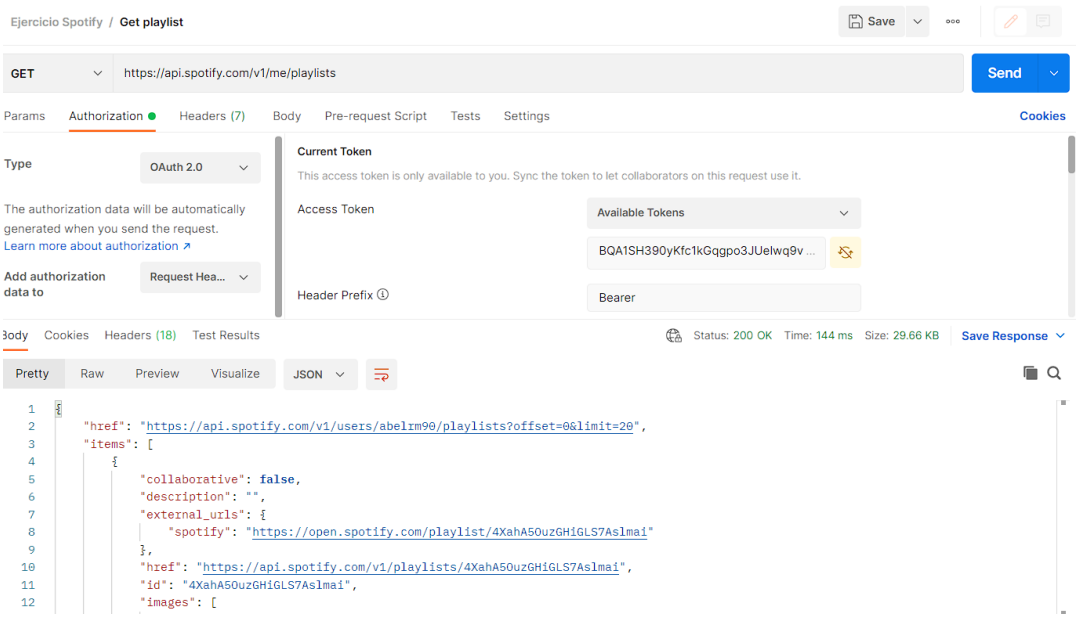
QA / TESTING - 09/05/2022 - Abel Rios

Vamos a ir probando los test con API. Para ello usamos de ejemplo la página de developer de spotify https://developer.spotify.com/console/

