Das Testframework Arquillian – eine Einführung

Leichtgewichtig testen

Mit Java EE 6 lässt sich inzwischen leichtgewichtig entwickeln, doch ist das Aufsetzen von Integrationstests immer wieder unerwartet kompliziert. Mit Arquillian steht nun ein modernes Testwerkzeug für die einfache Entwicklung von Integrationstests für Java-EE-Anwendungen zur Verfügung.

von Alphonse Bendt und Michael Schütz

Mit der Java-EE-6-Spezifikation steht ein durchgängig standardisierter Technologie-Stack für die Entwicklung zuverlässiger und skalierbarer Java-Enterprise-Anwendungen zur Verfügung. Nach dem Quantensprung zur Java-EE-5-Spezifikation hat man für die Version 6 weiter aufgeräumt und leichtgewichtige Konzepte in allen Bereichen konsequent umgesetzt, die eine effiziente Entwicklung ermöglichen. Leider existiert auf Ebene der Testframeworks bisher noch keine adäquate Unterstützung für den Java-EE-6-Entwickler. Mithilfe von *OpenEJB [1]*, einem eingebetteten EJB-Container von Apache können recht einfach Unit Tests für eine Java-EE-Anwendung realisiert werden. Echte Aussagen über die Lauffähigkeit des Gesamtsystems erfordern jedoch Integrationstests der gesamten Anwendung bzw. von Teilen davon. Bisherige Ansätze für Integrationstests wie *Cargo* [2] oder *Cactus* [3] lösen nur Teilprobleme, wie etwa die Ausführung von Tests innerhalb des Containers (Cactus) oder die Ansteuerung eines Applikationsservers (Cargo). Diese Projekte bieten einen Werkzeugkasten für die Realisierung von Integrationstests, allerdings muss ein hoher Aufwand in die Integration der verschiedenen Werkzeuge investiert werden. Erschwerend kommt hinzu, dass die Abhängigkeit, die

Integrationstests oft zum Build-System aufweisen, dazu führt, dass diese nicht direkt in der Entwicklungsumgebung ausführbar sind. Damit ist es meist ein langer Weg, bis ein Integrationstest tatsächlich vollautomatisch auf dem projekteigenen Continuous-Integration-Server läuft.

Noch ein weiteres Testframework?

Arquillian, ein relativ junges Projekt aus der JBoss-Familie, ist daher angetreten, die Realisierung von Integrationstests massiv zu vereinfachen. In der Projektvision heißt es dazu, dass die Erstellung von Integrationstests in Zukunft zum Kinderspiel werden soll („Arquillian makes integration testing a breeze"). Aus eigener Projekterfahrung und dem Einsatz der genannten Werkzeuge Cargo und Cactus wissen wir, wie schwierig die Entwicklung von Integrationstests für Java-EE-Anwendungen ohne Arquillian sein kann. Vom angepeilten Kinderspiel ist man dabei meist weit entfernt. Aus unserer Sicht ist die Entwicklung und Verwendung von Integrationstests aufgrund folgender Punkte kompliziert:

- Fehlende Unterstützung bei Testentwicklung durch komfortable Bibliotheken und Laufzeitumgebungen
- Ansteuern des Applikationservers aus dem Test heraus
- Hohe Durchlaufzeiten durch wiederholtes Zusammenbauen der Anwendung

Fehlende Unterstützung bei Testentwicklung

Für den Autor eines Tests stellt sich die Frage, wie er direkt aus dem Test heraus auf die zu testenden Ressourcen zugreifen soll. Der Entwickler einer EJB kann durch Verwendung der entsprechenden Annotation alle benötigten Ressourcen komfortabel in die eigene EJB-Implementierung injizieren lassen. Bei der Implementierung von Testfällen stehen diese Möglichkeiten leider nicht zur Verfügung. Stattdessen muss der Testautor auf programmatischem Weg umständlich über einen JNDI-Lookup auf die benötigten Ressourcen zugreifen und wird somit gezwungen, wieder mit expliziten Lookups zu hantieren, die man eigentlich hinter sich gelassen glaubte. Ein modernes Testwerkzeug sollte dem Testautor eine Programmierschnittstelle für die Entwicklung von Java EE-Integrationstests bieten. Idealerweise sollte diese eng an die EJB-API angelehnt sein, damit keine neuen Schnittstellen erlernt werden müssen.

