TDD con Javascript

KEEPCODING
Tech School

Contenido

- Qué es el testing
- Tipos de testing
- Herramientas para escribir y ejecutar tests
- TDD
- *Extra: Jest PRO*

Qué es el testing

Qué es el testing

Es una **buena práctica** de los desarrolladores para dotar de robustez y solidez a las aplicaciones que creamos.

¿Es necesario entonces?

¿Puede un código no tener conjunto de tests?

Es una cuestión **CULTURAL**

Objetivos del testing

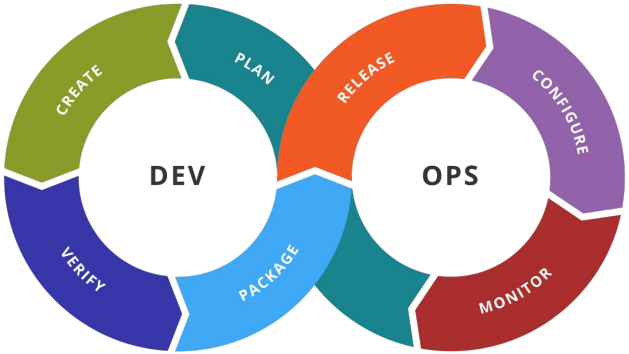
- Encontrar **bugs**
- Nos permite tener más **confianza en el código.** Aunque nunca vamos a poder afirmar al 100% que nuestra app no contiene errores, sabemos que ese código está comprobado en su mayor parte antes de lanzarlo a producción.
- Lógicamente, el testing nos ayudará a **decidir si es adecuado 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.

Relación con el PM

- **El testing** es una pieza relevante en cualquier tipología de gestión de proyecto. En la cadena de producción, sea cual sea, siempre se puede/debe introducir la fase de testing.
- Ejemplos de la relación entre el testing y tipos básicos de gestión de proyectos
  - Waterfall
  - Agile
  - DevOps

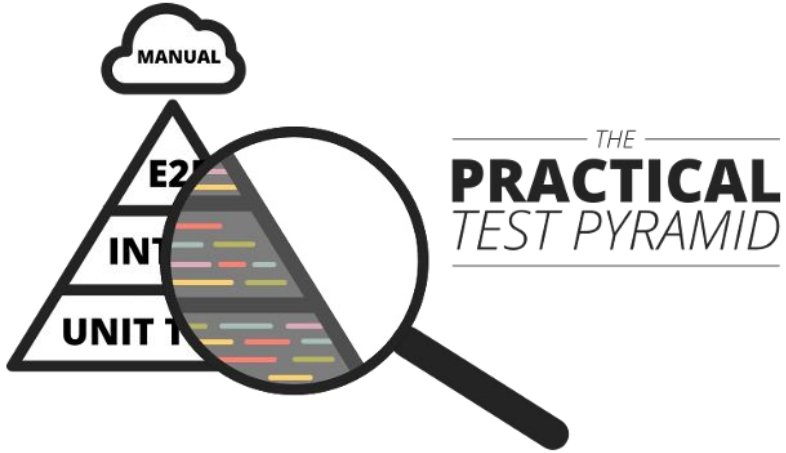
Relación con DevOps

- Es un elemento imprescindible en el loop infinito de producción.
- Como el valor añadido de DevOps pasa por la automatización de procesos, es obligatorio contar con elementos de testing.



Tipos de testing

Tipos de testing



https://martinfowler.com/articles/practical-test-pyramid.html

Tests 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: Se siguen una serie de pasos y se comprueba la funcionalidad
  - Automáticas: Grabamos una interacción con nuestra app y podemos especificar cada cuanto tiempo queremos que se vuelva a realizar esta comprobación
- Ejemplo: https://www.pingdom.com/

Tests funcionales

- A diferencia de los tests externos, estos se basan en la parte funcional de nuestra aplicación, por lo tanto, testeamos nuestro código.
- Tipos de tests funcionales:
  - Unitarios
  - Integración
  - e2e

Tests unitarios

- Son los tests más usados habitualmente (**jest**, mocha, jasmine…).
- Consiste en comprobar cada pequeña porción de código de nuestra app por separado para asegurarnos de su correcto funcionamiento.
- **Cometido único**: Estos test no deben comprobar varias cosas a la vez, cada test debe centrarse en algo muy específico y debemos escribir un test por cada funcionalidad que queramos testear.
- **Múltiples casos**: 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.

