Bohne und Malz

Bean-Testing von Java EE-Anwendungen mit CDI

Carlos Barragan, Gerhard Wanner, Dieter Baier

Unit-Tests sind der Kern jeder Strategie, qualitativ hochwertige Software herzustellen. Softwaresysteme ohne ausreichende Abdeckung an Unit-Tests degenerieren bei fachlichen oder technischen Änderungen schnell und werden, eher früher als später, unwartbar. Im Umfeld von Java EE erschwert allerdings die Notwendigkeit einer Ablaufumgebung das Unit-Testing. Es gibt zwar verschiedene Ansätze für dieses Problem, aber entweder sind diese aufwendiger als gewohnt oder die Ausführungsdauer der Tests verhindert die übliche Feedback-Geschwindigkeit. Bean-Testing ist ein schnelles und schlankes Testverfahren für Java EE-Anwendungen unter Einsatz von CDI. Im Gegensatz zu Unit-Tests, bei denen die Test-Einheiten üblicherweise einzelne Methoden sind, werden ganze Beans inklusive ihrer Abhängigkeiten getestet.

Bean-Tests für Java EE-Anwendungen

Ausgangspunkt des vorgestellten Verfahrens war die Entwicklung einer Java EE-Anwendung für Bewerbungsprozesse. Im Rahmen dieses Projektes wurde nach Möglichkeiten gesucht, Unit-Tests zu erstellen und durchzuführen. Aus dem Projekt heraus wurden dazu folgende Anforderungen an das Unit-Testing dieser Java EE-Anwendung gestellt:

- Das Feedback soll schnell sein, ähnlich zu dem eines normalen JUnit-Tests.
- Abhängigkeiten sollten automatisch injiziert werden, so als ob die Anwendung auf dem Applikationsserver laufen würde. Dabei soll das Mocken der Abhängigkeiten nicht notwendig sein.

Das erste betrachtete Werkzeug für die Umsetzung der genannten Anforderungen war Arquillian [Arquillian] von JBoss. Eigentlich ist Arquillian eher ein Tool für Integrationstests und weniger fürs Unit-Testing. Grund hierfür ist, dass Arquillian die Anwendung tatsächlich auf einen Applikationsserver deployt, mit dem Vorteil, dass es sich dabei um den ganzen Deployment-Ablauf kümmert. Auf Arquillian basierende Tests ermöglichen es, eine Anwendung ohne größeren Aufwand innerhalb eines Containers zu testen. Allerdings ist dieses Verfahren für Unit-Tests weniger gut geeignet, da wegen des Deployments die erforderliche Feedback-Geschwindigkeit nicht erreicht wird.

Eine zweite betrachtete Möglichkeit war der Einsatz des Frameworks Mockito [Mockito]. Mockito wurde in oben genanntem Projekt bereits für den Test der Geschäftslogik eingesetzt. Durch Mockito können die Services gemockt werden, die der Container normalerweise mittels Dependency Injection (DI) zur Verfügung stellt. Je nachdem, was getestet werden soll, fallen bei der Erstellung dieser Mocks in Mockito Aufwände an. Deren Umfang ist zwar nicht allzu groß, jedoch muss beim Einsatz von Mockito klar sein, welche Services gemockt werden sollen und welche nicht. Zudem sinkt zwangsläufig die Testqualität, da das Mocken eines Service eine neue Fehlerquelle darstellt. Für unsere Unit-Tests stellte sich die Frage, wie vorgegangen wird, wenn auch solche gemockten Services getestet werden sollen? Letztendlich besteht ein großer Teil der Geschäftslogik aus der Zusammenarbeit mehrerer solcher Services.

CDI – Contexts and Dependency Injection

CDI ist eine Ende 2009 als JSR 299 [JSR299] veröffentlichte Spezifikation des Java EE-Stacks. Ganz aktuell erschien CDI 1.1 [JSR346] als Teil von Java EE 7. Es gibt mehrere CDI-Implementierungen mit Weld [WeldHome] als Referenzimplementierung. Andere Implementierungen von CDI sind CanDI [CanDI] und OpenWebBeans [OWB].

