Testing in Java

In this article, I will try to cover all the basics for building tests in our Java projects. I will mainly use Maven, but Gradle has the same functionality and support all the concepts detailed here.

Main

It's far from the recommended way to test. But yes, sometimes it's easy to run a `main` to test some simple code, or maybe something we don't remember.

```
public static void main(String[] args) {
    var b1 = new BigDecimal("0.2");
    var b2 = new BigDecimal(0.2);
    var b3 = BigDecimal.valueOf(1/5f);

    System.out.println(b1);
    System.out.println(b2);
    System.out.println(b3);

    System.out.println(b1.equals(b2));
    System.out.println(b1.equals(b3));
    System.out.println(b1.equals(b3.setScale(2, RoundingMode.HALF_DOWN)));
    }
```

Output:

```
0.2
0.200000000000000011102230246251565404236316680908203125
0.20000000298023224
false
false
false
```

> The code above can be run them in any online editor. Find one that you can select JDK version.

JUnit

What is JUnit? It's a framework that provides the foundations -the core- for creating and running tests. It defines a TestEngine API for developing testing frameworks. **This means that JUnit is the very basic foundations for Testing in Java.**

> It was written by Kent Beck and Erich Gamma during a flight from Zurich to Atlanta, based on the SUnit (Smalltalk unit testing framework)

Current version is JUnit5, it requires Java8+ to run. But it can run Junit4 tests. Pay attention to the JDK required by your project: Java is more than 20 years old and many-many lines of code were written. So it might be the case, that maintaining legacy code is the project objective. This guide is for Junit 5, but JUnit4 can run most of the features shown here, and differences are a few somewhere below this article.

Get Started: Dependencies

This is the dependency for your project to run Junit tests.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.junit</groupId>
      <artifactId>junit-bom</artifactId>   <!-- This BOM defines all the artifacts
-->
      <version>5.9.2</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <!-- <version>5.9.2</version>  you can leave it out if you use the BOM above
or you can define the version you might need -->
    <scope>test</scope>       <!-- very important:: This dependency is only needed
for test execution -->
</dependency>
```

> The BOM actually inserts the referenced POM into your project's POM.

Again for maintainers, you can see if your project contains it by running the following command:

```
# unix
mvn  dependency:tree | grep -i junit

# windows cmd
mvn  dependency:tree | findstr /i junit

# windows powershell 😢
mvn  dependency:tree | Select-String -NotMatch junit
```

You can use the `mvn dependency:tree --debug` if some missing dependency makes the execution fail.

Structure

Luckily for us in most Java projects, we have a good project structure - I think it was originally proposed by Maven, but I am not able to confirm.

It is very probably that you would find this structure in your maintenance project.

Main application code

```
src/main/java
src/main/resources
src/main/filters
```

Tests application code

```
src/test/java
src/test/resources
src/test/filters
```

This is the ideal project structure. It is not mandatory to have testing code outside your main application code. Thus, you might forget 😮 that you've done some `main` inside a class to have a quick test.

> Don't put Testing code inside your main sources. 😵

Maven test: Surefire Plugin

Ok, we all do the `mvn clean install`, but inside the install goal several things happen, one of them is the `test` goal. Maven execute the `test` goal by running the *Surefire* plugin.

As it says in the doc, by default the Maven Surefire Plugin will scan for test classes whose fully qualified names match the following patterns.

```
**/Test*.java
**/*Test.java
**/*Tests.java
**/*TestCase.java
```

Pay attention to the `**` at the beginning: it means that `*Test` classes can be anywhere !! 😮 In this project, you can look at the MainServiceTest. This is reason for, let me repeat:

> Don't put Testing code inside your main sources. 😵

The importance: Continuous Integration

Having the Surefire plugin being executes every time on the Maven life cycle is very important for ***Continuous Integration*** : if a test fails, something is wrong with the code produced then it must **not** be published into a

productive release. Really the main idea behind the Continuous Integration is exactly this: to run automated tests with the latest code and if something is failing, flag it.

Writing tests

Every method with the `@Test` annotation under `src/test/java` 👍 directory is executed as a test by maven and is considered as such by your IDE.

This is how a test looks like:

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests {

  @Test
  public void addition() {
    assertEquals(Integer.parseInt("1")+1, 2);
  }

}
```

Some notes of the code above:

- `import static` is a way in Java to create *global alias* to functions local to the file, so `assertEquals` in this case will be a "global" function in this file.
- No annotation at class level, only the `@Test` is required.

