Fallstricke bei der Entwicklung einer JEE6 Anwendung

Moritz Lippe
syntegris information solutions GmbH
Neu-Isenburg

Schlüsselworte
Java, Java EE 6, JSF 2, EJB 3.1

Einleitung

Java EE 6 (JEE6) ist der aktuelle Standard für die Entwicklung von Java Enterprise Anwendungen. Der Standard verspricht verbesserte und einfachere Entwicklung durch aktualisierte API-Standards und die Integration populärer Frameworks bzw. deren Ideen.

In diesem Projektbericht wird beleuchtet, wie sich die Entwicklung einer JEE6-Anwendung in der Praxis dargestellt hat. Dabei wird darauf eingegangen, warum wir den Standard verwenden wollten und welche Vor- und Nachteile wir in der Entwicklungs-, Deployment- und Betriebsphase hatten. Zudem wird kurz auf die Migration der JEE6-Anwendung vom GlassFish Application Server auf einen JBoss Application Server eingegangen.

Ein neuer Technologiestack soll her

Die Auswahl eines Frameworks zur Entwicklung einer modernen Webanwendung ist im Java-Bereich alles andere als eine leichte Entscheidung. Es gibt unzählige Open Source und proprietäre Frameworks, aus denen man wählen kann. Ziel sollte natürlich immer die passende Technologie für das aktuell anliegende Projekt sein.

Die Entscheidungskriterien, welche Technologie man einsetzen kann, bestimmen sich häufig aus weiteren Punkten:

- Eine Technologie wurde bereits bei einem vorhandenen Projekt eingesetzt und soll auch für das neue Projekt verwendet werden.
- Die Projektanforderungen sehen eine bestimmte Technologie vor (z.B. im Lastenheft oder in der Ausschreibung vorgegeben).
- Ein Entscheidungsträger macht eine strikte Vorgabe.

Der Erfolg eines Projektes wird auch maßgeblich durch die Technologie-Auswahl beeinflusst. Hier stellen sich bei der Technologie-Auswahl folgende Fragen:

- Wie komplex und zeitintensiv gestaltet sich die Umsetzung einer Anforderung? Hier gibt es beispielsweise Frameworks, die schnell Resultate durch Code-Generierung liefern.
- Kann man alle Anforderungen mit der Technologie abbilden? Gerade bei Frameworks, die auf Code-Generierung setzen, kann man hier Probleme bekommen oder es ergibt sich ein recht hoher Anpassungsaufwand, wenn man vom Standardvorgehen abweichen muss.
- Wie zukunftssicher ist die Technologie? Bei langlaufenden Anwendungen ist dieser Punkt sehr relevant und auch mit einem recht hohen Risiko behaftet.
- Wie sehen Dokumentation und Support aus? Findet man im Internet Beispiele (gerade bei Open Source sehr wichtig)? Ein Framework kann noch so gut sein – wenn man nicht herausfindet, wie man bestimmte Aufgabenstellungen umsetzen kann, so erzeugt dies Zusatzaufwand oder führt womöglich zu einem Scheitern des Projekts.
- Ist die Technologie im Markt weit verbreitet und findet man dazu qualifizierte Entwickler? Gerade für Auftraggeber ist dieser Punkt wichtig, wenn man ein langlaufendes Projekt plant, das kontinuierlich weiterentwickelt werden soll.

Das Projekt, auf dem dieser Erfahrungsbericht beruht, hatte bezüglich der Technologie-Auswahl nur wenige Vorgaben. Ziel sollte die Ablösung einer Altanwendung durch eine neue Webanwendung sein. Hierbei wurde der Fokus auf die Neugestaltung der vorhandenen (fachlichen) Prozesse gelegt. Die Technologie konnte also durch Syntegris vorgeschlagen werden – die Wahl fiel auf JEE6. Ausschlaggebend hierfür war eine Evaluationsphase, in der die neue JEE Version überprüft wurde. Die Evaluationsphase fand im ersten Quartal 2011 statt.

Die Enterprise Edition der Java Plattform (kurz Java EE oder JEE) spezifiziert Softwarearchitekturstandards, die zur Entwicklung von Unternehmenssoftware eingesetzt werden können. JEE stellt dabei eine Sammlung an API-Standards bereit, die zu einer JEE-Version zusammengefasst werden. Der aktuelle Standard hat die Versionsnummer 6, deren Final Release am 10. Dezember 2009 veröffentlicht wurde. Für JEE7 (JSR 342) werden von der Expert Group bereits erste Spezifikationsentwürfe erstellt.