Weld ist ein Framework für DI, welches ursprünglich für das Projekt JBoss Seam entwickelt wurde. Aus diesem Framework heraus entstand die CDI-Spezifikation. Eine erweiterte Version von Weld, Weld SE, kann auch außerhalb eines Containers verwendet werden, sodass sich CDI auch bei einer normalen Java SE-Anwendung einsetzen lässt. Bis auf ganz wenige Ausnahmen behält CDI seine volle Funktionalität auch außerhalb des Containers bei, das heißt, nahezu alle Features funktionieren *as advertised*. Die Dokumentation von Weld schreibt dazu Folgendes: „Zusätzlich zur verbesserten Integration des Java EE-Stacks definiert die CDI-Spezifikation ein Framework für typsicheres und zustandsbehaftetes DI, welches einen großen Bereich an Anwendungsarten abdeckt. Weld stellt mit einfachen Mitteln sicher, dass CDI auch unter Java SE ausgeführt werden kann und unabhängig von irgendwelchen Java EE-APIs ist“ (angelehnt an [Weld]). Dies bedeutet, dass Weld SE bis auf wenige Ausnahmen, beispielweise asynchrone Events, alle CDI-Features unterstützt.

CDI als Tool für Bean-Tests

Warum ist nun gerade CDI für das Bean-Testing sehr gut geeignet? Am besten zeigt das der Vergleich mit den bereits genannten anderen Möglichkeiten.

Im Gegensatz zu Arquillian ist die Ablaufgeschwindigkeit von Tests mit CDI deutlich schneller. Die Tests werden in Millisekunden durchgeführt, denn es muss kein Applikationsserver gestartet werden. Sogar ein Embedded Applikationsserver braucht länger zum Starten als die CDI-Umgebung. Auf Details dazu wird später noch eingegangen.

Anders als Mockito werden bei CDI keine Abhängigkeiten gemockt, sondern sie werden von CDI injiziert. Somit kann der Entwickler die EJBs unverändert testen, lediglich eine geringfügige Anpassung der Konfiguration kann notwendig sein. Dadurch wird die Zeit für die Erstellung der Mocks eingespart und mit den Bean-Tests werden die tatsächlichen Abhängigkeiten und der Quellcode getestet. Bei CDI gibt es einen einmaligen Aufwand, um die Umgebung zur Verfügung zu stellen. Wenn die Umgebung vorhanden ist, können die EJBs direkt ohne Mocking getestet werden.

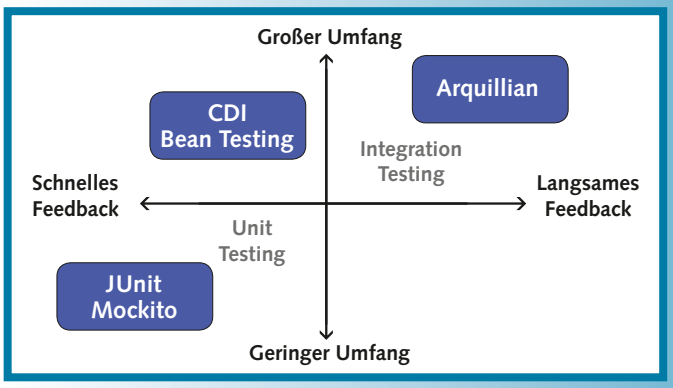

Abb. 1: CDI-Bean-Test im Vergleich zu Unit- und Integrationstest

Abbildung 1 stellt die Bean-Tests den Unit-Tests mit Mockito und den Integrationstests mit Arquillian gegenüber. CDI-Bean-Tests ermöglichen sowohl einen großen Umfang (Test der Abhängigkeiten) als auch ein schnelles Feedback (Tests werden mit ähnlicher Geschwindigkeit wie Unit-Tests durchgeführt). Als weiterer Vorteil kommt hinzu, dass Tests wesentlich schneller zu erstellen sind, da Abhängigkeiten nicht gemockt werden müssen.