This code is posted in this project.

A real non-spring test

Consider the following "encryption" method (very weak, don't use for real).

```
public class EncryptService {

  private final String KEY = "SECRET";

  public String encryptPassword(String password) {
    return password+KEY;
  }

  public String decryptPassword(String mangled) {
    return mangled.substring(0, mangled.length()-KEY.length());
  }

}
```

Once your test is created with you favourite IDE, it will look like this:

```
class EncryptServiceTest {

  private EncryptService encryptService;

  /***
   * Tests are classes and can have all classes things: relations, initialization,
inheritance.
   */
  public EncryptServiceTest() {
    this.encryptService = new EncryptService();
  }

  @Test
  @DisplayName("Positive test")
  void testEncryptPositive() {
    String password = "mypassword";
    assertEquals(password+ EncryptService.KEY,
this.encryptService.encryptPassword(password));
  }

  @Test
  @DisplayName("Negative test")
  void testEncryptNegative() {
    String password = "mypassword";
    assertNotEquals(password, this.encryptService.encryptPassword(password));
  }

  public void nonTestMethod() {
    // This is not a test method, because it does not have the @Test.
  }
}
```

Method naming is a subject on its own, I've found this article interesting https://medium.com/@stefanovskyi/unit-test-naming-conventions-dd9208eadbea for naming of methods. Very good analysis. I'm following the first suggestion by the author.

> For a reference on JUnit5 annotations you can look here

Arrange/Act/Assert vs GWT: Given/When/Then

They both have the same principle: to prepare conditions, to execute the code and to evaluate the results. The first is commonly used for Unit testing and TDD, and the later is more frequent in Behavior Driven Development (BDD) context.

A word on Processes & Paradigms

Although, these topics are not part of the scope of this article, let's take a "sentence explanatory" approach.

Test Driven Development (TDD)

TDD is a software process that instead of start by write code based on a requirement, you start writing test cases. So, the code produced should make pass all the tests. This is an approach or a process instead of a tool on its own.

Behavior Driven Development (BDD)

It's uses the Given-When-Then principle to describe User Stories or Scenarios at high level. BDD more thought for Use Cases and higher level definitions for test cases. It is kind of thought for Product Owner and Business users. It has its own language and it not specific for Java.

> They are not mutually exclusive.

Cucumber

Taken from 10 minutes Cucumber guide

Create an empty file called src/test/resources/hellocucumber/is_it_friday_yet.feature with the following content:

```
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario: Sunday isn't Friday
    Given today is Sunday
    When I ask whether it's Friday yet
    Then I should be told "Nope"
```

How is the test executed ??

You can run your tests directly from your IDE. But inside the Maven lifecycle, the Surefire plugin is triggered by the `test` goal, and it will execute all the tests as described in the section Maven test: Surefire Plugin

How do you finally know that your result is what you expected? *Assert*

In order to evaluate that our code was executed properly, we have to compare the result of the code being executed to the desired result.

In JUnit you've got the Asserts. The complete list could be found here:
https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

Useful Annotations

These are some everyday used in Junit:

- `@Test`: Main testing annotation
- `@DisplayName`: descriptive test
- `@BeforeAll, BeforeEach, AfterAll, AfterEach`: method to be execute before or after each or all

- `@Disable`: Force to disable a test

And here some I've found interesting:

- `@Tag`: to filter test execution like in `mvn -Dgroups="integration, fast, feature-168"` or `mvn -DexcludedGroups="slow"` (taken from https://mkyong.com/junit5/junit-5-tagging-and-filtering-tag-examples/)
- `@ParameterizedTest, @RepeatedTest, @Timeout`

JUnit 4 vs 5: Annotations

Yes, they've changed the annotations names. Here the cheatsheet:

JUnit4	JUnit5
@RunWith	@ExtendWith
@Before	@BeforeEach
@After	@AfterEach
@BeforeClass	@BeforeAll
@AfterClass	@AfterAll
@Ignore	@Disabled
@Category	@Tag
@Rule @ClassRule	😵

BeforeAll & AfterAll side note.

