Testing en javascript

JS

create curiosity

emanuel fontalba

developing ideas

Fullstack Dev

genially

@EmanuelFontalba

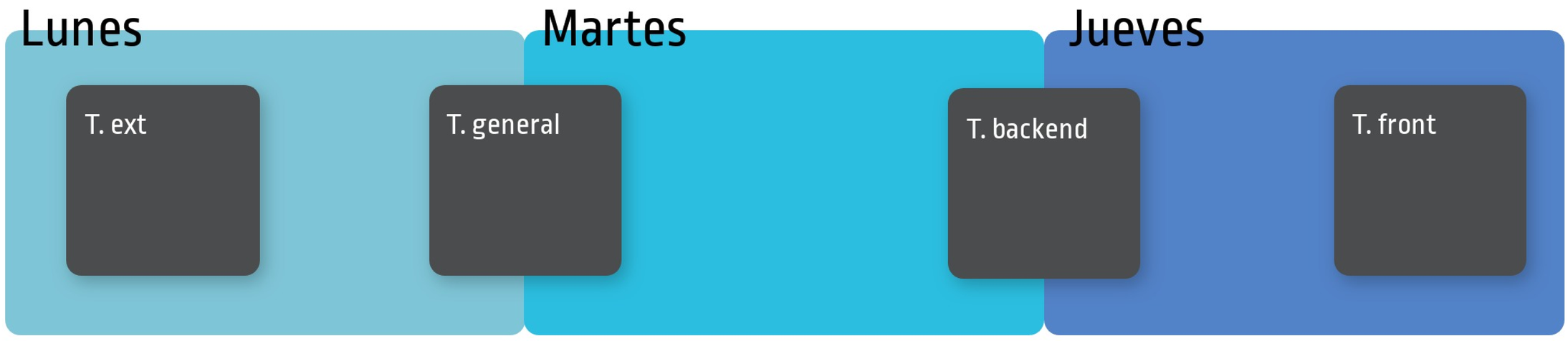
¿Qué es el testing?

Consiste en realizar pruebas sobre tu app para medir la calidad de la misma.

Tiene varios objetivos definidos:

- Encontrar bugs
- Nos permite tener más confianza en nuestro código porque, aunque nunca vamos a poder afirmar al 100% que nuestra app no contiene errores, sabemos que ese código está comprobado en su mayor parte antes de lanzarlo a producción
- Lógicamente, el testing nos ayudará a decidir si es adecuado subir ciertas funcionalidades a producción o, por el contrario, esperar más tiempo y subsanar los errores que podamos tener
- Evitar la aparición de defectos en la aplicación. Al hacer testing te acostumbras a escribir código de una manera más limpia, ordenada y con una mayor calidad.

¿Qué vamos a hacer?



Test externos

Son test realizados por herramientas externas.

Estas herramientas no puede acceder al código de nuestra aplicación, por lo que son tests que comprueban la experiencia de usuario y el correcto funcionamiento de nuestra web desde una parte visual.

Se dividen en dos tipos:

- **Manuales**: Las pruebas manuales son pruebas realizadas a mano, entrando en la app y siguiendo todos los pasos para comprobar una funcionalidad.
- **Automáticas**: Las pruebas automáticas son pruebas en las que grabamos una interacción con nuestra app y podemos especificar cada cuanto tiempo queremos que se vuelva a realizar esta comprobación

Test funcionales

Se basan única y exclusivamente en la parte funcional de nuestra aplicación.

Existen varios tipos:

- **Unitarios**
- **Aceptación**
- **Regresión**
- **Integración**

Test unitarios

Consiste en comprobar cada pequeña porción de código de nuestra app por separado para asegurarnos de su correcto funcionamiento.

Son los tests más usados habitualmente.

Estos test no deben comprobar varias cosas a la vez, cada test debe centrarse en algo muy específico, como puede ser una función, y debemos escribir un test por cada funcionalidad que queramos testear.