Beispiel

Das Beispiel greift ein typisches Java EE-Szenario auf: EJBs referenzieren andere EJBs und Entitäten werden mittels des EntityManagers persistiert. Das Beispiel besteht aus zwei EJBs: `MyEJBService` und `MyOtherEJBService`. `MyEJBService` referenziert `MyOtherEJBService`. Beide EJBs greifen auf den EntityManager zu, um Entitäten zu persistieren bzw. zu lesen (s. Abb. 2). Die Referenzen werden zur Laufzeit im Container aufgelöst. Darüber hinaus übernimmt der Container das Transaktionshandling und die Erstellung des EntityManagers.

Das Beispiel zeigt, wie CDI eingesetzt werden kann, um eine typische Java EE-Anwendung inklusive ihrer tatsächlichen Abhängigkeiten zu testen. Außerdem wird vorgestellt, wie Services, beispielsweise Transaktionen, die normalerweise vom Container zur Verfügung gestellt werden, auch durch CDI angeboten werden können.

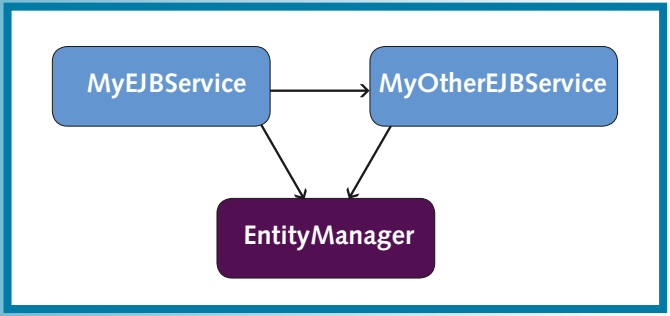

Abb. 2: Abhängigkeiten der EJBs und Entities im Beispiel

Setup der Umgebung

Die nachfolgenden Schritte zeigen das Setup der Umgebung. Sie sind nur einmalig durchzuführen und beinhalten die Einrichtung von Maven sowie die Nutzung einer Hilfsklasse zur einfachen Nutzung von Weld.

Maven einrichten

Für das Beispiel müssen die Abhängigkeiten zu Weld (enthält die Implementierung von CDI) und zum EJB 3.1-API (enthält die CDI- und EJB-Annotationen sowie den BeanManager) definiert werden (Listing 1). Für die zur Laufzeit benötigte Java EE-Implementierung nutzen wir Geronimo [Bien10]. Selbstverständlich würden an dieser Stelle auch die entsprechenden Bibliotheken von beispielsweise JBoss oder Glassfish funktionieren

Eine weitere Abhängigkeit definieren wir zu Apache DeltaSpike [DeltaSpike]. Diese Sammlung portabler CDI-Extensions und -Utilities wird später zur Initialisierung des CDI-Containers und zur Modifikation des Metamodells der Beans verwendet.

```
<dependencies>
  <dependency>
    <groupId>org.jboss.weld.se</groupId>
    <artifactId>weld-se</artifactId>
    <version>2.0.1.Final</version>
    <type>jar</type>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-ejb_3.1_spec</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.deltaspike.core</groupId>
    <artifactId>deltaspike-core-impl</artifactId>
    <version>0.4</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.deltaspike.cdictrl</groupId>
    <artifactId>deltaspike-cdictrl-weld</artifactId>
    <version>0.4</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Listing 1: Definition der Abhängigkeit in Maven

Ganz wichtig dazu ist der Eintrag des JBoss-Repositories in pom.xml, da Weld SE im Maven Central Repository nicht vorhanden ist (Listing 2).

