Testeo de aplicaciones desarrolladas con Spring (segunda parte)

1. Introducción al uso de Mockito

Hasta ahora, solo hemos revisado las pruebas de métodos simples que no dependen de dependencias externas, pero esto está lejos de ser normal para aplicaciones grandes. Por ejemplo, un servicio comercial probablemente se basa en una base de datos o en una llamada de servicio web para recuperar los datos con los que opera. Entonces, ¿cómo probaríamos un método en tal clase? ¿Y cómo simularíamos condiciones problemáticas, como un error de conexión a la base de datos o un tiempo de espera?

La estrategia de los objetos simulados es analizar la clase bajo prueba y crear versiones simuladas de todas sus dependencias, creando los escenarios que queremos probar. Puede hacer esto manualmente, lo cual es mucho trabajo, o puede aprovechar una herramienta como Mockito, que simplifica la creación e inyección de objetos simulados en sus clases. Mockito proporciona una API simple para crear implementaciones simuladas de sus clases dependientes, inyectar las simulaciones en sus clases y controlar el comportamiento de las simulaciones.

En el siguiento código encontramos un repositorio muy simple:

```
package com.javaworld.geekcap.mockito;

import java.sql.SQLException;
import java.util.Arrays;
import java.util.List;

public class Repository {
    public List<String> getStuff() throws SQLException {
        // Execute Query

        // Return results
        return Arrays.asList("One", "Two", "Three");
    }
}
```

Y en este otro código, encontramos un servicio que usa el repositorio anterior.

```
package com.javaworld.geekcap.mockito;

import java.sql.SQLException;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Service {
    private Repository repository;

    public Service(Repository repository) {
        this.repository = repository;
    }

    public List<String> getStuffWithLengthLessThanFive() {
        try {
            return repository.getStuff().stream()
                    .filter(stuff -> stuff.length() < 5)
                    .collect(Collectors.toList());
        } catch (SQLException e) {
            return Arrays.asList();
        }
    }
}
```

El código del repositorio tiene un único método, `getStuff()`, que presumiblemente se conectaría a la base de datos, ejecutaría una consulta y devolvería los resultados. En este ejemplo, símplemente devuelve una lista de varios `Strings`. El servicio del código anterior recibe el repositorio a través del constructor y define un solo método, `getStuffWithLengthLessThanFive`, que devuelve todos los `Strings` con longitud menor que 5. Si el repositorio lanza una excepción `SQLException`, entonces devuelve una lista vacía.

1.1 Prueba unitaria con JUnit5 y Mockito

Veamos ahora cómo sería nuestra prueba con JUnit5 y Mockito:

```
package com.javaworld.geekcap.mockito;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;

import java.sql.SQLException;
import java.util.Arrays;
import java.util.List;

@ExtendWith(MockitoExtension.class)
class ServiceTest {
    @Mock
    Repository repository;

    @InjectMocks
    Service service;

    @Test
    void testSuccess() {
        // Setup mock scenario
        try {
            Mockito.when(repository.getStuff()).thenReturn(Arrays.asList("A", "B",
"CDEFGHIJK", "12345", "1234"));
        } catch (SQLException e) {
            e.printStackTrace();
        }

        // Execute the service that uses the mocked repository
        List<String> stuff = service.getStuffWithLengthLessThanFive();

        // Validate the response
        Assertions.assertNotNull(stuff);
        Assertions.assertEquals(3, stuff.size());
    }

    @Test
    void testException() {
        // Setup mock scenario
        try {
            Mockito.when(repository.getStuff()).thenThrow(new
SQLException("Connection Exception"));
        } catch (SQLException e) {
            e.printStackTrace();
        }
```

```
        // Execute the service that uses the mocked repository
        List<String> stuff = service.getStuffWithLengthLessThanFive();

        // Validate the response
        Assertions.assertNotNull(stuff);
        Assertions.assertEquals(0, stuff.size());
    }
}
```

Lo primero que debe notar acerca de esta clase de prueba es que está anotado con `@ExtendWith(MockitoExtension.class)`. La anotación `@ExtendWith` se usa para cargar una extensión JUnit 5. JUnit define una API de extensión, que permite a un proveedor externo como Mockito conectarse al ciclo de vida de las clases de prueba en ejecución y agregar funcionalidad adicional. El `MockitoExtension` examina la clase de prueba, encuentra variables miembro anotadas con la anotación `@Mock` y crea una implementación simulada de esas variables. Luego encuentra las variables miembro anotadas con la anotación `@InjectMocks` e intenta inyectar sus simulaciones en esas clases, usando inyección de construcción o inyección de setter.

