JUnit Utils User Guide

Marc MAMIAH, Luxembourg

Version {junit-utils.version}, 2021-06-09

Table of Contents

Repo: github.com/mmamiah/junit-utils

Abstract

JUnit Utils aim as a set of good practices and tools, is to simplify software developers approach when it comes to planning, designing, and writing unit tests. The present user guide dive into the proposed features and solution, and provide the documentation for users and developers.

There are already a lot of frameworks providing an impressive added value to software testing, like JUnit, Mockito, Hamcrest, Awaitility and many others. Nevertheless, writing tests is sometimes painful for many reasons. You surely faced the desire to modify your production class to make it more 'testable', or even skip (leave it for integration tests) a peace of code because preparing the testcase for it could/will be time consuming.

The *JUnit Utils* brings few additional tools, to ease and simplify the unit tests design and implementation approach, and therefore be gratified by a cleanest and more readable code, an impressive code coverage score, and last but not least, prevent a large amount of bugs to occur.

> *Testing leads to failure, and failure leads to understanding.*
>
> — Burt Rutan, Retired American aerospace engineer and entrepreneur

Chapter 1. Installation

1.1. pom.xml

The journey trip with *JUnit Utils* start by including it in your project *pom.xml* file.

Listing 1. JUnit Utils dependency

```
<dependency>
    <groupId>lu.mms.common</groupId>
    <artifactId>junit-utils</artifactId>
    <version>${junit-utils.version}</version>
    <scope>test</scope>
</dependency>
```

Keep up to date

The *JUnit Utils* latest version is: [**{junit-utils.version}**]

It is also important to note that this version requires **Java 11** to work.

Having this done, we do not need anything else to enjoy unit testing our code.

1.2. Maven Dependencies

The following dependencies are ready to be used as soon as *JUnit Utils* has been installed:

1. org.yaml.snakeyaml
2. org.mybatis.mybatis
3. org.apache.commons
    - commons-text
    - commons-lang3
4. commons-io
5. logback-classic
6. org.reflections
7. spring-boot-starter-test
8. hamcrest
9. mockito
    - mockito-junit-jupiter
    - mockito-core
10. H2
11. awaitility

12. apiguardian-api
13. JUnit 5
    - junit-platform:
        - junit-platform-runner
        - junit-platform-engine
        - junit-platform-commons
        - junit-platform-launcher
    - junit-jupiter
        - junit-jupiter-api
        - junit-jupiter-engine
        - junit-jupiter-params

Chapter 2. Framework tour

2.1. Configuration Parameters

While using *JUnit Utils* library, you might need to modify some default behavior. You simply have to create a file named *junit-utils.XXX*, where *XXX* stands for the configuration file type. It could be a **YAML** file or a **properties** file. example of valid configuration file names:

- junit-utils.yaml
- junit-utils.yml
- junit-utils.properties

For those you prefer to keep the JUnit configuration together, it is possible to use the JUnit platform configuration to store the *JUnit Utils* configuration in the file named *junit-platform.properties*. The table below lists the configuration entries.

Table 1. JUnit Utils properties

Property	Description	Default value
`component-scan`	The package that will be scanned by the framework	
`log-reflections`	ON/OFF the reflections logging	true
`show-banner`	Shows the library banner	true
`fancy-banner`	Shows the fancy banner if 'show-banner' = true and 'fancy-banner' = true	true

Listing 2. Example of yaml configuration

```
junit-utils:
  component-scan:   lu.mms
  log-reflections:  false
  show-banner:      true
```

Listing 3. Example of .properties configuration

```
junit-utils.show-banner=true
junit-utils.fancy-banner=false
```

2.2. Extensions API

One of the major improvement brought by *JUnit 5* is it extension model. Such extension give us the possibility to customise the test execution. A couple of extension are proposed by *JUnit Utils* for that purpose.

2.2.1. ReturnsMocksExtension

Unit testing a class implies to mock a lot of classes around. Some of the mocked classes are often declared in the unit test in order to controller their behavior (with stubbers), and we expect from those classes the keep having the same behavior in the text context.

Let set the following context:

Listing 4. Testcase context

```
1 public class Customer {
2
3     private Identity identity;
4
5     public Identity getIdentity() {
6         return identity;
7     }
8 }
9
10 public class Identity {
11
12     private final Integer id = 987456;
13
14     public Integer getId() {
15         return id;
16     }
17 }
18
19 public class Report {
20
21     private Customer customer;
22
23     public Integer getCustomerId() {
24         return customer.getIdentity().getId();
25     }
26
27     public Customer getCustomer() {
28         return customer;
29     }
30 }
```

When unit testing the *CustomerManager* let say we would like to check that the *CustomerManager* return the right ID for a given customer, and that the correct method is fired in *Identity*. Doing this require a lot of operation to be done: **(1)** mock *Identity*, **(2)** stub to *getId()* so that it returns the expected ID, **(3)** mock *Customer* and **(4)** stub *getIdentity()* to return *IdentityMock*.