Ansteuern des Applikationservers aus dem Test heraus

Ein Framework für Integrationstests muss in der Lage sein, den Lebenszyklus des eingesetzten Applikationsservers zu verwalten. Abhängig vom Testszenario muss der Server gestartet, Testcode und zu testender Code installiert und schließlich der Container wieder heruntergefahren werden. Damit die Tests einfach zwischen verschiedenen Servern portierbar sind, ist eine produktunabhängige Realisierung dieses Mechanismus erforderlich.

Hohe Durchlaufzeiten

Für eine hohe Produktivität ist es notwendig, die Zeit zwischen einer Änderung an Produktiv- bzw. Testcode und dem Fehlschlag bzw. Erfolg eines Tests zu minimieren. Idealerweise sollte ein Entwickler durch die Tests auf jede Änderung ein schnelles Feedback erhalten. Langsam laufende Tests werden von den Entwicklern nicht regelmäßig ausgeführt, wodurch es zu Qualitätsproblemen kommen kann. Neben der reinen Ausführungszeit der Tests spielt hier der Zeitbedarf für eine Neuübersetzung und das Paketieren der Applikation die zentrale Rolle. Moderne Entwicklungsumgebungen unterstützen heutzutage die inkrementelle Übersetzung des Projekts. Dabei werden nach Änderungen am Sourcecode nur die davon abhängenden Codeteile neu übersetzt. Dies ist so schnell, dass das neu kompilierte Projekt üblicherweise direkt nach Abspeichern einer Änderung zur Verfügung steht. Unit-Tests können dadurch sehr schnell ausgeführt werden.

Betrachten wir nun eine nichttriviale Geschäftsanwendung. Diese besteht typischerweise aus mehreren JAR-, WAR- und EAR-Archiven. Für die korrekte Konstruktion dieser Archive wird ein Werkzeug wie *Apache Ant* oder *Apache Maven* verwendet. Diese profitieren nicht von der inkrementellen Übersetzung der IDE. Abhängig von der Größe des Projekts und der Komplexität der einzelnen Build-Schritte kann ein Rebuild schnell zeitintensiv werden. Mit den bisherigen Mitteln wurde für die Testausführung für jeden Test die gesamte Anwendung mit allen zur Anwendung gehörenden Ressourcen zusammengebaut. Dies ist entsprechend zeitaufwändig. Bei Integrationstests von Systemteilen wird jedoch nicht immer alles benötigt. Typischerweise benötigen Tests jeweils nur einen Ausschnitt der Anwendung. Damit ergeben sich die Anforderungen an ein modernes Testwerkzeug:

- Integrationstests sollten innerhalb der IDE ausführbar sein und
- Änderungen an Test und Produktivcode sollten keinen Rebuild durch das Konstruktionswerkzeug (Ant/Maven) erfordern

In unserem Blogartikel [4] haben wir bspw. beschrieben, wie man mit Maven, JSFUnit und Cargo Integrationstests auf Basis von JSFUnit schreibt. Die dort beschriebene Lösung funktioniert, bei einer Änderung müssen jedoch bis zu fünf (*war*, *ear*, *it-war*, *it-ear*, *it-cargo*) Maven-Projekte neu gebaut werden. Das ist entsprechend zeitintensiv.

Features – Was bietet mir Arquillian?

Betrachten wir nun, welche Lösungen Arquillian (http://jboss.org/arquillian) für die Erstellung von Tests bietet. JBoss hat sich mit diesem Framework das Ziel gesetzt, die genannten Schwächen bisheriger Testframeworks auszumerzen. Hierfür bietet Arquillian nicht nur ein entsprechendes API und eine Umgebung an, sondern auch Integrationen mit den gängigen Testframeworks TestNG und JUnit. Arquillian unterstützt damit den gesamten Lebenszyklus eines Integrationstests:

- Starten eines Applikationsservers
- Deployment des Tests und der zu testenden Klassen in den Container
- Ausführen des Tests direkt im Container
- Herunterfahren des Applikationsservers

Dieser Funktionsumfang lässt sich sehr einfach zeigen (Listing 1).