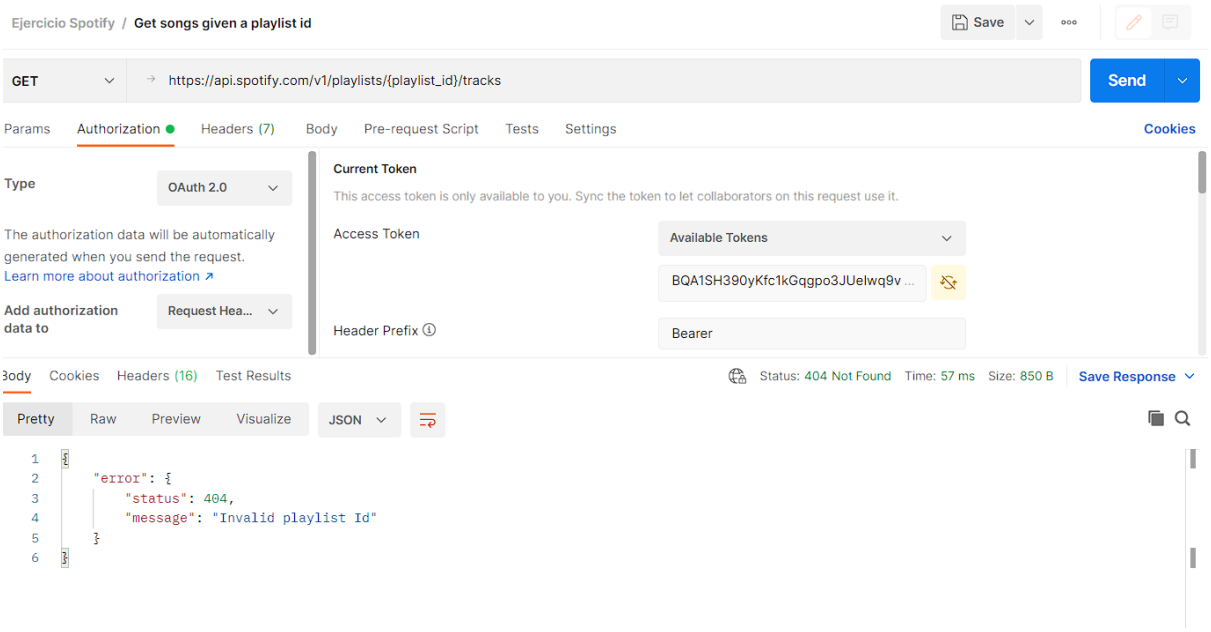


Abrimos el postman y creamos una nueva colección (Ejercicio Spotify). Una colección es simplemente una carpeta en la que almacenamos las diferentes requests de un proyecto o una api. Aquí lo vamos a utilizar para guardar todas las requests que haremos en el ejercicio practicando con la API de Spotify.

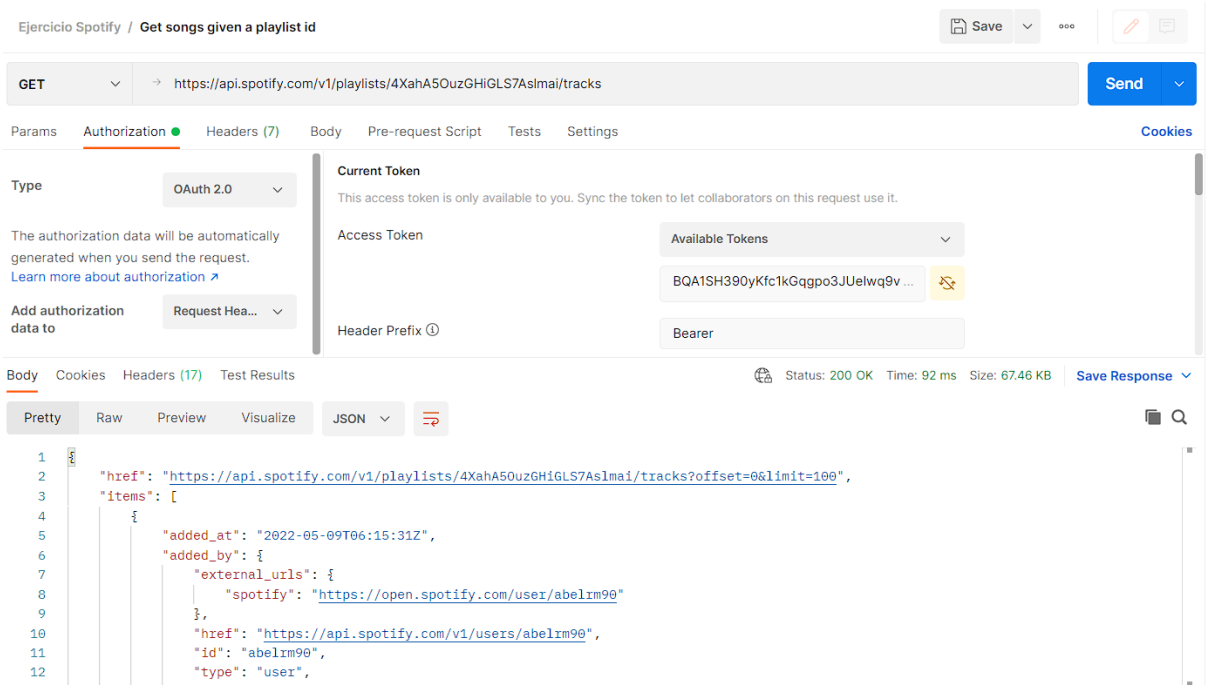
La primera va a ser de conseguir las playlists. En la página de spotify copiamos el enlace de la segunda request (get playlists). Si enviamos la request falla, pues no hemos enviado el token. En postman, en nuestra request de postman, vamos a “Authorization”, elegimos OAuth2.0, nos sale a la derecha una pestaña en la que podemos ingresar el token que hemos adquirido en la página de spotify iniciando nuestra sesión.



