



Testing



{desafío}
latam_



**Activen las cámaras los que puedan y
pasemos asistencia**





Inicio

{desafío}
latam_



Objetivos

`/* Testear el resultado de una operación aritmética con JEST */`

`/* Testear que el status code de una ruta GET sea el esperado usando el matcher toBe */`

`/* Testear que el tipo de dato de un resultado obtenido en una ruta GET sea el esperado con el matcher toBeInstanceOf */`

`/* Testear que un nuevo recurso se guarde en una consulta POST usando el matcher toContainEqual */`

`/* Testing */`

Testing

¿Qué es el testing?

El testing es un área de la programación dedicada a la prueba de funcionalidades y componentes en una aplicación.

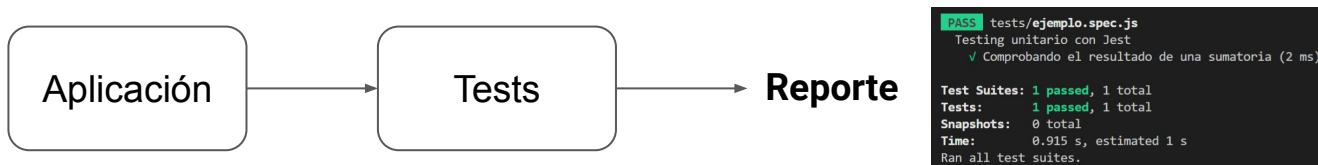
El objetivo del testing es preservar la calidad y consistencia de una aplicación.

Los tests se pueden escribir de manera independiente a la aplicación y acceden a las funcionalidades o componentes por medio de importaciones.

Testing

¿Qué es el testing?

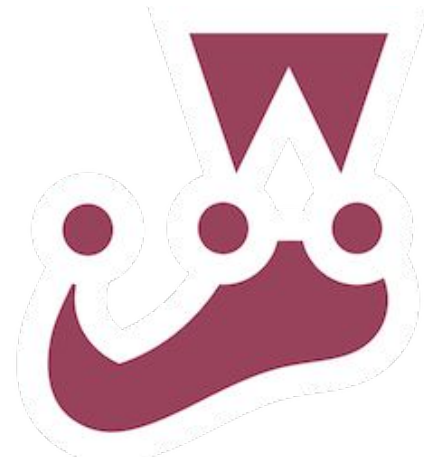
Realizar pruebas en una aplicación aumenta la confianza y garantía de su calidad y consistencia.



Al correr los tests obtenemos un reporte que refleja la cantidad de tests aprobados y reprobados.

Existen varias herramientas para realizar testing.

Una de las más populares y preferidas es **Jest**, un framework de JavaScript construido y diseñado para la realización de pruebas de tipo **unitario**.

