**Testing Service Layer Usuario**

Capa de Servicio Usuario: Contiene la logica de negocio de nuestra aplicacion y nos permite comunicarnos con la capa de Repository y Controller.

Para realizar testing a esta capa usaremos Mockito.

**¿Qué es Mockito?.**

Mockito es una librería de testing en Java que permite crear objetos “falsos” o “mock” para simular comportamientos y comprobar el correcto funcionamiento de nuestro código.

## **¿Qué es un Mock?**

Un mock es un objeto “falso” que se utiliza en pruebas para simular un comportamiento o dependencia.

Los mocks se utilizan para realizar pruebas de manera aislada, sin depender de componentes o servicios externos que puedan afectar al resultado de las pruebas.

De esta manera, podemos asegurarnos de que nuestro código está funcionando correctamente y no hay errores en su implementación.

## **¿Cómo funciona Mockito?**

Uno de los usos más comunes de Mockito es para simular dependencias en nuestro código.



Cuando inyectamos un object mock dentro de otro mocked object, usamos la anotación @**InjectMocks**. @**InjectMock** crea los object mock de la clase e inyecta los mocks que están marcados con la anotación @**Mock** dentro de esta.

1. @Mock: Esta anotación se utiliza para crear un objeto simulado (mock) de la clase UsuarioRepository. Este objeto simulado se utilizará en lugar de una instancia real de UsuarioRepository durante las pruebas unitarias.
2. @InjectMocks: Esta anotación se utiliza para inyectar los mocks en la clase bajo prueba, en este caso, UsuarioServiceImpl. Esto significa que cualquier instancia de UsuarioRepository que sea necesaria dentro de UsuarioServiceImpl durante las pruebas se remplazará por el mock creado con @Mock.

Por lo tanto, en resumen, este fragmento de código se utiliza en el contexto de pruebas unitarias para crear mocks de dependencias y luego inyectarlos en la clase que se está probando, UsuarioServiceImpl, para poder simular su comportamiento durante las pruebas.

BBDMockito

BDDMockito es una extensión de Mockito que proporciona un estilo de escritura basado en el "Behavior-Driven Development" (BDD). Este estilo se centra en describir el comportamiento esperado del sistema en términos de escenarios y comportamientos observables.

BDDMockito.when() es una forma de utilizar la sintaxis BDD para definir el comportamiento esperado de un mock en Mockito. En lugar de utilizar directamente Mockito.when(), puedes utilizar BDDMockito.given() para mejorar la legibilidad y claridad de tus pruebas.

Por ejemplo, en lugar de escribir:

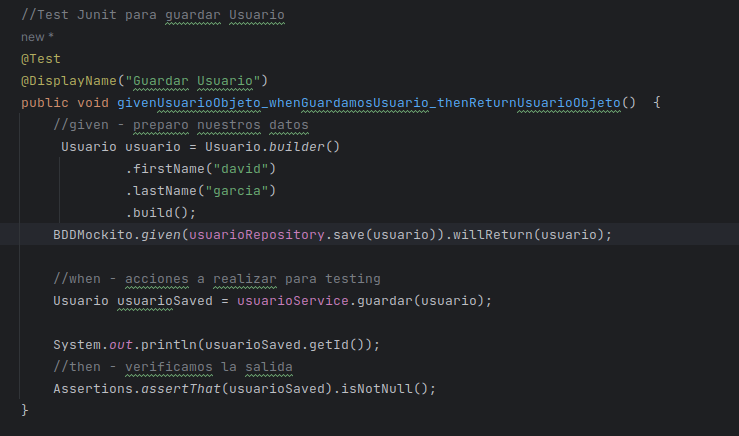
***Mockito.when(mockObjeto.metodo()).thenReturn(resultado);***

Por ejemplo, en lugar de escribir:

***BDDMockito.given(mockObjeto.metodo()).willReturn(resultado);***

1- Guardar Usuario

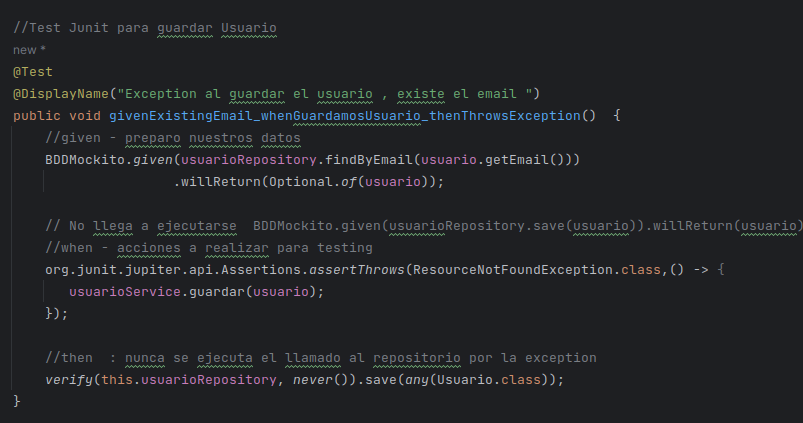
En el primer test vamos a mostrar como guardar el usuario y como crear el repositorio con Mockito y como inyectarse en la capa de Sevicio.



