Testen

Veel Java EE applicaties zijn gebaseerd op Enterprise Java Beans (EJB) 3.0. Het testen daarvan blijft een uitdaging, met name door de functionaliteit die de applicatieserver biedt. Het doel van dit artikel is een beginpunt geven voor het testen van EJB 3.0 omgevingen met OpenEJB. Verder laten we de kracht van OpenEJB in een testomgeving zien.

Kracht van OpenEJB in een testomgeving

Meer verificatiemogelijkheden dan Java EE6

Een Java-applicatieserver bestaat minimaal uit twee verschillende containers: een web-container en een EJB-container. EJB-functionaliteit is alleen binnen de EJB-container beschikbaar. Dit zorgt ervoor dat code niet los kan worden getest. De POJO-functionaliteit kan natuurlijk wel los worden getest, maar de containerfunctionaliteit zoals transactiemanagement, security, etcetera niet. Een losse applicatieserver neerzetten voor testdoeleinden is een oplossing, maar dit vereist extra hardware (servers) en inspanning (configuratie). Ook wordt hierdoor de gehele omgeving complexer, zodat nog steeds niet makkelijk kan worden getest.

Een oplossing is Apache OpenEJB. OpenEJB is een lightweight EJB 3.0 container die alle functionaliteit van EJB 3.0 bezit. OpenEJB kan vanuit de code worden opgestart en geconfigureerd. De meeste applicatieservers kunnen wel vanuit de code worden opgestart, maar vereisen losse configuratiefiles. OpenEJB vereist geen (configuratie) files of directories, maar gebruikt slechts zijn eigen bibliotheken. Dit maakt de weg vrij om vanuit JUnit, de Java standaard voor unit-testen, EJB 3.0 code te testen.

We kijken naar het testen van EJB 3.0 session beans, message beans en ook Java Persistence API (JPA) entiteiten. De laatste valt niet onder de EJB 3.0 specificatie, maar om een logisch geheel te testen nemen we deze toch mee. We gaan kijken hoe OpenEJB te configureren is en hoe de unit-tests kunnen worden geschreven, zodat alle onderdelen van EJB 3.0 kunnen worden getest.

EJB specificatie

De EJB 3.0 specificatie heeft veel verbetering gebracht in Java EE applicatieontwikkeling. Testen is echter een onderdeel dat de EJB 3.0 specificatie niet behandeld. De testbaarheid is niet veranderd ten opzichte van eerdere versies. De grootste verandering in de specificatie is de introductie van annotaties; de configuratie is verplaatst van xml-files naar annotaties. Annotaties hebben geen invloed op de betekenis van de code, maar op hoe bibliotheken en tools met de code moeten omgaan. EJB 3.0 is zelf een bibliotheek. Dit verklaart waarom testen moeilijk is: de annotaties hebben alleen betekenis binnen de EJB 3.0 bibliotheek. In eerdere EJB-versies waren er geen annotaties maar xml-files, dit geeft precies hetzelfde probleem.

Deze tekortkoming is ondertussen erkend en in de EJB 3.1 specificatie van eind 2009 gedeeltelijk opgelost. De EJB 3.1 specificatie is onderdeel van Java EE 6. Er zijn op dit moment slechts twee applicatieservers die Java EE 6 compleet ondersteunen: Resin en de referentie-applicatieserver Glassfish. Verder is Java EE 6 niet de oplossing voor alle testproblemen. Hier komen we nog op terug.

Unit-testen

Unit-tests zijn bedoeld om afzonderlijk stukken code (units) te testen. In een Java EE applicatie is het moeilijk de grenzen van stukken code te definiëren. Sommige stukken kunnen wel los getest worden, andere niet. JPA-entiteiten lenen zich perfect om afzonderlijk te worden getest. Session beans en message beans daarentegen gebruiken vaak ook andere componenten en kunnen moeilijk los getest worden. Mock-objecten zijn hier een optie, maar we maken daar geen gebruik van. Het gaat bij Java EE applicaties vooral om de samenhang van de code, vandaar dat we geen mock-objecten gebruiken.

Anton Gerdessen
is consultant bij Transfer Solutions. Hij werkt vooral met Java technologie, maar houdt zich ook bezig met de raakvlakken met 'oudere' Oracle technologie.

