KaffeeKlatsch
Das Magazin rund um Software-Entwicklung

ISSN 1865-682X
04/2015

Aus, Sitz, Platz!

CDI-Komponenten ohne Server testen

von Werner Eberling

Aus, Sitz, Platz!

CDI-Komponenten ohne Server testen

von Werner Eberling

Unit-Testing ist im Rahmen der Java Enterprise-Entwicklung schon immer ein unbeliebtes Stiefkind gewesen. Mit dem CDI-Standard bietet sich hier jedoch wieder die Möglichkeit zur Rückbesinnung auf klassische, lokale *Test-First*-Ansätze, wie sie aus der POJO-Entwicklung bekannt sind. Dieser Artikel zeigt wie einfach lokales Testen mit CDI sein kann.

„Ich kann keine vernünftigen Unit-Tests schreiben, ohne ständig den *Application Server* hoch und runter zu fahren." So oder so ähnlich hören sich die Standardargumente gegen testgetriebene Ansätze im Bereich der *Java EE*-Entwicklung an. Auch wenn hier in den letzten Jahren teilweise bereits ein Umdenken stattgefunden hat, begegnet man diesen Argumenten noch immer viel zu häufig in der freien Java Enterprise-Wildbahn.

Leere Versprechungen und neue Möglichkeiten

Zugegeben, lokale Container zur Ermöglichung von *Unit*-Tests wurden häufig vom Standard versprochen aber leider nie geliefert. Einige Hersteller haben zwar proprietäre Lösungen hierfür entwickelt, diese waren aber entweder zu schwergewichtig oder im Umkehrschluss nur wenig für den produktiven Einsatz im Java Enterprise Server geeignet. Und eine Anwendung mit einem lokalen Container z. B. *OpenEJB* zu testen und später in einem anderen z. B. *Glassfish*, *JBoss* oder *Weblogic* zu betreiben, stellt wegen der unterschiedlichen Laufzeitumgebungen oft die Sinnhaftigkeit der lokalen Tests in Frage.

Auch der jüngste Sprössling der Java EE-Komponentenfamilie, der CDI-Standard, wurde ursprünglich als server-seitiger Komponentenstandard spezifiziert. Allerdings hat sich das Blatt hier inzwischen gewandelt, so dass auch lokale CDI-Container verfügbar sind. So bieten beispielsweise die Referenzimplementierung *Weld* [1] von JBoss und die *Apache*-Implementierung *Open WebBeans* [2] CDI-Containerimplementierungen an, die auch für die *Java SE*-Umgebung geeignet sind. Hinzu kommt, dass im Rahmen des Apache-Projektes *Deltaspike* [3] eine einheitliche Schnittstelle zur Steuerung dieser Container bereitgestellt wird, womit einerseits das Problem der unterschiedlichen Laufzeitumgebungen gelöst und gleichzeitig der allseits oft so befürchtete *Vendor Lock-in* abgefedert wird.

Ein erster Test

Was benötigt man nun um seine CDI-Komponenten lokal zu testen? Nun, erst einmal den CDI-Container der Wahl (im folgenden Beispiel: *Weld* [1]), das *Deltaspike Container Control-API* [4] sowie das passende *Deltaspike*-Adapterprojekt für den gewählten Container. Zusätzlich bietet sich das *Deltaspike Core Modul* [4] an, um u. a. von den dort vorhandenen Tools zum *Bean*-Zugriff zu profitieren.

```
groupId:
   org.apache.deltaspike.core
artifactId:
   deltaspike-core-impl

groupId:
   org.apache.deltaspike.core
artifactId:
   deltaspike-core-impl

groupId:
   org.apache.deltaspike.cdictrl
artifactId:
   deltaspike-cdictrl-api

groupId:
   org.apache.deltaspike.cdictrl
artifactId:
   deltaspike-cdictrl-weld

groupId:
   org.jboss.weld.se
artifactId:
   weld-se
```

Listing 1: *Maven Dependencies* für Deltaspike und Weld

Sind alle benötigten Bibliotheken eingebunden, steht dem ersten *Unit*-Test nichts mehr im Wege. In diesem muss vor der Durchführung der eigentlichen Tests erst einmal der CDI-Container gestartet werden.

```
CdiContainer cdiCon =
   CdiContainerLoader.getCdiContainer();
cdiCon.boot();
cdiCon.getContextControl().StartContexts();
```

Beim Start des Containers wird der Klassenpfad nach CDI-Komponenten durchsucht und alle gefundenen CDI-Beans deployed, inklusive Lebenszyklus und *Injections*. Die Methode *startContexts()* startet dabei alle Standardkontexte. Wahlweise können mit *startContext()/ stopContext()* die einzelnen Kontexte auch jeder für sich gestartet und gestoppt werden. Dies bietet sich vor allem vor bzw. nach jeder Testmethode an, da der Start des Containers schnell zeitintensiv werden kann und nur einmal pro Testklasse erfolgen sollte.

