Enterprise

'Never a dull moment' in de wereld van Java. Op die regel is ook 2009 zeker geen uitzondering. Neem nu de overname van Sun Microsystems door Oracle. Eén van de ontwikkelingen die op 10 decmeber 2009 tot een hoogtepunt zou moeten komen, is Java EE6, de nieuwste release van de Java Enterprise Edition. Java Magazine besteedt dit hele jaar aandacht aan de belangrijkste componenten in Java EE6 - met in dit nummer de schijnwerper op EJB 3.1.

Het jaar van JEE6 (5)

Enterprise Java Beans 3.1

Enkele jaren geleden was het op zijn zachtst gezegd moeilijk om met Enterprise Java Beans te werken. De toenmalige versie was 2.0 of 2.1 en om een EJB te maken moesten verschillende interfaces worden geïmplementeerd en ingewikkelde configuratiebestanden worden geschreven. Met de komst van EJB 3.0 in Java EE5 is hier veel aan verbeterd. Een Session Bean hoefde nog slechts één interface (Local of Remote) te implementeren, persistence kon gedaan worden op basis van JPA en vrijwel alles dat met EJBs te maken had, kon aan de hand van annotaties worden geconfigureerd. De EJB 3.1 specificatie borduurt hierop voort. Naast enkele verdere simplificaties worden ook enkele nieuwe services en het concept van EJB Light geïntroduceerd.

Local Session interfaces in EJB 3.1?

In EJB 3.0 kon een Local Session Bean worden gedefinieerd door middel van een interface en een (of meerdere) implementerende class(es). Een Local Session Interface zag er bijvoorbeeld zo uit:

```
@Local
public interface MySessionLocal {
         List<Person> getAllPersons();
         Person findPersonByName(String name);
}
```

en de implementerende class bijvoorbeeld als volgt

```
@Stateless
public class MySessionBean implements MySessionLocal {
         public List<Person> getAllPersons() {
                  ...
         }

         public Person findPersonByName(String name) {
                  ...
         }
}
```

In EJB 3.1 is de Local interface overbodig geworden. Een Local Session Bean kan daarom als volgt worden gedefinieerd:

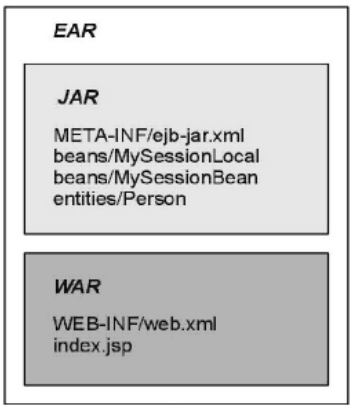
```
@Stateless
public class MySessionBean {
         public List<Person> getAllPersons() {
                  ...
         }

         public Person findPersonByName(String name) {
                  ...
         }
}
```

Let op dat dit exact dezelfde class is als in het EJB 3.0 voorbeeld, maar dat deze class nu geen interface implementeert! Indien een Remote Session Bean nodig is, dient er nog steeds een Remote interface te zijn die door de Bean moet worden geïmplementeerd.

WAR in plaats van EAR

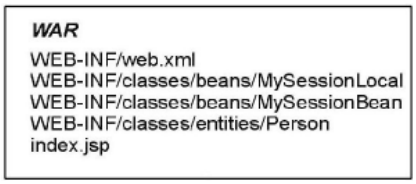
Een andere simplificatie is het feit dat EJBs niet meer in een aparte jar geplaatst hoeven te worden en dat ook ear bestanden overbodig zijn geworden. Je kunt nu je EJB classes tussen de andere classes van een web applicatie plaatsen en alles in een war

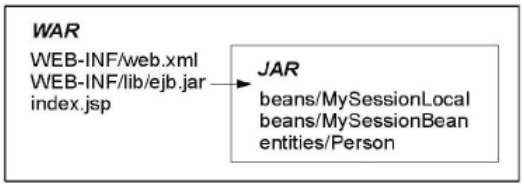


Wouter van Reeven
is JEE Technology Leader bij AMIS. Hij is te bereiken via email wouter.van.reeven@amis.nl.

deployen. Stel dat de Session Bean van de vorige paragraaf onderdeel uitmaakt van een web applicatie. Dan zou met EJB 3.0 een ear gemaakt moeten worden met bijvoorbeeld de inhoud zoals op de vorige pagina.

