

Test unitarios sobre la capa de servicios en Spring boot

Los test unitarios nos permiten probar componentes aislados de nuestro desarrollo y garantizar el correcto funcionamiento en este caso de los servicios los cuales manejan la logica mas importante dentro de nuestra aplicación.

Test unitarios en la capa de servicios con la tecnología spring boot

¿Qué es el testing?

El testing de software, o las pruebas de software son procesos para verificar el estado de un programa, para validar que el programa funciona correctamente y para garantizar una calidad sobre dicho software.



Realizar tests o pruebas sobre nuestro software es muy importante porque nos ayuda a encontrar problemas de manera temprana, así como también facilita la integración de nuevos módulos y comprobar fácilmente que el sistema sigue funcionando o hallar que piezas se pudieron ver afectadas y de esta manera logramos corregir esos fallos rápidamente, así que podemos decir que además de brindarle calidad al software también lo hace más eficiente.

Como objetivos principales tenemos:

- ▶ Detectar y corregir errores.
- ▶ Brindar calidad al sistema.
- ▶ Detección de errores en cualquier fase (actual, futura).
- ▶ Ayuda a garantizar los requerimientos.

Metodologías para tests TDD

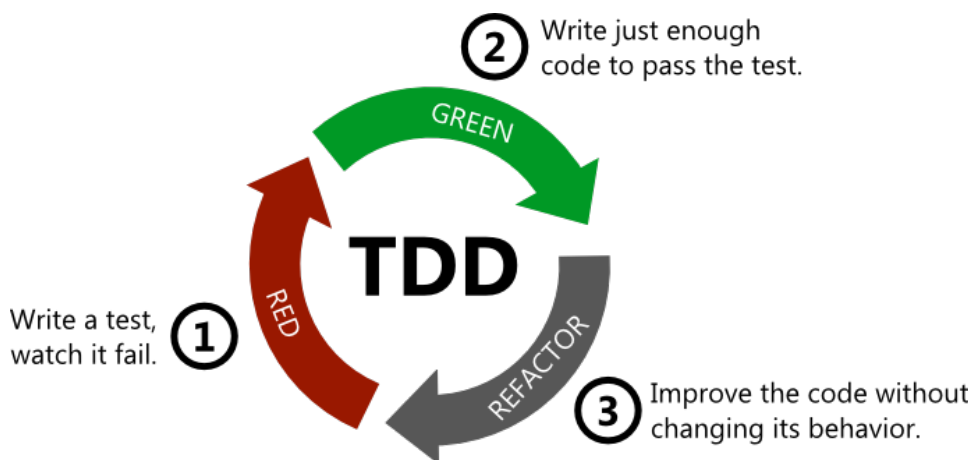
El test driven development (Desarrollo basado en pruebas) es una técnica de desarrollo de software basadas en pruebas y con un alcance sobre el desarrollador, y que se ocupa a nivel unitario o porción pequeña de aplicación en desarrollo.

En esta técnica la manera en la que se aborda el desarrollo es primero identificar los escenarios y requerimientos y diseñar las pruebas que debe cumplir nuestro código, de tal modo que en un inicio tenemos definidas todas las pruebas con todo lo que debe satisfacer nuestro código y todas fallarán, a medida que empezamos a desarrollar la funcionalidad debemos cumplir las pruebas una a una y luego organizar el código de ser necesario para tener un código limpio, refactorizado y entendible.

Aunque puede disminuir la velocidad de desarrollo, afecta positivamente en la calidad del software, pues disminuyen los errores o son más fáciles de identificar para ser resueltos, además el código resulta siendo un código reutilizable y flexible (menos redundante, menos duplicación).

Y al estar enfocado en los test pues por lo general se logra un alto nivel de cobertura entre 90 y 100 % lo cuál suele ser un estándar de satisfacción de código, realmente hay servicios y casos donde el código es muy complejo y resulta muy difícil testear a totalidad con lo cual ese margen del 10% suele garantizar o nos da la seguridad de que tenemos un código de calidad aun cuando no esté con una cobertura del 100%.

Es una metodología que se perfecciona con la practica y la experiencia pues es complicado o desafiante escribir los casos de prueba antes que el código, exige tener clara la idea a desarrollar y por ultimo es importante que se realiza la parte de test en su totalidad antes de empezar a desarrollar, no se hacen a la vez, no se hace después el test, siempre antes del desarrollo.



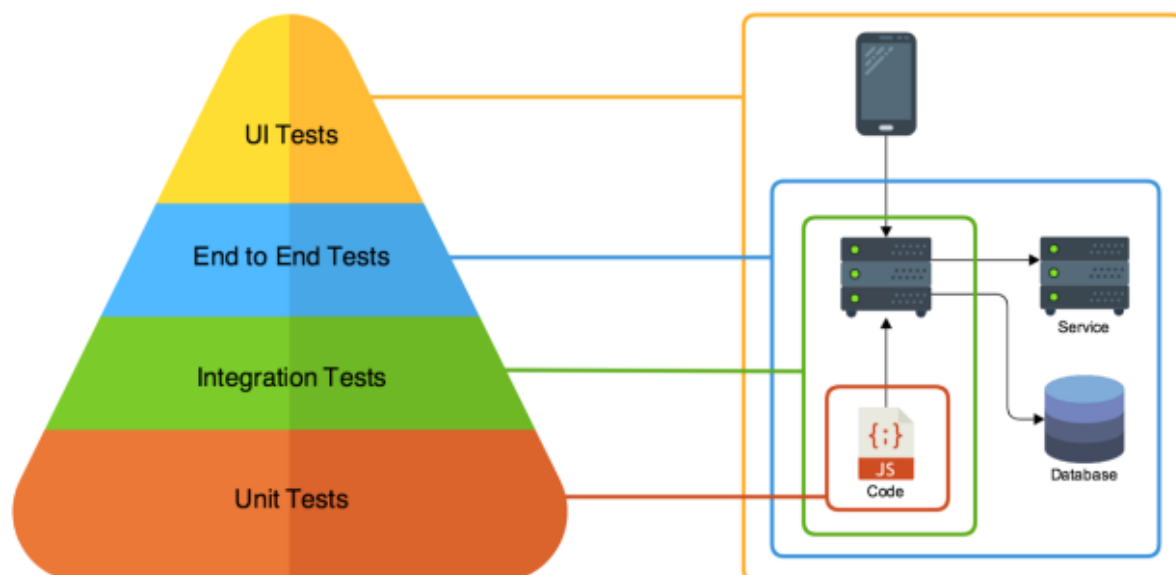
Pasos

- Creación de test basados en los requerimientos de la funcionalidad.
- Ejecución de test para comprobar que tanto se ha alcanzado.
- Escritura de código que satisfaga los test.
- Ejecución de test.
- Refactorización y organización del código.
- Todo completo ? Acabada fase de desarrollo : Agregamos más test y seguimos desarrollando (vuelta paso 1).

