Enterprise JavaBeans 3.1

Am 10. Dezember 2009 wurde die finale Version 6 der Java-EE-Spezifikation freigegeben. Einen ersten Überblick können Sie sich unter [1] machen. Java EE 6 enthält unter vielen Neuerungen auch ein neues Release der Enterprise-JavaBeans-Spezifikation. Mit einem Inkrement in der Nachkommastelle der Versionsnummer handelt es sich hierbei zwar nur um ein Minor-Release, das aber dennoch mit vielen interessanten Ergänzungen gegenüber der bisherigen Version aufwartet.

von Dirk Weil

Enterprise JavaBeans haftet der Malus an, starr, kompliziert und aufwändig zu sein. Diese Einschätzung stammt aus den Zeiten bis EJB 2.x, in denen die Kritik tatsächlich berechtigt war. Und wenn der Ruf einmal zerstört ist… Vielleicht wäre es angeraten, EJBs ab der Version 3.x einen anderen Namen zu geben, um sie von den psychologischen Altlasten zu befreien. EJBs der aktuellen Spezifikation sind nämlich alles andere als kompliziert oder schwergewichtig. Wir wollen uns in diesem Artikel den Neuerungen der Version 3.1 widmen. Die Grundlagen der bislang aktuellen Version 3.0 sind im Kasten „Was bisher geschah: EJB 3.0 im Überblick“ zusammengefasst. Weitere Details finden Sie auch in Adam Biens Artikel zum Thema aus dem vergangenen Jahr [2].

Bohnen ohne Interfaces

EJB 3.1 verfolgt die mit 3.0 eingeschlagene Richtung der einfachen serverseitigen Programmierung weiter, komplettiert den Themenbereich und erfüllt teilweise lange gehegte Wünsche. So stellt man häufig in EJB-Projekten fest, dass das im Allgemeinen richtige und wichtige Vorgehen, „gegen Interfaces“ zu programmieren, zu einer 1:1-Entsprechung von EJB-Klassen und Zugriffsinterfaces degeneriert. Zumindest im lokalen Umfeld, d. h. für Zugriffe innerhalb einer JVM, bringen die Zugriffsinterfaces kaum Vorteile. Seit EJB 3.1 kann man sie weglassen und damit immerhin 50 % der Programmartefakte einsparen. Das Beispiel zeigt dies anhand eines Codeausschnitts eines unserer jüngsten Projekte, bei dem Datenbankzugriffe mittels DAOs erfolgen, die wir als Stateless Session Beans realisiert haben:

```
// CarrierDaoBean.java
@Stateless
@LocalBean
public class CarrierDaoBean
{
 @PersistenceContext
 private EntityManager entityManager;

 public void insert(Carrier entity)
 {
  this.entityManager.persist(entity);
 }
…
}

// CarrierServiceBean.java
@Stateless
public class CarrierServiceBean implements
                                        CarrierService
{
 @EJB
 private CarrierDaoBean carrierDao;
…
```

Die Annotation *@LocalBean* fordert die Bereitstellung der so genannten No-Interface View für die EJB explizit an. Das geschieht allerdings automatisch, wenn die Bean kein Clientinterface besitzt, sodass *@LocalBean* im Beispiel nicht notwendig gewesen wäre. Der Verzicht auf das Local Interface stellt sicher eine sinnvolle, pragmatische Vereinfachung in Fällen wie dem geschilderten dar, ist aber im Hinblick auf die „Sauberkeit“ des Programmcodes durchaus kritisch zu betrachten, wird dadurch doch die Komponenteneigenschaft von EJBs verwässert und dem Entwickler suggeriert, dass sich Session Beans auch zur Laufzeit wie normale Klassen verhalten. Tatsächlich generiert der Container auch für No-Interface View Beans ein Hüllobjekt, das z. B. den Pre- und Post-Invocation-Code implementiert. Im gezeigten Beispiel ist daher das in das private Feld *carrierDao* injizierte Objekt nicht vom Typ *CarrierDaoBean*, sondern von einer davon abgeleiteten Klasse. Sie zweifeln vielleicht, ob No-Interface View Beans überhaupt sinnvoll sind und nicht gänzlich durch einfache Java-Klassen ersetzt werden könnten. Dann könnte man aber nicht mehr die gezeigten Dienste

> **Artikelserie: Java EE 6 ausgepackt**
>
> **Teil 1: EJB 3.1**
> Teil 2: JPA 2.0

des EJB-Containers wie Dependency Injection oder Transaktionssteuerung nutzen. Diese explizit auszuprogrammieren, wäre zweifellos aufwändiger und fehlerträchtiger.

