Baeldung

Move on to JUnit 5

Contents

1. Introduction

In this ebook, following the A Guide to JUnit 5 tutorial, we'll see how we can migrate our test codebase from using JUnit 4 to the latest JUnit 5 release.

We'll start looking into why we should migrate to JUnit 5, exploring its advantages, and then we'll see the different activities that should be performed to solve compatibility issues and take advantage of the new features in JUnit 5.

2. JUnit 5 Advantages

JUnit 5, following JUnit 4, was written with the idea to overcome a few limitations present in the previous version:

- The entire framework was contained in a single jar library. The whole library needs to be imported even when only a single feature is required. **In JUnit 5, we get more granularity and can import only what is necessary.**

- One test runner can only execute tests in JUnit 4 at a time (e.g., *SpringJUnit4ClassRunner or Parameterized*). **JUnit 5 allows multiple runners to work simultaneously**.

- JUnit 4 never advanced beyond Java 7, missing out on a lot of features from Java 8. **JUnit 5 makes good use of Java 8 features**.

Let's start looking at how the library is organized and what we need to import and for which purpose.

3. Dependencies

To overcome the fact that the entire framework was contained in a single *jar* library, JUnit 5 is composed of 3 main modules:

- **Platform -** serves as a foundation for launching a testing framework on the JVM and defines the *TestEngine* API for developing a testing framework that runs on the platform

- **Jupiter -** includes the new programming model for writing tests and the extension model for writing extensions in JUnit 5

- **Vintage -** provides a *TestEngine* that allows backward compatibility with JUnit 4 or even JUnit 3

Before starting to configure our JUnit 5 dependency, we have to remember that this version of the library **requires Java 8 to work.**

3.1. Configure JUnit 5

To start using JUnit 5, we can 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>
```

or to our *build.gradle* file:

```
testCompile('org.junit.jupiter:junit-jupiter-api:5.2.0')
testRuntime('org.junit.jupiter:junit-jupiter-engine:5.2.0')
```

3.2. Configure JUnit Vintage

In order to make the migration of the JUnit tests less painful, we can configure JUnit Vintage, that allows us to run JUnit 3 or JUnit 4 tests within the JUnit 5 context.

We can use it by simply configuring our *pom.xml*:

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <version>5.2.0</version>
    <scope>test</scope>
</dependency>
```

or our *build.gradle* file:

```
testRuntime('org.junit.jupiter:junit-vintage-engine:5.2.0')
```

4. Imports

Due to the different structure of the library, the first step that we have to complete is to replace the imports of old versions with the new ones.

First, we can replace all:

```
1.  import org.junit.Test;
```

with

```
1.  import org.junit.jupiter.api.*;
```

Then, we can replace all the imports related to the assertions:

```
1.  import static org.junit.Assert.*;
```

with

```
1.  import static org.junit.jupiter.api.Assertions.*;
```

Since all the other imports are deprecated, due to the several changes in the new version of the library, we can safely remove them.

The new version of the library introduced few changes to the different annotations that were available in JUnit 4. We'll start looking into the basic annotations.

5.1. Before Annotations

For configuring tests, there are a few changes to the basic annotation when performing the migration:

- The ***@Before* annotation was replaced by the new *@BeforeEach*** one, that denotes that a method should be executed before each *@Test*, *@RepeatedTest*, *@ParameterizedTest*, or *@TestFactory* method in the current class

- The ***@BeforeClass* annotation gets replaced by the new *@BeforeAll one***, that denotes that a method should be executed before the method annotated with *@Before*

With these changes, if we had configured a test class with the JUnit 4 annotations:

```
1.  @BeforeClass
2.  static void setup() {
3.      log.info("@BeforeClass - executes once before all test methods in this class");
4.  }
5.  @Before
6.  void init() {
7.      log.info("@Before - executes before each test method in this class");
8.  }
```

We'll have to change them accordingly:

```
1.  @BeforeAll
2.  static void setup() {
3.      log.info("@BeforeAll - executes once before all test methods in this
4.  class");
5.  }
6.  @BeforeEach
7.  void init() {
8.      log.info("@BeforeEach - executes before each test method in this class");
9.  }
```

5.2. After Annotations

In the same way as the *@Before* and the *@BeforeClass* annotations, the new version of the library removed the *@After* and the *@AfterClass* annotations:

- The ***@After annotation gets replaced by the new @AfterEach*** one, that denotes that a method should be executed after each *@Test*, *@RepeatedTest*, *@ParameterizedTest*, or *@TestFactory* method in the current class
- The ***@AfterClass annotation gets replaced by the new @AfterAll*** one, that denotes that a method should be executed after the method annotated with *@After*

Taking into considerations these changes, if we had configured a test class with the JUnit 4 annotations:

```
1.  @After
2.  void tearDown() {
3.      log.info("@After - executed after each test method.");
4.  }
5.
6.  @AfterClass
7.  static void done() {
8.      log.info("@AfterClass - executed after all test methods.");
9.  }
```

We'll have to change them accordingly:

```
1.  @AfterEach
2.  void tearDown() {
3.      log.info("@AfterEach - executed after each test method.");
4.  }
5.
6.  @AfterAll
7.  static void done() {
8.      log.info("@AfterAll - executed after all test methods.");
9.  }
```

5.3. Test Expectations

Considering the fact that JUnit 5 comes with important changes within its annotations, the most important one is that ***we can no longer use @Test annotation for specifying expectations.***

The *expected* parameter in JUnit 4:

```
@Test(expected = Exception.class)
public void shouldRaiseAnException() throws Exception {
    // ...
}
```

Now, we have to use the new assertion method *assertThrows*:

```
public void shouldRaiseAnException() throws Exception {
    Assertions.assertThrows(Exception.class, () -> {
        //...
    });
}
```

The *timeout* attribute in JUnit 4:

```
@Test(timeout = 1)
public void shouldFailBecauseTimeout() throws InterruptedException {
    Thread.sleep(10);
}
```

Now, we have to use the new assertion *assertTimeout* in JUnit 5:

```
@Test
public void shouldFailBecauseTimeout() throws InterruptedException {
 Assertions.assertTimeout(Duration.ofMillis(1), () -> Thread.sleep(10));
}
```

5.4. Disabling Tests

JUnit 5 no longer supports the *@Ignore* annotation, which was used in JUnit 4 to skip the execution of a specific test or a group of tests.

In its place, to disable a test or a test suite, we have to use the *@Disabled* annotation:

```
1. @Test
2. @Disabled
3. void disabledTest() {
4.     assertTrue(false);
5. }
```

We can apply this annotation, as the *@Ignore* one, both to a method and to test classes.

With the new version of the library, **the package for the assertions changed from *org.junit.Assert to org.junit.jupiter.api.Assertions.***

However, JUnit 5 kept many of the assertion methods of JUnit 4 while adding a few new ones that took advantage of the Java 8 support.

As in the previous versions, the different assertions are available for all primitive types, *Objects*, and arrays (either of primitives or Objects).

6.1. Assertion Messages

One of the major breaking changes for the *Assertions* is that **the order of the parameters changed with the output message parameter moved as the last parameter.**

For this reason, if we have specified a message error for an assertion:

```
1.  @Test
2.  public void whenAssertingArraysEquality_thenEqual() {
3.      char[] expected = { 'J', 'u', 'n', 'i', 't' };
4.      char[] actual = "JUnit".toCharArray();
5.
6.      assertArrayEquals("Arrays should be equal", expected, actual);
7.  }
```

We have to move it accordingly:

```
1.  @Test
2.  public void whenAssertingArraysEquality_thenEqual() {
3.      char[] expected = { 'J', 'u', 'n', 'i', 't' };
4.      char[] actual = "JUnit".toCharArray();
5.
6.      assertArrayEquals(expected, actual, "Arrays should be equal");
7.  }
```

6.2. Messages Supplier

Thanks to the support of Java 8, the output message can be a *Supplier*, allowing lazy evaluation of it.