The `@BeforeAll` and `@AfterAll` should be static methods. This is in favor of stateless test classes, as described in the https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle.

But if you can group some tests with the same test instance, you can use the `@TestInstance(Lifecycle.PER_CLASS)` class level annotation to reuse some setup. But this has to me explicitly done.

Junit 5 new Annotations

- @DisplayName
- @ParameterizedTest
- @ValueSource(strings = {"foo", "bar"})
- @NullAndEmptySource
- @NullSource
- @EmptySource

Exceptions

```
@Test
@DisplayName("Test Exception")
public void testException() {
    Exception exception = assertThrows(Exception.class, () -> {
Objects.requireNonNull(null); } );
    //exception.printStackTrace();
    System.out.println(exception.getMessage());
    assertNotNull(exception);
    }
```

Mockito (jMock, EasyMock)

As explained before (and also in the previous Webinar), Unit testing should be executed in the most ***isolation*** possible. How do I test my code in isolation if my code has dependencies with other services/classes/objects/external entities ??

That's where Mocking comes into picture: create facade objects and provide them the behavior we want for our test. So, there are leveraging libraries and frameworks that run over JUnit that let us mimic the responses we want for our tests. Mockito, jMock and EasyMock are the most used.

Maven dependency

Check de latest in https://github.com/mockito/mockito/releases

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.3.1</version>
</dependency>
```

Mocking

What we want to mock is not the service we want to test, but the dependencies it has. So considering the following service.

```
public class ManglingService {

  static final String SALT = "SALTKEY";

  static final String DEFAULT = "DEFAULT";

  // Injected bean
  RemoteMD5Client remoteMD5Client;

  public String saltedMD5(String value) {
```

```
    if(value != null && !("".equals(value))) {
      return remoteMD5Client.md5sum(value + SALT);
    } else {
      return remoteMD5Client.md5sum(DEFAULT );
    }
  }
}
```

The `RemoteMD5Client` is the dependency we want to mock. So the code will be like this:

```
@ExtendWith(MockitoExtension.class)
class ManglingServiceTest {

  @Test
  @DisplayName("Testing and mocking a service")
  public void testEncryptSimple() {
    // Mock instance
    RemoteMD5Client remoteMD5Client = Mockito.mock(RemoteMD5Client.class);

    // Conditions -> Arrange - Given  (Mockito can be statically imported)
    Mockito.when(remoteMD5Client.md5sum("simple")).thenReturn("empty");

    // Real service we want to test
    ManglingService manglingService = new ManglingService(remoteMD5Client);

    // The actual Execution ->  Act - When
    String result = manglingService.saltedMD5("simple");

    // Evaluate the result : Assert - Then
    assertEquals("empty" + ManglingService.SALT, result);   /// *** F
  }
}
```

Some notes of the code above:

- `@ExtendWith` is a way to tell JUnit5 what other extension should it use. In this case, only `Mockito`
- Our service to test is not an instance variable, but a local to the method.
- The test above will fail. Spot the error and you get 1 TYP.

Mock vs Spy

A mock object is a full fake object that you have to define all its behaviors. If in our case:

```
// Mock instance
@Mock
RemoteMD5Client remoteMD5Client;

@Test
@DisplayName("Testing and mocking a service")
```

```
public void testEncryptSimple() {
    // Mock instance : Mock Objects can also be create using this Sintax.
    //RemoteMD5Client remoteMD5Client = Mockito.mock(RemoteMD5Client.class);

    // Conditions -> Arrange - Given  (Mockito can be statically imported)
    Mockito.when(remoteMD5Client.md5sum("simple")).thenReturn("empty");

    // call the real method
    Mockito.when(remoteMD5Client.md5sum("notsimple")).thenCallRealMethod();
    Mockito.when(remoteMD5Client.md5sum(any())).thenCallRealMethod();

    // Real service we want to test
    ManglingService manglingService = new ManglingService(remoteMD5Client);

    // The actual Execution ->  Act - When
    String result = manglingService.saltedMD5("complex");

