A Look at JUnit 5's Core Features & New Testing Functionality

EUGEN PARASCHIVAUGUST 30, 2017DEVELOPER TIPS, TRICKS & RESOURCES1 COMMENT

JUnit 5 is the updated version of the highly popular testing library for Java applications, JUnit, **scheduled to be released in Q3 2017**. The new version enab a lot more testing options and finally adds support for Java 8 features. In fact, JUnit 5 requires Java 8 work.

The library is composed of several modules, organized in 3 main sub-projects:

- *JUnit Platform* – which enables launching testing frameworks on the JVM
- *JUnit Jupiter* – which contains new features for writing tests in JUnit 5
- *JUnit Vintage* – which provides support for running JUnit 3 and JUnit 4 tests on the JUnit 5 platfor

This article will explore the core functionality as well as the new additions to the library.

JUnit 5 Setup

To start using JUnit 5 in your Java project, you have to start by adding the *junit-jupiter-engine*depende to your project's classpath.

If you're using Maven, you can simply add the following to your *pom.xml*:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.0.0-M4</version>
</dependency>
```

As mentioned, a Java 8 baseline for your project is required.

Currently, only *IntelliJ IDEA* has JUnit 5 support in the IDE, while Eclipse just offers beta support.

Another way to run the tests is by using the Maven Surefire plugin:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.20</version>
    <dependencies>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-surefire-provider</artifac
tId>
            <version>1.0.0-M4</version>
        </dependency>
    </dependencies>
```

With this plugin set up, tests will run with the standard "mvn clean install" command.

JUnit 5 Test Annotations

Let's start by understanding a core feature of JUnit 5 – the annotations.

The new library provides a series of annotations for configuring tests, some of which are new to this

Let's go through the basics:

- *@Test* – denotes a test method; unlike the *@Test* annotation from previous versions, it doesn't accept any arguments
- *@DisplayName* – specifies a custom name for the test class or method
- *@BeforeEach*, *@AfterEach* – runs the annotated method before or after each test method in the same class; equivalent to the previous *@Before* and *@After*
- @BeforeAll, @AfterAll – runs the annotated method before any or after all of the test methods in t class; equivalent to the previous *@BeforeClass* and *@AfterClass*
- *@Disabled* – prevents a test class or method from running; similar to the previous *@Ignore*

All of these belong to the *org.junit.jupiter.api* package.

Now that we understand annotations better, **let's have a look at a simple example** of how we could use *@BeforeAll* and *@AfterAll* to setup some test data.

For example, in an application with a simple DAO-based persistence layer, we're going to use *@BeforeAll* to create a few *User* entities and save them to make them available to each test meth

```
@BeforeAll
public static void addData(){
    User user1 = new User("john@gmail.com", "John");
    User user2 = new User("ana@gmail.com","Ana");
    userDAO.add(user1);
    userDAO.add(user2);
}
```

Then, you can make sure this data is removed after all the tests have completed:

```
@AfterAll
public static void removeData(){
    userDAO.deleteAll();
}
```

This way you ensure a clean database before each set of tests runs.

Notice both of these **methods annotated with *@BeforeAll* and *@AfterAll* need to be static**.

Let's also add a simple test method with a custom display name that verifies the two users do exist:

```
@Test
@DisplayName("Test Get Users")
public void testGetUsers() {
    assertEquals(2, userDAO.findAll().size());
}
```

Assertions

JUnit 5 contains many of the JUnit 4 assertions as well as a number of interesting new ones. And, mo importantly, it **also adds support for lambda expressions to be used in assertions**.

One advantage of using a lambda expression for the assertion message is that it causes it to be lazily evaluated, which can save time and resources by avoiding the construction of complex messages like those:

```
@Test
public void testGetUser() {
    User user = userDAO.findOne("john@gmail.com");

    assertNotNull(user);
    assertEquals("John", user.getName(),
      "User name:" + user.getName() + " incorrect");
}
```

All the assertion methods can be imported through static import from the *Assertions* class:

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

Naturally, most of the JUnit 4 classic assertion methods are still available in the new format *(<expecte*
<actual>,<message>):

```
@Test
public void testClassicAssertions() {
    User user1 = userDAO.findOne("john@gmail.com");
    User user2 = userDAO.findOne("john@yahoo.com");

    assertNotNull(user1);
    assertNull(user2);

    user2 = new User("john@yahoo.com", "John");
    assertEquals(user1.getName(), user2.getName(), "Names are not
 equal");
```