Het voordeel van unit-testen is tweeledig: In de ontwikkelfase dwingt het ontwerpen en maken van unit-tests ontwikkelaars om goed na te denken over de code en mogelijke foutscenario's. Activiteitdiagrammen, die de programmastructuur definiëren, kunnen hierbij helpen. Het andere voordeel ligt bij het onderhoud van de software. De unittests als regressietests draaien, is enorm krachtig. Dit werkt vooral goed als de discipline opgebracht wordt, dat wanneer een fout in de software gevonden wordt, altijd eerst een test geschreven wordt die de fout reproduceert. Deze test blijft onderdeel van de regressietest en zo evolueert de kwaliteit van de tests samen met de software.

Apache OpenEJB

Apache OpenEJB is een EJB-implementatie en is beschikbaar onder de Apache 2.0 licentie. Het ondersteunt de EJB versies 2.0, 2.1 en 3.0. OpenEJB is niet ontwikkeld als testomgeving. OpenEJB wordt hier wel vaak voor ingezet. In principe kan iedere applicatieserver dit, maar het grote verschil zit in de snelheid van opstarten. Er mag geen afhankelijkheid zijn tussen de tests. Eigenlijk moet na elke test de applicatieserver geheel worden herstart. Zo kan volledige onafhankelijkheid tussen tests worden gegarandeerd.
De belangrijkste eigenschappen die OpenEJB goed toepasbaar maken in een testomgeving zijn dus:

- erg snelle EJB-container starttijd;
- geen xml configuratie;
- EJB 3.0 is volledig gesupport.

Omgeving en testopzet

We gebruiken de aangepaste tabellen, emp en dept, van het standaard Oracle-schema 'scott' en –uiteraard- EJB 3.0 en JPA. We gebruiken de volgende klassen:

- een JPA-entiteit genaamd Employee als representatie van de tabel emp;
- een JPA-entiteit genaamd Department als representatie van de tabel dept;
- een session bean genaamd EmployeeSessionBean als session facade voor de bijbehorende employee functionaliteit;
- een message bean genaamd EmployeeMDBean voor asynchrone toegang tot de EmployeeSessionBean;
- een utility klasse voor generieke testfunctionaliteit genaamd TestUtil. Hierin verwerken we de generieke OpenEJB configuratie;
- een superklasse voor alle testcases genaamd JUnitEmployeeSuper. Deze klasse voegt specifieke functionaliteit voor tests op de Employee functionaliteit toe aan OpenEJB, denk aan JMS-Queues;
- We gebruiken een groot gedeelte van de EJB 3.0 en JPA-specificatie met de zes klassen. Deze case is niet bedoeld om een compleet overzicht van de specificatie van EJB 3.0 of JPA te geven.

Unit-tests zijn enorm krachtig als voor elke fout in de software ook een test wordt geschreven.

OpenEJB configuratie

Om de EJB-container op te starten moet de InitialContext worden aangemaakt, verwijzend naar de OpenEJB LocalInitialContextFactory.

```
Properties config = new Properties();

config.put( Context.INITIAL_CONTEXT_FACTORY,
         "org.apache.openejb.client.
LocalInitialContextFactory");
```

Bijna alle Java EE applicaties maken gebruik van een database. In deze case wordt gebruikt gemaakt van een Oracle-database. Dus we zullen een databaseverbinding moeten configureren. Het volgende stuk code laat de configuratie van de datasource zien:

```
String dsName = "OpenEJBCaseDS";

config.put(dsName, "new://Resource?type=DataSource");
config.put(dsName+".JdbcDriver", "oracle.jdbc.
OracleDriver");
config.put(dsName+".JdbcUrl",
                  "jdbc:oracle:thin:@
SWH05932ORA1:1521:ds");
config.put(dsName +".UserName", "OpenEJBCase");
config.put(dsName +".Password", "OpenEJBCase");
```

Veel tools voor het testen van JPA-entiteiten verwachten voor testdoeleinden een andere persistence.xml dan voor ontwikkel- of productiedoeleinden. In de persistence.xml staat dan voor testcode aanvullende informatie over de databaseconnectie. Bij OpenEJB is de enige eis dat de naam van de opgegeven datasource dezelfde is als de naam in persistence.xml. Nu we de generieke configuratie ingesteld hebben, kunnen we de klasse JUnitEmployeeSuper verder uitwerken. Deze klasse leest de generieke configuratie vanuit de utility klasse en voegt voor de employee-functionaliteit specifieke configuratie toe. In dit geval gaan we voor de message bean alvast, de te gebruiken JMS-Queue configureren. Zie het volgende stuk code:

```
config.put("employeeQueue",                "new://
Resource?type=javax.jms.Queue");
config.put("employeeQueue.destination",    "jms/ejbcase/
employee");
```

