Bestandsaufnahme

Java EE 6 in der Praxis

Christian Unglaube

Die Java EE 6-Spezifikation ist nunmehr seit über zwei Jahren verabschiedet und in der Fachpresse und -literatur erschienen bereits zahlreiche Veröffentlichungen rund um die neuen Features. Diese werden durchweg positiv bewertet und versprechen, die Entwicklung von (Web-)Anwendungen mit Java EE deutlich zu vereinfachen. In der Praxis jedoch erweist sich die Implementierung von Anwendungen auf dieser Plattform in vielen Umfeldern zurzeit als noch nicht so problemlos, wie es auf den ersten Blick erscheinen mag. Bei der Suche nach alternativen Ansätzen für die Integration von Java EE 6-Teiltechnologien, wie JSF und JPA, stellt vor allem das Spring-Framework seine Nützlichkeit unter Beweis. Ziel dieses Artikels ist es nicht, für Java EE 6 oder Spring 3 eine Lanze zu brechen. Vielmehr geht es darum, die Erfahrungen aus einem realen Projekt, welches Ende 2010 produktiv ging, objektiv darzustellen.

Die Ausgangssituation

Manchmal scheinen die Dinge einfach gut zu laufen und das gerade dann, wenn man am wenigsten damit rechnet. Ein Kunde kommt mit einem Anliegen, das man als IT-Beratungsunternehmen nur zu gerne erfüllt: Er möchte seine in die Tage gekommene Webanwendung modernisiert haben. Diese läuft auf einer Java-basierten Plattform, für die es jedoch kaum noch Support gibt, und an Erweiterungen im Sinne einer modernen Webanwendung ist überhaupt nicht zu denken. Wichtig bei der Auswahl der neuen Plattform ist für den Kunden der Aspekt der Zukunftssicherheit und die damit verbundene Investitionssicherheit. Um dies zu erreichen, soll so weit wie möglich auf Standards gesetzt werden.

Was liegt näher, als diesem Kunden die neue Java EE 6-Spezifikation vorzustellen und ihm zu erläutern, welche standardisierten Features ihm damit zur Verfügung stehen? Die Begeisterung ist nicht nur beim Kunden groß: Auch unser Entwicklerteam freut sich darauf, eine Anwendung mit den neuen und vielversprechenden Java EE 6-Technologien umsetzen zu dürfen.

Das Projekt beginnt

Zu Projektbeginn muss neben dem Technologie-Stack auch die Infrastruktur festgelegt werden, in der die Anwendung laufen wird. Als Applikationsserver wünscht sich der Auftraggeber den verbreiteten JBoss-Server [JBos] aus dem Hause Red Hat, denn dieser wird von seinem Infrastrukturbetreiber unterstützt. Leider ist zu diesem Zeitpunkt (September 2010), ca. ein Jahr nach der Verabschiedung der JEE6-Spezifikation, lediglich ein einziger Anbieter in der Lage, einen konformen Applikationsserver anzubieten, und das ist leider nicht JBoss, sondern GlassFish, die Referenzimplementierung von Sun/Oracle.

Und damit beginnen die ersten Probleme: GlassFish wird vom Infrastrukturbetreiber abgelehnt. Java EE 6 alternativ auf dem JBoss 5.1 stabil zum Laufen zu bringen, erweist sich schnell als aussichtsloses Unterfangen, und so platzt der Java EE 6-Traum. Wir suchen fieberhaft nach Alternativen, um wenigstens einen Teil der Java EE 6-Versprechen einlösen zu können. Die erste Überlegung ist, zu versuchen, so viele Java EE 6-APIs wie möglich auf einer Java EE 5-Umgebung zu nutzen, und nur dort, wo es notwendig ist, auf Ersatzlösungen auszuweichen. Das vielseitige und seit Jahren bewährte Spring-Framework soll dabei helfen, Teiltechnologien der Java Enterprise Edition, wie JSF und JPA, miteinander zu verbinden.

