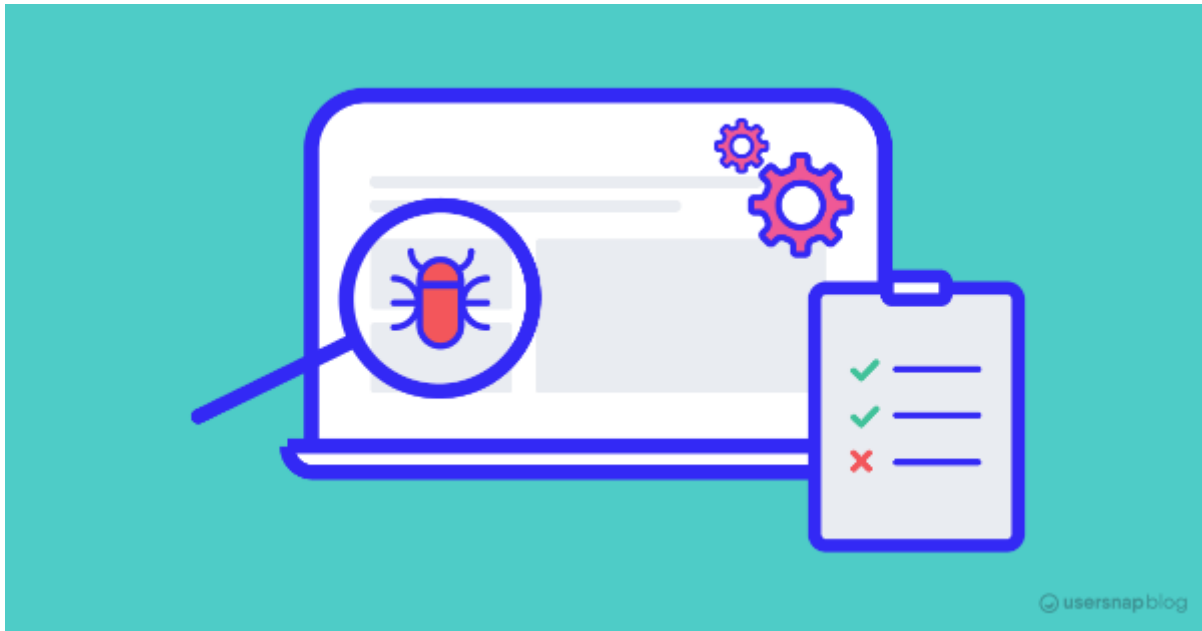


## Apunte 16 - TESTING

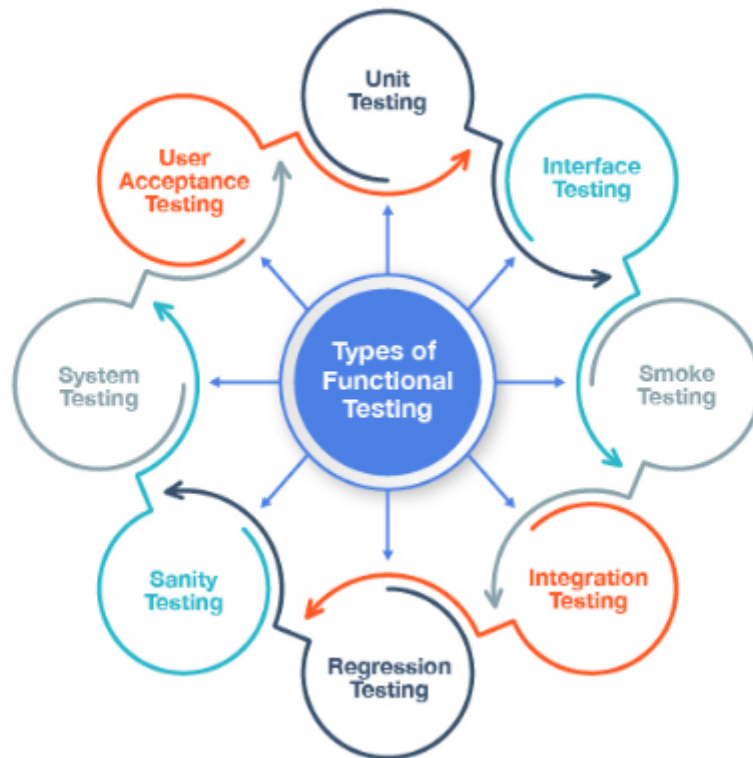


El testing o la prueba de software es el proceso de evaluar y verificar que un producto o aplicación de software hace lo que se supone que debe hacer. Los beneficios de las pruebas incluyen la prevención de errores, la reducción de los costos de desarrollo y la mejora del rendimiento.