Einzelgänger

Eine weitere Wunscherfüllung ist die Einführung von Singletons. Mithilfe der Annotation *@Singleton* kann eine EJB programmiert werden, von der zur Laufzeit nur eine Instanz erzeugt wird. Dies war bisher nur mit Umwegen und Fallstricken über serverspezifische Deskriptorparameter möglich oder durch Umgehung des Standards und Nutzung des Java-SE-Singleton-Patterns. Über die Annotation *@Startup* kann erzwungen werden, dass die Singleton-Instanz bereits beim Anwendungsstart initialisiert wird. *@DependsOn* deklariert Abhängigkeiten zu weiteren Singletons und sorgt insbesondere für die richtige Initialisierungsreihenfolge:

```
@Singleton
@Startup
@DependsOn("ApplicationPropertyServiceBean")
public class SystemServiceBean
{
 @Lock(LockType.READ)
```

Was bisher geschah: EJB 3.0 im Überblick

Bis zur Version 2.1 waren EJBs sehr stark durch den so genannten Container Contract geprägt, d. h. Schnittstellen und Methoden, die für die Kommunikation zwischen Bean und Container notwendig sind: Interfaces und Implementierungsklassen mussten von einer vorgegebenen Basis abgeleitet werden, diverse Callback-Methoden waren zu implementieren und jede Remote-Methode musste technische Exceptions deklarieren. Zudem musste jede EJB im XML-basierten Deployment Descriptor mit nicht wenigen Metadaten versehen werden. Diese Rahmenbedingungen waren zumindest lästig und führten zu einem höheren Entwicklungsaufwand mit diversen Fehlermöglichkeiten, die sinnvollerweise nur mit Einsatz von Tools wie XDoclet gemildert werden konnten.

POJOs

Für die Version 3.0 ist das Prinzip radikal umgebaut worden: EJBs sind einfache Java-Klassen (POJOs) bzw. Interfaces. Vorgegebene Basisklassen, Callback-Methoden und technische Exceptions sind entfallen. Für die notwendigen Metadaten gilt das Prinzip „Convention over Configuration“, d. h. es gibt Vorgabewerte, die nur bei Bedarf explizit überschrieben werden müssen. Nutzt man dazu Annotationen, kann auf XML-Deskriptoren komplett verzichtet werden:

```
// SomeServiceBean.java
@Stateless
public class SomeServiceBean
                    implements SomeService
{
 @Override
 public void doSomething(String s)
 {
 …
 }
}

// SomeService.java
@Remote
public interface SomeService
{
 public void doSomething(String s);
}
```

Der Codeausschnitt zeigt eine Stateless Session Bean mit einem Remote-Interface für den Zugriff aus einem Client. Kein XML-Deskriptor, keine technischen Interfaces, keine technischen Exceptions wie in der Version 2.x – einfacher geht es kaum.

Dependency Injection

EJBs stehen häufig nicht alleine, sondern verrichten ihre Arbeit durch Zugriff auf andere EJBs und weitere Ressourcen. Bis zur Version 2.x mussten diese Abhängigkeiten im Deployment Descriptor beschrieben und zur Laufzeit durch explizite Aufrufe des Namensdienstes JNDI aufgelöst werden. Ab 3.0 kann dies durch Dependency Injection geschehen: Eine passende Instanzvariable (oder alternativ eine Setter-Methode) wird mit der Annotation *@EJB* für EJBs, *@Ressource* für allgemeine Ressourcen oder speziellen Annotationen wie *@PersistenceContext* ausgezeichnet. Zur Laufzeit injiziert der Container den gewünschten Wert ganz automatisch:

```
// AnotherBean.java
@Stateless
public class AnotherBean
{
 @EJB
 private SomeService someService;
 …
}
```

Aspekte

Mit Blick auf andere Applikationsframeworks – allen voran Spring – könnte man vermuten, dass für EJB 3.0 einige Rosinen aus dem Kuchen gepickt wurden: POJOs und Dependency Injection sind hervorstechende Merkmale dieser Plattformen. Da reiht sich die aspektorientierte Programmierung als weitere Rosine ein, wobei hier allerdings nur Aspekte auf Ebene der EJB-Methoden realisiert werden, was für die Implementierung von Querschnittsaufgaben auch vollkommen ausreicht. EJB-Interceptors sind wiederum einfache Java-Klassen, mit denen EJB-Methodenaufrufe „umhüllt“ werden können:

```
// TraceCallInterceptor.java
public class TraceCallInterceptor
{
 @AroundInvoke
 protected Object trace
              (InvocationContext invocationCtx)
 {
  … // Do something before method call
  Object result = invocationCtx.
                   proceed(); // Call method
  … // Do something after method call
  return result;
 }
}
```