Variante 1: JBoss 5.1 mit Spring 3 und Java EE 6

Wir versuchen, auf die 2010 aktuelle JBoss-Version 5.1 zu setzen und dort die Anwendung auf Basis der Java EE 6-Standards JSF 2.0 und JPA 2.0 zu implementieren. Diese Idee müssen wir aber begraben, da es zu Problemen bei der Integration der notwendigen Bibliotheken in den JBoss 5.1-Server kommt. Die Klassenpfadkonflikte lassen sich nicht ohne massive Eingriffe in die Untiefen der JBoss-Konfiguration lösen. Also nehmen wir schnell Abstand davon und suche nach einer anderen Lösung.

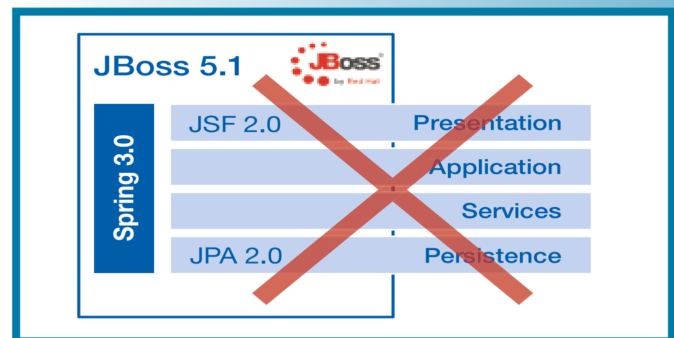

Abb. 1: Technologie-Stack, Variante 1

Variante 2: Tomcat 6 mit Spring 3 und Java EE 6

In Abstimmung mit unserem Kunden entschließen wir uns, auf den Einsatz von EJBs zu verzichten, denn damit kommt der produktiv langjährig erprobte Tomcat als Anwendungsplattform infrage. Tomcat wird glücklicherweise in der Version 6.0 vom Infrastrukturbetreiber unterstützt. Schnell stellt sich heraus, dass wir mit dieser Variante, Java EE 6-APIs im Tomcat zu verwenden, Erfolg haben: Ein kurzfristig implementierter Anwendungsprototyp läuft stabil und performant. In den Genuss der Neuerungen der Servlet-Spezifikation 3.0, z. B. für asynchrone Requestverarbeitung, kommen wir allerdings nicht, denn Tomcat 6.0 implementiert nur die Version 2.5. Für die Projektanforderungen spielt diese Einschränkung jedoch keine Rolle.

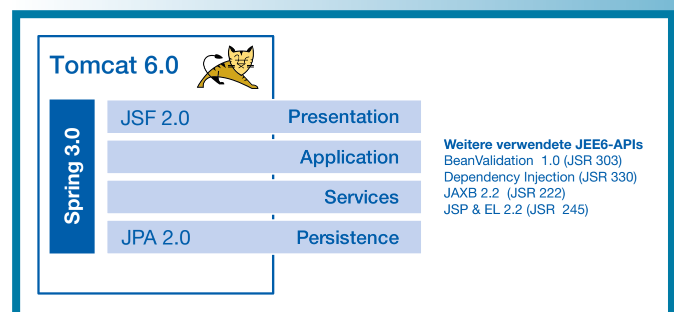

Abb. 2: Technologie-Stack, Variante 2

So viel Java EE 6 wie möglich

Um eine spätere Migration auf weitere Teile des Java EE 6-Stacks zu erleichtern, binden wir, wo es möglich ist, entsprechende Lösungen ein. So werden in der Persistenzschicht grundsätzlich JPA-konforme Annotationen bevorzugt und auf Hibernate-spezifische Annotationen wird weitestgehend ver-

zichtet. Genauso werden Spring-spezifische Annotationen vermieden, wo es nur möglich ist.

Die Entscheidungsfindung für eine auf Java EE 6 basierende Architektur hat sich als nicht so einfach erwiesen, wie zu Projektbeginn angenommen. Letztendlich haben wir mit Spring 3, JSF 2.0, JPA 2.0 und weiteren Java EE 6-APIs (Dependency Injection: JSR 330, Bean Validation: JSR303 usw.) aber einen modernen und leistungsfähigen Technologie-Stack für die Anwendung gefunden.

In den nachfolgenden Abschnitten wird im Einzelnen auf besondere Details und Herausforderungen bei der Realisierung der Anwendung eingegangen.