Puede que esa pequeña parte de código que estemos testeando, funcione diferente dependiendo de los parámetros que reciba, por ello debemos comprobar de manera individual cada uno de esos posibles casos para certificar el correcto funcionamiento de esa parte del código

Test de integración

Son pruebas en las que se comprueba el funcionamiento entero de nuestra aplicación.

Si anteriormente con los tests unitarios comprobamos cada pieza de nuestro código por separado para asegurar su correcto funcionamiento, ahora nos centramos en unir todas esas piezas y comprobar que funcionan todas en conjunto.

Es muy frecuente que aunque algo funcione individualmente de la forma esperada, a la hora de integrarlo con el resto de nuestra app esa función deje de funcionar por una razón externa a si misma.

En el caso del back-end, consistiría en realizar peticiones http a nuestra api y comprobar que responde de manera correcta con todos los parámetros que debería devolver.

Test de aceptación y regresión

Aceptación:

Los tests de aceptación suelen ser definidos y ejecutados por usuarios externos. Esto se debe a que aunque una app no contenga errores puede que no funcione de la manera en la que el usuario desea o que cierta funcionalidad sea demasiado compleja de realizar para el usuario.

Regresión:

Los tests de regresión se realizan cuando hemos introducidos cambios en nuestra app y, consisten en comprobar que partes de nuestra app, que no han sido tocadas en esos cambios que hemos introducido, siguen funcionando correctamente después de haber introducido esas modificaciones

Test no funcionales

Son tipos de pruebas que no se revisan nada sobre el aspecto funcional de nuestra app.

Hay varios tipos:

- **Seguridad:** Comprueba las posibles vulnerabilidades que puede tener nuestra app.
- **Rendimiento:** Comprueba el rendimiento de nuestra aplicación, como por ejemplo, el tiempo que transcurre desde que se recibe una petición http hasta que nuestra app responde.
- **Usabilidad:** Se centran en comprobar la usabilidad de nuestra aplicación y si realizar alguna acción en ella es intuitivo para el usuario o no.
- **Accesibilidad:** Se testea si se renderizan bien todos los elementos de una forma visual, además de comprobar si la aplicación tiene alternativas en sus funcionalidades para que puedan ser usadas por personas con algún tipo de discapacidad

Jasmine

Jasmine es framework que se basa en el comportamiento del código JavaScript dentro de una app.

No depende de ningún otro marco de JavaScript.
No requiere un DOM.
Tiene una sintaxis clara y obvia para que puedas escribir fácilmente las pruebas.

Según sus creadores, Jasmine es:

- **Rápido:** El core de Jasmine no requiere de dependencias externas, lo cual lo hace más rápido
- **Completo**: Jasmine contiene todo lo necesario para la realización de los tests por lo que no vas a necesitar nada más para poder testear tu app
- **Multiusos**: Se puede usar tanto para testear tus apps en NodeJS en el lado del servidor y, también, para testear tus apps en el front

Jasmine

```
describe("A suite is just a function", function() {
  var a;

  it("and so is a spec", function() {
    a = true;

    expect(a).toBe(true);
  });
});
```

Jest

Creado por Facebook.

En un primer lugar, nació como framework de testing para el entorno de React.

Pronto se convirtió en un framework de testing generalista.

En los último años, se ha establecido como un standard para el testing en la comunidad de Javascript.

Su primera versión fue lanzada el 14 de mayo de 2014

Jest

```
53  describe("Creating a new group", async function() {
54    const group = await makeAuthenticatedQuery(createGroup, {
55      iconUrl: "",
56      name: "",
57      description: ""
58    });
59
60    test("should exist", function() {
61      expect(group.id).toBeUndefined();
62    });
63  });
64
```

Jest

Como hemos visto en la imagen anterior, el método describe nos permite crear un contexto único en el que ejecutar nuestros test. Recibe una cadena de texto que, habitualmente es lo más descriptiva posible, y una función donde dentro crear nuestros test.