### Tipos de pruebas de software

En un nivel más amplio, los tipos de pruebas de software se pueden dividir en dos tipos, pruebas funcionales y pruebas no funcionales.

#### Pruebas Funcionales



Las pruebas funcionales son un tipo de prueba que tiene como objetivo determinar si cada característica de la aplicación funciona siguiendo los requisitos del proyecto. Las pruebas funcionales se ocupan de las especificaciones funcionales o los requisitos comerciales. Esta prueba es fundamental para evaluar la calidad y la funcionalidad del software. Cada función se compara con la condición correspondiente para ver si su resultado coincide con las expectativas del usuario final.

Las pruebas funcionales más simples se denominan **pruebas unitarias**; el desarrollador suele probar la unidad o el componente más básico de una aplicación para garantizar que el código más pequeño que se pueda probar funcione bien. Es una técnica de prueba de Caja Blanca y corresponde al tipo de prueba más elemental que se puede realizar.

Uno de los objetivos o el objetivo principal de las pruebas unitarias es garantizar que el código que funciona siga funcionando incluso luego de agregar nuevas funcionalidades o modificaciones por cambios en funcionalidades existentes en el contexto de un desarrollo incremental.

**Pruebas de integración:** cuando se integran dos o más componentes o unidades de software, deben interactuar entre sí en forma de comandos, intercambio de datos o llamadas a bases de datos. Ejecute pruebas de integración en ellos como un solo clúster para verificar que interactúan como se esperaba.

**Prueba de interfaz:** se incluye en la prueba de integración donde se prueba la corrección del intercambio o transmisión de datos entre dos componentes. Por ejemplo, un componente crea un archivo .csv como salida, otro componente conectado procesa el archivo .csv en XML y lo envía a un tercer componente. Durante esta transferencia de datos, los datos deben permanecer intactos y todos los componentes deben poder procesar el archivo y enviarlo con éxito al siguiente componente.

**Pruebas del sistema:** reúne todos los módulos de la aplicación y prueba todo el sistema como una unidad para verificar que cumple con la especificación de requisitos.

Las **pruebas de regresión** verifican un conjunto de escenarios que funcionaron correctamente en el pasado, para asegurar que continúen así. Una falla en una prueba de regresión significa que una nueva funcionalidad ha afectado otra funcionalidad que era correcta en el pasado, causando una "regresión"

Las **pruebas de humo** son pruebas que verifican la funcionalidad básica de una aplicación y su objetivo es asegurar que las características más importantes del sistema funcionen como se espera.

**Pruebas de saneamiento:** suele ser un subconjunto de las pruebas de regresión; se realiza cada vez que se lanza una nueva versión de una aplicación estable. Solo después de ejecutar el conjunto de pruebas de saneamiento, la compilación pasa al siguiente nivel de prueba. La diferencia entre humo y saneamiento es que las pruebas de humo se realizan en una aplicación inicialmente inestable, mientras que saneamiento se realiza en una aplicación estable.

Las **pruebas de aceptación** son pruebas formales, ejecutadas para verificar si un sistema satisface sus requerimientos de negocio, requieren que el software se encuentre en funcionamiento, y se centran en replicar el comportamiento de los usuarios, a fin de rechazar cambios si no se cumplen los objetivos.

### Pruebas no funcionales



Las pruebas no funcionales **se ocupan del rendimiento de la aplicación.** Las pruebas no funcionales son tan importantes como las pruebas funcionales, ya que nadie querrá usar una aplicación que no sea escalable o segura, o digamos, la aplicación no puede manejar un carga de muchas operaciones de base de datos en una instancia. Por lo tanto, el rendimiento de la aplicación en varias áreas se prueba durante las pruebas no funcionales.

**Pruebas de rendimiento:** el rendimiento de la aplicación se mide mientras la aplicación se encuentra en condiciones reales. En esta prueba se monitorean parámetros de desempeño como tiempo de respuesta, escalabilidad, estabilidad, eficiencia en el uso de recursos, etc., por lo que es útil para saber cuándo se degradaba la aplicación. Ayudará a mejorar el diseño o la arquitectura de la aplicación de manera que garantice la confiabilidad y un tiempo de respuesta rápido.

**Prueba de estrés:** La aplicación tendrá condiciones anormales, que no ocurrirán en circunstancias normales. La carga en la aplicación aumenta hasta el punto en que se rompe y se registra el comportamiento de la aplicación. Algunas de las preguntas que responde esta prueba son: ¿Cómo funciona el sistema en condiciones de estrés? Si falla, ¿es posible recuperarlo y reiniciarlo?

**Prueba de capacidad:** el propósito de esta prueba es determinar el rendimiento del sistema cuando la base de datos maneja una gran cantidad de datos u operaciones de datos. ¿La base de datos es capaz de almacenar y procesar grandes cantidades de datos? ¿Habrá algún problema debido a un gran volumen de datos?