New Assertions

In addition to the classic assertions, **it is now possible to group assertions using the *assertAll() A***
and have all the failed assertions reported together:

```
@Test
public void testGetUsers() {
    User user = userDAO.findOne("john@gmail.com");

    assertAll("user",
      () -> assertEquals("Johnson", user.getName()),
      () -> assertEquals("johnson@gmail.com", user.getEmail()));
}
```

The assertion failures will be reported in a *MultipleFailuresError* object:

```
org.opentest4j.MultipleFailuresError:
user (2 failures)
        expected: <Johnson> but was: <John>
        expected: <johnson@gmail.com> but was: <john@gmail.com>
        at com.stackify.test.UsersTest.testGroupedAssertions(UsersTest.java:71)
```

This behavior is **very helpful for testing sets of related properties** – as you can see the result of ea
as opposed to having separate assertions for them, where only the first failure would be shown.

To compare arrays and collections, you can now use
the *assertArrayEquals()* and *assertIterableEquals()* methods:

```
@Test
public void testIterableEquals() {
    User user1 = new User("john@gmail.com", "John");
    User user2 = new User("ana@gmail.com", "Ana");

    List<User> users = new ArrayList<>();
    users.add(user1);
    users.add(user2);

    assertIterableEquals(users, userDAO.findAll());
}
```

For this assertion to succeed, the *User* class naturally has to implement a relevant *equals()*method.

A list of *Strings* can also be compared using the *assertLinesMatch()* method, where the expected argument can contain *Strings* to compare as well as regular expressions:

```
@Test
public void testLinesMatch() {
    List<String> expectedLines = Collections.singletonList("(.*)@
(.*)");
    List<String> emails = Arrays.asList("john@gmail.com");
    assertLinesMatch(expectedLines, emails);
}
```

A quick interesting side-note – this feature was first developed internally to verify the output of the new *ConsoleLauncher*.

Next, since the *@Test* annotation no longer accepts arguments, such as an expected exception, JUni now provides **the *assertThrows()* method to define and verify expected exceptions**:

```
@Test
public void testThrows() {
    User user = null;
    Exception exception =
assertThrows(NullPointerException.class, () -> user.getName());
    logger.info(exception.getMessage());
}
```

An advantage of this method is that it returns the *Exception* object which can be further used to obtair more information about the thrown exception.

Finally, another new assertion in JUnit 5 is *fail()*, which **simply fails a test**:

```
@Test
public void testFail() {
    fail("this test fails");
}
```

Assumptions

Now that you've seen the most important assertions in JUnit 5, let's now focus on a new and very

An assumption defines the conditions which have to be met so that a test will be run. **A failing assumption does not mean a test is failing**, but simply that the test won't provide any relevant information, so it doesn't need to run.

Conditions for running tests can be defined using the methods: *assumeTrue(), assumeFalse()*and *assumingThat()*:

```
@Test
public void testAssumptions() {
    List<User> users = userDAO.findAll();
    assumeFalse(users == null);
    assumeTrue(users.size() > 0);

    User user1 = new User("john@gmail.com", "John");
    assumingThat(users.contains(user1), () -> assertTrue(users.si
ze() > 1));
}
```

Tagging and Filtering Tests

Groping tests that logically belong together has been historically difficult.

This is exactly what this new feature addresses; the *@Tag* annotation can be added to a test class or method **to group tests by a certain tag**. The tag can later be used to determine which tests should r

```
@Tag("math")
public class TaggedTest {
    @Test
    @Tag("arithmetic")
    public void testEquals(){
        assertTrue(1==1);
    }
}
```

You can then configure tags to run by using the *<groups>* or *<includeTags>* elements in surefire, and to be excluded via *<excludedGroups>* or *<excludeTags>*:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.19</version>
    <configuration>
        <properties>
            <excludeTags>math</excludeTags>
        </properties>
    </configuration>
</plugin>
```

Nested Tests

JUnit 5 also offers the possibility of creating nested tests by simply annotating an inner class with *@Nested*:

```
public class UsersTest {
    private static UserDAO userDAO;