Tipos de test

Tenemos pruebas de funcionalidad y pruebas de rendimiento principalmente donde:

En las **pruebas funcionales** se verifica cada función del software ya sea funcionalidad unitaria, o integrada.



Como por ejemplo:

- **Pruebas unitarias.**
- **Pruebas de integración.**
- **pruebas de sistema.**

- **pruebas de interfaz.**
- **etc.**

En las **pruebas de rendimiento** se tiene en cuenta la confiabilidad (Que tan bien se comporta nuestro sistema en diferentes condiciones) usabilidad (Que tan sencillo es nuestro software, que tan fácil de usar o de aprender es) y rendimiento (que tan eficiente es nuestro software).



Como por ejemplo:

- **Prueba de rendimiento.**
- **Prueba de carga.**
- **Prueba de estrés.**
- **Prueba de volumen.**
- **Prueba de seguridad.**
- **etc.**

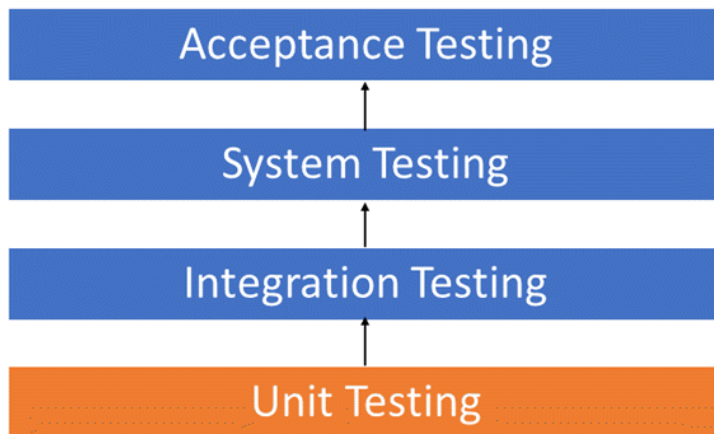
Test unitarios en profundidad

Las pruebas unitarias son una manera de validar que una unidad de código funciona correctamente, consisten en aislar una parte del código y comprobar que dicha parte o dicha unidad funciona correctamente, son la base de la pirámide de test y nos ayuda a de manera sencilla garantizar funcionalidad y detectar errores teniendo en cuenta que en esta fase de test las unidades se tratan de manera individual, es decir sin interactuar con otras unidades.

Ventajas:

- Las pruebas unitarias demuestran que la lógica del código está bien y que cumple con todos los casos de prueba.
- Aumentan la legibilidad del código, nos obliga a factorizar el código y a futuro agilizan los cambios.

- ▶ Son breves de ejecutar, se realizan en muy poco tiempo (ms) y esto garantiza que podemos ejecutar un gran volumen de las mismas.
- ▶ Aumentan la calidad de nuestro código.



Test en spring boot

Bajo el marco de trabajo de spring boot y en general en los proyectos java tenemos gran facilidad para el manejo de test, esencialmente en proyectos spring boot las funcionalidades principales y mas utilizadas se encuentran agrupadas en una dependencia llamada spring-boot-starter-test la cual por obvias razones debería tener un scope de test, con agregarla a nuestro proyecto ya tenemos las principales dependencias que son:

- ▶ **JUnit** - Que es el framework estandar para pruebas en java en la versión 5.
- ▶ **Mockito** - Que es un framework para imitar y simplificar el desarrollo de test y permite imitar comportamiento de dependencias externas.
- ▶ **Hamcrest** - Que es otro framework usado en las pruebas unitarias en java, que admite la creación de comparadores de aserciones personalizados.
- ▶ **AssertJ** - Que es una librería que provee un conjunto de métodos usados para comprobaciones en los test.

Para importar la librería debemos agregar lo siguiente en el POM.xml

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-test</artifactId>
4     <scope>test</scope>
5 </dependency>
```

Los test unitarios al ser aplicados a una única unidad se pueden considerar test aislados y en spring boot

tenemos diferentes anotaciones y manera de realizar dichos test de acuerdo a la capa de software que deseamos testear, Por ahora como primera fase de pruebas estamos interesados en test sobre la capa de servicios, aquella anotada con `@Service` pero vale aclarar que puede y debe ser testeada cada capa para considerarse un software de calidad de principio a fin, a continuación vemos un ejemplo básico de arquitectura de una aplicación y en nuestro caso al ser microservicios de tipo rest api pues dichos microservicios no manejan vistas únicamente controladores, servicios, capa de acceso a datos y base de datos.

Composición o capas de un Backend



Test unitarios capa de servicio

Para testear los servicios entonces tenemos que tener claro que suele tener dicha capa y como vamos a simularlo, además como inyectar dichos métodos desde la clase de pruebas y que anotaciones tenemos para facilitar este trabajo, también como realizar las imitaciones, y como realizar las comprobaciones, o verificaciones.

La capa de servicios por lo general está compuesta por unas inyecciones de repositorios y esto es debido a su naturaleza dónde se comunica con la capa de repositorio para el manejo de la lógica con base de datos, pues bien, estas dependencias son inyectadas por spring boot con **@Autowired**, en nuestro caso como los test se ejecutan de manera independiente y al ser test unitarios no cargan ningún contexto de aplicación debemos utilizar **mockito** para garantizar un comportamiento sobre estas dependencias, así como también inyectar el servicio a testear dentro de nuestra clase de test para tener control sobre los métodos y poder llevar a cabo el test.

Aunque spring boot ofrece sus propias anotaciones como **@MockBean** y **@Autowired**, esto nos obliga a crear un contexto de aplicación para pruebas basado en **spring boot** y hace que los test unitarios tarden un poco mas en ser ejecutados pues primero levanta dicho contexto para luego poder inyectar las dependencias y demás, esto no es necesario y podemos simplemente utilizar **Mockito** para imitar el comportamiento y no tener que trabajar instancias reales, recordemos que en las pruebas unitarias cada capa es probada de manera aislada, en test de integración ya es diferente.

Arquitectura, Anotaciones y explicación

En cuanto a la arquitectura de nuestro proyecto será la misma que la arquitectura main de desarrollo, y las clases a a testear terminarán con una notación de **test**, de esta manera si tenemos una clase

AuditoriaServiceImpl la cual deseamos testear y está en el arbol

src.main.java.co.edu.uis.project.service.auditor.impl deberemos tener entonces una rama similar que parte del árbol **src.test.java.co.edu.uis.project.service.auditor** y su nombre será **AuditoriaServiceTest**.

“

Ahora bien en cuanto a los métodos, mínimante se debe realizar un método de test por cada método del servicio, pueden ser utilizados mas métodos de test por uno solo método de servicio pero como mínimo un método de test para cada método de servicio.

Una vez creada la clase es necesario tener presente como se conformará dicha clase, que anotaciones debe llevar y demás, lo primero es saber que dependencias de terceros o de otras capas tendrá nuestro servicio, sigamos tomando como ejemplo el servicio de auditoria que tiene la siguiente estructura.