Die resultierende Anwendungsarchitektur

Die Anwendung wird im klassischen Schichtenaufbau für Webanwendungen realisiert. In Abbildung 3 sind die Schichten im Überblick sowie die wichtigsten Bibliotheken, die darin genutzt werden, dargestellt.

Frontend mit JSF 2.0 und PrimeFaces

JSF 2.0 ist Bestandteil der Java EE 6-Spezifikation und somit ohne große Diskussion als Frontend-Technologie gesetzt. Die Auswahl an Komponentenbibliotheken, die JSF 2.0 unterstützen, ist allerdings erstaunlich klein. Die populärsten Kandidaten wie RichFaces und IceFaces bieten noch keine finalen Versionen mit „echter" JSF 2.0-Unterstützung. Eine Evaluierung der aktuellen Marktsituation lässt unsere Entscheidung schließlich auf PrimeFaces [PF] fallen, denn diese Bibliothek ist für JSF 2.0 geeignet und bietet eine reichhaltige Auswahl an Komponenten.

Spring 3.0 für Querschnittsfunktionalität

Für die typischen Belange einer Webanwendung benutzen wir verschiedene Module des Spring-Frameworks: Logging, Transaktions- und Exceptionhandling werden mithilfe von AOP-Interceptoren deklarativ und damit zentralisiert gelöst.

Darüber hinaus stellt Spring natürlich den bekannten IoC-Container (Inversion of Control) bereit, welcher intensiv für das Zusammenspiel der Anwendungskomponenten genutzt wird. Die Konfiguration der Abhängigkeiten erfolgt mit JSR330-konformen Annotationen (`@Inject`, `@Named`), die sowohl von Spring als auch von Java EE 6 unterstützt werden. Sobald Komponenten jedoch umgebungsabhängig konfiguriert werden müssen, greifen wir aus Gründen der Wartbarkeit auf die XML-basierte Konfiguration zurück.

Verwendung von Scopes

Für die Deklaration der Scopes von Spring/JSF-Beans müssten wir eigentlich auch auf die proprietären Spring-Scopes zurückgreifen. Zum Glück gibt es aber die Möglichkeit, in Spring eine Annotation-Bridge zu implementieren, über die ein Mapping von CDI-Scopes (Context and Dependency Injection) auf die entsprechenden Spring-Scopes erfolgen kann. Hierfür kann das Interface `ScopeMetadataResolver` implementiert und konfiguriert werden [Wes10]. Auf diese Weise erreichen wir die applikationsweite Verwendung von ausschließlich Java EE 6-konformen Scope-Annotationen. In Tabelle 1 sind die Standard-Scopes der einzelnen APIs gegenübergestellt.

Leider gibt es nicht für alle Scopes in jedem Umfeld das entsprechende Pendant. Für Request-, Session- und Application-Scopes kann einfach mit dem beschriebenen Spring-`ScopeMetadataResolver` gearbeitet werden. Möchte man in seiner Anwendung jedoch den Conversation-Scope einsetzen, um z. B. einen Mehrfenster- oder Tabbed-Browsing-Betrieb zu ermöglichen, so muss man auf zusätzliche Bibliotheken zurückgreifen.

Der Versuch, Apache Orchestra dafür einzubinden, ist nicht von Erfolg gekrönt: Es kommt im Zusammenspiel von JSF 2.0 und Spring 3.0 zu einem Deadlock in einer Synchronisationsklasse, die eigentlich für die Vermeidung von Race-Conditions bei Bean-Aufrufen zuständig ist.