Bis zur Veröffentlichung von Java EE 5 im Jahr 2006 wurde dem JEE/J2EE-Stack in Teilen zu Recht die Anwendbarkeit abgesprochen. Zu viel unnötiger Code musste geschrieben werden, der vorwiegend technologisch begründet war. Mit der Einführung / Verwendung von Java Annotationen in JEE5 wurde der Anteil dieses „boilerplate code“ spürbar reduziert, so dass die Konzentration auf das Fachliche in den Vordergrund trat. JEE6 verspricht eine Weiterführung dieser Entwicklung.

Vorträge auf Java-Fachkonferenzen und Artikel, die die Neuerungen von JEE6 vorstellten, hörten sich vielversprechend an. Viele Ideen aus anderen Frameworks sollten in den Standard übernommen werden und für bereits vorhandene Teile wurden Updates geplant. Dabei wurde stets das bessere Zusammenspiel der im Standard enthaltenen Technologien hervorgehoben.

Als Alternative zu dem Standard-Technologiestack gab es das Spring-Framework in der damals neuen Version 3. Im direkten Vergleich zwischen Spring und JEE wurde seitens Syntegris die strategische Bedeutung dieser beiden Kontrahenten in den nächsten Jahren klar zugunsten von JEE eingeschätzt. Die im Verhältnis zu Spring bei JEE größere Entwicklergemeinde trägt zudem zu einem verringerten Risiko bei, was sich (trotz der zuvor genannten Startschwierigkeiten) in einer langen Historie erfolgreicher und auch heute noch in Betrieb befindlicher JEE Anwendungen wiederspiegelt.

Anfang 2011 gab es zwar bereits seit längerem die finalisierte JEE6-Spezifikation, jedoch benötigte man für die Umsetzung einen kompatiblen Application Server. Zum Zeitpunkt der Evaluation von JEE6 wurde deshalb der GlassFish Application Server ausgewählt. Der GlassFish Application Server stellt die Referenz-Implementierung von Oracle für JEE6 dar. Im Rahmen der Evaluation wurde ein Referenz-Projekt für die interne Entwicklung aufgesetzt. Zum Einsatz kamen dabei JavaServer Faces (JSF) 2, Enterprise Java Beans (EJB) 3.1, Java Persistence API (JPA) 2 und Context and Dependency Injection (CDI).

Letztendlich waren die Erwartungen an JEE6 relativ hoch:

- Kürzere Entwicklungszeit durch neue Features und weniger „boilerplate code“
- Leichtgewichtige Architektur durch CDI
- Hohe Akzeptanz und Verbreitung

Wie soll das funktionieren?
Die Evaluationsphase sollte zeigen, dass die Erwartungen an JEE6 erfüllt werden können, und dass das Projekt-Team einen Vorteil gegenüber der Verwendung eines alten (aber bewährten) Technologiestacks hat.

Der grundsätzliche Rahmen wurde durch die Referenz-Implementierung vorgegeben. Darüber hinaus gehend wurden einige weitere Frameworks und Bibliotheken herangezogen. So wurde ICEfaces als JSF-Komponentenbibliothek ausgewählt und Hibernate als JPA-Provider.

Das Evaluationsprojekt wurde für einen GlassFish 3.1 Server entwickelt und war die Vorlage für das folgende Projekt, das für einen Kunden in der Finanzdienstleistungsbranche entwickelt wurde. Start des Projektes war Juli 2011. Ziel war die Entwicklung einer Webanwendung zur Erfassung von Finanzierungskreditanträgen für Privatkunden.

Hindernisse
Bei der Entwicklung der ersten Unternehmensanwendung merkten wir schnell den Unterschied zwischen einem Testprojekt und der Umsetzung „realer“ Anforderungen. Die ersten Schwierigkeiten stellten sich bei der Suche nach geeigneten Beispielen und Dokumentation im Internet. Häufig handelte es sich bei den gefundenen Beispielen um simple Aufgabenstellungen, mit denen wir in einem komplexeren Zusammenhang relativ wenig anfangen konnten. Dies wurde vor allem bei dem Zusammenspiel zwischen CDI und EJB deutlich. Gute Vorlagen für dieses Szenario waren anfangs Mangelware.