Nach Start des Containers ermittelt die Testmethode, unter Zuhilfenahme des *Deltaspike Core API*, die gewünschte Komponente und führt ihre Prüfungen durch.

```
Customer customer = BeanProvider.
   getContextualReference(Customer.class);
Assert.
   assertEquals("Doe",customer.getName() );
```

Sind alle Tests durchgeführt, kann der Container wieder heruntergefahren werden:

```
cdiContainer.shutdown();
```

Etwas mehr Komfort

Auch wenn hier mit wenigen Zeilen ein Container gestartet und die gewünschte Komponente ermittelt werden kann, erscheint das Vorgehen im Zeitalter der *Dependency Injection-Frameworks* doch etwas „hemdsärmlich“. Grundsätzlich könnte zwar mit wenigen Zeilen eine Injection der gewünschten Komponenten in den *Unit*-Test realisiert werden, hier gibt es aber eine komfortablere Variante: das *Deltaspike Test Control Module* [4]. Dieses Projekt stellt einen *JUnit TestRunner* zur Verfügung, der sich um Container-Verwaltung und Injections kümmert. Alles was der Test nun noch tun muss, ist die Deklaration eines entsprechenden *Injectionpoints*.

```
@RunWith(CdiTestRunner.class)
public class CustomerServiceTest {
   @Inject
   CustomerService customerService;
//...
   @Test
   public void testeSomething(){
      Customer savedCustomer =
         customerService.saveCustomer(new Customer());
    Assert.assertNotNull(savedCustomer);
}
```

Listing 2: *Unit*-Test mit *CdiTestRunner*

Standardmäßig werden bei Verwendung des *CdiTest-Runners* alle Standardkontexte gestartet. Sollen nur bestimmte Kontexte aktiv sein, so kann dies mit der Annotation *@TestControl* erreicht werden.

Und die Abhängigkeiten?

Nun besteht eine CDI-Anwendung nicht nur aus den eigenen, selbst geschriebenen Komponenten. Gerade der einfache Zugriff auf Drittservices und Server-Dienste macht die Java EE Komponenten-Entwicklung interessant. Wenn beispielsweise eine Komponente Zugriff auf den angemeldeten Benutzer benötigt, so lässt sie sich das passende *Principal Objekt* einfach injizieren.

```
@Inject
private Principal currentUser;
```

Doch im *Unit*-Test gibt es keinen Applikations-Server, der einen entsprechenden *Security-Service* anbieten

würde, um den *Principal* zu liefern. Hier muss die *Dummy*-Implementierung eines Test-*Principals* die Arbeit übernehmen. Doch wie gelangt diese Implementierung nun an die richtigen Stellen in den Komponenten?

Dank des *Producer*-Konzepts von CDI reicht es aus, eine entsprechende *Producer*-Methode im Rahmen der Test-Klassen bereitzustellen. Den Rest erledigen die Standard CDI-Mechanismen.

```
private class PrincipalDummy
  implements Serializable, Principal {
  @Override
  public String getName() {
    return "TestDummy";
  }
}

@Produces
@SessionScoped
public Principal createDummyPrincipal() {
  return new PrincipalDummy();
 }
```

Wird das gewünschte Objekt bereits im Anwendungs-Code über eine *Producer*-Methode erzeugt, muss der *Dummy-Producer* von der entsprechenden Klasse abgeleitet werden und mit *@Spezializes* als Spezialisierung definiert werden.

Drittservices „abhängen“

Auf ähnliche Art und Weise lassen sich auch Drittservices, die durch *Dependency Injection* referenziert werden, durch *Mockups* ersetzen.

```
@Inject
private AuditService auditService;
```

Wird der entsprechende Service nicht über einen *Producer* erzeugt, sondern ist direkt als CDI-Komponente annotiert, so muss der *Mockup* als Spezialisierung von der Klasse der Originalimplementierung erben. Zusätzlich benötigte Methoden zur Kontrolle der Client-Aufrufe werden dabei direkt in der Spezialisierung definiert.

```
@Specializes
public class AuditServiceMockupImpl
  extends AuditServiceImpl {
  private Principal userReported;
 @Override
  public void logAction(Principal user, String action){
    userReported = user;
  }
  // Zusaetzliche Methode fuer den Test
  public Principal getUserReported(){
    return userReported;
  }
}
```

