DigitalHouse>

Certified Tech Developer
The Ultimate Degree

Back End I

INTRODUCCIÓN

1. *¿QUÉ ES BACK END?*

Básicamente, cuando hablamos de "detrás de escena", es decir, el servidor y la base de datos que ayudan a entregar información del usuario desde una interfaz, hablamos del ***back end****. Es la parte del sitio o aplicación con la que los usuarios no tienen contacto, sin embargo, resulta una parte fundamental de cualquier aplicación. Si está leyendo este texto, por ejemplo, es una señal de que la comunicación con el servidor fue exitosa y esto probablemente se deba al buen trabajo del programador* ***back-end****. El* ***back end*** *es la parte de la aplicación que se encarga de toda la lógica para que la misma funcione. Algunas de las funciones que se gestionan en esta parte son:*

- *Las peticiones del* ***front end****.*
- *Lógica de negocio.*
- *Conexión con* ***bases de datos*** *(relacionales y no relacionales).*
- *Logueo de errores, para encontrar luego, más rápidamente las soluciones.*
- *Uso de librerías del servidor web, por ejemplo, para implementar temas de caché o para comprimir las imágenes de la web.*
- *La seguridad de los sitios web que gestiona*
- *Optimización de los recursos para que las páginas sean performantes.*

Un ***back end*** *debe ser capaz de tener una capa de servicios para que el* ***front end*** *pueda consumirla y así poder realizar peticiones. En el desarrollo de esta capa hay que conectarse a una base de datos y definir que le es permitido mostrar al* ***front end****.*

2. *INTRO : TEST UNITARIOS vs TEST DE INTEGRACIÓN*

- *¿Cómo podemos comprobar que nuestro código funciona en todos los escenarios ?*
- *¿Qué pasaría si tenemos una **clase calculadora** e intentamos dividir por cero?*
- *¿Qué pasaría si tenemos una **clase persona** y en vez de un mail ponemos un número ?*

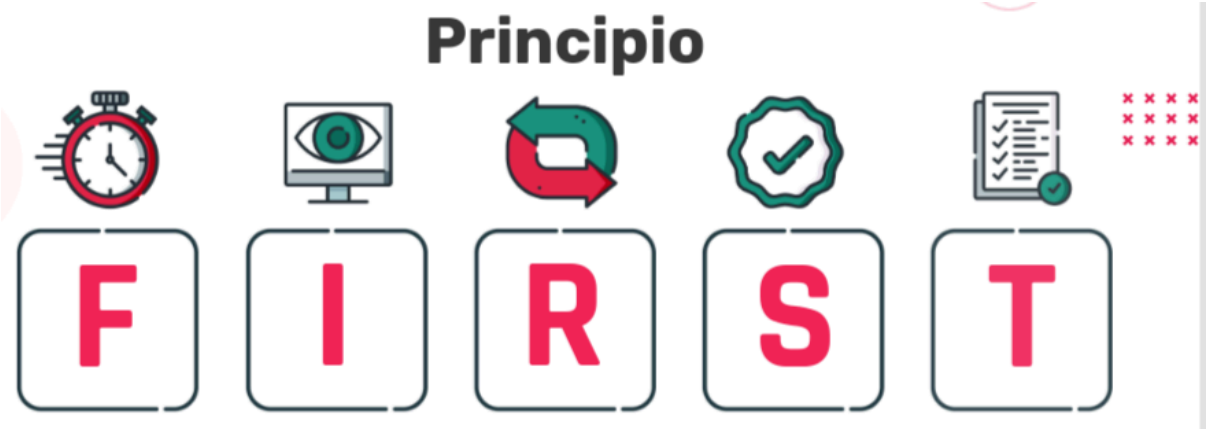
Para todo esto usaremos ***JUnit*** *, es la manera de testear nuestro código y saber que funciona correctamente.*

*Para testear nuestro código tenemos la opción de hacerlo de manera específica ósea cubriendo la parte del código mediante un **Test Unitario** o de manera global con los Test de Integración que cubren un módulo de test.*

*Recordemos que los **Test Unitarios** son aplicables a todo el código y mientras más tengamos más cubierta vamos a tener nuestra aplicación, con los **Test de Integración** podemos asegurarnos que los diferentes flujos funcionan correctamente.*

*Entonces ambos **Test** son importantes para saber si nuestro sistema cumple con las especificaciones y mejorar una parte del código. También es importante estar atento a un buen **Diseño de Test** para no tener un **Falso Positivo**.*

Continuando con los test unitarios, vamos a conocer las cinco características que deben tener para ser considerados tests con calidad. Estas características son conocidas como el principio ***F.I.R.S.T****.*



