Unit-Tests für CDI und Eclipse-MicroProfile-Projekte

Gunnar Hilling, Gunnar Hilling IT Consulting GmbH

Das Eclipse-MicroProfile-Projekt macht klassische JEE-Technologien wie zum Beispiel JAX-RS fit für die Microservice- und Cloud-Welt. Dabei geht es um neue Bereiche wie „Health Check", „Fault Tolerance" und „Metrics", allerdings auch um die Verwendung von CDI zur Dependency Injection. Dieser Standard ist beim Unit-Testen nach wie vor problematisch.

Im Vergleich zum Konkurrenten Spring Boot, bei dem das Test-Framework schon immer ein „first class citizen" war, ist das Testing dem CDI-Standard herzlich egal. Es handelt sich natürlich auch – im Gegensatz zu Spring – um einen offenen Standard und eben nicht um ein Produkt. Hinzu kommt, dass das Unit-Testen von Anwendungen, die wie das MicroProfile auf CDI basieren, nicht trivial ist, da für den Test zunächst ein Container zu starten ist. Dies war für eine Stand-alone-Java-Umgebung in den ersten Versionen des Standards nicht einmal spezifiziert. Es gibt das Arquillian-Framework, das Integrationstests realisieren soll, aber nicht gerade trivial zu verwenden ist.

Ein großes Hindernis im Vergleich zu Spring ist außerdem die Tatsache, dass in Spring die Injection explizit konfiguriert werden kann. Dies ist so in CDI nicht vorgesehen, sondern die Injection erfolgt ausschließlich über Annotations und CDI-Extensions. Hierdurch wird die Verwendung von Mocks als Test-spezifischer Ersatz für die echten Implementierungen erschwert.

Ein einfacher Lösungsansatz

Um Komponenten möglichst einfach testen zu können, hat der Autor aus einem Kundenprojekt heraus die JUnit-Extension „cdi-test" entwickelt. Folgende Ziele wurden dabei verfolgt:

- Möglichst einfache Anwendung, flache Lernkurve
- Schnelle Ausführung vieler Unit- oder Komponenten-Tests
- Erweiterbarkeit
- Inzwischen basierend auf JUnit 5

Die gesamte Extension enthält im Core weniger als 1.000 Zeilen Code und funktioniert mit verschiedenen Weld- und JDK-Versionen. *Listing 1* zeigt einen einfachen Test-Case.

```
@ExtendWith(CdiTestJunitExtension.class)
public class SimpleTest {
    @Inject
    private Person person;
    @Test
    public void testInjection() {
        assertNotNull(person);
    }
}
```

Listing 1

Komplette Beispiele für Unit-Tests sind auf der Homepage des Projekts bei GitHub *(siehe „https://github.com/guhilling/cdi-test")* verfügbar. Dort steht auch die aktuelle Dokumentation, die unter anderem zeigt, wie die Abhängigkeiten für einen solchen Test zu definieren sind.

Die Extension startet bei Test-Ausführung lediglich einen CDI-Container für alle Tests. Dadurch ist die Ausführungszeit auch bei großen Projekten niedrig. Die Isolation aller Tests gegeneinander ist dennoch dadurch gewährleistet, dass alle CDI-Scopes inklusive des ApplicationScope für jeden einzelnen Test neu gestartet werden. Die Testklasse selber ist übrigens keine CDI-Bean, da sie von JUnit 5 instanziiert wird. Die Extension nimmt lediglich die Injection vor.

Verwendung von Mocks

Angenommen, man möchte einen Komponenten-Test für den Service „SampleService" erstellen, der intern einen „BackendService" verwendet. Dieser soll für unseren Test gemockt werden. Dafür gibt es eine Integration mit Mockito. Ein Mock des „BackendService" wird einfach im Test für alle Konsumenten dieses Service aktiviert, indem man in der Testklasse ein Attribut von diesem Typ anlegt und mit „@Mock" annotiert. Der Test sieht nun wie in *Listing 2* aus.

```
@ExtendWith(CdiTestJunitExtension.class)
public class ServiceMockTest {
    @Mock
    private BackendService backendService;
    @Inject
    private SampleService sampleService;
    @Test
    public void createPerson() {
        Person person = new Person();
        sampleService.storePerson(person);
        verify(backendService).storePerson(person);
    }
    @Test
    public void doNothing() {
        verifyZeroInteractions(backendService);
    }
}
```

Listing 2

Die eigentliche Magie ist in dieser Klasse unsichtbar, es wird jedoch verifiziert, dass der „SampleService" tatsächlich den Mock aufgerufen hat. Die „CdiTestJunitExtension" setzt dies dadurch um, dass jede Injection durch einen Proxy geleitet wird. Die Proxies stehen unter Kontrolle der Extension und können dadurch abhängig von der aktiven Testklasse entweder auf die Ziel-Bean oder auf einen Mock verweisen.