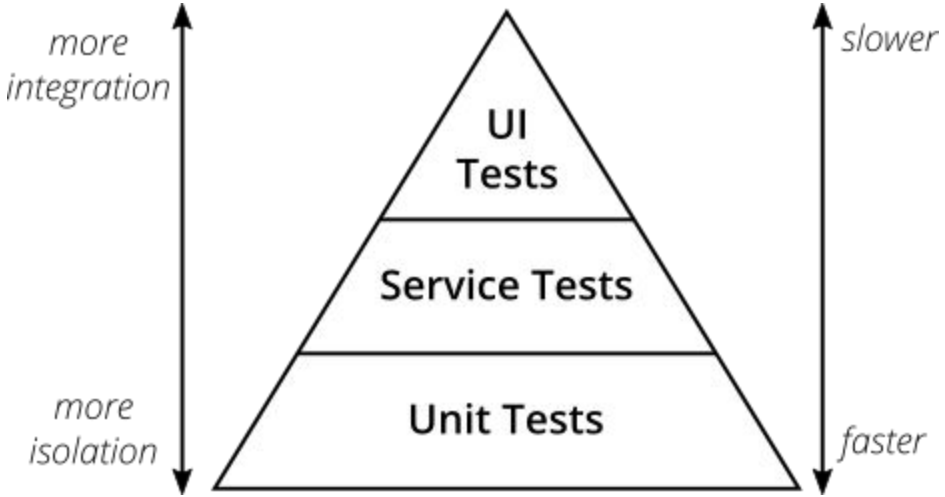
Tests de integración

- Son pruebas en las que se comprueba el funcionamiento entero de nuestra aplicación.
- Con unit testing testeamos cada una de las piezas. Ahora nos centramos en unir **dos o más** de 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 sí 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.

Tests E2E

- Son tests que replican comportamientos de usuario.
- Se conocen también como pruebas de alto nivel.
- Tienen por objetivo verificar que el sistema responde de forma correcta a los comportamientos del usuario (ej: login de un usuario, añadir items al carrito de la compra, pasarelas de pago, etc).
- Son costosas de producir y de mantener. Se recomienda tener pocas en general.

Resumen test funcionales



https://martinfowler.com/articles/practical-test-pyramid.html

Tests no funcionales

- Son tests que no tienen por objetivo analizar la funcionalidad del código.
- Tipos de tests funcionales:
  - Seguridad
  - Rendimiento
  - Usabilidad
  - Accesibilidad

Tests de seguridad

- El objetivo es comprobar las posibles vulnerabilidades que pueda tener nuestra app.
  - Paquetes y dependencias.
  - Encriptación.
  - Datos sensibles.

Tests de rendimiento

- El objetivo es comprobar la performance de nuestro código.
  - TTF (time to first byte).
  - Cálculo de costes de operación.

Tests de usabilidad

- Conocer si determinadas acciones realizadas por usuarios reales son intuitivas o no, si hay impedimentos, etc.

Tests de 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.

Herramientas

Intro

Podemos distinguir dos tipos de herramientas: imprescindibles, frameworks de testing (o herramientas técnicas).

