(/)

Una guía para JUnit 5

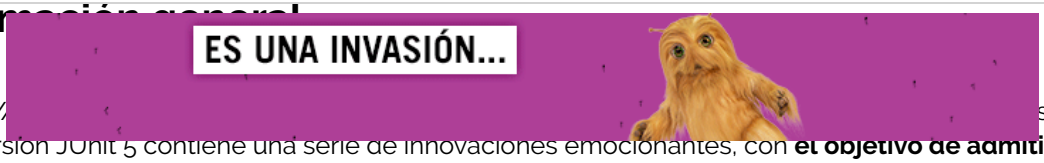
Última modificación: 13 de enero de 2020

por baeldung (https://www.baeldung.com/author/baeldung/)
(https://www.baeldung.com/author/baeldung/)

Pruebas (https://www.baeldung.com/category/testing/)

JUnit (https://www.baeldung.com/tag/junit/) **JUEGO 5 (https://www.baeldung.com/tag/junit-5/)**

1. Inform



JUnit (http:/ stema de Java. La versión JUnit 5 contiene una serie de innovaciones emocionantes, con **el objetivo de admitir nuevas funciones en Java 8 y** versiones **posteriores** , así como habilitar muchos estilos diferentes de prueba.

2. Dependencias de Maven

Configurar JUnit 5.x.0 (https://search.maven.org/search?q=a:junit-jupiter-engine) es bastante sencillo, necesitamos agregar la siguiente dependencia a nuestro *pom.xml* :

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

Es importante tener en cuenta que esta versión **requiere Java 8 para funcionar** .

Además, ahora hay soporte directo para ejecutar pruebas de Unidad en la Plataforma JUnit en Eclipse, así como en IntelliJ. Por supuesto, también puede ejecutar pruebas con el objetivo de Maven Test.

Por otro lado, IntelliJ admite JUnit 5 de forma predeterminada. Por lo tanto, ejecutar JUnit 5 en IntelliJ es bastante simple, simplemente haga clic derecho -> Ejecutar, o Ctrl-Shift-F10.

3. Arquitectura

JUnit 5 está

3.1. Plataforma JUnit

La plataforma es responsable de lanzar marcos de prueba en la JVM. Define una interfaz estable y potente entre JUnit y su cliente, como las herramientas de compilación.

El objetivo final es cómo sus clientes se integran fácilmente con JUnit para descubrir y ejecutar las pruebas.

También define la API TestEngine (http://junit.org/junit5/docs/current/api/org/junit/platform/engine/TestEngine.html) para desarrollar un marco de prueba que se ejecuta en la plataforma JUnit. Con eso, puede conectar bibliotecas de prueba de terceros, directamente en JUnit, implementando TestEngine personalizado.

3.2. JUnit Jupiter

Este módulo incluye nuevos modelos de programación y extensión para escribir pruebas en JUnit 5. Las nuevas anotaciones en comparación con JUnit 4 son:

- *@TestFactory* : denota un método que es una fábrica de pruebas para pruebas dinámicas
- *@DisplayName* : define el nombre para mostrar personalizado para una clase de prueba o un método de prueba
- *@Nested* - denota que la clase anotada es una clase de prueba anidada, no estática
- *@Tag* : declara etiquetas para filtrar pruebas
- *@ExtendWith* : se utiliza para registrar extensiones personalizadas

- *@BeforeEach:* denota que el método anotado se ejecutará antes de cada método de prueba (anteriormente *@Before*)
- *@AfterEach* : indica que el método anotado se ejecutará después de cada método de prueba (anteriormente *@After*)
- *@BeforeAll* : denota que el método anotado se ejecutará antes que todos los métodos de prueba en la clase actual (anteriormente *@BeforeClass*)
- *@AfterAll* : denota que el método anotado se ejecutará después de todos los métodos de prueba en la clase actual (anteriormente *@AfterClass*)
- *@Disable* : se utiliza para deshabilitar una clase o método de prueba (anteriormente *@Ignore*)

3.3. JUnit Vintage

Admite ejecutar pruebas basadas en JUnit 3 y JUnit 4 en la plataforma JUnit 5.

4. Anotaciones básicas

Para discutir nuevas anotaciones, dividimos la sección en los siguientes grupos, responsables de la ejecución: antes de las pruebas, durante las pruebas (opcional) y después de las pruebas:

4.1. *@BeforeAll* y *@BeforeEach*

A continuación se muestra un ejemplo del código simple que se ejecutará antes de los principales casos de prueba:

```
@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");
}
```

Es importante tener en cuenta que el método con la anotación *@BeforeAll* debe ser estático; de lo contrario, el código no se compilará.

4.2. *@DisplayName* y *@Disabled*

Pasemos a los nuevos métodos de prueba opcionales:

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

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

Como podemos ver, podemos cambiar el nombre para mostrar o deshabilitar el método con un comentario, utilizando nuevas anotaciones.