Test-Implementierung

Es gibt allerdings noch einen weiteren häufigen Wunsch alternativ zum Mock, nämlich eine maßgeschneiderte Test-Implementierung, die entweder in allen Tests oder wiederum pro Test eine Bean ersetzt. Auf globaler Ebene kann dies durch die Annotation „@GlobalTestImplementation" erreicht werden. Soll die Aktivierung analog zu Mocks pro Test-Case möglich sein, wird „@ActivatableTestImplementation" verwendet. Die Alternative wird einfach entsprechend annotiert, der Testfall ist analog zum vorherigen Beispiel *(siehe Listing 3)*.

Um diese Implementierung in einer Testklasse indirekt in allen CDI-Beans zu verwenden, die „BackendService" benutzen, wird die Implementierung einfach zusätzlich „injected". Dies dürfte in den meisten Fällen ohnehin notwendig sein, um die Test-Implementierung vorzubereiten und die Ergebnisse und Aufrufe zu verifizieren *(siehe Listing 4)*. Auch hier sollte die Benutzung nicht allzu viele Überraschungen bereithalten.

Erweiterbarkeit des Test-Frameworks

Häufig ist es wünschenswert, bestimmte Aspekte des eigenen Projekts auch in Test-Support-Klassen oder eben in bestimmten Test-Implementierungen abzubilden.

```
@ActivatableTestImplementation
public class BackendServiceTestImplementation extends BackendService {
     […]
     public int getInvocations() {
        return invocations;
     }
}
```

Listing 3

```
@ExtendWith(CdiTestJunitExtension.class)
public class ServiceTestImplementationTest {
    @Inject
    private SampleService sampleService;
    @Inject
    private BackendServiceTestImplementation testBackendService;

    @Test
    public void callTestActivatedService() {
        sampleService.storePerson(new Person());
        assertEquals(1, testBackendService.getInvocations());
    }
}
```

Listing 4

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface BackendServiceException {
    Class<RuntimeException> value();
}
```

Listing 5

„cdi-test" unterstützt dies durch zwei einfach zu nutzende Erweiterungspunkte:

- Es gibt zwei zusätzliche CDI-Scopes: „@TestScoped" und „@TestSuiteScoped". „@TestScoped"-Beans sind dabei für einen einzelnen Test gültig, der Scope wird aber vor allen anderen Scopes aktiviert und erst nach Beendigung des Tests als Letztes deaktiviert. „@TestSuiteScoped"-Beans sind während des gesamten Testlaufs gültig.
- Speziell für Test-Implementierungen von Services kann es nützlich sein, diese per Annotation an der Testmethode oder an der Testklasse zu konfigurieren. Dies lässt sich leicht durch einen CDI-Event-Listener realisieren. Die Test-Erweiterung erzeugt Events für das Starten und die Beendigung eines Tests.

Als Beispiel folgendes Szenario: Eine Test-Implementierung für einen Service soll, gesteuert durch eine Annotation, für verschiedene Tests unterschiedliche Ergebnisse liefern. Eine Annotation an der Testmethode kontrolliert, ob eine Exception geworfen wird *(siehe Listing 5)*. Um diese Annotation auszuwerten, lässt sich die vorgestellte Test-Implementierung erweitern *(siehe Listing 6)*.

Durch die Annotation „@ActivatableTestImplementation" wird nicht nur diese Klasse für die Umlenkung von Methoden-Aufrufen auf Beans berücksichtigt, sondern gleichzeitig der Scope auf „@TestScoped" festgelegt. Dieser Scope wird während der Testausführung vor allen Standard-Scopes gestartet und erst nach deren Beendigung wieder deaktiviert. Der Test mit Prüfung auf einen vorher definierten Fehler sieht dann wie in *Listing 7* aus.

Auf Basis dieser Mechanismen sind effiziente Erweiterungen mit geringem Programmier-Aufwand möglich; zum Beispiel kann man eine „EntityManagerFactory", deren Erzeugung zeitaufwendig ist, lediglich einmal für alle Tests bauen, dann aber für jeden einzelnen Test-Case einen neuen „EntityManager" aufsetzen und außerdem die Datenbank in einen definierten Zustand versetzen.

Eclipse MicroProfile

Eclipse MicroProfile spielt inzwischen eine wichtige Rolle in der ehemaligen Java-Enterprise-Welt. Zum einen wird ein Standard zur Entwicklung von leichtgewichtigen Microservices bereitgestellt, den auch eine Reihe von Firmen unterstützt. Zum anderen lassen sich hier Technologien evaluieren, die es dann vielleicht in eine noch zu definierende Version 9 von Java EE schaffen.

Um Unit-Tests für eine MicroProfile-Anwendung zu schreiben, sind zunächst die gleichen Anforderungen wie für jede Anwendung auf CDI-Basis gegeben: Im Prinzip kommt man also mit „cdi-test" weiter. Einen Großteil der MicroProfile-Standards wird man nicht unbedingt in Unit-Tests testen; Circuit-Breaker und Health-Checks sind eher ein Fall für Systemtests.

