DEV

EJB 3.0 Persistence (2)

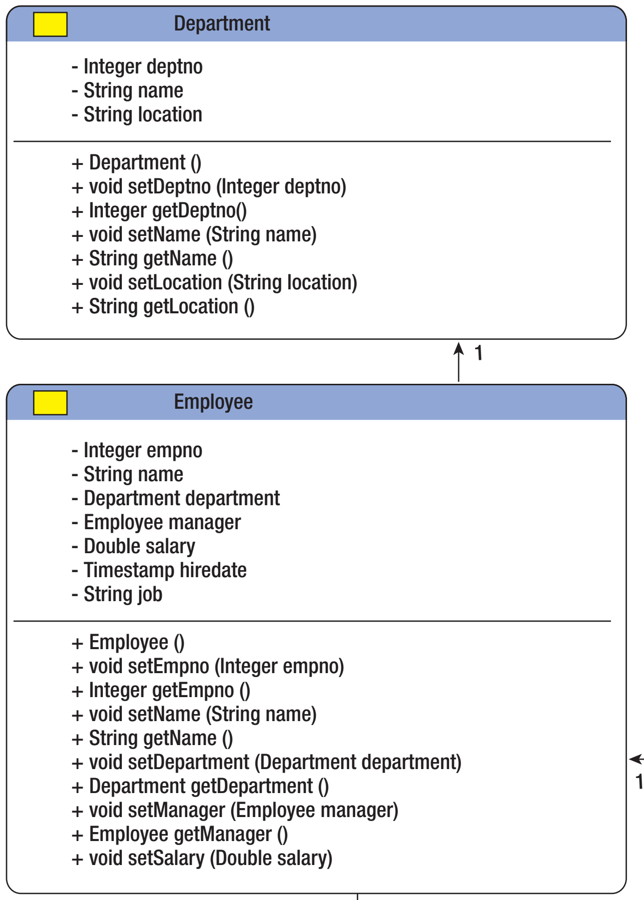
Toepassing in een J2SE-applicatie

EJB 3.0 Persistence is een nieuwe standaard voor Java-applicaties om met relationele databases te communiceren. De standaard wordt gesteund door alle belangrijke partijen die zich bezig houden met de mapping tussen Java-applicaties en relationele databases. EJB 3.0 Persistence is onderdeel van de later dit jaar te publiceren JEE5-standaard – de opvolger van J2EE 1.4. De komende periode zullen veel Java-ontwikkelaars die met databases te maken hebben in aanraking komen met EJB 3.0 Persistence. Het vorige artikel uit deze serie gaf een introductie van EJB 3.0 Persistence, dit keer laat Lucas Jellema zien hoe je er ook zelf mee aan de slag kan gaan.

Laten we eens een concreet voorbeeld bij de kop pakken. Het standaard SCOTT-schema in de Oracle-database met tabellen EMP en DEPT vormt ons uitgangspunt. Onze Java-applicatie gaat gebruikmaken van twee Entity's, Employee en Department geheten. De Employee Entity is een heel gewoon Java Object, met property's als Empno, Name, Job, Manager en Department en bijbehorende getter en setter methoden (zie afbeelding 1). Als je naar de code kijkt voor deze Java Classes wordt opeens duidelijk hoe Annotations aan de EntityManager duidelijk maken hoe deze Entity's gerelateerd zijn aan database-objecten:

```
package nl.amis.hrm.ejb30;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="DEPT")
public class Department {

    private Integer deptno;
    private String name;
    private String location;

    public Department() {
    }

    public void setDeptno(Integer deptno) {
        this.deptno = deptno;
    }

    @Id
    @Column
    public Integer getDeptno() {
        return deptno;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Column(name="DNAME")
    public String getName() {
        return name;
    }

    public void setLocation(String location) {
        this.location = location;
    }

    @Column(name="LOC")
    public String getLocation() {
        return location;
    }
}
```