```
<repositories>
  <repository>
    <id>JBoss</id>
    <name>JBoss repository</name>
    <url>https://repository.jboss.org/nexus/content/groups/public/</url>
  </repository>
</repositories>
```

Listing 2: Erweiterung pom.xml für die Nutzung von Weld SE

Hilfsklasse für die Weld-Nutzung

Die Klasse `BeanManagerHelper` übernimmt die Initialisierung des Weld-Containers und bietet Methoden zur Erstellung von „managed“ Instanzen an (Listing 3). Die Erstellung von Tests vereinfacht sich dadurch, weil der `BeanManagerHelper` sich um die Erstellung der Instanzen kümmert.

```
import java.lang.annotation.Annotation;
import javax.enterprise.inject.spi.BeanManager;
import org.apache.deltaspike.cdise.api.CdiContainer;
import org.apache.deltaspike.cdise.api.CdiContainerLoader;
import org.apache.deltaspike.core.api.provider.BeanProvider;

public class BeanManagerHelper {
  private CdiContainercdiContainer;

  public BeanManagerHelper() {
    cdiContainer = CdiContainerLoader.getCdiContainer();
    cdiContainer.boot();
    cdiContainer.getContextControl().startContexts();
  }
  public<T> T createInstance(Class<T>clazz) {
    returnBeanProvider.getContextualReference(clazz);
  }
  public<T> T createInstance(Class<T>clazz,
    Annotation... annotations) {
    returnBeanProvider.getContextualReference(clazz, annotations);
  }
  public BeanManager getBeanManager() {
    returncdiContainer.getBeanManager();
```

```
  }
  public void shutdown() {
    cdiContainer.shutdown();
  }
}
```

Listing 3: Vereinfachte Erstellung von „managed“ Instanzen mit dem BeanManagerHelper

Listing 3 zeigt, wie der CDI-Container mit Hilfe der Utility-Klasse `CdiContainerLoader` aus dem DeltaSpike-Framework im Konstruktor der Klasse initialisiert bzw. in der Methode `shutdown()` „ausgeschaltet“ wird. Mit Hilfe dieser Utility-Klasse werden auch alle Kontexte (Application-, Request-Kontext usw.) gestartet.

Nach dieser Initialisierung können die „managed“ Instanzen in den createInstance-Methoden mit Hilfe der Utility-Klasse `BeanProvider` aus dem DeltaSpike-Framework erstellt werden.

Die Beispielanwendung

Wir bereits erwähnt, besteht das Beispiel aus zwei EJBs (Listing 4, Listing 5). Die EJBs sind stateless und greifen auf den EntityManager zu.

`MyEJBService` hat eine Referenz auf `MyOtherEJBService`, deren Referenz zusammen mit dem EntityManager zur Laufzeit vom Container aufgelöst werden soll. In der Methode `MyEJBService.doSomethingWithTheOtherService()` wird die Methode `MyOtherEJBService.doSomething()` aufgerufen. Durch diesen Aufruf wird demonstriert, dass die Referenz von CDI zur Verfügung gestellt wird. Andernfalls würde eine `NullPointerException` geworfen werden. In dieser Methode wird auch eine neue Instanz von `MyEntity` erstellt und persistiert. Dies zeigt, dass der EntityManager durch CDI eingefügt wurde.

```
@Stateless
public class MyEJBService {
  @EJB
  MyOtherEJBService otherService;

  @PersistenceContext(unitName="db2")
  EntityManager em;

  public void doSomethingWithTheOtherService() {
    otherService.doSomething();
    MyEntity entity = new MyEntity();
    entity.setName("Hello");
    em.persist(entity);
    System.out.println("Entity persisted!");
  }
}
```

Listing 4: Der erste Service des Beispiels: MyEJBService

Die Methode `MyOtherEJBService.doSomething()` macht nichts anderes, als eine Nachricht in der Konsole zu zeigen. Die Methode `getAllEntities()` gibt alle MyEntity-Einträge mit Hilfe des EntityManagers zurück. Da in `MyEJBService` eine Instanz von `MyEntity` persistiert wurde, soll es durch `MyOtherEJBService.getAllEntities()` möglich sein, diese wieder zu lesen.