Dit is in OpenEJB voldoende om de JMS-Queue te configueren De JNDI-naam waarop deze JMS-Queue

beschikbaar is, geven we aan bij destination. We hebben nu voldoende geconfigureerd om OpenEJB op te starten. Dit gaat als volgt:

```
final Properties config = TestUtil.
getOpenEJBProperties();
…..
context = new InitialContext(config);

context.bind("inject", this);
```

Het interessante zit in de laatste regel van de code. Samen met de annotatie van de JUnitEmployeeSuper klasse met @LocalClient zorgt deze regel ervoor dat resources worden geïnjecteerd in de testklasse. We kunnen elke resource-injectie gebruiken die binnen een EJB-container beschikbaar is. In het volgende stuk code injecteren we in deze superklasse een EntityManager en de eerder aangemaakte JMS-Queue:

```
@PersistenceContext(unitName = "business_service")
private transient EntityManager entityManager;

@Resource(name = "jms/ejbcase/employee", mappedName =
"jms/ejbcase/employee", type = Queue.class)
private transient Queue employeeQueue;
```

Het kunnen injecteren van de benodigde EJB-resources in de testklasse is functionaliteit die OpenEJB uniek maakt.

Testen van een JPA-entiteit

We beginnen nu met het testen van de Employee JPA-entiteit. Omdat we de JPA-entiteiten los testen, maken we gebruiken van een UserTransaction. We kunnen bijvoorbeeld testen of we een Employee de database in krijgen. Dit doen we in twee stappen:

1. een Employee entiteit aanmaken en opslaan
2. deze opzoeken om te controleren of het opslaan goed gegaan is.

Stap 1 het opslaan:

```
getUserTransaction().begin();

Employee employee = new Employee("insertEmp");
getEntityManager().persist(employee);
long employeeId = employee.getNumber();

getUserTransaction().commit();

Vervolgens het opzoeken:

getUserTransaction().begin();

Employee retrievedEmp =        getEntityManager().
find(Employee.class,employeeId);
assertNotNull("Retrieved employee was empty, insert must
have     failed!", retrievedEmp);
```

Op dezelfde manier kunnen we JPA-QL-Queries testen of collecties doorlopen. De UserTransaction en EntityManager zijn beide in de testklasse geïnjecteerd en door een methode beschikbaar gemaakt. Het grote voordeel ten opzichte van andere manieren van JPA-entiteiten testen is dat we precies dezelfde configuratie (persistence.xml) gebruiken in de testomgeving als in de ontwikkel- en productieomgeving. Verder vergt het injecteren van resources weinig code of ontwikkeltijd.

Testen van een session bean

We kunnen ook session beans testen met OpenEJB, dit gaat op een soortgelijke manier. Voor session beans gebruiken we de InitialContext van OpenEJB om de bean op te zoeken. We voeren de test uit in drie stappen:

1. Controleren of de bean op te zoeken is in de JNDI-boom;
2. De methoden van de session bean aanroepen;
3. De resultaten controleren.

Het opzoeken van de session bean behandelen we eerst:

```
Object object = getContext().lookup("ejb/Employee");

assertNotNull("Employee session bean doesn't exist",
object);
assertTrue("Type object is no EmployeeSession",
                    object instanceof EmployeeSession);
```

We zoeken de bean en controleren of het type correct is. Vervolgens kunnen we de in deze session bean beschikbare methoden testen. In dit geval is dat één methode, die aan de hand van een meegegeven naam een nieuwe Employee aanmaakt. In deze test blijven we ook volledig toegang houden tot de EntityManager en kunnen zo controleren of de aanroep gelukt is. Belangrijk is dat de session bean in een volledige EJB-omgeving draait. In het voorbeeld van het testen van JPA-entiteiten hebben we via de UserTransaction de transactiegrenzen aangegeven. Zoals al eerder gezegd dienen we bij JPA de transactiegrenzen aan te geven, iets wat bij EJB 3.0 met annotaties kan. De transactiegrenzen worden nu volledige bewaakt door OpenEJB. Het aanroepen gaat niet anders dan met welke andere EJB-cliënt:

```
EmployeeSession empSession = (EmployeeSession) object;
Long empId = empSession.storeEmployee("empName");
```