Para crear nuestro test usaremos el método test, que como describe, recibe una cadena de texto descriptiva sobre el test que se va a realizar y una función donde escribir nuestro test.

Dentro de esta última función ejecutamos la parte de código que queremos testear y con el método expect podemos comprobar el resultado de este con el resultado real que esperamos.

Jest

Cabe destacar que la sintaxis de Jest es muy legible, sólo con leer el expect ya sabemos que es lo que se está comprobando exactamente. Esto favorece la facilidad para realizar los tests sin necesitas estar acudiendo a la documentación para comprobar que un método hace exactamente lo que queremos.

Posteriormente, entraremos más en detalle con Jest

Mocha

Se describe como un framework de testing tanto para NodeJS como para el navegador.

Su primera versión se lanzó el 22 de noviembre de 2011.

Ejecuta los tests en serie, es decir, hasta que no finaliza uno no pasa al siguiente facilitando que la detección de fallos en nuestros tests sea más precisa.

En principio, su punto fuerte fue la posibilidad de ejecutar acciones asíncronas en los tests y poder comprobar su resultado.

Es un duro competente de Jest.

Permite usar cualquier dependencia para la comprobación de los test, por ejemplo assert.

Mocha

```
1   const should = require("should");
2   const request = require("request");
3   const expect = require("chai").expect;
4   const baseUrl = "https://swapi.co/api";
5   const util = require("util");
6
7   describe('returns luke', function(){
8     it('returns luke', function (done) {
9       request.get({url: `${baseUrl}/people/1/`},
10        function(error, response, body) {
11          expect(response.statusCode).to.equal(200);
12          done();
13        }
14      )
15    })
16  });
```

Mocha

Como hemos visto en la imagen anterior, el método describe nos permite crear un contexto único en el que ejecutar nuestros test. Recibe una cadena de texto que, habitualmente es lo más descriptiva posible, y una función donde dentro crear nuestros test.

Para crear nuestro test usaremos el método it, que como describe, recibe una cadena de texto descriptiva sobre el test que se va a realizar y una función donde escribir nuestro test.

Dentro de esta última función ejecutamos la parte de código que queremos testear y con el método assert podemos comprobar el resultado de este con el resultado real que esperamos

Supertest

La versión 0.1 fue lanzada el 2 de julio de 2012 pero no liberaron la versión 1.0 hasta el 11 de mayo de 2015.

Se define como un framework que permite una abstracción del API usando peticiones HTTP.

Se usa junto con Mocha o Jest para comprobar el resultado exacto de las peticiones.

Se una para realizar test de integración.

Hoy en día, no debería haber ninguna app sin implementar test con supertest porque gracias a él puedes comprobar lo que devuelve tu API en todo momento y asegurarte de que no va a tener ningún comportamiento inesperado.

Supertest

```
import request from 'supertest';
import app from './app';

test('Hello world works', async () => {
    const response = await request(app.callback()).get('/');
    expect(response.status).toBe(200);
    expect(response.text).toBe('Hello World');
});
```

Supertest

Dentro de un test normal de Jest o Mocha, bastaría con llamar al método request y pasarle un HttpServer.

Disponemos de método GET, POST, PUT, DELETE y PATCH que, pasándole una url, nos permite realizar todo tipo de petición HTTP contra nuestra API.

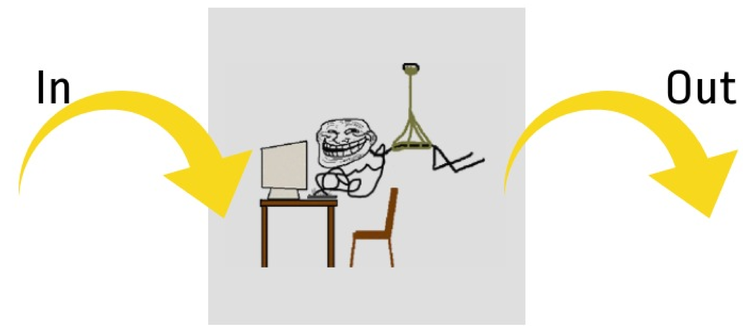
Los demás métodos nos permiten crear los headers y el body de nuestra petición.