Listing 5. Unit test straight approach

```
1  class MockExample1Test {
2
3      private static final Integer ID = 500;
4
5      private Report sut;
6      private Customer customerMock;
7      private Identity identityMock;
8
9      @BeforeEach
10     void init() {
11         // Prepare mocks
12         identityMock = Mockito.mock(Identity.class);
13         customerMock = Mockito.mock(Customer.class);
14         when(customerMock.getIdentity()).thenReturn(identityMock);
15
16         // Prepare SUT
17         sut = new Report();
18         ReflectionTestUtils.setField(sut, "customer", customerMock);
19     }
20
21     @Test
22     void shouldConfirmCustomerIdWhenIdentityHasValidId() {
23         // Arrange
24         when(identityMock.getId()).thenReturn(ID);
25
26         // Act
27         final Integer id = sut.getCustomerId();
28
29         // Assert
30         assertThat(id, equalTo(ID));
31
32         // assert that the mock returned is the one we declared.
33         assertThat(sut.getCustomer().getIdentity(), equalTo(identityMock));
34     }
35 }
```

Please note that the more behaviors to stub and cases to verify, the more we will have to repeat the *init()* section in our example. So lets try to automatise this initialization process. To do this, let us: **(1)** comment the stub *when(customerMock.getIdentity()).thenReturn(identityMock);*, **(2)** configure *customerMock* to return mocks (see line 14), and **(3)** and run the test again.

a bit surprised ... the test failed.

This is because the returned mock (by *customerMock*) is not the same as the one we declared in the our test. Just modify the asserts to confirm this state as you can see in the bellow listing.

Listing 6. Simplifying the testcase initialization step: attempt #1

```
class MockExample2Test {

    private static final Integer ID = 500;

    private Report sut;
    private Customer customerMock;
    private Identity identityMock;

    @BeforeEach
    void init() {
        // Prepare mocks
        identityMock = Mockito.mock(Identity.class);
        customerMock = Mockito.mock(Customer.class, Answers.RETURNS_MOCKS);
        // when(customerMock.getIdentity()).thenReturn(identityMock);

        // Prepare SUT
        sut = new Report();
        ReflectionTestUtils.setField(sut, "customer", customerMock);
    }

    @Test
    void shouldNotFindCustomerIdWhenIdentityMockDiffers() {
        // Arrange
        when(identityMock.getId()).thenReturn(ID);

        // Act
        final Integer id = sut.getCustomerId();

        // Assert

        // We will keep in following example to assert that the mock returned
        // is the one we declared, but by now, to keep the test passing we will
        // assert that the mock is different.
        assertThat(id, notNullValue());
        verify(identityMock, never()).getId();

        // Just to confirm we are not talking about the same mock.
        assertThat(id, IsNot.not(ID));
        assertThat(sut.getCustomer().getIdentity(), IsNot.not(identityMock));
    }
}
```

Assuming we are declaring a mock in our test class, we would like to interact with that mock all over the test run, and verify his behavior if needed. It comes to us that in the context of this JUNit 5 custom extension, to have a unique mock instance in the testcase context.

Fortunately, **@ReturnsMocksExtension** helps to resolve this anomaly and returns the mock declared in our test, so that any predefined behavior or stub applies to the test case. We just need to

extend our class with the **ReturnsMocksExtension**:

Listing 7. ReturnsMocksExtension installation

```
@ExtendWith(ReturnsMocksExtension.class)
class MockExample3Test {
    // ...
}
```

Of course, to keep the code readable and let the developer focus in the test case, lets get rid of this initialization phase, which can grow up really fast with the classes to mock and behaviors to stub, and add the **MockitoExtension** as well.

Listing 8. Simplifying the testcase initialization step: attempt #2

```
@ExtendWith({MockitoExtension.class, ReturnsMocksExtension.class})
class MockExample4Test {

    private static final Integer ID = 500;

    @InjectMocks
    private Report sut;

    @Mock(answer = Answers.RETURNS_MOCKS)
    private Customer customerMock;

    @Mock
    private Identity identityMock;

    @Test
    void shouldConfirmCustomerIdWhenIdentityIsInitialized() {
        // Arrange
        when(identityMock.getId()).thenReturn(ID);

        // Act
        final Integer id = sut.getCustomerId();

        // Assert
        // confirms that the mock returned is the one we declared in the test class
        assertThat(id, equalTo(ID));
        assertThat(sut.getCustomer().getIdentity(), equalTo(identityMock));
    }
}
```

As you already noted, the example **#4** if much more elegant and simple to read, understand and maintain than example **#1**. We can easily focus on the purpose of the test, by removing the noisy code. Of course, this example is really simplified but as soon as the code grows up, the proposed approach is really helpful.

2.2.2. MockInjectionExtension

When coding, it happens that we have to deal with beans which shares one or more common interfaces, or simply, extends the same base classes. Testing then is naturally simple, but we have to put some effort on declaring/creating the related mocks, prepare the collection and inject them personally ensure that they will be injected in the correct field, which some chance for the field to be renamed for example.

The *MockInjectionExtension* will help a lot here by doing it for us, and again, help us to keep the noisy code out of what we should focus on. It is also important to notice that this extension will inject the mocks & spies declared by the user in the test class instance. Last but not least, the subject under test is instantiated via appropriate constructor, setter and even lookup methods ('@Autowired' method) with the corresponding mocks as arguments. By mocks here, we understand the ones declared by the user in the test class, otherwise a new mock instance will be created.