Die Annotation *@Interceptors(TraceCallInterceptor.class)* verknüpft den Interceptor mit einer EJB-Methode oder – bei Nutzung auf Klassenebene – mit allen Methoden einer EJB.
Security- und Transaktionskonfigurationen können nun ebenfalls mithilfe von Annotationen vorgenommen werden *(@TransactionManagement, @TransactionAttribute, @RolesAllowed, @PermitAll, @DenyAll, @RunAs, …).*
Der XML-Deskriptor ist damit obsolet geworden. Er darf aber durchaus weiter verwendet werden, wenn Annotationen nicht genutzt werden sollen oder können, oder um deren Einstellungen nochmals zu modifizieren.

```
  public void foo() { … }

  @Lock(LockType.WRITE)
  @AccessTimeout(value = 1000, unit =
                              TimeUnit.MILLISECONDS)
  public void bar() { … }
  …
}
```

Eine Singleton Bean entspricht grob einer Stateful Bean, allerdings mit dem großen konzeptionellen Unterschied, dass Singletons von mehreren User-Threads konkurrierend aufgerufen werden können. Die Serialisierung der Zugriffe übernimmt per Default der Container. Zur Steuerung können Methoden mit *@Lock(LockType.READ)* und *@Lock(LockType.WRITE)* markiert werden, um einen Shared bzw. Exclusive Lock auf dem Singleton während der Methodenausführung zu erzeugen. Die Annotation kann auch auf Klassenebene verwendet werden und gilt dann für alle Methoden. Voreingestellt ist Exclusive Lock. Blockierungen führen naturgemäß zu Wartezeiten. Diese können mit der Annotation *@AccessTimeout* begrenzt werden: Bei Überschreitung der darin spezifizierten Wartezeit wird der Methodenaufruf mit dem Auswurf einer *ConcurrentAccessTimeoutException* abgebrochen.

Alternativ zur beschriebenen Container Managed Concurrency kann der Entwickler sich mittels *synchronized*, *volatile* und analogen Mitteln auch selbst um die Serialisierung der konkurrierenden Zugriffe kümmern. Dazu muss die Klasse mit der Annotation *@ConcurrencyManagement(ConcurrencyManagementType.BEAN)* versehen werden. Singleton-EJBs standardisieren, was auch mit klassischen Mitteln hätte programmiert werden können (dann allerdings mit Fallstricken wie Classloading etc.) – nämlich die Erzeugung von Einzelinstanzen pro Java-VM. In einem Cluster ist also Vorsicht geboten: Hier existieren die Objekte dann mehrfach.

Terminsachen

In vielen Anwendungen gibt es einen Bedarf für zeitgesteuerte Tätigkeiten, sei es zur Timeout-Überwachung von externen Ereignissen oder für regelmäßige, termingesteuerte Aufgaben. Dafür existiert seit EJB 2.1 ein Timer-Service, der aber bislang nur recht rudimentär parametrisiert werden konnte, insbesondere im Bereich der Terminsteuerung. Neidische Blicke auf Uraltwerkzeuge wie *cron* sind jetzt aber nicht mehr nötig, denn EJB 3.1 erweitert den Timer-Service u. a. um sehr umfangreiche Scheduling-Möglichkeiten. Eine Bean kann nun ein oder mehrere Methoden enthalten, die mit *@Schedule* annotiert sind. Über die Parameter der Annotation können die Aufruftermine sehr flexibel spezifiziert werden, wie die folgenden Beispiele zeigen:

```
@Schedule(dayOfWeek="Mon")
public void jedenMontag() { … }

@Schedule(minute="15", hour="3",
                        dayOfWeek="Mon-Fri")
public void jedenWochentagUm3Uhr15() { … }

@Schedule(minute="*/5", hour="*")
public void alle5Minuten() { … }

@Schedule(dayOfMonth="Last")
public void amMonatsletzten() { … }

@Schedule(dayOfMonth="2nd Fri", hour="20")
public void jeden2tenFreitagUm20Uhr() { … }
```

Timer können – per Parameter der Annotation oder über das Timer-API – persistent oder nicht persistent gestaltet werden. Persistente Timer bleiben über einen Shutdown der Applikation hinaus gültig. Während der Downtime „verpasste“ Timer-Events werden automatisch nachgeholt, wenn die Anwendung wieder gestartet wird.