Een Java Bean die als Entity moet worden beschouwd krijgt de @Entity-annotatie. Als de naam van class niet overeenkomt met de tabelnaam moet met de @Table-annotatie de naam van de tabel of view in de database worden aangegeven. Dit is een voorbeeld van Configuration by Exception: als de naam wel overeenkomt hoef je de @Table-annotatie niet te gebruiken! Van property's in de Java Bean kan met de @Column-annotatie bij de getter methode worden aangegeven dat het 'mapped, persistent' property's zijn die aan een database-column zijn gekoppeld. Als de naam van de property afwijkt van de naam van de kolom kan met (name="COLUMN_NAME") de juiste koppeling worden vastgelegd. De annotatie @Id tenslotte



Afbeelding 1. De Employee Entity is een Java-object met property's en bijbehorende getter en setter-methoden.

wordt gebruikt om aan te geven welk property de primary key vormt van de entiteit. Iedere entiteit is verplicht een primary key te hebben; deze kan wel samengesteld zijn. In Employee.java:

```
package nl.amis.hrm.ejb30;

import java.sql.Timestamp;
import java.io.Serializable;

import java.util.Collection;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.JoinColumn;
import javax.persistence.FetchType;
import javax.persistence.CascadeType;

@Entity
@Table(name="EMP")
public class Employee {
```

```
private Integer empno;
private String name;
private Department department;
private Employee manager;
private Double salary;
private Timestamp hiredate;
private String job;
private Collection<Employee> staff;

public Employee() {
}

public void setEmpno(Integer empno) {
    this.empno = empno;
}

@Id
@Column
public Integer getEmpno() {
    return empno;
}

public void setName(String name) {
    this.name = name;
}

@Column(name="ENAME")
public String getName() {
    return name;
}


public void setDepartment(Department department) {
    this.department = department;
}

@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="deptno")
public Department getDepartment() {
    return department;
}

public void setManager(Employee manager) {
    this.manager = manager;
}

@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="mgr")
public Employee getManager() {
    return manager;
}

public void setSalary(Double salary) {
    this.salary = salary;
}

@Column(name="SAL")
public Double getSalary() {
    return salary;
}

public void setHiredate(Timestamp hiredate) {
    this.hiredate = hiredate;
}

@Column
```

```
    public Timestamp getHiredate() {
        return hiredate;
    }

    public void setJob(String job) {
        this.job = job;
    }

    @Column
    public String getJob() {
        return job;
    }

    public void setStaff(Collection<Employee> staff) {
        this.staff = staff;
    }

    @OneToMany(mappedBy="manager", fetch=FetchType.LAZY)
    public Collection<Employee> getStaff() {
        return staff;
    }
}
```

In de Employee-entiteit wordt een nieuwe annotatie geïntroduceerd: @ManyToOne. Hiermee wordt een (foreign key) referentie gespecificeerd. In de Employee-entiteit zien we bijvoorbeeld:

```
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="DEPTNO")
    public Department getDepartment()
```

Dit wordt geïnterpreteerd door de EntityManager als: het department property van Entity Employee is een referentie naar een Department-object en wordt dus vertaald naar een kolom die DEPTNO heet in de onderliggende table EMP die verwijst naar de primary key van Entity Department – het property deptno dat is gemapped naar de kolom DEPTNO in table DEPT.
Verder is er aangegeven dat er sprake is van 'lazy fetch'. Dit wil zeggen dat de EntityManager het Department object waar een Employee naar refereert pas moet gaan instantiëren als er via een aanroep van getDepartment() voor de eerste keer expliciet om wordt gevraagd – en niet al direct bij het instantiëren van het Employee object. Deze lazy load – pas creëren als er om gevraagd wordt – is van belang om te voorkomen dat de EntityManager onnodig veel te veel data gaat lezen. De employee heeft ook een Manager, en die heeft ook weer een Manager, en zo verder. Als er niet af en toe door een 'fetch=lazy' een soort dam wordt opgeworpen, zou het lezen van een enkel object kunnen leiden tot het opbouwen van een zeer omvangrijke objectverzameling. De reciproke van de @ManyToOne is de @OneToMany. Deze zien we bij het property staff:

```
@OneToMany(mappedBy="manager", fetch=FetchType.LAZY)
public Collection<Employee> getStaff() {
```