```
@Stateless
public class MyOtherEJBService {

  @PersistenceContext(unitName="db2")
  EntityManager em;

  public void doSomething() {
    System.out.println("MyOtherEJBService did something");
  }
  public Collection<MyEntity> getAllEntities() {
    return em.createQuery("Select E from MyEntity as E",
    MyEntity.class).getResultList();
  }
}
```

Listing 5: Der zweite Service des Beispiels: MyOtherEJBService

Bean-Test

Der Test wird vom JUnit-Framework ausgeführt. Es handelt sich somit um einen typischen JUnit-Test (Listing 6). Durch die Methode `initialiseBeanManager()`, die mit `@BeforeClass` annotiert ist, wird der `BeanManagerHelper` initialisiert. In der Test-Methode `shouldInjectEJBAsCDIBean()` wird eine Instanz von `MyEJBService` durch den `BeanManagerHelper` erstellt und die Methode `doSomethingWithOtherService()` aufgerufen. Damit wird gezeigt, dass die Referenz auf `MyOtherEJBService` und den EntityManager vom CDI-Container aufgelöst wurde.

Darüber hinaus wird die Test-Methode `MyOtherEJBService.getAllEntities()` aufgerufen, um nachzuweisen, dass eine Instanz der Klasse `MyEntity` in der Datenbank existiert. Da diese Instanz vom `MyEJBService` persistiert wurde, bedeutet dies, dass sich beide EJBs den gleichen Persistence-Kontext innerhalb des Aufrufs der Test-Methode teilen. Das simuliert das Laufzeitverhalten in einem Container eines Java EE-Applikationsservers.

```
public class TestEJBInjection {
  private static BeanManagerHelper bm;
  @BeforeClass
  public static void initialiseBeanManager() {
    bm = new BeanManagerHelper();
  }
  @Test
  public void shouldInjectEJBAsCDIBean() {
    MyEJBService myService = bm.createInstance(MyEJBService.class);
    myService.doSomethingWithTheOtherService();
    MyOtherEJBService myOtherService =
      bm.createInstance(MyOtherEJBService.class);
    assertTrue(myOtherService.getAllEntities().size() > 0);
  }
}
```

Listing 6: Bean-Test in TestEJBInjection

Der CDI-Container kümmert sich um das DI jeder Bean und des EntityManagers. Allerdings haben wir keine CDI-Beans, sondern EJBs. Wie kann der CDI-Container die EJBs erkennen bzw. injizieren? Dank einem der mächtigsten Features von CDI – den CDI-Extensions. Bei der Initialisierung des CDI-Containers wird der ganze Classpath nach möglichen CDI-Beans (eigentlich ist jedes POJO eine potenzielle CDI-Bean), Interceptors, Events usw. gescannt. Auf diese Weise erkennt der CDI-Container, wo die Dependencies zu finden sind. Wie schon erwähnt, haben wir aber keine CDI-Bean. An dieser Stelle kommt die CDI-Extension zum Einsatz (Listing 7).

```
import info.novatec.cdi.test.utils.Transactional;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import javax.enterprise.context.RequestScoped;
import javax.enterprise.event.Observes;
import javax.enterprise.inject.spi.AnnotatedField;
import javax.enterprise.inject.spi.AnnotatedType;
import javax.enterprise.inject.spi.Extension;
import javax.enterprise.inject.spi.ProcessAnnotatedType;
```