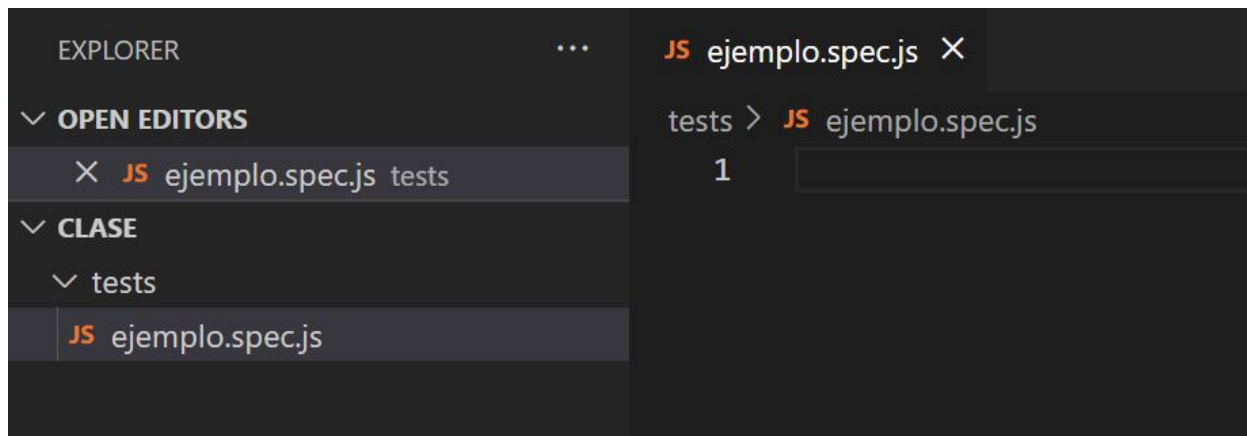


Realicemos nuestra primera prueba unitaria creando un nuevo proyecto con npm e instalando **Jest**.

```
npm install jest
```

Nuestro primer test consistirá en comprobar que una función que realiza una operación aritmética retorna el valor correcto.

Ahora crea una carpeta llamada **tests** e incluye un archivo llamado **ejemplo.spec.js**



Lo siguiente será agregar el siguiente código en el archivo creado:

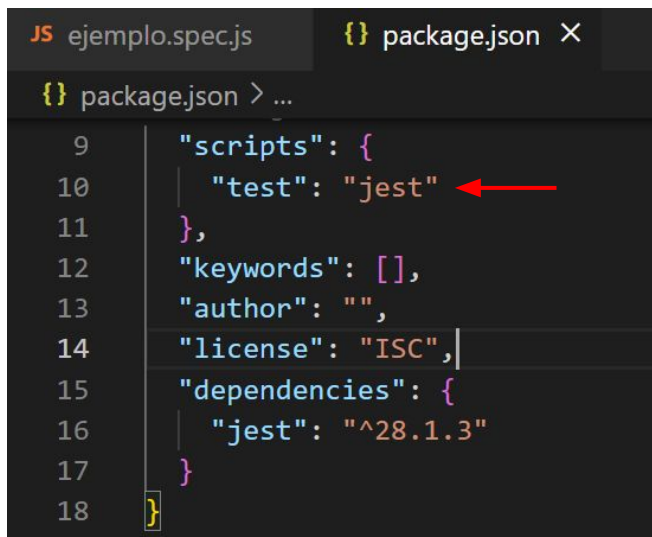
```
const sumar = (a, b) => a + b

describe('Testing unitario con Jest', () => {
  it("Comprobando el resultado de una sumatoria", () => {
    const n1 = 4
    const n2 = 5
    const resultado = sumar(n1, n2)

    expect(resultado).toBe(9)
  })
})
```

Para correr el test que hemos escrito debemos realizar un par de pasos:

1. Modifica el script **test** del **package.json** para que ejecute el framework **jest**



```
JS ejemplo.spec.js  {} package.json X
{} package.json > ...
9   "scripts": {
10   |   "test": "jest" ←
11   | },
12   | "keywords": [],
13   | "author": "",
14   | "license": "ISC",
15   | "dependencies": {
16   |   "jest": "^28.1.3"
17   | }
18 }
```

2. Ejecuta por terminal la siguiente línea de comando:

```
npm run test
```

Esto hará que corran todos los tests que tengamos escritos y nos devolverá por consola el reporte

```
PASS tests/ejemplo.spec.js
  Testing unitario con Jest
    ✓ Comprobando el resultado de una sumatoria (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.915 s, estimated 1 s
Ran all test suites.
```

Testing

Anatomía de los tests

Los tests poseen básicamente la siguiente anatomía:

```
describe('Titulo del grupo de tests', () => {  
  it("Nombre de cada test unitario", () => {  
    // lógica  
    expect( <a> ).<matcher>( <b> )  
  })  
})
```

- **describe:** Es un método por jest que representa un grupo de tests
- **it:** Representa un test unitario
- **expect:** Es un método integrado que recibe como argumento un valor en juicio (**a**)
- **matcher:** Es un método que compara el valor en juicio (**a**) con el valor que se espera idealmente (**b**)

Los tests también pueden salir mal...