Fehlende Dokumentation war eine weitere Hürde. So sollte zum Beispiel der neu eingeführte Conversation-Scope für die JSF-ManagedBeans verwendet werden. Eine Dokumentation, was man denn überhaupt alles mit dem neuen Scope anfangen kann, fehlte komplett bzw. konnte in den Tiefen des Internets nicht gefunden werden. Dass wir nicht alleine mit derartigen Problemen waren, konnten wir anhand einiger Forumsbeiträge und Fragen&Antwort-Seiten erkennen.

Die Entwicklung der Anwendung schritt voran und wir merkten schnell, dass es auch im JEE-Bereich wichtig ist, sich über die eingesetzten Technologien zu informieren. So wurde die im JSF-Bereich verwendete Annotation `@ManagedBean` durch die CDI-Annotation `@Named` überflüssig. Bei diesem Punkt merkten wir, dass wir bei den Beispielen ganz genau hinschauen müssen und dass ein Beispiel zu JSF 2 noch lange nicht JEE6-konform sein muss. Dass dies an der späten Finalisierung der CDI-Implementierung lag, wurde erst durch einen JSF 2 Vortrag der letztjährigen DOAG deutlich.

Neben Beispielen und Dokumentation benötigt man im Open Source Umfeld doch ab und zu den Quellcode der eingesetzten Software. JEE6 bringt eine Vielzahl an Komponenten mit, die vom Application Server verwaltet werden. So wird die komplette EJB- und CDI-Verwaltung vom Container gehandhabt. Exceptions, die vom Container geworfen werden, müssen manchmal wohl oder übel per Debugging untersucht werden. Hierbei erkannten wir eine Sache sehr schnell: mit JEE6 müssen wir zwar erheblich weniger Anwendungs-Code schreiben, jedoch sind dafür auch die Ursachen von Exceptions schwerer zu lokalisieren. Zudem waren die Source-Dateien für den GlassFish-Server der Version 3.1 relativ schwierig zu finden. Im öffentlichen Maven-Repository wurde zwar eine Source-JAR abgelegt, diese war allerdings komplett leer (genauso wie die JavaDoc-Datei). Warum das GlassFish-Team dies so gemacht hat, bleibt bis heute ein Rätsel…

Qualitätssicherung
Ein wichtiger Aspekt der Anwendungsentwicklung ist das Erstellen automatisierter Tests für die Qualitätssicherung. In Zeiten von Continuous Integration ist dieser Schritt fester Bestandteil des Release- und Entwicklungszyklus. Der im Projekt eingesetzte Hudson erlaubt zusammen mit dem Build-Tool Maven eine einfache Konfiguration des Build-Prozesses, so dass die Tests bei jedem Neubau ausgeführt werden und bei Fehlschlag der komplette Build-Prozess entsprechend fehlschlägt.

Die Wichtigkeit von automatisierten Tests dürfte wahrscheinlich jedem klar sein, jedoch stellte man sich bei dem JEE6-Projekt sehr schnell die Frage nach dem „Wie". Hierbei muss man sich erst einmal im Klaren darüber sein, welche Teile der Anwendung man eigentlich automatisiert testen kann (und will). Vergangene Projekte haben gezeigt, dass eine Verknüpfung von Oberflächentests und Backend-Tests durchaus sinnvoll aber auch aufwändig ist. Ziel war es deshalb erst einmal Tests für einzelne Anwendungskomponenten zu erstellen.

Da wir in fast allen Teilen der Anwendung regen Gebrauch von CDI und EJB gemacht oder geplant hatten, kamen wir schnell zu der Grenze der Container-verwalteten Ressourcen. Hier hatten wir nun die Wahl, ob wir eine sogenannte „embedded" Version des Application Server verwenden wollen (bei GlassFish: Embedded GlassFish) oder ob wir mittels Mock-Frameworks eigene Anwendungskontexte für den Test bereitstellen. Leider war der Einsatz des Embedded GlassFish nicht so einfach, wie man es sich im Team vorgestellt hatte. Die Anwendung, die auf dem GlassFish Application Server ohne Probleme deployt werden konnte, konnte nicht auf dem Embedded GlassFish ausgeführt werden. Erste Anpassungen führten dann zu einem positiven Ergebnis, jedoch gab es erneut Probleme, als wir die Anwendung in verschiedene Unter-Projekte aufteilen mussten. Dabei wurden Remote-EJBs aus einem anderen Deployment verwendet. Dies konnte im Embedded GlassFish nicht analog zum GlassFish Application Server nachgezogen werden. Der Aufwand zur Erstellung automatisierter Tests für die Container-verwalteten Komponenten wurde zu hoch in Relation zur Entwicklung dieser Komponenten. Hierbei fehlten leider auch wieder brauchbare Dokumentation und Beispiele.

