Mockito has become a very popular and well-documented open source mock testing library.

Spring is obviously the application framework of choice these days. Most of the time when writing unit tests for your Service layer, you will either be writing a Mockito test or a Spring test. Mockito tests are used to isolate testing to only the method being tested. Mock objects stand-in for and simulate the behavior of real objects and thus allow the tester to control the behavior of any dependent objects.

On the other hand, Spring tests are commonly used when you want to verify that Spring dependency injection is working and wire-up certain classes that have been configured for testing. A common usage of a Spring test is for testing data access objects (DAO) that access an in-memory database for testing instead of the actual database.

Here is an example JUnit4 test case that uses Mockito as the mocking framework:

|  |
| --- |
| @RunWith(MockitoJUnitRunner.class)  public class AccountServiceTest {       @Mock      private NotificationService notificationService;       @Mock      private AccountDAO accountDAO;       @InjectMocks      private AccountServiceImpl accountService = new AccountServiceImpl();        @Test      public void createNewAccount() {           // Expected objects          String name = "Test Account";          Account accountToSave = new Account(name);          long accountId = 12345;          Account persistedAccount = new Account(name);          persistedAccount.setId(accountId);            // Mockito expectations       when(accountDAO.save(any(Account.class))).thenReturn(persistedAccount);       doNothing().when(notificationService).notifyOfNewAccount(accountId);            // Execute the method being tested          Account newAccount = accountService.createNewAccount(name);            // Validation          assertNotNull(newAccount);          assertEquals(accountId, newAccount.getId());          assertEquals(name, newAccount.getName());            verify(notificationService).notifyOfNewAccount(accountId);          verify(accountDAO).save(accountToSave);      }  } |

You’ll notice the use of annotations throughout the test. The @RunWith annotation causes the MockitoJunitRunner to execute the test. This runner extends a JUnit runner and takes care of creating any mocks and setting them on the class under test. In the test above, the NotificationService and AccountDAO are mocked and injected into the AccountServiceImpl.

Below is an example Spring test case:

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration({ "classpath:test-context.xml" })  @TransactionConfiguration(transactionManager = "transactionManager", defaultRollback = true)  @Transactional  public class AccountDAOTest {        @Autowired      private AccountDAO accountDAO;        @PersistenceContext      private EntityManager entityManager;        @Test      @Transactional      public void save() {             Account newAccount = new Account("Test Account");             // Execute the method being tested           accountDAO.save(newAccount);             // Validation           assertNotNull(newAccount.getId());           Account reloaded = flushAndGet(newAccount.getId());           assertEquals(newAccount.getId(), reloaded.getId());           assertEquals(newAccount.getName(), reloaded.getName());      }        private Account flushAndGet(long id) {           entityManager.flush();           return entityManager.find(Account.class, id);      }  } |

In this example Spring test case, the SpringJUnit4ClassRunner initializes the Spring application framework and configures and loads the objects in the test-context.xml file. This configuration initializes and configures an in-memory database (such as the H2 Database Engine). The AccountDAO is autowired into the test and its save() method is tested. By using a Spring test instead of a Mockito test, the actual Hibernate mappings and underlying database queries can be tested.

Sometimes, however, you may want to be able to mock a dependency or set of dependencies and also use Spring to inject the remaining dependencies. This is a scenario that is probably not real common and typically not the best practice for most tests since it introduces additional complexity into the test case but there are some legitimate reasons for wanting to set up a test this way. Some scenarios where you may want to use both Mockito and Spring are as follows:

* You are testing a complex method that has dependencies on both a service that accesses the database and a service that makes a remote procedure call to another server. You want to use Mockito to mock the service that makes the remote procedure call but you want to use Spring to wire all the other dependencies.
* You are testing exception handling and it’s difficult to reproduce a specific exception from the real object. It’s easy to test the exception handling behavior of an object using Mockito because you can mock an object and declare that a method throws a specific exception whenever it’s called. You may want to autowire all other dependencies using Spring.
* A system may be under development, and the interfaces but not the implementations may exist for a number of dependencies. You may want to autowire all other (implemented) dependencies using Spring but mock the unimplemented implementations.
* You may want to mock a service that is slow to execute in order to speed up test runs but autowire all other dependencies using Spring.

Continuing on with the AccountService and AccountDAO examples above, here is a hypothetical example: Let’s say that we want to test the delete() method on AccountService. An account can only be deleted if the logged in user has the correct permissions. Spring Security is used to implement permissions, so we want to use a Spring test so that we can verify that the Spring Security configuration is correct.

However, if an account is deleted, then as part of the service layer logic a REST call is made to another system for auditing purposes. That third party system is difficult to set up and run within a unit test. Besides, the third party system shouldn’t be set up anyway because this is a unit test and we just want to test the behavior of the AccountService delete() method and not the third party system. In other words, it’s good enough to test that the notify() method is called – perfect for a mock.

But how can we do this, use both Spring’s testing framework and Mockito? We can use a MockitoJUnitRunner or a SpringJUnit4ClassRunner, but not both. The first time I ran across a case like this I was stumped. I went down the path of creating my own mock and setting this mock up as a Spring bean in the test application context so that it could be autowired into my Spring test. It turns out that there is a much easier way.

MockitoJUnitRunner calls MockitoAnnotations.initMocks() to do its setup work. So if we just use the initMocks() method instead of the MockitoJUnitRunner and keep SpringJUnit4ClassRunner, it all works!  Here’s some example code:

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration({ "classpath:test-context.xml" })  @TransactionConfiguration(transactionManager = "transactionManager", defaultRollback = true)  @Transactional  public class AccountServiceSpringAndMockitoTest {        @Mock      private AuditService mockAuditService;        @InjectMocks      @Autowired      private AccountService accountService;        @Before      public void setup() {          MockitoAnnotations.initMocks(this);      }        @Test      @Transactional      public void deleteWithPermission() {            // Set up Spring Security token for testing          SecuredUser user = new SecuredUser();          user.setUsername("test1");          TestingAuthenticationToken token = new TestingAuthenticationToken(user, null, "accountFullAccess");          SecurityContextHolder.getContext().setAuthentication(token);            // Create account to be deleted          Account accountToBeDeleted = accountService.createNewAccount("Test Account");          long accountId = accountToBeDeleted.getId();            // Mockito expectations          doNothing().when(mockAuditService).notifyDelete(accountId);            // Execute the method being tested          accountService.delete(accountToBeDeleted);            // Validation          assertNull(accountService.get(accountId));          verify(mockAuditService).notifyDelete(accountId);      }  } |

Notice the setup() method that is run before each test. In it, MockitoAnnotations.initMocks() is called. By the time initMocks() is called, Spring has already injected all of the other dependencies. The AccountService implementation is injected by Spring and then the mockAuditService is set on this implementation by Mockito.  Pretty cool!

I am hopeful that this article describes not only how to use Mockito and Spring together, but also provides some good examples about why or when you would want to use them together in a unit test. The references section below provides a couple of links that give some other good examples of using Mockito and Spring together. Happy testing!

**References**

* [**http://stackoverflow.com/questions/10906945/mockito-junit-and-spring**](http://stackoverflow.com/questions/10906945/mockito-junit-and-spring)
* [**http://markchensblog.blogspot.com/2013/02/use-spring-mvc-test-framework-and.html**](http://markchensblog.blogspot.com/2013/02/use-spring-mvc-test-framework-and.html)
* [**https://en.wikipedia.org/wiki/Mock\_object**](https://en.wikipedia.org/wiki/Mock_object)