Entraremos más en detalle en la parte práctica.

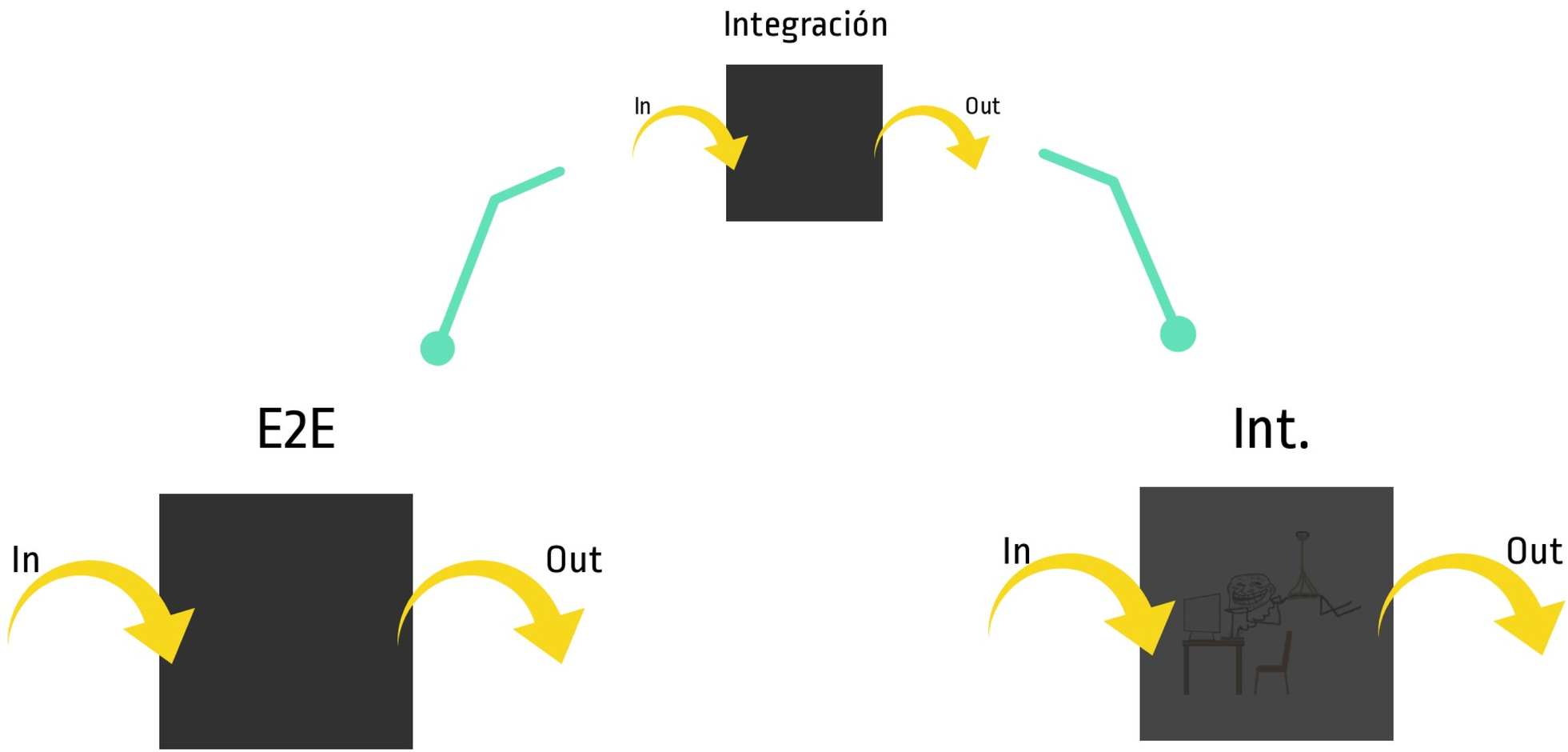
Resumen hasta ahora

Integración

Unitarios



Resumen hasta ahora



Testing en front-end

¿Realmente esto existe?¿Vas a testear una vista sin lógica? El testing es front-end es una práctica no demasiado habitual por desgracia.

