**Junit 5**

**What is test scope for Maven or Gradle Junit dependencies?**

Test scope indicates that Junit dependencies will not be part of final build (deliverables). It will be only used during testing of an application.

**Why a test class is added in the same package (/src/test/java folder) as your main class (/src/main/java)?**

Test class is added in same package as that of main class so that it can access protected methods from main class.

**What is typical test scenario in a Junit test?**

1. Create an instance of a class under test
2. Setup inputs
3. Execute the code (method) you want to test
4. Verify the result is what you expect (assertion)

**Why a method should be static for @BeforeAll and @AfterAll annotations?**

@BeforeAll and @AfterAll annotated methods are executed even before a test class instance is created. Methods annotated with @BeforeAll and @AfterAll annotations are restricted to be static. This is due to the fact that static method do not have dependency on an instance and it can be executed even before a test class instance is created.

In Junit5, by default a new test class instance is created for each test. However, when this behavior is changed to create only one instance per class ([@TestInstance(TestInstance.Lifecycle.PER\_CLASS](mailto:@TestInstance(TestInstance.Lifecycle.PER_CLASS))) then static method restriction for @BeforeAll do not apply. Here, since there is only one instance of a class, Junit takes care to execute @BeforeAll annotated method on same instance before any other method is executed.

**What are differences between Junit 4 and Junit 5?**

In Junit 5, the @Test annotation doesn’t accept any parameters. In Jnuit 4, it accepts.

In Junit 5, neither test classes nor test methods need to be public. In Jnuit 4, it needs to be public.

***JUnit 5 vs JUnit 4***

1) ***Annotations***

Same as in JUnit 4 - @Test

Renamed

* @BeforeAll instead of @BeforeClass
* @AfterAll instead of @AfterClass
* @BeforeEach instead of @Before
* @AfterEach instead of @After
* @Disable instead of @Ignore
* @Tag instead of @Category

New

* @DisplayName
* @TestFactory for dynamic tests
* @Nested for nested tests
* @RepeatedTest to executed tests multiple times
* @EnabledOnOs
* @DisabledOnOs
* @EnabledOnJre

***2) Assertions***

Same as JUnit 4

* AssertTrue
* AssertSame
* AssertNull
* AssertNotSame
* AssertNotEquals
* AssertNotNull
* AssertFalse
* AssertEquals
* assertArrayEquals

New

* assertThrows
* AssertTimeout

***3) @RunWith vs @ExtendWith***

***@RunWith*** - When a class is annotated with @RunWith or extends a class annotated with @RunWith, JUnit will invoke the class it references to run the tests in that class instead of the runner built into JUnit.

***@ExtendWith -*** @ExtendWith is a [repeatable](https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Repeatable.html?is-external=true) annotation that is used to register [extensions](https://junit.org/junit5/docs/5.0.3/api/org/junit/jupiter/api/extension/Extension.html) for the annotated test class or a test method.

To use @Mock and @Spy annotations, we need to add @RunWith with Junit4 and @ExtendWith for Junit5.

* @RunWith(MockitoJUnitRunner.class) - Junit4 way
* @ExtendWith(MockitoExtension.class) - Junit5 way

In JUnit 4, it’s necessary to use the @RunWith annotation to change the test’s execution. There are some limitations using it. It’s not possible to add more than one runner in the same class and the annotation is used only at class level. JUnit 5 has created a concept called extension model, adding an annotation @ExtendWith, which replaces @RunWith. Using the extension model, you can add functionalities during the test execution. It’s possible to compose extensions, add more than one at the same time, use at class or method level and create your own extensions.

