

## Prueba de integración JUnit y Mockito con Maven



En aplicaciones grandes, las clases pueden depender de otros servicios o componentes (como bases de datos, sistemas externos, servicios web, etc.). Mockito permite crear simulaciones (mocks) de estos componentes para que el enfoque de las pruebas sea únicamente la **lógica de negocio** de la clase bajo prueba. Esto asegura que las pruebas sean rápidas y aisladas, y que no se interfiera con el sistema completo.

Supongamos que tienes un **servicio** que interactúa con un repositorio de base de datos para obtener información de usuarios. Usando Mockito, puedes simular el repositorio en las pruebas y verificar si el servicio realiza correctamente las acciones esperadas con los datos, sin necesidad de una base de datos real.

Vamos a hacer un ejemplo de un servicio que necesita información de usuarios.

1. Crea un **proyecto en Maven** (simple project (skip archetype selection)) (En IntelliJ → New project, Java, en Build seleccionar Maven.).

- Group Id: com.example (Identificador del grupo)
- Artifact Id: UserAppMock (Nombre del proyecto)
- Version: 1.0-SNAPSHOT (Se mantiene por defecto)
- Packaging: jar

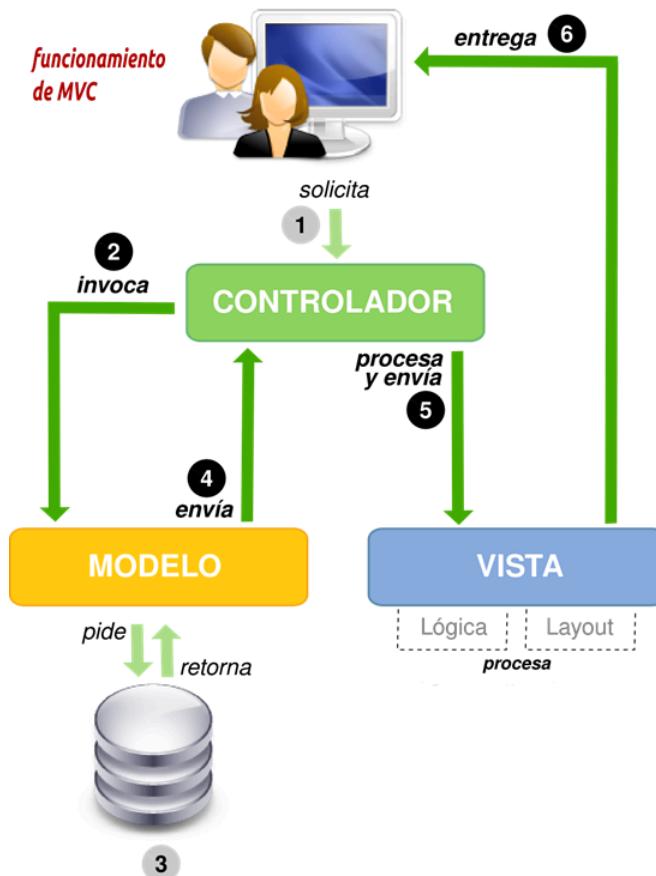
2. Añade las dependencias de JUnit y Mockito a tu **pom.xml**:

```
<dependencies>
  <!-- JUnit 5 -->
  <dependency>
    <groupId>org.junit.jupiter</groupId>
```

```
<artifactId>junit-jupiter-api</artifactId>
<version>5.9.1</version>
<scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.1</version>
    <scope>test</scope>
</dependency>

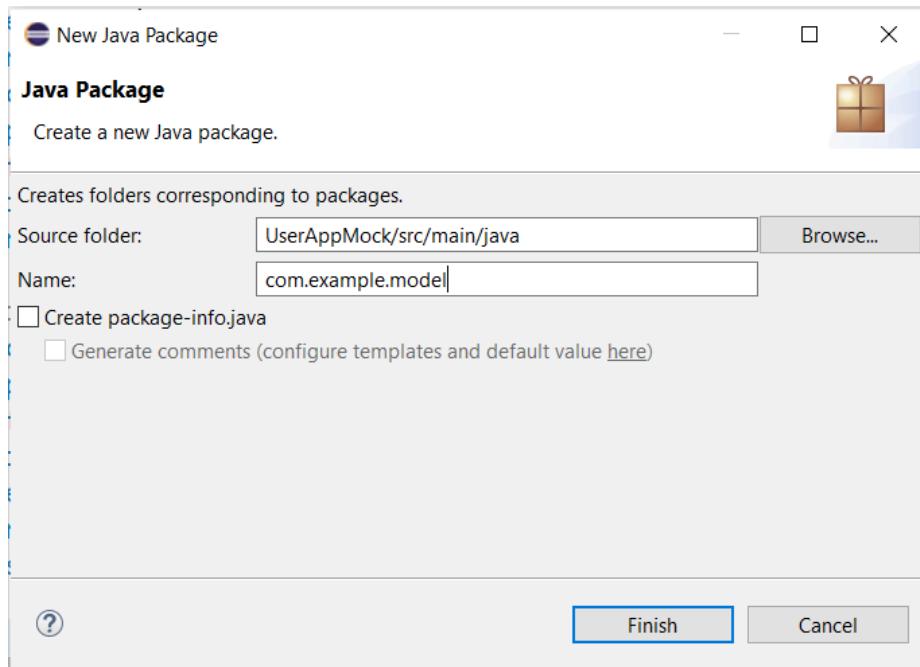
<!-- Mockito -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.0.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>4.0.0</version>
    <scope>test</scope>
</dependency>
</dependencies>
```