Met EJB 3.1 volstaat een war die op een van onderstaande manieren opgebouwd kan worden:





Globale JNDI namen

Op het moment dat in een applicatie een resource (Session Bean, Datasource, JMS queue) wordt geïnjecteerd, wordt een instance hiervan opgezocht in de Java Naming and Directory Interface, JNDI. Tot EJB 3.1 was er geen standaard voor de namen van classes die in JNDI geregistreerd worden.

Iedere vendor van een Enterprise Container stond het vrij een JNDI naming scheme op te stellen wat het op zijn zachts gezegd moeilijk maakte om dezelfde applicatie op verschillende containers te deployen.

De EJB 3.1 specificatie bevat nu een JNDI naming convention waar iedere vendor aan moet voldoen. In het algemeen ziet een JNDI naam er nu als volgt uit:

```
java:global[/<app-name>]/<module-name>/<bean-
name>[!<fully-qualified-interface-name>]
```

De [] betekenen dat dat deel optioneel is. Hierbij is <app-name> alleen van toepassing als de applicatie in een ear wordt gedeployed. Module-name is de naam van de jar of war waarin de op te zoeken bean zich bevindt.
Bean-name is de naam van de bean. Als een bean meerdere client interfaces implementeert dan kan de specifieke interface opgegeven worden met de fully-qualified-interface-name.
Merk op dat dit bovenstaande kan worden geconfigureerd. Zo kun je app-name specificeren in application.xml, module-name in ejb-jar.xml (in het geval van een jar) of web.xml (in het geval van een war) en bean-name in het name attribuut van de @Stateless, @Stateful of @Singleton annotaties.

Singletons

Veel ontwikkelaars van applicaties hebben aangegeven behoefte te hebben aan Singleton beans, ofwel beans die éénmaal per applicatie worden geïnstantieerd en door iedereen hergebruikt. Verschillende vendors van Enterprise Containers hebben het op hun manier mogelijk gemaakt om aan te geven hoe vaak een bean mag worden geïnstantieerd. Dit leidde weer tot deployment problemen over verschillende containers heen.

Met EJB 3.1 wordt een derde type Session Bean geïntroduceerd: de Singleton Bean welke wordt geïdentificeerd met de @Singleton annotatie. Een @Singleton bean is voor alle clients over de hele applicatie beschikbaar en ondersteunt concurrent toegang tot de bean.

De lifecycle van een Singleton bean wordt bepaald door de Enterprise Container maar kan worden beïnvloed met de @Startup annotatie. Met deze annotatie is het mogelijk om aan te geven dat een Singleton afhankelijk is van een andere Bean. Als de Singleton dan wordt geïnstantieerd, zorgt de container ervoor dat de andere Bean al geïnstantieerd is.

Voor configuratie van de concurrent toegang van Singleton beans worden twee strategies gedefinieerd, namelijk Container Managed Concurrency (CMC, de default) en Bean Managed Concurrency (BMC). In het geval van CMC is de container verantwoordelijk voor het bijhouden van Read en Write locks op iedere method van de Singleton Bean. Deze kunnen gespecificeerd worden met de @Lock annotatie. Indien een Write Lock is gezet, dient een aanroep aan de method te wachten tot het Lock vrijkomt. Het is mogelijk per method aan te geven wat de timeout hiervoor moet zijn.

Timer-Service

Timers bestaan al sinds EJB 2.1 en worden frequent gebruikt. Helaas kleven er enkele nadelen aan de Timers. Zo moeten alle Timers programmatisch aangemaakt worden, ontbreekt flexibiliteit bij het schedulen van Timers en is clustering support erg slecht.

Vooral de eerste twee punten zijn aangepakt in EJB 3.1. Het is mogelijk automatisch timers te laten creëren door de container met behulp van de @Schedule annotatie. Daarnaast zijn er verregaande mogelijkheden om timers te schedulen. Het gaat te ver om alle mogelijkheden in dit artikel te bespreken, maar hier zijn wel enkele voorbeelden:

```
// elke maandag om middernacht
@Schedule(dayOfWeek="Mon")

// elke weekdag om 3:15
```

Er is door developers veel vraag geweest naar Singleton beans en daar is gehoor aan gegeven.