**Pruebas de carga:** se mide el rendimiento de la aplicación bajo la carga esperada como en el mundo real. Se simulan todas las cargas posibles en la aplicación y se comprueba el rendimiento. Por ejemplo, en un día normal, se espera que 1000 usuarios visiten un sitio web, el rendimiento del sitio web se mide simulando 1000 usuarios simultáneos, esta es una carga normal durante el uso en el mundo real. Sin embargo, cuando se ejecuta una promoción, la cantidad de usuarios puede aumentar a 2000, en este caso, la aplicación se prueba para una carga de 2000 usuarios también porque esta es la carga esperada. A diferencia de las pruebas de estrés, las condiciones anormales no se prueban, solo se prueba en el sistema la carga esperada, es decir, la carga que se espera que la aplicación enfrente en el mundo real. Si la aplicación se degrada bajo carga, se pueden realizar cambios en la arquitectura para evitar dicho cuello de botella.

**Pruebas de seguridad:** se prueba la seguridad de la aplicación desde el punto de vista de la red, los datos, el sistema y la aplicación. Cualquier persona no autorizada no debería poder acceder a la aplicación y no debería poder acceder a ningún dato confidencial. Esta prueba garantiza la fiabilidad del cliente y la confianza en la aplicación. Las pruebas de penetración son un ejemplo de las pruebas de seguridad.

**Pruebas de escalabilidad:** en caso de que la aplicación deba ser escalable en el futuro, ¿lo permitirán la arquitectura y el diseño? Por ejemplo, si agregamos más servidores, tenemos más transacciones de base de datos, aumentamos la carga de usuarios en el futuro, ¿lo permitirá nuestra aplicación? Esto se prueba en las pruebas de escalabilidad de la aplicación.

**Pruebas de usabilidad:** se ocupa de la usabilidad del sistema desde la perspectiva del usuario. ¿Puede el usuario navegar por la aplicación sin ayuda? La próxima vez que el usuario visite la aplicación, ¿podrá recordar la aplicación sin ningún tipo de ayuda o problema? ¿Qué tan eficientemente un usuario puede usar el sistema? Todas estas preguntas se responden a través de pruebas de usabilidad.

**Pruebas de mantenibilidad:** diseñar una aplicación es una cosa y mantener la aplicación para futuras expansiones y requisitos es otra igualmente importante. La capacidad de la aplicación para adaptarse a los cambios y actualizaciones realizados en el código o el sistema se somete a pruebas de mantenimiento.

**Pruebas de compatibilidad:** la compatibilidad de la aplicación se prueba con respecto a diferentes sistemas operativos, navegadores, hardware, capacidad de red, dispositivos, etc. La aplicación debe poder funcionar bien en los entornos priorizados por el cliente.

## Pruebas Manuales y de Automatización

Si categorizamos los servicios de prueba en función del esfuerzo (humano o de máquina), tenemos dos categorías amplias.

- **Prueba manual** Como sugiere el nombre, las pruebas manuales las llevan a cabo manualmente personas reales. Los probadores realizan pruebas manuales siguiendo los pasos de prueba en el caso de prueba; haciendo clic, proporcionando entradas en la aplicación y finalmente validando los resultados de la prueba.
- **Pruebas de automatización** Las pruebas de automatización implican el uso de herramientas y scripts de automatización para ejecutar los casos de prueba automáticamente, sin intervención manual. Estas herramientas ejecutan los casos de prueba, registran los resultados de las pruebas y, a veces, registran los defectos en la herramienta de seguimiento de defectos.

## Desarrollo guiado por pruebas de software, o Test-driven development (TDD)

Es una práctica de ingeniería de software que involucra otras dos prácticas: **Escribir las pruebas primero (Test First Development)** y **Refactorización (Refactoring)**. Para escribir las pruebas generalmente se utilizan las pruebas unitarias. En primer lugar, se escribe una prueba y se verifica que la nueva prueba falla. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del *desarrollo guiado por pruebas* es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido.