    // Evaluate the result : Assert - Then
    assertEquals("empty" + ManglingService.SALT, result);
}
```

This previous code will fail, because the mock will only give response based on the **Arrange** section we have set. And, we have set for `simple`, but we test for `complex`. (*)

Make sure you test your tests 🐞 👎

Some notes of the code above:

- The `@Mock` annotation will create an instance mock variable.
- Mock will never invoke the real method. You have to explicitly define it to `thenCallRealMethod`
- With the code above, you'll get a `Strict stubbing argument mismatch.`, meaning that you have to define the cases.
- To return `null` instead of the error, the code should be:

```
    lenient().when(remoteMD5Client.md5sum("simple")).thenReturn("empty");
```

- Mode details can be found here

Spy

Instead of a complete fake object, with Spy you can modify the behavior of an existing object instance for special conditions. If the method does not have a condition, it will execute the original class code.

```
  @Spy
  RemoteMD5Client remoteMD5Client;

  @Test
  @DisplayName("Testing and mocking a service")
```

```
    public void testEncryptSpy() {
      // Conditions -> Arrange - Given
      Mockito.when(remoteMD5Client.md5sum("simple")).thenReturn("empty");

      // Real service we want to test
      ManglingService manglingService = new ManglingService(remoteMD5Client);

      // The actual Execution ->  Act - When
      String result = manglingService.saltedMD5("simple");

      // Evaluate the result : Then - Assert
      assertEquals("SIMPLE" + ManglingService.SALT,result) ;
    }
```

The main difference is that for arguments other than `simple`, the `remoteMD5Client` will code it's original backing code.

Some notes of the code above:

- The `@Spy` annotation will spy on the injected object. Without annotation it will be like:

```
RemoteMD5Client remoteMD5ClientSpied = Mockito.spy(new RemoteMD5Client());
```

InjectMocks

In order not to have to call constructor manually for the class we want to test, the `@InjectMocks` annotation will find any mock object in our test and call the constructor with it.

```
    @Mock
    RemoteMD5Client remoteMD5ClientInstance;

    @InjectMocks
    ManglingService manglingServiceMockInjected;

    @Test
    @DisplayName("Test Mocked objects")
    public void testMockedInjectedObject() {
        // Conditions -> Arrange - Given
        Mockito.when(this.remoteMD5Client.md5sum("simple" +
ManglingService.SALT)).thenReturn("empty");

        // The actual Execution ->  Act - When
        String result = this.manglingServiceMockInjected.saltedMD5("simple");

        // Evaluate the result : Then - Assert
        assertEquals("empty", result);
    }
```

Mocking `static` methods

Static method are controversial, but they are there and they exist. Again, Java is 20+ years old and many hands has written code on it. So we will find several situations like this.

Setting up

To let Mockito be able to mock static methods, you have to create a file into the `src/test/resources/mockito-extensions/org.mockito.plugins.MockMaker` directory of your project with a single line :

```
mock-maker-inline
```

More details on https://github.com/mockito/mockito/wiki/What%27s-new-in-Mockito-2#mock-the-unmockable-opt-in-mocking-of-final-classesmethods. They say that this comes with a performance cost, but I can't confirm.

So going to the test, our code will be like:

```
public class CommonUtils {

  static String EmptyOrRandom(String value) {
    if("".equals(Objects.toString(value, ""))) {
      return Long.toString(Random.from(RandomGenerator.getDefault()).nextLong());
    }
    return value;
  }
}
```

And the test be like:

```
@ExtendWith(MockitoExtension.class)
class CommonUtilsTest {

    @Test
    @DisplayName("To test Static Method")
    public void testStatic() {
        assertNotEquals("", CommonUtils.EmptyOrRandom(""));
        assertEquals("something", CommonUtils.EmptyOrRandom("something"));

        try (MockedStatic<CommonUtils> utilities =
Mockito.mockStatic(CommonUtils.class)) {
          utilities.when(() ->
CommonUtils.EmptyOrRandom("value")).thenReturn("othervalue");
          assertEquals("othervalue", CommonUtils.EmptyOrRandom("value"));
        }
```