Wo findet man Hilfe?
Eine große Hilfe bei der Entwicklung war der bereits erwähnte Besuch von Fachkonferenzen, bei denen JEE6-Features vorgestellt wurden und dazugehörige Workshops, in denen Beispiel-Anwendungen entwickelt wurden. Zudem waren Blogs bei „schlecht" dokumentierten Themen eine gute Anlaufstelle. Erwähnenswert sind hierbei die Beiträge zur neuen EJB-Annotation `@Singleton`. Die korrekte Verwendung dieser Annotation wurde nur auf Blog-Seiten beschrieben (inzwischen findet man das Thema auch im Oracle Java EE 6 Tutorial).

Die Entwicklung der Anwendung zeigte trotz aller oben aufgeführten negativen Punkte eines: die Entwicklung einer JEE6-Anwendung ist für den Entwickler deutlich angenehmer als bei vorherigen Versionen. Vor allem das Hinzufügen von CDI und der neuen JSF-Version führen zu einer produktiveren Entwicklung in vielen Teilen einer JEE-Anwendung.

Fallstricke
Die Anwendung wurde Ende 2011 nach einigen Testreleases produktiv genommen und ist seitdem kontinuierlich weiterentwickelt worden.

Im laufenden Betrieb zeigten sich nach einiger Zeit, dass ein erfolgreiches Deployment keine funktionsfähige Anwendung garantiert. Zudem stellte sich heraus, dass es doch eine nicht unerhebliche Anzahl an Bugs in den verschiedenen JEE6-Komponenten des ApplicationServers

gibt. Nachfolgend werden einige Fallstricke aufgeführt, die bei der Entwicklung einer JEE6-Anwendung berücksichtigt werden sollte.

Falscher CDI-Scope
Ein Problem war die Verwendung eines „falschen“ Scopes für die `@Produces`-Methode einer Konfigurationsklasse.

Die KonfigurationsBean ist eine Singleton-EJB, die mittels der `@Produces`-Annotation eine Konfiguration für eine JMS-Verbindung zur Verfügung stellt.

```
@Singleton
@Startup
public class KonfigurationsBean {
...

  @Produces
  @SessionScoped
  public JmsObjectWrapper getJmsObjectWrapper() {
    initRefreshJmsObjectWrapper();
    return jmsObjectWrapper;
  }
}
```

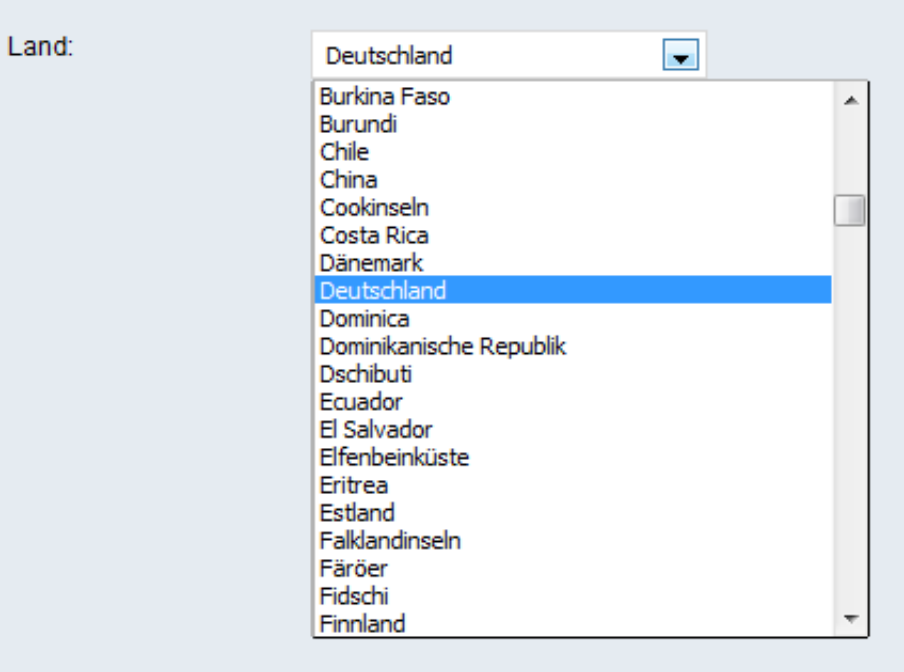
In der Consumer-Klasse wird das Konfigurations-Objekt verwendet. Hier wird mittels der CDI-Annotation `@Inject` das Objekt injiziert.

```
@Stateless
public class Consumer implements Serializable {

  @Inject
  JmsObjectWrapper jmsObjectWrapper;

...
}
```

Jetzt hatten wir allerdings das Problem, dass wir bei der @Produces-Methode einen Scope verwendeten, der im Consumer nicht verwendet werden kann. Dies hatten wir erst durch eine NullPointerException zur Laufzeit gemerkt.