En React, por ejemplo, se testean nuestros componentes y el funcionamiento del mismo.

Aunque tengamos un componente que sólo pinte un texto en color rojo y dicho texto se reciba por las props, es adecuado testearlo para asegurarnos de que se pinta correctamente.

También se usa para testear que nuestros llamadas al API se ejecutan correctamente y se realizan las transformaciones adecuadas sobre los objetos que usaremos en nuestra app.

Testing en back-end

Se testea absolutamente todo, desde transformaciones de objetos, funciones que usemos para alguna tarea específica, servicios y las respuestas de nuestro API.

Aunque sea muy repetitivo, es muy importante la realización de testing en el back-end, en esta parte solemos tener toda la lógica de nuestro negocio, así como la gestión de pagos o de usuarios en las que un bug cualquiera puede ser crítico y afectar al comportamiento de nuestra app.

En el mundo laboral, su uso está más extendido que en aplicaciones para el front-end.

Testing en el mundo real

¿Realmente se testean todas las aplicaciones y todas las partes de la misma?

Testing en el mundo real

Por supuesto, todas las empresas de nivel implemente tests en todas sus apps para certificar la calidad de las mismas y prevenir cualquier tipo de error.

Empresas con pocos ingresos, sobretodo consultoras, suelen ser muy tendentes a no implementar tests para reducir los tiempos en el desarrollo.

A la larga, el ahorro en tiempos de desarrollo es un arma de doble filo, nadie es perfecto y los bugs empezarán a aparecer con el uso de la app. Dependiendo de la cantidad y la gravedad de estos bugs el ahorro realizado previamente se termina perdiendo y lo normal es que el presupuesto del producto se quede en negativo.

La realización de test debería ser aproximadamente el 20% del tiempo de la estimación de una funcionalidad, algo que no siempre es habitual por la misma razón que hay empresas que no realizan tests.

Mocks

En POO se conoce a Mock como a los objetos que imitan el comportamiento de objetos reales de una forma controlada Se usan para probar otros objetos en tests unitarios que esperan ciertas respuestas de alguna librería, base de datos o de una clase y esas respuestas no son necesarias para la ejecución de nuestra prueba.

Por ejemplo, estamos testeando un servicio que devuelve todos los usuarios de la app pero este servicio llama al logger e imprime algo. En primer lugar, testeando no necesitaríamos ver que se imprime eso por consola, por lo tanto, podemos sustituir nuestro logger por un mock de nuestro logger que imite su comportamiento pero no nos imprima nada.

También puede ser una llamada a base de datos que no necesitas recibir datos reales, con algo que imite esa respuesta ya tendrías para comprobar lo que necesitas.

Mocks

Cada framework de test implementa sus mocks de una forma. En jest podemos crear mocks de cualquier cosa.

Podemos crear un mock de una clase, una dependencia externa que se encuentre en el node_modules, etc...
Para crear mocks de una clase en javascript bastaría llamar al método mock de Jest.

```
import SoundPlayer from './sound-player';
import SoundPlayerConsumer from './sound-player-consumer';
jest.mock('./sound-player'); // SoundPlayer is now a mock constructor

beforeEach(() => {
  // Clear all instances and calls to constructor and all methods:
  SoundPlayer.mockClear();
});
```

Mocks

En Mocha, no disponemos de una forma de crear mocks pero podemos usar alguna dependencia externa que nos lo permita como Sinon.JS.

