**Spring Mvc unit test using Mockito + MockMVC**

* **MockMvc** is the main entry point for server-side Spring MVC test support. Perform a request and return a type that allows chaining further actions, such as asserting expectations, on the result.
* **@Mock** creating a mock. This can also be achieved by using org.mockito.mock(..) method.
* **@InjectMocks** injects mock or spy fields into tested objects automatically.
* **MockitoAnnotations.initMocks(this)** initializes fields annotated with Mockito annotations.
* **MockMvcBuilders.standaloneSetup(..).build()** builds a MockMvc instance by registering one or more @Controller instances and configuring Spring MVC infrastructure programmatically.

when(userService.exists(user)).thenReturn(**false**); for methods with RETURN

doNothing().when(userService).create(user); for VOID methods

verify(userService, times(1)).exists(user); VERIFY a certain pre-condition

perform() -🡪 HTTP Operations

andExpect() -🡪 Resultant HTTP status code , values

@Mock creates a mock. @InjectMocks creates an instance of the class and injects the mocks that are created with the @Mock (or @Spy) annotations into this instance.

**Mockito provides two different syntax for creating stubs like:**

* doReturn and thenReturn
* doNothing (no thenNothing)
* doThrow and thenThrow

The thenReturn way of stubbing is a type-safe way of setting up stubs. What this essentially means is that it does a compile-time check against the return types that you want to stub too.

**Let’s understand this with an Example:**

Assume a method **getItemDetails** on **mockedItemService**which return object of type**ItemSku.**So with **thenReturn,**you will not be able to return anything other than of type ItemSku but with doReturn you can set up the stub to return anything and the test will fail (or throw exception) during execution.

**// works**

*when*(mockedItemService.getItemDetails(123)).thenReturn(new ItemSku());

**// throws compile time exception**

*when*(mockedItemService.getItemDetails(123)).thenReturn(expectedPrice);

// with doReturn,  both the stub setup works as its not compile safe.

// here we are trying to return an object of type double which still works and does not throw any compile time warning.

*doReturn*(expectedPrice).when(mockedItemService.getItemDetails(123));

*doReturn*(new ItemSku()).when(mockedItemService.getItemDetails(123));

**b)** Another important difference between these 2 ways to the stub is for Mocked objects, apart from compile safety there is not much difference.

However for Spied objects, “thenReturn” kind of stub setup will not work, as it will result in calling the real method before the stubbed response is return as the call and not on a Mock, but on Spy which is wrapping a real object instance.

So suppose, there is a spy named **spiedObject and it has a method testMethod which returns an integer, then to setup a stub on this you will need to use doReturn instead of thenReturn.**

*doReturn*(10).when(spiedObject.testMethod());

Spy is a type of partial mock supported by Mockito.

**What this essentially means is a type of instance where:**

**a)** When no mock is setup , any interaction on spy results in calling the real methods. But it still allows you to verify the interactions with the spied object like – was a method actually called, how many times the method was called, what were the arguments using which the method was called etc.

**b)** It gives you the flexibility to set up partial mocks.

**For Example,** if you have an object with 2 methods – method1 and method2 and you want method1 to be called and method2 to be mocked. Spies provide this kind of setup.

So what is the difference between a mock and a stub in simple terms is – a mock is created from a type and not from an instance whereas a stub wraps an actual instance of the class object.

**static methods cannot be mocked using Mockito**

Static methods are associated with the class itself and not any particular instance of the class. This means that all instances/objects of the class use the same instance of the static method.

Static methods are more like procedural code and are mostly used in legacy systems in general.

Mock libraries typically create mocks by dynamical instance creation at runtime, either through interfaces or through inheritance and as the static method is not associated with any particular instance it’s not possible for mocking frameworks (like mockito, easy mock etc) to mock Static methods.

Frameworks like PowerMock which do have support for static methods perform bytecode manipulation at runtime in order to mock static methods.

Setting up a stub on a mocked object (or a spied instance) does not guarantee whether the stubbed setup was even invoked.