```
import javax.inject.Inject;
import javax.persistence.PersistenceContext;
import
  org.apache.deltaspike.core.util.metadata.AnnotationInstanceProvider;
import
  org.apache.deltaspike.core.util.metadata.builder.AnnotatedTypeBuilder;

public class CDITestExtension implements Extension {
  public<X> void processInjectionTarget(
      @Observes ProcessAnnotatedType<X> pat) {
    if(pat.getAnnotatedType().isAnnotationPresent(Stateless.class)) {
      createEJBWrapper(pat, pat.getAnnotatedType());
    }
  }
  private<X> void createEJBWrapper(ProcessAnnotatedType<X> pat,
      finalAnnotatedType<X> at) {
    Transactional transactionalAnnotation=
      AnnotationInstanceProvider.of(Transactional.class);
    RequestScopedrequestScopedAnnotation=
      AnnotationInstanceProvider.of(RequestScoped.class);
    AnnotatedTypeBuilder<X> builder=
      newAnnotatedTypeBuilder<X>().readFromType(at);
    builder.addToClass(transactionalAnnotation).
      addToClass(requestScopedAnnotation) ;
    Inject injectAnnotaiont= AnnotationInstanceProvider.of(Inject.class);
    for(AnnotatedField<? super X>field:at.getFields()) {
      if ((field.isAnnotationPresent(EJB.class) ||
        field.isAnnotationPresent(PersistenceContext.class))&&
        ! field.isAnnotationPresent(Inject.class)) {
        builder.addToField(field, injectAnnotaiont);
      }
    }
    // Wrapper anstatt der tatsächlichen Bean setzen
    pat.setAnnotatedType(builder.create());
  }
}
```

Listing 7: Dependency-Injection mittels CDITestExtension

Die Extension wird angewendet, sobald der CDI-Container gestartet wird. Dies passiert noch, bevor die Beans zur Verfügung gestellt werden, sodass Injection-Points, Bean-Definitionen usw. geändert werden können, ohne den Code anpassen zu müssen. Die Extension macht Folgendes:

- Es wird überprüft, ob die entsprechende Bean mit **`@Stateless`** markiert ist.
- Es werden die Annotationen **`@Transactional`** und **`@RequestScoped`** zu der Bean „hinzugefügt". Es handelt sich dabei um eine Änderung am Metamodell. Der Bytecode wird also nicht modifiziert.
- Es wird überprüft, ob die Bean Felder hat, die mit **`@EJB`** markiert sind. Falls ja, werden diese Felder mit **`@Inject`** markiert.
- Die modifizierte Bean wird anstelle der ursprünglichen Bean zur Verfügung gestellt.

Grundsätzlich wird jede EJB in eine CDI-Bean konvertiert. Diese Konvertierung findet statt, ohne dass der Quellcode in irgendeiner Form angepasst werden muss. Für die Modifikation des Metamodells verwenden wir erneut Apache DeltaSpike.

Im Prinzip kann der EJB-Wrapper auch „manuell", also ohne die Hilfe von DeltaSpike, erstellt werden. Das bedeutet aber viel Boilerplate-Code.

Was macht die Annotation **`@Transactional`**? Das ist keine standardisierte Annotation, sondern lediglich ein ganz normales CDI-InterceptorBinding (Listing 8). Den zugehörigen CDI-Interceptor zeigt Listing 9.

```
@InterceptorBinding
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Transactional {
  @Nonbinding TransactionAttributeType transactionAttribute() default
    TransactionAttributeType.REQUIRED;
}
```

Listing 8: CDI-InterceptorBinding @Transactional

```
@Interceptor
@Transactional
public class TransactionalInterceptor {
  @Inject
  @PersistenceContext
  EntityManager em;