Um aus dem *Unit*-Test Zugriff auf den verwendeten *Mock* zu bekommen, genügt es nun einen entsprechenden *Injectionpoint* zu definieren und schon sieht der Test den Service als *Mockup*, die zu testenden Komponenten weiterhin den „altbekannten“ Drittservice.

```
@Inject
AuditServiceMockupImpl auditMock;
```

Persistenz und Transaktionen

Nutzt die zu testende Komponente *JPA* für den Zugriff auf die Datenbank, so liegt die Grundlage der Testbarkeit im Anwendungsdesign. Sofern die *EntityManager* per *@Inject* in die Bean injiziert werden, also eine *Producer Bean* existiert, die als Adapter zwischen der Injection mittels *@PersistenceContext* und der Erzeugung nach CDI-Standard fungiert, reicht es aus nach bereits vorgestelltem Muster eine Spezialisierung des *Entity Manager Producers* zu implementieren. Im Zusammenspiel mit einer lokalen, oder besser noch *In-Memory*-Datenbank können sämtliche JPA-Funktionalitäten im Rahmen des Tests genutzt werden.

```
@Specializes
@ApplicationScoped
public class LocalEntityManagerProducer
  extends EntityManagerProducer {
  private EntityManagerFactory emf;
  //...
  @Produces @RequestScoped
  public EntityManager getEntityManager() {
    return emf.createEntityManager();
  }
  public void closeEntityManager(
    @Disposes EntityManager em){
    em.close();
  }
}
```

Für die von JPA benötigten Transaktionen sorgt ein Transaktions-*Interceptor* inkl. entsprechendem *Interceptor Binding*. Wer dies nicht selbst schreiben möchte, dem sei das JPA Modul von Deltaspike [4] ans Herz gelegt, das Beides als CDI-Eweiterung bereitstellt.

Resümee

Dank der lokal verfügbaren CDI-Container und den *CDI-Utilities* von Deltaspike gibt es keine Ausreden mehr. Die testgetriebene Java EE-Entwicklung ist damit nicht nur „machbar“ sondern praktische, handhabbare Realität geworden. Ein Grund mehr auf den eigentlich gar nicht mehr so neuen CDI-Standard „umzusatteln“.

Doch was machen all die „reinrassigen“ *EJB*-Projekte, die (noch) kein CDI nutzen? Auch hierfür gibt es Lösungen:

Da EJBs grundsätzlich CDI-Komponenten sind, ist es mit etwas „CDI-Magie“ möglich, diese auch ohne explizite Annotationen im CDI-Container zu betreiben. Details hierzu finden sich z. B. im Rahmen des *BeanTest-Projektes* der Kollegen von NovaTec Consulting [5].

Alternativ bietet Apache Deltaspike Unterstützung für *OpenEJB* an [6], so dass der lokale EJB-Container auf die selbe Art und Weise genutzt werden kann, wie der CDI-Container im Rahmen dieses Artikels.

Auf welche Variante nun auch immer die Wahl fällt: Happy testing!

Ressourcen

[1] Weld
http://weld.cdi-spec.org

[2] OpenWebBeans
http://openwebbeans.apache.org

[3] Apache Deltaspike *CDI Extensions*
https://deltaspike.apache.org

[4] Deltaspike *Modulübersicht*
http://deltaspike.apache.org/documentation/modules.html

[5] GitHub *BeanTest*
https://github.com/NovaTecConsulting/BeanTest

[6] Deltaspike *Open EJB Integration*
http://deltaspike.apache.org/documentation/test-control.html#_openejb

Kurzbiografie

Werner Eberling ist Principal Consultant und Technical Lead bei der MATHEMA Software GmbH in Erlangen. Seit 1999 beschäftigt er sich mit verteilten Systemen, sein Fokus liegt dabei auf CORBA und der JEE. Neben seiner Projekttätigkeit hält er Technologie-Trainings und ist als Sprecher auf nationalen und internationalen Konferenzen anzutreffen. Sein aktuelles Steckenpferd ist die Anwendungsentwicklung für mobile Endgeräte bzw. deren Anbindung an Enterprise Systeme. Werner Eberling ist u.a. Autor des Buches „Enterprise JavaBeans 3.1 – Das EJB-Praxisbuch für Ein- und Umsteiger“, erschienen im Hanser Verlag.