### Ciclo de desarrollo conducido por pruebas TDD

En primer lugar se debe definir una lista de requisitos y después se ejecuta el siguiente ciclo:

1. **Elegir un requisito:** Se elige de una lista el requisito que se cree que nos dará mayor conocimiento del problema y que a la vez sea fácilmente implementable.
2. **Escribir una prueba:** Se comienza escribiendo una prueba para el requisito. Para ello el programador debe entender claramente las especificaciones y los requisitos de la funcionalidad que está por implementar. Este paso fuerza al programador a tomar la perspectiva de un cliente considerando el código a través de sus interfaces
3. **Verificar que la prueba falla:** Si la prueba no falla es porque el requisito ya estaba implementado o porque la prueba es errónea.
4. **Escribir la implementación:** Escribir el código más sencillo que haga que la prueba funcione. Se usa la expresión "Déjelo simple" ("Keep It Simple, Stupid!")
5. **Ejecutar las pruebas automatizadas:** Verificar si todo el conjunto de pruebas funciona correctamente.
6. **Eliminación de duplicación:** El paso final es la refactorización, que se utilizará principalmente para eliminar código duplicado. Se hace un pequeño cambio cada vez y luego se corren las pruebas hasta que funcione.
7. **Actualización de la lista de requisitos:** Se actualiza la lista de requisitos tachando el requisito implementado. Asimismo se agregan requisitos que se hayan visto como necesarios durante este ciclo y se agregan requisitos de diseño (P. ej que una funcionalidad esté desacoplada de otra).

Tener un único repositorio universal de pruebas facilita complementar TDD con otra práctica recomendada por los procesos ágiles de desarrollo, la "Integración Continua". Integrar continuamente nuestro trabajo con el del resto del equipo de desarrollo permite ejecutar toda la batería de pruebas y así descubrir, si nuestra última versión es compatible con el resto del sistema. Es recomendable y menos costoso corregir pequeños problemas cada pocas horas, que enfrentarse a problemas enormes cerca de la fecha de entrega fijada.

## JEST

Jest es el framework de pruebas unitarias más popular en JavaScript, utilizado por grandes empresas como Airbnb, Twitter, Spotify y cuenta con plugins que se integran muy bien con frameworks de front-end como React, Vue, Angular, etc.

Es una biblioteca de prueba creada por Facebook para ayudar a probar el código JavaScript, los componentes de React y mucho más. Lo bueno de Jest es que no solo tiene una sintaxis similar a otras bibliotecas de prueba/afirmación como Jasmine y Chai, sino que con Jest sus pruebas se ejecutan en paralelo, por lo que se ejecutan mucho más rápido que otros marcos de prueba.

Para usar jest globalmente (instalado en el sistema operativo) podemos instalarlo con:

```
npm install -g jest.
```

Otra forma es agregarlo en el archivo `package.json` de esta manera se instala la versión adecuada para el proyecto y se encapsula solo para el uso del mismo.

En archivo `package.json` agregar:

```
...
  "scripts": {
    ...
    "test": "jest --testTimeout=10000 --force-exit",
    "test:jsmodule" : "node --experimental-vm-modules jest --verbose --silent --detectOpenHandles"
  },
  "devDependencies": {
    "jest": "~29.7.0"
  }
...
```

Dado que son bibliotecas para hacer pruebas no deberían quedar en producción por eso de lo agrega en la sección `devDependencies`.

Si instalamos Jest de forma global, podemos agregar el script `test` para poder ejecutar luego `npm run test`. El otro script `test:jsmodule` es para el caso su se esta usando el `"type": "module"` en el `package.json` y se ejecutaría `npm run test:jsmodule`. El `"type": "module"` es cuando se usa el `import` en lugar del `require` en los archivos js.

En el caso que lo hayamos instalado en forma local debemos configurar los scripts de la siguiente manera:

```
...
  "scripts": {
    ...
    "test": "node node_modules/jest/bin/jest.js --verbose --silent --
testTimeout=10000 --force-exit",
    "test:jsmodule" : "node --experimental-vm-modules
node_modules/jest/bin/jest.js --verbose --silent --force-exit"
  },
  "devDependencies": {
    "jest": "~29.7.0"
  }
...

```

Despues instalamos las dependencias

```
npm install
```

Si desea simplificar los dos pasos anteriores en uno puede ejecutar el comando, pero si o si va a hacer falta agregar los scripts:

```
npm install --save-dev
```

Para empezar, vamos a usar jest para escribir algunas pruebas unitarias simples.

- inicialmente creamos una carpeta para nuestro proyecto
  - ej DemoTest
- abrimos una terminal, ubicándonos dentro de dicha carpeta y ejecutamos:

```
npm init -y #inicializamos el proyecto sin preguntas interactivas
```

- dentro del proyecto creamos
  - un archivo vacío: **index.js**, donde almacenaremos el código de nuestra aplicación,

```
// index.js
export function sumar(a, b) {
  return a + b;
}

export default sumar;
```