**What are different Argument Sources for @ParameterizedTest annotation?**

* @ValueSource
* @NullSource
* @EmptySource
* @NullAndEmptySource
* @EnumSource
* @CsvSource
* @CsvFileSource
* @MethodSource
* implementation of an interface called ArgumentsProvider

**Explain Junit 5 framework architecture?**

JUnit 5 = Platform + Jupiter + Vintage

1. Platform(engine) = Building blocks - Library for running tests, provides way to call testRunner and execution context- These are core features which developers do not interact with directly.
2. Jupiter = New Programming Model (JUnit Jupiter API for writing tests - Predefined Assertions, Annotations) + Extension Model (JUnit Jupiter API for writing extensions - Provides the way to extend features provided by Junit5 ) + Functions - Functional interfaces used within JUnit Jupiter like Executable for functional programming
3. Vintage - Support for old Junit Versions (3 and 4) - Execute old version test cases with Junit5. To use JUnit 5 and run tests created in JUnit 3 or JUnit 4, you need to add the JUnit Vintage dependency.

**What are JUnit Best Practices?**

1) Test case should be readable - Readability of a test can be improved by using meaningful variable and method names. One should be able to tell purpose of test just in one look.

2) Test should be executed FAST.

3) Test should be isolated from external interfaces or databases. We can achieve this using mocking frameworks like Mockito where test do not have to depend on external interfaces. Such a test fails only when there is issue with code being tested and not due to failure in db or an external interface.

4) Design test cases to be run very often i.e. on every code commit (Continuous Integration).

**What is MavenSurefire Plugin?**

There is equivalent Gradleplugin as well for Surefire.

The Surefire Plugin is used during the test phase of the build lifecycle to execute the unit tests of an application. It generates reports in two different file formats:

* Plain text files (\*.txt)
* XML files (\*.xml)

By default, these files are generated in ${basedir}/target/surefire-reports/TEST-\*.xml.

**Explain test life cycle in Junit5?**

In Junit5 a new test class instance is created for each test.

Below JUnit lifecycle hook annotations are available --

@BeforeAll - Executed once before all the test cases - like initialize DB connection

@AfterAll - Executed once after all the test cases - like close DB connection

@BeforeEach - Executed before every test - Setting up data in every test

@AfterEach - Executed after every test - Data clean up after every test

In Junit, it is a bad practice for test cases to share a member variable as it increases coupling/dependency between tests which is not an ideal scenario. Moreover, tests are executed in random order by default and hence coupled test cases with shared member variable can behave differently when ordering is changed.

**What is @Order annotation in Junit?**

Ideally Junit test cases should be decoupled and run independent of each other, but in some cases you may need to order them.

By default, JUnit runs tests using a deterministic, but unpredictable order. In most cases, that behavior is perfectly fine and acceptable; but there’re cases when we need to enforce a specific ordering.

In JUnit 5, we can use @TestMethodOrder to control the execution order of tests. We can use our own MethodOrderer, or we can select one of three built-in orderers:

***1) Using the @Order Annotation***

We can use the @Order annotation to enforce tests to run in a specific order.

In the following example, the methods will run in this order — firstTest(), then secondTest(), and finally, thirdTest():

@TestMethodOrder(OrderAnnotation.class)

public class OrderAnnotationUnitTest {

private static StringBuilder output = new StringBuilder("");

@Test

@Order(1)

public void firstTest() {

output.append("a");

}

@Test

@Order(2)

public void secondTest() {

output.append("b");

}

@Test

@Order(3)

public void thirdTest() {

output.append("c");

}

}

**2) Alphanumeric Order**

We can also run tests based on their names in alphanumeric order:

@TestMethodOrder(Alphanumeric.class)

public class AlphanumericOrderUnitTest {

private static StringBuilder output = new StringBuilder("");

@Test

public void myATest() {

output.append("A");

}

@Test

public void myBTest() {

output.append("B");

}

@Test

public void myaTest() {

output.append("a");

}

@AfterAll

public static void assertOutput() {

assertEquals(output.toString(), "ABa");

}

}

Note that alphanumeric order is case sensitive, so uppercase characters come first then lowercase ones.

The tests will run in this order: myATest(), myBTest() and finally myaTest().