We kunnen controleren of de aanroep naar de session bean geslaagd is door de nieuwe Employee op te zoeken via de EntityManager. Dit gaat op precies dezelfde manier als de JPA-entiteiten test:

```
getUserTransaction().begin();

Employee retrievedEmp = getEntityManager().find(
Employee.class, empId);
assertNotNull("Retrieved Employee was empty, session
bean call must have failed!", retrievedEmp);
```

Het voorbeeld dat we hier geven is welliswaar simplistisch, maar we kunnen op deze manier alle resultaten van een session bean controleren. OpenEJB biedt ons hier vooral snelheid; het starten van OpenEJB is een kwestie van 2 tot 3 seconden (Laptop dual core en 3G geheugen). Als we een soortgelijke

Injecteren van EJB-resources in de testklasse maakt OpenEJB uniek.

test uitvoeren op een externe applicatieserver moeten we de applicatie eerst deployen en dan via remote aanroepen de applicatie te testen. Bovendien hebben we dan geen directe toegang tot de EntityManager en dus geen directe controlemogelijkheid.

Testen van een Message bean

We kunnen ook message beans testen via OpenEJB. OpenEJB gebruikt Apache ActiveMQ als JMS-implementatie. Dit is volledig te configureren, een uitgebreid voorbeeld is te vinden in de klasse TestUtil. Waar vooral op moet worden gelet is redelivery. Standaard staat deze op 1 seconde, dit is voor testdoeleinden wat traag. In de voorbeeldcode is deze naar 300 milliseconden gezet. De message bean gebruikt de EmployeeSessionBean om de logica achter de aanroep af te handelen. We gebruiken de door eerdere injectie verkregen JMS-Queue en ConnectionFactory. We voeren de test weer uit in een aantal stappen:

- een bericht in de JMS-Queue zetten
- 1 seconde wachten om er zeker van te zijn dat het bericht verwerkt is
- de resultaten controleren.

De eerste twee stappen laten we in één stuk code zien.

```
sendMessage(getEmployeeQueue(), “mbdEmp”);
Thread.sleep(1000L);
```

We gebruiken een hulpmethode om het bericht te versturen, deze is terug te vinden in de voorbeeldcode. We gebruiken de methode getEmployeeQueue om de geïnjecteerde referentie naar de JMS-Queue op te halen. Vervolgens wachten we een seconde om zeker te weten dat het bericht is verwerkt. We doen dit, omdat JMS asynchroon werkt en het versturen van het bericht niet gelijk staat aan de verwerking van het bericht.
De controle of het bericht ook daadwerkelijk is verwerkt, gaat op vrijwel dezelfde manier als bij de session bean test. Omdat we nu asynchroon werken hebben we geen sleutel van de aangemaakte JPA-entiteit teruggekregen en moeten we de Employee opzoeken via de naam. We gaan er vanuit dat na iedere test de database weer leeg is, anders gaat deze test de tweede keer altijd goed.

```
getUserTransaction().begin();
Query query = getEntityManager().createNamedQuery(
“Employee.findEmpByName”);
query.setParameter(“name”, text);
Employee retrievedEmp = (Employee) query.
getSingleResult();
assertNotNull(“Employee stored via MDBean not found”,
retrievedEmp);
```

We gebruiken de EntityManager ditmaal om een Query uit te voeren die de Employee opzoekt.
Via de injectie van OpenEJB kunnen we alle functionaliteit van JMS testen. Herafleveren is volledig ondersteund om foutscenario's te testen. Verder is ook een 'dead letter queue' aanwezig. Eigenlijk zijn foutscenario's de scenario's waarop getest moet worden, daarom is de ondersteuning hiervan in OpenEJB zo belangrijk.

OpenEJB opties

We hebben alle relevante functionaliteit van OpenEJB gedemonstreerd. In dit onderdeel gaan we kijken naar wat extra opties van OpenEJB.
Ten eerste wordt bij OpenEJB, OpenJPA als JPA-implementatie gebruikt, maar deze kan worden vervangen door een implementatie als Hibernate.
Ten tweede kijken we naar de verschillende versies van OpenEJB en hun eventuele problemen. Ook kijken we kort naar de combinatie van verschillende versies OpenEJB en JMS-implementaties. Vervolgens gaan we kort in op de combinatie OpenEJB en bean validation (JSR-303). Bean validation is in opkomst, hierbij worden JPA-entiteiten op een uniforme manier gevalideerd door middel van annotaties.