Listing 9. MockInjectionExtension & mock/spy injection.

```
@ExtendWith({MockitoExtension.class, MockInjectionExtension.class})
class MockInjectionExtensionTest {

    private static final String BY_CONSTRUCTOR_TEMPLATE = "by_constructor_%s";
    private static final String BY_METHOD_TEMPLATE = "by_method_%s";

    @InjectMocks
    private Computer sut;

    @Mock
    private PointingDevice pointingDeviceMock;

    @Mock
    private Mice miceMock;

    @Spy
    private Keyboard keyboardSpy;

    @Test
    void shouldInjectMockViaAutowiredConstructor() {
        // Arrange

        // Act
        final Map<String, Device> items = sut.getDevicesByConstructor();

        // Assert
        assertThat(items, IsNot.not(anEmptyMap()));
        assertThat(items.values(), hasItem(oneOf(pointingDeviceMock, miceMock,
keyboardSpy)));
    }

    @Test
    void shouldInjectMockWhenAutowiredMethodCalled() {
```

```
33          // Arrange
34
35          // Act
36
37          // Assert
38          assertThat(sut.getValueTwo(), notNullValue());
39          assertThat(sut.getDevices(), notNullValue());
40      }
41
42      @Test
43      void shouldInitTheMocksCollection() {
44          // Arrange
45
46          // Act
47
48          // Assert
49          assertThat(sut.getCollection(), iterableWithSize(3));
50          assertThat(sut.getList(), iterableWithSize(3));
51          assertThat(sut.getSet(), iterableWithSize(3));
52          assertThat(sut.getArray(), arrayWithSize(2));
53      }
54
55      static class Computer {
56          // --- attributes initialized via constructor
57          private final Map<String, Device> devicesByConstructor;
58
59          // --- attributes initialized via @Autowired methods
60          private Object valueTwo;
61          private Map<String, Device> devices;
62
63          private Collection<Device> collection;
64
65          private List<Device> list;
66
67          private Set<Device> set;
68
69          private PointingDevice[] array;
70
71          public Computer(final List<Device> devices) {
72              this.devicesByConstructor = Stream.ofNullable(devices)
73                      .flatMap(Collection::stream)
74                      .collect(Collectors.toMap(
75                              item -> String.format(BY_CONSTRUCTOR_TEMPLATE,
    MockUtil.getMockName(item).toString()),
76                              Function.identity())
77                      );
78          }
79
80          @Autowired
81          private void initOne(final String aValue) {
82              this.valueTwo = StringUtils.defaultString(aValue);
```

```
83          }
84
85          @Autowired
86          private void initTwo(final List<Device> aValue) {
87              this.devices = aValue.stream().collect(Collectors.toMap(
88                      item -> String.format(BY_METHOD_TEMPLATE, MockUtil.
    getMockName(item).toString()),
89                      Function.identity()));
90          }
91
92          public Map<String, Device> getDevicesByConstructor() {
93              return devicesByConstructor;
94          }
95
96          public Object getValueTwo() {
97              return valueTwo;
98          }
99
100         public Map<String, Device> getDevices() {
101             return devices;
102         }
103
104         public Collection<Device> getCollection() {
105             return collection;
106         }
107
108         public List<Device> getList() {
109             return list;
110         }
111
112         public Set<Device> getSet() {
113             return set;
114         }
115
116         public PointingDevice[] getArray() {
117             return array;
118         }
119     }
120
121     // Interface used to group the mocks
122     interface Device {
123         // empty interface
124     }
125
126     // simple example of use of the interface
127     static class PointingDevice implements Device {
128         // empty class
129     }
130
131     // extending another a class implementing the desired interface
132     static class Mice extends PointingDevice {
```

```
133         // empty class
134     }
135 
136     // Another interface implementation to check multiple interface implementation
137     static class Keyboard implements Device {
138         // empty class
139     }
140 
141 }
```

2.2.3. ReinforceMockExtension

This extension is purely technical. It helps to avoid NPE when lifecycle methods are executed, and some method members were not mocked at all or simply injected. This extension will scan the test instance, recover the properties annotated with @Mock, and inject the empty fields with the declared mocks when relevant.

Listing 10. ReinforceMockExtension in action.

```
@ExtendWith({MockitoExtension.class, ReinforceMockExtension.class})
class ReinforceMockExtensionExampleTest {

    @Mock
    private Report reportMock;

    @Mock
    private Customer customerMock;

    @Mock
    private Identity identityMock;

    @Test
    void shouldConfirmThatReportMockPropertiesAreAllDefaultedWenRelevant() {
        // Arrange

        // Act
        final Customer customer = (Customer) ReflectionTestUtils.getField
(reportMock,"customer");

        // Assert
        assertThat(customer, notNullValue());
        assertThat(customer, equalTo(customerMock));
        assertThat(MockUtil.isMock(customer), equalTo(true));
    }

    @Test
    void shouldConfirmThatMockPropertiesAreAllDefaultedWenRelevant() {
        // Arrange

        // Act
        final Identity identity = (Identity) ReflectionTestUtils.getField
(customerMock,"identity");

        // Assert
        assertThat(identity, notNullValue());
        assertThat(identity, equalTo(identityMock));
        assertThat(MockUtil.isMock(identity), equalTo(true));
    }
```

2.2.4. MockitoSpyExtension

As many developers, we try our best to test solitary unit as the require less work in test preparation. Most of the border classes are mocked and we are just dealing with the behaviors we have prepared. But not all unit testers use solitary tests. It is some times required to define sociable units, and indeed without focusing on the neighboring classed we try to reproduce the way the unit behave in production environment. A solution to do so is ito use the spies, and of course, they are not fully initialised.