En este ejemplo, `MockitoExtension` encuentra la anotación `@Mock` en la variable miembro `Repository`, por lo que crea una implementación simulada de la misma y la asigna a dicha variable. Cuando ve la anotación `@InjectMocks` en la variable miembro `Service`, crea una instancia de la clase `Service`, pasando el simulacro de `Repository` a su constructor. Esto nos permite controlar el comportamiento de la clase simulada `Repository` usando las API de Mockito.

En el método `testSuccess`, usamos la API de Mockito para devolver un conjunto de resultados específico cuando se llama al método `getStuff`. La API funciona de la siguiente manera:

- `Mockito::when` define la condición, que en este caso es la invocación del método `repository.getStuff()`.
- El método `when()` devuelve una instancia de `org.mockito.stubbing.OngoingStubbing`, que define un conjunto de métodos que determinan qué hacer cuando se llama al método especificado.
- En este caso, invocamos el método `thenReturn()` para decirle al *stub* que devuelva un listado específico de cadenas de caracteres.

En este punto, tenemos una instancia de `Service` con un instancia *simulada* de `Repository`. Cuando el método `getStuff` del repositorio se invoca, devuelve una lista de cinco cadenas conocidas. Invocamos el método `getStuffWithLengthLessThanFive()` del servicio, que invocará el método `getStuff()` del repositorio, y devolverá una lista filtrada de `Strings` cuya longitud es menor que cinco. Entonces podemos afirmar que la lista devuelta no es nula y que su tamaño es tres. Este proceso nos permite probar la lógica en el método específico del servicio, con una respuesta conocida del `Repository`.

El método `testException` configura Mockito para que cuando se llame al método `getStuff()`, arroje un `SQLException`. Si esto sucede, `Service` no debería lanzar una excepción; más bien, debería devolver una lista vacía.

Mockito es una herramienta poderosa y solo hemos arañado la superficie de lo que puede hacer. Si alguna vez te has preguntado cómo puedes probar condiciones complicadas, como la red, la base de datos, el tiempo de espera u otras condiciones de error de E/S, Mockito es la herramienta para ti y funciona de manera muy elegante con JUnit 5. Si te encuentras con situaciones que Mockito no admite, como burlarse de

variables miembro estáticas o constructores privados, entonces existe otra herramienta poderosa pero compleja llamada PowerMock .

2. Pruebas unitarias con Spring Boot, JUnit 5 y Mockito

Spring Boot nos provee de todo lo necesario para poder realizar el testeo de nuestras pruebas. No en vano, cuando generamos el esquelto de un proyecto con https://start.spring.io, lo importamos con nuestro IDE, encontramos la ruta `src/main/test`, preparada para escribir nuestros test de JUnit5.

Además, se añade por defecto la siguiente dependencia:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

Esta dependencia, para la versión actual de Spring Boot, incluye JUnit 5 y Mockito, entre otras herramientas.

2.1 No user Spring en los test unitarios

Al escribir test para una aplicación de Spring o Spring Boot, tenemos que tener en cuenta que no debemos utilizar Spring en los test. ¿A qué se debe eso?

Considera el siguiente código de ejemplo, en el que vamos a realizar la prueba de "unidad" a un método de la clase `RegisterUseCase`:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
class RegisterUseCaseTest {

  @Autowired
  private RegisterUseCase registerUseCase;

  @Test
  void savedUserHasRegistrationDate() {
    User user = new User("zaphod", "zaphod@mail.com");
    User savedUser = registerUseCase.registerUser(user);
    assertThat(savedUser.getRegistrationDate()).isNotNull();
  }