- Imprescindibles:
  - Conocer el lenguaje de programación que se va a testear.
  - Conocer el código que se va a testear.
  - **Conocer las funcionalidades de nuestra aplicación.**
- Software:
  - Tests unitarios
  - Testing funcional
  - Tests e2e

Software

- Unit testing:
  - Jasmine
  - **Jest**
  - Mocha
  - Chai
- Integración
  - **Supertest**
- E2E
  - Selenium
  - **Puppeteer**
  - Protractor (Angular)
  - Playwright

https://npmtrends.com/chai-vs-jasmine-vs-jest-vs-mocha

Unit testing: Jest

- Desarrollado por facebook (igual que React).
- 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.
- Destaca por:
  - Sintaxis clara
  - Contextos únicos
  - Tests descriptivos

https://jestjs.io/es-ES/docs/getting-started

Unit testing: Jest

```
describe("Creating a new group", async function() {
  const group = await makeAuthenticatedQuery(createGroup, {
    iconUrl: "",
    name: "",
    description: ""
  });

  test("should exist", function() {
    expect(group.id).toBeUndefined();
  });
});

```

Integration testing: 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.

https://github.com/ladjs/supertest#readme

Integration testing: Supertest

```
const request = require('supertest')
const app = require('../server')
describe('Post Endpoints', () => {
  it('should create a new post', async () => {
    const res = await request(app)
      .post('/api/posts')
      .send({
        userId: 1,
        title: 'test is cool',
      })
    expect(res.statusCode).toEqual(201)
    expect(res.body).toHaveProperty('post')
  })
})
```

Testing E2E: Puppeteer

- Es una herramienta de automatización de tests generando respaldado en Chrome.
- Lo que conseguimos es usar el developer tools de chrome.

https://pptr.dev/

```
const puppeteer = require('puppeteer');

(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto('https://example.com');
  await page.screenshot({path: 'example.png'});

  await browser.close();
})();
```


Resumen

Resumen

- El uso de testing está cada vez más integrado en las cadenas de producción y en los diferentes modelos de gestión de proyectos.
- Existen diferentes tipos de testing. Cada tipo de testing tiene su propio cometido.
- Para cada tipo de testing podemos contar con diferentes herramientas.
- Podemos testear tanto elementos de backend como de frontend.

Resumen

- ¿Se debe testear todo?
- ¿Es viable dedicar recursos al testing?

Beneficios del testing:

- Robustez
- Agilidad
- Mejora la documentación
- Se destila la delegación de responsabilidades

Contras del testing:

- Necesita recursos (tiempo, organización… en definitiva, €)

Unit testing

“ Talk is cheap. Show me the code. ”

Linus Torvalds

Jest PRO

Contenido

Conceptos que vamos a cubrir

- Tunning Jest
- Hooks
- Manejo de Excepciones
- Testear promesas
- Mock
- Coverage

Tunning Jest

Jest dispone de un buen montón de matchers, pero a veces para la lectura y testeo rápido de unit testing, iría bien poder expandir las capacidades de Jest. Esto lo conseguimos tocando la configuración.

Ejemplos:

- axios-mock-adapter
- jest-extended

Hooks

Podemos customizar acciones que se realicen a cada test, antes o después y por cada test suite también.

Estos son:

- beforeAll
- beforeEach
- afterAll
- afterEach

Manejo de excepciones

En nuestro código en muchas ocasiones tenemos que gestionar excepciones.

Jest nos permite evaluar los tests cuyas funciones o métodos que estamos evaluando lanzan excepciones.

La particularidad es que lo que le vamos a pasar a expect() será un callback y no un valor calculado.

Testear promesas

Habitualmente nuestro código necesita datos externos, como llamadas a apis de terceros, que se devuelven promesas.

Dado que nosotros estamos programando usando esas apis, lo que deberemos hacer es comprobar que **nuestro código** funciona (pasa los tests) correctamente usando esos recursos de terceros.

No debemos testear los recursos en sí

Testear promesas

Jest nos provee un par de soluciones:

- Invocar aquellas funciones que contengan promesas y una vez se resuelven, testear su correcta manipulación, transformación, etc.
- Utilizar los métodos .resolves y .rejects para evaluar cuando la promise resuelve o rechaza, respectivamente.