De EJB Expert Group heeft goed geluisterd naar het commentaar vanuit het veld.

```
@Schedule(minute="15", hour="3", dayOfWeek="Mon-Fri")

// elke 5 minuten van ieder uur
@Schedule(minute="*/5", hour="*")

// elke laatste do in nov om 1 uur
@Schedule(hour="1", dayOfMonth="Last Thu", month="Nov")
```

Asynchrone invocaties

Asynchrone invocaties op EJBs is een van de meest belangrijke vernieuwingen in de specificatie. Vanaf EJB 3.1 kan iedere EJB asynchroon aangeroepen worden. Bij een dergelijke aanroep moet de container de controle over de runtime teruggegeven aan de client voor deze de aanroep aan de betreffende bean doorgeeft.

Om asynchrone aanroepen mogelijk te maken, kan de @Asynchronous annotatie worden gebruikt. Deze annotatie kan op methods gezet worden, maar ook op classes, superclasses of interfaces, hetgeen betekent dat alle methods in de betreffende class of interface asynchroon aangeroepen kunnen worden.

```
@Stateless
@Asynchronous
public class A { (...) }

@Singleton
public class B {
    @Asynchronous
    public void myMethod() { (...) }
}

@Stateless
public class C implements CLocal, CRemote {
    public void myMethod() { (...) }
}

@Local
@Asynchronous
public interface CLocal {
    public void myMethod();
}

@Remote
public interface CRemote {
    public void myMethod();
}
```

In het bovenstaande voorbeeld zijn alle methods van A asynchroon aanroepbaar, van B alleen de myMethod method en van C alleen de myMethod method indien deze via een local interface wordt aangeroepen.

De return type van een asynchroon aanroepbare method moet void zijn of Future<V>, waarbij V het type is van het resultaat van de aanroep. Future is een interface die in Java 5 geïntroduceerd is en die het resultaat van een asynchrone aanroep representeert.
De interface bevat vier methods namelijk cancel (probeert de aanroep te stoppen), get (geeft de return waarde terug als de aanroep klaar is), isCancelled (geeft aan of de aanroep gestopt is) en isDone (geeft aan of de aanroep klaar is). Een EJB 3.1 container moet een implementatie van de Future interface aanbieden, die AsyncResult heet. Via deze AsyncResult kan de return waarde van de method aanroep uit de container worden opgehaald.

EJB Light

De EJB API bestaat uit veel onderdelen die gezamenlijk het implementeren van business logica in een breed scala aan enterprise omgevingen mogelijk maken. De aanwezigheid van deze volledige set aan APIs is in lang niet alle omgevingen noodzakelijk. Daarnaast kan deze brede inzetbaarheid van EJB een drempel vormen voor beginnende ontwikkelaars.

Om deze redenen is er met EJB 3.1 een minimale subset va de EJB API gedefinieerd die bekend staat als EJB 3.1 Lite. Merk op dat EJB 3.1 Lite geen product is! Je moet het zien als een op zichzelf staande subset die een kleine, krachtige selectie van EJB onderdelen bevat die geschikt is voor het ontwikkelen van business logic voor een groot deel van de toepassingen van EJB.

EJB 3.1 Lite bestaat volledig uit APIs uit EJB 3.1. Er zijn geen extra APIs opgenomen in EJB 3.1 Lite. Dat betekent dat iedere EJB 3.1 Lite applicatie ook op een EJB 3.1 omgeving gedeployed kan worden.
De zaken die onder EJB 3.1 Lite vallen zijn:

- Stateless, Stateful en Singleton beans
  - Alleen Local zonder interface
  - Alleen Synchrone method aanroepen
- Container en Bean Managed Transacties
- Declaratieve en programmatische security
- Interceptors
- Deployment Descriptor (ejb-jar.xml) support

Zaken die derhalve buiten EJB 3.1 Lite vallen zijn Message Driven Beans, Remote Session Beans, JAX-WS Web Service End Points, EJB Timer Service en Asynchrone aanroepen.

Conclusie

De EJB expert group heeft goed geluisterd naar het commentaar, de suggesties en opmerkingen van leveranciers van EJB containers en van ontwikkelaars.
Veel punten waarop EJB de afgelopen jaren becommentarieerd werd zijn verholpen en verbeterd. Dit is een trend die niet alleen in de nieuwe EJB specificatie terug te zien is, maar over de gehele Java EE linie. Polls op internet en tijdens conferenties tonen aan dat Java EE in populariteit aan het stijgen is en dat er halsreikend uitgekeken wordt naar de definitieve Java EE specificaties.
Tijdens Devoxx kondigde Roberto Chinnici (de voorzitter van de Java EE 6 Expert Group) aan dat Java EE 6 op 10 december definitief vast zal liggen.
Ik kijk uit naar die datum.
En ik hoop jij ook. «