GeekBrains

Автоматизация тестирования Web UI на Java

JUnit

[Java 11]

На этом уроке

1. Познакомимся с JUnit.
2. Разберём принципы создания тестов.

Оглавление

Знакомство с JUnit

JUnit — самый популярный фреймворк для модульного тестирования Java-приложений, а последняя версия этого инструмента, т. е. JUnit 5, обладает ещё более полезным и мощным инструментарием.

Обзор

JUnit — одна из самых популярных платформ модульного тестирования для Java. Поэтому в сообществе разработчиков появляются большие проблемы, когда выходят новые основные версии.

Первая версия JUnit 5 вышла в далёком 2016 году. Но из-за недостатка документации и настроенной интеграции с основными решениями, используемыми в разработке на Java, довольно популярной остаётся версия JUnit 4. На нашем курсе мы рассмотрим последнюю, пятую, версию, так как новые проекты уже пишутся в основном на ней.

Созданный Кентом Беком и Эриком Гаммой, JUnit принадлежит семье фреймворков xUnit для разных языков программирования, берущей начало в SUnit Кента Бека для Smalltalk. JUnit породил экосистему расширений — jMock, EasyMock, DbUnit, HttpUnit и т. д.

JUnit переведён на другие языки, включая PHP (PHPUnit), C# (NUnit), Python (PyUnit), Fortran (fUnit), Delphi (DUnit), Free Pascal (FPCUnit), Perl (Test::Unit), C++ (CppUnit), Flex (FlexUnit), JavaScript (JSUnit), COS (COSUnit).

Зависимости и настройка

Установка JUnit 5 довольно проста.

Требуется просто добавить следующую зависимость в наш pom.xml:

```
<dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.6.2</version>
      <scope>test</scope>
</dependency>
```

Многие проблемы с интеграцией JUnit 5 в действующих IDE решаются тяжело, поэтому применяется готовое решение для отчётов:

```
<build>
       <plugins>
           <plugin>
               <artifactId>maven-surefire-plugin</artifactId>
               <version>3.0.0-M5</version>
           </plugin>
           <plugin>
```

```
                <artifactId>maven-failsafe-plugin</artifactId>
                <version>2.12</version>
            </plugin>
            <plugin>
                <artifactId>maven-surefire-report-plugin</artifactId>
                <version>3.0.0-M5</version>
            </plugin>
<!-- Лечение проблемы для mvn site:   A required class was missing while
executing org.apache.maven.plugins:maven-site-plugin:3.3:site:
org/apache/maven/doxia/siterenderer/DocumentContent-->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-site-plugin</artifactId>
                <version>3.7.1</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-project-info-reports-plugin</artifactId>
                <version>3.0.0</version>
            </plugin>
        </plugins>
    </build>
```

Важно отметить, что эта версия требует Java 8 для работы.

Чтобы получить SureFire-отчёты, требуется сначала запустить тесты:

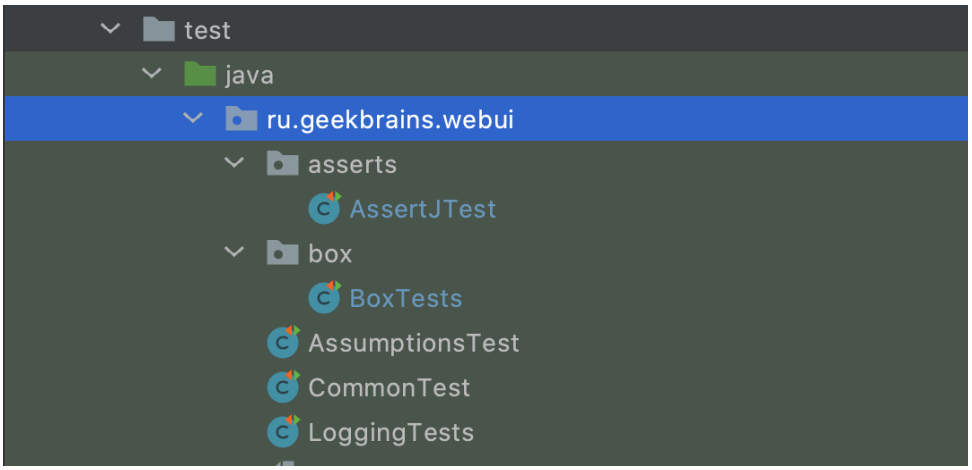
`mvn clean install` или `mvn clean test`