CDI und JSF 2
Mit JSF 2 ist es möglich beliebige Objekte in Listenwerten zu verwenden, indem man einen passenden Converter dazu schreibt, der die Konvertierung String/Objekt realisiert. In älteren Versionen konnten immer nur Strings verwendet werden oder man verwendete ein zusätzliches Framework wie etwa JBoss Seam, das einem die Konvertierung erleichterte.

Als Beispiel wird hier ein Konverter für eine Liste an Staaten erzeugt. Die Daten kommen dabei aus der Datenbank.



Liste an Staaten im Webfrontend

Die Liste an Staaten besteht dabei aus JPA-Entities:

```
@Entity
@Immutable
@Table(name = "STAMMDATEN_STAATEN")
public class StaatenEntity implements Serializable {

  @Id
  @Column(name = "id", nullable = false)
  private String id;
...
}
```

Die Liste wird in der xhtml-Seite über ein Value-Binding verwendet:

```
<h:selectOneMenu converter="staatenConverter">
                 value="#{antragsteller.adresse.land}">
    <f:selectItems value="#{kundenDaten.landSelectItems}" var="land"
                   itemLabel="#{land.staatsname}" />
</h:selectOneMenu>
```

Der StaatenConverter muss hierbei einen String in eine Entity verwandeln und umgekehrt. Eindeutig identifizieren kann man die Entity über das „id“-Feld, das auch den Primärschlüssel in der Datenbank darstellt. Die Converter-Methode getAsString() sollte also die ID zurückliefern.

```
@FacesConverter(value = "staatenConverter", forClass =
StaatenEntity.class)
public class StaatenConverter implements Converter {

  @Override
```

```
  public String getAsString(FacesContext ctx, UIComponent c, Object v){
    if (v == null) {
      return "";
    }
    return ((StaatenEntity) v).getId();
  }
...
}
```

Bei der Umwandlung von String zu einem StaatenEntity-Objekt, muss man sich überlegen, wie man das passende Objekt zu dem gegebenen String erhält. Da man die ID hat, kann man über einen passenden Service sich das Objekt aus der Datenbank auslesen lassen. Jetzt könnte man auf die Idee kommen per CDI den Service in den FacesConverter zu injizieren und in der getAsObject()-Methode entsprechend zu verwenden.

```
@Inject IStaatenDaoService staatenDAOService;

@Override
public Object getAsObject(FacesContext ctx, UIComponent c, String v) {
  if (!StringUtils.isEmpty(v)) {
    return staatenDAOService.findEntityById(v);
  }
  return null;
}
```

Leider funktioniert CDI innerhalb eines FacesConverter nicht. Auch die Nutzung der EJB-Annotation `@EJB` nützt hier nichts.

Zur Lösung des Problems kann man gänzlich auf die Annotation `@FacesConverter` verzichten und stattdessen aus dem Converter eine CDI-Bean per `@Named` machen. Eine alternative Lösung ist die Verwendung eines JNDI-Lookups innerhalb der getAsObject()-Methode.

Conversation-Scope
Eine Neuerung in JEE6 ist der Conversation-Scope. Der Conversation-Scope ist kürzer als der Session-Scope aber länger als der Request-Scope. Theoretisch kann man sämtliche JSF-ManagedBeans in den Session-Scope legen. Dies zieht allerdings einige Nachteile mit sich, so wird die Session zu groß, die Thread-Sicherheit ist bei Verwendung mehrerer Tabs nicht gewährleistet und die Anwendungs-Performance leidet unter den großen Datenmengen.

Der Conversation-Scope ist vor allem für die Abbildung eines Seitenflusses/Wizard gedacht. Die Conversation wird mittels eines Request-Parameters identifiziert und kann programmatisch gestartet und gestoppt werden. Mit JSF 1.x war die Nutzung des Conversation-Scopes über Frameworks wie beispielsweise JBoss Seam oder Apache MyFaces Orchestra möglich. Mit JEE6 wurde nun ein neuer Conversation-Scope in den Standard integriert.

