JUnit5 Grundlagen

Architektur

JUnit 5 besteht aus 3 Teilen:

* Das Jupiter API für Testautoren
* Das JUnit Platform Launcher API für BuildTools und IDE Integration
* Das Test Engine SPI für andere Testing Framework

Test Engines

JUnit 5 besteht aus 2 Test-Engine-Implementierungen:

* Jupiter Engine zur Ausführung von JUnit 5 Testfällen
* Vintage Engine zur Ausführung von JUnit 3 oder JUnit 4 Testfällen

In einem Projekt kann entweder die eine oder die andere Engine oder beide zugleich zum Einsatz kommen.

Über die Vintage Engine bleibt die Möglichkeit Frameworks zu nutzen, welche nur das JUnit 4 API unterstützen z.B. compile-testing.

Da im gleichen Projekt zusätzlich zur Vintage Engine auch die Jupiter Engine eingesetzt werden kann, können die eigenen Tests dann in JUnit 5 geschrieben werden und somit von den neuen Features provitieren.

Integration in Build Tools

* Gradle ab 4.6
* Maven Surefire ab 2.22.0
* Ant ab 1.10.3

Integration in IDEs

* IntelliJ IDEA ab 2016.2
* Eclipse ab 4.71a
* Visual Studio Code Java Testrunner ab 0.4.0
* Netbeans ab 10.0 (noch in Arbeit)

Features von JUnit 5

Features von JUnit 5

* Trennung der Testnamen von den Methodennamen

Über @DisplayName lassen sich Sätze zur fachliche Testbezeichnung festlegen. So können die Methoden immer noch alle mit testXXXX beginnen um in der Aufrufhierarchie Testmethoden sofort erkennen zu können. Im Report steht trotzdem ein sinnvoller fachlicher Name oder Satz welcher den Test beschreibt.

* Parameterisierte Tests

Über @ParameterizedTest lassen sich parametrisierte Tests erstellen. Bei diesen ist es möglich die Quelle der Parameter separat festzulegen. Beispielsweise kann mit @CsvSource eine inline kommaseparierte Liste als Datenquelle genutzt werden. Über @CsvFileSource lässt sich eine CSV Datei als Datenquelle nutzen und damit auch über diverse Tests sharen.

Allgemein ist es nicht notwendig aber wer möchte kann über @ConvertWith Konverter für den Datenimport verwenden. CSV ist nicht das einzige Format für Datenquellen, hier entstehen kontinuierlich weitere.

* Dynamische Tests

Dynamische Tests sind eine neue Art von Tests im JUnit Universum. Über die @TestFactory Annotation lassen sich dynamisch, zur Laufzeit Testfälle erzeugen. Wofür ist das gut? Angenommen wir haben 10 Tests geschrieben von denen jeder jeweils eine andere Eigenschaft zur Barrierefreiheit einer Webseite überprüft. Um diese Tests auf 3 Webseiten anzuwenden, müssten wir jeden der 10 Tests als parametrisierten Test realisieren und stets eine Liste der 3 zu prüfenden Webseiten übergeben. Mit dynamischen Tests kann man sich dieses Vorgehen sparen. Die Tests werden in einer Methode dynamisch zusammengebaut per return Befehl an das Testframework zurück gegeben. Dieses führt die Tests dann aus und lässt diese in Reports erscheinen als ob sie wie normale Tests geschrieben wurden.. **Achtung**: Die Lifecycle Methoden @Before, @BeforeAll, @After, @AfterAll werden von dynamische Tests nicht berücksichtigt - bleiben also ohne Wirkung auf die dynamischen Tests.

Eine sehr gute Einführung in das Thema findet sich man bei Baeldung

* Repeated Tests

Hierbei handelt es sich um eine weitere neue Art von Testfällen.

