JS-3-12 – iwaszkiewicz-schuetz – k1

Das volle Programm

Entwicklung mit Java EE 6 anhand eines ganzheitlichen Beispiels

Marek Iwaszkiewicz, Michael Schütz

Die Java EE 6-Spezifikation bietet mit den zahlreichen kombinierbaren Einzelspezifikationen mehr als nur eine Grundlage für die effiziente Entwicklung moderner Softwaresysteme. Gerade durch die Kombinierbarkeit der einzelnen Spezifikationen untereinander sowie mit weiteren im Java EE-Bereich verfügbaren Technologien steht eine robuste und zukunftsträchtige Technologielandschaft bereit. Es wird Zeit, diese ganzheitlich exemplarisch zu beleuchten.

Der erste Schritt bei der Umsetzung eines Java EE-Projekts ist bekanntlich das Aufsetzen der Softwareinfrastruktur und -architektur. Dieser Schritt erfordert umfangreiches und tief gehendes technisches Wissen um die beteiligten Technologien. Bereits hier wird die Grundlage für die Effizienz bei der Umsetzung des Projekts sowie für die Wartbarkeit der fertiggestellten Software gelegt. Ein Problem stellt hierbei nicht nur die Komplexität der Aufgabe als solche dar, sondern auch der in Projekten übliche Zeitdruck, durch welchen oft wichtige Aspekte, wie zum Beispiel eine Umgebung für die Testautomatisierung, unter den Tisch fallen gelassen werden.

Mit dem Java EE 6-Technologie-Stack steht ein eleganter und mächtiger, aus einer Vielzahl von einzelnen Spezifikationen bestehender Standard zur Verfügung. Ganzheitliche Beispiele eines professionellen Setups sind jedoch kaum bis gar nicht vorhanden. Beim ganzheitlichen Ansatz gilt es nicht nur, die einzelnen Technologien des Java EE 6-Stacks in Einklang zu bringen, sondern auch Aspekte wie Build-System, Schichtentrennung sowie Testautomatisierung zu betrachten und optimal umzusetzen.

An genau dieser Stelle setzen wir an. Wir werden im Verlauf dieses Artikels die Architektur und Infrastruktur eines professionellen Java EE 6-Projekts vorstellen. Um den Mehrwert des Artikels für Sie zu erhöhen, orientieren wir uns hierbei an einer vollständigen und sofort lauffähigen Demoanwendung, deren Quellcode wir zum Download [youTask] bereitstellen. Hiermit möchten wir auch dem Manko der kaum verfügbaren umfassenden Beispiele entgegen wirken.

Wir konzentrieren uns im Rahmen dieses Artikels auf einige der am meisten genutzten Technologien aus der Spezifikation – gehen aber weder tiefer auf die einzelnen Technologien ein noch werden wir die komplette Spezifikation mit allen Facetten betrachten. Die vorgestellte Lösung ist als Basis für die individuellen Anforderungen einzelner Projekte zu verstehen.

Das Problem im Detail

Bevor wir uns den technischen Details der Demoanwendung widmen, wollen wir beleuchten, warum das Aufsetzen von Java EE-Projekten und -Architekturen mit professionellem Anspruch überhaupt eine so große Herausforderung darstellt.

Generell sind hier zunächst Aspekte der technischen Infrastruktur sowie anschließend Aspekte der Softwarearchitektur zu klären. Bei der technischen Infrastruktur müssen unter anderem die folgenden Fragen geklärt werden: Welches Frontend-Paradigma (Web, RIA) eignet sich am besten und welche Technologie (JSF, Vaadin, GWT, Flex) ist die beste? Welche Schnittstellen (REST, Webservice, RMI) werden nach außen geboten? Auf welches Build-System (Ant, Maven) soll gesetzt werden?

Im zweiten Schritt wird die Softwarearchitektur aufgesetzt. Hier unterscheidet man grob die fachliche Anwendungssicht und die technische Sicht. In der technischen Sicht werden unter anderem die Schichten und deren Schnittstellen definiert und es wird die Integration der im Infrastruktur-Schritt ausgewählten Technologien vorgenommen.