Bei der Verwendung des Conversation-Scopes merkten wir aber schnell, dass die Idee zwar gut ist, die Umsetzung jedoch völlig unzureichend. So hat der JEE6 Conversation-Scope im Gegensatz zu den Conversation-Scopes anderer Frameworks folgende Nachteile:

- Es gibt keine „nested Conversations“, die Kind-Conversations darstellen. Kind-Conversations sind äußerst hilfreich, wenn man komplexere Seitenflüsse realisieren will.
- Es gibt keine Möglichkeit die Conversation an die Navigation anzubinden, wie es bei JBoss Seam der Fall ist. Dadurch wird das Starten und Stoppen der Conversation umständlich.

- Man muss manuell prüfen, ob die Conversation begonnen wurde.

Der Conversation-Scope von JEE6 ist in seiner jetzigen Form nahezu unbrauchbar. Sinnvolle Alternativen werden von anderen Frameworks angeboten, die durch die Nutzung von Custom-Scopes einen eigenen Conversation-Scope bereitstellen, der in ein JEE6-Projekt integriert werden kann.

JSF 2 und Ajax

Mit der Version 2 wurde die Ajax-Unterstützung in den JSF-Standard übernommen. Musste man früher noch zusätzliche Abhängigkeiten für Ajax-Funktionalität einbinden, so kann man nun über das Standard-Tag `f:ajax` die JSF-Komponenten mit Ajax anreichern.

Eine Standardaufgabe für die Ajax verwendet wird, ist die Neudarstellung einzelner Seitenfragmente. Dieses „Neurendern“ wird durch das Attribut `render` realisiert. Hierbei gibt man die Komponente(n) an, die man nach dem Auslösen des JavaScript-Events neu darstellen will.

```
<h:inputText id="betrag" value="#{konto.betrag}">
  <f:ajax event="change" execute="@this" render="@this otherComponent"/>
</h:inputText>
```

In diesem Beispiel wird neben dem Eingabefeld „betrag“ zusätzlich eine Komponente mit der ID „otherComponent“ neu gerendet, sobald das JavaScript-Event „onchange“ ausgelöst wird.

Hierbei gibt es nun eine Besonderheit, auf die man achten muss. Es ist nicht möglich eine Komponente neu darzustellen, die aktuell nicht im Komponentenbaum dargestellt wird. D.h. sobald die Komponente „otherComponent“ eine „rendered“-Bedingung hat (und initial nicht dargestellt wird), muss man ein Vaterelement dieser Komponente neu rendern, das bereits im Komponentenbaum enthalten ist.

Das folgende Beispiel funktioniert also nicht. Die Komponente „otherComponent“ wird nicht korrekt neu dargestellt, sofern die Rendered-Bedingung initial „false“ liefert.

```
<h:inputText id="betrag" value="#{konto.betrag}">
  <f:ajax event="change" execute="@this" render="@this otherComponent"/>
</h:inputText>

<h:outputText id="otherComponent" value="{bean.text}"
         rendered="#{bean.otherComponentRendered} />
```

Das Problem behebt man, indem die Komponente eine zusätzliche Vater-Komponente erhält. Nun wird die Vater-Komponente neu dargestellt:

```
<h:inputText id="betrag" value="#{konto.betrag}">
  <f:ajax event="change" execute="@this" render="@this parent"/>
</h:inputText>


<h:panelGroup id="parent">
  <h:outputText id="otherComponent" value="{bean.text}"
           rendered="#{bean.otherComponentRendered} />
</h:panelGroup>
```

Migration von GlassFish auf JBoss
Zu Beginn des Projektes gab es eigentlich nur einen möglichen Application Server, den wir zur Entwicklung und zum Ausliefern der JEE6-Anwendung nutzen konnten. Der GlassFish Application Server wurde sowohl für die Entwicklung als auch für den laufenden Betrieb der Anwendung genutzt. Allerdings kam seitens des Auftraggebers schon recht früh der Wunsch auf, die Anwendung auf dem JBoss Application Server zu installieren. Da dies zu Projektstart nicht möglich war (der JBoss Application Server war zu dem Zeitpunkt noch nicht JEE6 zertifiziert), wurde die Migration nach hinten verschoben. Ein nicht unerhebliches Problem, wie sich am Ende herausstellte…

Der JBoss Application Server ist seit der Version 7.1 Java EE6 zertifiziert. Die Implementierungen der einzelnen JEE6 Bestandteile sind größtenteils deckungsgleich mit den Implementierungen, die der GlassFish Application Server nutzt.