Hier is aangegeven dat deze property wordt gespecificeerd als een OneToMany, een referentie naar een verzameling van nul, één of meer entiteiten – we zien hier overigens toepassing van de Java 5 Generic constructie waarbij van de Generieke Collection wordt aangegeven welk type object er in de Collection zit, namelijk Employees. De entiteit waarmee de relatie bestaat is dus Employee en het ManyToOne property waar de OneToMany mee correspondeert heet manager. Daarmee heeft de EntityManager voldoende informatie om de Collection te instantiëren.

EntityManager

Om nu in code met deze Entity's te gaan stoeien moeten we een EntityManager verkrijgen. Dit is kinderlijk eenvoudig zoals blijkt het volgende codefragment:

```
package nl.amis.hrm.ejb30;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class HrmClient {
    public HrmClient() {
    }

public static void main(String[] args) {
  // verkrijg een EntityManagerFactory instance op basis van de pu1
  // persistence-unit in de META-INF\persistence.xml file
  EntityManagerFactory emf = Persistence.createEntityManagerFactory
  ("pu1");
  EntityManager em = emf.createEntityManager();

  // voorbeeld van het creëren van nieuw Department
  Department dept = new Department();
  dept.setDeptno(60);
  dept.setName("OPTIMIZE");
  dept.setLocation("Alphen");
  EntityTransaction tx = em.getTransaction();
  tx.begin();
  em.persist(dept);
  tx.commit();
  // wijzig de nieuwe entiteit
  dept.setLocation("Nieuwegein");
  // en maak die wijziging permanent
  tx.begin();
  em.merge(dept);
  tx.commit();
}
}
```

In dit voorbeeld gebruiken we twee standaardservices van de EntityManager: persist() en merge(). Andere services zijn refresh(), remove(), find() – op primary key – flush() en create-

Query(). Tijdens de uitvoering van bovenstaand Java-fragment heeft de EntityManager twee SQL Statements op de database afgevuurd:

```
INSERT INTO DEPT (DEPTNO, LOC, DNAME) VALUES (60, 'Alphen', 'OPTIMIZE')
UPDATE DEPT SET LOC = 'Nieuwegein' WHERE (DEPTNO = 60)
```

Deze statements zijn volledig op grond van de Entity dept, de persistence.xml file en vooral de Annotaties in de Department.java class definitie geconstrueerd. Het update-statement doet alleen een update van de LOC-column. De EntityManager heeft dus kennelijk geconstateerd dat alleen het location-property was gewijzigd. Het verkrijgen van de EntityManager instance, via een EntityManagerFactory, gebeurt op basis van een configuratie-file – een kleine en de enige. Deze file, persistence.xml, staat in de directory META-INF in het classpath. Deze file bevat drie elementen:

- de concrete implementatie van de EntityManagerFactoryProvider (het <provider> element); in dit voorbeeld is dat de provider van de open source GlassFish reference implementation
- een opsomming van alle Entity's met hun volledig gekwalificeerde naam, inclusief package
- de connectie details: JDBC driver, database url, username en password

