Programa de Mockito

Mockito

Academia Java

Entregable

Natalia Esquivel Ocadiz

19 de septiembre 2025

Introducción:
En el siguiente proyecto se implementó Mockito al proyecto Rest Controller con Mongo DB para asegurar la calidad del código. Mockito (Mocking Framework) es una herramienta para probar una clase de forma aislada minimizando la configuración de sus dependencias, utilizado comúnmente para probar DAO, DB, REST API, entre otros. Mockito usa Mocks para crear "dobles" de la dependencia.

Objetivos:

- Implementar Mocking en el proyecto RestMongoDB para asegurar el funcionamiento de la clase Service y sus métodos
- Verificar las veces que se llama a un método del Mock
- Probar que las excepciones implementadas sean lanzadas correctamente

Implementación: Explicación concisa de la solución

En un proyecto Maven no es necesario añadir la dependencia Mockito, ya que está soportada por JUnit y se agrega en la librería por default.
Los tests usados tienen la siguiente estructura:

1. En el package test, se define la dependencia a mockear, la cual es el Repositorio ya que está es la dependencia que conecta nuestro Service con la BD.

@Mock: Se "mockea" el ProductoRepository
@InjectMocks: Inyectamos el ProductoRepository a ProductoService Impl ya que es la clase que deseamos probar

2. Se instancia la clase a probar, en este caso Producto
3. Se usa la anotación @BeforeEach para crear y setear los atributos de un producto que se ocupa en todo los tests, de esta manera evitamos la duplicación
4. Al igual que en JUnit 5, @Test y @DisplayName ayudan a marcar a un método como método de prueba, y display para darle un nombre a la prueba, respectivamente. Se siguen las buenas prácticas para el nombre de test.
5. Set up: Se definen las expectativas de las respuestas del mock, con when(this methodiscalled).thenReturn(this). En este ejemplo, se prueba "save"

```
when(productoRepository.save(producto)).thenReturn(producto);
```

6. Execute: se manda a llamar el método a probar

```
Producto productoGuardado = productoService.save(producto);
```

7. Assert: Se revisa que el resultado sea el esperado, empleando assertEquals para asegurar que apunten al mismo objeto

```
assertEquals("Libro", productoGuardado.getNombreProducto());
    assertEquals("Farenheit 591", productoGuardado.getDescripcion());
```

8. Verify; Se verifica que el se haya llamado al método de Repositorio y cuántas veces

```
verify(productoRepository, times(1)).save(producto);
```

9. Se realizó el mismo procedimiento para cada uno de los métodos del Service: get, update, delete and post. Además de la excepción que se arroja cuando se quiere actualizar un producto que no existe:

```
@DisplayName("When & Verify Caso excepción de actualizar producto")
    @Test
    public void updateProductoNoExistente() {
       String idInexistente= "00";
       Producto productoActualizado = new Producto("Libro Clásico",
"1984", 400.00,10);
       when(productoRepository.findById(idInexistente))
       .thenReturn(Optional.empty());

       RuntimeException exception =
*assertThrows(RuntimeException.class,
                    ()-> {
                      productoService.update(idInexistente,
productoActualizado);});
       assertEquals("Producto no encontrado con id: "+
idInexistente,exception.getMessage() );
       verify(productoRepository, times(1)).findById(idInexistente);
       verifyNoMoreInteractions(productoRepository);
    }
```