}
```

Esta prueba puede tomar unos 5 segundos en ejecutarse en un proyecto Spring vacío en mi portátil.

Sin embargo, **una buena prueba unitaria debería tomar solamente milisegundos**. De no ser así, obstaculiza el flujo "prueba/código/prueba" promovida por el TDD.

Así que hemos iniciado toda la aplicación solo para conectar automáticamente una `RegisterUseCase` instancia a nuestra prueba . Tomará aún más tiempo una vez que la aplicación crezca y Spring tenga que cargar más y más beans en el contexto de la aplicación.

2.1.1 Cómo crear un bean de Spring que sea testeable

Hay determinadas cosas que podemos hacer para que nuestros beans sean más prácticos de cara que que podamos probarlos.

La inyección con `@Autowired` sobre un atributo es mala

Comencemos con un mal ejemplo. Considere la siguiente clase:

```
@Service
public class RegisterUseCase {

  @Autowired
  private UserRepository userRepository;

  public User registerUser(User user) {
    return userRepository.save(user);
  }

}
```

Esta clase no se puede probar unitariamente sin Spring porque no proporciona ninguna forma de pasar una instancia de `UserRepository`. En su lugar, debemos escribir la prueba de la forma que se discutió en la sección anterior para permitir que Spring cree una instancia de `UserRepository` e inyecte en el campo anotado con `@Autowired`.

La lección aquí es no usar la inyección de campo.

Proporcionar un constructor

En realidad, no usemos la anotación `@Autowired` en absoluto:

```
@Service
public class RegisterUseCase {

  private final UserRepository userRepository;

  public RegisterUseCase(UserRepository userRepository) {
    this.userRepository = userRepository;
  }

  public User registerUser(User user) {
    return userRepository.save(user);
  }
```

```
}
```

Esta versión permite la inyección de constructores al proporcionar un constructor que permite pasar una instancia de `UserRepository`. En la prueba unitaria, ahora podemos crear una instancia de este tipo (quizás una instancia simulada como veremos más adelante) y pasarla al constructor.

Spring utilizará automáticamente este constructor para crear una instancia de un RegisterUseCase al crear el contexto de la aplicación de producción. Tenga en cuenta que antes de Spring 5, necesitamos agregar la anotación `@Autowired` al constructor, pero desde la versión 5 esto ya no es necesario.

También tenga en cuenta que el campo `UserRepository` es ahora final. Esto tiene sentido, ya que el contenido del campo nunca cambiará durante la vida útil de una aplicación. También ayuda a evitar errores de programación, porque el compilador se quejará si nos hemos olvidado de inicializar el campo.

Reducir el código repetitivo

Usando la anotación `@RequiredArgsConstructor` de Lombok podemos dejar que el constructor se genere automáticamente:

```
@Service
@RequiredArgsConstructor
public class RegisterUseCase {

  private final UserRepository userRepository;

  public User registerUser(User user) {
    user.setRegistrationDate(LocalDateTime.now());
    return userRepository.save(user);
  }

}
```

Ahora, tenemos una clase muy concisa sin código repetitivo que se puede instanciar fácilmente en un caso de prueba simple de Java:

```
class RegisterUseCaseTest {

  private UserRepository userRepository = ...;

  private RegisterUseCase registerUseCase;

  @BeforeEach
  void initUseCase() {
    registerUseCase = new RegisterUseCase(userRepository);
  }

  @Test
```

```
  void savedUserHasRegistrationDate() {
    User user = new User("zaphod", "zaphod@mail.com");
    User savedUser = registerUseCase.registerUser(user);
    assertThat(savedUser.getRegistrationDate()).isNotNull();
  }

}
```

Todavía falta una pieza, y así es como simular la instancia de `UserRepository` de la que depende nuestra clase bajo prueba, porque no queremos depender de la cosa real, que probablemente necesita una conexión a una base de datos.

2.1.2 Uso de Mockito para simular dependencias

La biblioteca de *mocking* estándar de facto hoy en día es Mockito. Proporciona al menos dos formas de crear un simulacro de `UserRepository` para llenar el espacio en blanco en el ejemplo de código anterior.

Programáticamente

La primera forma es usar Mockito mediante programación:

```
private UserRepository userRepository = Mockito.mock(UserRepository.class);
```

Esto creará un objeto que se verá como un `UserRepository` desde fuera. De forma predeterminada, no hará nada cuando se llame a un método y regresará null si el método tiene un valor de retorno .

Nuestra prueba ahora fallaría con un `NullPointerException` en `assertThat(savedUser.getRegistrationDate()).isNotNull()` porque `userRepository.save(user)` ahora regresa `null`.

Entonces, tenemos que decirle a Mockito que devuelva algo cuando se llame a `userRepository.save()`. Hacemos esto con el metodo `when` estático:

```
@Test
void savedUserHasRegistrationDate() {
  User user = new User("zaphod", "zaphod@mail.com");
  when(userRepository.save(any(User.class))).then(returnsFirstArg());
  User savedUser = registerUseCase.registerUser(user);
  assertThat(savedUser.getRegistrationDate()).isNotNull();
}
```

Esto hará que se `userRepository.save()` devuelva el mismo objeto de usuario que se pasa al método.

Mockito tiene muchas más funciones que permiten usar dobles, hacer coincidir argumentos y verificar llamadas a métodos. Para obtener más información, consulte la documentación de referencia.

Con anotaciones

Una forma alternativa de crear objetos simulados es la anotación `@Mock` de Mockito en combinación con la `MockitoExtension` de JUnit Jupiter:

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

  @Mock
  private UserRepository userRepository;

  private RegisterUseCase registerUseCase;

  @BeforeEach
  void initUseCase() {
    registerUseCase = new RegisterUseCase(userRepository);
  }

  @Test
  void savedUserHasRegistrationDate() {
    // ...
  }

}
```

La anotación `@Mock` especifica los campos en los que Mockito debe inyectar objetos simulados. El `@ExtendWith(MockitoExtension.class)` le dice a Mockito que evalúe esas anotaciones `@Mock` porque JUnit no lo hace automáticamente.

El resultado es el mismo que si se llama `Mockito.mock()` manualmente, es cuestión de gustos qué forma de usar.

Tenga en cuenta que en lugar de construir un objeto `RegisterUseCase` manualmente, también podemos usar la anotación `@InjectMocks` en el campo `registerUseCase`. Luego, Mockito creará una instancia para nosotros, siguiendo un algoritmo específico :

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

  @Mock
  private UserRepository userRepository;

  @InjectMocks
  private RegisterUseCase registerUseCase;

  @Test
  void savedUserHasRegistrationDate() {
    // ...
  }

}
```

3. Test de consultas JPA con Spring Boot y `@DataJpaTest`

Aparte de las pruebas unitarias, las pruebas de integración juegan un papel vital en la producción de software de calidad. Un tipo especial de prueba de integración se ocupa de la integración entre nuestro código y la base de datos.

Con la anotación @DataJpaTest, Spring Boot proporciona una forma conveniente de configurar un entorno con una base de datos incorporada para probar nuestras consultas de base de datos.

En este tutorial, primero discutiremos qué tipos de consultas son dignas de prueba y luego discutiremos diferentes formas de crear un esquema de base de datos y un estado de base de datos para probar.

3.1 ¿Qué probar?

La primera pregunta que debemos respondernos es qué debemos probar. Consideremos un repositorio de Spring Data responsable de los objetos `UserEntity` :

```
interface UserRepository extends CrudRepository<UserEntity, Long> {
  // query methods
}
```

Disponemos de diferentes opciones para crear consultas. Veamos algunos de ellos en detalle para determinar si debemos cubrirlos con pruebas.

3.1.1 Consultas inferidas

La primera opción es crear una consulta inferida :

```
UserEntity findByName(String name);
```

No necesitamos decirle a Spring Data qué hacer, ya que infiere automáticamente la consulta SQL del nombre del nombre del método.

Lo bueno de esta función es que Spring Data **también verifica automáticamente si la consulta es válida al inicio**. Si cambiamos el nombre del método a `findByFoo()` y `UserEntity` no tiene una propiedad `foo`, Spring Data nos lo señalará con una excepción:

```
org.springframework.data.mapping.PropertyReferenceException:
  No property foo found for type UserEntity!
```

Entonces, siempre que tengamos al menos una prueba que intente iniciar el contexto de la aplicación Spring en nuestra base de código, no es necesario que escriba una prueba adicional para nuestra consulta inferida.

Tenga en cuenta que esto no es cierto para las consultas inferidas de nombres de métodos largos como `findByNameAndRegistrationDateBeforeAndEmailIsNotNull()`. **El nombre de este método es difícil de entender y fácil de equivocarse, por lo que deberíamos probar si realmente hace lo que pretendíamos.**

Habiendo dicho esto, es una buena práctica cambiar el nombre de dichos métodos a un nombre más corto y significativo y agregar una anotación `@Query` para proporcionar una consulta JPQL personalizada.

3.1.2 Consultas JPQL personalizadas con `@Query`

Si las consultas se vuelven más complejas, tiene sentido proporcionar una consulta JPQL personalizada:

```
@Query("select u from UserEntity u where u.name = :name")
UserEntity findByNameCustomQuery(@Param("name") String name);
```

De manera similar a las consultas inferidas, obtenemos una verificación de validez para esas consultas JPQL de forma gratuita . Usando Hibernate como nuestro proveedor JPA, obtendremos un `QuerySyntaxException` al inicio si encuentra una consulta no válida:

```
org.hibernate.hql.internal.ast.QuerySyntaxException:
  unexpected token: foo near line 1, column 64 [select u from ...]
```

Sin embargo, las consultas personalizadas pueden resultar mucho más complicadas que encontrar una entrada mediante un solo atributo. Pueden incluir combinaciones con otras tablas o devolver DTO complejos en lugar de una entidad, por ejemplo.

Entonces, ¿deberíamos escribir pruebas para consultas personalizadas? La respuesta insatisfactoria es que **tenemos que decidir por nosotros mismos si la consulta es lo suficientemente compleja como para requerir una prueba**.

3.1.3 Consultas nativas con @Query

Otra forma es utilizar una consulta nativa :

```
@Query(
  value = "select * from user as u where u.name = :name",
  nativeQuery = true)
UserEntity findByNameNativeQuery(@Param("name") String name);
```

En lugar de especificar una consulta JPQL, que es una abstracción sobre SQL, estamos especificando una consulta SQL directamente. Esta consulta puede utilizar un dialecto SQL específico de la base de datos.

Es importante tener en cuenta que ni Hibernate ni Spring Data validan las consultas nativas al inicio. Dado que la consulta puede contener SQL específico de la base de datos, no hay forma de que Spring Data o Hibernate puedan saber qué buscar.

Por tanto, las consultas nativas son las principales candidatas para las pruebas de integración. Sin embargo, si realmente usan SQL específico de la base de datos, es posible que esas pruebas no funcionen con la base de datos incorporada en la memoria, por lo que tendríamos que proporcionar una base de datos real en segundo plano (por ejemplo, en un contenedor de ventana acoplable que se configura a pedido en proceso de integración continua).

3.2 Uso de `@DataJpaTest`

Supongamos que tenemos el siguiente modelo:

```
package com.bezkoder.spring.data.jpa.test.model;

import javax.persistence.*;

@Entity
@Table(name = "tutorials")
public class Tutorial {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Column(name = "title")
    private String title;

    @Column(name = "description")
    private String description;

    @Column(name = "published")
    private boolean published;

    public Tutorial() {

    }

    public Tutorial(String title, String description, boolean published) {
        this.title = title;
        this.description = description;
        this.published = published;
    }

    public long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
```

```
    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public boolean isPublished() {
        return published;
    }

    public void setPublished(boolean isPublished) {
        this.published = isPublished;
    }

    @Override
    public String toString() {
        return "Tutorial [id=" + id + ", title=" + title + ", desc=" + description
+ ", published=" + published + "]";
    }
}
```

Y el siguiente repositorio:

```
package com.bezkoder.spring.data.jpa.test.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import com.bezkoder.spring.data.jpa.test.model.Tutorial;

public interface TutorialRepository extends JpaRepository<Tutorial, Long> {
  List<Tutorial> findByPublished(boolean published);

  List<Tutorial> findByTitleContaining(String title);
}
```

Ahora podemos usar métodos de JpaRepository: `save()`, `findOne()`, `findById()`, `findAll()`, `count()`, `delete()`, `deleteById()`... y sin la aplicación de estos métodos.

También definimos métodos de búsqueda personalizados:

- `findByPublished()`: devuelve todos los tutoriales con valor `published` como entrada.
- `findByTitleContaining()`: devuelve todos los tutoriales cuyo título contiene la entrada `title`.

Spring Data JPA conecta la implementación automáticamente.

3.2.1 Pruebas unitarias con `@DataJpaTest`

Para probar los repositorios Spring Data JPA, o cualquier otro componente relacionado con JPA, Spring Boot proporciona la anotación `@DataJpaTest` . Podemos simplemente agregarlo a nuestra prueba unitaria y configurará un contexto de aplicación Spring:

```
@ExtendWith(SpringExtension.class)
@DataJpaTest
class UserEntityRepositoryTest {

  @Autowired private DataSource dataSource;
  @Autowired private JdbcTemplate jdbcTemplate;
  @Autowired private EntityManager entityManager;
  @Autowired private UserRepository userRepository;

  @Test
  void injectedComponentsAreNotNull(){
    assertThat(dataSource).isNotNull();
    assertThat(jdbcTemplate).isNotNull();
    assertThat(entityManager).isNotNull();
    assertThat(userRepository).isNotNull();
  }
}
```

> **@ExtendWith**
> Ya no es obligatorio su uso desde la versión 2.1 de Spring Boot.

El contexto de la aplicación así creado no contendrá todo el contexto necesario para nuestra aplicación Spring Boot, sino solo una "porción" que contiene los componentes necesarios para inicializar cualquier componente relacionado con JPA como nuestro repositorio Spring Data.

Podemos, por ejemplo, inyectar a `DataSource`, `@JdbcTemplate` o `@EntityManager` en nuestra clase de prueba si los necesitamos. Además, podemos inyectar cualquiera de los repositorios de Spring Data desde nuestra aplicación. Todos los componentes anteriores se configurarán automáticamente para apuntar a una base de datos incorporada en memoria en lugar de la base de datos "real" que podríamos haber configurado en `application.properties` o `application.yml`.

Tenga en cuenta que, de forma predeterminada, el contexto de la aplicación que contiene todos estos componentes, incluida la base de datos en memoria, se comparte entre todos los métodos de prueba dentro de todas las clases de prueba anotadas con `@DataJpaTest`.

Esta es la razón por la que, de forma predeterminada, cada método de prueba se ejecuta en su propia transacción, que se revierte una vez que se ha ejecutado el método . De esta manera, el estado de la base de datos permanece impecable entre las pruebas y las pruebas se mantienen independientes entre sí.

3.2.2 Creación y llenado de la base de datos

ESTE APARTADO ESTÁ PENDIENTE DE TERMINAR

Por ahora, nuestros proyectos no han utilizado una base de datos real, y por tanto, lo que vamos a hacer ahora con las pruebas se parece mucho a lo que venimos haciendo con nuestras bases de datos H2 embebidas. Podremos notar la diferencia cuando trabajemos con una base de datos real como postgresql.

A incluir:

- Uso de Flyway
- Uso de Liquibase
- Uso de DBUnit

3.2.3 Pruebas utilizando la capa de persistencia de JPA

> Si con las pruebas unitarias de los servicios, nuestra intención era proporcionar una serie de valores de entrada *acordados* y creados específicamente para tratar de obtener un resultado esperado conocido de antemano, en el caso de las pruebas a nivel de base de datos **buscamos dejar la base de datos en un estado concreto (valores de entrada) para que cuando ejecutemos una consulta, podamos obtener el resultado esperado**.

Una primera forma de implementar las pruebas de nuestros repositorios sería utilizando los componentes, bien de JPA (como el *Entity Manager*) o bien a nivel de JDBC (como `@JdbcTemplate`) para realizar operaciones de inserción o actualización que me permitan dejar la base de datos en el estado deseado.

Veamoslo con un ejemplo:

```
@DataJpaTest
public class JPAUnitTest {

  @Autowired
  private TestEntityManager entityManager;

  @Autowired
  TutorialRepository repository;

  @Test
  public void should_find_no_tutorials_if_repository_is_empty() {
    Iterable<Tutorial> tutorials = repository.findAll();

    assertThat(tutorials).isEmpty();
  }

  @Test
  public void should_store_a_tutorial() {
    Tutorial tutorial = repository.save(new Tutorial("Tut title", "Tut desc",
true));

    assertThat(tutorial).hasFieldOrPropertyWithValue("title", "Tut title");
    assertThat(tutorial).hasFieldOrPropertyWithValue("description", "Tut desc");
    assertThat(tutorial).hasFieldOrPropertyWithValue("published", true);
  }