Modifica el código del test para cambiar el valor del **n2** por un 5 en formato string:

```
it("Comprobando el resultado de una sumatoria", () => {  
  const n1 = 4  
  const n2 = "5"  
  const resultado = sumar(n1, n2)  
  
  expect(resultado).toBe(9)  
})
```


Si ejecutas el script **test** nuevamente verás lo siguiente:

```
Expected: 9
Received: "45" ←

   7 |         const resultado = sumar(n1, n2)
   8 |
>  9 |         expect(resultado).toBe(9)
      |                             ^
  10 |     })
  11 | })

at Object.toBe (tests/ejemplo.spec.js:9:27)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
```

El desarrollo de tests es subjetivo y depende enteramente del criterio del tester.

De manera que es posible crear un test mal diseñado o que no considere diferentes situaciones, comprometiendo así la veracidad del reporte final.

Ejercicio

Realiza ahora un nuevo test unitario realizando los siguientes pasos:

1. Crea una función **multiplicar**
2. Dentro del **describe**, crea otro **it** que defina 2 números y ejecute la función *multiplicar*
3. Usa el **expect** y un **matcher** para comprobar que el resultado de la función *multiplicar* es correcto

Ejercicio

¡Manos al teclado!



Objetivos

/* Testear el resultado de una operación aritmética con JEST */ ✓

/* Testear que el status code de una ruta GET sea el esperado usando el matcher toBe */

/* Testear que el tipo de dato de un resultado obtenido en una ruta GET sea el esperado con el matcher toBeInstanceOf */

/* Testear que un nuevo recurso se guarde en una consulta POST usando el matcher toContainEqual */

Testing

Testing de servidores con supertest

También podemos testear servidores probando las diferentes rutas y comprobando que las respuestas son las esperadas.

Para esto podemos ocupar el paquete **supertest** que nos proveerá de diferentes métodos para utilizar una instancia de nuestro propio servidor y realizarle consultas http.

Testing

Testing de servidores con supertest

Entre los archivos de esta unidad encontrarás un material de apoyo con una API REST con temática de productos deportivos que contiene diferentes rutas que testaremos con ***supertest***.

Descárgalo e instala las dependencias antes de continuar con la clase.

Testing

El primer test a mi servidor

En el archivo **productos.spec.js** ubicado en la carpeta **tests** agreguemos un test para realizar una consulta **GET** a la ruta **/productos** y verificar que el status code de respuesta sea **200**.

```
const request = require("supertest");
const server = require("../index");

describe("Operaciones CRUD", () => {
  it("Obteniendo un 200", async () => {
    const response = await request(server).get("/productos").send();
    const status = response.statusCode;
    expect(status).toBe(200);
  });
});
```

Testing

El primer test a mi servidor

Si ejecutas el comando **npm run test** en la terminal podremos ver que efectivamente el test pasa sin problemas

```
PASS tests/productos.spec.js
  Operaciones CRUD
    ✓ Obteniendo un 200 (30 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.214 s, estimated 2 s
Ran all test suites.
```

Testing

Anatomía de un supertest

El paquete supertest tiene la siguiente anatomía base:

```
const request = require("supertest");  
  
request(server).get("/productos").send()
```

Instancia del servidor

Método de consulta

Ruta a consultar

Envío de la consulta

Ejercicio

Realiza ahora una consulta a una ruta que no exista y confirma que el código que recibimos sea 404

Ejercicio

¡Manos al teclado!



Objetivos

/* Testear el resultado de una operación aritmética con JEST */ ✓

/* Testear que el status code de una ruta GET sea el esperado usando el matcher toBe */ ✓

/* Testear que el tipo de dato de un resultado obtenido en una ruta GET sea el esperado con el matcher toBeInstanceOf */

/* Testear que un nuevo recurso se guarde en una consulta POST usando el matcher toContainEqual */

Testing

Validación de Data Type

Entre los tantos matchers de Jest encontramos el método ***toBeInstanceOf*** que utilizaremos para verificar que la respuesta de una consulta es una instancia de un tipo de dato declarado como argumento.