```

1  @Service
2  public class AuditorServiceImpl implements IAuditorService {
3
4
5      private IClaseSituacionAdministrativaRepository claseSituacionAdministrativaRepository;
6
7      /**
8       * Inserta un registro
9       *
10      * @param claseSituacionAdministrativa -
11      */
12      @Override
13      @Transactional
14      public ClaseSituacionAdministrativa saveClaseSA(ClaseSituacionAdministrativa claseSituacionAdministrativa) {
15
16          ...
17      }
18
19      /**
20       * Elimina un registro por id
21       *
22       * @param id -
23       */
24      @Override
25      @Transactional
26      public void deleteClaseSA(Long id) {
27          ...
28      }

```

```

29
30
31     @Autowired
32     public void setClaseSituacionAdministrativaRepository(IClaseSituacionAdmin:
33         this.claseSituacionAdministrativaRepository = claseSituacionAdministra
34     }
35 }

```

Como vemos en el servicio la única dependencia es con el repositorio de

`ClaseSituacionAdministrativaRepository` y como bien lo dice el test unitario la gracia de probar el servicio es como una unidad aislada entonces el repositorio debería tener su propio test, en las pruebas unitarias sobre los servicios asumimos que las demás unidades funcionan correctamente y para ello utilizamos los `@Mock`, los `@Mock` nos permiten imitar un comportamiento sobre dichas dependencias y simular un estado de éxito y unas acciones sin tener que trabajar con el servicio real.

Una vez que están definidos los `@Mock` pasamos a inyectar el servicio que deseamos testear y para esto hay que tener claro cuando se usa el `@Autowired` y cuando el `@InjectMocks`.

Bien, sucede que en los test unitarios como la idea es que sean test sobre unidades independientes pues no deseamos inyectar unidades funcionales reales, todo será simulado es por esto que no es necesario levantar un contexto de aplicación de spring boot para que almacene los beans y los inyecte, puesto que esto hace mas costosa la ejecución de los test debido a la carga de todo el entorno, por lo tanto no es necesario realizar configuraciones o inyecciones con spring, caso distinto a test mas complejos, como integración u otras capas y demás.

Spring boot mantiene sus propias anotaciones para manejo de test como `@Autowired`, `@Transactional` y también `@MockBean`, pero para disponer de ellas como aclaramos debemos ejecutar el test con spring boot y tarda un poco más.

Las anotaciones equivalentes proporcionadas por **Mockito** consisten en `@Mock`, la cual reemplaza el `@MockBean(spring boot)` y lo que hace es simular el **bean** que deseamos, en este caso el **bean** del repositorio, la siguiente anotación `@InjectMocks` es la que reemplazara al `@Autowired`, y lo que hace no es mas que instanciar el servicio inyectando los **Mocks** y la manera en la que inyecta los **mocks** en la unidad es primero por una inyección por **constructor**, si detecta que hay un constructor con parámetros toma el mayor y trata de inyectarle los **mocks**, en caso de no tener un constructor o que el único sea el constructor vacío por defecto entonces intenta inyectar los **mocks** por **setters** (nunca ambos a la vez) y en caso de no lograrlo pasa a inyectarlos como **propiedad**.

Igualmente se aclarará un poco más adelante.

A continuación como quedaría el test para el servicio en cuestión:

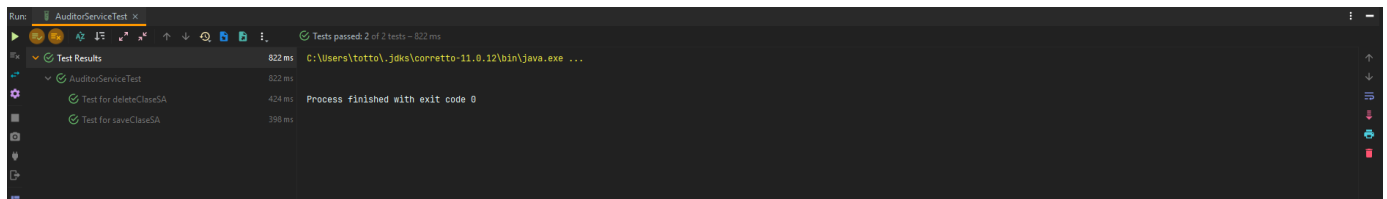

```
1  @ExtendWith(MockitoExtension.class)
2  class AuditorServiceTest {
3
4      // @MockBean remplazo de @Mock
5      // Autowired remplazo de @InjectMocks pero en un contexto de spring boot es
6      // con un @RunWith bajo springclass o motor de spring
7      // y sobre un @SpringBootTest
8      @Mock
9      private IClsaseSituacionAdministrativaRepository clsaseSituacionAdministrativaRepository;
10
11     @InjectMocks
12     private AuditorServiceImpl auditorService;
13
14     private ClaseSituacionAdministrativa clsaseSituacionAdministrativa;
15
16     @BeforeEach
17     public void setup() {
18         // clsaseSituacionAdministrativaRepository = Mockito.mock(ClaseSituacionAdministrativaRepository.class);
19         // auditorService = new EmployeeServiceImpl();
20         // inyectamos por setter el repositorio
21         clsaseSituacionAdministrativa = new ClaseSituacionAdministrativa();
22         clsaseSituacionAdministrativa.setId(1L);
23         clsaseSituacionAdministrativa.setDescripcion("Clase de prueba");
24         clsaseSituacionAdministrativa.setSenalGrupal(true);
25         clsaseSituacionAdministrativa.setIdentificador("Identificador Test");
26     }
27
28     @DisplayName("Test for saveClaseSA")
29     @Test
30     void givenASObjectWhenSaveThenReturnASObject() {
31         // Given (precondition or setup)
32
33         given(clsaseSituacionAdministrativaRepository.save(any(ClaseSituacionAdministrativa.class)))
34             .willReturn(clsaseSituacionAdministrativa);
35
36         // When (action or the behaviour that we are going test)
37         var savedClass = auditorService.saveClaseSA(clsaseSituacionAdministrativa);
38
39         // Then (verify the output and asserts)
40         assertThat(savedClass).isNotNull();
41         assertThat(savedClass.getDescripcion()).isEqualTo("Clase de prueba");
42     }
43
44     @DisplayName("Test for deleteClaseSA")
45     @Test
```