```
@ActivatableTestImplementation
public class BackendServiceTestImplementation extends BackendService {

    private int invocations;
    private RuntimeException exceptionToThrow;

    @Override
    public void storePerson(Person person) {
        if (exceptionToThrow != null) {
            throw exceptionToThrow;
        } else {
            invocations++;
        }
    }

    protected void testStarted(@Observes @TestEvent(EventType.STARTING) ExtensionContext context) {
        context.getTestMethod().ifPresent(m -> {
            // Setzen der zu werfenden Exception
        });
    }
```

Listing 6

```
@Test
@BackendServiceException(RuntimeException.class)
public void callTestActivatedServiceWithBackendException() {
    Assertions.assertThrows(RuntimeException.class, () -> {
        sampleService.storePerson(new Person());
    });
}
```

Listing 7

```
@ApplicationScoped
public class Controller {

@Inject
@ConfigProperty(name = "some.string.property")
private String stringProperty;
[…]
}
```

Listing 8

Es gibt jedoch eine zentrale Ausnahme innerhalb einer MicroProfile-Anwendung: MicroProfile-Config. Damit definiert man eine einheitliche Möglichkeit, MicroProfile-Anwendungen zu konfigurieren, und das dürfte in Zukunft in vielen MicroProfile-Beans verwendet werden. Dies erfolgt einfach durch die Annotation „@ConfigProperty" *(siehe Listing 8)*.

Das Beispiel dürfte selbsterklärend sein. Weitere Features sind automatische Umwandlungen in die gängigsten Datentypen. Eine Eigenschaft wie „stringProperty" im obigen Beispiel muss auch tatsächlich definiert sein. Wenn dies optional sein soll, muss (logisch) „Optional" verwendet werden. In einem „cdi-test" für den oben definierten Controller muss man also einen entsprechenden CDI-Producer für die im Projekt definierten Properties implementieren. Hierzu gibt es bereits Implementierungen – warum also das Rad neu erfinden.