```
    }
}
```

Some code common references

```
// To Throw an Exception
Mockito.when(remoteMD5Client.md5sum(isNull())).thenThrow(new
IllegalStateException(""));

// To answer with the params passed
Mockito.when(remoteMD5Client.md5sum(anyString())).thenAnswer(invocation ->
        invocation.getArgument(0, String.class).toUpperCase());

// To assert on exceptions
assertThrows(IllegalStateException.class, () -> manglingService.saltedMD5(null));
```

PowerMock

Taken from https://github.com/powermock/powermock

> *PowerMock is a framework that extends other mock libraries such as EasyMock with more powerful capabilities.*
>
> *PowerMock uses a custom classloader and bytecode manipulation to enable mocking of static methods, constructors, final classes and methods, private methods, removal of static initializers and more.*

Some notes about PowerMock

> It's not a daily-maintained library. Be careful when using. Last commit was done 4 months ago.

How to PowerMock

I'll not go into details here as most of the things were already covered by Mockito section.

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(Static.class)

PowerMockito.mockStatic(Static.class);

Mockito.when(Static.firstStaticMethod(param)).thenReturn(value);
```

Testing Private methods

One question from previous Webinars is how to test Private. You can do that with PowerMock, but also spring has it Spring ReflectionTestUtils which has same functionality. We will cover Spring later in this article.

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/util/ReflectionTestUtils.html

```
@Component
public class PrivateMethodService {

    public boolean allApproved(List<Integer> grades) {
        for(Integer grade : grades) {
            if(evaluateApproved(grade))
                return false;
        }
        return true;
    }

    private boolean evaluateApproved(final Integer grade) {
        return grade > 70;
    }


}


@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = PrivateMethodService.class)
class PrivateMethodServiceTest {

  @Autowired
  PrivateMethodService privateMethodService;

  @Test
  void testSomething() {
    Boolean b = ReflectionTestUtils.invokeMethod(privateMethodService,
"evaluateApproved", 50);
    System.out.println(b);
  }

}
```

Maven Dependency

Check the latest on https://github.com/powermock/powermock/wiki/Mockito-Maven

```
<properties>
<powermock.version>2.0.2</powermock.version>
</properties>
<dependencies>
<dependency>
  <groupId>org.powermock</groupId>
  <artifactId>powermock-module-junit4</artifactId>
```

```
  <version>${powermock.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.powermock</groupId>
  <artifactId>powermock-api-mockito2</artifactId>
  <version>${powermock.version}</version>
  <scope>test</scope>
</dependency>
</dependencies>
```

Code Coverage

Code coverage is a *software metric* that says how much of our code has been executed during our tests. The IntelliJ has a plugin, but also we have the JaCoCo maven plugin.

In IntelliJ, you can **Run with Coverage** plugin and the results look like this:

JaCoCo

For JaCoCo, you have to run the maven plugin like this.

```
mvn -Djacoco.skip=false test jacoco:report
./target/site/jacoco/index.html
```

Because it's a simple to get metric and works, many projects (managers?) will require a minimum of Code Coverage to pass the tests Continuos Integration.

```
<configuration>
    <rules>
        <rule>
            <element>BUNDLE</element>
            <limits>
                <limit>
                    <counter>INSTRUCTION</counter>
```

```
                    <value>COVEREDRATIO</value>
                    <minimum>0.10</minimum>
                </limit>
                <limit>
                    <counter>BRANCH</counter>
                    <value>COVEREDRATIO</value>
                    <minimum>0.10</minimum>
                </limit>
                <limit>
                    <counter>CLASS</counter>
                    <value>MISSEDCOUNT</value>
                    <maximum>99</maximum>
                </limit>
            </limits>
        </rule>
    </rules>
</configuration>
```

Jacoco home page : https://www.jacoco.org/jacoco/trunk/index.html

---

Integration tests

As explain in my `Software testing` webinar, Unit Testing is used when **Testing in isolation**, whilst Integration Testing is to test one or more things.

The *isolation* part is not physical: you can think as it as method, a class, a service, multiple units. To me, it's when the code you're testing is ***not*** interacting with anything outside it.