Wir entscheiden uns schließlich für eine eigene Implementierung des Conversation-Scopes. Dafür erweitern wir den Spring-Scope `AbstractRequestAttributesScope` und

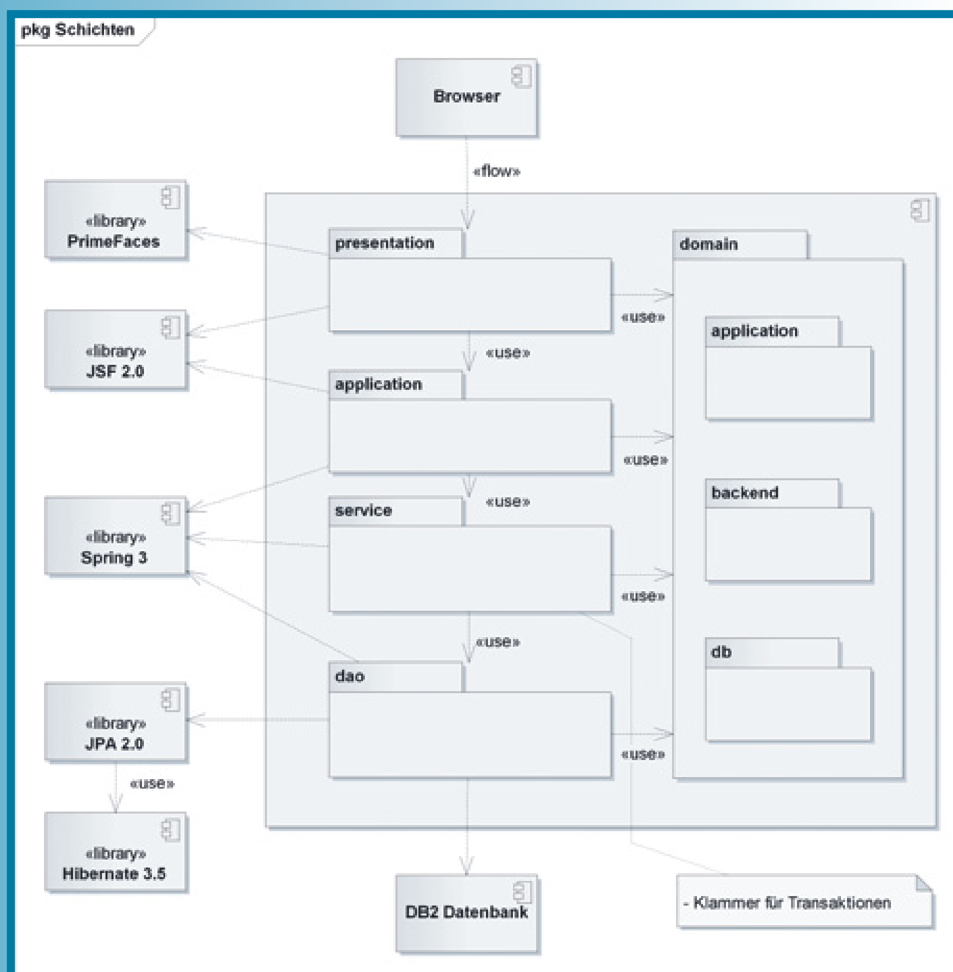

Abb. 3: Anwendungsarchitektur

CDI-Scopes	Spring-Scopes	JSF-Scopes
@RequestScoped	@Scope(„request“)	@RequestScoped
@SessionScoped	@Scope(„session“)	@SessionScoped
@ApplicationScoped	@Scope(„singleton“)	@ApplicationScoped
@ConversationScoped	--	--
--	--	@ViewScoped
--	@Scope(„prototype“)	--
--	@Scope(„global session“)	--
@Dependent	--	--

Tabelle 1: Gegenüberstellung der Scopes

implementieren einen JSF-Listener, welcher in der JSF-Phase „Update-Model-Values“ für die Zuordnung der passenden Conversation anhand der Conversation-ID zum jeweiligen Request zuständig ist.

Möchte man in der Anwendung den View-Scope nutzen, der für CDI nicht definiert ist, so bleibt die Möglichkeit, diesen Scope selbst zu implementieren. Eine einfache Beispielimplementierung stellt Steven Verborgh in seinem Blog vor [Ver10].

Komfortable Validierung mit Bean Validation

Eines der Java EE 6-Highlights stellt die Bean Validation (JSR 303) dar, welche nun nicht nur für die Entity-Validierung beim Persistieren von Objekten mittels JPA, sondern auch für die Eingabe-Validerung am JSF-Frontend nutzbar ist. Das bedeutet, dass die Validierungsannotationen in Entity-Klassen und die damit zusammenhängenden Fehlermeldungen von den JSF-Backing-Beans implizit mit genutzt werden. Hierdurch kann eine verteilte und oftmals redundante Implementierung der Validierungslogik in Front- und Backend vermieden werden [ZO11].

