Ejercicio 17 – Pruebas en Spring. Parte 1: Servicio

Spring Boot nos proporciona una serie de utilidades y anotaciones para facilitar poder hacer pruebas tanto unitarias como integradas en nuestras aplicaciones. En esta práctica, vamos a realizar pruebas a diversos niveles en nuestra aplicación.

Pruebas automatizadas

Las pruebas automatizadas permiten validar que el comportamiento de la aplicación es el esperado. Permiten verificar que ante cambios no se introducen errores, generan mayor confianza en que la aplicación funciona como se espera y evitan la necesidad de realizar pruebas manuales lentas y repetitivas.

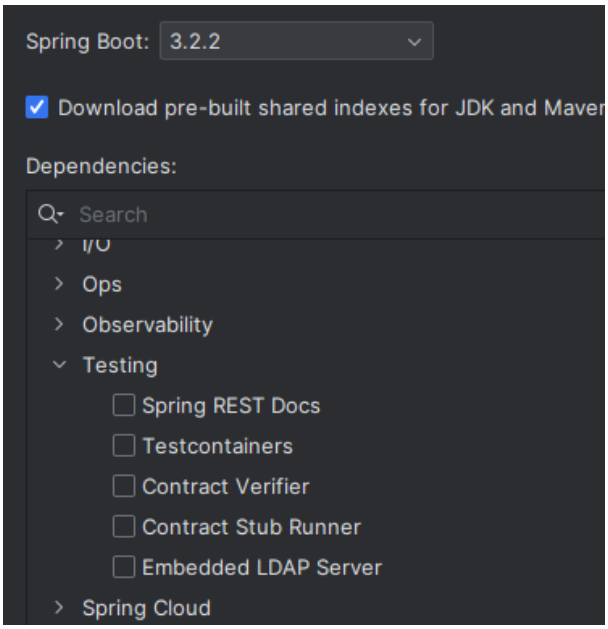
Las pruebas más numerosas son las unitarias, que prueban un componente de la aplicación de forma aislada de sus dependencias sin invocar operaciones de red. En caso de que el componente bajo prueba (*System Under Test, SUT*) tenga dependencias, el comportamiento de estas podría sustituirse por ***mocks***.

El objetivo de sustituir una dependencia real por un *mock* es programar el comportamiento de la dependencia *mockeada* y ejecutar la prueba de forma aislada y rápida. Una dependencia real es una base de datos como PostgreSQL o MongoDB, o un servicio que requiere comunicación por red como GraphQL.

Sin embargo, el código a ejecutar en la aplicación finalmente hace uso de las dependencias reales, y estas no se prueban en las pruebas unitarias. Por tanto, podríamos estar cometiendo errores, ya que, en algunos casos, las dependencias reales quizá se comporten de forma diferente a los mocks. Las pruebas de integración permiten probar el funcionamiento de dos componentes relacionados.

Herramientas de Spring para pruebas

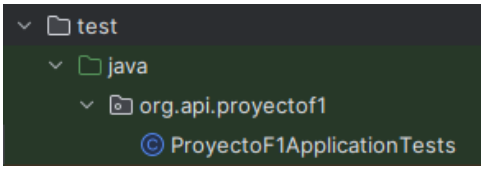
Cuando creamos un proyecto Spring, podemos ver entre las dependencias un apartado de ***Testing***, con añadidos que, según el alcance del proyecto, podrían interesarnos.



Sin embargo, la mayoría de desarrolladores utilizan las herramientas que incorpora el propio Spring en su ***spring-boot-starter-test***, entre las que destacamos ***Junit 5***, ***Spring Test***, ***Hamcrest*** o ***Mockito***.

Pruebas en la aplicación Spring

Si vemos la estructura de directorios de un proyecto, hay un directorio test generado automáticamente. La clase ***ProyectoF1ApplicationTests*** es la clase creada por defecto, pero que no utilizaremos.



Convenciones en el nombre de los métodos de prueba

Antes de empezar con las pruebas, vamos a hablar sobre la convención de nombres de métodos a utilizar.

El nombre de las pruebas es importante para los equipos de desarrollo de proyectos, igual que lo es cualquier otra convención de estilo de codificación. Al aplicar una convención de código en las pruebas, los nombres de los métodos de las pruebas serán legibles, comprensibles y tendrán un patrón de nomenclatura bien conocido para todo el equipo.

Hay muchos ejemplos de convenciones para nombrar las pruebas, simplemente sé consistente con uno.

Pruebas unitarias en la capa de servicio

En los casos en los que necesitamos realizar pruebas unitarias de las diferentes capas de nuestra aplicación, es importante aislar estas dependencias mediante el uso de la librería ***Mockito***.