Erste Test-Deployments ließen schnell die Hoffnung auf eine problemlose und schnelle Migration schwinden:

- Der JBoss Application Server hat eine striktere Prüfung der JEE6-Konformität. So wurde in einer EJB das Attribut „mappedName“ verwendet. Das Attribut ist in EJB 3.1 deprecated. Der GlassFish Application Server hatte damit keine Probleme, der JBoss AS verweigerte das Deployment.
- Die JSF-Anwendung verweigerte schon auf der ersten Seite ihren Dienst. Kurioserweise funktionierte die Anwendung mit Version 7.1.0, jedoch nicht mit 7.1.1. Eine Fehleranalyse zeigte, dass im JBoss in Version 7.1.1 eine Änderung an der Request-Parameterverarbeitung vorgenommen wurde. Im Endeffekt konnten wir damit die JSF-Anwendung im aktuellsten JBoss AS nicht verwenden. Hier half nur ein „Downgrade“ auf Version 7.1.0.
- Der JNDI-Lookup wird vom JBoss AS anders realisiert als im GlassFish. Dies hatte einige Änderungen an der Nutzung der `@Inject` und `@EJB` Annotationen zur Folge. Der GlassFish war hier weit weniger restriktiv.
- Die Server sind unterschiedlich „informationsfreudig“, wenn eine Exception im Container geworfen wird. Die eigentliche Fehlerursache ist im JBoss meist nur sehr schwer zu erkennen. Durch einige schwerwiegende Bugs in der Version 7.1.1 wurde die Suche zusätzlich erschwert.

Die Migration der JEE6-Anwendung war zum Zeitpunkt dieses Erfahrungsberichts noch nicht abgeschlossen. Man kann allerdings zusammenfassend sagen, dass man sich von Anfang an für einen bestimmten Application Server entscheiden sollte oder mit doch recht unterschiedlichen Verhaltensweisen zurechtkommen muss.

Fazit
Eine neue Technologie zu erlernen gehört zum Beruf des Entwicklers und ist für die Entwicklung einer modernen Unternehmensanwendung unerlässlich. Bei einem Fazit sollte man bewerten, ob man durch die neue Technologie produktiver geworden ist, die Komplexität beherrschbar ist und ob sich Probleme mit angemessenem Aufwand lösen lassen.

Zusätzlich kann man natürlich noch weitere Punkte aufführen, die meist einen subjektiven Charakter haben. So gibt es beispielsweise die berühmte „Annotation Hell“ – viele XML-Konfigurationen sind inzwischen überflüssig geworden aber werden manchmal durch eine riesige Anzahl an Annotationen ersetzt. Die Bewertung, welche Variante man „schlimmer“ findet, bleibt jedem selbst überlassen.

Positiv

- Die Dokumentation war zwar Anfang/Mitte 2011 mehr als dürftig, inzwischen hat sich dies aber verbessert. Es gibt zahlreiche Literatur und Blogeinträge, die JEE6-Neuerungen behandeln und Best-Practices aufzeigen.
- Die Entwicklung einer „leichtgewichtigen" Anwendungsarchitektur ist dank JEE6 sehr einfach. Man kommt schnell zu ersten Ergebnissen, wenn man weiß, wie man mit den neuen Features CDI, JSF 2 und EJB 3.1 umgehen muss.
- Die entwickelte Anwendung läuft performant und stabil. Memory-Leaks sind bisher nicht aufgetreten.
- Die Entwicklungszeit für einen fachlichen Use Case hat sich in Bezug auf JSF reduziert bzw. ist vergleichbar zu einer Entwicklung mit der alten Version 1.2, die zusammen mit einem Zusatzframework eingesetzt wurde.

Negativ

- Automatisierte Tests für Container-Managed Komponenten waren für uns nur sehr schwer zum Laufen zu bringen.
- Es gibt Features, die zwar vom Konzept her gut sind, bei denen jedoch die Umsetzung für den realen Entwicklungsalltag irrelevant ist. So kann man sich den Conversation-Scope von JEE6 in seiner Standard-Form schenken. Hier sollte man eher auf die Lösungen anderer Frameworks zurückgreifen.
- Für einige Features gab es nur unzureichende Beschreibungen und Beispiele. Hier wurde scheinbar etwas am „Marketing" der neuen JEE6-Features gespart.
- Die Fehleranalyse bei Exceptions, die vom Container geworfen werden, ist oft nur recht schwer durchzuführen. So wurden beispielsweise bei einem Redeployment vom GlassFish Exceptions geworfen, die nach einem Server-Neustart jedoch nicht mehr auftraten. Mit den Fehlermeldungen in den Logdateien konnten wir meistens nur wenig anfangen.