Datenhaltung über JPA

Die Speicherung von Anwendungsdaten in einer relationalen Datenbank erfolgt über einen JPA-konformen O/R-Mapper unter Verwendung des DAO-Patterns (Data Access Object). Hierzu verwenden wir die JPA-Implementierung von Hibernate und das Spring-JPA-Template. Beim Annotieren der Entitys achten wir darauf, mit zu JPA konformen Annotationen zu arbeiten, um standardkonform zu bleiben.

Über den Einsatz des Spring-JPA-Templates kann man geteilter Meinung sein. Es bietet funktional gesehen wenig Mehrwert, erleichtert aber den Spring-affinen Entwicklern die Eingewöhnung, denn es ist wie das gewohnte JDBC- oder Hibernate-Template aufgebaut und unterstützt in gleicher Weise beim Umgang mit dem EntityManager.

Mit der Version 2.0 von JPA wurde eine standardisierte Criteria-API eingeführt, die zumindest teilweise die von Hibernate bekannte Funktionalität bereitstellt und mit der sich der Hibernate-erfahrene Entwickler relativ schnell zurechtfindet, auch wenn sie etwas umständlicher in der Handhabung ist.

Sicherheit mit Spring-Security

Die Sicherheitsanforderungen an die Anwendung sind hoch und komplex. Es gibt eine Vielzahl von Benutzerrollen, Anwendungsbereichen und Funktionalität, die nur für bestimmte Rollen zugänglich sein dürfen. Mit Spring-Security lassen sich diese Bereiche einfach deklarativ über Regeln in Form von URL-Mustern und Package-, Klassen- und Methoden-Mustern absichern, ohne dass dabei die Geschäftslogik im Code durch Sicherheitsabfragen verwässert wird.

```
<security:http
  entry-point-ref="casProcessingFilterEntryPoint"
  access-denied-page="/globalAuthenticationError.jsf">
<security:custom-filter ref="casAuthenticationFilter"
                        after="CAS_FILTER"/>
<security:intercept-url
  pattern="/javax.faces.resource/**" filters="none" />
<security:intercept-url pattern="/resources/css/**" filters="none" />
<security:intercept-url pattern="/resources/img/**" filters="none" />
<security:intercept-url pattern="/detail.jsf" access="ROLE_EDITOR" />
<security:intercept-url
  pattern="/uebersicht.jsf" access="ROLE_SUPERUSER" />
<security:intercept-url pattern="/**/*" access="ROLE_READER" />
</security:http>
security:global-method-security>
<security:protect-pointcut
  expression="execution(* de.bit.app.service.KonfigService.save(..))"
  access="ROLE_SUPERUSER"/>
<security:protect-pointcut
  expression="execution(*de.bit.app.service.TestService*.execute*(..))"
  access="ROLE_EDITOR,ROLE_SUPERUSER "/>
<security:protect-pointcut
  expression="execution(*de.bit.app.service.*Service*.save(..))"
  access="ROLE_EDITOR,ROLE_SUPERUSER"/>
<security:protect-pointcut
  expression="execution(*de.bit.app.service.*Service*.*(..))"
  access="ROLE_READER,ROLE_EDITOR,ROLE_SUPERUSER"/>
</security:global-method-security>
```

Listing 1: Deklarative Spring-Security-Konfiguration

Die Integration eines SingleSignOn über ein proprietäres Token-Verfahren lässt sich mit Spring-Security einfach in die Anwendung integrieren und auch der nachträgliche Austausch dieses Verfahrens gegen eine CAS-Server-Authentifizierung [CAS] ist mit wenig Aufwand und ohne Änderungen an der Anwendungsarchitektur möglich. Erscheint Spring-Security zu Beginn doch recht komplex, zeigt sich an dieser Stelle, wie flexibel es einsetzbar ist. Im Vergleich dazu lässt die JAAS-Implementierung der Applikationsserverhersteller auch mit Java EE 6 noch Wünsche offen. Eine Auswahl von JAAS-konformen Authentifizierungsverfahren ist in die Container integriert und deckt die Standardfälle (BASIC, DIGEST, FORM, CLIENT-CERT) ab. Für darüber hinaus gehende Authentifizierungsverfahren ist man jedoch auf die JAAS-Implementierung von Drittanbietern angewiesen, die nicht immer für jeden Applikationsserver verfügbar sind.