Los tipos de datos se declaran con la primera letra en mayúsculas:

- String
- Object
- Array
- Boolean

Testing

Validación de Data Type

Agrega el siguiente test en nuestro test suite para verificar que la respuesta de una ruta **GET** **/productos/:id** nos devuelve un **Object**.

```
it("Obteniendo un producto", async () => {  
  
    const { body } = await request(server).get("/productos/1").send();  
    const producto = body;  
    expect(producto).toBeInstanceOf(Object);  
  
});
```

Testing

Validación de Data Type

Ejecutando el comando **npm run test** nuevamente obtendremos el siguiente reporte:

```
PASS tests/productos.spec.js
```

```
Operaciones CRUD
```

```
✓ Obteniendo un 200 (30 ms)
```

```
✓ Consultando una ruta que no existe (11 ms)
```

```
✓ Obteniendo un producto (5 ms)
```

```
Test Suites: 1 passed, 1 total
```

```
Tests: 3 passed, 3 total
```

```
Snapshots: 0 total
```

```
Time: 0.944 s, estimated 1 s
```

Ejercicio

Agrega un nuevo test a nuestro test suite que confirme que la respuesta de la ruta **GET /productos** incluye un cuerpo de tipo **Array**

Ejercicio

¡Manos al teclado!



Objetivos

/* Testear el resultado de una operación aritmética con JEST */ ✓

/* Testear que el status code de una ruta GET sea el esperado usando el matcher toBe */ ✓

/* Testear que el tipo de dato de un resultado obtenido en una ruta GET sea el esperado con el matcher toBeInstanceOf */ ✓

/* Testear que un nuevo recurso se guarde en una consulta POST usando el matcher toContainEqual */

Testing

Enviando un payload en un supertest

Para probar las rutas que reciben un payload en la consulta podemos cambiar el método por un **POST** e incluir un argumento en el método **send** que representará el **body** en la consulta.

```
it("Enviando un nuevo producto", async () => {  
  const id = Math.floor(Math.random() * 999);  
  const producto = { id, nombre: "Nuevo producto" };  
  
  const { body: productos } = await request(server)  
    .post("/productos")  
    .send(producto);  
  expect(productos).toContainEqual(producto);  
});
```

La API REST está diseñada para devolver todos los productos al momento de agregar uno nuevo.

El matcher **toContainEqual()** nos ayudará a confirmar que el producto enviado efectivamente fue agregado.

Ejercicio

La API REST contiene una ruta **PUT /productos** que recibe como payload un producto a sobrescribir y utiliza el **id** de este objeto para identificar el registro a actualizar.

Escribe un nuevo test que compruebe que si enviamos un producto cuyo id no existe entre los productos registrados, recibiremos un status code **404**.

Ejercicio

¡Manos al teclado!



Objetivos

/* Testear el resultado de una operación aritmética con JEST */ ✓

/* Testear que el status code de una ruta GET sea el esperado usando el matcher toBe */ ✓

/* Testear que el tipo de dato de un resultado obtenido en una ruta GET sea el esperado con el matcher toBeInstanceOf */ ✓

/* Testear que un nuevo recurso se guarde en una consulta POST usando el matcher toContainEqual */ ✓



Cierre

{desafío}
latam_



¿Existe algún concepto que no
hayas comprendido?

Reflexionemos

- Revisemos el material de estudio asincrónico.
 - En donde además aprenderemos cómo incluir un token en una consulta con supertest
- Revisemos el desafío de la unidad leyendo la descripción y requerimientos.

¿Qué sigue?

¿Tienen alguna duda respecto al Desafío?



*Academia de
talentos digitales*

www.desafiolatam.com



/DesafioLatam



/DesafioLatam



/DesafioLatam



/DesafioLatam