2 - Refactorizar y utilizamos @Mock and @InjectMock anotaciones para crear los objetos Mock en este caso UsuarioRepository e injectarlas en UsuarioServiceImpl.

3 - Throws custom Exception al guardar el Usuario.

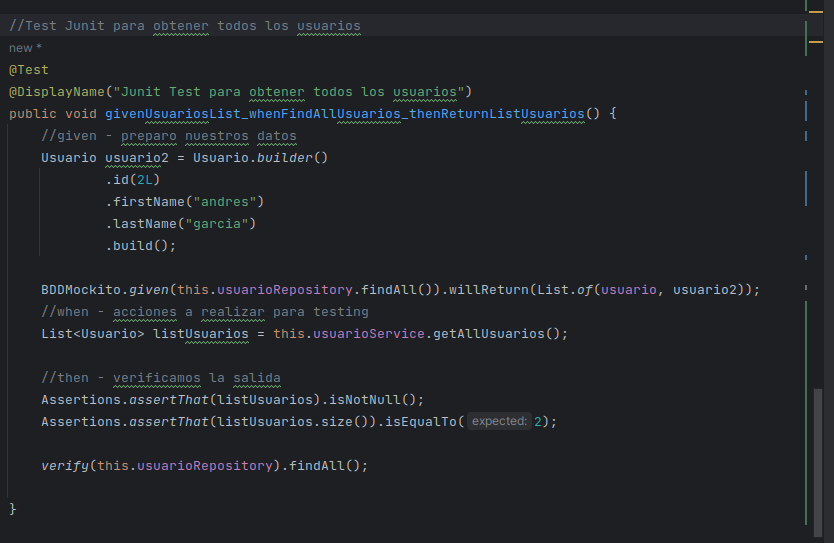
Al guardar un determinado usuario en el codigo se verifica que existe un email y si existe indica una exception que “Existe el email asignado a otra persona”.



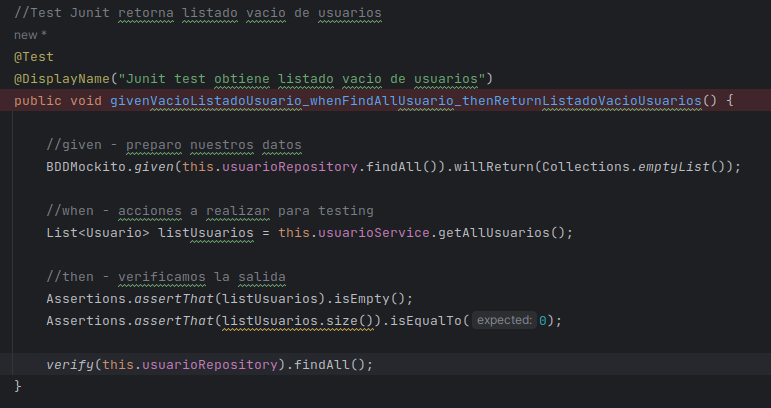
En este caso se observa que se produce una exception y para capturar usamos assertThrows , este captura puesto existe el email y no permite guardar el usuario.

4 - Junit Test para todos los usuarios.

Creamos en la capa de Servicio el metodo de obtener usuarios listado, realizamos la implementacion del mismo metodo y creamos el test para este metodos.



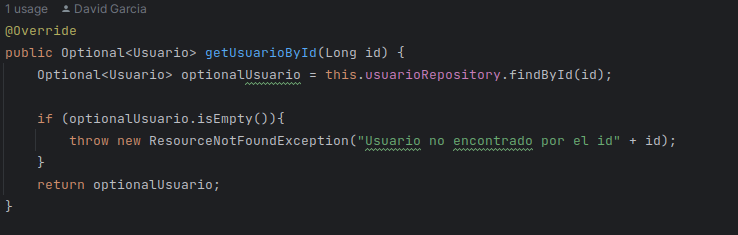
5 - Caso contrario que devuelva un listado vacio de usuarios.