3. Vamos a recordar el Modelo Vista Controlador para estructurar un proyecto.



- Define la **estructura de carpetas** siguiente, en Eclipse no se crean por defecto:

```
UserAppMock
├── src/main/java
│   ├── com.example.model      (Clases del modelo, aquí va la clase User.java)
│   ├── com.example.repository (Capa de datos, aquí va la interfaz UserRepository.java)
│   └── com.example.service    (Lógica del negocio, aquí va la clase UserService.java)
└── src/test/java
    └── com.example.service    (Pruebas unitarias, aquí va UserServiceTest.java)
    pom.xml
```



### 5. Añade la clase **User.java**:

La clase User es un modelo de datos que representa a un usuario en el sistema.

Tiene dos propiedades principales:

- `id`: Un identificador único de tipo Long para el usuario.
- `name`: El nombre del usuario.

```
package com.example.model;

public class User {
    private Long id;
    private String name;

    // Constructor
    public User(Long id, String name) {
        this.id = id;
        this.name = name;
    }

    // Getters y Setters
    public Long getId() {
        return id;
    }
```

```
public void setId(Long id) {  
    this.id = id;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
}
```

## 6. Añade la clase **UserRepository.java**

La interfaz `UserRepository` define una **abstracción para acceder a los datos** de los usuarios. En una aplicación real, esta interfaz probablemente sería implementada por un framework como Spring Data JPA, pero en este caso la interfaz es simple.

Método `findById(Long id)` : Este método permite buscar un usuario por su id. Devuelve un `Optional<User>`, que es una forma segura de manejar valores que podrían ser null (es decir, si no se encuentra un usuario con el id dado, se retorna un `Optional.empty()`).

```
package com.example.repository;  
  
import com.example.model.User;  
import java.util.Optional;  
  
public interface UserRepository {  
    Optional<User> findById(Long id);  
}
```

## 7. Añade la clase **UserService.java**

La clase `UserService` contiene la lógica de negocio relacionada con los usuarios.

**Dependencia de UserRepository:** El servicio depende del UserRepository, que es un repositorio encargado de acceder a los datos (en este caso, los usuarios). El repositorio se inyecta a través del constructor.

Método `getUserName(Long id)`:

Llama al método `findById(id)` del repositorio para obtener un `Optional<User>`.

El método `isPresent()` es para verificar si el usuario existe (es decir, si el Optional no está vacío).

- Si el usuario existe, obtiene el nombre del usuario con `user.get().getName()` y lo devuelve.
- Si el usuario no se encuentra, devuelve el mensaje "Usuario no encontrado".

Este es el servicio que manejaría la lógica de obtener el nombre de un usuario dado un `id`.

```
package com.example.service;

import com.example.model.User;
import com.example.repository.UserRepository;
import java.util.Optional;

public class UserService {
    private final UserRepository userRepository;

    // Constructor
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public String getUserName(Long id) {
        Optional<User> user = userRepository.findById(id);
        if (user.isPresent()) {
            return user.get().getName();
        } else {
            return "Usuario no encontrado";
        }
    }
}
```

### 8. Añade la clase **UserServiceTest.java**:

Esta es una prueba unitaria que valida el comportamiento de la clase UserService usando JUnit y Mockito para simular el repositorio.

