Common mistakes while using Mockito

and how to avoid them with examples

Bubu Tripathy Follow 9 min read · Mar 27, 2023

Introduction

Mockito is a popular framework for testing Java applications. It provides a powerful and easy-to-use way of *mocking* dependencies and writing unit tests. However, developers who are new to Mockito may make mistakes that can result in unreliable tests or even unexpected behavior in their applications. In this article, we will discuss common mistakes that developers make while working with Mockito framework in a Spring Boot application, along with code examples and explanations.

Misusing @Mock and @InjectMocks Annotations

One of the most common mistakes that developers make while using Mockito is misusing the *@Mock* and *@InjectMocks* annotations. *The @Mock annotation is used to create a mock object for a particular class, while the @InjectMocks annotation is used to inject the mock object into the class being tested.* It is important to note that @InjectMocks can only be used with classes, not with interfaces.

Example:

```
@RunWith(MockitoJUnitRunner.class)
public class MyServiceTest {

    @Mock
    private MyRepository myRepository;

    @InjectMocks
    private MyService myService;

    // test methods

}
```

Not Resetting Mock Objects

Mockito creates mock objects that are *reusable* across multiple tests. If a mock object is not *reset* between tests, it can lead to unexpected behavior and unreliable tests. Mockito provides a method called *Mockito.reset()* that can be

Example:

```
@Before
public void setUp() {
    MockitoAnnotations.initMocks(this);
}

@Test
public void test1() {
    Mockito.when(myRepository.findById(1)).thenReturn(Optional.of(new MyObject()
    // test code
}

@Test
public void test2() {
    Mockito.when(myRepository.findById(2)).thenReturn(Optional.of(new MyObject()
    // test code
}

@After
public void tearDown() {
    Mockito.reset(myRepository);
}
```

Using Wrong Scope for Mock Objects

Mockito creates mock objects by default with a scope of "**per class**". This means that the same mock object will be used across all test methods in a class. However, if a mock object needs to have a different state or behavior

for each test method, it should be created with a scope of "**per method**" instead.

> *To create mock objects with the correct scope, we can use the `@MockBean` annotation provided by Spring Boot. The `@MockBean` annotation creates a mock object and registers it as a bean in the application context. The scope of the mock object is "**per test method**", meaning that a new instance is created for each test method.*

Here's an example of how to use the `@MockBean` annotation to create mock objects with the correct scope:

```
@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
public class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @MockBean
    private UserRepository userRepository;

    @Test
    public void testGetUserById() throws Exception {
        // arrange
        Long userId = 1L;
        User user = new User();
        user.setId(userId);
        user.setName("John Doe");
        Mockito.when(userService.getUserById(userId)).thenReturn(user);

        // act
        MvcResult result = mockMvc.perform(get("/users/{id}", userId))
                .andExpect(status().isOk())
                .andReturn();

        // assert
```

```
        String response = result.getResponse().getContentAsString();
        assertThat(response).isEqualTo("{\"id\":1,\"name\":\"John Doe\"}");
        Mockito.verify(userService, times(1)).getUserById(userId);
    }

    @Test
    public void testAddUser() throws Exception {
        // arrange
        User user = new User();
        user.setName("Jane Doe");
        Mockito.when(userService.addUser(user)).thenReturn(user);

        // act
        MvcResult result = mockMvc.perform(post("/users")
                .contentType(MediaType.APPLICATION_JSON)
                .content("{\"name\":\"Jane Doe\"}"))
                .andExpect(status().isOk())
                .andReturn();

        // assert
        String response = result.getResponse().getContentAsString();
        assertThat(response).isEqualTo("{\"id\":null,\"name\":\"Jane Doe\"}");
        Mockito.verify(userService, times(1)).addUser(user);
    }
}
```

In this example, we use the `@WebMvcTest` annotation to test the `UserController` class, and we inject the `MockMvc` object to simulate HTTP requests. We also use the `@MockBean` annotation to create mock objects for the `UserService` and `UserRepository` classes.