Wenn’s mal ein bisschen länger dauert

Geschäftslogik kann so umfangreich sein, dass ihre Abarbeitung mehr Zeit benötigt, als man den Aufrufer normalerweise blockieren möchte. Da EJB-Aufrufe bislang ausschließlich synchron waren, blieb in solchen Fällen nur ein Ausweichen auf asynchrone Kommunikation mit JMS o. ä. In EJB 3.1 ist nun kein Umweg mehr nötig. Langlaufende Methoden können mit *@Asynchronous* annotiert werden:

```
@Asynchronous
public void archiviereBelege(int jahr) { … }
```

Der Aufrufer der Methode erhält sofort die Kontrolle zurück, während die Geschäftslogik der Bean parallel abgearbeitet wird. Die Annotation kann für die gewünschte Methode der EJB-Klasse verwendet werden, oder aber bei der Deklaration der Methode im Remote- oder Local-Interface. Somit kann eine Methode z. B. für einen Remote-Aufruf asynchron ablaufen, während sie lokal synchron aufgerufen wird. Meistens liefern asynchrone Methoden wie im Beispiel keinen Ergebniswert, da sich der Aufrufer nach dem Anstoß der Methode nicht mehr für ihre Ausführung interessiert. Es ist jedoch auch möglich, einen Wert zurückzuliefern, und zwar in Form eines Futures. Für diesen Zweck enthält der Standard die Klasse *AsyncResult* als konkrete Implementierung des Interfaces *Future*:

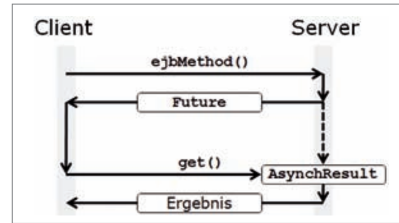

Abb. 1: Asynchroner Methodenaufruf

```
@Asynchronous
public Future<Integer> getAnswerToQuestionAbout
                        LifeUniverseAndEverything()
{
  … // Time passes (7.5 million years …?)
  return new AsynchResult<Integer>(42);
}
```

Der Aufrufer der EJB-Methode erhält wiederum sofort die Kontrolle zurück und kann das Ergebnis der parallelen Berechnung zu einem späteren Zeitpunkt mithilfe der Methode *get* des *Future*-Werts abholen.

Portable JNDI-Namen

Der Zugriff auf EJBs lässt sich, wie zuvor gezeigt, bequem über Dependency Injection herstellen. Dies ist in allen gemanagten Anwendungsteilen möglich, also z. B. in anderen EJBs, in Servlets oder auch in Application Clients. Andere Anwendungsteile wie „normale“ Clients können auf EJBs per Lookup im Namenssystem zugreifen. Leider waren die automatisch vergebenen JNDI-Namen bis zur Version 2.1 nicht standardisiert und nur in providerabhängigen Deskriptoren parametrisierbar. EJB 3.1 führt hier portable Namen

ein. Der globale Namensraum *java:global* enthält für jedes Remote und Local Interface einer EJB einen Eintrag der Form *java:global[/<app-name>]/<module-name>/<bean-name>!<fully-qualified-interface-name>*. Dabei bedeuten

- *<app-name>*: Applikationsname. Kann im Deployment-Deskriptor eines EAR-Files gesetzt werden und hat als Default den Namen des EAR-Files ohne Endung. Entfällt, wenn die EJB nicht in ein EAR-File gepackt wird.
- *<module-name>*: Modulname. Kann im Deployment-Deskriptor der EJB gesetzt werden und hat als Default den Namen des EJB-JAR-Files ohne Endung. Bei einem Deployment in einem WAR-File werden analog dessen Deskriptor und Name verwendet.
- *<bean-name>*: Name der EJB aus *@Stateless/@Stateful/@Singleton* bzw. aus ihrem Deployment-Deskriptor. Default ist der einfache Klassenname.
- *<fully-qualified-interface-name>*: Voll qualifizierter Name des Clientinterfaces. Für die No-Interface View wird hier entsprechend der Klassenname verwendet.

Hat eine EJB nur genau ein Clientinterface, wird dafür zusätzlich ein Eintrag unter dem Namen *java:global[/<app-name>]/<module-name>/<bean-name>* gemacht. Neben dem beschriebenen globalen Namensraum führt EJB 3.1 zwei weitere Namensräume ein:

- *java:app*: Namen relativ zur Anwendung. Entspricht dem globalen Namen ab dem Modulnamen und ist nur innerhalb einer Anwendung gültig.
- *java:module*: Namen relativ zu einem Modul (EJB-JAR bzw. WAR). Entspricht dem globalen Namen ab dem Bean-Namen und ist nur innerhalb eines Moduls gültig.