This is where the *MockitoSpyExtension* will help by injecting the declared test class mocks and spies if possible. As any other JUnit 5 extension, we just need to register the extension for the tests class spy to be initialized with relevant mock and spies.

Listing 11. MockitoSpyExtension in action

```
@ExtendWith(MockitoSpyExtension.class)
class MockitoSpyExtensionExampleTest {
    @Spy
    private House houseSpy;

    @Mock
    private Room roomMock;

    // ...
}
```

In the above listing the 'roomMock' will be injected in the 'houseSpy', if of course, there is a such field (with) the same type in the 'House' class.

2.2.5. BeanLifeCycleExtension

This extension make it possible to test the bean life cycle method. One of the coolest annotation we use to use when coding is the *@PostConstruct*/*@PreDestroy* annotations. As per the *@PostConstruct* javadoc for example, a method annotated with this annotation should be invoked before the class is put into service, and therefore, before the class is also tested. In integration test for example we do not have so much to do, as it is usually properly executed after dependency injection. Let set the following testcase context.

Listing 12. Testcase context

```
public class Identity {

    private String id;

    @PostConstruct
    private void init() {
        this.id = "initial-id";
    }

    public String getId() {
        return id;
    }
}
```

Unit testing this class will required to run the *init()* method, if we want to reproduce its full behavior like in production context. Lets give a try.

Listing 13. PostConstruct Initialization: attempt #1

```
1 class BeanLifeCycleExtensionExample1Test {
2
3     private final Identity sut = new Identity();
4
5     @Test
6     void shouldHaveValueInitializedWhenPostConstructExecuted() {
7         // Arrange
8         assumeTrue(sut.getId() == null);
9
10        // Act
11        ReflectionTestUtils.invokeMethod(sut, "init");
12
13        // Assert
14        assertThat(sut.getId(), notNullValue());
15    }
16 }
```

As in the previous section example, we need to remember to aware of all the *@PostConstruct* methods, and execute via reflection feature to init the testcase, or even worse, make the *@PostConstruct* method available for other class in the production code, just because are struggling to access it in the unit test. Moreover, renaming the *@PostConstruct* method implies aligning the test as well.

With the *BeanLifeCycleExtension*, we automatise this process, remove the noisy initialization part of the test case, and ensure that the test case will be modified only when the logic in the production class change, with optimized effort for code maintenance and code review. The improved unit test looks like this:

Listing 14. PostConstruct Initialization: attempt #2

```
1 @ExtendWith(BeanLifeCycleExtension.class)
2 class BeanLifeCycleExtensionExample2Test {
3
4     private final Identity sut = new Identity();
5
6     @Test
7     void shouldHaveValueInitializedWhenPostConstructExecuted() {
8         // Ac / Assert
9         assertThat(sut.getId(), notNullValue());
10    }
11 }
```

We can notice that the noisy code just disappeared, and the developer can focus on the behaviors to test.

2.2.6. MocksContextParameterResolver

This is a helper extension, as its name states, it is a paramter provider, which goal is to provide the mocks context to the test instance method. Any debugging and mocks context check is therefore possible to verify the value using during the test case initiatization.

2.2.7. MyBatisMapperExtension

(see annotation *@MyBatisMapperTest*)

2.3. Basic Annotations

As designed, annotation make it easy to add a given behavior to and object or a class. This is why annotations have been a good candidate to ease developer life when it comes to software testing. *JUnit Utils* comes with couples of annotations to achieve that goal.

2.3.1. *@MockValue*

When coding most of the time with properties values, referring to them via the *@Value* annotation, we need extra steps in the test case settings, like inject value to the annotated fields. Some times, the annotated field is modified after code reviews or simple code refactoring. This implies to comeback to the test and align it as well.

With the ***@MockValue*** annotation, we automatise this process and initialise the subject under test (*@InjectMock* annotated) properties. We will focus on the following context in the upcoming examples.

Listing 15. Testcase context

```
1 public class Identity {
2
3     @Value("${identity-default-value}")
4     private String id;
5
6     public String getId() {
7         return id;
8     }
9 }
```

To unit this class, we usually have to **(1)** modified the production code by adding a setter, or **(2)** within the testcase, inject a value to the *id* variable.

Listing 16. Injecting a value to the @Value field.

```
1 class MockValueExample1Test {
2
3     private final Identity sut = new Identity();
4
5     @Test
6     void shouldFindIdWhenIdManuallySet() {
7         // Arrange
8         assumeTrue(sut.getId() == null);
9
10        final String id = "id_123";
11        ReflectionTestUtils.setField(sut, "id", id);
12
13        // Act / Assert
14        assertThat(sut.getId(), equalTo(id));
15    }
16 }
```

This operation look quit simple, but lets think about any change in the production code, as shown in *MockValueExample1Test* listing. This implies to align the test with the code, and more code to review for the colleagues. This unit test can be improved using the **@MockValue** (and **MockValueExtension**). Please take a look at the reworked previous unit test case.