> *Notice that we do not need to reset the mock objects between tests, as the `@MockBean` annotation creates new instances of the mock objects for each test method.*

Not Verifying Mock Objects

Mockito provides a method called *Mockito.verify()* that can be used to verify that a mock object was called with specific parameters. If mock objects are not verified, it can lead to unreliable tests and unexpected behavior. Here's an example of how to use the `Mockito.verify()` method to verify mock objects:

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void testGetUserById() {
        // arrange
        Long userId = 1L;
        User user = new User();
        user.setId(userId);
        user.setName("John Doe");
        Mockito.when(userRepository.findById(userId)).thenReturn(Optional.of(use

        // act
        User result = userService.getUserById(userId);

        // assert
        assertThat(result).isEqualTo(user);
        Mockito.verify(userRepository, times(1)).findById(userId);
    }

    @Test
    public void testGetUserByIdNotFound() {
        // arrange
        Long userId = 1L;
        Mockito.when(userRepository.findById(userId)).thenReturn(Optional.empty(

        // act
        UserNotFoundException exception = assertThrows(UserNotFoundException.cla
            userService.getUserById(userId);
        });
```

```
        // assert
        assertThat(exception.getMessage()).isEqualTo("User not found with ID: "
        Mockito.verify(userRepository, times(1)).findById(userId);
    }
}
```

Notice that we use the `Mockito.verify()` method to verify that the `findById()` method of the `UserRepository` class was called exactly once with the correct ID in both test methods. We use the `times(1)` argument to specify that the method should be called exactly once, and we pass in the correct ID as a parameter. If the method was not called with the correct ID, or if it was called multiple times, the test would fail.

Not Specifying the Behavior of Mock Objects

Mockito creates mock objects by default with a behavior of "**do nothing**". This means that if a method is called on a mock object and no behavior has been specified, the method will simply return null or the default value for its return type. It is important to specify the behavior of mock objects to ensure that they behave as expected in tests. Here's an example of how to use the `Mockito.when()` method to specify the behavior of mock objects:

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void testGetAllUsers() {
        // arrange
```

```
        List<User> users = Arrays.asList(
                new User(1L, "John Doe"),
                new User(2L, "Jane Doe")
        );
        Mockito.when(userRepository.findAll()).thenReturn(users);

        // act
        List<User> result = userService.getAllUsers();

        // assert
        assertThat(result).isEqualTo(users);
    }

    @Test
    public void testGetAllUsersEmpty() {
        // arrange
        List<User> users = Collections.emptyList();
        Mockito.when(userRepository.findAll()).thenReturn(users);

        // act
        List<User> result = userService.getAllUsers();

        // assert
        assertThat(result).isEqualTo(users);
    }
}
```

Notice that we use the `Mockito.when()` method to specify the behavior of the `UserRepository` mock object in both test methods. We pass in the desired return value as a parameter to the `when()` method, which tells Mockito to return this value when the specified method is called on the mock object.

Using Wrong Method for Verifying Mock Objects

Mockito provides several methods for verifying that a mock object was called with specific parameters, such as *Mockito.verify()*, *Mockito.verifyZeroInteractions*(), and *Mockito.verifyNoMoreInteractions*(). It is important to use the correct method for the desired verification, as using the wrong method can lead to unreliable tests and unexpected behavior. Here's

an example of how to use the `Mockito.verify()` method to verify mock objects:

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void testGetAllUsers() {
        // arrange
        List<User> users = Arrays.asList(
                new User(1L, "John Doe"),
                new User(2L, "Jane Doe")
        );
        Mockito.when(userRepository.findAll()).thenReturn(users);

        // act
        List<User> result = userService.getAllUsers();

        // assert
        assertThat(result).isEqualTo(users);
        Mockito.verify(userRepository).findAll();
        Mockito.verifyNoMoreInteractions(userRepository);
    }

    @Test
    public void testEmptyUserList() {
        // arrange
        List<User> users = Collections.emptyList();
        Mockito.when(userRepository.findAll()).thenReturn(users);

        // act
        List<User> result = userService.getAllUsers();

        // assert
        assertThat(result).isEqualTo(users);
        Mockito.verify(userRepository).findAll();
        Mockito.verifyNoMoreInteractions(userRepository);
        Mockito.verifyZeroInteractions(userRepository);
```

```
    }
}
```

Notice that in the second test case, we use the `Mockito.verifyZeroInteractions()` method to verify that no interactions occurred with our mock object during our test. This ensures that we are only testing the behavior we want to test, and that there are no unexpected interactions happening in our code.

Not Handling Exceptions

Here's an example of how to handle exceptions when using Mockito:

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void testGetUserById() {
        // arrange
        Long userId = 1L;
        User user = new User();
        user.setId(userId);
        user.setName("John Doe");
        Mockito.when(userRepository.findById(userId)).thenReturn(Optional.of(use

        // act
        User result = userService.getUserById(userId);

        // assert
        assertThat(result).isEqualTo(user);
    }

    @Test
```

```
    public void testGetUserByIdNotFound() {
        // arrange
        Long userId = 1L;
        Mockito.when(userRepository.findById(userId)).thenReturn(Optional.empty(

        // act and assert
        UserNotFoundException exception = assertThrows(UserNotFoundException.cla
            userService.getUserById(userId);
        });

        assertThat(exception.getMessage()).isEqualTo("User not found with ID: "
    }
}
```

In the `testGetUserByIdNotFound()` method, we mock the `findById()` method of the `UserRepository` class to return an empty optional. We then call the `getUserById()` method of the `UserService` class with a specific ID, and we expect the method to throw a `UserNotFoundException`. We use the `assertThrows()` method to verify that the correct exception is thrown, and we also use the `getMessage()` method of the exception to verify that the correct message is returned.