Затем — запустить генерацию документации и отчёта:

```
mvn site surefire-report:report
```

Написание тестов

Тесты, где используется Maven для сборки проекта, должны лежать в пакете src/test/java. Внутри этого пакета тесты делятся на разные классы и даже пакеты для более удобного запуска:



Тесты по умолчанию должны находиться в классе, название которого начинается на Test или оканчивается на Test, TestCase или Tests. Как настроить запуск тестов из других классов — здесь. Тестом называется метод public static void, помеченный аннотацией @org.junit.jupiter.api.Test — здесь и далее все названия даются для JUnit 5.

Тело теста традиционно делится на две части:

- функциональная часть (шаги теста) обычно находится прямо в проверке;
- проверки (asserts), где происходит сравнение актуального результата с ожидаемым.

Самый простой тест мы видели, когда создавали проект из архетипа Quickstart maven:

```
@Test
public void rigorousTest(){
Assertions.assertTrue(true)
}
```

Название теста содержит слово test в начале или в конце (Test).

Аннотации JUnit

Кроме аннотации @Test, которой обозначаются все тесты, есть ещё и другие. Для начала перечислим те, что используются для тестовых методов:

1. @DisplayedName (название метода) используется для обозначения методов в отчёте на понятном человеку языке.
2. @Disabled (причина пропуска теста) — для пропуска, то есть исключения из прогона теста.
3. @Tag (имя тега) тегирует тесты для фильтрации при запуске.
4. @RepeatedTest (число) повторяет тест указанное количество раз.

5. @ParameterizedTest используется вместо аннотации для передачи параметров в тест (DataProvider в TestNG).
6. @ValueSource(strings = { "s1", "s2" }) используется вместе с предыдущей аннотацией для объявления конкретных параметров для теста. Сейчас поддерживаются только данные примитивных типов: int, long, double, String.

Параметризованные тесты

Поговорим подробнее о параметризованных тестах. Кроме указанных выше аннотаций, в аргументах теста требуется указать название используемого параметра как в обычном методе:

```
@DisplayName("Слово является палиндромом")
    @ParameterizedTest
    @ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
    public void isPalindromeTest(String word){
        Assertions.assertTrue(functions.isPalindrome(word));
    }
```

Хотя в @ValueSource можно использовать лишь примитивы, есть другие варианты указания источника данных:

Пример с разбором CSV.

```
@ParameterizedTest
@CsvSource({ "foo, 1", "bar, 2", "'baz, qux', 3" })
// или даже так: @CsvFileSource(resources = "/two-column.csv")
void testWithCsvSource(String first, int second) {
    assertNotNull(first);
    assertNotEquals(0, second);
}
```

Пример с Enum

```
@ParameterizedTest
@EnumSource(value = TimeUnit.class, names = { "DAYS", "HOURS" })
void testWithEnumSourceInclude(TimeUnit timeUnit) {
    assertTrue(EnumSet.of(TimeUnit.DAYS, TimeUnit.HOURS).contains(timeUnit));
}
```

Пример с источником данных

```
@ParameterizedTest
@ArgumentsSource(MyArgumentsProvider.class)
void testWithArgumentsSource(String argument) {
    assertNotNull(argument);
}

static class MyArgumentsProvider implements ArgumentsProvider {
    @Override
    public Stream<? extends Arguments> provideArguments(ExtensionContext
context) {
        return Stream.of("foo", "bar").map(Arguments::of);
    }
}
```

Подготовка данных

Для подготовки и удаления тестовых данных используются методы @BeforeAll (@AfterAll) — запускаются перед (после) всеми тестами один раз. Есть также методы @BeforeEach (@AfterEach), которы не запускаются перед (после) каждым тестом.