Der Test ist im Vergleich zur entsprechend benötigten Cactus/Cargo-Konfiguration überraschend kurz. Im entsprechenden Maven-POM sind die benötigten Abhängigkeiten auf das Arquillian-API definiert. Da der Test auf JUnit basiert, kann der Test einfach mittels Maven oder direkt aus der IDE heraus ausgeführt werden. Alternativ wird auch die Verwendung von TestNG unterstützt. Betrachten wir die Elemente des Tests im Detail:

1. *@RunWith*: Diese Annotation gehört zum API von JUnit. Arquillian implementiert einen eigenen so genannten *Runner* zum Ausführen der Tests. Wir werden darauf im weiteren Verlauf dieses Artikels detailliert eingehen.

Listing 1

```
package de.akquinet.arquillian.jm;

import javax.ejb.EJB;

import org.jboss.arquillian.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.Test;
import org.junit.runner.RunWith;

import static org.junit.Assert.assertEquals;

// 1. JUnit-konforme Integration von Arquillian
@RunWith(Arquillian.class)
public class HelloEJBTest {

   // 2. EJB injection
   @EJB
   private HelloEJB helloEJB;

   // 3. Archiv deployen
   @Deployment
   public static JavaArchive createTestArchive() {
      // 4. Archive erstellen
      return ShrinkWrap.create(JavaArchive.class, "helloEJB.jar")
            .addClasses(HelloEJB.class, HelloEJBBean.class);
   }

   @Test
   public void testHelloEJB() {
      String result = helloEJB.sayHelloEJB("Michael");
      assertEquals("Hello Michael", result);
   }
}
```

2. *@EJB Injection*: Dies ist die bekannte Annotation aus dem EJB-API. Arquillian injiziert hier die entsprechend referenzierten Container-Ressourcen. Neben *@EJB* unterstützt Arquillian standardmäßig auch *@Ressource* und *@Inject* (CDI).
3. *@Deployment*: Diese Annotation gehört zum API von Arquillian. Der Testautor markiert damit die Methode innerhalb des Tests, die das für die Testausführung zu installierende Archiv erzeugt. Innerhalb dieser Methode wird *ShrinkWrap* zum Erzeugen des Deployments verwendet.
4. *ShrinkWrap*: Dies ist ein Werkzeug, das im Rahmen des Arquillian-Projekts entstanden ist. Es dient dazu, programmatisch Deployment-Archive (wie *.jar*, *.war* oder *.ear*) zu erzeugen. Innerhalb des Tests werden somit die zu testenden Klassen definiert. Aus diesen wird von ShrinkWrap zusammen mit dem Testcode ein Deployment-Archiv erzeugt. Dieses Archiv enthält nur die benötigten Klassen und kann somit schnell erzeugt werden. Innerhalb der Entwicklungsumgebung werden die benötigten Klassen direkt aus dem Klassenpfad der Entwicklungsumgebung zusammengesammelt. Damit ist die Testausführung unabhängig vom verwendeten Build-Werkzeug.

Wie benutze ich das?

Das oben skizzierte Beispiel steht unter http://github.com/akquinet/arquillian-javamagazin zum Download und direkten Ausprobieren bereit. Folgende Voraussetzungen müssen erfüllt sein:

- Java-Version >= 1.5
- Maven-Version >= 2.2

Diese Schritte sind zum Ausführen der Demo notwendig:

1. Herunterladen [5] bzw. direkt mit Git
2. (ggfs. auspacken)
3. cd <arquillian-javamagazin>
4. mvn test

Nachdem alle notwendigen Maven-Artefakte heruntergeladen wurden, wird ein JUnit-basierter Arquillian-Test ausgeführt. Dieser verwendet OpenEJB, einen eingebetteten EJB-Container von Apache. In der Ausgabe kann man sehen, wie der Container gestartet und das Testarchiv installiert wird. Nach der Ausführung findet sich die Ausgabe des Tests direkt auf der Konsole wieder. In unserem Beispiel haben wir den verwendeten Container mittels eines Maven-Profils konfiguriert. Wir haben dafür drei Profile vorbereitet. Neben dem standardmäßig aktiven Profil *openejb-embedded-3* stehen noch *jbossas-embedded-6* und *glassfish-embedded-3* zur Verfügung. Sie verwenden den *JBoss-* respektive *Glassfish Embedded-Server*. Hierdurch kann der Integrationstest profilbasiert ohne programmatische Anpassungen auf verschiedenen Applikationsservern ausgeführt werden. Die Auswahl des Profils erfolgt mit der Maven-Option *-P*. Beispiel:

1/1 RailsCon

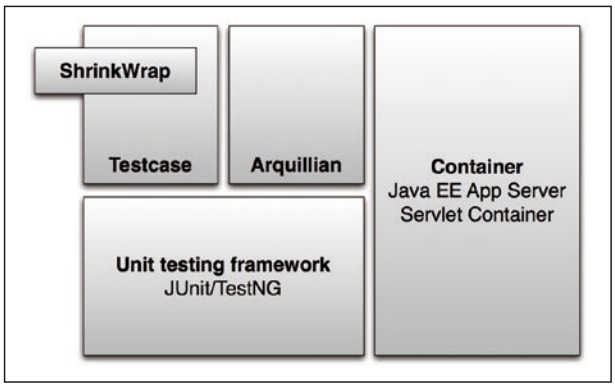


Abb. 1: Grobe Architektur von Arquillian

```
mvn test -Pjbossas-embedded-6
```

Weitere Maven-Profile für zusätzliche Container-Konfigurationen können leicht hinzugefügt werden [6].

Arquillian im Detail

Werfen wir nun einen Blick unter die Haube von Arquillian. Arquillian ist modular aufgebaut und besteht neben dem Kern aus folgenden Komponenten (vgl. **Abbildung 1**):

- ShrinkWrap
- Integration in die Testframeworks JUnit und TestNG
- Anbindung der Container

Listing 2

```
// hier wird ein WAR definiert
final WebArchive war = ShrinkWrap.create(WebArchive.class, "test.war")
        .setWebXML("war/web.xml")
        .addWebResource("war/components.xml");

// hier wird ein EJB-JAR definiert
final JavaArchive ejb = ShrinkWrap.create(JavaArchive.class, "test.jar")
        // specific test dependent classes
        .addClasses(clazzes)
        .addClasses(
                AbstractSeamItTest.class, SeamUtil.class, AbstractEntity.class)

        .addResource("ejb/seam.properties", "seam.properties")
        .addResource("ejb/components.properties", "components.properties")
        .addManifestResource("ejb/ejb-jar.xml", "ejb-jar.xml")
        .addManifestResource("ejb/persistence.xml", "persistence.xml");

// das EAR enhält das EJB-JAR, das WAR und zusätzliche Maven-Artefakte
final EnterpriseArchive ear = ShrinkWrap.create(EnterpriseArchive.class, "test.ear")
        // Module liegen in der Wurzel des EAR und werden in der application.
        // xml Datei des EAR aufgeführt
        .addModule(ejb)
        .addModule(war)
        .addModule("jboss-seam-2.2.1.CR2.jar")
        // wird dem lib-Verzeichnis des EAR hinzugefügt
        .addLibrary("jboss-el-1.0._02.CR5.jar");
```

ShrinkWrap

Arquillian verwendet ShrinkWrap, um „On the Fly" das Archiv für das Deployment in den Zielcontainer zu erzeugen. Dadurch ist die Erzeugung des Archivs vom eingesetzten Build-Werkzeug unabhängig. Stattdessen wird das Archiv programmatisch im so genannten *Fluent Style* definiert. Damit hat man die Möglichkeit, maßgeschneiderte „Mikro-Deployments" für den Test zu erzeugen. Da ShrinkWrap direkt auf die benötigten Klassen und Ressourcen im Classpath zugreift, profitiert es an dieser Stelle von der inkrementellen Übersetzung der Entwicklungsumgebung. ShrinkWrap ist als eigenständiges Projekt unter der Apache-Lizenz verfügbar (http://www.jboss.org/shrinkwrap). Neben der im Codebeispiel demonstrierten Methode *addClass* existieren zahlreiche Methoden, um dem Archiv Klassen und Ressourcen hinzuzufügen:

```
ShrinkWrap.create(JavaArchive.class, "test.jar")
  .addPackage("de/akquinet")
  .addManifestResource("test-persistence.xml", "persistence.xml");
```

Im obigen Beispiel wird ein Java-Archiv definiert, das alle Klassen aus dem Package *de.akquinet* und die Ressource *META-INF/persistence.xml* mit dem Inhalt der Datei *„test-persistence.xml"* enthält.

Es ist auch möglich, ein existierendes JAR-Archiv hinzuzufügen. Hier ist der Wunsch nahe liegend, direkt Maven-Artefakte angeben zu können. Dafür wird im nächsten Release von ShrinkWrap Unterstützung existieren. Diese Implementierung basiert auf der *Maven-Aether-Bibliothek* [7] und lädt alle benötigten Abhängigkeiten Maven-konform aus entsprechenden Repositories und fügt sie dem Archiv hinzu.