```

47     void givenIdDeleteObject() {
48         //given
49         willDoNothing().given(claseSituacionAdministrativaRepository).deleteBy:
50         //when
51         auditorService.deleteClaseSA(claseSituacionAdministrativa.getId());
52         //verity
53         verify(claseSituacionAdministrativaRepository, times(2)).deleteById(any
54
55     }
56
57 }

```

Tiempo de ejecución del test sin uso de spring



Anotaciones a nivel de clase

Ahora bien, tenemos diferentes configuraciones con las cuales trabajar una clase de prueba, algunas de ellas suponen extensiones para el manejo de anotaciones, que aunque nos hagan dependientes del framework pues agilizan el trabajo de desarrollo y la facilidad de lectura.

Tenemos `@ExtendWith()` donde especificamos de que clase deseamos extender la configuración y es útil pues habilita el uso de las mencionadas anotaciones, un ejemplo para poder utilizar las anotaciones de mockito `@Mock` y `@InjectMocks` es usando `@ExtendWith(MockitoExtension.class)`

En Junit4 teníamos `@RunWith` el cual era como el motor de ejecución y especificábamos sobre que runners se deseaba correr el test y por debajo esto hacía que funcionaran ciertas anotaciones, como lo hace el `extendWith` en Junit5.

Así como extendemos de `mockitoExtension`, hay otras como la propia de `SpringBoot`, cuál usar y cuando usarlas, bueno sabiendo que el propósito de las extensiones de `junit5` es extender el comportamiento de las clases de test o métodos pues tenemos que ver que nos ofrece cada una, la extensión de `spring`, `SpringExtension` nos da ciertas opciones pero las más fáciles de apreciar son que nos habilitan el uso de las anotaciones `@MockBean` y `@Autowired`, esto porque ahora estamos indicando que el test requiere un `Spring test context` y esta extensión nos ofrece tales opciones.

Por su parte la extensión de `mockito` nos habilita las anotaciones ya mencionadas y trabaja sin el `Spring test context`, esto hace que sean más de ejecutar los test y sencillo de manejar.

Tenemos también una última anotación `@SpringBootTest` la cual contiene la extensión de `spring boot`, y es útil

cuando manejamos test con inyección de spring y queremos disponer de mas configuraciones y opciones que define spring boot e incluso son configurables, lo que hace esta anotación es que carga el contexto completo de aplicación para test.

A nivel de métodos

A nivel de método cada método de prueba debe estar anotado con `@Test` y esto nos permitirá ejecutar dichas sentencias y lógica para probar aquellos servicios o empezar el proceso de desarrollo en caso de usar la metodología tdd.

Además como buena practica el nombre del método debe ser claro en cuanto a lo que se desea garantizar, si vamos a probar un metodo que toma un un objeto lo guarda y lo devuelve pues lo correcto sería especificarlo en el método, `createEmployeeTest()` o podemos utilizar notación **give when y then**, `givenEmployeeWhenSaveThenReturnEmployee()`, esto facilita la lectura y entender que hace el test desde su cabecera.

Ahora en las clases de test nosotros para evitar duplicación de código o para inicializar una configuración o unas acciones comunes a varios test podemos utilizar algunos metodos anotados con Anotaciones de Junit para facilitar estos procesos, en Junit5 tenemos las siguientes anotaciones:

@BeforeAll → Nos permite ejecutar cualquier lógica previa a cualquier método, se ejecuta con la clase y por lo tanto debe ser un método estático, y es como lo primero que se ejecuta al cargarse la clase, y queda disponible para todos los métodos.

@AfterAll → Siguiendo la misma lógica no permite ejecutar acciones, código, al finalizar la ejecución de clase, esto es util si deseamos por ejemplo cerrar conexiones, limpiar variables o cualquier acción que se ejecute al final de los test.

@BeforeEach → Tiene un comportamiento similar el **BeforeAll** pero ahora la ejecución se realiza inmediatamente antes de la ejecución de cualquier método, un ejemplo de uso sería refrescar alguna variable para facilitar la ejecución de test, algún contador de ejecuciones, reiniciar algún dato modificado en cada test, etc.

@AfterEach → Nos permite definir las acciones a realizar tan pronto finaliza la ejecución de un método de prueba y puede llegar a ser útil en lógicas de actualización de variables, modificadores, contadores o cualquier otra que se nos ocurra.

A nivel de dependencias - Mockito

Spring boot maneja sus propias anotaciones que también son efectivas sobre los test

@MockBean → Nos permitemockear o imitar cualquier dependencia de nuestro servicio, como no estamos interesados en pruebas de integración, aquellas dependencias del servicio como repositorios, apis externas

deberán ser simuladas, esto lo hacemos importando dichas dependencias y anotandolas con `@MockBean` y de esta manera se creará un mock en vez de la implementación.

@Autowired → Nos permite inyectar las dependencias utilizando spring y su contexto de aplicación para test, spring almacena las instancias de nuestras clases y las podemos inyectar en cualquier parte.

“

Para utilizar spring en nuestros test es importante que se agregue la anotación `@SpringBootTest`, que los mocks se inyecten con `@MockBean` y que los servicios a probar se inyecten con `@Autowired`, un ejemplo de la clase con estas anotaciones es el siguiente. en caso de necesitar otra configuración para test se puede definir un archivo properties exclusivo para test añadiendo `@TestPropertySource(value = "classpath:test.properties")` y en la carpeta `test.resources` un archivo `test.properties` con la configuración que queramos utilizar.

```

1  @SpringBootTest
2  @TestPropertySource(value = "classpath:test.properties")
3  class AuditorServiceTest {
4
5      // @MockBean remplazo de @Mock
6      // Autowired remplazo de @InjectMocks pero en un contexto de spring boot es
7      // con un @RunWith(Junit4) o @ExtendsWith(Junit5) bajo spring, con un boots
8
9      @MockBean
10     private IClsaseSituacionAdministrativaRepository clsaseSituacionAdministrativ
11
12     @Autowired
13     private IAuditorService auditorService;
14
15     private ClaseSituacionAdministrativa clsaseSituacionAdministrativa;
16
17
18     @Value("${prop}") //Variable del test.resources.test.properties
19     private String prop;
20
21     @BeforeEach
22     public void setup() {
23
24
25         clsaseSituacionAdministrativa = new ClaseSituacionAdministrativa();
26         clsaseSituacionAdministrativa.setId(1L);
27         clsaseSituacionAdministrativa.setDescripcion("Clase de prueba");
28         clsaseSituacionAdministrativa.setSenalGrupal(true);
29         clsaseSituacionAdministrativa.setIdentificador("Identificador Test");
30     }