Not using Correct Matchers

Here's an example of how to use the correct matchers when using Mockito:

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void testAddUser() {
```

```
        // arrange
        User user = new User();
        user.setName("John Doe");
        user.setAge(30);

        // act
        userService.addUser(user);

        // assert
        ArgumentCaptor<User> captor = ArgumentCaptor.forClass(User.class);
        Mockito.verify(userRepository).save(captor.capture());
        assertThat(captor.getValue().getName()).isEqualTo("John Doe");
        assertThat(captor.getValue().getAge()).isEqualTo(30);
    }
}
```

Notice that we use the `ArgumentCaptor` class to capture the argument value passed to the `save()` method of the `UserRepository` class. We also use the `Mockito.eq()` method to specify the argument values for the method call, using the `user.getName()` and `user.getAge()` methods to get the correct values. This helps to ensure that the correct arguments are passed to the method and avoids unexpected behavior in tests.

Here's another example of how to use the correct matchers when using Mockito:

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void testDeleteUserById() {
        // arrange
```

```
        Long userId = 1L;

        // act
        userService.deleteUserById(userId);

        // assert
        Mockito.verify(userRepository, Mockito.times(1)).deleteById(Mockito.eq(u
    }
}
```

Notice that we use the `Mockito.eq()` method to specify the argument value for the `deleteById()` method call. This ensures that the correct ID is passed to the method and avoids unexpected behavior in tests.

Not using Correct Annotation for Mock Objects

Here's an example of how to use the `@MockBean and @RunWith` annotations:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class UserServiceTest {

    @Autowired
    private UserService userService;

    @MockBean
    private UserRepository userRepository;

    @Test
    public void testGetAllUsers() {
        // arrange
        List<User> users = Arrays.asList(
                new User(1L, "John Doe"),
                new User(2L, "Jane Doe")
        );
        Mockito.when(userRepository.findAll()).thenReturn(users);

        // act
        List<User> result = userService.getAllUsers();
```

```
        // assert
        assertThat(result).isEqualTo(users);
    }
}
```

Notice that we use the `@RunWith` and `@SpringBootTest` annotations to configure the Spring Test framework for our unit tests. By using these annotations, we can ensure that the application context is loaded and that the dependencies are injected correctly.

Not using Correct Configuration for Tests

We want to use the correct configuration for our tests to ensure that the application context is loaded correctly and that the dependencies are injected as expected. Here's an example of how to use the `@ContextConfiguration` annotation:

```
@RunWith(MockitoJUnitRunner.class)
@ContextConfiguration(classes = {UserService.class, UserRepository.class})
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void testGetAllUsers() {
        // arrange
        List<User> users = Arrays.asList(
                new User(1L, "John Doe"),
                new User(2L, "Jane Doe")
        );
        Mockito.when(userRepository.findAll()).thenReturn(users);

        // act
```

```
        List<User> result = userService.getAllUsers();

        // assert
        assertThat(result).isEqualTo(users);
    }
}
```

Notice that we use the `@ContextConfiguration` annotation to specify the configuration for our tests. We pass in an array of classes that includes the `UserService` and `UserRepository` classes to ensure that they are loaded into the application context.

Not using Correct Method for Creating Mock Objects

We want to use the correct method for creating mock objects to ensure that the behavior of the dependencies can be controlled and that the tests are reliable. Here's an example of how to use the `Mockito.mock()` method:

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    private UserService userService;

    private UserRepository userRepository;

    @Before
    public void setUp() {
        userRepository = Mockito.mock(UserRepository.class);
        userService = new UserService(userRepository);
    }

    @Test
    public void testGetAllUsers() {
        // arrange
        List<User> users = Arrays.asList(
                new User(1L, "John Doe"),
                new User(2L, "Jane Doe")
        );
```

```
        Mockito.when(userRepository.findAll()).thenReturn(users);

        // act
        List<User> result = userService.getAllUsers();

        // assert
        assertThat(result).isEqualTo(users);
    }
}
```

Notice that we use the `Mockito.when()` method to specify the behavior of the mock object, which is to return a list of `User` objects when the `findAll()` method is called.