In practice, Maven uses 2 different plugins to run them and the behavior they have is different. By default, the **Maven Surefire Plugin** executes unit tests during the test phase, while the **Failsafe* plugin runs integration tests in the integration-test phase. Taken from here

The **Failsafe Plugin** does not fail the build immediately, but executes one final phase, the *post-integration-test* to tear down and clean up your test execution, like stop a server, clean SQL DB, or remove files.

Run below to execute only Unit Tests (*Test)

```
mvn clean test
```

You can execute below command to run the tests (both unit and integration *IT)

```
mvn clean verify
```

SpringBoot and Integration Tests

To make sure that the lifecycle of your Spring Boot application is properly managed around your integration tests, you can use the start and stop goals, as shown in the following example:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <executions>
                <execution>
                    <id>pre-integration-test</id>
                    <goals>
                        <goal>start</goal>
                    </goals>
                </execution>
                <execution>
                    <id>post-integration-test</id>
                    <goals>
                        <goal>stop</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Taken from https://docs.spring.io/spring-boot/docs/2.5.x/maven-plugin/reference/htmlsingle/#integration-tests

Same as with Surefire plugin, only the below file patterns will be considered for integration test.

```
<includes>
    <include>**/IT*.java</include>
    <include>**/*IT.java</include>
    <include>**/*ITCase.java</include>
</includes>
```

Reference : https://maven.apache.org/surefire/maven-failsafe-plugin/examples/inclusion-exclusion.html

More References

- https://stackoverflow.com/questions/1399240/how-do-i-get-my-maven-integration-tests-to-run/1399277
- https://stackoverflow.com/questions/28986005/what-is-the-difference-between-the-maven-surefire-and-maven-failsafe-plugins
- https://www.baeldung.com/maven-surefire-vs-failsafe

Spring & SpringBoot

https://docs.spring.io/spring-framework/reference/testing/introduction.html

The basics

- Use the `@ExtendWith(SpringExtension.class)` to test Spring apps that create the SpringContext
  - The `@RunWith(SpringRunner.class)` is for JUnit4. It should be replaced in favor of `@ExtendWith(SpringExtension.class)`
- `@SpringJUnitConfig` : is a composed annotation that combines `@ExtendWith(SpringExtension.class)` from JUnit Jupiter with `@ContextConfiguration`
- `@SpringBootTest` :

SpringBoot tests

SpringBoot tests are a kind of integration tests, because they initialize the ApplicationContext and the SpringBootApplication itself (not really executing the `main`).

The starter

As per SpringBoot testing documentation, "*Most developers use the spring-boot-starter-test "Starter", which imports both Spring Boot test modules as well as JUnit Jupiter, AssertJ, Hamcrest, and* ***a number of other useful libraries***."

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

Details from https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#features.testing.test-scope-dependencies

- JUnit 5: The de-facto standard for unit testing Java applications.
- Spring Test & Spring Boot Test: Utilities and integration test support for Spring Boot applications.
- Mockito: A Java mocking framework.
- JSONassert: An assertion library for JSON.

```
String result = "{id:1,name:\"Juergen\"}";
JSONAssert.assertEquals("{id:1}", result, false); // Pass
JSONAssert.assertEquals("{id:1}", result, true); // Fail
```

- JsonPath: XPath for JSON.

```
String author = JsonPath.parse(json).read("$.store.book[0].author");
Book book = JsonPath.parse(json).read("$.store.book[0]", Book.class);
```

- AssertJ A fluent assertion library.
- Hamcrest: A library of matcher objects (also known as constraints or predicates).

Annotations

- `@MockBean`: Creates Mock bean into the application context and will be later injected into the objects that make use of it. The difference between `@InjectMocks` is that the MockBean can be shared between several objects.
- `@SpyBean`: Same, but with "Spy"s.