4.3. *@AfterEach* y *@AfterAll*

Finalmente, analicemos los métodos relacionados con las operaciones después de la ejecución de las pruebas:

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

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

Tenga en cuenta que el método con *@AfterAll* también debe ser un método estático.

5. Afirmaciones y supuestos

JUnit 5 intenta aprovechar al máximo las nuevas características de Java 8, especialmente las expresiones lambda.

5.1. Aserciones

Las aserciones se han movido a *org.junit.jupiter.api.Assertions* y se han mejorado significativamente. Como se mencionó anteriormente, ahora puede usar lambdas en afirmaciones:

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

Aunque el ejemplo anterior es trivial, una ventaja de usar la expresión lambda para el mensaje de afirmación es que se evalúa de manera perezosa, lo que puede ahorrar tiempo y recursos si la construcción del mensaje es costosa.

Ahora también es posible agrupar aserciones con *afirmarAll ()* que informará cualquier aserción fallida dentro del grupo con un *error 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)
    );
}
```

Esto significa que ahora es más seguro hacer afirmaciones más complejas, ya que podrá determinar la ubicación exacta de cualquier falla.

5.2. Supuestos

Los supuestos se utilizan para ejecutar pruebas solo si se cumplen ciertas condiciones. Esto se usa generalmente para condiciones externas que se requieren para que la prueba se ejecute correctamente, pero que no están directamente relacionadas con lo que se está probando.

Puede declarar una suposición con *assumeTrue ()* , *assumeFalse ()* y *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)
    );
}
```

Si falla una suposición, se lanza una *TestAbortedException* y la prueba simplemente se omite.

Los supuestos también entienden las expresiones lambda.

6. Pruebas de excepción

Hay dos formas de realizar pruebas de excepción en JUnit 5. Ambas pueden implementarse mediante el método *ClaimThrows ()* :

```
@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);
    });
}
```

El primer ejemplo se usa para verificar más detalles de la excepción lanzada y el segundo simplemente valida el tipo de excepción.

7. Suites de prueba

Para continuar con las nuevas características de JUnit 5, trataremos de conocer el concepto de agregar múltiples clases de prueba en un conjunto de pruebas para que podamos ejecutarlas juntas. JUnit 5 proporciona dos anotaciones: *@SelectPackages* y *@SelectClasses* para crear suites de prueba.

Tenga en cuenta que en esta etapa inicial, la mayoría de los IDE no son compatibles con esas características.

Echemos un vistazo al primero:

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

@SelectPackage se utiliza para especificar los nombres de los paquetes que se seleccionarán al ejecutar un conjunto de pruebas. En nuestro ejemplo, ejecutará todas las pruebas. La segunda anotación, *@SelectClasses* , se usa para especificar las clases que se seleccionarán al ejecutar un conjunto de pruebas:

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

Por ejemplo, la clase anterior creará una suite que contiene tres clases de prueba. Tenga en cuenta que las clases no tienen que estar en un solo paquete.

8. Pruebas dinámicas

El último tema que queremos presentar es la función JUnit 5 Dynamic Tests, que permite declarar y ejecutar casos de prueba generados en tiempo de ejecución. A diferencia de las pruebas estáticas que definen un número fijo de casos de prueba en el momento de la compilación, las pruebas dinámicas nos permiten definir el caso de pruebas dinámicamente en tiempo de ejecución.

Las pruebas dinámicas se pueden generar mediante un método de fábrica anotado con *@TestFactory.* Echemos un vistazo al código de ejemplo:

```
@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));
          })
    );
}
```

Este ejemplo es muy sencillo y fácil de entender. Queremos traducir palabras usando dos *ArrayList* , nombradas *dentro* y *fuera* , respectivamente. El método de fábrica debe devolver un *Stream* , *Colección* , *Iterable* o *Iterator* . En nuestro caso, elegimos Java 8 *Stream.*

Tenga en cuenta que los métodos *@TestFactory* no deben ser privados o estáticos. El número de pruebas es dinámico y depende del tamaño de *ArrayList* .

9. Conclusión

El informe fue una descripción general rápida de los cambios que vienen con JUnit 5.

Podemos ver que JUnit 5 tiene un gran cambio en su arquitectura relacionado con el iniciador de plataforma, la integración con la herramienta de compilación, IDE, otros marcos de prueba de la Unidad, etc. Además, JUnit 5 está más integrado con Java 8, especialmente con los conceptos de Lambdas y Stream .

Los ejemplos utilizados en este artículo se pueden encontrar en el proyecto GitHub (https://github.com/eugenp/tutorials/tree/master/testing-modules/junit-5-basics) .

¡Los comentarios están cerrados en este artículo!

CATEGORIAS

PRIMAVERA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP DEL LADO DEL CLIENTE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL DE JAVA "VOLVER A LO BÁSICO" (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
RESTO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
TUTORIAL SPRING PERSISTENCE (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON PRIMAVERA (/SECURITY-SPRING)

ACERCA DE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJOS (/TAG/ACTIVE-JOB/)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANUNCIE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)