    @Nested
    class DeleteUsersTest {
        @Test
        public void addUser() {
            User user = new User("bob@gmail.com", "Bob");
            userDAO.add(user);
            assertNotNull(userDAO.findOne("bob@gmail.com"));
```

The nested test class must be an inner class, meaning a non-static nested class.

And, since inner classes cannot have static fields and methods, this prohibits the use of the *@BeforeAll* and *@AfterAll* annotations in nested tests.

Repeated Tests

The new release also introduces **the *@RepeatedTest* annotation to mark a test that needs to run several times**. The annotation must specify the number of times you want a test to run.

The *@RepeatedTest* benefits from the full JUnit lifecycle support. This means that if you define a *@BeforeEach* or *@AfterEach* method, it will be run before each execution of the test.

In this following example, the message “Before Each Test” will be displayed 3 times:

```
public class IncrementTest {

    private static Logger logger = LogManager.getLogger(Increment
Test.class);

    @BeforeEach
    public void increment() {
        logger.info("Before Each Test");
    }

    @RepeatedTest(value=3, name=RepeatedTest.SHORT_DISPLAY_NAME)
```

The *name* attribute can be used to display more information about the repetitions.

Each *@RepeatedTest* can also take a *RepetitionInfo* parameter which contains repetition metadata.

The output of the above example will be:

```
Running com.stackify.test.IncrementTest
02:23:25.322 [main] INFO  com.stackify.test.IncrementTest - Before Each Test
02:23:25.331 [main] INFO  com.stackify.test.IncrementTest - Repetition #1
02:23:25.336 [main] INFO  com.stackify.test.IncrementTest - Before Each Test
02:23:25.337 [main] INFO  com.stackify.test.IncrementTest - Repetition #2
02:23:25.341 [main] INFO  com.stackify.test.IncrementTest - Before Each Test
02:23:25.342 [main] INFO  com.stackify.test.IncrementTest - Repetition #3
```

Dependency Injection for Constructors and Methods

You may have noticed in the previous section that we added a parameter of type *RepetitionInfo*to the *test()* method. This wasn’t possible in previous versions of JUnit.

And given just how useful constructor injection can be, JUnit 5 now allows defining parameters for tes constructors and methods and enables dependency injection for them. This mechanism works by usi an instance of a *ParameterResolver* to dynamically resolve parameters at runtime.

Currently, there are only 3 built-in resolvers for parameters of type *TestInfo*, *RepetitionInfo* and *TestReporter*.

Let's see how the *TestInfo* parameter can be used to obtain metadata about a test method:

```
@Test
@DisplayName("Test Get Users")
public void testGetUsersNumberWithInfo(TestInfo testInfo) {
    assertEquals(2, userDAO.findAll().size());
    assertEquals("Test Get Users", testInfo.getDisplayName());
    assertEquals(UsersTest.class, testInfo.getTestClass().get());

    logger.info("Running test method:" +
testInfo.getTestMethod().get().getName());
}
```

The *getTestClass()* and *getTestMethod()* methods are followed by a *get()* call since they return an *Optional* object.

Parameterized Tests

Parameterized tests allow **running the same test multiple times, but with different arguments**.

In order to enable parameterized tests, you need to add the *junit-jupiter-params* dependency to the classpath:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.0.0-M4</version>
</dependency>
```

You can then define this style of test using the *@ParameterizedTest* annotation and at least one sour of arguments; there are **several types of parameter sources you can pick from**:

- *@ValueSource* – defines an array of literals of primitive types, and can only provide a single parameter per test invocation
- *@EnumSource* – uses an *Enum* as a parameter source
- *@MethodSource*– uses one or more methods of the test class; the methods must return an array a *Stream*, *Iterable* or *Iterator* object, and must be static and have no arguments
- *@CsvSource* and *@CsvFileSource* – uses parameters defined in CSV format, either in *String*obje or read from a file
- *@ArgumentsSource* – uses a custom *ArgumentsProvider*

Let's see a quick example of a repeated test which uses a *@ValueSource* with a string parameter:

```
@ParameterizedTest
@ValueSource(strings = { "john@gmail.com", "ana@gmail.com" })
public void testParameterized(String email) {
```

Dynamic Tests

In addition to the standard static tests, defined with the *@Test* annotations, **JUnit 5 introduces the possibility of defining tests at runtime**. These dynamic tests can be generated using a factory method annotated with *@TestFactory*.

Simply put, this test factory must return a *Stream*, *Collection*, *Iterable* or *Iterator* of *DynamicTest*.

Note that dynamic tests do not support lifecycle callbacks. Therefore, methods annotated with *@BeforeEach* or *@AfterEach* will not be executed.