Listing 17. MockValue and MockValueExtension in action.

```
1 @ExtendWith({MockitoExtension.class, MockValueExtension.class})
2 class MockValueExample2Test {
3
4     @InjectMocks
5     private Identity sut;
6
7     @MockValue("${identity-default-value}")
8     private String idDefaultValue = "id_123";
9
10    @Test
11    void shouldFindTheIdWhenInitializedWithMockValue() {
12        // Act / Assert
13        assertThat(sut.getId(), equalTo(idDefaultValue));
14    }
15 }
```

One of the coolest functionality brought by ***@MockValue*** and its extension, is the benefit of the @Value defaulting. It simply work in a last move wins mode. That means:

1. First we consider the code defaulting
2. Then the test case defaulting
3. and then if an explicit value is set to the property then it is the one which will be considered (injected).

Listing 18. Context: @Mock with defaulting.

```
private static class DummyTarget {

    @Value("${int_property:5}")
    private int intProperty;

    @Value("${long_property:30}")
    private long longProperty;

    @Value("${string_property:hello}")
    private String stringProperty;

    @Value("${int_array_property:236}")
    private int[] intArrayPropertyWithSingleElement;

    @Value("${long_array_property:78,56}")
    private long[] longArrayWithMultipleElements;

    int getIntProperty() {
        return intProperty;
    }

    long getLongProperty() {
        return longProperty;
    }

    String getStringProperty() {
        return stringProperty;
    }

    int[] getIntArrayPropertyWithSingleElement() {
        return intArrayPropertyWithSingleElement;
    }

    long[] getLongArrayWithMultipleElements() {
        return longArrayWithMultipleElements;
    }

}
```

Listing 19. MockValue defaulting example.

```
@MockValue(
    value = "${int_property:10}",
    testcase = "shouldInitAttributedWhenPropertyIsDefaulted"
)
private Integer defaultedIntegerProperty;

@Test
void shouldInitAttributedWhenPropertyIsDefaulted() {
    // Arrange / Act
    final Integer value = sut.getIntProperty();

    // Assert
    assertThat(value, notNullValue());
    assertThat(value, equalTo(10));
}
```

Mockito inclusion

You probably noted we have included Mockito *@InjectMocks* in the unit test. The benefit is to have the subject under test initialized before any other operation, and the custom extension will look for the field annotated with *@InjectMocks* and inject the desired values.

Extension Registration order

As per JUnit 5 user guide, the extensions registered via *@ExtendWith* will be executed in the order in which they are declared in the source code. So be aware of the interaction between extensions when using them together.

Please find the properties provided by *@MockValue* in the next table.

Table 2. @MockValue properties

Property	Description	Default value
`value`	An array of values to look for in the subject under test	
`testcase`	The test method where to apply the value	

 # value

The value is an array of String, that's define that should be initialized. Lets say we have the following entity:

Listing 20. Class with multiple properties annotated with @value.

```
1 public class Customer {
2
3     @Value("${customer-mother-name}")
4     private String motherName;
5
6     @Value("${customer-father-name}")
7     private String fatherName;
8
9     public String getMotherName() {
10        return motherName;
11    }
12
13    public String getFatherName() {
14        return fatherName;
15    }
16 }
```

When unit testing this class, we would like to initialize all the annotated properties with default value. With the traditional way, we need to set each property (via reflection for example). With *@MockValue#value* we simply declare all the target fields as show bellow.

Listing 21. @MockValue#value in action.

```
1 @ExtendWith({MockitoExtension.class, MockValueExtension.class})
2 class MockValueExample3Test {
3
4     @InjectMocks
5     private Customer sut;
6
7     @MockValue({"${customer-mother-name}", "${customer-father-name}"})
8     private String familyName = "no_name";
9
10    @Test
11    void shouldFindTheIdWhenInitializedWithMockValue() {
12        // Act / Assert
13        assertThat(sut.getFatherName(), equalTo(familyName));
14        assertThat(sut.getMotherName(), equalTo(familyName));
15    }
16 }
```

 # testcase

Some times we want to initialize the values for some test cases and not the other. We can achieve this with *@MockValue#testcase* as follows:

Listing 22. @MockValue#testcase in action.

```
@ExtendWith({MockitoExtension.class, MockValueExtension.class})
class MockValueExample4Test {

    @InjectMocks
    private Customer sut;

    @MockValue(
        value = {"${customer-mother-name}", "${customer-father-name}"},
        testcase = "shouldFindTheIdWhenInitializedWithMockValue"
    )
    private String familyName = "no_name";

    @Test
    void shouldFindTheIdWhenInitializedWithMockValue() {
        // Act / Assert
        assertThat(sut.getFatherName(), equalTo(familyName));
        assertThat(sut.getMotherName(), equalTo(familyName));
    }

    @Test
    void shouldNotInitPropertyWhenTestcaseNotMatching() {
        // Act / Assert
        assertThat(sut.getFatherName(), nullValue());
        assertThat(sut.getMotherName(), nullValue());
    }
}
```

2.3.2. *@MyBatisMapperTest*

One part of application development is oriented as well in data retrieval and manipulation from database. Mybatis mappers are being used for this purpose in many projects and as other part of the code, need to be tested. It is possible to apply the configuration manually, but ... of course to save the time we need to test and maintain the production code, this task has been automatized and standardized with the *@MyBatisMapperTest* annotation (and the relevant extension *MyBatisMapperExtension*, which registered by default by *@MyBatisMapperTest*_).