***3) Custom order***

we can use our own custom order by implementing the MethodOrderer interface.

**What is @TestInstance annotation?**

In Junit5, by default a new test class instance is created for each test. This default behavior can be changed by using @TestInstance annotation on a test class.

* @TestInstance(TestInstance.Lifecycle.PER\_METHOD) // Default
* @TestInstance(TestInstance.Lifecycle.PER\_CLASS) // Change behavior to create one instance per class

**How tests are executed conditionally in Junit?**

* @EnabledOnOS(OS.Linux)
* @EnabledOnJre(JRE.JAVA\_11)
* @EnabledIf
* @EnabledIfSystemProperty
* @EnabledIfEnvironmentVariable

There are other ways to handle external factors - like using assumptions which prevent test execution if certain condition is not fulfilled for test to run. When assumption fails, TestAbortedException is thrown and further execution of test is skipped.

***How Junit5 uses Dependency Injection?***

Junit5 provides DI for below classes as a method argument -

RepetitionInfo, TestInfo and TestReporter are Java interfaces and not classes, but we should not worry about underlying implementation. We can use these interfaces directly.

***RepetitionInfo:*** RepetitionInfo instance is injected when @RepeatedTest annotation is used. RepetitionInfo is used to inject information about the current repetition of a repeated test into @RepeatedTest, @BeforeEach, and @AfterEach methods.

If a method parameter is of type RepetitionInfo, JUnit will supply an instance of RepetitionInfo corresponding to the current repeated test as the value for the parameter.

RepetitionInfo cannot be injected into a @BeforeEach or @AfterEach method if the corresponding test method is not a @RepeatedTest. Any attempt to do so will result in a ParameterResolutionException.

***TestInfo:*** TestInfo is used to inject information about the current test or container into to @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, @BeforeEach, @AfterEach, @BeforeAll, and @AfterAll methods. TestInfo gives meta-information about test.

If a method parameter is of type TestInfo, JUnit will supply an instance of TestInfo corresponding to the current test or container as the value for the parameter.

***TestReporter:*** Parameters of type TestReporter can be injected into @BeforeEach and @AfterEach lifecycle methods as well as methods annotated with @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, etc. Within such methods the injected TestReporter can be used to publish report entries for the current container or test to the reporting infrastructure. TestReporter lets you write to the Junit output (Eclipse output or Maven/Gradleoutput).

**Mockito with Junit 5**

**What is Mockito?**

Mockito is a mocking framework for Java. Mockito allows convenient creation of substitutes of real objects for testing purposes. Enjoy clean tests with mock objects, improved TDD experience and beautiful mocking API.

Spring-boot-starter-test is a starter for testing Spring Boot applications with libraries including JUnit, Hamcrest, assertj and Mockito.

**What is difference between Mock and Stub Object?**

It is important to understand the difference between a mock and Stub object. A Stub object is an actual instance of a class and any method invoked using object reference will execute the actual method body defined in the class file.

A mock object is an interface to hide a dependency with cannot be tested in test environment e.g. database, network locations etc. A method invoked using mocked reference does not execute method body defined in class file, rather the method behavior is configured using when-thenReturn methods combinations.

**What is @Mock annotation in Mockito?**

The @Mock annotation is alternative to Mockito.mock(classToMock). They both achieve the same result. Using @Mock is usually considered “cleaner“, as we don’t fill up the tests with boilerplate assignments that all look the same.

***Using the @Mock annotation –***

* allows shorthand creation of objects required for testing.
* minimizes repetitive mock creation code.
* makes the test class more readable.
* makes the verification error easier to read because field name is used to identify the mock.

**What is @InjectMocks annotation in Mockito?**

In Mockito, we need to create the object of class to be tested and then insert it’s dependencies (mocked) to completely test the behavior. To do this, we use @InjectMocks annotation.

@InjectMocks marks a field on which injection should be performed. Mockito will try to inject mocks only either by constructor injection, setter injection, or property injection – in this order.

