Guia de JUnit5

Raúl Estrada

Octubre 2020

1. Overview

JUnit is one of the most popular unit-testing frameworks in the Java ecosystem. The JUnit 5 version contains a number of exciting innovations, with **the goal to support new features in Java 8 and above**, as well as enabling many different styles of testing.

2. Maven Dependencies

Setting up JUnit 5.x.0 is pretty straightforward, we need to add the following dependency to our *pom.xml*:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.1.0</version>
    <scope>test</scope>
</dependency>
```

It is important to note that this version **requires Java 8 to work**.

What's more, there is now direct support to run Unit tests on the JUnit Platform in Eclipse as well as IntelliJ. You can, of course, also run tests using the Maven Test goal.

On the other hand, IntelliJ supports JUnit 5 by default. Therefore, running JUnit 5 on IntelliJ is pretty simple, simply Right click –> Run, or Ctrl-Shift-F10.

3. Architecture

JUnit 5 is composed of several different modules from three different sub-projects:

3.1. JUnit Platform

The platform is responsible for launching testing frameworks on the JVM. It defines a stable and powerful interface between JUnit and its client such as build tools.

The final objective is how its clients get integrated easily with JUnit in discovering and executing the tests.

It also defines the TestEngine API for developing a testing framework that runs on the JUnit platform. By that, you can plug-in 3rd party testing libraries, directly into JUnit, by implementing custom TestEngine.

3.2. JUnit Jupiter

This module includes new programming and extension models for writing tests in JUnit 5. New annotations in comparison to JUnit 4 are:

- *@TestFactory* – denotes a method that is a test factory for dynamic tests
- *@DisplayName* – defines custom display name for a test class or a test method
- *@Nested* – denotes that the annotated class is a nested, non-static test class
- *@Tag* – declares tags for filtering tests
- *@ExtendWith* – it is used to register custom extensions
- *@BeforeEach* – denotes that the annotated method will be executed before each test method (previously *@Before*)
- *@AfterEach* – denotes that the annotated method will be executed after each test method (previously *@After*)
- *@BeforeAll* – denotes that the annotated method will be executed before all test methods in the current class (previously *@BeforeClass*)
- *@AfterAll* – denotes that the annotated method will be executed after all test methods in the current class (previously *@AfterClass*)
- *@Disable* – it is used to disable a test class or method (previously *@Ignore*)

3.3. JUnit Vintage

Supports running JUnit 3 and JUnit 4 based tests on the JUnit 5 platform.

4. Basic Annotations

To discuss new annotations, we divided the section into the following groups, responsible for execution: before the tests, during the tests (optional) and after the tests:

4.1. *@BeforeAll* and *@BeforeEach*

Below is an example of the simple code to be executed before the main test cases:

```
@BeforeAll
static void setup() {
    log.info("@BeforeAll - executes once before all test methods in this class");
}

@BeforeEach
void init() {
    log.info("@BeforeEach - executes before each test method in this class");
}
```

Important to note is that the method with *@BeforeAll* annotation needs to be static, otherwise the code will not compile.

4.2. *@DisplayName* and *@Disabled*

Let's move to new test-optional methods:

```
@DisplayName("Single test successful")
@Test
void testSingleSuccessTest() {
    log.info("Success");
}

@Test
@Disabled("Not implemented yet")
void testShowSomething() {
}
```

As we can see, we may change display name or to disable the method with a comment, using new annotations.

4.3. *@AfterEach* and *@AfterAll*

Finally, let's discuss methods connected to operations after tests execution:

```
@AfterEach
void tearDown() {
    log.info("@AfterEach - executed after each test method.");
}

@AfterAll
static void done() {
    log.info("@AfterAll - executed after all test methods.");
}
```

Please note that method with *@AfterAll* needs also to be a static method.

5. Assertions and Assumptions

JUnit 5 tries to take full advantage of the new features from Java 8, especially lambda expressions.

5.1. Assertions

Assertions have been moved to *org.junit.jupiter.api.Assertions* and have been improved significantly. As mentioned earlier, you can now use lambdas in assertions:

```
@Test
void lambdaExpressions() {
    assertTrue(Stream.of(1, 2, 3)
      .stream()
      .mapToInt(i -> i)
      .sum() > 5, () -> "Sum should be greater than 5");
}
```

Although the example above is trivial, one advantage of using the lambda expression for the assertion message is that it is lazily evaluated, which can save time and resources if the message construction is expensive.

It is also now possible to group assertions with *assertAll()* which will report any failed assertions within the group with a *MultipleFailuresError*.