Mocks

- 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.
- Ejemplos:
  - Devolver registros de una DB
  - Insertar elementos en una DB
  - Llamadas a apis de terceros que consumen por llamada
  - Imitar registros de actividad en un blog.

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

- En cambio, si usamos Mocha no hay una manera directa de crear mocks, sino que deberíamos apoyarnos en librerías externas como Sinon.js

Mocks

Gracias al potencial de los mocks, podemos tener métricas de:

- Las veces que se llama una función.
- Los parámetros con los que se ha llamado a dicha función.
- El output que haya generado la llamada al mock

Y también podremos modificar su comportamiento.

En jest, usaremos normalmente:

- .mock: para cargar nuestros propios mocks.
- .fn: para generar funciones mock desde 0.
- .spyOn: para generar también funciones mock de una función ya
- existente.

Coverage - Jest

- 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 - Jest

- 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.

TDD

Contenido

- Qué es el TDD
- Otras aproximaciones al TDD
  - BDD
  - ATDD

TDD - Qué es

TDD === “Test Driven Development”.

- Es una forma de desarrollo **iterativa** 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.

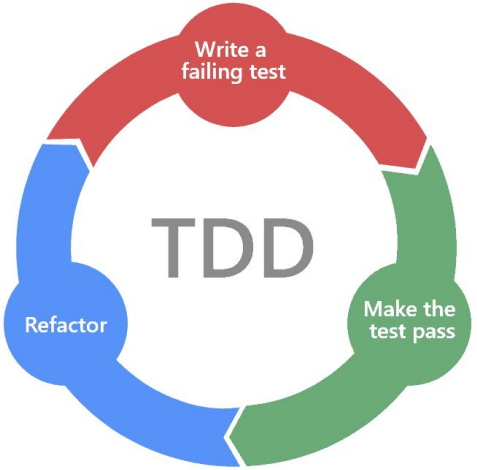
TDD - Las leyes

Robert C. Martin describe la esencia del TDD como un proceso que atiende a las siguientes tres reglas:

- No escribirás código de producción sin antes escribir un test que falle.
- No escribirás más de un test unitario suficiente para fallar (y no compilar es fallar).
- No escribirás más código del necesario para hacer pasar el test.

TDD - Ciclo Red-Green-Refactor

1. Red: Escribimos un test que falle.
    a. Puede ser unit o integración
2. Green: Implementamos lo necesario
3. Refactor: Analizar si podemos mejorar el código que hemos hecho
4. Cerrado el ciclo, pasamos al siguiente requisito.



TDD - Reglas (I)

TDD especifica una serie de reglas que han de cumplirse:

1. Tener bien definidos los requisitos de la funcionalidad a realizar.
    a. Un mala definición de los requisitos provocaría que no se siguiera TDD de forma adecuada e implicaría una pérdida innecesaria de tiempo.
2. Contemplar todos los casos posibles, tanto de éxito como de error en los criterios de aceptación de la funcionalidad.

TDD - Reglas (II)

3. Cómo vamos a diseñar el test.
    a. 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
4. Qué queremos probar.
    a. 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.
5. ¿Cuántos test son necesarios?
    a. El número de test nunca está especificado, se basa en el número de casuísticas que contemple nuestra funcionalidad.

TDD - Principios

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

SOLID:

- (S) Principio de responsabilidad simple (Single Responsability Principle)
  - Una clase o módulo debe tener una única responsabilidad
- (O) Principio de abierto/cerrado (OCP)
  - Una clase debe permitir ser extendida sin necesidad de ser modificada
- (L) 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
- (I) Principio de segregación de interfaces (ISP)
  - No debemos obligar a clases o interfaces a depender de otras que no necesitan
- (D) 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.