Im Kontext von Java EE 6-Projekten klären sich einige der genannten Fragen alleine schon durch die Wahl des Java EE 6-Standard-Stacks. Jedoch schränkt man dadurch zunächst nur die Technologiewelt ein. Der Standard bietet aber den großen Vorteil, auf eine Vielzahl moderner und bewährter Technologien zurückgreifen zu können. Dennoch gibt es immer einige über den Standard hinausgehende Aspekte, die frühzeitig in die Architektur einzubauen sind. Hierzu zählen zum Beispiel Aspekte wie Exception-Handling, Security und Internationalisierung.

Unterm Strich sind also verschiedene Paradigmen, Technologien und Frameworks unter einen Hut zu bringen. Der in Projekten übliche Zeitdruck führt oft dazu, dass die Umsetzung einiger wichtiger Aspekte weggelassen oder suboptimal umgesetzt wird – allen voran gilt dies für die Testumgebung. Für ein optimales Setup wird tief gehendes Wissen in allen Bereichen benötigt. Es müssen die Module mit dem Build-System der Wahl aufgesetzt und in diese die einzelnen Technologien eingebunden werden. Dies hört sich zunächst einfacher an, als es ist. Man darf nicht vergessen, dass dieser Schritt aus vielen kleinen Einzelschritten besteht. Hier müssen unter anderem Fragen zu Programmversionen und der Versionen der genutzten Frameworks und Technologien beantwortet sowie die Versionen aufeinander abgestimmt werden.

Besonders anspruchsvoll ist jedoch das Aufsetzen einer Testumgebung. Die Herausforderungen ergeben sich hier schon alleine aus der Heterogenität der Aufgabe. So braucht man zum Beispiel für Persistenztests eine andere Testgrundlage als für Unit-Tests im Business-Bereich, bei welchen Abhängigkeiten durch Mock-Objekte ersetzt werden. Das Gleiche gilt für eine Umgebung für Integrationstests.

Idealerweise nimmt man für diese Aufgaben daher entweder einen entsprechend qualifizierten Architekten oder man

orientiert sich an einer in der Praxis erfolgreich eingesetzten, nach Möglichkeit vollständigen Vorlage. Da man von beiden nur die Vorlage vervielfältigen und der Community zur Verfügung stellen kann, haben wir uns dafür entschieden, das in der Einleitung erwähnte Java EE 6-Projekt vorzustellen und zum Download anzubieten. Das Projekt erhebt aber nicht den Anspruch der Vollständigkeit, enthält aber Lösungen zu den wichtigsten eben genannten Aspekten.

Vorhang auf – Java EE 6

Bisher haben wir uns den Grundsätzen und Herausforderungen professioneller Softwareentwicklung sowie den potenziellen Problemen in diesem Kontext gewidmet. Bevor wir das Demoprojekt im Detail betrachten und aufzeigen, wie sich diese Herausforderungen in Java EE 6-Technologien und -Konzepten im konkreten Fall auswirken, widmen wir uns einer kurzen Einführung in den Java EE 6-Technologie-Stack.

Bereits mit der Spezifikation von Java EE 5 fand im Vergleich zur Vorgängerversion ein großer Umbruch statt. Im Zuge dieses Umbruchs fanden viele Features und Konzepte alternativer, leichtgewichtiger Frameworks Eingang in den Standard. Durch die vollführte Konsolidierung und Vereinfachung wurde der Java EE-Spezifikation quasi wieder neues Leben eingehaucht. Mit der aktuellen Version 6 der Spezifikation wurde dieser Weg weiter verfolgt und es fanden noch weitere Konsolidierungen und Neuerungen von Einzelspezifikationen und Technologien statt. Fehler der Vergangenheit wurden somit inzwischen behoben und es steht uns nun ein moderner, mächtiger und robuster, aber auch leichtgewichtiger Standard zur Verfügung.

Als die herausragende Neuerung von Java EE 6 ist das einheitliche Komponenten-Modell CDI (Contexts and Dependency Injection) zu nennen, mit welchem Dritthersteller spezifische Erweiterungen des Standards entwickeln und diese nahtlos in die Java EE-Technologielandschaft integrieren können.