“verification” matchers, give a facility to validate whether the stub that was set up was actually invoked or not, how many times was the call made, what arguments were the stubbed method called with etc.

In essence, it allows to verify the test setup and expected outcome in a more robust manner.

**Q #6) What is good testable code?**

**Answer:**

**Few points about testable code (meaning which could be easily unit tested) include:**

* **Reduced no of dependencies or tight coupling –** **Example:** Dependencies should be injected rather than instantiated directly.
* **Code that adheres to SRP (Single Responsibility Principle)** – This essentially means that the class should not have multiple reasons to change. Adherence to SRP avoids classes creating dependency on itself and keeps the code cohesive and clean.
* **Less / Minimal usage of static methods and final classes –** These generally indicate code smells and were mostly associated with the legacy code.

**What are the limitations of Mockito?**

**Answer:**

Mockito is a framework of choice for most of the java based projects. It is easy to implement, read and understand.

**Some of the drawbacks or limitations in terms of functionality are:**

* Its inability to mock static methods.
* Constructors, private methods and final classes cannot be mocked.

**Which frameworks can support mocking Private and Static methods?**

**Answer:**

Frameworks like PowerMockito (which are essentially extensions of Mockito framework), JMockit etc. do provide means to mock private and static methods.

**a)** **Using comma separated:** This works with thenReturn.

For Example taking the above code sample, let us try to setup consecutive stub for method – getGrade which will return different values depending on the sequence of iterations:

when(mockDatabaseImpl.getGrade(anyInt())).thenReturn("A","B", "C");

This means that when getGrade methods get called in the method under test, the first invocation will return “A”, the second invocation will return “B” and so on.

**b)** **Consecutive thenReturn:** This is an approach that is chained with thenReturn statements. Applying chained calls to the same example will look as shown below.

when(mockDatabaseImpl.getGrade(anyInt())).thenReturn("A").thenReturn("B").thenReturn("C");

**c) Consecutive doReturn:** The last approach is using doReturn in the chained format as above.

doReturn("A").doReturn("B").doReturn("C").when(mockDatabaseImpl).getGrade(anyInt())

**What are the different types of mocking frameworks and how do they work?**

**Answer:**

Types of the Mocking framework and how do they work are explained below.

**There are broadly 2 categories of mocking frameworks:**

1. **Proxy-based** – **E.g.** Mockito, EasyMock etc.
2. **Bytecode based** – **E.g.**PowerMock, JMockit etc.

**Let’s compare both these frameworks on different parameters.**

|  | **Proxy based** | **Bytecode based** |
| --- | --- | --- |
| Simplicitly | More simple and easy to use | Might involve complex mock setup logic |
| Mode of creation | A proxy or fake object which does not actually require instance of class / interface is created | It essentially involves creating objects and at runtime manipulates the instances for the mocked/stubbed behavior |
| Functionality | Mocking classes and interfaces | In addition to classes and interfaces, allows mocking static methods, final classes etc |
| Java dependency | Not very tightly coupled to java versions | Since these frameworks involve bytecode manipulation they are tightly coupled and might not be backward/forward compatible across java versions. |
| Examples | Mockito, EasyMock etc. | PowerMock, JMockit etc. |

* **@Mock** creates a mock implementation for the classes you need.
* **@InjectMock** creates an instance of the class and injects the mocks that are marked with the annotations **@Mock** into it.

We can use the @MockBean to add mock objects to the Spring application context. The mock will replace any existing bean of the same type in the application context.

Mockito.mock() & @Mock serve same purpose .

**Powermock Framework**

**It can be used to mock private , static , final methods .**

**@RunWith(PowerMockRunner.class)**

**PowerMockito.mockStatic(ClassName.class);**

List mockList = Mockito.mock(List.class);

ArgumentCaptor<String> arg = ArgumentCaptor.forClass(String.class);

@Captor

**Mockito doesn't support injecting mocks into spies**