OpenEJB en Hibernate

OpenJPA gebruikt OpenJPA als de standaard JPA-implementatie. OpenEJB kijkt naar de persistence.xml om te beoordelen of een andere implementatie moet worden gebruikt. We kunnen forceren dat een andere JPA-implementatie wordt gebruikt, zonder de persistence.xml aan te passen. Gebruik hiervoor de volgende Java property:

```
System.setProperty(“javax.persistence.provider”,
        “ org.hibernate.ejb.HibernatePersistence “);
```

Bovenstaande code forceert JPA om altijd Hibernate te gebruiken. Om betrouwbare testresultaten te krijgen is het verstandig als de test- en productieomgeving zo veel mogelijk op elkaar lijken. Een veel gemaakte keuze voor de JPA-implementatie is Hibernate. Ook binnen OpenEJB is dit mogelijk. Hiertoe zijn de volgende bibliotheken nodig:

- Hibernate
  - o Hibernate core
  - o Hibernate annotations
  - o Hibernate entitymanager
- slf4j (een recente versie b.v. 1.5.11)

Op de website van Hibernate is te vinden dat, als we Hibernate als EnitityManager willen gebruiken, de eerste drie genoemde bibliotheken van Hibernate nodig zijn. Hibernate en OpenEJB gebruiken beide slf4j, maar verschillende versies. Dit geeft conflicten. Om dit op te lossen kunnen we het beste alle bibliotheken van slf4j van Hibernate en van OpenEJB verwijderen, of deze niet meenemen in het classpath. Vervolgens kunnen we uit de slf4j de volgende twee jars uit de distributie gebruiken:

- slf4j-api<versie>.jar
- slf4j-log4j<versie>.jar

De test- en productie-omgeving moeten zo veel mogelijk op elkaar lijken om goede resultaten te krijgen.

Dit moet voldoende zijn om Hibernate als JPA-implementatie binnen OpenEJB te gebruiken. Ook in de voorbeeldcode wordt Hibernate als JPA-implementatie gebruikt.

OpenEJB en versies

De huidige versie van OpenEJB is 3.1.2 (14 oktober 2009). Er wordt druk gewerkt aan versie 3.1.3.
Versie 3.1.2 kan niet samenwerken met JDK versie 1.6.0.18. Dit kan handmatig opgelost worden door een klasse te vervangen. Een andere oplossing is de huidige ontwikkelversie (3.1.3) te gebruiken. Hierin zit een aantal verbeteringen en treedt het genoemde probleem niet meer op.

OpenEJB en JMS

Open EJB versie 3.1.2 gebruikt als JMS-implementatie Apache ActiveMQ versie 4.x. In deze versie geeft het afleveren van berichten problemen. Als een message bean een Exceptie gooit, accepteert de JMS-implementatie geen andere berichten meer. Dit is niet volgens de Java EE specificatie. In ActiveMQ versie 5.x is dit probleem opgelost. Deze versie zit standaard in OpenEJB 3.1.3.

OpenEJB en bean validation

JSR-303 bean validation behandelt het probleem van validatie van een JPA-entiteit. Bean validation is onderdeel van JPA 2.0. Als we bijvoorbeeld een veld specificeren van het type String, missen we nog informatie. Zoals lengte, het wel of niet leeg mogen zijn etcetera. Bean validation gaat in op deze problemen. OpenEJB kan overweg met bean validation.
De benodigdheden zijn een entity-listener en een orm.xml om de entity-listener te laten afgaan. Dit is niet specifiek voor OpenEJB, maar werkt voor alle JPA-implementaties die nog JPA 1.0 gebruiken. Hibernate validations is de referentieimplementatie voor bean validation en werkt goed samen met OpenEJB.

Java EE 6 testen

Zoals we in het begin van dit artikel al aangaven werkt OpenEJB met EJB specificatie 3.0. In EJB 3.1 is het probleem van moeizaam testen onderkend en gedeeltelijk aangepakt. Voor JPA-entiteiten verandert er niets; er zal nog altijd getest moeten worden met een andere persistence.xml waarin de connectieparameters staan. Hierin blijft OpenEJB voorlopen op EJB 3.1.