Die Finalisierung des Standards in Version 6 liegt mittlerweile mehr als zwei Jahre zurück. Das mag erstaunen, da wir hier angetreten sind, ein modernes Technologieumfeld vorzustellen. Zum einen liegt dies daran, dass es über zwei Jahre gedauert hat, bis namhafte Hersteller von Applikationsservern wie JBoss oder IBM eine Java EE 6 zertifizierte Version bereit gestellt haben. Zum anderen liegt es auch daran, dass erst im Laufe dieses Jahres CDI-Module als Erweiterungen der Standard-Funktionalität entwickelt und bereit gestellt wurden. In dem Beispielcode werden wir uns im Speziellen eine Erweiterung von Seam [Seam] anschauen. Erst durch diesen Schritt der Erweiterbarkeit entfaltet der Stack sein volles Leistungsvermögen.

Die beschriebenen Entwicklungen zeigen, dass es nicht nur darum geht, sauber Spezifikationen zu erstellen, sondern vor allem um deren stimmiges Zusammenspiel. So entsteht eine moderne und professionelle Komplettlösung, die wir im Folgenden vorstellen werden.

Die Demoanwendung YouTask

Einstieg in die Anwendung

Für den Beispielcode des Artikels haben wir uns für eine einfache Webanwendung zur Verwaltung von Aufgaben entschieden, kurz „YouTask“. In YouTask können Aufgaben angelegt, kategorisiert und verwaltet werden. Ein einfacher Anwendungsfall, der uns dennoch ausreicht, um den Java EE 6-Stack zu beleuchten. Schließlich ist unser Ziel, ein Template-Projekt vorzustellen, welches als Grundlage zum Aufsetzen individueller Projekte dienen kann.

Abb. 1: Die Technologien von YouTask

Im Folgenden werden wir auf die eingangs angeschnittenen Themen eingehen, die ein einheitliches und komplexes Ganzes ergeben. Wir werden unsere Aussagen an geeigneten Stellen mit Beispielcode untermauern.

Die Implementierung basiert komplett auf Java EE 6-Technologien und wurde auf dem JBoss AS7 getestet. Abbildung 1 stellt die relevanten Technologien der Anwendung dar.

Der Stack

Vor den Details der Implementierung stellen wir den Technologie-Stack vor. Generell betrachten wir primär den klassischen Web-Technologie-Stack im Rahmen des Java EE 6-Web-Profils. In unserem Beispielcode setzen wir die Spezifikationen JSF 2.0, EJB 3.1, CDI 1.0, Bean Validation 1.0 und JPA 2.0 ein. Zusätzlich verwenden wir RichFaces 4 zur Darstellung der JSF-Oberflächen. Als Erweiterung des Java EE 6-Standards setzen wir unter anderem auf die folgenden CDI-Module aus dem Seam 3-Kontext: Seam Validation, Seam Faces und Seam Catch. Die Anwendung wird mit Maven 3 [Maven] gebaut und anschließend in den JBoos AS7 veröffentlicht. In diesem läuft auch die Open-Source-Datenbank H2, welche für die Persistenz verwendet wird.

Projekt-Infrastruktur

Wie bereits im Einführungsteil beschrieben, beginnt jedes Projekt mit dem Einrichten einer sauberen und tragfähigen Projektinfrastruktur inklusive des Einbindens eines Build-Tools. Im ersten Schritt wird also eine Infrastruktur erstellt, mit der wir in der Lage sind, unseren Programmcode strukturiert zu verwalten und jederzeit ein deploybares Artefakt erstellen können. Da-

zu kommt, dass das zu veröffentlichende Artefakt sich je nach Zielumgebung, z. B. lokal oder produktiv, unterscheiden kann.

Nachdem Ant jahrelang das Tool der Wahl war, hat sich in den letzten Jahren Maven als Build-Tool im Projektalltag etabliert und ist daher auch das Build-Tool unserer Wahl. Versionsverwaltung, Erweiterbarkeit und Übersichtlichkeit sind die für uns ausschlaggebenden Argumente bei der Entscheidung pro Maven.