Ahora vamos a hacer otra request, para conseguir las canciones de una lista:



Nos vuelve a fallar pues necesita la playlist ID, de la request anterior podemos sacar la ID de cualquier lista, la insertamos en la nueva request y magia, funciona:



Ahora Fran nos ha pasado una carpeta de colección de requests sobre la que vamos a trabajar (Rick and Morty API).

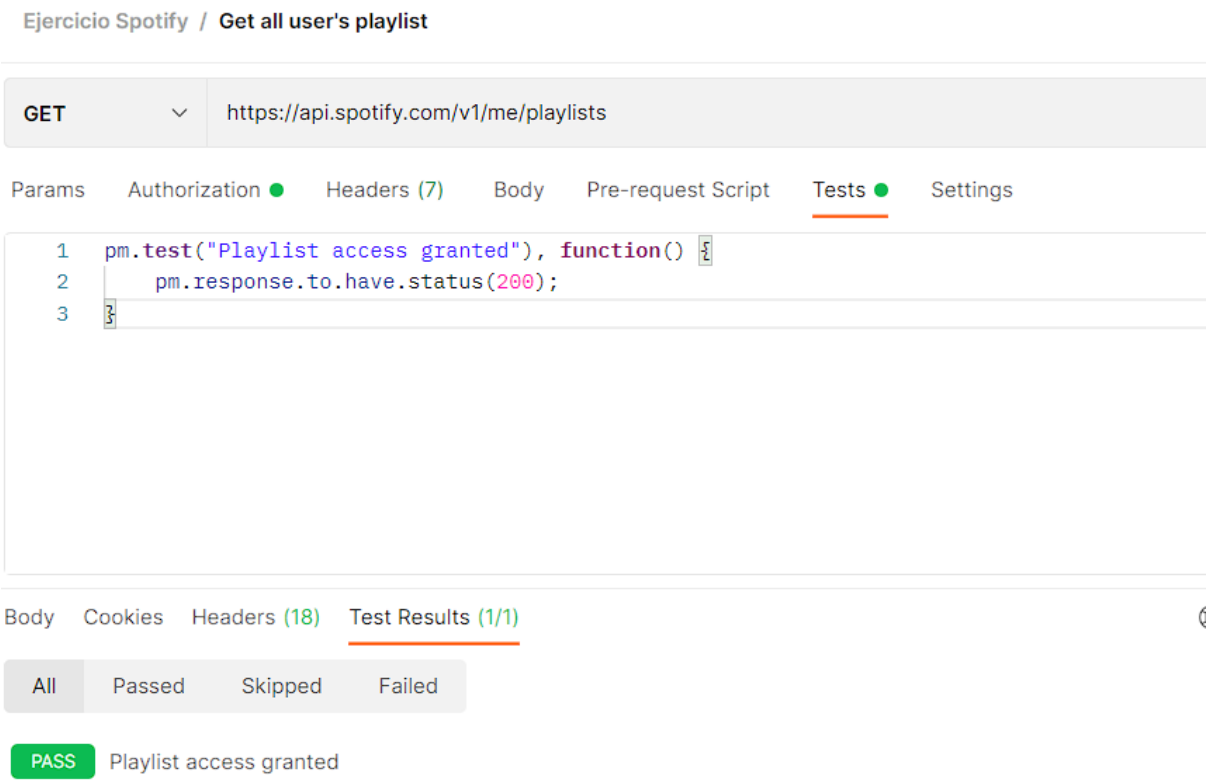
Postman nos permite crear **entornos**, que son eso, entornos en los que a uno o más parámetros se les da un valor y se queda ese valor guardado. Estos parámetros en postman van entre dos llaves {{}}. Se añaden en el icono del ojo (arriba a la derecha).