In case we want to enhance our test methods during the migration, we can replace the *String* messages with *Supplier*:

```
1.  @Test
2.  public void whenAssertingArraysEquality_thenEqual() {
3.      char[] expected = { 'J', 'u', 'n', 'i', 't' };
4.      char[] actual = "JUnit".toCharArray();
5.
6.     assertArrayEquals(expected, actual, () -> "Arrays should be equal");
7.  }
```

6.3. Grouping Assertions

One of the issues of JUnit 4 was that all the assertions were performed sequentially and if one of the assertions failed, those after the failure weren't executed.

```
1.  @Test
2.  public void givenMultipleAssertion_whenAssertingAll_thenOK() {
3.      assertEquals("4 is 2 times 2", 4, 2 * 2);
4.      assertEquals("java", "JAVA".toLowerCase());
5.      assertEquals("null is equal to null", null, null);
6.  }
```

JUnit 5 solved this problem with the introduction of the new assertion *assertAll*.

This assertion allows the creation of grouped assertions, where all the assertions are executed, and their failures are reported together.

In details, this assertion accepts a heading, that will be included in the message *String* for the *MultipleFailureError*, and a *Stream of Executable*.

Let's see how we can define a grouped assertion:

```
@Test
public void givenMultipleAssertion_whenAssertingAll_thenOK() {
    assertAll(
      "heading",
      () -> assertEquals(4, 2 * 2, "4 is 2 times 2"),
      () -> assertEquals("java", "JAVA".toLowerCase()),
      () -> assertEquals(null, null, "null is equal to null")
    );
}
```

When one of the executables throws a blacklisted exception (*OutOfMemoryError for example*), it interrupts the execution of the grouped assertion.

6.4. Deprecated Assertions

The new version of the library removed a few deprecated assertions that were available within JUnit 4:

```
assertEquals(String message, double expected, double actual)
assertEquals(String message, Object[] expecteds, Object[] actuals)
```

However, we can still use the other two deprecated assertions from JUnit 4, since they are still available in JUnit 5 and are no longer marked as deprecated:

```
assertEquals(double expected, double actual)
assertEquals(Object[] expecteds, Object[] actuals)
```

6.5. Migrate AssertThat

It's worth noting that the *assertThat* assertion is no longer available in JUnit 5.

So if we used this assertion in a test method, this wouldn't compile:

```
1.  @Test
2.  public void testAssertThatHasItems() {
3.      assertThat(
4.        Arrays.asList("Java", "Kotlin", "Scala"),
5.        hasItems("Java", "Kotlin"));
6.  }
```

However, we don't have to rewrite the test method, because what we can do instead is use the *assertThat* provided by the Hamcrest testing library.

Therefore, what we need to do is simply change the imports from:

```
1.  import static org.junit.Assert.assertThat;
```

to:

```
1.  import static org.hamcrest.MatcherAssert.assertThat;
```

Additional information on the use of the *assertThat* assertion with Matcher object, is available at Testing with Hamcrest.

7. Assumption

When we want to run tests only if certain conditions are met, we can use assumptions. 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.

The new ***Assumptions*** **class is now in** ***org.junit.jupiter.api.Assumptions*****, instead of** ***org.junit.Assume.***

The new *Assumptions* class supports only a few of the assumptions that were available with JUnit 4: *assumeTrue, assumeFalse*. A new one was introduced that is the *assumingThat*, while the others (*assumeNoException*, *assumeNotNull* and *assumeThat*) weren't kept and are therefore not available in the new version of the library.

In case we have used the *assumeTrue* and *assumeFalse*, there are no issues when upgrading to JUnit 5 unless an output message was specified:

```
1.  @Test
2.  public void trueAssumption() {
3.      assumeTrue("5 is greater the 1", 5 > 1);
4.      assertEquals(5 + 2, 7);
5.  }
6.
7.  @Test
8.  public void falseAssumption() {
9.      assumeFalse("5 is less then 1", 5 < 1);
10.     assertEquals(5 + 2, 7);
11. }
```

In JUnit 4 we could group tests by using the *@Category* annotation.
With JUnit 5, we have to replace the *@Category* annotation with the *@Tag* annotation:

```
@Tag("annotations")
@Tag("junit5")
@RunWith(JUnitPlatform.class)
public class AnnotationTestExampleUnitTest {
    /*...*/
}
```

We can include/exclude particular tags using the *maven-surefire-plugin*:

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <configuration>
                <properties>
                    <includeTags>junit5</includeTags>
                </properties>
            </configuration>
        </plugin>
    </plugins>
</build>
```

9. Display Names

Thanks to the addition of the new annotation *@DisplayName* in JUnit 5, we can declare a custom display name on test classes and methods, that the test runners and test reporting will display:

```
@DisplayName("Test case for assertions")
public class AssertionUnitTest {

    @Test
    @DisplayName("Arrays should be equals")
    public void whenAssertingArraysEquality_thenEqual() {
        char[] expected = {'J', 'u', 'p', 'i', 't', 'e', 'r'};
        char[] actual = "Jupiter".toCharArray();

        assertArrayEquals(expected, actual, "Arrays should be equal");
    }

    @Test
    @DisplayName("The area of two polygons should be equal")
    public void whenAssertingEquality_thenEqual() {
        float square = 2 * 2;
        float rectangle = 2 * 2;

        assertEquals(square, rectangle);
    }
}
```

This single annotation allows us to improve the tests reporting, writing more verbose and human-readable output.

With the introduction of nested tests, we can express complex relationships between different groups of tests. The syntax is quite straightforward – all we have to do is to annotate an inner class with *@Nested*.

With this new annotation, let's see how it's possible to create a hierarchy of tests executions:

```
1.  public class NestedUnitTest {
2.
3.      Stack<Object> stack;
4.
5.      @Test
6.      @DisplayName("is instantiated with new Stack()")
7.      void isInstantiatedWithNew() {
8.          new Stack<>();
9.      }
10.
11.     @Nested
12.     @DisplayName("when new")
13.     class WhenNew {
14.         @BeforeEach
15.         void init() {
16.             stack = new Stack<>();
17.         }
18.         @Test
19.         @DisplayName("is empty")
20.         void isEmpty() {
21.             Assertions.assertTrue(stack.isEmpty());
22.         }
23.         @Test
24.         @DisplayName("throws EmptyStackException when popped")
25.         void throwsExceptionWhenPopped() {
26.             assertThrows(EmptyStackException.class, () -> stack.pop());
27.         }
28.         @Test
29.         @DisplayName("throws EmptyStackException when peeked")
30.         void throwsExceptionWhenPeeked() {
31.             assertThrows(EmptyStackException.class, () -> stack.peek());
32.         }
```

```
37.         @Nested
38.         @DisplayName("after pushing an element")
39.         class AfterPushing {
40.
41.             String anElement = "an element";
42.
43.             @BeforeEach
44.             void init() {
45.                 stack.push(anElement);
46.             }
47.
48.             @Test
49.             @DisplayName("it is no longer empty")
50.             void isEmpty() {
51.                 Assertions.assertFalse(stack.isEmpty());
52.             }
53.
54.             @Test
55.             @DisplayName("returns the element when popped and is empty")
56.             void returnElementWhenPopped() {
57.                 Assertions.assertEquals(anElement, stack.pop());
58.                 Assertions.assertTrue(stack.isEmpty());
59.             }
60.
61.             @Test
62.                @DisplayName("returns the element when peeked but remains not empty")
63.             void returnElementWhenPeeked() {
64.                 Assertions.assertEquals(anElement, stack.peek());
65.                 Assertions.assertFalse(stack.isEmpty());
66.             }
67.         }
68.     }
69. }
```

With this kind of structure, the output will be hierarchical, expressing the structure created with the nesting.

With the introduction of the *ExecutionCondition* in JUnit 5, that defines the *Extension* API for programmatic conditional test execution, it's possible to either enable or disable the execution of a test class or a test method, based on certain conditions programmatically.

In case we define multiple *ExecutionCondition* extensions for a test class or method, we disable the execution of the test method as soon as one of the conditions returns disabled.

Let's start looking at the Operating System Conditions.

11.1. Operating System Conditions

We can decide to enable or disable a test class or a method based on a particular operating system, using the *@EnableOnOs* and *@DisableOnOs.*

Let's see how we can use these annotations:

```
1.  @Test
2.  @EnabledOnOs({ OS.MAC })
3.  void whenOperatingSystemIsMac_thenTestIsEnabled() {
4.      assertEquals(5 + 2, 7);
5.  }
6.
7.  @Test
8.  @DisabledOnOs({ OS.WINDOWS })
9.  void whenOperatingSystemIsWindows_thenTestIsDisabled() {
10.     assertEquals(5 + 2, 7);
11. }
```

11.2. Java Runtime Environment Conditions

We can enable or disable a test class or method depending on a particular version of the Java Runtime Environment (JRE), using the *@EnableOnJre* and *@DisableOnJre*. Let's see how we can use these annotations to enable/disable tests for Java 8 and one Java 9:

```
1.  @Test
2.  @EnabledOnJre({ JRE.JAVA_8 })
3.  void whenRunningTestsOnJRE8_thenTestIsEnabled() {
4.      assertTrue(5 > 4, "5 is greater the 4");
5.      assertTrue(null == null, "null is equal to null");
6.  }
7.
8.  @Test
9.  @DisabledOnJre({ JRE.JAVA_10})
10. void whenRunningTestsOnJRE10_thenTestIsDisabled() {
11.     assertTrue(5 > 4, "5 is greater the 4");
12.     assertTrue(null == null, "null is equal to null");
13. }
```

11.3. System Property Conditions

In case we want to enable or disable a test class or method based on the value of a named JVM system property, we can respectively use the *@EnabledIfSystemProperty* and *@DisabledIfSystemProperty* annotations:

```
1.  @Test
2.  @EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
3.  public void whenRunningTestsOn64BitArchitectures_thenTestIsDisabled() {
4.      Integer value = 5; // result of an algorithm
5.
6.      assertNotEquals(0, value, "The result cannot be 0");
7.  }
8.  @Test
9.  @DisabledIfSystemProperty(named = "ci-server", matches = "true")
10. public void whenRunningTestsOnCIServer_thenTestIsDisabled() {
11.     Integer value = 5; // result of an algorithm
12.
13.     assertNotEquals(0, value, "The result cannot be 0");
14. }
```

This annotation will interpret the value supplied, via the *matches* attribute, as a regular expression.

11.4. Environment Variable Conditions

With the additions of the *@EnabledIfEnvironmentVariable* and *@DisabledIfEnvironmentVariable* annotations, we can enable or disable a test class or a test based on the value of the named environment variable from the underlying operating system:

```
@Test
@EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")
public void whenRunningTestsStagingServer_thenTestIsEnabled() {
    char[] expected = {'J', 'u', 'p', 'i', 't', 'e', 'r'};
    char[] actual = "Jupiter".toCharArray();

    assertArrayEquals(expected, actual, "Arrays should be equal");
}
@Test
@DisabledIfEnvironmentVariable(named = "ENV", matches = ".*development.*")
public void whenRunningTestsDevelopmentEnvironment_thenTestIsDisabled() {
    char[] expected = {'J', 'u', 'p', 'i', 't', 'e', 'r'};
    char[] actual = "Jupiter".toCharArray();

    assertArrayEquals(expected, actual, "Arrays should be equal");
}
```

11.5. Deactivating Conditions

In case we want to run a test suite without certain conditions being active, we can simply provide a pattern for the *junit.jupiter.conditions.deactivate* configuration parameter to specify which condition we want to deactivate.

For example, we can run all our tests, even if annotated with *@Disabled*, by starting our JVM with the proper system property:

```
-Djunit.jupiter.conditions.deactivate=org.junit.*DisabledCondition
```

In contrast to JUnit 4, with the JUnit Jupiter extension model, the Runner, *@Rule*, and *@ClassRule* extension points consists of a single, coherent concept: the *Extension*, a marker interface for all the extensions.

12.1. Extension Model

JUnit 5 extensions are related to a certain event in the execution of a test, referred to as an extension point. When the execution of a test reaches a certain lifecycle phase, the JUnit engine calls registered extensions.

We can use five main types of extension points:

- test instance post-processing
- conditional test execution
- life-cycle callbacks
- parameter resolution
- exception handling

Note that this is only relevant for the Jupiter engine; other JUnit 5 engines will not share the same extension model.

Let's see how we can register an extension.

12.2. Registering an Extension

In JUnit4, we used the *@RunWith* to integrate the test context with other frameworks or to change the overall execution flow in the test cases.

With JUnit 5, we can now use the *@ExtendWith* annotation to provide similar functionality.

As an example, to use the Spring features in JUnit 4:

```
1. @RunWith(SpringJUnit4ClassRunner.class)
2. @ContextConfiguration(
3.   {"/app-config.xml", "/test-data-access-config.xml"})
4. public class SpringExtensionTest {
5.     /*...*/
6. }
```

Now, in JUnit 5 it is a simple extension:

```
1. ExtendWith(SpringExtension.class)
2. @ContextConfiguration(
3.   { "/app-config.xml", "/test-data-access-config.xml" })
4. public class SpringExtensionTest {
5.     /*...*/
6. }
```

The *@ExtendWith* annotation accepts any class that implements the *Extension* interface.

Additional information on the Extension Model, the creation of an *Extension* and its creation are available at A Guide to JUnit 5 Extensions

In JUnit 4, we used the *@Rule* and *@ClassRule* annotations to add special functionality to tests.

The new version of the library doesn't support rules natively anymore. However, in order to enable a gradual migration, the JUnit team has decided to support a selection of JUnit 4 rules.

Let's start looking at the supported rules.

13.1. Supported Rules

JUnit 5 support a few rules using adapters and consider only those rules that are semantically compatible with the JUnit Jupiter extension model.

Therefore, the support includes only those rules that do not completely change the overall execution flow of the test.

First of all, we have to add a dependency to our *pom.xml*:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-migrationsupport</artifactId>
    <version>${junit.vintage.version}</version>
    <scope>test</scope>
</dependency>
```

Once we have configured our dependency, we have the support of three Rule types including subclasses of those types:

- *org.junit.rules.ExternalResource (including org.junit.rules.TemporaryFolder)*
- *org.junit.rules.Verifier (including org.junit.rules.ErrorCollector)*
- *org.junit.rules.ExpectedException*

We can use this limited form of Rule support by switching on by the class-level annotation *org.junit.jupiter.migrationsupport.rules.EnableRuleMigrationSupport:*

```
@EnableRuleMigrationSupport
public class RuleMigrationSupportUnitTest {

    @Rule
    public ExpectedException exceptionRule = ExpectedException.none();

    @Test
    public void whenExceptionThrown_thenExpectationSatisfied() {
        exceptionRule.expect(NullPointerException.class);
        String test = null;
        test.length();
    }

    @Test
    public void whenExceptionThrown_thenRuleIsApplied() {
        exceptionRule.expect(NumberFormatException.class);
        exceptionRule.expectMessage("For input string");
        Integer.parseInt("1a");
    }
}
```

Since JUnit 4 *Rule* support in JUnit Jupiter is currently an experimental feature, if we to develop a new extension for JUnit 5, we should use the new extension model of JUnit Jupiter instead of the rule-based model of JUnit 4.

13.2. Rule Migration to Extension

In JUnit 5, we can reproduce the same logic using the *@ExtendWith* annotation.

For example, say we have a custom rule in JUnit 4 to write log traces before and after a test:

In JUnit 5, we can reproduce the same logic using the *@ExtendWith* annotation.

For example, say we have a custom rule in JUnit 4 to write log traces before and after a test:

```
public class TraceUnitTestRule implements TestRule {

@Override
    public Statement apply(Statement base, Description description) {
        return new Statement() {
            @Override
            public void evaluate() throws Throwable {
                // Before and after an evaluation tracing here
                ...
            }
        };
    }
}
```

And we implement it in a test suite:

```
@Rule
public TraceUnitTestRule traceRuleTests = new TraceUnitTestRule();
```

In JUnit 5, we can write the same in a much more intuitive manner:

```
public class TraceUnitExtension implements AfterEachCallback,
BeforeEachCallback {

    @Override
    public void beforeEach(TestExtensionContext context) throws Exception {
        // ...
    }

    @Override
    public void afterEach(TestExtensionContext context) throws Exception {
        // ...
    }
}
```

Using JUnit 5's *AfterEachCallback* and *BeforeEachCallback* interfaces available in the package *org.junit.jupiter.api.extension*, we easily implement this rule in the test suite:

```
1.  @RunWith(JUnitPlatform.class)
2.  @ExtendWith(TraceUnitExtension.class)
3.  public class RuleExampleTest {
4.
5.      @Test
6.      public void whenTracingTests() {
7.          /*...*/
8.      }
9.  }
```

14. Conclusion

In this ebook, we went through all the steps to perform a migration from JUnit 4 to JUnit5 and how to improve our tests by exploiting the different enhancements of JUnit 5.

As always, all code examples in this ebook can be found over on GitHub.

V3.1