6 - Agregamos el metodo Buscar Usuario por Id en este caso vamos a agregar en la interfaz de usuario service el metodo getUsuarioById(Long id).

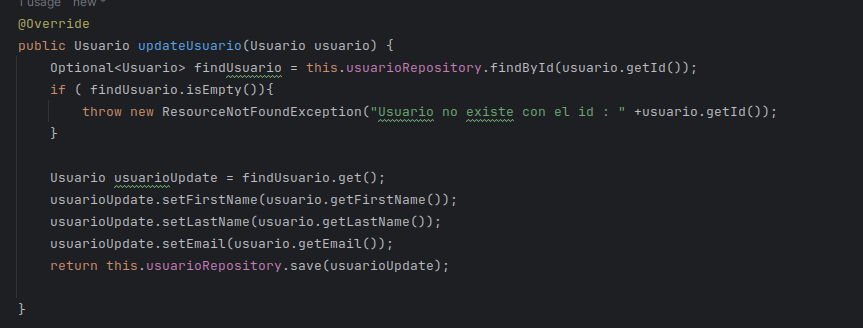
Agregamos la implementación del método .

Agregamos el test para el método getUsuarioById .



7 - Agregamos el metodo de actualizar un Usuario por el id del Usuario.

Agregamos la implementacion del metodo en la capa de servicio



Agregamos el test para actualizar un Usuario. Que paso en el siguiente código que dio error :



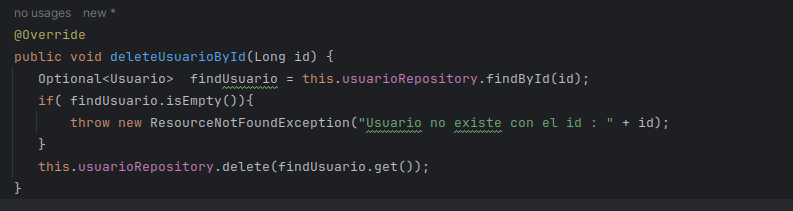
El error que estás viendo en tu test se debe a que, aunque estás mockeando el método save del usuarioRepository, no has mockeado el comportamiento del método findById, que es invocado dentro de tu método updateUsuario para buscar el usuario antes de actualizarlo. Dado que no se ha especificado un comportamiento para findById, por defecto, Mockito devolverá un Optional.empty(), lo que hace que tu implementación lance una ResourceNotFoundException.

Para corregir este error, debes asegurarte de mockear también el comportamiento de findById para que devuelva un Optional que contenga el usuario que quieres actualizar. Aquí te muestro cómo podrías hacerlo:

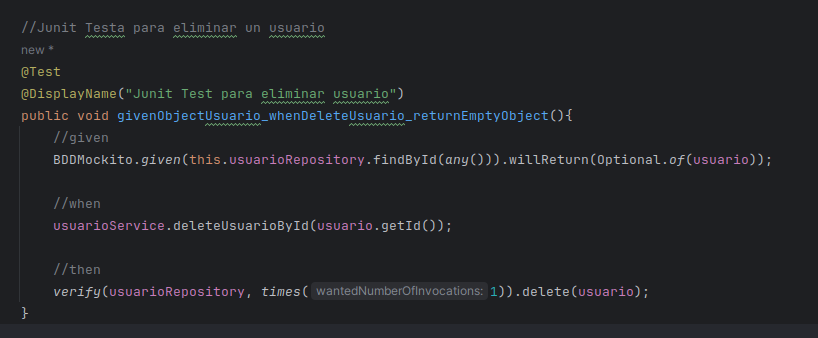


8 - Delete usuario by Id

Creamos el metodo en la capa de Servicio y la implementación del eliminado



Agregamos el test para el eliminado del usuario



**verify**: Es una función de Mockito que se utiliza para verificar que se hayan realizado ciertas interacciones (llamadas a métodos) en un objeto mockeado durante la prueba. Es parte de asegurar que la lógica que estás probando se comporta como esperas, no solo en términos de resultados (retornos de métodos o cambios de estado), sino también en cómo interactúa con sus dependencias.

**times(1)**: Especifica la cantidad exacta de veces que esperas que se haya llamado al método delete sobre el objeto usuarioRepository durante la ejecución de la prueba. En este caso, estás esperando que se llame exactamente una vez. Si el método se llamara más veces o ninguna, la prueba fallaría.

9 - Test cuando no existe el id a eliminar