Die Module im Detail

Zum Einstieg in die Applikation dient zunächst eine Übersicht aller Module der mit Maven gebauten Anwendung:

- *Root:* In größeren auf Maven basierenden Projekten erfolgt üblicherweise eine hierarchische Strukturierung der beteiligten Module. Das Root-Modul stellt den zentralen Einstiegspunkt und gibt die anderen beteiligten Module an. Hier werden unter anderem zentral die Versionen der eingebundenen Bibliotheken verwaltet, sodass die Versionen Modulübergreifend konsistent bleiben.
- *Domain:* Die Entitäten werden im Domain-Projekt verwaltet. Neben Standard JPA kommt hier auch Bean Validation zum Einsatz.
- *DAO:* Dieses Modul wurde für die Kapselung des Datenzugriffs erstellt. Hierbei geht es jedoch weniger um die dem DAO-Muster (data access object) zugrunde liegende Austauschbarkeit des Datenzugriffs, sondern um Erweiterungen der vom EntityManager mitgelieferten Funktionalitäten. Zum einen bieten die DAO-Klassen Typsicherheit, zum anderen ein leichteres Arbeiten mit Abfragen.
- *Bootstrap:* In diesem Modul beantworten wir die Frage nach dem Einspielen von Initialdaten. Das Modul kann über Maven-Profile gesteuert werden. Über ein Maven-Profil für das Produktivsystem kann dieses Projekt abgeklemmt und der Import der Daten unterbunden werden.
- *Business:* Mit Hilfe dieses Moduls kapseln wir die Logik unabhängig von Persistenz und Oberfläche. Hier gehören zum Beispiel Implementierungen hinein, welche über die reine Persistenz hinausgehen oder DAO-übergreifende Operationen ausführen.
- *Web:* Dieses Modul hat mehrere Aufgaben. Zum einen beinhaltet es die JSF-Views in Form von XHTML inklusive der dazugehörigen Controller-Beans. Zum anderen wird hier im Build-Prozess das Kompilat mit der Webanwendung inklusive aller benötigten Bibliotheken erstellt. Seit EJB3.1 ist es möglich, auch EJB-Komponenten in ein Webarchiv (WAR) zu integrieren, was eine Vereinfachung für Webanwendungen mit sich bringt. Für die Demoanwendung haben wir uns daher für Paketierung vom Typ WAR entschieden.
- *IT:* IT steht hier für Integrationstests. Die einzelnen Unit-Tests befinden sich in den Test-Packages der entsprechenden Module. Das heißt zum Beispiel, dass Persistenztests entsprechend im DAO-Modul vorzufinden sind. Übergreifende Integrationstests werden hingegen hier verwaltet.

Schichtenarchitektur

Bevor wir einen genaueren Blick auf die Konfiguration des Maven-Builds werfen, sollen hier noch ein paar Worte zum Thema Schichtenarchitektur fallen. Dies ist ein Thema, bei dem sich regelmäßig die Geister scheiden. Während noch vor Java EE 5 die Softwarearchitektur in viele einzelne Schichten inklusive der dazugehörigen Konstrukte wie Data Transfer Objects (DTO) zerteilt war, geht der Trend wieder mehr in die Richtung, weniger Schichten zu bauen. Wie so oft hängt auch hier die Entscheidung von mehreren Faktoren ab – allen voran sind hier die Größe des Projekts, das Vorgehen bei der Entwicklung sowie auch die jeweiligen Präferenzen des Teams beziehungsweise des Architekten ausschlaggebend.

Wir haben uns für einen „modern konservativen" Ansatz entschieden – eine Art Kompromiss der eben skizzierten Lösungswege. In YouTask setzen wir zwar auf Schichtentrennung, verzichten aber auf die Einführung von DTOs pro Schicht. Zudem verfolgen wir auch den Weg, die Anzahl der Schichten so gering wie möglich zu halten.

Maven-Konfiguration

An dieser Stelle soll ein kurzer Blick auf ein paar der zentralen Stellen der Maven Konfiguration geworfen werden. Hierzu zeigen wir exemplarisch ein Maven-Plug-in und ein Maven-Profil.

Das Maven-Plug-in „properties-maven-plugin" ermöglicht das Einbinden und Auslesen von Properties-Dateien. In unserem Fall befindet sich im Root-Modul die Datei build.properties, welche Properties enthält, die sich auf den einzelnen Umgebungen unterscheiden. Dazu zählt zum Beispiel die absolute Pfadangabe zum JBoss-Applikationsserver.