Use @InjectMocks to inject both @Spy and @Mock instances.

**How to enable Mockito annotations ?**

In order for Mockito annotations to be enabled, we’ll need to annotate the JUnit test with a runner – MockitoJUnitRunner as in the following example:

@RunWith(MockitoJUnitRunner.class)

public class MockitoAnnotationTest {

...

}

Alternatively, we can enable these annotations programmatically as well, by invoking MockitoAnnotations.initMocks() as in the following example:

@Before

public void init() {

MockitoAnnotations.initMocks(this);

}

**What are the limitations of Mockito?**

***Mockito 2.x specific limitations***

* Requires Java 6+
* Cannot mock static methods
* Cannot mock constructors
* Cannot mock equals(), hashCode(). Firstly, you should not mock those methods. Secondly, Mockito defines and depends upon a specific implementation of these methods. Redefining them might break Mockito.
* Mocking is only possible on VMs that are supported by Objenesis. Don't worry, most VMs should work just fine.
* Spying on real methods where real implementation references outer Class via OuterClass. This is impossible. Don't worry, this is extremely rare case.

**Do you mock classes & interfaces?**

Yes, the api is the same for mocking classes or interfaces.

**What are Mock Object’s Default return Values when mock object is not stubbed?**

* numeric - 0
* boolean - false
* Object - null
* Collection - Empty Collection

**Can I mock static methods?**

No. Mockito prefers object orientation and dependency injection over static, procedural code that is hard to understand & change. If you deal with scary legacy code you can use JMockit or Powermock to mock static methods.

**Can I mock private methods?**

No. From the standpoint of testing... private methods don't exist.

**Can I verify toString()?**

No. You can stub it, though. Verification of toString() is not implemented mainly because:

When debugging, IDE calls toString() on objects to print local variables and their content, etc. After debugging, the verification of toString() will most likely fail.

toString() is used for logging or during string concatenation. Those invocations are usually irrelevant but they will change the outcome of verification.

**What are unfinished verification/stubbing errors?**

Mockito validates if you use it correctly all the time. Examples of incorrect use:

***//Oups, someone forgot thenReturn() part:***

when(mock.get());

***//Oups, someone put the verified method call inside verify() where it should be outside:***

verify(mock.execute());

***//Oups, someone has used EasyMock for too long and forgot to specify the method to verify:***

verify(mock);

Mockito throws exceptions if you misuse it so that you will know if your tests are written correctly. The only problem is that Mockito does the validation next time you use the framework. Therefore sometimes the exception is thrown in the next test and you have to manually find the previous test that was not written correctly.

**Can I thenReturn() an inlined mock() ?**

Unfortunately you cannot do this:

when(m.foo()).thenReturn(mock(Foo.class));

// ^

The reason is that detecting unfinished stubbing wouldn't work if we allow above construct. We consider this as a 'trade off' of framework validation. However, you can slightly change the code to make it working:

//extract local variable and start smiling:

Foo foo = mock(Foo.class);

when(m.foo()).thenReturn(foo);

**Can I stub chained getters?**

when(mock.getA().getB()).thenReturn(...);

This sort of stubbing, e.g. mock to return mock, to return mock, etc. should be used very sporadically, ideally never. It clearly points out violation of the Law of Demeter. You don't want to mess with Demeter. Since you have been warned check out Mockito deep stubs.

**Do you use Spring in a unit test?**

1) Spring framework has great support for testing.

2) The testable POJOs should be instantiated without any container.

3) We don’t necessarily need Spring in a unit test.

4) However Spring IOC makes both unit and integration testing easier.