- un archivo vacío: **primer.test.js** (el código de nuestros testing)

```
// primer.test.js
import sumar from "./index.js";

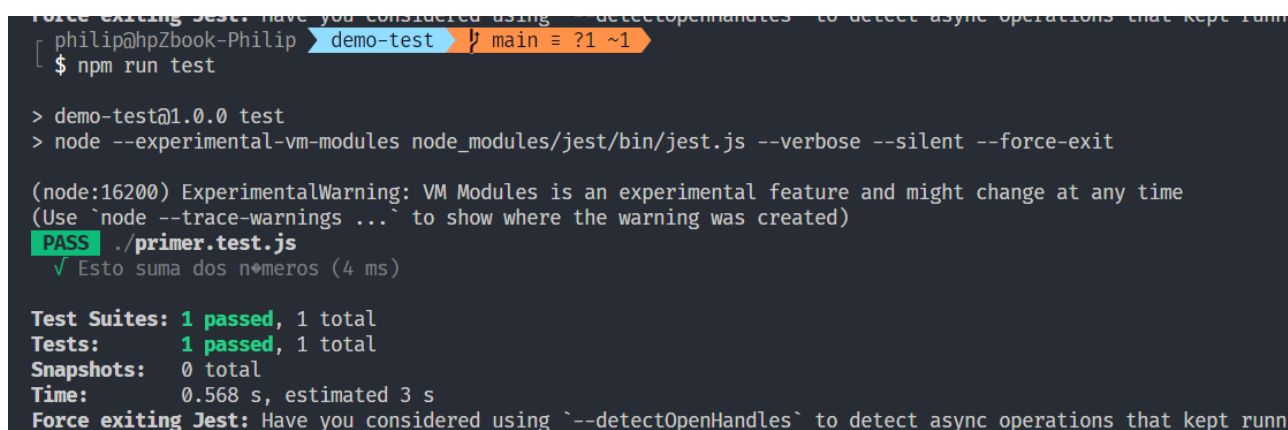
test("Esto suma dos números", () => {
  const a = 1;
  const b = 1;
  const expected = a + b;
  const res = sumar(a, b);
  expect(res).toBe(expected);
})
```

[!NOTE] En este caso, la funcionalidad a probar es la suma de dos números que está implementada en la función suma en el archivo index.js. El test se encarga de probar que cada vez que se invoque la función suma su resultado será el esperado o sea la suma de los dos números enviados como parámetros.

- ahora en la terminal ejecutemos **jest** (asegúrese de haberlo instalado, preferentemente agregandolo en el package.json) y deberíamos visualizar lo siguiente:

Luego de ejecutar:

```
npm run test
```



```
Force exiting Jest: have you considered using `--detectOpenHandles` to detect async operations that kept running
[ philip@hpZbook-Philip demo-test ] main ≡ ?1 ~1
$ npm run test

> demo-test@1.0.0 test
> node --experimental-vm-modules node_modules/jest/bin/jest.js --verbose --silent --force-exit

(node:16200) ExperimentalWarning: VM Modules is an experimental feature and might change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
PASS ./primer.test.js
  ✓ Esto suma dos números (4 ms)

Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 0.568 s, estimated 3 s
Force exiting Jest: Have you considered using `--detectOpenHandles` to detect async operations that kept running
```

- Haciendo una revisión de nuestro primer test encontramos:

**test** es una función que toma un nombre o descripción de la prueba que se plantea y una función callback que se ejecutará al invocarse el test. La función de callback es la prueba real que se ejecutará. En este caso, estamos usando una función flecha para definir nuestra función callback.

**expect** es una función de afirmación que toma un valor y devuelve un objeto que tiene una serie de funciones de afirmación que podemos usar para afirmar cosas sobre ese valor. En este caso, estamos



usando `toBe()`, que es una función de afirmación que verifica que el valor que pasamos sea exactamente igual al valor que esperamos.

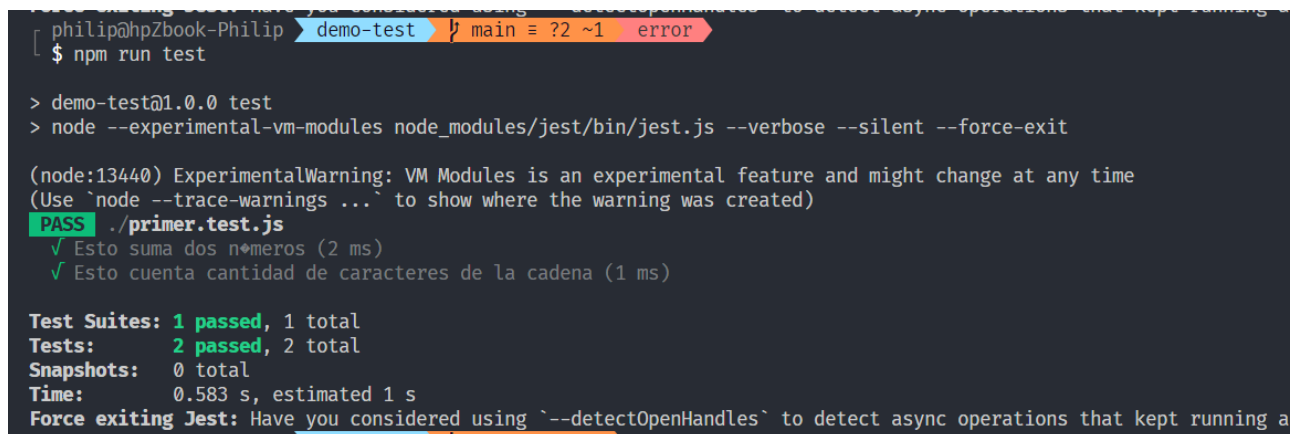
- Ahora vamos a **agregar más funcionalidad a nuestra aplicación**, que luego queremos testear, al archivo **index.js**, le agregamos la siguiente función que simplemente cuenta los caracteres de una palabra:

```
...
export function cuentaLetras(texto) {
  return texto.length;
}
...
```