In Listing 1 sehen wir einen Auszug aus dem Root-Modul, in welchem das Plug-in eingebunden und konfiguriert wird. Damit die Werte der hier eingebundenen Datei build.properties auch in allen anderen Modulen angezogen werden, muss in der POM (Project Object Model) der jeweiligen Module wie in Listing 2 eine Property definiert werden. Dadurch stehen global definierte Properties überall zur Verfügung und müssen nicht dupliziert werden.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>properties-maven-plugin</artifactId>
  <version>1.0-alpha-2</version>
  <executions>
    <execution>
      <phase>initialize</phase>
      <goals>
        <goal>read-project-properties</goal>
      </goals>
      <configuration>
        <files>
          <file>${project.base.dir}/build.properties</file>
        </files>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Listing 1: Konfiguration des Properties-Maven-Plug-ins

```
<properties>
  <project.base.dir>${project.parent.basedir}</project.base.dir>
</properties>
```

Listing 2: Einbinden von globalen Properties in Sub-Projekten

Mit einem Maven-Profil können diverse Profil-abhängige Konfigurationen vorgenommen werden. Hierzu zählt zum Beispiel das in der kurzen Beschreibung zum Bootstrap-Modul erwähnte Abklemmen von Bibliotheken. Listing 3 enthält einen Ausschnitt aus dem Entwickler-Profil, in welchem die Konfiguration des Persistence-Providers Hibernate durchgeführt wird. Gleichzeitig wird die Datenquelle zur H2-Datenbank des JBoss AS 7 referenziert. Damit dies funktioniert und die im Profil definierten Properties wirksam werden, müssen dieselben Properties in der Datei persistence.xml vorhanden und entsprechend parametrisiert sein. Hierdurch wird erreicht, dass die Datenbankanbindung der Anwendung je nach Zielsystem durch ein Profil einfach konfigurier- und austauschbar ist.

```
<profile>
  <id>dev</id>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
  <properties>
    <hibernate.dialect>org.hibernate.dialect.H2Dialect
    </hibernate.dialect>
    <hibernate.hbm2ddl.auto>create-drop</hibernate.hbm2ddl.auto>
    <hibernate.show_sql>true</hibernate.show_sql>
    <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
  </properties>
</profile>
```

Listing 3: Konfiguration des Entwickler-Profils

Module, Technologien und die Testumgebung im Detail

Nachdem wir den Infrastruktur-Teil angerissen haben, beleuchten wir als nächstes einige Aspekte aus dem Bereich der Architektur. Dazu werfen wir zunächst einen Blick auf das Bootstrap-Projekt und anschließend auf einige der eingesetzten Technologien.

Bootstrap-Modul

In den letzten Jahren haben wir mit nativem SQL-Import, Import mittels Hibernate-Tools oder einer Java-basierten Persistierung Erfahrungen gesammelt. Welcher der beste Weg ist, hängt vom jeweiligen Kontext ab und kann nicht pauschal beantwortet werden. Für die Demoanwendung haben wir uns für die Java-basierte Lösung entschieden. Dafür sprechen für uns die saubere Anbindung der bestehenden API (Refactoring sicher) sowie das leicht mögliche Abklemmen des kompletten Moduls in einer produktiven Umgebung. In einem separaten Maven-Modul (dem Bootstrap-Modul) werden initial die Daten durch Aufruf der entsprechenden DAO-Klassen aufgerufen. Den hierfür benötigten Code enthält Listing 4.

```
@Singleton
@Startup
public class YouTaskBootstrapBean {
  @Inject
  private TaskDao taskDao;

  @PostConstruct
  public void insert() {
    taskDao.persist(new Task("write article", Category.IDEA));
  }
}
```

Listing 4: Initialdaten mittels Bootstrap-Bean

Technologien

Wie zu Beginn bereits beschrieben, stellt das Abstimmen der vielen gemeinsam genutzten Technologien ebenfalls eine nicht triviale Aufgabe dar. In unserem Fall, wie auch in den meisten Webanwendungen, gilt es, das Zusammenspiel der folgenden Technologien aus dem Java EE-Stack zu regeln: EJB, JPA, CDI, JSF, Bean Validation. Außerdem kommt noch die Integration von Erweiterungen hinzu. Im Fall der Demoanwendung ist dies die Integration einiger Module von Seam 3.