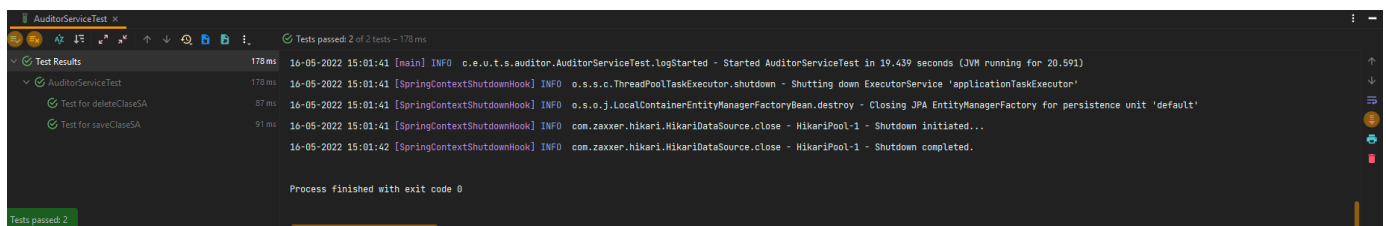
```

```

31
32 @DisplayName("Test for saveClaseSA")
33 @Test
34 void givenASObjectWhenSaveThenReturnASObject() {
35     //Given (precondition or setup)
36     //when then return - > given will
37     given(claseSituacionAdministrativaRepository.save(any(ClaseSituacionAdr
38         .willReturn(claseSituacionAdministrativa);
39
40     //When (action or the behaviour that we are going test)
41     var savedClass = auditorService.saveClaseSA(claseSituacionAdministrati
42
43     //Then (verify the output and asserts)
44     assertThat(savedClass).isNotNull();
45     assertThat(savedClass.getDescripcion()).isEqualTo("Clase de prueba");
46
47 }
48
49 @DisplayName("Test for deleteClaseSA")
50 @Test
51 void givenIdDeleteObject() {
52     //given
53     willDoNothing().given(claseSituacionAdministrativaRepository).deleteBy:
54     //when
55     auditorService.deleteClaseSA(claseSituacionAdministrativa.getId());
56     //verity
57     verify(claseSituacionAdministrativaRepository, times(2)).deleteById(any
58
59 }
60
61 }

```

A continuación podemos ver los resultados del test utilizando spring.



“

En la imagen se puede apreciar como tarda 20 segundos en iniciar spring para luego ser ejecutadas las pruebas con lo cual esta no es la recomendación en caso de no ser realmente necesario y aunque es verdad que existen más configuraciones bajo las anotaciones la idea es tener los test lo más simplificados y sencillos posibles y sí las pruebas se pueden realizar sin

spring entonces lo mejor es no usarlo.

@Mock → Es el equivalente a @MockBean y nos permite imitar el comportamiento de una dependencia para ser simulada y no afectar el concepto de test de integración.

@InjectMocks → Nos permite instanciar un servicio y le intenta inyectar las dependencias a dicha instancia, primero por constructor luego por setter y luego por propiedad

@Spy → Es similar a un @Mock pero de manera sencilla en vez de ser aplicado a una interfaz por lo general se utiliza sobre una implementación, se pueden usar comportamientos reales de dichos servicios o imitarlos y definir el comportamiento como si fuese un @Mock.

A nivel de verificación - Asserts

A continuación los métodos más usados para verificaciones, la mayoría son métodos de Junit, sin embargo tenemos `assertThat(obj o property).method()` que es proporcionado por AssertJ y nos facilita realizar muchas verificaciones.

Verificaciones más usadas.

assertAll

Para agrupar un grupo de asserts y que todos funcionen.

assertEquals

(esperado, actual)
Compara que dos valores sean iguales.

assertNotEquals

(esperado, actual)
Compara que dos valores no sean iguales.

assertFalse

(condición)
La condición dada no es verdadera.

assertNull

Verifica que el valor esperado sea **null**.

assertNotNull

Verifica que el valor esperado **no** sea nulo

assertThrows

Verifica que la ejecución lanza una excepción.

assertTrue

(condición)
La condición dada es verdadera.

assertSame

Verifica que dos referencias de objeto apunten al mismo objeto.

assertNotSame

Verifica que dos referencias de objeto **no** apunten al mismo objeto.

assertThat(obj).method();

Método no de Junit sino de hamcrest para aplicar métodos sobre un objeto.

Ejemplos complementarios sobre distintos servicios.

Tenemos un servicio y queremos testear un metodo que actualiza un registro, realiza sus validaciones e incluso lanza excepciones, como tratar esto, en el siguiente ejemplo vemos la implementación del servicio:

```

2      * Actualiza un cargo ignorando los nulos. los demás se colocaron como not null
3      * a nivel de dto. fecha hasta si puede ser nulo
4      */
5
6      @Override
7      @Transactional
8      public CategoriaTipoVinculacionPersona updateCategoriaTipoVinculacionPersona(
9          List<String> validateCategoriaTipoVinculacionPersona = new ArrayList<>()
10         validateCategoriaTipoVinculacionPersona(aCategoriaTipoVinculacionPersona,
11
12         CategoriaTipoVinculacionPersona categoriaTV = this.categoriaTipoVinculacionPersona;
13         ITools.copiarPropiedadesObjetoAHaciaBIgnorandoNulosDeA(aCategoriaTipoVinculacionPersona, categoriaTV);
14         categoriaTV.setFechaHasta(aCategoriaTipoVinculacionPersona.getFechaHasta());
15         aCategoriaTipoVinculacionPersona = categoriaTV;
16
17         if (!validateCategoriaTipoVinculacionPersona.isEmpty()) {
18             throw new RSIValidacionException(validateCategoriaTipoVinculacionPersona);
19         }
20         return this.categoriaTipoVinculacionPersonaRepository.save(aCategoriaTipoVinculacionPersona);
21     }
22 }

```

¿Cómo testearlo?

Lo primero es recalcar que esto no es metodología TDD puesto que estamos probando un método ya existente. Tenemos entonces dos puntos claves, el flujo normal y el flujo de excepción por validación, para hacerlo más legible podemos separarlo en dos metodos uno para el update exitoso y otro que active la validación.

Update Exitoso

```

1      @DisplayName("test for update a category... ")
2      @Test
3      void updateCategoriaTipoVinculacionPersonaTest() {
4
5          //given
6          given(categoriaTipoVinculacionPersonaRepository.save(categoriaTipoVinculacionPersona))
7              .willReturn(categoriaTipoVinculacionPersona);
8
9          given(categoriaTipoVinculacionPersonaRepository.findById(any(Long.class)))
10             .willReturn(Optional.of(categoriaTipoVinculacionPersona));
11
12         //when action
13         var categoryUpdated = categoriaTipoVinculacionPersonaService.
14             updateCategoriaTipoVinculacionPersona(categoriaTipoVinculacionPersona);

```

```

15
16     //then verifications
17     assertThat(categoryUpdated).isNotNull();
18 }

```

Como se aprecia en el código, tenemos la primera prueba donde definimos que comportamiento tendrá el repositorio (mock) en los distintos metodos requeridos por la lógica, lo siguiente es invocar el metodo del servicio y corroborar que se está actualizando y que el objeto devuelto existe, nada complicado y quedaría probado el primer caso, para el segundo caso que es el siguiente:

```

1  @DisplayName("test for update a category with exception... ")
2  @Test
3  void updateCategoriaTipoVinculacionPersonaTestCheckException() {
4
5      //given
6      var categorySimilar =(CategoriaTipoVinculacionPersona)ITools.copiarPropiedad
7      categorySimilar.setId(2L);
8
9      given(categoriaTipoVinculacionPersonaRepository.findById(any(Long.class)))
10         .willReturn(Optional.of(categoriaTipoVinculacionPersona));
11
12     given(categoriaTipoVinculacionPersonaRepository.findByIdPersonaAndIdCatTipo
13         .willReturn(Arrays.asList(categoriaTipoVinculacionPersona, category{
14
15     //then verifications
16     assertThrows(RSIValicacionException.class,() -> categoriaTipoVinculacionPer
17         .updateCategoriaTipoVinculacionPersona(categoriaTipoVinculacionPer
18 }

```

El análisis es un poco similar, definimos el comportamiento de cada método del repositorio necesario y en este caso la invocación del método se valida directamente con un **assertThrows**, este método recibe una excepción y un ejecutable desde el cual podemos invocar el servicio.

Ahora bien no por utilizar dicho método **assertThrows** significa que el servicio lance la excepción, para ello es requerido tener clara la lógica del servicio, en este caso lanza una excepción si al consultar el metodo **findByIdPersonaAndIdCatTipoVinculacion** devuelve una lista con dos objetos con mismos campos pero diferentes identificadores, esto lo simulamos en la fase "given" y por lo tanto la ejecución lanzará la excepción, y el **assertThrows** la verificará y el test sale exitoso.

Siempre es importante conocer y tener clara la lógica de lo que se quiere probar, esto facilita el desarrollo de test y el desarrollo en general.

Coverage - intellij

Una de las opciones que tiene IntelliJ sobre los test es el coverage, el coverage nos da estadísticas de que tanto del servicio se ha logrado testear, que métodos, que cantidad de líneas, etc.

Para activarlo podemos hacer uso de las opciones visuales que posee IntelliJ, ubicadas a los laterales de los archivos, en la guía de página.

```
@ExtendWith(MockitoExtension.class)
class CategoriaTipoVinculacionPersonaServiceTest {
```

```
//Methods
@Test
void givenIdWhenFindThenReturnObject() {
```

Run 'CategoriaTipoVinculacionPersonaServiceTest' with Coverage

Inmediatamente se abra una pestaña de coverage que nos da las estadísticas de la clase que estamos probando o de las clases que intervienen en el test, pero por orden suele ser de relación 1 a 1.

En caso de correr la ejecución de la clase tenemos algo como lo siguiente. En la parte derecha el coverage, a la izquierda el código y en la parte inferior el estado de los test y el tiempo de ejecución de los mismos.

The screenshot displays the IntelliJ IDEA interface during a test run. The main editor shows the `CategoriaTipoVinculacionPersonaServiceTest` class with annotations like `@ExtendWith(MockitoExtension.class)` and `@Mock`. The right sidebar shows the 'Coverage' tab for the same class, indicating 100% classes and 71% lines covered. The bottom panel shows 'Test Results' with a list of tests and their execution times.

Element	Class, %	Method, %	Line, %
co.edu.uis.training.service.rechumanol.impl...	100% (1/1)	100% (10/10)	71% (47/66)

Test Results:

- test for create a category... 438 ms
- test for find all categories... 9 ms
- test for deleting a category... 9 ms
- test for update a category with exception... 254 ms
- test for identification... 2 ms
- test for find by persona documento o categoria... 3 ms
- test for update a category... 2 ms
- test for get by id... 2 ms

Además IntelliJ nos ofrece un coverage línea a línea en la clase que queremos probar y nos indica que líneas han sido probadas (marcadas en verde) y que líneas están pendientes de prueba (marcadas en rojo) y luce de la siguiente manera.

Líneas probadas en verde

```

10  /**
11   * Actualiza un cargo ignorando los nulos. Los demás se colocaran como not null
12   * a nivel de dto. fecha hasta si puede ser nulo
13   */
14  @Override
15  @Transactional
16  public CategoriaTipoVinculacionPersona updateCategoriaTipoVinculacionPersona(CategoriaTipoVinculacionPersona aCategoriaTipoVinculacionPersona) {
17      List<String> validateCategoriaTipoVinculacionPersona = new ArrayList<>();
18      validateCategoriaTipoVinculacionPersona(aCategoriaTipoVinculacionPersona, IConstantes.MODO_EDICION, validateCategoriaTipoVinculacionPersona);
19
20      CategoriaTipoVinculacionPersona categoriaTV = this.categoriaTipoVinculacionPersonaRepository.findById(aCategoriaTipoVinculacionPersona.getId()).orElseThrow(RSIDataNotFoundException::new);
21      ITools.copiarPropiedadesObjetoAObjetoIgnorandoNulosDeA(aCategoriaTipoVinculacionPersona, categoriaTV);
22      categoriaTV.setFechaHasta(aCategoriaTipoVinculacionPersona.getFechaHasta());
23      aCategoriaTipoVinculacionPersona = categoriaTV;
24
25      if (!validateCategoriaTipoVinculacionPersona.isEmpty()) {
26          throw new RSIVValidationException(validateCategoriaTipoVinculacionPersona.stream().map(Object::toString).collect(Collectors.joining( " ")));
27      }
28
29      return this.categoriaTipoVinculacionPersonaRepository.save(aCategoriaTipoVinculacionPersona);
30  }

```

Líneas pendientes en rojo

```

162
163      if ((fechaHastaBD != null) && ((ITools.isFechaAMayorIgualQueFechaB(fechaDesdeIn, fechaDesdeBD, aTipoComparacion: "≥"))
164          && (ITools.isFechaAMenorIgualQueFechaB(fechaDesdeIn, fechaHastaBD, aTipoComparacion: "≤")))) {
165          validateCategoriaTipoVinculacionPersona.add(ITools.getMensaje( aEtiqueta: "EQUIVALENCIA_CATEGORIA_EXISTE"));
166          return true;
167      }
168      //-----
169
170      if ((fechaHastaBD != null)
171          && ((ITools.isFechaAMayorIgualQueFechaB(fechaHastaIn, fechaDesdeBD, aTipoComparacion: "≥"))
172          && (ITools.isFechaAMenorIgualQueFechaB(fechaDesdeIn, fechaHastaBD, aTipoComparacion: "≤")))) {
173          validateCategoriaTipoVinculacionPersona.add(ITools.getMensaje(EQUIVALENCIA_DETALLE_EXISTE));
174          return true;
175      }
176      //-----
177
178      if ((fechaHastaIn == null) && (ITools.isFechaAMenorIgualQueFechaB(fechaDesdeIn, fechaDesdeBD, aTipoComparacion: "≤")))) {
179          validateCategoriaTipoVinculacionPersona.add(ITools.getMensaje(EQUIVALENCIA_DETALLE_EXISTE));
180          return true;
181      }

```

TESTING USANDO SPRING (cargando contexto de aplicación y con conexión a base de datos)

Cuando testeamos aplicaciones con un contexto de spring suele cargar todo un contexto de aplicación y se realizan incluso conexiones a base de datos y demás, y cabe la posibilidad de insertar datos al realizar los test si no se maneja adecuadamente ciertas consideraciones.

1. Para pruebas con spring debemos agregar la anotación de clase `@SpringBootTest`.
2. Debemos agregar cada dependencia asociada al servicio que deseamos testear y anotarla con `@MockBean`.
3. para el servicio interesado en probarse se inyecta con `@Autowired`.
4. En caso de querer probar comportamiento real siempre debe añadirse la anotación `@Transactional` sobre el método de prueba y eliminar los `@MockBean` para que el comportamiento sea el real.

Primer caso - conexión con `@MockBean` para las dependencias y `@Autowired` para el servicio únicamente:

1 | `@SpringBootTest`