Durch die Verwendung von Spring-Security bringt die Anwendung ihre komplette Sicherheitsinfrastruktur selbst zum Deployment mit und ist somit nicht an einen bestimmten Container gebunden. Daraus resultiert auch eine einfachere Portierbarkeit der Anwendung auf andere Java EE-Server. Aus diesen Gründen bevorzugen wir Spring-Security als Sicherheits-Framework, auch wenn wir dabei die Java EE 6-konformität vernachlässigen müssen.

Testen mit Spring

Das automatisierte Testen von Anwendungen über klassische Modultests hinaus stellt immer wieder eine Herausforderung dar, besonders dann, wenn Tests im Kontext von Frameworks ausgeführt werden müssen. Gerade, wenn bei der Implementierung der Test-Driven-Development-Ansatz [TDD] im Zusammenspiel mit einer Continuous-Integration-Umgebung [Fow06] gelebt wird, ist es wichtig, dass möglichst alle Anwendungskomponenten automatisiert testbar sind.

Wir verwenden jUnit, um sowohl klassische Modultests als auch Integrationstests durchzuführen. Bei Letzteren erweist sich das Spring-Test-Framework als wertvolle Hilfe, denn dieses bietet die Möglichkeit, innerhalb von Unit-Tests mit we-

nigen Codezeilen einen Anwendungskontext zu generieren, über den die Anwendung auch Schichten übergreifend getestet werden kann. Aus dem Anwendungskontext heraus kann mit Spring 3 außerdem sehr einfach eine Datenbank on-the-fly generiert und mit Testdatensätzen initialisiert werden.

Deploymentprozess

Die Anwendung wird mit Maven gebaut, getestet und in eine WAR-Datei gepackt. Während des Entwicklungsprozesses durchläuft sie verschiedene Umgebungen: lokale Entwicklungsumgebung, CI-Server inhouse, Entwicklungsumgebung inhouse, CI-Server Kunde, Entwicklungsumgebung Kunde, Integrationsumgebung und schließlich Produktivumgebung sind die einzelnen Stationen. Für jede dieser Umgebungen müssen neben den gemeinsamen Konfigurationsdaten eigene Konfigurationsparameter gepflegt werden, damit die Anwendung in der jeweiligen Umgebung lauffähig ist. Über eine Umgebungsvariable können wir beim Generieren des Applikationskontextes definieren, für welche Umgebung die Anwendung konfiguriert werden soll. Die entsprechende Konfiguration wird dann beim Starten der Anwendung geladen.

JSF bietet ebenfalls das Konzept von umgebungsabhängigen Konfigurationen in Form von ProjectStages (Production, Development, UnitTest, SystemTest und Extension) an, die das Verhalten von JSF z. B. beim Anzeigen von Fehlermeldungen beeinflussen. Da sich diese nicht mit unserer Projektlandschaft decken, passen wir sie kurzerhand mithilfe einer eigenen `ServletContextListener`-Implementierung an unsere Umgebungen an. Das Mapping von ProjectStage zu Umgebung definieren wir in einer von Spring verwalteten Map (s. Listing 2).

```
<util:map id="prodLevel2StagesMap" map-class="java.util.HashMap">
<entry key="local_hsql"  value="Development" />
  <entry key="local_db2"   value="Development" />
  <entry key="unit_test"   value="UnitTest" />
  <entry key="ci"     value="UnitTest" />
  <entry key="dev"     value="SystemTest" />
  <entry key="integra"   value="SystemTest" />
  <entry key="prod"     value="Production" />
</util:map>
```