Von besonderem Interesse und beispielhaft dafür, dass trotz des Reifegrades des Java EE-Stacks die Technologien durch Erweiterungen an Eleganz gewinnen können, ist das Modul Seam Validation. Mit Seam Validation wird eine Schwäche im Kontext der Bean Validation behoben. Mit den Standardmitteln kann im Rahmen der Bean Validation aus der Implementierung eines eigenen Validators nicht per Dependency Injection auf andere CDI-Beans zugegriffen werden. In Listing 5 ist die Implementierung eines eigenen Validators zu sehen, welcher überprüft, ob ein Anwender mit dem übergebenen Login in der Datenbank bereits existiert. Erst durch Seam Validation funktioniert die Injizierung des UserDaos.

```
public class UserNotExistsValidator implements
    ConstraintValidator<UserNotExists, String> {

  @Inject
  private UserDao userDao;

  public boolean isValid(String login,
                         ConstraintValidatorContext context) {
    return userDao.exists(login);
  }
}
```

Listing 5: Implementierung eines eigenen Validators mittels CDI

Auf alle hier genannten Technologien können wir im Rahmen eines Artikels nicht eingehen. In unserem Beispielcode bieten wir jedoch Lösungen zu der Integration und Abstimmung der genannten Technologien und Frameworks und stellen auch Lösungen für typische Querschnittsaspekte wie Internationalisierung, Exception Handling, Security und Logging zur Verfügung. Bei Internationalisierung setzen wir auf Standards, die anderen drei Aspekte bilden wir mit Seam 3-Modulen ab. Den interessierten Leser verweisen wir an dieser Stelle auf den von uns zum Download [youTask] angebotenen Quellcode.

Testumgebung

Der von uns schon mehrfach im Rahmen dieses Artikels als sehr wichtig bezeichnete Aspekt der Testumgebung soll nicht, wie so häufig in Projekten, zu kurz kommen. Die große technische Herausforderung beim Aufsetzen einer Testumgebung ist das Nachstellen eines technischen Setups, wie es der Anwendung zur Laufzeit zur Verfügung steht. In einigen Fällen ist es jedoch eher sinnvoll, die technischen Details der zugrunde liegenden Infrastruktur auszublenden und eine Funktionseinheit ohne alles andere drum herum zu testen. Es ergeben sich daher die folgenden drei Haupt-Umgebungen, die es für Tests immer wieder in Projekten aufzusetzen gilt:

- *Persistenztests*: Bei Persistenztests muss eine Umgebung erstellt werden, in welcher eine konkrete Datenbank zur Verfügung steht. In der Regel werden hierfür hauptspeicherresidente Datenbanken verwendet, welche nach jeder Testausführung auf den initialen Stand zurück gesetzt werden.
- *Mock-Tests:* Was Unit-Tests ausmacht, ist, dass diese nur eine bestimmte Funktionalität testen sollen. Schwierig wird dies jedoch, wenn die zu testende Komponente mit weiteren Komponenten verflochten ist. Diese sollen jedoch nicht mitgetestet werden. An dieser Stelle kommen Mocking-Frameworks zum Einsatz, mit welchen Aufrufe so vorgetäuscht werden können, dass nur die gewünschte Funktionalität getestet wird.
- *Integrationstests*: Bei Integrationstests geht es darum, das Zusammenspiel verschiedener voneinander abhängiger Komponenten zu testen. Die Herausforderung hier besteht darin, dass die gesamte Anwendungsumgebung während der Testausführung zur Verfügung stehen muss. Daher werden bei Integrationstests häufig Applikationsserver hochgefahren und auf diesen die Tests ausgeführt.

Für alle drei Arten der Testumgebung haben wir im Demoprojekt eine Lösung integriert. Im Rahmen der Persistenztests wird eine hauptspeicherresidente Datenbank verwendet. Zudem stehen dort alle Entitäten und die üblichen in der JPA bekannten Mechanismen zur Verfügung.

Im Bereich der Business-Schicht sollen nur die Komponenten selbst getestet werden. Daher haben wir hier eine auf Easymock [EasyMock] basierende Testumgebung erstellt.