Методы @Before… обычно называют методами setUp(), а методы @After… — tearDown().

Для @BeforeAll и @AfterAll важно, чтобы они были объявлены как static, так как его инициализация происходит до остальных методов. Если требуется использовать такие методы не в контексте static, есть эта статья.

Зависимые тесты

Зависимость тестов можно показать через внутренние классы с тестами, помеченными аннотацией @Nested:

```
public class BoxTests {
    Box box;

    @Test
    void canBeInitializedTest() {
        box = new Box();
    }

    @Nested
    @DisplayName("when new")
    class WhenNew {

        @BeforeEach
        void createNewBox() {
            box = new Box();
        }

        @Test
```

```
        @DisplayName("is empty")
        void isEmptyTest() {
            assertThat(box.isEmpty());
        }
    }
}
```

Таким образом, выстраивается иерархия вложенных классов, например, рядом или в классе WhenNew может быть сколько угодно классов, чтобы у каждого теста была установка и отключение всех тестов по иерархии. Документация JUnit предлагает подробный example, иллюстрирующий одно из возможных применений.

Расширения

Иногда хочется добавить к тестам дополнительные возможности — обычно это касается фильтрации (группировки) тестов при запуске, подготовки тестовых данных и логирования. Можно написать своё расширение, которое встраиваться в JUnit-фреймворк через аннотацию @ExtendWith (НазваниеКлассаРасширения.class), прикреплённой на класс с тестами. Подробно эту тему мы разберём на занятии, также об этом можно почитать здесь.

Самое полезное расширение для JUnit 5 — это написание расширения для выполнения методов перед всеми классами тестов сразу.

Полный список аннотаций JUnit есть в официальной документации на сайте.

Утверждения (Assertions)

Утверждения находятся в org.junit.jupiter.api.Assertions и были значительно улучшены по сравнению с предыдущими версиями.

Самые простые assertions выглядят следующим образом:

1. Assertions.assertTrue(condition) проверяет, что булево значение внутри скобок равно true.
2. Assertions.assertFalse(condition) проверяет, что булево значение внутри скобок равно false.
3. Assertions.assertEquals(number1, number2) проверяет, что два значения равны. Примечательно, что методы реализуются для всех числовых типов.

В JUnit 5 используется лямбда-выражение в утверждениях:

```
@Test
void lambdaExpressions() {
    assertTrue(Stream.of(1, 2, 3)
      .stream()
      .mapToInt(i -> i)
```

```
      .sum() > 5, () -> "Sum should be greater than 5");
}
```

Одно из преимуществ использования лямбда-выражения состоит в том, что оно лениво инициализируется. Это сэкономит время и ресурсы, если построение сообщения считается дорогостоящим.

Теперь есть возможность группировать утверждения путём использования assertAll (), который сообщит о любых неудачных утверждениях в группе через 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)
     );
 }
```

Это означает, что безопаснее делать более сложные утверждения, поскольку мы сможем определить точное местоположение любого отказа.

Предположения

Допущения (предположения, assumptions) используются для запуска тестов только при соблюдении конкретных условий.

Это обычно используется для внешних условий, которые требуются для правильной работы теста. Однако внешние условия не имеют прямого отношения к тому, что тестируется.

Мы можем объявить предположение через assumeTrue (), assumeFalse () и assumingThat () .

```
@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)
    );
}
```

Если предположение не выполняется, то генерируется TestAbortedException, и тест просто пропускается.

Предположения также понимают лямбда-выражение.

Исключения

JUnit 5 улучшает поддержку исключений. Добавился метод assertThrows (), который проверяет, что выбрасывает выражение:

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

Теперь можно легко получить любую информацию, которая понадобится в исключении. Мы уже сделали это в нашем примере, изучив сообщение об исключении.

Практическое задание

1. Напишите функцию, вычисляющую площадь треугольника по трём сторонам (int a, int b, int c). Разместите класс с функцией в src/main/java.
2. Разместите тесты на эту функцию в классе src/test/java/.../TriangleTest.java.
3. Настройте генерацию отчёта и по желанию — логирование.