Da mein persönlicher technologischer Schwerpunkt die Entwicklung von Webanwendungen mit JSF ist, möchte ich hier noch ein gesondertes Fazit zu JSF 2 geben:

- Es gibt so gut wie keine Änderungen im Konzept. JSF ist immer noch eine viel zu bunte Mischung aus Java/EL/xhtml-Syntax. Für Neueinsteiger ist diese Kombination nicht immer verständlich und stellt eine nicht zu unterschätzende Lernhürde dar. Der Fokus liegt scheinbar generell auf der Weiterführung des eingeschlagenen Weges, auch wenn manche Konzepte heutzutage im Vergleich mit anderen Frameworks zur Erstellung von Webanwendungen mehr als altbacken und überladen daher kommen.
- Immer noch recht hoher Aufwand für „wenig" Anwendung: Der Code, den man für relativ wenig Oberflächenfunktionalität schreiben muss, ist meistens umfangreich.
- Es gibt immer noch eine starke Abhängigkeit von guten Komponentenbibliotheken und Frameworks. Ziel des Standards ist beispielsweise nicht die Bereitstellung einer umfangreichen Komponentenbibliothek und eines Layout-Konzepts. Deshalb muss man hier auf ausgereifte Komponentenbibliotheken von Open Source oder kommerziellen Anbietern zurückgreifen.
  Dieser Punkt kann bei gewissen Anforderungen eine richtige Qual sein, wenn man sich auf die Funktionalität dieser Bibliotheken verlässt. ICEfaces war im Nachhinein gesehen die absolut falsche Wahl für das Projekt. Die eingesetzte Version 2 strotzte nur so vor Bugs und funktionierte im Browser des Kunden nicht einwandfrei.
- In vielen Punkten kommt JSF 2 nicht an die Mächtigkeit von JSF 1.2 und dem JBoss Seam Framework heran. So fehlt z.B. die globale Unterstützung von EL-Ausdrücken, eine vergleichbar mächtige EL, komplexe Navigationsregeln und Security-Features.

- Es gibt immer noch zu wenige Beispiele für bestimmte Neuerungen. So finden sich zwar zu Hauf Beispiele für Composite Components – wie man jedoch z.B. optionale Attribute umsetzen kann, wird nirgends erklärt. Ein weiteres Beispiel ist die neue Multi-Field Validierung, die eigentlich sehr praktisch ist, jedoch nur am Rande erwähnt wird.
- Die Dokumentation ist in vielen Teilen unzureichend, missverständlich oder nicht mehr aktuell. Häufig kann man nicht mehr unterscheiden, ob die Dokumentation für JSF 1.x oder 2 gilt.
- Die Entwicklung von JSF 2 schreitet scheinbar schneller voran als die Entwicklung der alten Version. So gibt es bereits gute Ideen für die Version 2.2 und kontinuierliche Fehlerbehebungen für die Version 2.1.x

Abschließend kann man sagen, dass der neue Java EE 6 Standard ein brauchbares Mittel zur Entwicklung einer komplexen Unternehmensanwendung ist. Sinnvolle Neuerungen wurden eingeführt und die Entwicklung geht in vielen Teilen einfacher vonstatten als zuvor. Die Arbeit der Expert Group und der Entwickler von JEE6 ist auf jeden Fall zu loben, auch die Weiterentwicklung findet in einem rasanten Tempo statt. Allerdings sollte man sich vor der Verwendung von JEE6 über die Möglichkeiten und Grenzen des Frameworks informieren. Dies ist ein nicht unerheblicher Aufwand, da die Aggregierung der Informationen doch einiges an Zeit in Anspruch nimmt. Ich bin jedoch zuversichtlich, dass sich der Standard durchsetzen wird und es viele interessante und auch erfolgreiche Projekte mit JEE6 geben wird.

Kontaktadresse:
Moritz Lippe
syntegris information solutions GmbH
Hermannstraße 54-56
D-63263 Neu-Isenburg

Telefon: +49 (0) 6102-29 86 68
Fax: +49 (0) 6102-55 88 06
E-Mail moritz.lippe@syntegris.de
Internet: http://www.syntegris.de