Ein Problem bei Integrationstests war in der Vergangenheit, dass das Hochfahren des Applikationsservers mit der gesamten Anwendungsumgebung zu hohen Performanceverlusten führte. Dieses Problem ist aber seit dem Erscheinen von Arquillian [Arquillian] obsolet, da hiermit der Applikationsserver mit einem minimalen auf den Test zugeschnittenen Setup gestartet werden kann. Für Integrationstests existiert im Demoprojekt das bereits kurz vorgestellte separate Maven-Modul. In diesem sind ebenfalls Arquillian-Tests enthalten.

Deployment

Da wir uns in der Demoanwendung vollständig auf den Standard-Java-EE-6-Technologie-Stack stützen, kann die Anwendung generell in jedem Java EE 6 (Web-Profile) zertifizierten Container betrieben werden. Unsere Auswahl fiel auf die neueste Version aus dem Hause JBoss, da wir neben praktischen Erfahrungen mit diesem Container vor allem auf das saubere Zusammenspiel diverser JBoss-Projekte (Seam, Arquillian, RichFaces) setzen. Gerade die Version 7 glänzt im Vergleich zur Vorgängerversion mit beeindruckender Performance.

Unsere fertige Anwendung steht nach erfolgreichem Maven-Build als WAR-Datei zur Verfügung. Diese bringen wir am schnellsten mittels Management-Oberfläche zur Entfaltung. Alternativ steht auch ein Maven-JBoss-Plug-in zur Verfügung. Die lokal gestartete Webanwendung kann im Browser unter `http://localhost:8080/youTask` betrachtet werden.

Beispielcode

Der Quelltext der Beispielanwendung steht zur freien Verfügung unter [youTask] zum Download bereit. Die Fachlichkeit der Aufgabenverwaltung haben wir bewusst so kompakt gehalten, dass große Teile der Anwendung als Rahmenwerk zur Entwicklung eigener Applikationen verwendet werden können. Da es sich um ein GitHub-Projekt [GitHub] handelt, freuen wir uns auch über Feedback in Form von Pull Requests.

Fazit/Ausblick

Im Rahmen des Artikels haben wir eine Vorlage vorgestellt, um robuste Java EE-Anwendungen zu designen, zu entwickeln und zu erweitern. Nach einer technologischen Eingrenzung des Stacks haben wir die Bereiche Infrastruktur, Architektur und Technologie beleuchtet. Es ist entscheidend herauszustellen, wie wichtig das Zusammenspiel aller Bereiche und Komponenten ist. Gerade deshalb sehen wir den Stack mit Maven und JBoss AS7 als runde Einheit, die aus unserer Sicht den Grundstein professioneller Softwareentwicklung für die nächsten Jahre legen kann. Für einen schnellen Einstieg in das eigene Projekt dient der Quellcode der im Artikel beschriebenen Beispielanwendung. Wir hoffen, ein paar elegante und effektive Architektur- und Technologie-Entscheidungen vorgestellt zu haben.

Links

[Arquillian] `http://www.jboss.org/arquillian`
[EasyMock] `http://easymock.org`
[GitHub] Social Coding, `http://github.com`
[Maven] `http://maven.apache.org`
[Seam] JBoss Seam Webframework, `http://seamframework.org`
[youTask] Java EE sample project, `https://github.com/akquinet/javaEE6-javaspektrum`

Beispielprojekt:
https://github.com/akquinet/javaEE6-javaspektrum

Marek Iwaszkiewicz ist für die akquinet AG als Senior Software Engineer tätig. Den Schwerpunkt seiner Arbeit bilden Java EE-Architekturen und -Technologien. Sein aktueller Fokus liegt in den Bereichen Seam und jBPM.
E-Mail: Marek.Iwaszkiewicz@akquinet.de

Michael Schütz arbeitet als Senior Software Engineer bei der akquinet AG in Berlin und beschäftigt sich seit Jahren mit Java-Enterprise-Architekturen und -Technologien. Er wirkt regelmäßig an JBoss-Open-Source-Projekten mit und hat auch in diesem Jahr gemeinsam mit seinem Team die offene Java-Konferenz BED-Con 2012 in Berlin organisiert.
E-Mail: Michael.Schuetz@akquinet.de

(Twitter @michaelschuetz)