Listing 2: Mapping PROD_LEVEL auf JSF-Stages

Migration: Der Weg zu Java EE 6 pur

Unsere Anwendungsarchitektur auf einen reinen Java EE 6-Stack zu migrieren, ist mit relativ überschaubaren Aufwänden möglich und das ohne tief greifende Eingriffe in die Schichtenarchitektur. Da sowohl der Spring-IoC-Container als auch ein CDI-Container (z. B. Weld als Referenzimplementierung) den Dependency-Injecton-Standard (JSR 330) implementieren, können sämtliche Spring-Beans, die mit `@Named` und `@Inject` annotiert sind, einfach in den CDI-Container „geworfen“ werden und sind dort genauso lauffähig. Aber es gibt auch Bereiche, die nicht so einfach portierbar sind.

Der Hauptunterschied zwischen einer Spring-basierten und einer Java EE 6-Anwendung, der ins Auge fällt, ist, dass Spring mehr Anwendungskonfiguration benötigt, während für Java EE 6-Anwendungen ein großer Teil dieser Konfigurationen nicht in der Anwendung, sondern im JEE-Container erfolgen. Das heißt, die JEE-Anwendung wird leichtgewichtiger, aber auch stärker in Abhängigkeit zum Java EE-Server konfiguriert. Spring-Anwendungen werden im Gegensatz dazu vollständig in sich selbst konfiguriert und nutzen dafür in der Regel eine Reihe von application-context.xml-Dateien, während es für Java EE 6-Anwendungen lediglich eine Standardkonfigurationsdatei für jede API gibt.

Java EE API	Konfiguration
Java Persistence API (JPA)	`/META-INF/persistence.xml`
Enterprise Java Beans WAR	`/WEB-INF/ejb-jar.xml`
Java Server Faces (JSF)	`/WEB-INF/faces-config.xml`
Contexts & Dependency Injection (CDI)	`/WEB-INF/beans.xml`
Web Configuration	`/WEB-INF/web.xml`
JAX-RS (REST Web Services)	`/WEB-INF/web.xml`

Tabelle 2: Java EE-Konfigurationsdateien

In Tabelle 2 sind die typischen Java EE-Konfigurationsdateien dargestellt, die für die Einrichtung der einzelnen Technologien zu verwenden sind. Nur die fett dargestellten Dateien sind dafür notwendig, alle anderen Konfigurationen können auch vollständig durch Annotationen erfolgen.

Beim vollständigen Eliminieren des Spring-Frameworks aus der Anwendung wird deutlich, in wie vielen Bereichen es Unterstützung bei der Framework-Integration liefert. Vor allem eine gleichwertige Alternative zum Spring-Security-Modul zu finden, ist problematisch, sodass wir zunächst darauf verzichten und eine ansonsten „reinrassige“ Java EE 6-Anwendung weiterhin mit Spring-Security absichern.

Für Beans, welche einen transaktionalen Kontext benötigen, könnte man beispielsweise das Apache MyFaces CODI-Projekt [COD] für das Transaktionshandling nutzen. Allerdings stellt sich dann unmittelbar die Frage, warum man in diesem Fall nicht gleich den EJB-Container des Applikationsservers nutzt und die Bean einfach entsprechend als `@EJB` annotiert. Das Gleiche trifft auf die Verwendung von zeitgesteuerten Komponenten zu: Spring bietet hierfür ein einfaches Task-Scheduling über die Spring-Annotation `@Scheduled` an, welches mit EJB 3.1 durch eine `@Schedule`-Annotation ersetzt werden kann.

Ansonsten fällt beim Portieren auf, dass es eine ganze Reihe von Spring-Helferlein und -Modulen gibt, die den Entwickleralltag deutlich vereinfachen. Für diese gibt es häufig kein standardisiertes Pendant, sodass meist nur der Weg einer Eigenimplementierung bleibt oder die Suche nach einer zuverlässigen Alternative in den Weiten des Internets. Und ob das sinnvoll ist, darüber lässt sich streiten.