```
<dependency>
    <groupId>de.hilling.junit.cdi</groupId>
    <artifactId>cdi-test-microprofile</artifactId>
    <version>1.0.0</version>
    <scope>test</scope>
</dependency>
```

Listing 9

Um die Arbeit zu erleichtern, hat der Autor die „cdi-test"-Erweiterung „cdi-test-microprofile" geschrieben. Beispiele und Source-Code sind wie üblich bei GitHub verfügbar *(siehe „https://github.com/guhilling/cdi-test-microprofile")*. Bei einem bereits lauffähigen Test genügt es, diese Abhängigkeit zusätzlich zu vereinbaren *(siehe Listing 9)*. Die Bibliothek beruht auf „smallrye-config", das auch von „thorntail", ehemals „wildfly-swarm", verwendet wird.

Die Properties für das Testing werden momentan noch aus der Datei „microprofile-config.properties" gelesen, die als Test-Ressource angelegt werden sollte. In Zukunft wird es auch möglich sein, die Werte aus den „project-defaults.yml" zu verwenden, wie bei „thorntail" üblich.

Leider gibt es momentan ein unschönes Problem beim Packaging durch das „thorntail-maven"-Plug-in: Falls im Test-Scope ein „deltaspike-core-api" in einer zu neuen Version, nämlich größer 1.8,

vorhanden ist, erstellt das Plug-in kein sinnvolles „uberjar" mehr. Diese Bibliothek wird allerdings von „cdi-test" in Version 1.9.0 benötigt. Das Problem lässt sich umgehen, indem für die transitive Abhängigkeit „deltaspike-core-api" explizit die Abhängigkeit „provided" vereinbart wird *(siehe Listing 10)*. Die Ursache für dieses Problem ist allerdings im „thorntail"-Projekt zu suchen: Das Packaging sollte nicht abhängig von Test-Dependencies sein.

Custom Converter

Für eigene Datentypen ist es im Zusammenhang mit „microprofile-config" oft nützlich, eigene Converter zu definieren. Das in „cdi-test-microprofile" verwendete „smallrye-config" kann zwar schon recht intelligent auch andere als Standard-Datentypen zum Beispiel über einen passenden Konstruktor mit einem String als Argument erzeugen, man stößt jedoch recht schnell auf Fälle, in denen dies nicht geht. Der Autor lässt sich gerne seine Value-Objekte von der Immutables-Compiler-Erweiterung erzeugen *(siehe Listing 11)*. Aus diesem Interface entsteht beim Compile-Vorgang eine Implementierung „ImmutableHorse", die zur Instanziierung eine innere Builder-Klasse bereitstellt. Ein Custom-Converter für MicroProfile-Config würde dann wie in *Listing 12* aussehen.

Durch die hohe Priorität wird dieser Converter stets benutzt, selbst wenn die Config-Implementierung selbst in der Lage wäre, diesen Datentyp aus String zu erzeugen. Um den Konverter in einer Standard-Implementierung zu nutzen, muss er über das „ServiceLoader"-Framework aktiviert werden, hierzu wird im Classpath eine Datei mit dem Namen „services/org.eclipse.microprofile.config.spi.Converter" angelegt. Inhalt sind die Namen der zusätzlichen Converter „de.hilling.junit.cdi.microprofile.HorseConverter". Dieser Mechanismus funktioniert genauso auch für Unit-Tests, die mit „cdi-test-microprofile" erstellt wurden, da diese letztlich eine Standard-Implementierung nutzen.

```
<dependency>
    <groupId>org.apache.deltaspike.core</groupId>
    <artifactId>deltaspike-core-api</artifactId>
    <version>1.9.0</version>
    <scope>provided</scope>
</dependency>
```

Listing 10

```
@Value.Immutable
public interface Horse {
    String getName();
}
```

Listing 11

```
@Priority(1000)
public class HorseConverter implements Converter<Horse> {
    @Override
    public Horse convert(String value) {
        return ImmutableHorse.builder().name(value).build();
    }
}
```

Listing 12

Fazit

Das Thema „Testing" ist leider in der Vergangenheit sowohl von den JEE-Anbietern als auch vor allem von den Standardisierungsgremien sehr stiefmütterlich behandelt worden. Eclipse MicroProfile bietet für die Anwendungsentwicklung sehr gute und praxisnahe Lösungen. Der Ansatz, hier offene und praxisnahe Standards zu entwickeln, die auch in Zukunft die Wahl zwischen verschiedenen Technologien lassen, ist zu begrüßen. Der Autor meint allerdings, dass auch hier das Thema „Testbarkeit" bisher nicht ausreichend behandelt wurde. Der Erfolg von Spring und Spring-Boot im Vergleich zu JEE ist nicht zuletzt darauf zurückzuführen, dass beide es den Entwicklern leicht machen, damit entwickelte Anwendungen zu testen.

Genau wie bei JEE fehlt beim MicroProfile bis heute ein starkes Commitment der Community und der Member zum Thema „Testbarkeit". Der Autor würde sich auch deshalb über Feedback und weitere Anforderungen zum Thema sehr freuen.

Gunnar Hilling
gunnar@hilling.de

Gunnar Hilling ist nach seinem Abschluss als Dipl.-Physiker direkt in die IT gewechselt. Dazu beigetragen hat sicherlich die frühe Beschäftigung im Studium mit Unix- und Shell-Programmierung sowie mit den für das 20. Jahrhundert grandiosen NeXTstations, die die Osnabrücker Uni damals angeschafft hatte. Nach einem mehrjährigen Ausflug in die Lehre an der Uni Stuttgart hat er sich selbstständig gemacht und arbeitet seitdem im Trainingsgeschäft sowie in einer Reihe von Großprojekten hauptsächlich im Bereich „Java Enterprise". In den letzten Jahren hat sich dabei der Fokus mehr in Richtung DevOps und Integration sowie Automation verschoben.