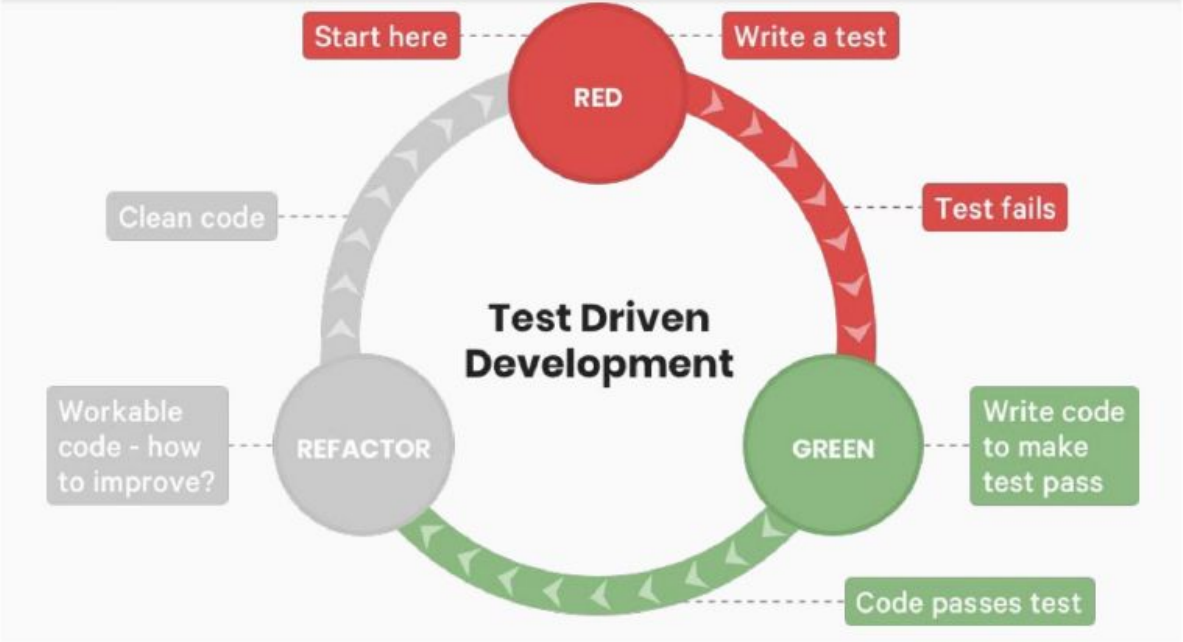
TDD - Beneficios

- Mayor calidad en el código desarrollado.
- Diseño orientado a las necesidades.
- Simplicidad. El código, no los tests.
- Menos redundancia.
- Mayor productividad al necesitar menor tiempo de debugging.
- Se reduce el número de errores.

TDD - Contrapartidas

- Gran curva de aprendizaje.
- Se puede perder la visión general del proyecto.
- Errores no identificados en el alcance de la funcionalidad.
- Si se usa mal, podemos afectar servicios como 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 - Resumen



Otras aproximaciones al TDD

BDD

BDD === “Behavior Driven Development”.

- Es un proceso de desarrollo de software basado en el testing.
- BDD busca un lenguaje común para unir la parte técnica y la de negocio.
- En BDD las pruebas de aceptación son las conocidas en agile como historias de usuario
- El objetivo de BDD es un lenguaje específico que permite describir un comportamiento en tu app sin importar cómo ese comportamiento está implementado.
- Usa Gherkins para describir los test y que sean lo más descriptivo posible.