Bastaría con llamar al método fake para crear el mock del objeto o función que queramos.

```
function once(fn) {
    var returnValue, called = false;
    return function () {
        if (!called) {
            called = true;
            returnValue = fn.apply(this, arguments);
        }
        return returnValue;
    };
}
```

```
it('calls the original function', function () {
    var callback = sinon.fake();
    var proxy = once(callback);

    proxy();

    assert(callback.called);
});
```

Coverage

El coverage es una medida de calidad de nuestras pruebas unitarias.

Gracias a esto se pueden sacar varias conclusiones:

- Podemos necesitar más tests.
- Hay código en la app que, actualmente, no se usa y por lo tanto se puede eliminar.
- ¿Entonces para testear bien una aplicación hay que tener el coverage al 100%?
- ¿Haría falta testear una función que recibe una string y evalúa esa string con un switch?¿Habría que testear cada una de la salidas posibles de esa función?

Coverage

En Jest podemos ver nuestro coverage ejecutando nuestros tests con el flag --coverage, esto nos devuelve una tabla en terminal donde se especifica el porcentaje de código que tenemos testeado en cada uno de los archivos de nuestra app.

También, en la configuración podemos especificar un mínimo para que hasta que nuestros tests no superen ese porcentaje no sea dado por válido.

Jest también nos devuelve en coverage en forma de fichero html, mostrando más información acerca del coverage de nuestros tests.

All files

84% Statements 21/25 **100%** Branches 6/6 **100%** Functions 13/13 **84%** Lines 21/25

File	Statements		Branches		Functions		Lines	
src	33.33%	2/6	100%	0/0	100%	1/1	33.33%	2/6
src/component/counter	100%	5/5	100%	0/0	100%	5/5	100%	5/5
src/component/header	100%	2/2	100%	0/0	100%	1/1	100%	2/2
src/component/headline	100%	5/5	100%	2/2	100%	2/2	100%	5/5
src/component/login	100%	7/7	100%	4/4	100%	4/4	100%	7/7

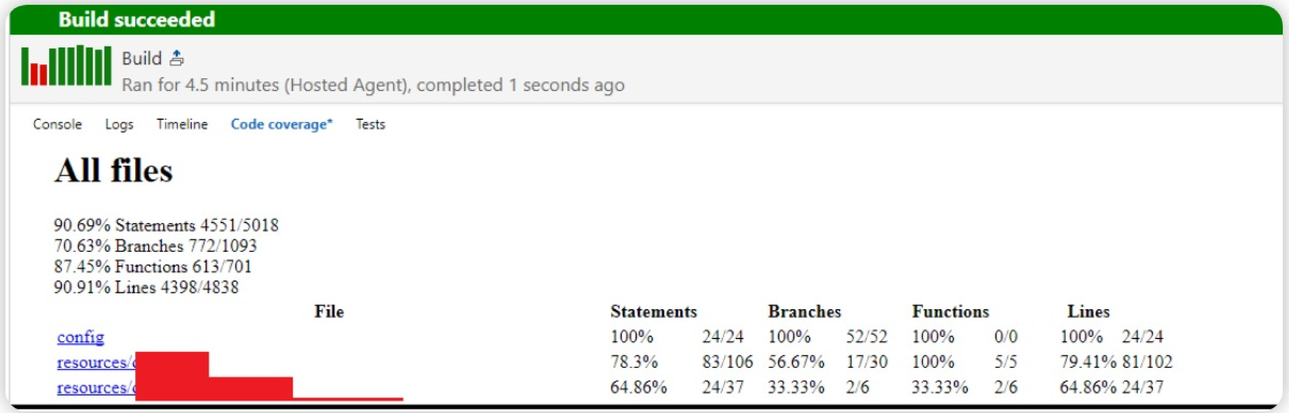
Coverage

En Mocha, como para la comprobación de los tests necesitamos una librería externa llamada Istanbul.