  @Test
```

```
  public void should_find_all_tutorials() {
    Tutorial tut1 = new Tutorial("Tut#1", "Desc#1", true);
    entityManager.persist(tut1);

    Tutorial tut2 = new Tutorial("Tut#2", "Desc#2", false);
    entityManager.persist(tut2);

    Tutorial tut3 = new Tutorial("Tut#3", "Desc#3", true);
    entityManager.persist(tut3);

    Iterable<Tutorial> tutorials = repository.findAll();

    assertThat(tutorials).hasSize(3).contains(tut1, tut2, tut3);
  }

  // resto de métodos

}
```

> NOTA: Este ejemplo hace uso de AssertJ para hacer más legibles los asertos

En el método `should_store_a_tutorial()` encontramos lo siguiente:

- Haciendo uso del propio repositorio insertamos un nuevo tutorial.
- Obtenemos la referencia devuelta por el método `save(...)`.
- Comprobamos que el objeto de tipo `Tutorial` devuelto tiene todos los campos que debería tener.

En el método `should_find_all_tutorials()` encontramos lo siguiente:

- Usando el `entityManager` persistimos hasta 3 instancias de `Tutorial`.
- Obtenemos los tutoriales existentes actualmente en la base de datos a través del método `findAll()`.
- Comprobamos que este último devuelve 3 elementos (los que insertamos al principio) y que estos elementos son los que hemos creado en el mismo método.

Algunos métodos más que podríamos encontrar en la clase podrían ser:

```
@DataJpaTest
public class JPAUnitTest {

  @Autowired
  private TestEntityManager entityManager;

  @Autowired
  TutorialRepository repository;


  // métodos ya explicados

  // algunos métodos más
  @Test
  public void should_find_tutorials_by_title_containing_string() {
```

```
    Tutorial tut1 = new Tutorial("Spring Boot Tut#1", "Desc#1", true);
    entityManager.persist(tut1);

    Tutorial tut2 = new Tutorial("Java Tut#2", "Desc#2", false);
    entityManager.persist(tut2);

    Tutorial tut3 = new Tutorial("Spring Data JPA Tut#3", "Desc#3", true);
    entityManager.persist(tut3);

    Iterable<Tutorial> tutorials = repository.findByTitleContaining("ring");

    assertThat(tutorials).hasSize(2).contains(tut1, tut3);
  }


  @Test
  public void should_update_tutorial_by_id() {
    Tutorial tut1 = new Tutorial("Tut#1", "Desc#1", true);
    entityManager.persist(tut1);

    Tutorial tut2 = new Tutorial("Tut#2", "Desc#2", false);
    entityManager.persist(tut2);

    Tutorial updatedTut = new Tutorial("updated Tut#2", "updated Desc#2", true);

    Tutorial tut = repository.findById(tut2.getId()).get();
    tut.setTitle(updatedTut.getTitle());
    tut.setDescription(updatedTut.getDescription());
    tut.setPublished(updatedTut.isPublished());
    repository.save(tut);

    Tutorial checkTut = repository.findById(tut2.getId()).get();

    assertThat(checkTut.getId()).isEqualTo(tut2.getId());
    assertThat(checkTut.getTitle()).isEqualTo(updatedTut.getTitle());
    assertThat(checkTut.getDescription()).isEqualTo(updatedTut.getDescription());
    assertThat(checkTut.isPublished()).isEqualTo(updatedTut.isPublished());
  }

  // resto de métodos

}
```

El método `should_find_tutorials_by_title_containing_string()` realiza lo siguiente:

- Persiste 3 instancias de tutorial nuevas.
- Solamente 2 de ellas incluyen la cadena `ing`.
- Busca, usando la consulta `findByTitleContaining("ring");` aquellos tutoriales que incluyan dicha cadena en el título.
- Comprueba que encuentra a dos, que coinciden con las creadas anteriormente.

Por otro lado, el método `should_update_tutorial_by_id()` realiza lo siguiente:

- Persiste, usando el `entityManager`, dos tutoriales, `tut1` y `tut2`.
- Crea una instancia de otro tutorial, para posteriormente usar sus datos para actualizar.
- Busca uno de los tutoriales creados antes con `findById`.
- Actualiza sus campos y lo salva con `save`.
- Rescata de nuevo el tutorial actualizado y verifica que los valores que provienen de la base de datos son correctos.

Para probar las consultas de la base de datos, necesitamos los medios para crear un esquema y completarlo con algunos datos. Dado que las pruebas deben ser independientes entre sí, es mejor hacer esto para cada prueba por separado.

Para pruebas simples y entidades de base de datos simples, basta con crear el estado manualmente creando y guardando entidades JPA. Para escenarios más complejos, necesitamos otras herramientas, como el uso de `@Sql`.

3.2.4 Pruebas utilizando `@Sql`

Un enfoque diferente es usanod la anotación `@Sql` de Spring. De esta forma, podemos crear ficheros SQL que inserten los datos necesarios en la base de datos para poder realizar nuestras pruebas.