- Para testear esta nueva función, agregamos a `primer.test.js` otro test mediante el siguiente código:

```
test("cuentaLetras funciona", () => {
  expect(cuentaLetras("importante")).toBe(10);
});
```

- probemos ahora ejecutar desde consola nuevamente con las pruebas actuales y también cambiando el **valor esperado** en: **`toBe(..)`**, notar que en esta última prueba la cantidad de tests a ejecutarse aumentó a dos y podemos ver la ejecución de ambos en el resultado.



```
philip@hpZbook-Philip demo-test ❯ main ≡ ?2 ~1 error
$ npm run test

> demo-test@1.0.0 test
> node --experimental-vm-modules node_modules/jest/bin/jest.js --verbose --silent --force-exit

(node:13440) ExperimentalWarning: VM Modules is an experimental feature and might change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
PASS ./primer.test.js
  ✓ Esto suma dos números (2 ms)
  ✓ Esto cuenta cantidad de caracteres de la cadena (1 ms)

Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
Snapshots: 0 total
Time: 0.583 s, estimated 1 s
Force exiting Jest: Have you considered using `--detectOpenHandles` to detect async operations that kept running a
```

## Testing de backend

El software moderno no sería posible sin las **API HTTP**. En una **arquitectura de microservicio**, se conectan componentes de front-end y back-end o diferentes componentes de back-end. Los desarrolladores también confían en las **API de terceros** para funciones esenciales del producto, como pagos y comunicaciones. **Cada API es un contrato** que ambas partes deben seguir al pie de la letra para mantenerse conectados. En el caso de las API, eso significa que los **códigos de estado**, los **encabezados** y los **cuerpos de respuesta** siempre **deben coincidir con la documentación de la API**. Para un proveedor de API, las pruebas funcionales son cruciales para garantizar que puedan cumplir constantemente lo que prometen. Incluso los consumidores de API pueden optar por ejecutar pruebas para API de terceros. Ya sea que desee probar API internas o externas, necesita buenas herramientas.

## Supertest

SuperTest es una biblioteca de Node.js que ayuda a los desarrolladores a probar las API http. Extiende otra biblioteca llamada `superagent`, un cliente HTTP de JavaScript para Node.js y el navegador. Los desarrolladores pueden usar SuperTest como una biblioteca independiente o con marcos de prueba de JavaScript como Mocha o Jest. En nuestro caso usaremos Supertest en combinación con Jest, entonces en esta combinación Supertest será la herramienta que nos permitirá consumir los endpoints desarrollados a través de HTTP y Jest la herramienta que permite, tanto la construcción de los tests y la evaluación de los resultados como, la ejecución de los mismos de forma automatizada.

SuperTest está disponible en NPM. Puede instalarlo escribiendo el siguiente comando en su consola:

```
npm install supertest --save-dev
```

Partamos ahora entonces, con un nuevo ejemplo donde vamos a construir una mínima aplicación de backend con algunos endpoints para probar. El siguiente es el `app.js` del ejemplo que acompaña el presente material:

```
apunte-16 > js > demo-test-api > JS app.js > ...
```

You, 1 minute ago | 1 author (You)