Let's see a simple example of a test factory method returning a *Collection* with a *DynamicTest*object:

```
@TestFactory
Collection<DynamicTest> dynamicTestCollection() {
    return Arrays.asList(DynamicTest.dynamicTest("Dynamic Test",
() -> assertTrue(1==1)));
}
```

For a more dynamic method, you can create an iterator that provides inputs, a display name generatc and a test executor – then use these in a *DynamicTest.stream()* method:

```
@TestFactory
Stream<DynamicTest> dynamicUserTestCollection() {
    List<User> inputList = Arrays.asList(new
User("john@yahoo.com", "John"), new User("ana@yahoo.com",
"Ana"));

    Function<User, String> displayNameGenerator = (input) -> "Sav
ing user: " + input;

    UserDAO userDAO = new UserDAO();
    ThrowingConsumer<User> testExecutor = (input) -> {
```

Test Annotations in Interfaces

JUnit 5 also allows several **annotations to be added to test interfaces**:

- *@Test*, *@TestFactory*, *@BeforeEach* and *@AfterEach* can be added to default methods in interfaces (introduced in Java 8)
- *@BeforeAll* and *@AfterAll* can be added to static methods in interfaces
- *@ExtendsWith* and *@Tag* can be declared on interfaces

And, as expected, the classes that implement these interface will inherit the test cases:

```
public interface DatabaseConnectionTest {

    @Test
    default void testDatabaseConnection() {
        Connection con = ConnectionUtil.getConnection();
        assertNotNull(con);
    }
```

```
public class UsersTest implements DatabaseConnectionTest { .... }
```

In this example, the *UsersTest* class will run the *testDatabaseConnection()* test in addition to its own tests.

In small projects, this can be a nice feature, but in larger, complex codebases with extensive code sui this can be a game changer, as **it leads to much nice composition semantics in the system**.

Conditional Test Execution

JUnit 5 allows defining custom annotations that act as conditions to determine whether a test should b run or not. The classes that contain the conditional logic need
to **implement *ContainerExecutionCondition* to evaluate tests in a test class,
or *TestExecutionCondition*to evaluate test methods**.

To define a custom condition, you first need to create the annotation:

```
@Target({ ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@ExtendWith(DisabledOnEnvironmentCondition.class)
public @interface DisabledOnEnvironment {
    String[] value();
}
```

Notice we've created an annotation called *DisabledOnEnvironment* which can now be applied to a method and will mark the test disabled on these environments.

With the annotation implemented, you now need to define the *DisabledOnEnvironmentCondition*class

This simply needs to implement the *TestExecutionCondition* interface and override the *evaluate()*meth The *evaluate()* implementation will load the environments from a *.properties* file and check them agai the list from the annotation itself:

```
public class DisabledOnEnvironmentCondition implements TestExecut
ionCondition {

    @Override
    public ConditionEvaluationResult evaluate(TestExtensionContex
t context) {
        Properties props = new Properties();
        String env = "";
        try {

props.load(ConnectionUtil.class.getResourceAsStream("/applicatio
```

The method returns a *ConditionEvaluationResult* that specifies whether the test method will be enable or not.

Then, you can simply add the new annotation to a test method:

```
@Test
@DisabledOnEnvironment({ "dev", "prod" })
void testFail() {
    fail("this test fails");
```

Migrating from JUnit 4

JUnit 5 packs quite a punch.

But, you've likely been writing unit tests for a while now, and have a legacy test suite that's already running and producing value.

And so, a proper migration plan will be critical. That's exactly why JUnit 4 tests can still run using JUn simply by using the *junit-vintage-engine* dependency:

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <version>4.12.0-M4</version>
</dependency>
```

However, if you want to fully migrate your tests from JUnit 4 to the JUnit 5 API – here are some of the changes you will most likely encounter:

- change everything from *org.junit* to the *org.junit.jupiter.api* package
- replace *@BeforeClass* and *@AfterClass* with *@BeforeAll* and *@AfterAll*
- replace *@Before* and *@After* with *@BeforeEach* and *@AfterEach*
- replace *@Ignore* with *@Disabled*
- remove *@Rule, @ClassRule* and *@RunWith*

Conclusion

The new JUnit 5 library not only measures up to its predecessor but adds a host of highly powerful ar useful features improvements over the previous JUnit incarnation. And, beyond all the new additions, finally also get the nice Java 8 syntax missing in JUnit 4. Support for the version 4 is, of course, availa through the Vintage Platform, so the transition to the new version can be smoother and gradual.