```
-- createUser.sql
INSERT INTO USER
            (id,
             NAME,
             email)
VALUES      (1,
             'Zaphod Beeblebrox',
             'zaphod@galaxy.net');
```

En nuestra prueba, simplemente podemos usar la anotación `@Sql` para hacer referencia al archivo SQL para completar la base de datos:

```
@ExtendWith(SpringExtension.class)
@DataJpaTest
class SqlTest {

  @Autowired
  private UserRepository userRepository;

  @Test
  @Sql("createUser.sql")
  void whenInitializedByDbUnit_thenFindsByName() {
    UserEntity user = userRepository.findByName("Zaphod Beeblebrox");
    assertThat(user).isNotNull();
  }

}
```

Si necesitamos más de un script, podemos utilizar la anotación `@SqlGroup` para combinarlos.

Podemos configurar la forma en que analizamos y ejecutamos los scripts SQL usando la anotación `@SqlConfig` .

`@SqlConfig` se puede declarar a nivel de clase, donde sirve como configuración global. O puede usarse para configurar una anotación `@Sql` particular .

Veamos un ejemplo en el que especificamos la codificación de nuestros scripts SQL así como el modo de transacción para ejecutar los scripts:

```
@Test
@Sql(scripts = {"/import_senior_employees.sql"},
  config = @SqlConfig(encoding = "utf-8", transactionMode =
TransactionMode.ISOLATED))
public void testLoadDataForTestCase() {
    assertEquals(5, employeeRepository.findAll().size());
}
```

Y veamos los diversos atributos de `@SqlConfig` :

- `blockCommentStartDelimiter` : delimitador para identificar el inicio de los comentarios de bloque en archivos de script SQL
- `blockCommentEndDelimiter` : delimitador para indicar el final de los comentarios de bloque en archivos de script SQL
- `commentPrefix` - prefijo para identificar comentarios de una sola línea en archivos de script SQL
- `dataSource` : nombre del bean `javax.sql.DataSource` en el que se ejecutarán los scripts y las sentencias.
- `encoding` : codificación para los archivos de script SQL, el valor predeterminado es la codificación de plataforma
- `errorMode` : modo que se utilizará cuando se encuentre un error al ejecutar los scripts.
- `separator` : cadena utilizada para separar declaraciones individuales, el valor predeterminado es "-"
- `transactionManager` : nombre de bean del `PlatformTransactionManager` que se utilizará para las transacciones
- `transactionMode` : el modo que se utilizará al ejecutar scripts en una transacción.

PENDIENTE DE TERMINAR

Ver https://www.concretepage.com/spring-5/sql-example-spring-test

Veamos un ejemplo en el que aplicaremos cierta configuración

> Hasta ahora todos nuestros proyectos han trabajado con H2. Esto en la realidad no será así. Seguro que utilizaremos bases de datos "de verdad", como Postgresql, Mysql u Oracle. H2 está orientada a tareas de prototipado, testeo (como el que vamos a hacer ahora), etc...

Dado que en nuestros proyectos ya hemos utilizado como **base de datos principal** una base de datos H2, vamos a tener que configurar una base de datos secundaria, también en H2, para realizar el testeo.

Bibliografía

1. https://reflectoring.io/unit-testing-spring-boot/
2. https://www.infoworld.com/article/3537563/junit-5-tutorial-part-1-unit-testing-with-junit-5-mockito-and-hamcrest.html
3. https://bezkoder.com/spring-boot-unit-test-jpa-repo-datajpatest/