BDD - Herramientas

El framework más popular es Cucumber: https://cucumber.io/

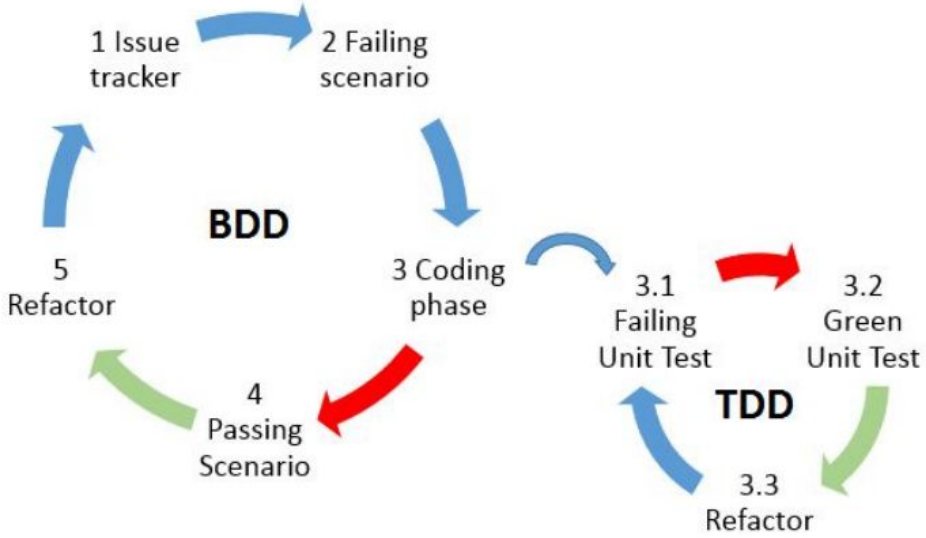
- Existe tanto para java, como javascript, Ruby y Kotlin

La sintaxis Gherkins es la siguiente:

```
Given I am on the home page
When I enter my username "myemail@gmail.com"
And I enter my password "mypassword"
Then I should see my email "myemail@gmail.com" on the dashboard
```

Permite testear totalmente la app desde el punto de vista del usuario sin importar lo que hay detrás. Ejemplo: https://cucumber.io/docs/guides/10-minute-tutorial/.

BDD - Ciclo de desarrollo



BDD - Ventajas

- No defines pruebas, defines comportamientos.
- Mejora la comunicación entre desarrolladores, testers, usuarios y la dirección
- La curva de aprendizaje es menor que la de TDD.
- Como su naturaleza no es técnica, puede llegar a un mayor público.
- Encaja perfectamente en las metodologías ágiles que se usan actualmente.
- El enfoque de definición ayuda a una aceptación común de las funcionalidades antes de empezar el desarrollo.

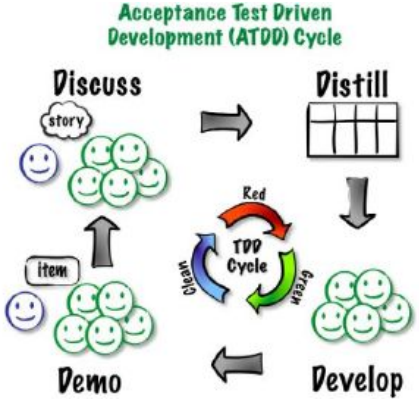
BDD - Inconvenientes

- **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.
- Es necesaria una gran comunicación entre desarrollador y cliente.
- Necesidad de tener un equipo de desarrolladores centrados en el trabajo con los clientes.

ATDD - Qué es

ATDD = “Acceptance Test Driven Development”.

- Es más cercano a un proceso que a una actividad.
- Según su definición está más cerca de BDD que de TDD.
- 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.



ATDD - Ventajas

- No trabajaremos en funciones que finalmente no se van a usar.
- Forjaremos un código listo para cambiar si fuera necesario porque su diseño no está limitado por una interfaz de usuario o por el diseño de una base de datos.
- Se puede comprobar muy rápido si se están cumpliendo los objetivos o no.
- Conocemos en qué punto estamos y cómo se progresa.
- El product owner puede revisar los test de aceptación y comprobar cuando se están cumpliendo. Esto mejora la confianza del product owner en sus desarrolladores.

KEEPCODING

Tech School

Madrid | Barcelona | Bogotá

Datos de contacto