Neben einfachen Archiven können auch komplexere Archive wie EAR, WAR, SAR und RAR erzeugt werden. Im folgenden Beispiel wird ein Auszug aus dem Arquillian-Test einer Seam-2-Anwendung gezeigt. Es wird ein EAR-Archiv erzeugt, das WAR und EJB-JAR beinhaltet:

Das innerhalb eines Tests erzeugte Archiv wird von Arquillian vor dem Deployment in den Zielcontainer um weitere Klassen und Ressourcen erweitert. Dazu gehören mindestens die Arquillian-Bibliotheksklassen sowie weitere containerspezifische Klassen. Dieser Mechanismus ist über einen Erweiterungspunkt ausbaufähig. Durch Implementierung der Schnittstelle *AuxiliaryArchiveAppender* kann das erzeugte Archiv gemäß den Anforderungen des Zielcontainers bzw. der eingesetzten Frameworks angepasst werden.

Integration der Testframeworks

Arquillian unterstützt die Realisierung von Tests auf Basis von *JUnit* oder *TestNG*. Für JUnit-basierte Tests wird der Standard-Testrunner mit einem Arquillian-spezifischen Testrunner ersetzt. Dies wird durch die Verwendung der JUnit-spezifischen Annotation *@RunWith* erreicht. Arquillian erhält damit vollständige Kontrolle über die Testdurchführung. Für TestNG-basierte

Tests existiert kein entsprechendes Konzept. Stattdessen werden hier die Tests von einer Basisklasse abgeleitet. Elegant an beiden Ansätzen ist die Tatsache, dass mit Arquillian Tests direkt aus der Entwicklungsumgebung heraus aufgerufen werden können.

Wie im Beispielcode ersichtlich, kann der Testcode mit *@EJB*, *@Resource* oder *@Inject* annotierte Attribute enthalten. Die korrekte Initialisierung dieser Attribute liegt in der Verantwortung von Arquillian. Statt die entsprechende Logik im Arquillian-Kern zu verankern, wurde hier ein erweiterbarer Ansatz gewählt. Die Initialisierung von annotierten Attributen wird an einen sog. *Enricher* delegiert. Ein Enricher ist dafür verantwortlich, die Felder mit einer bestimmten Annotation innerhalb eines Tests zu injizieren. Damit kann Arquillian leicht um Enricher für weitere Typen erweitert werden. So kann man z.B. für das Testen einer Seam-2-Anwendung einen Enricher für die Annotation *@In* realisieren. Arquillian liefert Enricher für die genannten Annotations-Typen. *@EJB* und *@Ressource* haben einen JNDI Lookup zur Folge, *@Inject* nutzt die neuen JSR-299-(CDI-)Features aus Java EE 6.

Integration der Container

Die Ansteuerung der von Arquillian unterstützten Container erfolgt jeweils mittels containerspezifischer Plug-ins. Die Kommunikation zwischen Arquillian und dem jeweiligen Plug-in erfolgt dabei über ein *Service Provider Interface (SPI)*, also über eine definierte Schnittstelle zwischen Arquillian und dem Container. Durch Verwendung dieses SPI ist Arquillian damit auch einfach um zukünftige Zielcontainer erweiterbar. Container-Plug-ins sind jeweils als eigenständige JAR-Archive gepackt. Arquillian bietet aktuell 17 Implementierungen an. Dazu gehören vollständige Applikationsserver wie JBoss oder GlassFish, Servlet-Container wie Tomcat oder Jetty, Embedded-Container wie OpenEJB und weitere Container wie Weld (CDI) oder IronJacamar (vgl. [6]).

Diese sind als Maven-Artefakte im JBoss-Repository verfügbar. Durch Definition des entsprechenden Maven-Artefakts im Maven-Projekt wird der entsprechende Container durch Scannen des Classpath von Arquillian erkannt und steht im Test als Zielcontainer zur Verfügung. In folgender Maven-Konfiguration wird bspw. ein JBoss AS 5.1.0.GA eingebunden (Listing 3).