  private final Logger logger =
    LoggerFactory.getLogger(TransactionalInterceptor.class);
  private static final ThreadLocal<Stack<String>> stackHolder=
      new ThreadLocal<Stack<String>>() {
    @Override
    protected Stack<String> initialValue() {
      return new Stack<String>();
    }
  };
  @AroundInvoke
  public Object manageTransaction(InvocationContext ctx)
      throws Exception {
    TransactionAttribute transaction =
      ctx.getTarget().getClass().getAnnotation(
        TransactionAttribute.class);
    logger.debug("EntityManager in interceptor {}", em);
    if (!em.getTransaction().isActive()) {
      logger.debug("Transaction is not active. " +
        "A new transaction will be activated");
      em.getTransaction().begin();
    }
    String clazzName = ctx.getMethod().getDeclaringClass().getName();
    stackHolder.get().push(clazzName + "." + ctx.getMethod().getName());
    Object result = ctx.proceed();
    if (em.isOpen() && em.getTransaction().isActive() &&
        stackHolder.get().isEmpty()) {
      logger.debug(
        "Transaction is active and will be commited for {}.{}",
        clazzName, ctx.getMethod().getName());
      em.getTransaction().commit();
    }
    return result;
  }
}
```

Listing 9: CDI-Interceptor TransactionalInterceptor

Der Interceptor startet eine Transaktion und führt ein Commit aus, wenn es keine Aufrufe mehr im Stack gibt. Somit werden die EntityManager-Operationen innerhalb einer Transaktion durchgeführt. Die Implementierung des Interceptors ist für die Durchführung solcher Bean-Tests gedacht. Sie sollte auf keinen Fall in der Produktion eingesetzt werden: Die Implementierung greift auf das Transaction-Objekt des EntityManagers zu, während in einem Java EE-Container eher der TransactionManager verwendet wird. Auch wird die Transaktion im Container anders propagiert.

Im Interceptor sowie in den EJBs wird ein EntityManager injiziert. Dazu muss dem CDI-Container mitgeteilt werden, wie der EntityManager zur Verfügung gestellt wird. Hierzu verwenden wir einen CDI-Producer (Listing 10).

```
@RequestScoped
public class EntityManagerProducer {
  private EntityManagerFactory emf;
  private EntityManager em;

  @PostConstruct
  private void initializeEntityManagerFactory() {
    HashMap<String, String> properties = new HashMap<String, String>();
```