El último endpoint es un filtrado, en el que en la url está formado en querystring, y se puede rellenar simplemente uno de los valores y eliminar el resto y la api nos devuelve todos los resultados que coinciden en ese parámetro. En postman los test se denominan con **pm**.

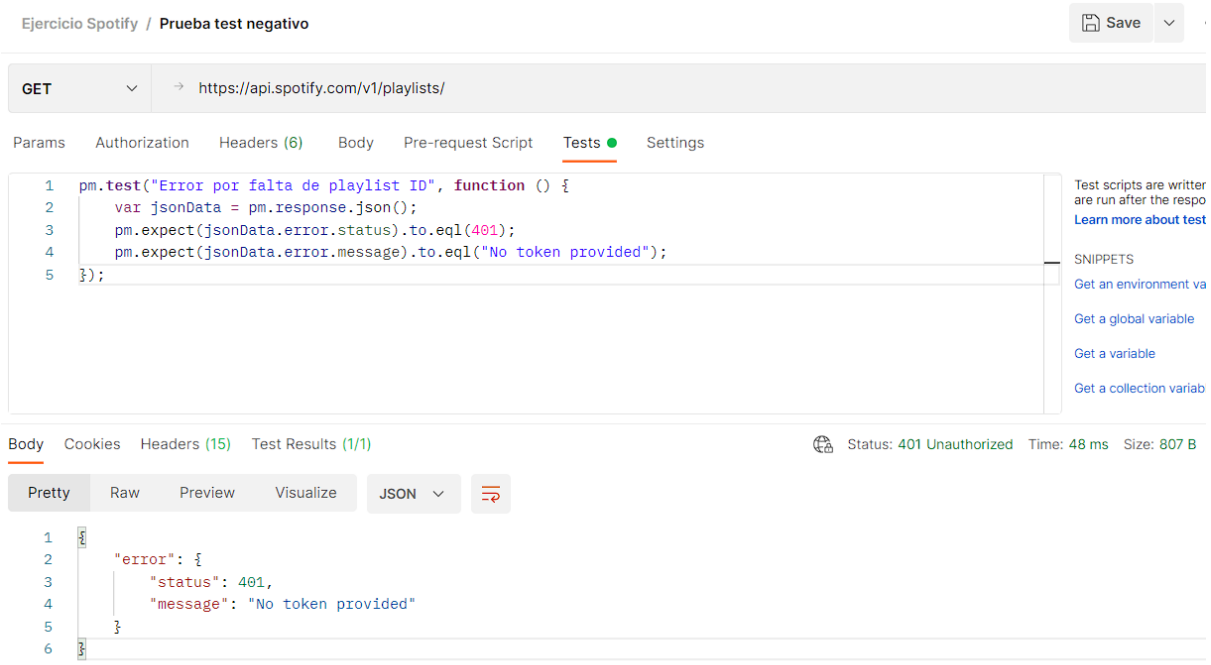
Hasta ahora esto son pruebas manuales, vamos a aprender a hacer tests automáticos. Por ejemplo en el primer endpoint, tenemos la pestaña “tests”.

Tenemos algunos de ejemplo en la colección de Rick y Morty. Ahora vamos a hacer un par de test nosotros, uno positivo y uno negativo.