By using the `Mockito.mock()` method provided by Mockito, we can create mock objects for our unit tests that can be used to control the behavior of the dependencies.

Not using Correct Method for Stubbing Mock Objects

We want to use the correct method for stubbing mock objects to ensure that the behavior of the dependencies can be controlled and that the tests are reliable. Here's an example of how to use the `when().thenReturn()` method:

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void testGetAllUsers() {
        // arrange
```

```
        List<User> users = Arrays.asList(
                new User(1L, "John Doe"),
                new User(2L, "Jane Doe")
        );
        Mockito.when(userRepository.findAll()).thenReturn(users);

        // act
        List<User> result = userService.getAllUsers();

        // assert
        assertThat(result).isEqualTo(users);
    }
}
```

By using the `when().thenReturn()` method provided by Mockito, we can specify the behavior of our mock objects and ensure that the dependencies are controlled in our tests. This helps to ensure that our tests are reliable and that the behavior of our classes can be verified accurately.

Not using Correct Method for Verifying Interactions with Mock Objects

Mockito provides several methods for verifying interactions with mock objects, such as *Mockito.verify()*, *Mockito.verifyZeroInteractions()*, and *Mockito.verifyNoMoreInteractions()*. It is important to use the correct method for the desired behavior, as using the wrong method can lead to unreliable tests and unexpected behavior.

```
@Test
public void test() {
    MyObject myObject = new MyObject();
    myObject.setName("Name");
    Mockito.when(myRepository.findById(1)).thenReturn(Optional.of(myObject));

    MyObject result = myService.findById(1);

    Mockito.verify(myRepository).findById(1);
```

```
    Mockito.verifyNoMoreInteractions(myRepository);
    Assert.assertEquals("Name", result.getName());
}
```

Not using Correct Method for Verifying Order of Interactions with Mock Objects

Mockito provides a method called *Mockito.inOrder()* that can be used to verify the order of interactions with mock objects. It is important to use this method when verifying the order of interactions, as using other methods can lead to unreliable tests and unexpected behavior.

```
@Test
public void test() {
    MyObject myObject1 = new MyObject();
    myObject1.setName("Name 1");
    MyObject myObject2 = new MyObject();
    myObject2.setName("Name 2");
    InOrder inOrder = Mockito.inOrder(myRepository);

    Mockito.when(myRepository.findById(1)).thenReturn(Optional.of(myObject1));
    Mockito.when(myRepository.findById(2)).thenReturn(Optional.of(myObject2));

    MyObject result1 = myService.findById(1);
    MyObject result2 = myService.findById(2);