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

Features von JUnit 5 (cont)

Bei diesen kann über die Annotation @RepeatedTest die Anzahl der Testausführungen angegeben werden. Denkbare Einsätze für diese Testart wären zur Realisierung von Zufallstests (eine ergänzende Testart beim Modultest) oder zur Erkennung von Flackertests. Falls man diese Art Teställe für Zufallstests nutzt, sollte man darauf achten, dass die verwendeten Testparameter im Report erscheinen, da sonst bei der Auswertung nicht klar ist unter welchen Bedingungen der Test fehl geschlagen ist.. * Extensions

In JUnit 5 wurde das Konzept der @RunWith und @Rule Erweiterungen durch das Konzept der @ExtendWith ersetzt. Der Vorteil ist ein nun einheitliches Konzept zur Erweiterung. Extensions lassen sich verstehen wie Plugins welche bestimmte Extensions Points des JUnit 5 Frameworks erweitern. In der Regel nutzen Frameworks wie Mockito, Spring, Testcontainers, Wiremock, Kafka und viele mehr diese Schnittstelle aber natürlich kann man auch selbst hierüber das Framework erweitern. Zur Erweiterung stehen folgende Arten zur Verfügung:. ** Verarbeitung der Testinstanz nach Testausführung

** Bedingte Testausführung

** Lifecycle Callbacks

** Parameterauflösung

** Exception handling

Ein sehr guter Einstieg in dieses Thema findes sich wieder bei Baeldung

* Kotlin Styled Tests

Um ein Feeling von Kotlin in das JUnit Framework zu bringen wurden folgende Neuerungen eingeführt:

* @TestInstance(Lifecycle.PER_CLASS) Alle Testfälle teilen sich die gleiche Testsinstanz

* assertAll und assertThrow

* Parallele Testausführung

Mit Hilfe des junit.jupiter.execution.parallel.enabled Konfigurationsparameters kann dem Framework mitgeteilt werden, dass alle Testfälle parallel ausgeführt werden sollen. Weiterhin kann eingestellt werden, wieviel maximal parallel ausgeführt werden.

Da in der Regel nicht alle Testfälle eine parallele Ausführung vertrage, kann per @Excecution Annotation festgelegt werden, welche Testfälle im gleichen Thread ausgeführt werden sollen.

Features von JUnit 5 (cont)

Um bei der parallelen Ausführung nicht in Deadlocks zu laufen, kann per @ResourceLock Annotation beschrieben werden, auf welche Resourcen ein Testfall lesend oder schreibend zugreift. JUnit sorgt dann dafür, dass Testfälle mit ungünstiger Kombination (schreibend/lesend) nicht zu gleich ausgeführt werden.. * Warum sollte ich von JUnit 4 umsteigen?

* Möglicherweise wolltest Du schon immer einmal fachliche Sätze in Deinen Testreports sehen - das geht jetzt.

* Du wolltest einen Spring-Boot Tests als Parametrisierten Test ausführen und dabei keine super kniffligen Worarounds verstehen - das geht jetzt.

* Du wolltest schon mal zwei @RunWith Annotationen an einer Testklasse befestigen und hast gemerkt, dass das nicht geht? Mit Extensions geht es jetzt.

* Du wolltest letztens Deine Testfälle direkt aus der Testinstanz ableiten und dynamisch erstellen aber es ging nicht - jetzt kannst Du @TestFactory nutzen.

* Die Laufzeit Deiner Tests ist extrem lang weil Du viele kleine Tests hast, die eigentlich auf dem selben Objekt arbeiten könnten ohne sich zu stören aber JUnit 4 baut jedes Mal eine neue isolierte Umgebung auf und das dauert - jetzt kannst Du @TestInstance(Lifecycle.PER_CLASS) nutzen

* Du wolltests schon immer Deine Tests mit diversen Tags versehen und dann selbst per Tagauswahl entscheiden welche Tests zur Ausführung kommen - das geht jetzt.

* Du wolltest schon immer Lambda Ausdrücke und andere Java 8 Features im JUnit Test nutzen - das geht jetzt.

* Du brauchst JUnit 4 weiterhin weil ein Framework das Du nutzt nicht mehr weiterentwickelt wird und nur mit JUnit 4 klar kommt - JUnit 5 ist kompatible über die Vintage Engine. Du kannst JUnit 5 und JUnit 4 Testfälle in einem Projekt realisieren ohne störende Nebenwirkungen.

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com