PRINCIPIO	*DESCRIPCIÓN*
Fast ***(Rápidos)***	*Es posible tener miles de test en nuestro proyecto y deben ser rápidos de correr.*
Isolated/Independent ***(Aislados/Independientes)***	*Un método de test debe cumplir con los «3A» (arrange, act, assert). Además, no debe ser necesario que sean corridos en un determinado orden para funcionar, es decir, deben ser independientes unos de los otros.*
Repeatable *(Repetibles)*	*Resultados determinísticos. No deben depender de datos del ambiente mientras están corriendo —por ejemplo, la hora del sistema—.*
Self-validating *(Auto Validados)*	*No debe ser requerida una inspección manual para validar los resultados.*
Thorough *(Completos)*	*Deben cubrir cada escenario de un caso de uso y no solo buscar una cobertura del 100%. Probar mutaciones, edge cases, excepciones, errores, entre otros.*

3. *INTRO JUNIT*

JUnit *nos da la posibilidad de generalizar lo que hay dentro de cada test.*

ANOTACIÓN	*DESCRIPCIÓN*	*EJEMPLO*
@BeforeAll	*Se ejecuta solo una vez antes de todos los* ***Test Unitarios****, y es un buen momento para inicializar variables.*	`@BeforeAll static void initAll() { }`
@BeforeEach	*Se ejecuta antes de cada* ***Test*** *y también es útil para inicializar variables comunes a todos los test.*	`@BeforeEach void init() { }`
@Test	*Sirve para generar un* ***Test Unitario.***	`@Test void regular_testi_method() { ... }`
@Disabled	Se usa cuando queremos que el ***Test*** no se ejecute.	`@Test @Disabled("este test no se ejecuta") void skippedTest() { // not executed }`
@AfterEach	Se ejecuta después de cada **Test.**	`@AfterEach void tearDown() { }`
@AfterAll	Se ejecuta después de todos los ***Test***.	`@AfterAll static void tearDownAll() { }`

Además de las anotaciones, ***JUnit*** *cuenta con algunas funcionalidades particulares para realizar validaciones dentro del Test.*

ASSERTION	DESCRIPCIÓN	EJEMPLO
assertEquals	*Funciona para comparar si dos resultados son iguales.*	`@Test void standardAssertions() { assertEquals(2, 2); assertEquals(4, 4, "Ahora el mensaje opcional de la aserción es el último parámetro.");`
assertTrue	*Funciona para saber si el resultado es **Verdadero** o **Falso**.*	`assertTrue(edad == 2, "¿Los números son iguales?");}`
Expect thrown	*Nos aseguramos que recibimos una excepción. Por ejemplo, la división por cero debería devolverme el error.*	`@Rule public ExpectedException thrown= ExpectedException.none(); @Test public void myTest() { thrown.expect( Exception.class ); thrown.expectMessage("Init Gold must be >= 0"); rodgers = new Pirate("Dread Pirate Rodgers" , -100); }`

```
1   import org.junit.jupiter.api.Test*;
2
3   import static
4   org.junit.jupiter.api.Assertions.*;
5
6   class AssertionsTest {
7
8      @Test
9      void standardAssertions() {
10         assertEquals(2, 2);
11         assertEquals(4, 4, "Ahora el mensaje opcional de
12  la aserción es el último parámetro.");
13         assertTrue(edad == 2, "¿Los números son
14  iguales?");}
15
16
17  @Rule public ExpectedException thrown=
18  ExpectedException.none();
19
20  @Test
21  public void myTest() {
22     thrown.expect( Exception.class );
23     thrown.expectMessage("Init Gold must be >= 0");
24
25     rodgers = new Pirate("Dread Pirate Rodgers" , -100);
26  }
```

DigitalHouse>

¿Para qué se utilizan las siguientes anotaciones?

Te invitamos a arrastrar las mismas con su definición.

	Es necesario anotar cada método para que JUnit lo reconozca como un test y lo ejecute.
	Permite correr el test con múltiples argumentos. Puede tomar los parámetros de diferentes fuentes, como un método, unos valores o un archivo csv.
	Deshabilitar un test para que no se ejecute, un test anotado así, será ignorado.
	Ejecuta un método antes de la ejecución de cada test.
	Ejecuta un método después de la ejecución de cada test.
	Ejecuta un método antes de la ejecución de todos los test de la clase.
	Ejecuta un método después de la ejecución de todos los test de la clase.

@AfterEach

@Test

@AfterAll

@ParameterizedTest

@Disable

@BeforeEach

@BeforeAll

¿Para qué se utilizan las siguientes anotaciones?

Te invitamos a arrastrar las mismas con su definición.

@Test	Es necesario anotar cada método para que JUnit lo reconozca como un test y lo ejecute.
@ParameterizedTest	Permite correr el test con múltiples argumentos. Puede tomar los parámetros de diferentes fuentes, como un método, unos valores o un archivo csv.
@Disable	Deshabilitar un test para que no se ejecute, un test anotado así, será ignorado.
@BeforeEach	Ejecuta un método antes de la ejecución de cada test.
@AfterEach	Ejecuta un método después de la ejecución de cada test.
@BeforeAll	Ejecuta un método antes de la ejecución de todos los test de la clase.
@AfterAll	Ejecuta un método después de la ejecución de todos los test de la clase.

¡Muy bien! Ordenaste correctamente las anotaciones.