Container-Modi

Arquillian kann den Lebenszyklus des Zielcontainers auf drei unterschiedliche Arten ansteuern. Dies sind im Einzelnen:

- remote
- managed
- embedded

Im *remote*-Szenario wird der Container in einer externen JVM gestartet. Der Arquillian-Test verwendet ein containerspezifisches Protokoll, um mit der entfernten JVM zu kommunizieren. Die Kontrolle über den Lebenszyklus dieses Containers liegt hier außerhalb der Verantwortung von Arquillian. Arquillian installiert das Archiv mittels JMX JSR-88-konform in einen bereits gestarteten Container. Dem Standard entsprechend werden die Konfigurationsdateien für einen Remote-Container wie dem Provider-URL in der Datei *jndi.properties* gepflegt.

Der *managed*-Fall entspricht weitgehend dem *remote*-Fall. Der wichtige Unterschied ist, dass hier der Lebenszyklus des Containers von Arquillian verwaltet wird. Das heißt konkret, dass Arquillian sich um das Starten und Stoppen des Containers kümmert. Die für einen Managed-Container erforderlichen Konfigurationen wie Installationsverzeichnis oder Http-Port werden in der Datei *arquillian.xml* gepflegt. Da jeder Container verschiedene Konfigurationsmöglichkeiten aufweist, gibt es entsprechend für jeden Container einen eigenen XML-Namespace.

Listing 3

```
<project>
  [...]

  <properties>
    <arquillian.version>1.0.0.Alpha4</arquillian.version>
    <jboss.version>5.1.0.GA</jboss.version>
  </properties>

 <dependencies>
   <dependency>
     <groupId>org.jboss.arquillian.container</groupId>
     <artifactId>arquillian-jbossas-remote-5.1</artifactId>
     <version>${arquillian.version}</version>
   </dependency>

   <dependency>
     <groupId>org.jboss.jbossas</groupId>
     <artifactId>jboss-as-client</artifactId>
     <version>${jboss.version}</version>
     <type>pom</type>
    </dependency>
  </dependencies>
</project>
```

Listing 4

```
<?xml version="1.0"?>

<arquillian xmlns="http://jboss.com/arquillian"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jboss="urn:arq:org.jboss.arquillian.container.jbossas.managed_5_1">

   <!-- Konfiguration für JBossAS-Managed-Container -->
   <jboss:container>
      <jbossHome>/home/hudson/jboss-5.1</jbossHome>
      <profileName>default</profileName>
      <bindAddress>127.0.0.1</bindAddress>
      <httpPort>8080</httpPort>
   </jboss:container>

</arquillian>
```

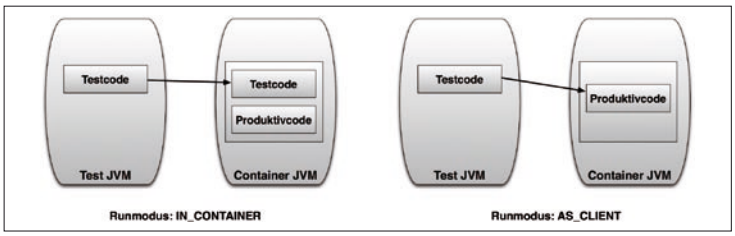


Abb. 2: Arquillian-Run-Modi

Listing 4 zeigt ein Beispiel aus der *arquillian.xml*-Datei für die Konfiguration eines JBossAS 5.1 Managed-Containers.

Es werden die Parameter zum Zugriff auf einen JBoss AS definiert. Mit der angegebenen Datei verwendet Arquillian den im Verzeichnis */home/hudson/jboss-5.1* installierten JBoss im Default-Profil. Die weiteren Parameter werden beim Start des Servers verwendet. Ein Embedded-Container wird in derselben JVM wie der Arquillian-Test gestartet. Im Gegensatz zu *managed* und *remote* ist hier keine gesonderte Konfiguration notwendig, da der Embedded-Container vollständig von Arquillian kontrolliert werden kann.

Die verschiedenen Container-Modi können verwendet werden, um die Integrationstests optimal an das Testszenario anzupassen. Nehmen wir etwa an, dass ein Projektteam eine Java-EE-5-/EJB3-Anwendung entwickelt, die den JBoss AS 5.1 als Zielsystem hat. Die Entwickler haben jeweils einen lokalen JBoss-Server installiert. In diesem Umfeld könnte Arquillian wie folgt zum Einsatz kommen:

- Lokale Entwicklung: OpenEJB-Embedded-Container für schnelle EJB-Tests und Tests gegen einen remote JBoss AS 5.1. Dieser wird vom Entwickler jeweils manuell gestartet.
- Continuous Integration mittels Hudson: Tests gegen eine managed JBoss AS 5.1. Dieses Setup ermöglicht schnelle Turnaround-Zeiten während der Entwicklung und regelmäßige Tests des kompletten Systems mittels Continuous Integration.