Listing 23. @MapperTest at method level.

```
@MyBatisMapperTest(script = "schema.sql")
void shouldUseSpecificConfigConfigWhenConfigAtMethodLevel() {
    // Arrange
    final int firstIdInDB = 1; // ID in 'data.sql'

    // Act
    final String name = sut.findCustomerNameById(firstIdInDB);

    // Assert
    assertThat(name, nullValue());

    // Mock not initialized as not declared in my batis config.
    assertThat(nonMapperBean, nullValue());

    final int itemCount = SessionFactoryUtils.countRowsInTableWhere(CUSTOMER,
ID_GREATHER_THAN_0);
    assertThat(itemCount, equalTo(0));
}
```

Full test class example

see full class in: *MyBatisMapperTestExample1Test.java*

Listing 24. @MyBatisMapperTest at class level.

```
@MyBatisMapperTest(script = {"schema.sql", "data.sql"})
class MyBatisMapperTestExample2Test {

    @InjectMapper
    private CustomerMapper sut;

    @Test
    void shouldInitMyBatisMapperWhenClassConfig() {
        // Act
        final String name = sut.findCustomerNameById(1);

        // Assert
        assertThat(name, equalTo("alpha"));
    }
}
```

Please find the properties provided by *@MyBatisMapperTest* in the next table.

.*@MyBatisMapperTest* properties

Property	Description	Default value
`script`	The SQL scripts to run when configuring the DataSource	
`testIsolation`	Manage the connection to the database. If `true` the database connection will be closed after each test method	`true`

2.3.3. *@InjectMapper*

As you probable noticed in previous example (*MyBatisMapperTestExample2Test*), the subject under test is *@InjectMapper* annotated. In fact, initializing MyBatis context is not enough, we still need to retrieve the target mapper we want to test. This is where *MyBatisMapperExtension* come into the game and inject the target mapper in the field annotated with *@InjectMapper*.

2.3.4. *@SpringContextRunner*

When the project we are involved in grows up, with many modules and classes, the project configuration as a critical should keep under control to avoid tricky issues.

Spring provide the *AbstractApplicationContextRunner* which is very well suited for configuration test. However, when using directly it, we face some minor issue like:

1. trouble to add a mock in the context
2. copy/paste the code in case of multiple testcase
3. not easy maintenance of the test afterwards

JUnit Utils provide the annotation *@SpringContextRunner* to solve the points listed above and helps by way to keep the testcase readable. Please find a use case of the *@SpringContextRunner* usage (together with the related extension *SpringContextRunnerExtension.class*).

Listing 25. Testing with SpringContextRunner

```
@ExtendWith({MockitoExtension.class, SpringContextRunnerExtension.class})
class SpringContextRunnerExample1Test {

    @SpringContextRunner(
        withUserConfiguration = ConfigUser.class
    )
    private ApplicationContextRunner appContextRunner;

    @BeforeEach
    void init() {
        assumeTrue(appContextRunner != null, "The <appContextRunner> cannot be
null");
    }

    @Test
    void shouldInjectMockDefinedInOutTestInTheApplicationContextWhenSimpleContext()
{
        // Arrange
        assumeTrue(appContextRunner != null, "the ApplicationContext is not
initialized");

        // Act
        appContextRunner.run(assertProvider -> {
            // Assert
            assertThat(assertProvider).doesNotHaveBean(AnnotatedComponent.class);

            // This entity is declared in the User configuration
            assertThat(assertProvider).hasSingleBean(EntityBrown.class);
        });
    }
}
```

You can denote here how easy it is to add a mock in the context. Please find the properties provided by *@SpringContextRunner* in the next table.

Table 3. @SpringContextRunner properties

Property	Description	Default value
`withAllowBeanDefinitionOverriding`	Allow bean definition overriding	true
`withPropertyValues`	Add the specified Environment property pairs	
`withSystemProperties`	Add the specified System property pairs	
`withConfiguration`	Register the specified configuration class with the ApplicationContext	None.class
`withUserConfiguration`	Register the specified user configuration classes with the ApplicationContext	

Property	Description	Default value
withBeans	Register the specified user beans with the ApplicationContext	
withMocks	Register the specified classes as mocks with the ApplicationContext	
injectDeclaredMocks	Register the mocks & spies declared in the test class	true
mappersPackage	Mock the mappers under given packages	

There another point where *@SpringContextRunner* will be very interesting for us. In fact, when it comes to mock something and/or declare a mock, we need to insert it again the the context. Now with the *@SpringContextRunner* we just need to declare the mock in the test for it to be injected in the context, as shown below:

Listing 26. Friendly mocks in @SpringContextRunner

```
@ExtendWith({MockitoExtension.class, SpringContextRunnerExtension.class})
class SpringContextRunnerExample1Test {

    @SpringContextRunner(
        withUserConfiguration = ConfigUser.class
    )
    private ApplicationContextRunner appContextRunner;

    @BeforeEach
    void init() {
        assumeTrue(appContextRunner != null, "The <appContextRunner> cannot be
null");
    }

    @Test
    void shouldInjectMockDefinedInOutTestInTheApplicationContextWhenSimpleContext()
{
        // Arrange
        assumeTrue(appContextRunner != null, "the ApplicationContext is not
initialized");

        // Act
        appContextRunner.run(assertProvider -> {
            // Assert
            assertThat(assertProvider).doesNotHaveBean(AnnotatedComponent.class);

            // This entity is declared in the User configuration
            assertThat(assertProvider).hasSingleBean(EntityBrown.class);
        });
    }
}
```

2.3.5. *@Fixture*

This is a class and field level annotation, that's enable *JUnit Utils* with powerful fixture files. As you may already know, a fixture is a standalone unit or functionality of a project. It is usually best practice, when testing that before deriving the test scripts, we should determine the fixture to be tested. Such a fixture usually contains a list of steps, which, combined in a given sequence, give a test case.