    inOrder.verify(myRepository).findById(1);
    inOrder.verify(myRepository).findById(2);
    Assert.assertEquals("Name 1", result1.getName());
    Assert.assertEquals("Name 2", result2.getName());
}
```

Conclusion

Mockito is a powerful and useful framework for testing Java applications. However, developers who are new to Mockito may make mistakes that can lead to unreliable tests and unexpected behavior in their applications.

Thanks for your attention! Happy Learning!

Mockito Spring Boot

Written by Bubu Tripathy

9.3K Followers · 246 Following

Senior Software Engineer I Microservices I Cloud Computing I DevOps || LinkedIn:https://www.linkedin.com/in/bubu-tripathy-8682187/

Follow

Responses (1)

Write a response

What are your thoughts?

Stepan Mozyra
Apr 15, 2023

Oververifying mock objects is also bad. In your example you show the test against the implementation, that should not be. You should test behaviour and contract, but not an implementation.

13 1 reply Reply

More from Bubu Tripathy

Bubu Tripathy

OAuth 2 using Spring Boot

Introduction

Mar 5, 2023 122 1

Bubu Tripathy

Best Practices: Entity Class Design with JPA and Spring Boot

In the world of modern software development, efficient design and implementation of entit...

Aug 10, 2023 821 11

Bubu Tripathy

JSON Serialization and Deserialization in Java

with FasterXML Jackson Annotations

Apr 8, 2023 81

Bubu Tripathy

Implementing Transactions in a Spring Boot Application

Introduction

Apr 26, 2023 329 2

See all from Bubu Tripathy

Recommended from Medium

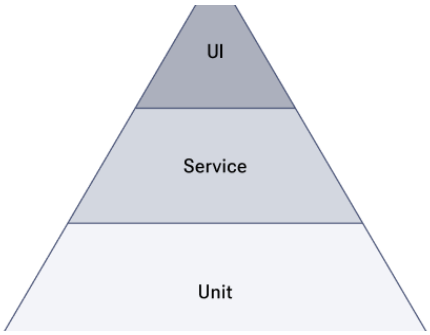
	@MockBean (Deprecated)	@MockitoBean (Re
	Before 3.2	3.2+
	Slower (requires Spring Context)	Faster (Mockito dir
	Global (affects multiple tests)	Local (scoped per t
	Uses Spring proxies	Directly integrates
	❌ No	✅ Yes

Ramesh Fadatare

Avoid @MockBean! Use @MockitoBean for Unit Testing in...

This is a member-only article. For non-members, read this article for free on my blo...

Mar 6 71 2

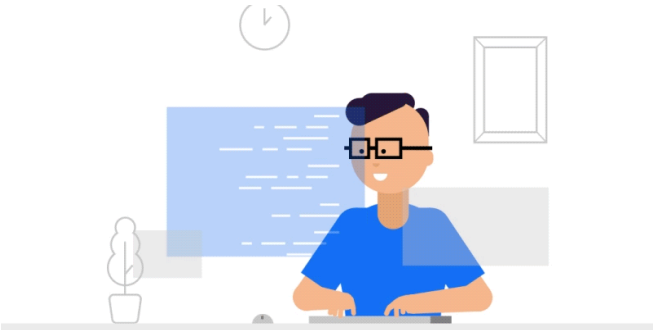


In Optivem by Valentina (Cupać) Jemuović

The Old Test Pyramid is Dead

You’ve all heard of the Test Pyramid—E2E Tests, Integration Tests, and Unit Tests. It’s...

Dec 26, 2024 4 1

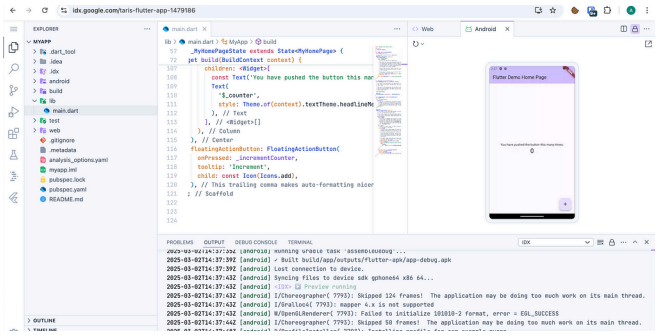


Himanshu

Stop Using Fat JARs in Spring Boot—They’re Killing Your Pipeline an...

If you’re still packaging your Spring Boot app as a fat JAR in 2025, you’re already falling...

Apr 18 200

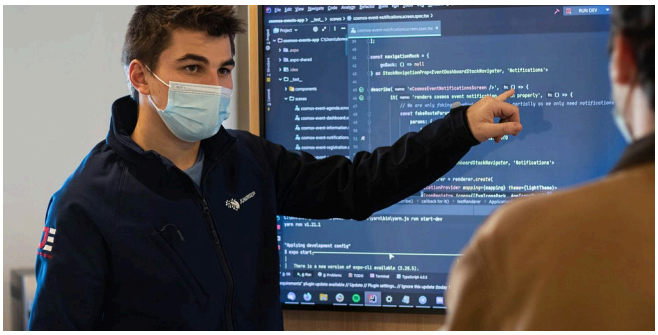


In Coding Beauty by Tari Ibaba

This new IDE from Google is an absolute game changer

This new IDE from Google is seriously revolutionary.

Mar 11 4.91K 285

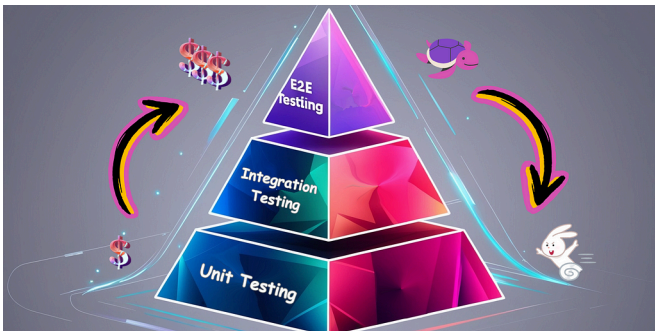


Vinod Pal

How I Review Code As a Senior Developer For Better Results

I have been doing code reviews for quite some time and have become better at it. Fro...

Jan 25 2.5K 66



JS In JavaScript in Plain English by Mohamed Mayallo

Unit, Integration, and E2E Testing in One Example Using Jest

Let's Implement Unit, Integration, and End-to-End (E2E) Testing with Jest, Puppeteer, and...

Nov 13, 2024 6

See more recommendations