Positivo:

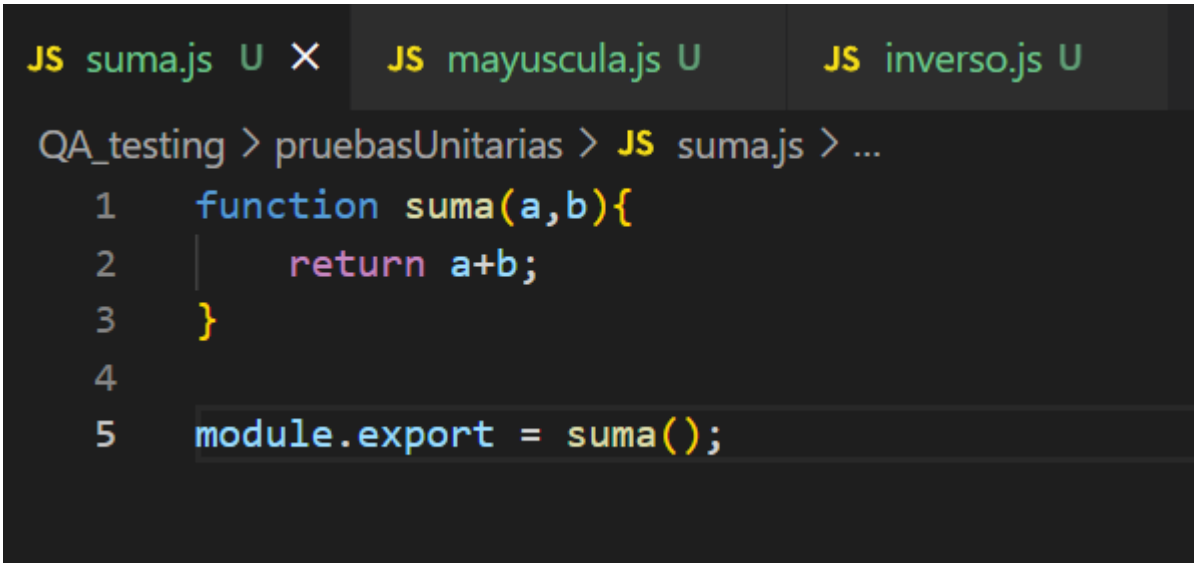


Negativo:



PRUEBAS UNITARIAS

Hemos creado una carpeta nueva para las pruebas unitarias. Usaremos el programa JEST. Hemos creado un archivo js llamado suma, con una función simple de suma, otro de convertir a mayúsculas y otro para hacer el reverse de un string.



JS suma.js U | JS mayuscula.js U | JS inverso.js U

QA_testing > pruebasUnitarias > JS mayuscula.js > ...

```
1   function mayusculas(aux){
2       return aux.toUpperCase()
3   }
4
5   module.export = mayusculas();
```

JS suma.js U | JS mayuscula.js U | JS inverso.js U

QA_testing > pruebasUnitarias > JS inverso.js > ...

```
1   function inverso(aux){
2       return aux.split("").reverse.join(""); // le damos la vuelta a una palabra
3   }
4
5   module.export = inverso();
```

Ahora instalamos el JEST (npm install jest), y en nuestra carpeta “pruebasUnitarias” creamos una carpeta llamada “__test__” esta es la nomenclatura estándar para las pruebas unitarias con jest. Ahora, dentro de la carpeta __test__ creamos los archivos suma.test.js, mayuscula.test.js, inverso.test.js Aquí los test se llaman **test**.