You, 1 minute ago • Versión 2024

```
1 import express from "express";
2
3 const app = express();
4 const usuarios = [{ id: 1, nombre: "Usuario 1" }, { id: 2, nombre: "Usuario 2" }];
5
6 app.use(express.json());
7
8 // Handler function para la ruta raíz
9 app.get("/api", (req, res) => {
10 |   res.send("Servidor iniciado y escuchando ...");
11 | });
12
13 // Handler function para la ruta /usuarios
14 app.get("/api/usuarios", (req, res) => {
15 |   res.json(usuarios);
16 | });
17
18 // Handler function para la ruta /usuario/:id
19 app.get("/api/usuarios/:id", (req, res) => {
20 |   const userId = req.params.id;
21 |   const usuario = usuarios[userId - 1];
22 |   res.json(usuario);
23 | });
24
25 app.post("/api/usuarios", (req, res) => {
26 |   const newUserario = { id: usuarios.length + 1, ... req.body };
27 |   usuarios.push(newUserario);
28 |   res.status(201).json(newUserario);
29 | });
30
31 // Handler function para rutas no encontradas
32 app.use((req, res) => {
33 |   res.status(404).send("Ruta no encontrada");
34 | });
35
36 (async function start() {
37 |   // Escuchar en el puerto 3000
38 |   const PORT = 3000;
39 |   app.listen(PORT, () => {
40 |     console.log(`Servidor REST escuchando en el puerto ${PORT}`);
41 |   });
42 | }());
43
44 export default app;
```

que está construido en el contexto de un proyecto de npm con el siguiente `package.json`:

```

1 {
2   "name": "demo-test-api",
3   "version": "1.0.0",
4   "description": "Ejemplo de tests de endpoints de una API",
5   "type": "module",
6   "main": "app.js",
7   "scripts": {
8     "dev": "npx nodemon app.js",
9     "lint": "eslint .",
10    "lint:fix": "eslint --fix .",
11    "test": "node --experimental-vm-modules node_modules/jest/bin/jest.js --verbose --silent --force-exit"
12  },
13  "author": "Philip",
14  "license": "ISC",
15  "devDependencies": {
16    "@eslint/eslintrc": "^3.0.2",
17    "@eslint/js": "^9.0.0",
18    "@jest/globals": "^29.7.0",
19    "@types/jest": "^29.5.12",
20    "eslint": "^8.57.0",
21    "eslint-config-airbnb-base": "^15.0.0",
22    "eslint-plugin-import": "^2.29.1",
23    "globals": "^15.0.0",
24    "jest": "^29.7.0",
25    "nodemon": "^3.1.0",
26    "supertest": "^7.0.0"
27  },
28  "dependencies": {
29    "express": "^4.19.2"
30  }
31 }

```

Ahora comencemos a escribir los tests, en primer lugar vamos a probar nuestro servidor, es decir vamos a trabajar probando nuestro proyecto de forma que los test prueben los endpoints del servidor local.

1. Una forma de inicializar su solicitud, puede pasar el objeto de la aplicación si creó su API con Node.js y Express. En el último caso, SuperTest iniciará un servidor de prueba si es necesario y automáticamente enviará solicitudes allí. Aquí hay un ejemplo de cómo crear una aplicación Express con EndPoint (inspirado en Dog API) y luego solicitar ese mismo EndPoint:

```

tests > app.test.js > test("El servidor debe iniciar") callback > expect() callback
1 import request from "supertest";
2 import app from "../app.js";
3
4 test("El servidor debe iniciar", async () => {
5   // realizo una petición a la ruta por defecto del servidor
6   await request(app)
7     .get("/api")
8     .expect(200)
9     .expect(async (res) => {
10       expect(res.text).toBe("Servidor iniciado y escuchando ...");
11     });
12 });
13

```

En el código anterior hay varias cosas a observar: en primero lugar importar `request` de `supertest` que será nuestro canal de ejecución a nuestra api a probar. Luego también necesitamos importar nuestro servidor que debe ser previamente exportado en `app.js`, en este ejemplo lo estamos importando nuevamente con el nombre `app`.

2. También se puede probar una API externa o una API interna en ejecución en otro proceso, proporcionando la URL como parámetro. Este es un ejemplo de cómo realizar una solicitud de API externa. Usamos la API pública para perros que ofrece imágenes de perros de código abierto. Es una

API simple que no requiere registro y, como tal, es perfecta para demostraciones del uso de API externas:

```
import request from 'supertest';
test('API remota', async () => {
  const res = await request('https://dog.ceo')
    .get('/api/breeds/image/random');
  console.log(res.body);
});
```

El método **get()** le dice a SuperTest que estamos usando el verbo HTTP GET. Puede encadenar métodos adicionales en la llamada para configurar la autenticación, encabezados o cuerpo HTTP personalizados, etc., pero esto no es necesario para este ejemplo. Tenga en cuenta que la función que recibe el comando test es asíncrona necesaria para el uso interno de await.

Tener en cuenta que para ver el resultado de `console.log(...)` hay que quitar el `--silent` de la ejecución.