**What type of tests typically use Spring?**

Spring provides mock objects and testing support classes for Unit Testing --

1) Tests one unit of functionality

2) Keeps dependencies minimal

3) Isolate from the environment (including Spring)

Spring provides first-class support for integration testing --

1) Tests the interaction of multiple units working together

2) Integrates infrastructure like database

**How is @ContextConfiguration used?**

In spring-test library, @ContextConfiguration is a class-level annotation, that defines the location configuration file, which will be loaded for building up the application context for integration tests.

if @ContextConfiguration is used without any attributes defined, the default behavior of spring is to search for a file named {testClassName}-context.xml in the same location as the test class and load bean definitions from there if found.

***@RunWith(SpringJUnit4ClassRunner.class)***

***@ContextConfiguration(classes={KindergartenConfig.class, HighschoolConfig.class}) @ActiveProfiles("kindergarten")***

***public class ProfilesJavaConfigTest {***

***@Autowired***

***FoodProviderService foodProviderService;***

***}***

Spring Boot provides a @SpringBootTest annotation, which can be used as an alternative to the standard spring-test @ContextConfiguration annotation when you need Spring Boot features. The annotation works by creating the ApplicationContext used in your tests through SpringApplication.

***@RunWith(SpringRunner.class)***

***@SpringBootTest(properties = "spring.main.web-application-type=reactive")***

***public class MyWebFluxTests { }***

**How can you create a shared application context in a JUnit integration test?**

Spring’s integration testing support has the following primary goals:

To manage Spring IoC container caching between tests. By default, once loaded, the configured ApplicationContext is reused for each test.

To provide Dependency Injection of test fixture instances.

To provide transaction management appropriate to integration testing.

To supply Spring-specific base classes that assist developers in writing integration tests.

To access the Context with the TestContext Framework in JUnit, two options to access the managed application context.

The first option is by implementing the ApplicationContextAware interface or using @Autowired on a field of the ApplicationContext type. You can specify this in the @RunWith annotation at the class level.

***@RunWith(SpringRunner.class)***

***@ContextConfiguration(classes = BankConfiguration.class)***

***public class AccountServiceJUnit4ContextTests implements ApplicationContextAware { }***

The SpringRunner class, which is an alias for SpringJUnit4ClassRunner, is a custom JUnit runner helping to load the Spring ApplicationContext by using @ContextConfiguration(classes=AppConfig.class). In JUnit, you can simply run your test with the test runner SpringRunner to have a test context manager integrated.

By default, the application context will be cached and reused for each test method, but if you want it to be reloaded after a particular test method, you can annotate the test method with the @DirtiesContext annotation so that the application context will be reloaded for the next test method.

Inject Test Fixtures with the TestContext Framework in JUnit. In JUnit, you can specify SpringRunner as your test runner without extending a support class.

***@RunWith(SpringRunner.class)***

***@ContextConfiguration(classes = BankConfiguration.class)***

***public class AccountServiceJUnit4ContextTests { }***

The second option to access the managed application context is by extending the TestContext support class specific to JUnit: AbstractJUnit4SpringContextTests.

Note that if you extend this support class, you don’t need to specify SpringRunner in the @RunWith annotation because this annotation is inherited from the parent.

***@ContextConfiguration(classes = BankConfiguration.class)***

***public class AccountServiceJUnit4ContextTests extends AbstractJUnit4SpringContextTests { }***

**When and where do you use @Transactional in testing?**

At method level: the annotated test method(s) will run, each in its own transaction. By default, automatically rolled back after completion of the test.

You can alter this behavior by disabling the defaultRollback attribute of @TransactionConfiguration.

At class level: each test method within that class hierarchy runs within a transaction.

You can override this class-level rollback behavior at the method level with the @Rollback annotation, which requires a Boolean value, @Rollback(false), This is equivalent to another annotation introduced in Spring - @Commit.