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = ManglingService.class)
@RequiredArgsConstructor
@TestInstance(Lifecycle.PER_CLASS)
class EncryptControllerBeanTest {

  @Nullable
  @MockBean
  RemoteMD5Client remoteMD5ClientInstanceBean;

  @Autowired
  ManglingService manglingService;

  @BeforeAll
  void init() {
    // Arrange

Mockito.when(remoteMD5ClientInstanceBean.md5sum(any())).thenReturn("SOMEVALUE");
  }

  @Test
  public void simpleTest() {
    // The actual Execution ->  Act - When
    String result = manglingService.saltedMD5("simple");

    // Evaluate the result : Then - Assert
    assertNotEquals("SIMPLE" + SALT, result);

  }
}
```

Some notes on the code above:

- I'm not using any SpringBoot specific feature, so I only `@ExtendWith(SpringExtension.class)`
- I'm using another package structure and the `@ContextConfiguration(classes = ManglingService.class)` sets the ManglingService in the Application Context.

- I use the `@TestInstance(Lifecycle.PER_CLASS)` to setup the conditions at instance level.
- I have not configured the RemoteMD5Client.class on purpose: I'm setting a mock out of it.
- To use a `@Spy`, I should have add it to the `@ContextConfiguration`

`@SpringBootTest` Annotation

"Spring Boot provides the @SpringBootTest annotation which we can use to create an application context containing all the objects we need for all of the above test types. Note, however, that overusing @SpringBootTest might lead to very long-running test suites." (taken from here)

"Spring Boot's auto-configuration system works well for applications but can sometimes be a little too much for tests. It often helps to load only the parts of the configuration that are required to test a "slice" of your application. For example, you might want to test that Spring MVC controllers are mapping URLs correctly, and you do not want to involve database calls in those tests" (takten from here)

```
@SpringBootTest
@AutoConfigureMockMvc
class EncryptControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @Test
  public void testSayHello() {
    try {
      this.mockMvc.perform(MockMvcRequestBuilders.get("/sayHello"))
          .andExpect(MockMvcResultMatchers.status().isOk())
          .andDo(MockMvcResultHandlers.print());
    } catch (Exception e) {
      throw new RuntimeException(e);
    }

  }

  @Test
  public void testDoATest() {
    try {
      this.mockMvc.perform(MockMvcRequestBuilders.get("/do-a-test"))
          .andExpect(MockMvcResultMatchers.status().isOk())
          .andDo(MockMvcResultHandlers.print());
    } catch (Exception e) {
      throw new RuntimeException(e);
    }

  }
}
```

Some notes on the code above:

- I needed to `@AutoConfigureMockMvc` in order to have the MockMvc for my tests

- This loaded the whole SpringBoot context. In this case is needed because the `/sayHello` is a SpringCloud Function like

```
    @Bean
    public Supplier<String> sayHello() {
        return () -> "Hello";
    }
```

Spring "Slices" or reduced application scopes

So Spring has provided a set of reduced "slices" Autoconfigured environments with the **Autoconfigurations** https://docs.spring.io/spring-boot/docs/current/reference/html/test-auto-configuration.html. The main ones are:

- `@WebMvcTest`: Limits to Spring MVC related components. Details, specially the `@AutoConfigureMockMvc`
- `@DataJpaTest`: To test JPA functionality. Scans for `@Entity`s
- `@Data*Test`: To test database specific Templates, like Cassandra, MongoDB, Redis, etc

> Each of the `@...Test` annotation include one or more `@...AutoConfiguration`

```
@WebMvcTest
class EncryptControllerIT {

  @Autowired
  private MockMvc mockMvc;

  @Test
  public void testSayHello() {
    try {
      // Spring Cloud Function is not created when using the WebMvcTest
      this.mockMvc.perform(MockMvcRequestBuilders.get("/sayHello"))
          .andExpect(MockMvcResultMatchers.status().is4xxClientError());
    } catch (Exception e) {
      throw new RuntimeException(e);
    }

  }

  @Test
  public void testDoATest() {
    try {
      this.mockMvc.perform(MockMvcRequestBuilders.get("/do-a-test"))
          .andExpect(MockMvcResultMatchers.status().isOk())
          .andDo(MockMvcResultHandlers.print());
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
```

