Ejercicio 18 – Pruebas en Spring. Parte 2: Repositorio

En la práctica anterior hicimos pruebas unitarias sobre la capa de Servicio de nuestra aplicación creada con Spring Boot. En esta práctica, vamos a realizar pruebas de integración en la capa de Repositorio.

Pruebas de integración

Las pruebas de integración son una parte crucial del proceso de desarrollo de software, ya que garantizan que los diferentes componentes de una aplicación funcionen perfectamente juntos.

Sin embargo, configurar y mantener un entorno de pruebas coherente y aislado puede resultar complicado, especialmente cuando se trata de dependencias externas como bases de datos, colas de mensajes u otros servicios.

El caso más habitual, y el que vamos a trabajar en esta práctica, son las bases de datos. Y, por si se te ha ocurrido la idea, aunque son muy útiles en pruebas unitarias, no perdamos de vista que, en general, hacer mocks en pruebas de integración desvirtúa el propósito de este tipo de pruebas.

Evolución de las pruebas de integración

A lo largo del tiempo, las técnicas para realizar estas pruebas han evolucionado, buscando un equilibrio entre la precisión y la eficiencia.

1. Creando una base de datos como la de producción

El método tradicional implicaba crear una copia exacta de la base de datos de producción para las pruebas.

Aunque este enfoque es preciso al usar el mismo software de bases de datos que en producción, también presenta varios inconvenientes, ya que puede resultar costoso y complejo de mantener, especialmente en entornos donde la estructura de la base de datos es cambiante. Además, esta solución hace complicado ejecutar las pruebas de forma automática y llevarlas a un entorno moderno de integración continua.

2. Uso de bases de datos en memoria

Las bases de datos en memoria, como H2 o SQLite, ofrecen una alternativa más rápida y ligera. Se ejecutan en la memoria del sistema, lo que elimina la necesidad de una instalación y configuración compleja.

Sin embargo, tener entornos de prueba y de producción tan dispares es problemático, ya que no todas las características de las bases de datos tradicionales están disponibles en sus versiones en memoria.

3. Testcontainers

Testcontainers surge como una solución moderna que combina la precisión de las bases de datos reales con la eficiencia de las bases en memoria.

Esta herramienta permite ejecutar contenedores Docker que encapsulan una base de datos real, como MySQL o PostgreSQL, durante las pruebas, garantizando una mayor similitud entre el entorno de prueba y el entorno de producción.

Además, Testcontainers ofrece una gran flexibilidad y escalabilidad, permitiendo ejecutar pruebas utilizando múltiples contenedores, lo que lo convierte en una opción atractiva para equipos de desarrollo ágiles.

Dependencias en Spring

Como puedes comprobar en el proyecto compartido, hemos añadido una dependencia a la ***base de datos H2***. El motivo de esta adición lo veremos en esta primera parte de la práctica.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

También hemos añadido las dependencias necesarias para utilizar ***Testcontainers***, sobre los que trabajaremos en la segunda y última parte de la práctica.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-testcontainers</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <scope>test</scope>
</dependency>
```

Probando el repositorio con una base de datos en memoria

Para realizar pruebas en los componentes de la capa *Repository* usamos la anotación ***@DataJpaTest***. Esta anotación escanea aquellas clases anotadas con *@Entity* y los repositorios, aplicando solo la configuración relevante para las pruebas JPA.

Con la anotación ***@AutoConfigureTestDatabase*** configuramos una base de datos de prueba en lugar de la base de datos de la aplicación. El atributo ***connection*** especifica el tipo de conexión a establecer, en este caso, ***EmbeddedDatabaseConnection.H2***, que indica que se usará una base de datos H2 embebida.

Además, con ***@Autowired*** inyectamos el repositorio para acceder a sus métodos desde la prueba.

```
alejandroroig
@DataJpaTest
@AutoConfigureTestDatabase(connection = EmbeddedDatabaseConnection.H2)
class DriverRepositoryTest {

    @Autowired DriverRepository driverRepository;

    // Datos de prueba
    6 usages
    Driver driver;
```

Ejecutemos el primer test:

```
@Test
void shouldReturnSavedDriverWhenSave() {
    // Given
    // When
    Driver driverSaved = driverRepository.save(driver);

    // Then
    assertThat(driverSaved).isNotNull();
    assertThat(driverSaved.getId()).isGreaterThan( other: 0);
}
```

Como podemos ver, el código prueba el funcionamiento del método ***save()*** para guardar pilotos. Ahora eres tú quien debe probar el resto de métodos que nos proporciona JpaRepository y que utilizamos en la API:

- Método a probar: *findAll()*
    - Nombre de la prueba: ***shouldReturnMoreThanOneDriverWhenSaveTwoDrivers()***
- Método a probar: *findByCodeIgnoreCase()*
    - Nombre de la prueba: ***shouldReturnDriverNotNullWhenFindByCode()***
- Método a probar: *save() (Actualización)*
    - Nombre de la prueba: ***shouldReturnDriverNotNullWhenUpdateDriver()***
- Método a probar: *deleteByCode()*
    - Nombre de la prueba: ***shouldReturnNullDriverWhenDelete()***

Probando el repositorio con TestContainers

Siendo la 3.2.2 la versión actual de Spring Boot, la versión 3.1, lanzada en mayo de 2023, trajo considerables mejoras para dar soporte a TestContainers, por lo que la mayoría de la documentación que encontraremos en Internet estará desactualizada.

En concreto, con Spring Boot 3.1, podemos utilizar la anotación ***@ServiceConnection*** para configurar automáticamente las propiedades de conexión para los contenedores, sin la necesidad de la obsoleta @DynamicPropertySource o la configuración manual de propiedades. Esta característica funciona para muchos tipos de contenedores admitidos por TestContainers, como PostgreSQL, MongoDB, Redis o Kafka.

Para que TestContainers funcione, necesitamos ejecutar Docker localmente o un servicio en la nube como TestContainers Cloud. Dicho esto, vamos a ver cómo configurar el entorno de pruebas para el repositorio.

```
@Testcontainers
@DataJpaTest
class ConstructorRepositoryTest {

    @Container
    @ServiceConnection
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>( dockerImageName: "postgres:15.5");

    @Autowired ConstructorRepository constructorRepository;
```

Vamos a analizar las anotaciones:

- ***@Testcontainers***: Esta anotación se utiliza en pruebas de integración para gestionar contenedores Docker.
- ***@Container***: Esta anotación se utiliza para definir un contenedor Docker. En este caso, se crea un contenedor PostgreSQL con la versión 15.5. La variable postgres se inicializa con este contenedor.

El cuerpo de los métodos de prueba no cambiará respecto a los métodos anteriores, por lo que puedes adaptar los métodos realizados para el repositorio de Drivers a los de Constructors.