Listing 5

```
@RunWith(Arquillian.class)
@Run(AS_CLIENT)
public class LocalRunServletTestCase
{
  @Deployment
  public static WebArchive createDeployment() {
    return ShrinkWrap.create(WebArchive.class, "test.war") .addClass(TestServlet.class);
  }

  @Test
  public void shouldBeAbleToCallServlet() throws Exception {
    String body = readAllAndClose(new URL("http://localhost:8080/test/Test").
                                                                openStream());
    Assert.assertEquals("Verify that the servlet was deployed and returns the expected
                                                        result","hello",body); }
  }
```

Listing 6

```
  @Selenium
  private WebDriver driver;

  @Test
  public void testLoginAndLogout()
  {
    driver.get("http://localhost:8080/demo/home.jsf");

    driver.findElement(By.id("loginForm:username")).sendKeys(demo);
    driver.findElement(By.id("loginForm:password")).sendKeys(password);
    driver.findElement(By.id("loginForm:login")).click();
  }
```

Run-Modi

Die im Codebeispiel gezeigten Tests werden mit dem Produktivcode zusammen in den Zielcontainer installiert und dort ausgeführt. Entsprechend kann der Testcode auf alle Ressourcen des Containers zugreifen. Neben dieser Variante können auch Tests gegen die externe Schnittstelle der Applikation realisiert werden. Zur Konfiguration dieses Verhaltens bietet Arquillian die Annotation *@Run* an. Gültige Werte sind *AS_CLIENT* und *IN_CONTAINER*, wobei *IN_CONTAINER* das beschriebene Standardverhalten darstellt. Der unterschiedliche Ablauf wird in folgendem Diagramm ersichtlich:

Zukunft

Arquillian liegt derzeit in Version 1.0.0.Alpha4 vor. Version 1.0.0.Beta1 steht kurz vor der Tür. Ende des Jahres ist mit einem ersten finalen Release zu rechnen. Neben einem Ausbau des API liegt der aktuelle Fokus der weiteren Entwicklung auf diesen Punkten:

- Integration weiterer Container
- Deployment mehrerer Archive in einer Testdurchführung
- Deployment eines Tests gegen mehrere Container
- Deployment in die Cloud
- Anbindung weiterer Frameworks
- Tool-Support

Integration weiterer Container

Arquillian hat den Anspruch, als umfassendes Testwerkeug eine große Bandbreite an Laufzeitumgebungen zu unterstützen. Der Begriff Container ist hier etwas weitläufiger als der des klassischen Applikationsservers zu betrachten. So wird neben einer Unterstützung der Applikationsserver Oracle Weblogic, IBM WebSphere oder JBossAS 4.2 auch an einer containerartigen Infrastruktur für Spring, Hibernate und Drools gearbeitet.

Deployment mehrerer Archive in einer Testdurchführung

Das ist ein sehr nützliches Feature, das es ermöglicht, leicht komplexere Tests aufzubauen. Damit kann man

zwei voneinander abhängige EAR-Archive in einem Testdurchlauf deployen, um ein möglichst realitätsnahes Ergebnis zu erlangen. Alternativ könnte man auch auf diesem Weg elegant neben dem Archiv eine von diesem benötigte DataSource direkt in den Container deployen.

Deployment eines Tests gegen mehrere Container

Hier wird die Idee verfolgt, einen Test in einem Durchlauf beispielweise gegen alle Java-EE-6-konformen Container laufen zu lassen.

Deployment in die Cloud

Arquillian wird es ermöglichen, Testarchive direkt in eine Cloud-Umgebung zu deployen. Dafür werden die bestehenden Bibliotheken *jclouds* und *DeltaCloud* eingebunden. Ein Deployment in eine Cloud ermöglicht neben einem Integrationstest unter realistischen Bedingungen auch die einfache Realisierung von Lasttests.

Anbindung weiterer Frameworks

Arquillian bindet bereits JSFUnit [8] an. Weitere Frameworks wie Selenium [9] oder DBUnit [10] sind in Planung. Gerade der Support für Selenium 2.0 ist bereits fertiggestellt und wird mit dem nächsten Release veröffentlicht. Selenium ist ein Werkzeug, mit dem das Erstellen von Testfällen für Webanwendungen extrem erleichtert wird. Die einfache Integration verdeutlicht Listing 6.