**How does Spring Boot simplify writing tests?**

spring-boot-starter-test pulls in the following all within test scope:

* JUnit: De-facto standard for testing Java apps
* JSON Path: XPath for JSON
* AssertJ: Fluent assertion library
* Mockito: Java mocking library
* Hamcrest: Library of matcher objects
* JSONassert: Assertion library for JSON

***Spring Test and Spring Boot Test***: Test libraries provided by the Spring Framework and Spring Boot.

testCompile('org.springframework.boot:spring-boot-starter-test')

**Testing with Spring Boot, Mockito and Junit 5**

**How @WebMvcTest annotation works ?**

Annotation that can be used in combination with @RunWith(SpringRunner.class) for a typical Spring MVC test. Can be used when a test focuses only on Spring MVC components.

Using this annotation will disable full auto-configuration and instead apply only configuration relevant to MVC tests (i.e. @Controller, @ControllerAdvice, @JsonComponent, Converter/GenericConverter, Filter, WebMvcConfigurer and HandlerMethodArgumentResolver beans but not @Component, @Service or @Repository beans).

By default, tests annotated with @WebMvcTest will also auto-configure Spring Security and MockMvc (include support for HtmlUnit WebClient and Selenium WebDriver). For more fine-grained control of MockMVC the @AutoConfigureMockMvc annotation can be used.

Typically @WebMvcTest is used in combination with @MockBean or @Import to create any collaborators required by your @Controller beans.

If you are looking to load your full application configuration and use MockMVC, you should consider @SpringBootTest combined with @AutoConfigureMockMvc rather than this annotation.

**How @DataJpaTest annotation works ?**

Annotation that can be used in combination with @RunWith(SpringRunner.class) for a typical JPA test. Can be used when a test focuses only on JPA components.

Using this annotation will disable full auto-configuration and instead apply only configuration relevant to JPA tests.

By default, tests annotated with @DataJpaTest are transactional and roll back at the end of each test. They also use an embedded in-memory database (replacing any explicit or usually auto-configured DataSource). The @AutoConfigureTestDatabase annotation can be used to override these settings.

If you are looking to load your full application configuration, but use an embedded database, you should consider @SpringBootTest combined with @AutoConfigureTestDatabase rather than this annotation.

**How @SpringBootTest annotation works ?**

@SpringBootTest is a very powerful annotation which works by creating the ApplicationContext used in your tests through SpringApplication.

Annotation that can be specified on a test class that runs Spring Boot based tests. Provides the following features over and above the regular Spring TestContext Framework:

Uses SpringBootContextLoader as the default ContextLoader when no specific @ContextConfiguration(loader=...) is defined.

Automatically searches for a @SpringBootConfiguration when nested @Configuration is not used, and no explicit classes are specified.

Allows custom Environment properties to be defined using the properties attribute.

Provides support for different webEnvironment modes, including the ability to start a fully running web server listening on a defined or random port. @SpringBootTest with webEnvironment = WebEnvironment.RANDOM\_PORT will start the management server on a separate random port if your application uses a different port for the management server.

Registers a TestRestTemplate and/or WebTestClient bean for use in web tests that are using a fully running web server.

ApplicationContext is initialized by @SpringBootTest as follows -

1. Searches for package with class annotated with @SpringBootApplication
2. Once it finds class annotated with @SpringBootApplication, it launches entire Spring boot application context
3. All the Components, controllers, services, repositories, queries in data.sql are initiated
4. @SpringBootTest uses in-memory database for testing. Creates tables using Entity classes and loads data using data.sql and run tests against it.
5. Now our test cases are ready with complete application context and can launch integration tests involving all the application components.

**How WebEnvironment attribute of @SpringBootTest annotation works ?**

By default, @SpringBootTest will not start a server. You can use the webEnvironment attribute of @SpringBootTest to further refine how your tests run:

* ***MOCK(Default)*** : Loads a web ApplicationContext and provides a mock web environment. Embedded servers are not started when using this annotation. If a web environment is not available on your classpath, this mode transparently falls back to creating a regular non-web ApplicationContext. It can be used in conjunction with @AutoConfigureMockMvc or @AutoConfigureWebTestClient for mock-based testing of your web application.
* ***RANDOM\_PORT:*** Loads a WebServerApplicationContext and provides a real web environment. Embedded servers are started and listen on a random port.
* ***DEFINED\_PORT:*** Loads a WebServerApplicationContext and provides a real web environment. Embedded servers are started and listen on a defined port (from your application.properties) or on the default port of 8080.
* ***NONE:*** Loads an ApplicationContext by using SpringApplication but does not provide any web environment (mock or otherwise).