In context of *JUnit Utils*, the fixture file contains the atomic steps that can be applied to prepare a test case execution. Please take a look of an example of fixture file.

Listing 27. An example of @Fixture file.

```
@Fixture
public class FixtureFileExample {

    private Report report;

    void givenCustomerIdIsTwenty() {
        if (MockUtil.isMock(report)) {
            when(report.getCustomerId()).thenAnswer(arg -> 20);
        }
    }
}
```

Having already defined our fixture file, can therefore use it in our unit test as show below.

Listing 28. @Fixture file in action.

```
@ExtendWith({MockitoExtension.class, FixtureExtension.class})
class FixtureExample1Test {

    @Mock
    private Report reportMock;

    @Fixture
    private FixtureFileExample fixture;

    @Test
    void shouldHaveAConfiguredFixtureFile() {
        // Arrange
        fixture.givenCustomerIdIsTwenty();

        // Act
        final Integer id = reportMock.getCustomerId();

        // Assert
        assertThat(id, equalTo(20));
    }
}
```

As you already noticed, we are free of initializing the fixture file by injecting the mocks from our test class. Indeed, the mocks (and spies) declared in the test class are injected in the fixture, if the relevant field hasn't yet been initialized. Doing so help to keep cleaner and readable test and the developer can focus on the functionality to test.

The table below shows *@Fixture* annotation the properties.

Table 4. @Fixture properties

Property	Description	Default value
`injectMocks`	Inject test class (test case) mocks to the class with *@Fixture* or not	`true`

2.3.6. *@UnitTest*

Having all the previous features up and running, some behaviour has been defaulted to simplify unit test design, plan, implementation and configuration. The *@UnitTest* annotation, in addition to emphasizing the fact our test is a unit test, is setting in the main time de most common default behavior by registering the following extension (in order):

- MockitoExtension
- ReturnsMocksExtension — *[#1] ensure the created mock will return the test instance declared @Mock/@Spy*
- MockValueExtension — *[#2] initialize the @Value (via @MockValue) when relevant*
- ReinforceMockExtension — *[#3] enrich the declared mocks to avoid NPE on lifecycle methods call*
- MockitoSpyExtension — *[#4] enrich the @Spy fields with the previous (#1 and #2) knowledge*
- MockInjectionExtension — *[#5] ensure the SUT (@InjectMocks) is properly initialized ...*
- UnitTestExtension (log test start/stop)
- BeanLifeCycleExtension
- FixtureExtension
- MocksContextParameterResolver

Please find the properties provided by *@UnitTest* in the next table.

Table 5. @UnitTest properties

Property	Description	Default value
`strictness`	The MockitoSettings strictness	STRICT_STUBS
`tags`	The value to be set as junit Tag	'unit_test'
`returnMocks`	Answer (*Mockito.Answers*) with mocks defined in user test (see: *ReturnsMocksExtension*)	`true`
`initMocks`	The MockitoSettings strictness	`true`
`initSpies`	The flag to init mockito spies (@Spy) with test case mock and spies if relevant	`true`

Property	Description	Default value
`lifeCycle`	The flag to enable/disable the bean lifecycle method execution	`true`
`reinforceMock`	The flag to enable/disable the mock reinforcement	`true`

Listing 29. Testing with @UnitTest

```
@UnitTest
class UnitTestExample1Test {

    @InjectMocks
    private Report sut;

    @Mock(answer = Answers.RETURNS_MOCKS)
    private Customer customerMock;

    @Mock
    private Identity identityMock;

    @Test
    void shouldEnsureThatIdentityIsRecognizedInTestcase() {
        // Arrange
        final Integer id = 456;
        when(identityMock.getId()).thenReturn(id);
        // Act
        final Integer result = sut.getCustomerId();

        // Asset
        assertThat(result, equalTo(id));
    }
}
```

2.4. Other delicacies

2.4.1. FileArgumentMatchers

Mockito provides arguments matchers that make it possible to mocks and verifiers to answer a wider range of values and unknown values. *FileArgumentMatchers* comes as a support for the existing arguments matchers, to make it possible for mocks and verifiers to answer in case of *File* and *Path* arguments.

- **anyFile:** Shortcut for argument matcher *any(File.class)*
- **fileThat:** Customized *File* argument matcher, compatible with Java 8 lambdas
- **anyPath:** Shortcut for argument matcher *any(Path.class)*
- **pathThat:** Customized *Path* argument matcher, compatible with Java 8 lambdas

To illustrate those arguments matchers usage, lets set the following context:

Listing 30. The arguments matchers context.

```
1 public class CustomerFile {
2
3     /**
4      * Return the string representation of, if the file exists or not.
5      * @param file The file
6      * @return the "true"/"false" string.
7      */
8     public String readCustomerFile(final File file) {
9         return Boolean.toString(file != null && file.exists());
10    }
11
12    /**
13     * Return the string representation of, if the path exists or not.
14     * @param path The path
15     * @return the "true"/"false" string.
16     */
17    public String readCustomerPath(final Path path) {
18        return Boolean.toString(path != null && path.toFile().exists());
19    }
20 }
```

 # anyFile(...) and fileThat(...)

Creates an argument matcher that matches if the examined object is a file. Please see in the listing below an example of *anyFile()* and *fileThat(...)* usage.