### Establecer expectativas de respuesta

Como sabemos las API HTTP devuelven varios códigos de estado. Los códigos en el rango 4xx indican errores del cliente y los códigos en el rango 5xx indican errores del servidor. Para una solicitud de API exitosa, generalmente se espera 200. A veces es 201 o 204 en su lugar. Ampliemos nuestra solicitud de prueba con la expectativa de que devuelva 200:

```
tests > app.test.js > test("El servidor debe iniciar") callback > expect() callback
1  import request from "supertest";
2  import app from "../app.js";
3
4  test("El servidor debe iniciar", async () => {
5    // realizo una petición a la ruta por defecto del servidor
6    await request(app)
7      .get("/api")
8      .expect(200)
9      .expect(async (res) => {
10        expect(res.text).toBe("Servidor iniciado y escuchando ... ");
11      });
12  });
13
```

Como puede ver, las llamadas a **expect()** siempre las hacemos después de recibir la respuesta. En el primer caso estamos simplemente indicando el código de estado esperado y en el segundo caso estamos indicando un callback que será ejecutado cuando la respuesta sea obtenida.

La mayoría de las API HTTP modernas devuelven respuestas JSON. La API Dog no es una excepción. Para estas respuestas, el valor del encabezado content-type debe ser application/json. Agreguemos entonces esta expectativa:

```

14 describe("Pruebas para la api usuarios", () => {
15   it("Debe responder la lista de usuarios como un array json", async () => {
16     await request(app)
17       .get("/api/usuarios")
18       .expect(async (res) => {
19         expect(res.statusCode).toEqual(200);
20         expect(res.headers["content-type"]).toEqual("application/json; charset=utf-8");
21         expect(res.body).toHaveLength(2);
22       });
23   });
24 });

```

En el código anterior se utilizó **it()** en lugar de **test()** es un alias para compatibilidad con otras librerías; y la función **describe()** sirve para agrupar varios test con algún criterio, por ej el mismo recurso.

```

24
25   it("Debe responder el usuario con id = 2 com un objeto json", async () => {
26     await request(app)
27       .get("/api/usuarios/2")
28       .expect(async (res) => {
29         expect(res.statusCode).toEqual(200);
30         expect(res.headers["content-type"]).toEqual("application/json; charset=utf-8");
31         expect(res.body).toEqual(expect.objectContaining({
32           id: 2,
33           nombre: "Usuario 2"
34         }));
35       });
36   });
37 });

```

- **objectContaining()** es una función de expect que verifica que el objeto contenga las propiedades especificadas.

El test debería ejecutarse correctamente sin generar fallas. Puede intentar cambiar la expectativa con algo diferente para ver cómo falla la prueba.

## Comprobación de los cuerpos de respuesta

Un error común con las pruebas de API funcionales es que los desarrolladores solo verifican los encabezados y los códigos de estado HTTP. Estas pruebas poco profundas se denominan pruebas de humo. Ayuda a establecer las expectativas mínimas de que la API esté activa y responda, pero también es esencial verificar los cuerpos de respuesta. Los consumidores de API esperan propiedades específicas en el objeto JSON, por lo que deberían estar presentes. La API de Usuarios devuelve un el id generado para un objeto al crearlo:

```

38   it("Debe responder con el usuario creado con id = 3 como un objeto json", async () => {
39     await request(app)
40       .post("/api/usuarios")
41       .send({ nombre: "Nuevo Usuario" })
42       .expect(async (res) => {
43         expect(res.statusCode).toEqual(201);
44         expect(res.headers["content-type"]).toEqual("application/json; charset=utf-8");
45         expect(res.body).toHaveProperty("id");
46         expect(res.body.id).toBe(3);
47         expect(res.body).toEqual(expect.objectContaining({
48           id: 3,
49           nombre: "Nuevo Usuario"
50         }));
51       });
52   });

```

Por supuesto, puede usar otras bibliotecas de aserciones, como **Chai.js**. También hay diferentes formas de comprobar los cuerpos de respuesta integrados en SuperTest. Puede esperar un array de un tipo específico de

objetos. Aquí hay otro ejemplo:

A tener en cuenta cuando comparamos expectativas con **.toBe()** y **.toEqual()**

**toBe()** compara valores primitivos o verifica la identidad referencial de instancias de objetos, mientras que **toEqual()** busca una igualdad profunda.

```
expect({ name: 'john doe' }).toEqual({ name: 'john doe' }); // PASSES  
  
expect({ name: 'john doe' }).toBe({ name: 'john doe' }); // FAILS
```

## Próximos pasos

Recordemos que el objetivo de las pruebas unitarias es lograr asegurar que lo que antes funcionaba sigue funcionando luego de recibir una nueva iteración de funcionalidades / cambios. Con esto expresamos que debemos escribir las pruebas pensando en lo que queremos comprobar en cada test, no tiene sentido escribir pruebas de humo como por ahí antes mencionábamos sino que lo que sirve es escribir pruebas de acuerdo con lo que la interfaz de la API debe ofrecer y lo que los clientes de la API estarán esperando.

## Bibliografía

- Documentación de Jest (<https://jestjs.io/docs/getting-started>)