```

2  class AuditorServiceTest {
3
4      //@MockBean remplazo de @Mock
5      //Autowired remplazo de @InjectMocks pero en un contexto de spring boot es
6      //con un @RunWith bajo springclass o motor de spring
7      //y sobre un @SpringBootTest
8      @MockBean
9      private IClsaseSituacionAdministrativaRepository claseSituacionAdministrativaRepository;
10
11     private IAuditorService auditorService;
12
13     private ClaseSituacionAdministrativa claseSituacionAdministrativa;
14
15     @BeforeEach
16     public void setup() {
17         //claseSituacionAdministrativaRepository = Mockito.mock(ClsaseSituacionAdministrativaRepository.class);
18         // auditorService = new EmployeeServiceImpl();
19         // inyectamos por setter el repositorio
20         claseSituacionAdministrativa = new ClaseSituacionAdministrativa();
21         claseSituacionAdministrativa.setId(1L);
22         claseSituacionAdministrativa.setDescripcion("Clase de prueba danny");
23         claseSituacionAdministrativa.setSenalGrupal(true);
24         claseSituacionAdministrativa.setIdentificador("Identificador Test");
25     }
26
27     //Metodos de prueba
28     @DisplayName("Test for saveClaseSA")
29     @Test
30     void givenASObjectWhenSaveThenReturnASObject() {
31         //Given (precondition or setup)
32         given(claseSituacionAdministrativaRepository.save(any(ClsaseSituacionAdministrativa.class)))
33             .willReturn(claseSituacionAdministrativa);
34
35
36         //When (action or the behaviour that we are going test)
37         var savedClass = auditorService.saveClaseSA(claseSituacionAdministrativa);
38
39         //Then (verify the output and asserts)
40         assertThat(savedClass).isNotNull();
41         assertThat(savedClass.getDescripcion()).isEqualTo("Clase de prueba danny");
42
43     }
44 }

```

“

Incluso si olvidamos el given, no habrá posibilidad de insertar datos en la base de datos, no es necesario el `@Transactional` porque no estamos trabajando con un `@Autowired` sobre las dependencias sino que estamos utilizando un `@MockBean`, esto no genera persistencia real en ningún caso.

Un mock no persistirá datos, la clase no tiene implementación alguna y el comportamiento siempre será simulado.

Segundo caso - Testing con Autowired en lugar de @MockBean

Sí nosotros no realizamos el `@MockBean` el código será ejecutado sin problemas puesto que el servicio inyecta sus dependencias internamente con `@Autowired` y `spring boot` se levanta en su totalidad y se conecta a la base de datos, por tal razón siempre es necesario el `@MockBean`, si queremos probar comportamiento real entonces debemos añadir la anotación `@Transactional` al método de prueba para que al finalizar el test haga **rollback** sobre los datos modificados y todo quede en su estado original.

“

Dicho de otra manera **si le quitamos el `@MockBean` no tendremos posibilidad de definir el comportamiento, no podremos hacer un given ... will return ... etc.** si tenemos estas líneas al inicio como es sugerido spring nos avisará que hay un error porque no es un mock y no ejecutará nada más pero **si se nos olvida el given ... willreturn o cualquiera de sus variantes y ejecutamos el servicio, el servicio internamente carga sus dependencias, se conecta a la base de datos y modifica los datos, por lo tanto estaríamos cometiendo un error.**

Si dado el caso es lo que queremos entonces para asegurarnos de que no se almacena "basura" o datos de prueba añadimos el `@Transactional` y listo los datos no persisten mas allá del test. **El método mas sencillo para probar esto quedaría de la siguiente manera.**

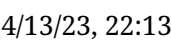
```

1 | @DisplayName("Test for deleteClaseSA")
2 |     @Test
3 |     @Transactional
4 |     void givenIdDeleteObject() {
5 |
6 |         auditorService.deleteClaseSA(287L);
7 |
8 |     }
```

sin el transactional ejecutaría el delete de manera permanente, pero con el `@Transactional` el **rollback** se produce al acabar la ejecución del test y solo queda probado el comportamiento en real.



En cuanto al coverage en intelliJ tenemos la opción coverage as y de la misma manera la ejecutamos, como resultado obtenemos una vista similar y unas estadísticas como se aprecia a continuación.



```

package co.edu.uis.training.service.auditor.impl;

import co.edu.uis.library.util.ITools;

/**
 * Hecho para pruebas de auditoria
 */
@Service
public class AuditorServiceImpl implements IAuditorService {

    private IClsaseSituacionAdministrativaRepository clsaseSituacionAdministrativaRepository;

    /**
     * Inserta un registro
     * @param clsaseSituacionAdministrativa
     */
    @Override
    @Transactional
    public ClaseSituacionAdministrativa saveClaseSA(ClaseSituacionAdministrativa clsaseSituacionAdministrativa) {

        var obj2 = (ClaseSituacionAdministrativa)
            ITools.copiarPropiedadesObjetoAObjeto(clsaseSituacionAdministrativa, new ClaseSituacionAdministrativa());
        this.clsaseSituacionAdministrativaRepository.save(clsaseSituacionAdministrativa);
        this.clsaseSituacionAdministrativaRepository.flush();
        clsaseSituacionAdministrativa.setIdentificador(clsaseSituacionAdministrativa.getIdentificador() + " diferente");
        return this.clsaseSituacionAdministrativaRepository.save(clsaseSituacionAdministrativa);
    }

    /**
     * Elimina un registro por id
     * @param id
     */
    @Override
    @Transactional
    public void deleteClaseSA(Long id) {
        this.clsaseSituacionAdministrativaRepository.deleteBy(id);
    }

    @Override
    @Transactional
    public boolean metodoRandom(ClaseSituacionAdministrativa clsaseSituacionAdministrativa){
        var obj2 = (ClaseSituacionAdministrativa)
            ITools.copiarPropiedadesObjetoAObjeto(clsaseSituacionAdministrativa, new ClaseSituacionAdministrativa());
        this.clsaseSituacionAdministrativaRepository.save(clsaseSituacionAdministrativa);
    }
}

```

y en caso de falla tenemos:

```

List<String> validateCategoriaTipoVinculacionPersona,
CategoriaTipoVinculacionPersona descriptoresSimilar) {

    Date fechaHastaBD = descriptoresSimilar.getFechaHasta();
    Date fechaDesdeBD = descriptoresSimilar.getFechaDesde();

    Date fechaHastaIn = aCategoriaTipoVinculacionPersona.getFechaHasta();
    Date fechaDesdeIn = aCategoriaTipoVinculacionPersona.getFechaDesde();