Het testen van session beans en message beans is wel aangepakt. Containerleveranciers zijn verplicht een interface beschikbaar te stellen om de container op te starten, zoals OpenEJB nu voor EJB 3.0 biedt. We geven nu twee korte voorbeelden van hoe dit eruit gaat zien.

Als eerste de session bean:

```
EJBContainer container = EJBContainer.
createEJBContainer();

Object object = container.getContext().lookup("ejb/
Employee");

assertNotNull("Employee session bean doesn't exist",
object);
assertTrue("Type object is no EmployeeSession ",
                    object instanceof EmployeeSession);
```

Het enige dat anders is in vergelijking met de OpenEJB test is de InitialContext creatie. OpenEJB start op door gegevens mee te geven in de aanroep van de InitialContext creatie. In Java EE 6 is het maken van de EJB-container een losse aanroep en kan aan deze EJB-container de InitialContext gevraagd worden. Met deze InitialContext kan vervolgens de session bean opgezocht worden.
Voor message beans geldt hetzelfde principe: We maken een instantie van een EJB-Container en zoeken via de context de ConnectionFactory en de JMS-Queue op:

```
EJBContainer container = EJBContainer.
createEJBContainer();

container.getContext().lookup("jms/ConnectionFactory");
container.getContext().lookup("ms/ejbcase/employee");
```

We kunnen nu berichten sturen, maar verliezen op deze manier wel de injectie van de EntityManager. En dit stelde ons nou juist in staat om aanroepen te controleren. Als we voor het testen een andere persistence.xml gebruiken, kunnen we wel een Persistence context opzoeken en het op die manier oplossen. Maar dan hebben we twee configuraties van de persistence.xml.
In het kort, EJB 3.1 lost problemen op, maar maakt niet alles mogelijk wat met OpenEJB mogelijk is.

Conclusie

We hebben laten zien hoe we gemakkelijk Java EE applicaties kunnen testen die gebruik maken van EJB 3.0. We hebben de injectiemogelijkheden van OpenEJB laten zien. Gecombineerd met de starttijd van de EJB-container maakt dit OpenEJB een goede keuze voor een testomgeving. Ook hebben we een vergelijking gemaakt tussen OpenEJB en de Java EE 6 testmogelijkheden. Hierin komt naar voren dat OpenEJB verificatiemogelijkheden biedt, waar Java EE 6 nog tekort schiet. Verder hebben we een omgeving geïntroduceerd die als basis testomgeving kan fungeren.

Alle voorbeeldcode is te downloaden. Er zijn twee versies van de code, één zonder alle externe bibliotheken en één met. Beide versies gebruiken Hibernate als JPA-implementatie en de OpenEJB 3.1.3 ontwikkelrelease van 12 april 2010.De databasescripts zijn aanwezig om de databaseomgeving aan te maken. Let wel op dat deze uitgaan van een Oracle RDBMS. «

Referenties

- Apache OpenEJB - http://openejb.apache.org/
- JUnit - http://www.junit.org/
- Java annotations - http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html
- Resin – http://caucho.com/resin-4.0/
- Glassfish - https://glassfish.dev.java.net/
- Voorbeeldcode zonder bibliotheken www.gerdessen.com/anton/OpenEJBCaseNoLibs.zip
- Voorbeeldcode met alle bibliotheken www.gerdessen.com/anton/OpenEJBCaseFull.zip
- Mockobjecten - http://nl.wikipedia.org/wiki/Mockobject
- Hibernate - http://www.hibernate.org/
- OpenEJB JDK 6_18 bug - https://issues.apache.org/jira/browse/OPENEJB-1131
- Apache ActiveMQ - http://activemq.apache.org/
- DBUnit - http://www.dbunit.org/
- Bean validation - http://agoncal.wordpress.com/2010/03/03/bean-validation-with-jpa-1-0/
- JSR 3-3 Bean validation - http://jcp.org/aboutJava/communityprocess/edr/jsr303/index.html
- Simple Logging Facade for Java (SLF4J) – http://www.slf4j.org/
- Hibernate stack - http://www.hibernate.org/mainColumnParagraphs/00/image/HibernateStacks.png