```
    properties.put("hibernate.connection.driver_class",
      "org.h2.Driver");
    properties.put("hibernate.connection.url",
      "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1");
    properties.put("hibernate.dialect",
      "org.hibernate.dialect.H2Dialect");
    properties.put("hibernate.hbm2ddl.auto", "create");
    properties.put("hibernate.show_sql", "true");
    emf=Persistence.createEntityManagerFactory("test", properties);
  }
  @Produces
  public EntityManager getEntityManager(InjectionPoint ip) {
    PersistenceContext ctx =
      ip.getAnnotated().getAnnotation(PersistenceContext.class);
    //An dieser Stelle hat man die Möglichkeit,
    //die Konfiguration zur Persistenz zu verändern.
    if(em == null) {
      em = emf.createEntityManager();
    }
    return em;
  }
}
```

Listing 10: CDI-Producer EntityManagerProducer

Der EntityManager wird durch das JPA-Bootstrapping-Verfahren zur Verfügung gestellt. Darüber hinaus wird er für die Verwendung einer In-Memory-H2-Datenbank konfiguriert. Diese Definitionen könnten beispielweise in eine Datei ausgelagert werden, um je nach Testumgebung eine entsprechende Konfiguration zu verwenden. Es wäre auch möglich, unsere Anwendung mit verschiedenen Persistence-Providern zu testen.

Bisher haben wir ausschließlich standardisierte Mittel verwendet, um EJBs zu testen. Die EJBs mussten weder gemockt noch deployt werden. Das heißt, Dependencies werden tatsächlich aufgelöst und Code wird so aufgerufen, wie er auch im Container ablaufen würde. Insgesamt stellt sich die Frage, ob die Tests beim Zusammenspiel all dieser Komponenten nicht zu langsam werden? Immerhin wird ein CDI-Container (Weld) gestartet, die Dependencies werden aufgelöst, Interceptoren aufgerufen usw.

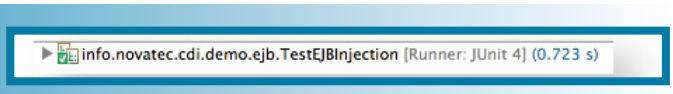

Abb. 3: Ausführungsgeschwindigkeit des Bean-Tests

Die Ausführungszeit der JUnit-Tests zeigt Abbildung 3. Der Test wird auf einem normalen Entwicklerrechner in unter einer Sekunde ausgeführt. Eine solche Feedback-Geschwindigkeit kann man durchaus als „Unit-Testing“ betrachten.

Übrigens: Der Interceptor, die Extension sowie die Konfigurationsdateien (persitence. xml, beans.xml usw.) befinden sich im „Test-Verzeichnis“ (src/test/java ggf. src/test/resources). Der Quellcode und die Konfiguration der Anwendung werden für den Test mittels CDI nie berührt.

Fazit

In diesem Beitrag wurde gezeigt, dass CDI eine interessante sowie mächtige Vorgehensweise für das Testen einer Java EE-Anwendung schafft. Allerdings hören damit die Möglichkeiten noch lange nicht auf. Mittels CDI sind beispielsweise Security-Tests sehr einfach realisierbar: Es können Tests mit unterschiedlichen Benutzern und Rollen durchgeführt werden, um so die korrekte Definition der Sicherheitsregeln zu überprüfen.

Ein wichtiger Aspekt ist auch, dass CDI das Java EE-Paradigma *Convention over Configuration* einhält: CDI wird aktiviert, indem eine leere beans.xml-Datei im Classpath zur Verfügung gestellt wird. Darüber hinaus bietet CDI ein typsicheres Verfahren für DI an. Das führt dazu, dass Fehler aufgrund nicht existierender bzw. mehrdeutiger Abhängigkeiten bereits bei der Initialisierung der CDI-Implementierung, und nicht erst zur Laufzeit, aufgedeckt werden.

In der Version 1.1 von CDI wurden wichtige Features ergänzt und eine verbesserte Integration mit anderen Spezifikationen erreicht. Weld selbst ist eine sehr ausgereifte Implementierung, welche bereits seit langer Zeit bei JBoss Seam eingesetzt wird. Einem produktiven Einsatz steht daher aus Sicht der Autoren nichts im Weg.

Links

[Arquillian] http://www.jboss.org/arquillian
[Bien10] A. Bien, Trouble with crippled Java EE 6 APIs in Maven repository and the solution, 20.10.2010, http://www.adam-bien.com/roller/abien/entry/trouble_with_crippled_java_ee
[CanDI] http://www.caucho.com/resin-application-server/candi-java-dependency-injection/
[DeltaSpike] Apache DeltaSpike, http://deltaspike.apache.org
[JSR299] Java Specification Request 299: Contexts and Dependency Injection for the Java™ EE platform, http://jcp.org/en/jsr/detail?id=299
[JSR346] Java Specification Request 346: Contexts and Dependency Injection for Java™ EE 1.1, http://jcp.org/en/jsr/detail?id=346
[Mockito] http://code.google.com/p/mockito/
[OWB] http://openwebbeans.apache.org/
[Weld] JSR-346 Reference Implementation, http://docs.jboss.org/weld/reference/2.0.1.Final/en-US/html_single/
[WeldHome] http://seamframework.org/Weld

Carlos Barragan beschäftigt sich seit über zehn Jahren mit Anwendungsentwicklung. Er ist Senior Consultant bei der NovaTec GmbH und leitet die Competence-Group „Enterprise Application Architecture Java EE“.
E-Mail: carlos.barragan@novatec-gmbh.de

Professor Dr. Gerhard Wanner lehrt und forscht an der Hochschule für Technik Stuttgart im Studienbereich Informatik mit den Schwerpunkten Software-Engineering und Middleware-Technologie.
E-Mail: wanner@hft-stuttgart.de

Dieter Baier beschäftigt sich seit über zwanzig Jahren mit Anwendungsentwicklung im Enterprise-Umfeld. Als Managing Consultant der NovaTec GmbH liegt sein Arbeitsschwerpunkt in der Integration von Enterprise-Anwendungen.
E-Mail: dieter.baier@novatec-gmbh.de