#### Anotaciones de Mockito:

`@Mock`: Se usa para simular (mockear) el comportamiento del UserRepository. Esto permite que no tengas que interactuar con una base de datos real, sino que simules el comportamiento del repositorio.

`@InjectMocks`: Esta anotación hace que Mockito inyecte automáticamente el userRepository simulado en el objeto userService.

#### Método setUp():

`MockitoAnnotations.openMocks(this)` inicializa las anotaciones de Mockito antes de que cada prueba se ejecute.

Método de prueba `testUserName()`:

`when(userRepository.findById(1L)).thenReturn(Optional.of(new User(1L, "Carlos")));`  
Usa Mockito para simular que el repositorio devuelve un `Optional<User>` con el usuario "Carlos" cuando se busca el usuario con id = 1L.

`assertEquals("Carlos", userService.getUserName(1L));` Llama al método `getUserName(1L)` de userService y verifica que el resultado sea "Carlos". Si el valor devuelto por el servicio es diferente, la prueba fallará.

```
package com.example.service;

import com.example.model.User;
import com.example.repository.UserRepository;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;

class UserServiceTest {
```

```

@Mock
private UserRepository userRepository;

@InjectMocks
private UserService userService;

@BeforeEach
void setUp() {
    MockitoAnnotations.openMocks(this);
}

@Test
void testGetUserName() {
    when(userRepository.findById(1L)).thenReturn(Optional.of(new User(1L, "Carlos")));

    assertEquals("Carlos", userService.getUserName(1L));
}
}

```

## 9. Ejecuta el proyecto Maven, deberías obtener 0 fallos en el test:

```

Console >
<terminated> C:\Users\vanma\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.8.v20230831-1047\jre\bin\javaw.exe (10 feb 2025 11:08:35) [pid: 9044]
[INFO] --- surefire:3.0.0:test (default-test) @ UserAppMock ---
[INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.example.service.UserServiceTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.892 s - in com.example.service.UserServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jar:3.3.0:jar (default-jar) @ UserAppMock ---
[INFO] Building jar: C:\Users\vanma\enclince-workspace\UserAppMock\target\UserAppMock-0.1-SNAPSHOT.jar

```

## 10. Si cambias Carlos por Paco en el test, debería lanzar un error:

```

16 class UserServiceTest {
17     @Mock
18     private UserRepository userRepository;
19
20     @InjectMocks
21     private UserService userService;
22
23     @BeforeEach
24     void setUp() {
25         MockitoAnnotations.openMocks(this);
26     }
27
28     @Test
29     void testGetUserName() {
30         when(userRepository.findById(1L)).thenReturn(Optional.of(new User(1L, "Carlos")));
31
32         assertEquals("Paco", userService.getUserName(1L));
33     }
34 }

```



## DESARROLLO DE INTERFACES

DAM2

Deberías encontrar los siguientes errores al ejecutar la prueba:

```
Console X
<terminated> UserAppMock (2) [Maven Build] C:\Users\vanma\p2\pool\plugins\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64.17.0.8.v20230831-1047\jre\bin\javaw.exe (10 feb 2025 10:14:27 – 10:14:43) [pid: 21120]
    at org.apache.maven.surefire.booter.ForkedBooter.run(ForkedBooter.java:507)
    at org.apache.maven.surefire.booter.ForkedBooter.main(ForkedBooter.java:495)

[INFO]
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]     UserServiceTest.testGetUserName:32 expected: <Paco> but was: <Carlos>
[INFO]
[ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time:  10.765 s
[INFO] Finished at: 2025-02-10T10:14:43+01:00
[INFO] -----
[WARNING] The requested profile "mockito" could not be activated because it does not exist.
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:3.0.0:test (default-test) on project UserA
[ERROR]
[ERROR] Please refer to C:\Users\vanma\eclipse-workspace\UserAppMock\target\surefire-reports for the individual test res
[ERROR] Please refer to dump files (if any exist) [date].dump, [date]-jvmRun[N].dump and [date].dumpstream.
[ERROR]
```