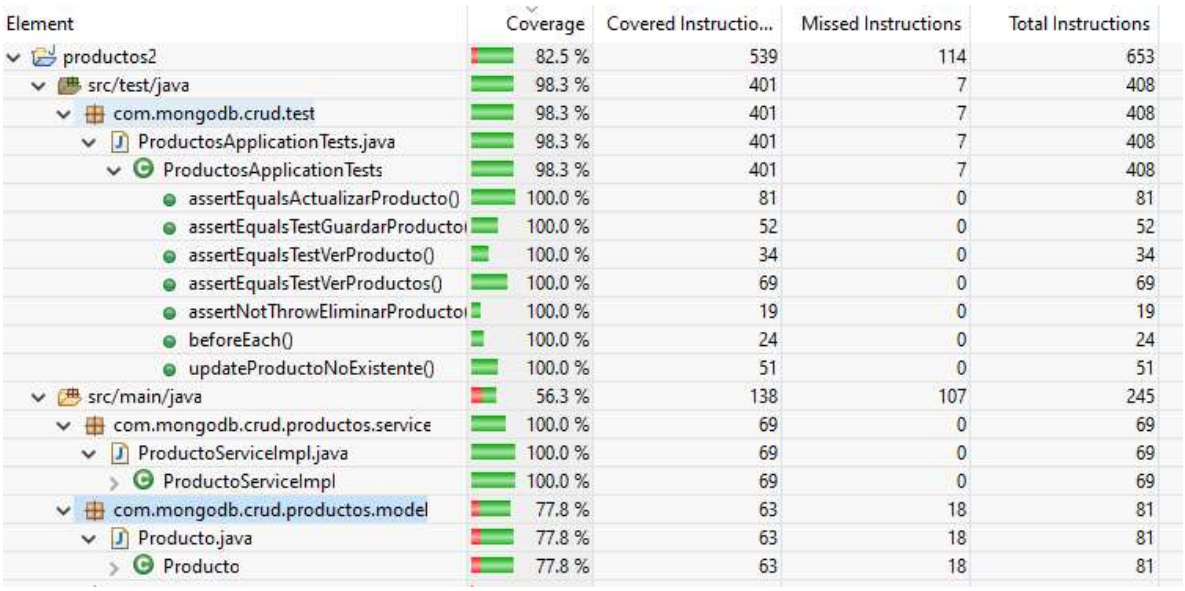
*Se hizo uso del Assertion assertThrows para verificar que el bloque de código lance la excepción especificada y la sentencia verifyNoMoreInteractions verifica que el código no ejecute ninguna otra acción con el idInexistente ya que no cumple con la condición de que debe existir un producto con ese id.

Resultados:

Se realizaron 7 tests en total, y todos resultaron exitosos en lo esperado

```
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.742 s -- in com.mongodb.crud.test.ProductosApplicationTests
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.035 s
[INFO] Finished at: 2025-09-20T20:11:44-06:00
[INFO] ------------------------------------------------------------------------
```

Métricas de cobertura: Se logró un Coverage del 82.5%, 2.5% más del mínimo requerido.



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
productos2	82.5 %	539	114	653
src/test/java	98.3 %	401	7	408
com.mongodb.crud.test	98.3 %	401	7	408
ProductosApplicationTests.java	98.3 %	401	7	408
ProductosApplicationTests	98.3 %	401	7	408
assertEqualsActualizarProducto()	100.0 %	81	0	81
assertEqualsTestGuardarProducto	100.0 %	52	0	52
assertEqualsTestVerProducto()	100.0 %	34	0	34
assertEqualsTestVerProductos()	100.0 %	69	0	69
assertNotThrowEliminarProducto	100.0 %	19	0	19
beforeEach()	100.0 %	24	0	24
updateProductoNoExistente()	100.0 %	51	0	51
src/main/java	56.3 %	138	107	245
com.mongodb.crud.productos.service	100.0 %	69	0	69
ProductoServiceImpl.java	100.0 %	69	0	69
ProductoServiceImpl	100.0 %	69	0	69
com.mongodb.crud.productos.model	77.8 %	63	18	81
Producto.java	77.8 %	63	18	81
Producto	77.8 %	63	18	81

Reflexiones:

Las pruebas con Mockito han demostrado ser una herramienta fundamental para el testing de unidades aisladas, permitiendo probar la lógica de negocio del Service sin depender de la base de datos real. El uso de anotaciones como @Mock y @InjectMocks simplificó significativamente la configuración de las pruebas y mejoró la legibilidad del código. La capacidad de verificar el número exacto de interacciones con las dependencias proporcionó mayor confianza en el comportamiento esperado del sistema. Se cumplió con el coverage mínimo requerido, validando tanto los casos exitosos como los escenarios de excepción.