Fazit

Nach langem Warten und vielen Vorschusslorbeeren nimmt Java EE 6 allmählich Fahrt auf. Es wird sich sicherlich als feste Größe etablieren und damit auch dem Spring-Framework zunehmend Konkurrenz machen. Die Threads mit dem Thema Spring 3 contra Java EE 6, die in verschiedenen Foren, Blogs und Artikeln zu beobachten sind, erscheinen überzogen, denn letztendlich haben beide Technologien ihre Daseinsberechtigung und decken unterschiedliche Bedürfnisse ab. Es zeigt sich viel mehr, dass sich Spring 3 und Java EE 6 gut kombinieren lassen und damit einen tatsächlichen Mehrwert für eine Anwendungsarchitektur bieten können.

Java EE 6 ermöglicht die Auslieferung kleinerer Applikationen (auf die Größe des Deployment-Artefakts bezogen), da im Gegensatz zu Spring-Anwendungen die Infrastruktur bereits auf dem Server vorhanden ist und nicht mit deployed werden muss. Aber in welchem realen Projektumfeld spielt das tatsäch-

lich eine relevante Rolle? Auf der anderen Seite verliert man bei einer Java EE 6-Anwendung auch ein Stück weit die Kontrolle über die Versionen der verwendeten Bibliotheken bzw. über die verwendete Implementierung von Spezifikationen wie JPA, denn man muss sich danach richten, was die Serverumgebung anbietet.

Spring bietet weit mehr als nur den IoC-Container und stellt eine große Bandbreite an etablierten Modulen für eine Vielzahl an Belangen zur Verfügung. Weld und andere CDI-Container mit ihrem umgebenden Ökosystem werden sich hier erst noch beweisen müssen. Die Möglichkeit der typsicheren Injection ist auf jeden Fall ein Pluspunkt für CDI. Genauso die Einführung von CDI-Events, welche bei der Konzeption von Java EE-Lösungen mit Sicherheit nützlich sein werden.

Mitte 2010, als das oben beschriebene Projekt begann, war die vorgestellte Architektur die aus unserer Sicht einzig sinnvolle Alternative. Die Anwendung ist mittlerweile stabil im produktiven Betrieb. Unsere Erfahrungen mit dieser Konstellation waren durchweg positiv, sie hat zwischenzeitlich erfolgreich Verwendung in weiteren Projekten gefunden. Inzwischen hat sich an der Java EE-Front einiges getan, etwa mit der Finalisierung des JBoss 6.0. Im Laufe der nächsten Monate werden sicherlich mehr und mehr Projekte als reine Java EE 6-Anwendungen entwickelt werden, und auch wir freuen uns darauf, die erste Anwendung unter vollständiger Nutzung dieser Plattform zu entwickeln.

Literatur und Links

[CAS] Central Authentication Service, `http://www.jasig.org/cas`
[COD] Apache MyFaces Extensions CDI, CODI, `https://cwiki.apache.org/EXTCDI/`
[Fow06] Continuous Integration, `http://martinfowler.com/articles/continuousIntegration.html`
[JBos] JBoss Application Server, `http://www.jboss.org/jbossas/`
[JEE6] Java EE 6 Technologies, `http://www.oracle.com/technetwork/java/javaee/tech/index.html`
[PF] PrimeFaces, `http://www.primefaces.org/`
[SpringSource] `http://www.springsource.org/`
[TDD] Testgetriebene Entwicklung, `http://de.wikipedia.org/wiki/Testgetriebene_Entwicklung`
[Ver10] St. Verborgh, Blog, `www.verborgh.be/articles/2010/01/06/porting-the-viewscoped-jsf-annotation-to-cdi/`
[Wes10] M. Wessendorf, Webblog, Blog, `matthiaswessendorf.wordpress.com/2010/05/06/using-cdi-scopes-with-spring-3/`
[ZO11] S. Zambrovski, O. Ochs, Türsteher für Bohnen – Bean Validation mit domänenspezifischen Typen in der Praxis, in: Java Magazin, 4/2011

Christian Unglaube ist Senior Consultant beim IT-Beratungsunternehmen BridgingIT GmbH. Als Java/JEE-Experte und Softwarearchitekt verfügt er über langjährige Erfahrung bei der Durchführung von Projekten in der Finanzdienstleistungs- und Pharmabranche. Christian Unglaube ist Sun Certified Enterprise Architect.
E-Mail: christian.unglaube@bridging-it.de