    Long documentoSoporteBD = descriptoresSimilar.getIdDocSoporte();
    Long documentoSoporteIn = aCategoriaTipoVinculacionPersona.getIdDocSoporte();

    if (documentoSoporteBD.equals(documentoSoporteIn)) {
        validateCategoriaTipoVinculacionPersona.add(ITools.getMensaje("DOCUMENTOS_SIMILARES"));
        return true;
    }

    if ((fechaHastaBD != null) && ((ITools.isFechaAMayorIgualQueFecha(fechaDesdeIn, fechaDesdeBD, ">=")
        && (ITools.isFechaAMenorIgualQueFecha(fechaDesdeIn, fechaHastaBD, "<=")))) {
        validateCategoriaTipoVinculacionPersona.add(ITools.getMensaje("EQUIVALENCIA_CATEGORIA_EXISTE"));
        return true;
    }

    if ((fechaHastaBD != null)
        && ((ITools.isFechaAMayorIgualQueFecha(fechaHastaIn, fechaDesdeBD, ">=")
        && (ITools.isFechaAMenorIgualQueFecha(fechaDesdeIn, fechaHastaBD, "<=")))) {
        validateCategoriaTipoVinculacionPersona.add(ITools.getMensaje("EQUIVALENCIA_DETALLE_EXISTE"));
        return true;
    }

    if ((fechaHastaIn == null) && (ITools.isFechaAMayorIgualQueFecha(fechaDesdeIn, fechaDesdeBD, ">="))) {
        validateCategoriaTipoVinculacionPersona.add(ITools.getMensaje("EQUIVALENCIA_DETALLE_EXISTE"));
        return true;
    }

    if ((fechaHastaBD == null) && (ITools.isFechaAMayorIgualQueFecha(fechaDesdeIn, fechaDesdeBD, ">=") || ITools.isFechaAMayorIgualQueFecha(fechaHastaIn, fechaDesdeBD, ">="))) {
        validateCategoriaTipoVinculacionPersona.add(ITools.getMensaje("EQUIVALENCIA_DETALLE_EXISTE"));
        return true;
    }

    if ((fechaHastaBD == null && fechaHastaIn == null) {
        validateCategoriaTipoVinculacionPersona.add(ITools.getMensaje("EQUIVALENCIA_DETALLE_EXISTE"));
        return false;
    }
}

```

Consideraciones y recomendaciones



- **No realizar clases estáticas** pues son más difíciles de probar (suplantar) ni métodos

estáticos en lo posible.

- ▶ **Tener cuidado** con el uso de test basados en spring pues levantan un contexto de aplicación, se conectan a base de datos y pueden llegar a alterar la base de datos, tener claro cuando usar `@MockBean` y cuando usar `@Transactional` para dichos casos.
- ▶ **En cuanto al coverage**, la manera en la que se testean las "líneas testeadas" es por medio de la ejecución del código, línea que se ejecute es una línea que ha sido probada por lo tanto para probar los diferentes casos de uso, ifs, excepciones y demás será necesario construir los objetos que generan dichos flujos y así controlar siempre la ejecución y el coverage de los métodos.
- ▶ **Las clases de test deben finalizar con la terminación Test**, esto para que el plugin de maven las pueda reconocer y sean ejecutados en la fase de test. (pueden añadirse patrones pero mejor usar los estándar).
- ▶ **Estar pendientes del coverage** y testear **todos** los métodos públicos, mínimo 1 método de test para cada método público del servicio.
- ▶ Los métodos privados son más difíciles de testear y no son testeables directamente sino indirectamente, en caso de complejidad se sugiere o bien utilizar un modificador `protected` para ser testeados directamente en el paquete equivalente de test o bien realizar pruebas con los distintos casos de prueba que los invocan.
- ▶ **De no ser necesario el uso de spring para los test lo mejor es no utilizarlo**, esto evitará que se cargue todo spring y su contexto de aplicación para test y demore la ejecución de las pruebas, por eficiencia se sugiere utilizar Mockito, `@mock` y `@InjectMocks`.
- ▶ En el tema de importaciones, siempre utilizar las importaciones de Junit, en el caso de asserts utilizar la importación estática de los métodos tanto de Junit como AssertJ o Hamcrest, esto facilita la legibilidad del código.
- ▶ **La notación de los métodos debe ser clara**, debe ser sencilla de entender y deben nombrarse de la forma `given...When...Then...` para explicar que se está probando o bien nombrando el método igual al servicio pero añadiendo la terminación Test.
- ▶ **El cuerpo de los métodos debe tener la misma estructura given, when, then** para mantener un orden, y si bien se pueden realizar pruebas de varios casos en un mismo método por orden y legibilidad se recomienda separar los métodos para cada caso realmente distinto.
- ▶ Para casos similares y pruebas de diferentes asserts, un mock puede ser configurado para que devuelva más de un valor, de modo que la primera invocación devolverá `val1`, la siguiente `val2` y podemos facilitar el desarrollo de los test y todo en un único método `@test`, **(en los códigos de este documento y en el código de gitlab estarán ejemplos)**.
- ▶ Los métodos `@test` pueden además tener la anotación `@Disable` que indica que el método no está listo para ser probado y de esta manera no nos limita la ejecución del proyecto y mejora la legibilidad al no estar comentando código y evita olvidarse algún método.
- ▶ Los métodos `@test` pueden además tener la anotación `@DisplayName("name here")` que facilita la identificación en consola cuando se ejecutan las pruebas.
- ▶ Para el uso de logs además de **Logger** también es válido el uso del **`System.out.println()`**.



► Enlace gitlab: https://gitlab.uis.edu.co/rsi/training/Back-training/-/tree/feature/68_predev_unit_test

Conclusiones

Los test son algo que no debe ser pasado por alto un proceso de desarrollo, aunque a priori se piensa que solo retrasará el tiempo de desarrollo la realidad es que a futuro ese tiempo es recompensado pues el test únicamente se crea una vez y está disponible siempre y controla que ante nuevas implementaciones, modificaciones y demás la lógica persiste y sigue funcionando, o falla y es sencillo encontrar el error para ser solucionado.

Aunque la metodología TDD no está muy clara y es algo que apenas iniciaremos, la literatura nos dice que es una manera correcta y eficiente de plantear el desarrollo de una historia y nos obliga a tener conceptos y acciones bien definidas, esto es la base de cualquier desarrollo consistente y de calidad.

Aunque aún queda mucho por aprender, lectura e investigación sobre maneras de abordar situaciones específicas, capas, niveles de test (integración, e2e...) los test unitarios son claves, y al ser sobre la capa de servicio, que es la capa que maneja la lógica del negocio, hace que esté más que justificado dar los primeros pasos acá, de cara al futuro será necesario aprender los diferentes tipos de test y mejorar aún más nuestra capacidad como desarrolladores y la calidad de nuestro código, **por ahora**, es todo.

Video: Charla sobre test unitarios

<https://drive.google.com/drive/u/4/folders/1ETdwWqgh8xtkuQBAD8VPM4BQ8mJ-5lZ4>