Bastaría con instalar la dependencia y cambiar el comando de ejecución de nuestros tests por el siguiente.

```
{
  "scripts": {
    "test": "nyc mocha"
  }
}
```

Coverage



Build succeeded

Build

Ran for 4.5 minutes (Hosted Agent), completed 1 seconds ago

Console Logs Timeline Code coverage* Tests

All files

90.69% Statements 4551/5018
70.63% Branches 772/1093
87.45% Functions 613/701
90.91% Lines 4398/4838

File	Statements		Branches		Functions		Lines	
config	100%	24/24	100%	52/52	100%	0/0	100%	24/24
resources/	78.3%	83/106	56.67%	17/30	100%	5/5	79.41%	81/102
resources/	64.86%	24/37	33.33%	2/6	33.33%	2/6	64.86%	24/37

TDD

Sus siglas significan Test Driven Development.

Es una forma de desarrollo que consiste en la realización de los tests en primer lugar, desarrollo de la funcionalidad y refactorizar.

Una descripción más sencilla es crear el test y que falle y desarrollar la funcionalidad hasta que el test esté en verde, posteriormente si cambia esa funcionalidad habría que adaptar primero el test y que falle hasta que realicemos la refactorización.

Aporta una visión más global de la funcionalidad y ayuda a que no se nos olviden ciertos aspectos de lo que vamos a desarrollar.

Su uso aporta además una mejor visión del código que desarrollamos, acostumbrándonos poco a poco a desarrollar de una más adecuada y sin dejar muchas “ñapas” por nuestro código.

TDD

PERDÓN POR EL TOCHO QUE VIENE AHORA

TDD

Para que un test sea realmente útil, TDD especifica una serie de reglas que han de cumplirse:

- Tener bien definidos los requisitos de la funcionalidad a realizar
- Un mala definición de los requisitos provocaría que no se siguiera TDD de forma adecuada e implicaría una pérdida innecesaria de tiempo
- Contemplar todos los casos posibles, tanto de éxito como de error en los criterios de aceptación de la funcionalidad
- Cómo vamos a diseñar el test
- Para realizar un buen test debemos ceñirnos a testear únicamente la lógica de cada elemento, utilizando mocks para abstraernos de otras posibles capas o servicios que necesitemos utilizar. Esto es flexible, por ello es el desarrollador el encargado de decidir si es mejor utilizar mocks o usar los servicios reales en el test
- Qué queremos probar
- Esto implica también el punto de vista del desarrollador, ya que cada uno puede pensar que se debería testear sólo una cosa en especial y otro puede pensar que deben testearse más
- ¿Cuántos test son necesarios?
- El número de test nunca está especificado, se basa en el número de casuísticas que contemple nuestra funcionalidad

Principios en los que se basa TDD

Algunos de los principios en los que se basa TDD son los conocidos como principios SOLID:

- Principio de responsabilidad simple (Single Responsability Principle)
  - Una clase o módulo debe tener una única responsabilidad
- Principio de abierto/cerrado (OCP)
  - Una clase debe permitir ser extendida sin necesidad de ser modificada
- Principio de sustitución de Liskov (LSP)
  - Si una función recibe un parámetro de un tipo, esta función debe ejecutarse correctamente si recibe un parámetro de ese tipo de alguna de sus subclases
- Principio de segregación de interfaces (ISP)
  - No debemos obligar a clases o interfaces a depender de otras que no necesitan
- Principio de inversión de dependencia (DIP)
  - Son técnicas para lidiar con las colaboraciones entre clases produciendo código limpio y reutilizable. El módulo A no debe depender del módulo B para su correcto funcionamiento