Mockito es un framework de mocking que nos va a resultar muy útil para aislar la lógica de negocio de la capa de servicio de los detalles de acceso a datos de la capa de repositorio, y así poder probarla de forma independiente y sin necesidad de usar una base de datos real.

Para escribir una prueba unitaria de la capa de servicio con Mockito, tienes que seguir estos pasos:

- Crear una clase de prueba con la anotación ***@ExtendWith(MockitoExtension.class)***, que le indica a JUnit que use la extensión de Mockito para procesar las anotaciones de Mockito.
- Anotar la clase de servicio que quieres probar con ***@InjectMocks***, que le dice a Mockito que cree una instancia de esa clase e inyecte las dependencias simuladas que se hayan definido con *@Mock*.
- Anotar las clases de repositorio que sean dependencias de la clase de servicio con ***@Mock***, que le dice a Mockito que cree objetos simulados de esas clases, que se usarán en lugar de las reales.
- Escribir los métodos de prueba con la anotación ***@Test***, que le dice a JUnit que son casos de prueba que se deben ejecutar.
- Dentro de cada método de prueba, usar el método ***when*** de Mockito para especificar el comportamiento de los objetos simulados, es decir, qué deben devolver cuando se les llama con ciertos argumentos.
- Dentro de cada método de prueba, usar los métodos ***assert*** de AssertJ para verificar que el resultado de la clase de servicio es el correcto, comparándolo con el valor esperado.
- Dentro de cada método de prueba, usar el método ***verify*** de Mockito para comprobar que los objetos simulados se han usado de la forma esperada, es decir, qué métodos se han llamado y con qué argumentos.

Además, estructuraremos las pruebas con la estrategia ***BDD*** (Behaviour Driven Development), centrada en el comportamiento de una aplicación para el usuario final. BDD promueve la escritura de pruebas bajo el patrón ***Given-When-Then***, que se define como:

- ***Given*** (Escenario): Se especifica el escenario, las precondiciones.
- ***When*** (Acción): Las condiciones de las acciones que se van a ejecutar.
- ***Then*** (Resultado): El resultado esperado, las validaciones a realiza

A continuación, veremos un ejemplo de la inyección de dependencias y la forma correcta de hacer pruebas unitarias sobre un método

```java
@ExtendWith(MockitoExtension.class)
class DriverServiceImplTest {

    // Dependencias
    2 usages
    @Mock
    DriverRepository driverRepository;

    // SUT
    1 usage
    @InjectMocks
    DriverServiceImpl driverService;

    // Datos de prueba
    4 usages
    Driver driver;
    1 usage
    DriverDTO driverDTO;

    alejandroroig
    @BeforeEach
    public void setup() {
        driver = Driver.builder().id(1L).code("AAA").forename("Ayrton").surname("Senna").build();
        driverDTO = DriverDTO.builder().id(1L).code("AAA").forename("Ayrton").surname("Senna").build();
    }

    alejandroroig
    @Test
    void shouldReturnDriverDTOWhenCreateDriver() {
        // Given. Configurar el comportamiento del mock
        when(driverRepository.save(any(Driver.class))).thenReturn(driver);

        // When. Ejecutar el método a probar
        DriverDTO savedDriver = driverService.saveDriver(driver);

        // Then. Verificar el resultado
        assertNotNull(savedDriver);
        assertEquals( expected: "AAA", savedDriver.code());
        // Verificar que el método del mock fue invocado
        verify(driverRepository, times( wantedNumberOfInvocations: 1)).save(driver);
    }
}
```

Como podemos ver, el código prueba el funcionamiento del método ***saveDriver()*** para llama al método *save()* del repositorio para guardar pilotos, el cual está mockeado. Fíjate que definimos datos de prueba del tipo de dato *Driver* y *DriverDTO*, que son los tipos involucrados en este servicio.

Ahora eres tú quien debe probar el resto de métodos del servicio y que utilizamos en la API:

- Método a probar: *getDriverByCode()*
    - Nombre de la prueba: ***shouldReturnDriverDTOWhenFindDriverByCode()***
- Método a probar: *updateDriver()*
    - Nombre de la prueba: ***shouldReturnDriverDTOWhenUpdateDriver()***
- Método a probar: *deleteDriverByCode()*
    - Nombre de la prueba: ***shouldReturnNothingWhenDeleteDriverByCode()***
- Método a probar: *getDrivers()*
    - Nombre de la prueba: ***shouldReturnDriverResponseWhenGetAllDrivers()***

Realiza pruebas también para casos donde tenemos que detectar funcionamiento incorrecto, por ejemplo, al actualizar un piloto buscamos por el código previo a la actualización y si dicho código no existe debe darnos una excepción de tipo *DriverNotFoundException*.

Además, realiza pruebas sobre el servicio ***ConstructorService*** para tener ambas interfaces cubiertas.