**How @TestPropertySource annotation works?**

Typically test configuration will be loaded from application.properties in folder /src/main/resources. If application.properties is there in folder /test/main/resources then that will take precedence over main application.properties.

@TestPropertySource is a class-level annotation. When a configuration defined using @TestPropertySource, it applies to tests defined in that specific class. Even if it is used in main application test class where @SpringBootTest is used, it still applies only to that specific class and not others.

With @TestPropertySource, we can define configuration sources that have higher precedence than any other source used in the project. Thus, test property sources can be used to selectively override properties defined in system and application property sources. Furthermore, inlined properties() have higher precedence than properties loaded from resource locations().

If @TestPropertySource is declared as an empty annotation (i.e., without explicit values for locations() or properties()), an attempt will be made to detect a default properties file relative to the class that declared the annotation. For example, if the annotated test class is com.example.MyTest, the corresponding default properties file is "classpath:com/example/MyTest.properties". If the default cannot be detected, an IllegalStateException will be thrown.

Typically, whenever we use @TestPropertySource annotation, we will also include the @ContextConfiguration so as to load and configure the ApplicationContext for the scenario.

Additionally, we can change the default configuration file location, or add extra properties that will have even higher precedence:

***@TestPropertySource(locations = "/other-location.properties",***

***properties = "baeldung.testpropertysource.one=other-property-value")***

Finally, we can specify whether we want to inherit locations and properties values from superclasses or not. Hence, we can toggle the inheritLocations and inheritProperties attributes, which are true by default.

**Which libraries are included by default in spring-boot-starter-test dependency?**

The spring-boot-starter-test “Starter” (in the test scope) contains the following provided libraries:

* ***Spring Test & Spring Boot Test:*** Utilities and integration test support for Spring Boot applications.
* ***JUnit 4:*** The de-facto standard for unit testing Java applications.
* ***Mockito:*** A Java mocking framework.
* ***AssertJ:*** A fluent assertion library. Better approach for assertions
* ***Hamcrest:*** Used for assertions. A library of matcher objects (also known as constraints or predicates).
* ***JSONassert (skyscreamer):*** An assertion library for JSON.
* ***JsonPath:*** XPath for JSON. Used for JSON parsing. Actual assertions are used from Junit, AssertJ or Hamcrest. Better approach fro assertions compared to JSONassert

**What is Unit Testing, Smoke Testing, Integration Testing, System Testing ?**

***Unit test:*** Specify and test one point of the contract of single method of a class. This should have a very narrow and well defined scope. Complex dependencies and interactions to the outside world are stubbed or mocked. In a Spring application, unit test can test Controller endpoint, Service method or repository method independent of each other. When testing one component, other components can be mocked.

***Integration test:*** Test the correct inter-operation of multiple subsystems of an application. A Spring application itself has different layers like Web (Controller), Business(Service) and Data (Repository). During integration testing, it is important to test flow between these components - Web, Business and Data. Databases and external interfaces can be mocked.

***Smoke test (aka Sanity check):*** A simple integration test where we just check that when the system under test is invoked it returns normally and does not blow up. Results of smoke test make sure that the major functionality is not broken.

Smoke testing is both an analogy with electronics, where the first test occurs when powering up a circuit (if it smokes, it's bad!)

In a Spring application, smoke testing can involve hitting an actuator Health endpoint and checking health status of important components (Main application, DB etc.).

***System Testing:*** System testing is next step after integration testing which involves testing of all major interfaces and databases of a system as a whole.