Listing 31. anyFile() and fileThat(...) in action

```
@ExtendWith(MockitoExtension.class)
class FileArgumentMatchersExample1Test {

    private static final List<String> MY_BOOLEANS = Arrays.asList("false", "true");

    @Mock
    private CustomerFile sut;

    @Test
    void shouldConfirmMockRespondsProperlyWhenAnyFileMatcherIsUsed() {
        // Arrange
        final String filename = "dummyFile.txt";
        final String anyFileExpectedAnswer = "anyFileAnswer";
        final String fileThatExpectedAnswer = "fileThatAnswer";

        when(sut.readCustomerFile(anyFile())).thenReturn(anyFileExpectedAnswer);

        when(sut.readCustomerFile(
            fileThat(file -> file.getName().equals(filename)))
        ).thenReturn(fileThatExpectedAnswer);

        // Act
        final String expectationOne = sut.readCustomerFile(new File("any"));
        assumeFalse(MY_BOOLEANS.contains(expectationOne));
        assumeTrue(expectationOne.equals(anyFileExpectedAnswer));

        final String expectationTwo = sut.readCustomerFile(new File(filename));
        assumeFalse(MY_BOOLEANS.contains(expectationTwo));
        assumeTrue(expectationTwo.equals(fileThatExpectedAnswer));

        // Assert
        verify(sut, times(2)).readCustomerFile(anyFile());
        verify(sut).readCustomerFile(fileThat(f -> f.getName().equals(filename)));
        verify(sut, never()).readCustomerPath(anyPath());
    }
}
```

 # anyPath() and pathThat(...)

Creates an argument matcher that matches if the examined object is a path. Please see in the listing below an example of *anyPath()* and *pathThat(...)* usage.

Listing 32. anyPath() and pathThat(...) in action

```
@ExtendWith(MockitoExtension.class)
class FileArgumentMatchersExample1Test {

    private static final List<String> MY_BOOLEANS = Arrays.asList("false", "true");

    @Mock
    private CustomerFile sut;

    @Test
    void shouldConfirmMockRespondsProperlyWhenAnyFileMatcherIsUsed() {
        // Arrange
        final String filename = "dummyFile.txt";
        final String anyFileExpectedAnswer = "anyFileAnswer";
        final String fileThatExpectedAnswer = "fileThatAnswer";

        when(sut.readCustomerFile(anyFile())).thenReturn(anyFileExpectedAnswer);

        when(sut.readCustomerFile(
            fileThat(file -> file.getName().equals(filename)))
        ).thenReturn(fileThatExpectedAnswer);

        // Act
        final String expectationOne = sut.readCustomerFile(new File("any"));
        assumeFalse(MY_BOOLEANS.contains(expectationOne));
        assumeTrue(expectationOne.equals(anyFileExpectedAnswer));

        final String expectationTwo = sut.readCustomerFile(new File(filename));
        assumeFalse(MY_BOOLEANS.contains(expectationTwo));
        assumeTrue(expectationTwo.equals(fileThatExpectedAnswer));

        // Assert
        verify(sut, times(2)).readCustomerFile(anyFile());
        verify(sut).readCustomerFile(fileThat(f -> f.getName().equals(filename)));
        verify(sut, never()).readCustomerPath(anyPath());
    }
}
```

2.4.2. DataSourceMockBuilder

This builder has been introduced to provide a ready to go mocked DataSource. Up to the developer to declare and inject it in the test context, the spring context or event to use it directly in the test class.

Listing 33. Mocked DataSource as a Bean

```
@Configuration
class DataSourceMockExample1Test {
    @Bean
    @Primary
    public DataSource dataSource() {
        return DataSourceMockBuilder.newDataSourceMock("oracle").build();
    }
}
```

Listing 34. Verifying the mocked DataSource properties

```
@Configuration
class DataSourceMockExample1Test {

    @Test
    void shouldBuildH2DataSourceMockWhenCallingH2Builder() throws SQLException {
        // Act
        final DataSource dataSource = DataSourceMockBuilder.newH2Mock().build();

        // Assert
        assertThat(dataSource, notNullValue());
        assertThat(MockUtil.isMock(dataSource), equalTo(true));
        assertThat(MockUtil.getMockSettings(dataSource).getDefaultAnswer(),
  equalTo(Answers.RETURNS_DEEP_STUBS));

        final Connection connection = dataSource.getConnection();
        assertThat(MockUtil.isMock(connection), equalTo(true));

        final DatabaseMetaData databaseMetaData = connection.getMetaData();
        assertThat(databaseMetaData.getDatabaseProductName(), equalTo("H2"));
        assertThat(MockUtil.isMock(databaseMetaData), equalTo(true));
    }
}
```

Chapter 3. Future works

There are still many subject uncovered in the *JUnit Utils* framework and in this user guide as well, that are in the area of interest of most of the developers. Including some tools for integration tests are in the bags as well, as integration with tools like *Cucumber* and *Selenium*, when it comes to integration tests.

Chapter 4. Last thought

The solutions and features presented in this user guide presents a the general approach we adapted to improve the tests quality. We hope they will be helpful to solve impediment software developer are facing during coding and testing. In case they do not fit to your project, do not hesitate to contribute with new ideas and propositions.

Happy coding !