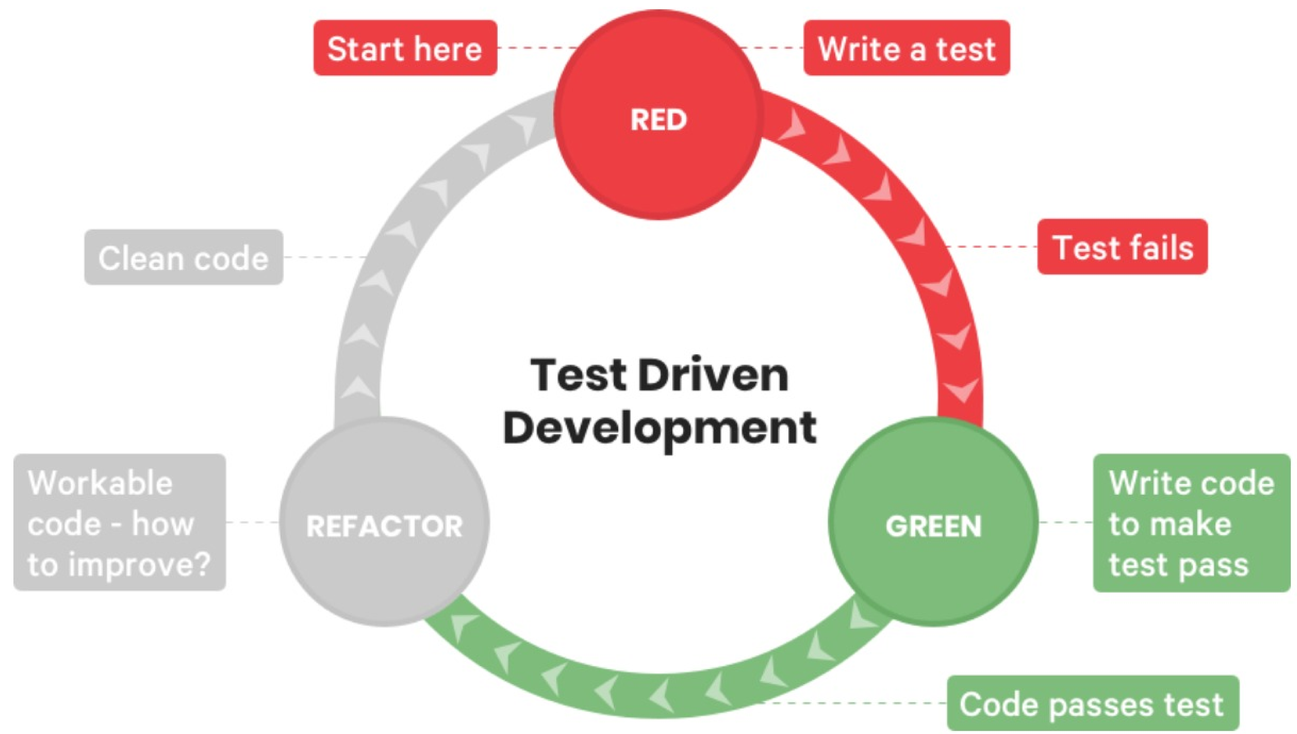
Beneficios de usar TDD

- Mayor calidad en el código desarrollado
- Diseño orientado a las necesidades
- Simplicidad
- Menos redundancia
- Mayor productividad al necesitar menor tiempo de debugging
- Se reduce el número de errores

Contras de usar TDD

- Gran curva de aprendizaje
- Perder la visión general del proyecto
- Errores no identificados en el alcance de la funcionalidad
- Uso con bases de datos
- Es difícil de implementar en el front-end, puesto que está diseñado para el testeo de la lógica de negocio
- Gran inversión de tiempo

TDD


Start here
Write a test
RED
Test fails
Clean code
Test Driven Development
Workable code - how to improve?
REFACTOR
GREEN
Write code to make test pass
Code passes test

Otros procesos de desarrollo de software basados en testing

BDD
Behavior Driven Development

El objetivo de BDD es un lenguaje específico que permite describir un comportamiento en tu app sin importar cómo ese comportamiento está implementado

ATDD
Acceptance Test Driven Development

ATDD busca que lo que se esté haciendo se haga de forma correcta pero también que lo que se hace es lo correcto a hacer

create curiosity

emanuel fontalba

developing ideas