Die Selenium-Erweiterung nutzt den weiter oben beschriebenen Enricher-Mechanismus, um den WebDriver innerhalb des Selenium-Tests verfügbar zu machen. WebDriver gehört zum Selenium-API und erlaubt es, einen entfernten Webbrowser fernzusteuern und damit auf die eigene Webanwendung zuzugreifen. Entsprechend startet Arquillian den Selenium-Test im Run-Modus *@Run(AS_CLIENT)*. Die Anwendung wird über ihre externe Schnittstelle getestet.

Tool-Support

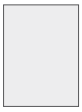
Es sind mehrere Tools in Planung. Die Entwickler des Projekts *JBossTools* (für Eclipse) arbeiten aktuell an einer aktiven Unterstützung.

Fazit

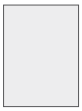
In unserem Artikel haben wir gezeigt, wie man mittels Arquillian EJB3-Komponenten direkt innerhalb eines Applikationsservers testen kann. Arquillian erzeugt dafür das Testarchiv, installiert es in den Applikationsserver und führt die Tests aus. Arquillian kann optional den Lebenszyklus des Applikationsservers verwalten und bietet neben einem effizienten Rebuild-Ansatz eine klare Schnittstelle für den Entwickler von Tests. Damit steht mit Arquillian erstmals ein integriertes und leistungsfähiges Framework zum Testen des Java EE Stacks zur Verfügung. Die Hauptvorteile gegenüber bisherigen Ansätzen sind das einfache Programmiermodell zum Erstellen von JUnit/TestNG-basierten Tests und die hohe Performance.

Das Testframework ist aktuell noch Alpha-Software, das erste Beta-Release steht allerdings kurz vor der Tür. Trotz des Alpha-Status hat Arquillian bereits einen hohen Reifegrad erreicht. Dies ist v.a. darauf zurückzuführen, dass Arquillian in immer mehr JBoss-Projekten als Basis-Test-Framework eingesetzt wird. Beispiele hierfür sind JBoss Seam [11], JSFUnit, JBoss Embedded [12] und die JBoss-EJB3-Implementierung [13].

Wir setzen Arquillian im Unternehmen bereits produktiv in Seam-2-Projekten ein. Die Testqualität wird gesteigert, weil Entwickler einfacher ihre Java-EE-Anwendungen testen können. Dadurch, dass die Testausführung nicht mehr so lange dauert, wird auch die Bereitschaft zum Testen und somit das Testqualitätsbewusstsein verbessert.



Michael Schütz arbeitet als Software Engineer bei der akquinet AG in Berlin und beschäftigt sich seit Jahren mit Java-Enterprise-Architekturen und -Technologien. Er ist Gründer und Moderator der Seam-Plattform auf Xing und wirkt regelmäßig an JBoss-Open-Source-Projekten mit. Michaels berufliche Interessen liegen derzeit in den Bereichen Seam/CDI, Java EE Testing, Java EE Cloud Hosting und Git. Kontakt: Michael.Schuetz@akquinet.de.



Alphonse Bendt ist für die akquinet AG in Berlin als Berater und Softwarearchitekt tätig. Hauptsächlich beschäftige er sich dort mit der Entwicklung von JEE-Anwendungen und dem Build Management mit Maven/Ant.

Links & Literatur

[1] OpenEJB: http://openejb.apache.org/

[2] Cargo: http://cargo.codehaus.org/

[3] Cactus: http://jakarta.apache.org/cactus/

[4] How to test an EAR based JSF Application using JSFUnit: http://blog.akquinet.de/2010/06/01/how-to-test-an-ear-based-jsf-application-using-jsfunit/

[5] Download-Link für das Beispielprojekt: http://github.com/akquinet/arquillian-javamagazin/archives/master

[6] Arquillian-Container-Konfigurationen: http://docs.jboss.org/arquillian/reference/latest/en-US/html_single/#d0e708

[7] Maven-Aether-Bibliothek: http://www.sonatype.com/people/2010/08/introducing-aether/

[8] SFUnit: http://www.jboss.org/jsfunit

[9] Selenium: http://seleniumhq.org/

[10] DBUnit: http://www.dbunit.org/

[11] http://seamframework.org

[12] http://docs.jboss.org/ejb3/embedded/embedded.html

[13] http://jboss.org/ejb3