```
@Test
void groupAssertions() {
    int[] numbers = {0, 1, 2, 3, 4};
    assertAll("numbers",
        () -> assertEquals(numbers[0], 1),
        () -> assertEquals(numbers[3], 3),
        () -> assertEquals(numbers[4], 1)
    );
}
```

This means it is now safer to make more complex assertions, as you will be able to pinpoint the exact location of any failure.

5.2. Assumptions

Assumptions are used to run tests only if certain conditions are met. This is typically used for external conditions that are required for the test to run properly, but which are not directly related to whatever is being tested.

You can declare an assumption with *assumeTrue()*, *assumeFalse()*, and *assumingThat()*.

```java
@Test
void trueAssumption() {
    assumeTrue(5 > 1);
    assertEquals(5 + 2, 7);
}

@Test
void falseAssumption() {
    assumeFalse(5 < 1);
    assertEquals(5 + 2, 7);
}

@Test
void assumptionThat() {
    String someString = "Just a string";
    assumingThat(
        someString.equals("Just a string"),
        () -> assertEquals(2 + 2, 4)
    );
}
```

If an assumption fails, a *TestAbortedException* is thrown and the test is simply skipped.

Assumptions also understand lambda expressions.

6. Exception Testing

There are two ways of exception testing in JUnit 5. Both of them can be implemented by using *assertThrows()* method:

```
@Test
void shouldThrowException() {
    Throwable exception = assertThrows(UnsupportedOperationException.class, () -> {
      throw new UnsupportedOperationException("Not supported");
    });
    assertEquals(exception.getMessage(), "Not supported");
}

@Test
void assertThrowsException() {
    String str = null;
    assertThrows(IllegalArgumentException.class, () -> {
      Integer.valueOf(str);
    });
}
```

The first example is used to verify more detail of the thrown exception and the second one just validates the type of exception.

7. Test Suites

To continue the new features of JUnit 5, we will try to get to know the concept of aggregating multiple test classes in a test suite so that we can run those together. JUnit 5 provides two annotations: *@SelectPackages* and *@SelectClasses* to create test suites.

Keep in mind that at this early stage most IDEs do not support those features.

Let's have a look at the first one:

```
1 @RunWith(JUnitPlatform.class)
2 @SelectPackages("com.baeldung")
3 public class AllUnitTest {}
```

@SelectPackage is used to specify the names of packages to be selected when running a test suite. In our example, it will run all test. The second annotation, *@SelectClasses*, is used to specify the classes to be selected when running a test suite:

```
@RunWith(JUnitPlatform.class)
@SelectClasses({AssertionTest.class, AssumptionTest.class, ExceptionTest.class})
public class AllUnitTest {}
```

For example, above class will create a suite contains three test classes. Please note that the classes don't have to be in one single package.

8. Dynamic Tests

The last topic that we want to introduce is JUnit 5 Dynamic Tests feature, which allows to declare and run test cases generated at run-time. In contrary to the Static Tests which defines fixed number of test cases at the compile time, the Dynamic Tests allow us to define the tests case dynamically in the runtime.

Dynamic tests can be generated by a factory method annotated with *@TestFactory*. Let's have a look at the code example:

```
@TestFactory
public Stream<DynamicTest> translateDynamicTestsFromStream() {
    return in.stream()
      .map(word ->
          DynamicTest.dynamicTest("Test translate " + word, () -> {
            int id = in.indexOf(word);
            assertEquals(out.get(id), translate(word));
          })
    );
}
```

This example is very straightforward and easy to understand. We want to translate words using two *ArrayList*, named *in* and *out*, respectively. The factory method must return a *Stream*, *Collection*, *Iterable*, or *Iterator*. In our case, we choose Java 8 *Stream*.

Please note that *@TestFactory* methods must not be private or static. The number of tests is dynamic, and it depends on the *ArrayList* size.

9. Conclusion

The write-up was a quick overview of the changes that are coming with JUnit 5.

We can see that JUnit 5 has a big change in its architecture which related to platform launcher, integration with build tool, IDE, other Unit test frameworks, etc. Moreover, JUnit 5 is more integrated with Java 8, especially with Lambdas and Stream concepts.