Päckchen packen

Das Standardpaketierungsformat für Enterprise-Applikationen bleibt weiterhin das EAR-File, in dem EJBs, Konnektoren, Webanwendungen und Application Clients als separate Deployment-Einheiten zu einem großen Paket zusammengefasst werden können. Wird die gesamte Mächtigkeit dieses Anwendungsformats nicht benötigt, z. B. weil nur eine Webanwendung mit einigen EJBs zusammengeführt werden soll, so kann ab EJB 3.1 (und Servlet 3.0) auf einfache WAR-Files ausgewichen werden, die die genutzten EJBs in ihrem *classes*- bzw. *lib*-Anteil mitbringen – WAR-Files als EAR Lite sozusagen. Wie zuvor die No-Interface Views, stellt die Möglichkeit der direkten Integration von EJBs in Webanwendungen eine pragmatische Vereinfachung dar, die von anderer Warte betrachtet auch Nachteile mit sich bringt. Die klare Trennung von Präsentation und Geschäftslogik wird hier aufgegeben – zumindest bei der Paketierung –, womit die Projektstruktur unübersichtlicher werden kann.

Embeddable Container

Automatische Softwaretests sind ab einer gewissen Anwendungsgröße unverzichtbar und demzufolge auch Bestandteil der meisten Vorgehensweisen zur Softwareentwicklung. Ein Test einer EJB in ihrer Umgebung benötigt einen entsprechenden Container. Ein Deployment in einen Server ist im Testumfeld aufwändig und problematisch. EJB 3.1 bietet daher einen so genannten Embeddable Container an, der z. B.innerhalb eines Unit-Tests instanziert werden kann und in dem die zu betrachteten EJBs testweise aufgerufen werden können:

```
EJBContainer ejbContainer =
                    EJBContainer.createEJBContainer();
Context jndiCtx = ejbContainer.getContext();

SomeService someService
= (SomeService) jndiCtx.lookup("java:global/
                                   classes/SomeService");

someService.doSomething();
```

Mit der Methode *createEJBContainer* wird ein „echter“ EJB-Container in der aktuellen JVM gestartet. Die im Classpath liegenden EJBs werden automatisch deployt. Im Beispiel wurden die EJBs nicht in ein EJB-JAR gepackt, sondern ausgepackt im Classpath-Verzeichnis *classes* belassen – daher der ungewöhnliche JNDI-Name. Wenn Sie dies mit GlassFish v3 ausprobieren wollen, müssen Sie die Bibliothek *glassfish-embedded-static-shell.jar* aus dem Verzeichnis *lib/embededded* in den Classpath aufnehmen.

Fazit und Ausblick

Die Neuerungen im EJB-Framework fallen erwartungsgemäß nicht allzu umfangreich aus – es ist ja schließlich auch „nur“ ein kleiner Versionssprung. Die mit 3.0 eingeschlagene Richtung wird konsequent weiter verfolgt und mit den gezeigten Features abgerundet. Dass keine strategischen Dinge hinzugekommen sind, ist durchaus positiv zu werten, zeigt sich damit doch, dass Enterprise JavaBeans die erwünschte Stabilität und Vollständigkeit erreicht haben. Auch wenn andere Frameworks mit größerer Flexibilität locken: Mehr braucht's eigentlich nicht, um produktiv serverseitige Geschäftslogik implementieren zu können. Die Persistenz von Java-Objekten war bis zur Version 2.1 Teil der EJB-Spezifikation. Seit Java EE 5 steht dafür JPA als EJB-Spin-Off zur Verfügung. Schon die bislang verfügbare Version 1.0 erfreut sich nicht zuletzt wegen der Provider Hibernate, TopLink/EclipseLink und OpenJPA einer wachsenden Beliebtheit. Im nächsten Artikel dieser Serie wollen wir uns anschauen, welche Änderungen die neue Version JPA 2.0 bringt.

Dirk Weil ist seit 1998 als Berater im Bereich Java tätig. Als Geschäftsführer der GEDOPLAN GmbH in Bielefeld [3] ist er für die Konzeption und Realisierung von Informationssystemen auf Basis von Java EE verantwortlich. Seine langjährige Erfahrung in der Entwicklung anspruchsvoller Unternehmenslösungen machen ihn zu einem kompetenten Ansprechpartner und anerkannten Experten auf dem Gebiet Java EE. Er ist Autor in Fachmagazinen, hält Vorträge und leitet Seminare und Workshops aus einem eigenen Java-Curriculum.

Links & Literatur

[1] http://it-republik.de/jaxenter/artikel/Java-EE-6-auf-einen-Blick-2759.html

[2] Adam Bien: „Die Wiederkehr der Einfachheit,“ in Java Magazin 7.09

[3] http://www.gedoplan.de