```
    }
}
```

See example from https://www.baeldung.com/spring-boot-testing

More on Test Slices

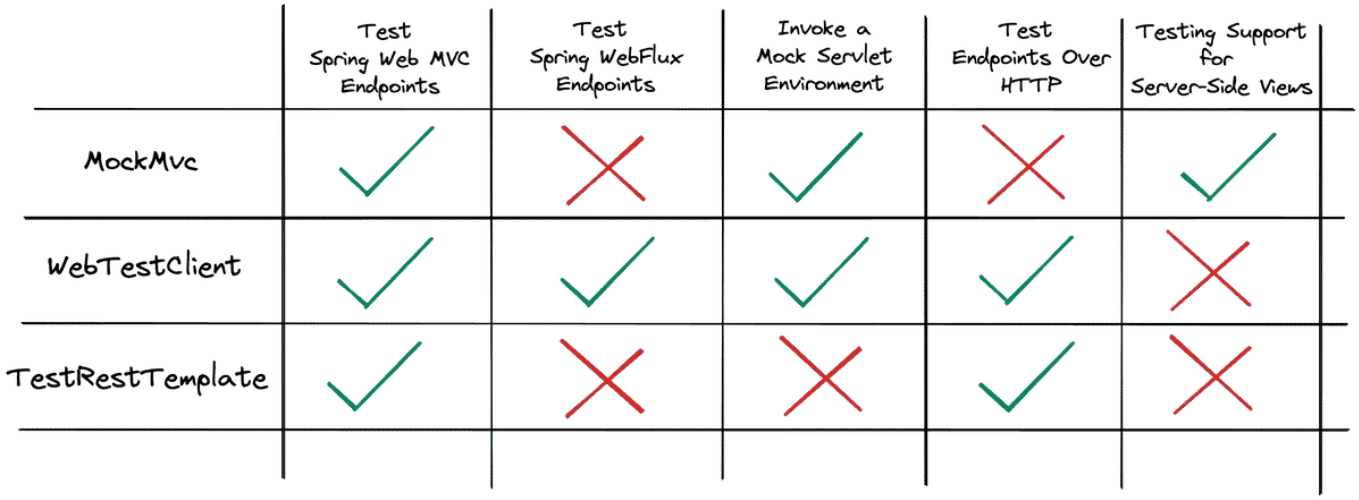
https://tanzu.vmware.com/developer/guides/spring-boot-testing/

https://www.baeldung.com/spring-boot-testing https://www.baeldung.com/spring-boot-testing-pitfalls#3-test-slices https://reflectoring.io/spring-boot-test/

Reference: https://docs.spring.io/spring-boot/docs/current/reference/html/test-auto-configuration.html

MockMvc vs WebTestClient vs TestRestTemplate

Taken from https://rieckpil.de/spring-boot-testing-mockmvc-vs-webtestclient-vs-testresttemplate/:



- **MockMvc**: Fluent API to interact with a mocked servlet environment. No real HTTP communication. The perfect solution to verify blocking Spring WebMVC controller endpoints. We either bootstrap the MockMvc instance on our own, use @WebMvcTest or @SpringBootTest (without a port configuration). Includes API support for verifying the model or view name of a server-side rendered view endpoint.
- **WebTestClient**: Originally the testing tool for invoking and verifying Spring WebFlux endpoints. However, we can also use it to write tests for a running servlet container or MockMvc. Fluent API that allows chaining the request and verification. There's no API support for verifying the model or view name of a server-side rendered view endpoint.
- **TestRestTemplate**: Test and verify controller endpoints for a running servlet container over HTTP. Less fluent API. If our team is still familiar with the RestTemplate and hasn't made the transition to the WebTestClient (yet), we may favor this for our integration tests. We can't use the TestRestTemplate to interact with mocked servlet environment or Spring WebFlux endpoints. There's no API support for verifying the model or view name of a server-side rendered view endpoint.

Final notes and references

First Principle

https://www.appsdeveloperblog.com/the-first-principle-in-unit-testing/

Trade offs

Creating test is time consumer and if project is on a tight schedule, it could be reduced and impact quality. That' a decision beyond our programming scope.

Squaretest

Browsing some IntelliJ, I've seen the https://squaretest.com/ and looks promising.

> Automatically generate unit tests for your Java classes with the Squaretest plugin for IntelliJ IDEA.

I have not tried, but if someone did, let me know. And it's not pricey.

References

---

https://methodpoet.com/unit-testing-best-practices/

To convert MD to DOCx: https://cloudconvert.com/md-to-docx