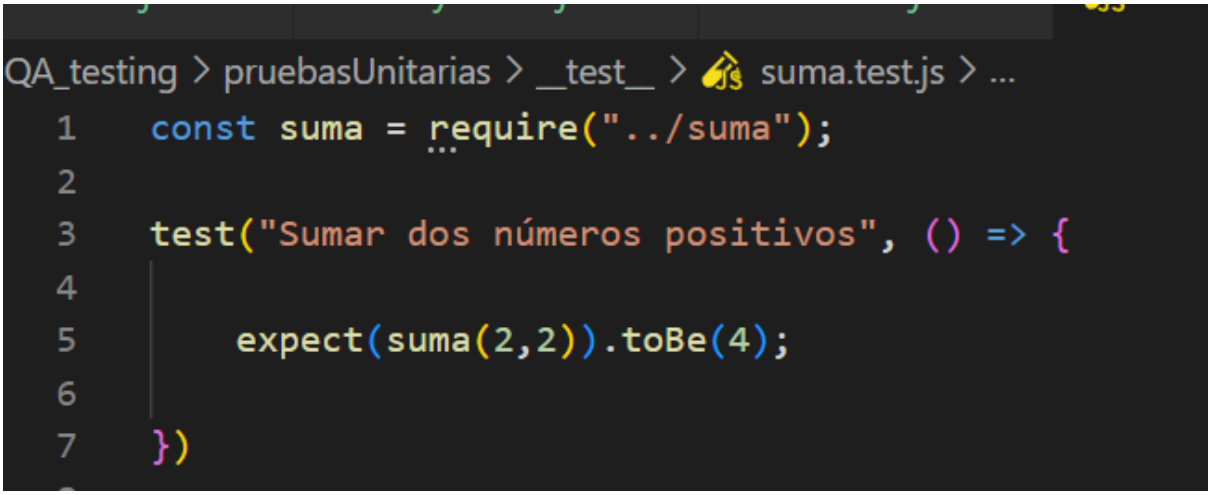
Debemos modificar el package.json de esto:

```
▷ Debug
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1"
8   },
9   "keywords": [],
10  "author": "",
```

A esto:

```
6   "scripts": {
7     "test": "jest"
8   },
9   "keywords": [],
```

Tenemos los archivos de las funciones y las funciones.test en la carpeta de archivos.
Ejemplo de suma.test.js



La sintaxis es:

```
test("Título del test", ()=> {
        expect(funciónatestear()).toBe(resultadoesperado);
})
```

Para correr los test en consola escribimos el comando: npm run test

Métodos comunes de comparación:
.toBe() => usa Object.is para probar la igualdad exacta.
.toEqual() => comprueba recursivamente cada campo de un objeto o array.
.toBeNull => solo coincide con nulo
.toBeUndefined => coincide solo con undefined
.toBeDefined => es lo opuesto a toBeUndefined
.toBeTruthy => coincide con cualquier cosa que una declaración if trate como verdadera
.toBeFalsy => coincide con cualquier cosa que una declaración if trate como falsa

Ahora mismo hemos estado dándole valores a las funciones a lo hardcore, pero vamos a hacer una sóla función que le podamos dar los valores que queramos (DDT- Data Driven Testing).

```
28
29    test.each([[2,2,4],[1,1,2],[3,5,8]])(
30        'i%+i% igual a i%', (a,b,expected) =>{
31            expect(suma(a,b)).toBe(expected)
32        }
33    )
```

Testing de Aceptación de Usuario

Usamos el framework Cucumber. Lo vamos a hacer en la misma carpeta de las pruebas de Cypress, usando ahora el comando npm install cucumber

Y para que cypress pueda funcionar con cucumber hacemos:

npm install cypress-cucumber-preprocessor

E instalamos la extensión de visual studio: Cucumber (Gherkin) Full Support

En el package.json hay que modificar y añadir:

```
10    "author": "",
11    "license": "ISC",
12    "dependencies": {
13      "cucumber": "^6.0.7",
14      "cypress": "^9.6.0",
15      "cypress-cucumber-preprocessor": "^4.3.1"
16    },
17    "cypress-cucumber-preprocessor": {
18      "nonGlobalStepDefinitions": true
19    }
20  }
21
```

Y dentro de la carpeta integration creamos otra carpeta llamada BDD. Y dentro de ésta un archivo llamado: step_definitions.js

Y ahora dentro de integration creamos un archivo que se llame login.feature

step_definitions: (hecho con el lenguaje Gherkin)

GIVEN(‘ I navigate main page’)
WHEN(‘ I click on Form Authentication’)
AND(‘ I enter username)
AND...
AND...
THEN (‘Message Appears’)

Y ahora en el login.feature hacemos algo tal que así (las descripciones del Given, When, And’s y Then deben ser las mismas que el step_definitions):