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence">
    <persistence-unit name="pu1">
  <provider>oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider
</provider>
        <!-- All persistence classes must be listed -->
        <class>nl.amis.hrm.ejb30.Department</class>
        <class>nl.amis.hrm.ejb30.Employee</class>
        <properties>
            <!-- Provider-specific connection properties -->
            <property name="jdbc.driver" value="oracle.jdbc.driver.
            OracleDriver"/>
            <property name="jdbc.connection.string" value="jdbc:oracle:
            thin:@localhost:1521:ORCL"/>
            <property name="jdbc.user" value="scott"/>
            <property name="jdbc.password" value="tiger"/>
        </properties>
    </persistence-unit>
</persistence>
```

EJB QL: query-taal

EJB 3.0 bevat een eigen query-taal, EJB QL 3.0, vergelijkbaar met EJB QL uit de EJB 2.1 specificatie maar meer nog Toplink Query Language en Hibernate's HQL. Query's in EJB QL worden geschreven in termen van entity's, property's en relaties. De query's worden vervolgens door de EntityManager omgezet in SQL. Het resultaat van de query wordt weer in de vorm van entity's aan de applicatie teruggegeven.

Ten opzichte van eerdere versies van EJB QL is de huidige 3.0 versie enorm verrijkt met onder meer subquery's, joins, bulk DML-operaties, (named) parameter binding, projectie (een query kan meer dan één type entity opleveren) en nieuwe functies – zoals concat, locate, trim, substring en length. Daarnaast ondersteunt de Query API het gebruik van NativeQueries, niet geschreven in EJB QL maar in het SQL van de onderliggende database. Hoewel daarmee uiteraard database-portabiliteit verloren gaat, kan op die manier wel gebruik worden gemaakt van geavanceerde database-features als PL/SQL-functies, Analytische Functies, Scalar Subquery en Connect By. Het fraaie van de NativeQuery is dat het resultaat nog steeds als een collection van Entities kan worden teruggegeven. Een eenvoudig EJB QL-voorbeeld om alle Departments te tonen:

```
...
  List<Department> allDepartments = em.createQuery("select object(o)
  from Department o").getResultList();
  // use a little Java 5.0 Autoboxing for simple, elegant for-looping
  for( Department d:allDepartments) {
      System.out.println("Department "+d.getDeptno()+"  "+d.getName());
  }
<< einde kader met computercode >>

Uiteraard kan dat wel interessanter. Denk bijvoorbeeld aan alle
Employees met meer dan drie ondergeschikten:
```

```
...
    List<Employee> superManagers = em.createQuery("select object(o)
    from Employee o where size(o.staff) > 2").getResultList();
    for( Employee e:superManagers) {
        System.out.println("Employee "+e.getName()+"  Count
Subordinates: "+e.getStaff().size());
    }
```

De SQL die de EntityManager genereert voor deze query luidt:

```
SELECT t0.EMPNO, t0.SAL, t0.JOB, t0.ENAME, t0.HIREDATE, t0.mgr,
t0.deptno FROM EMP t0
WHERE ((SELECT COUNT(*)
        FROM   EMP t1
        WHERE  (t1.mgr = t0.EMPNO)
       ) > 2)
```

Nu formuleren we een query waarmee het gebruik van beans, property's en relaties helder wordt geïllustreerd. Zoek alle medewerkers die een manager hebben die in dezelfde locatie werkt als zijzelf:

```
List<Employee> superManagers = em.createQuery("select object(o) from
Employee o where o.manager.department.location = o.department.location").
getResultList();
```

De SQL voor deze vraag luidt:

```
SELECT t0.EMPNO, t0.SAL, t0.JOB, t0.ENAME, t0.HIREDATE, t0.mgr,
t0.deptno
FROM DEPT t3, EMP t2, DEPT t1, EMP t0
WHERE ((t1.LOC = t3.LOC)
AND (((t2.EMPNO = t0.mgr)
AND (t1.DEPTNO = t2.deptno))
AND (t3.DEPTNO = t0.deptno)))
```

Niet iedere query hoeft een verzameling entity's op te leveren. We kunnen ook gewoon scalaire resultaten opvragen:

```
    Object[] salarayAggregates = (Object[])em.createQuery ("select
    avg(e.salary), max(e.salary), min(e.salary) from Employee e").
    getSingleResult();
    Double averageSalary = (Double)salarayAggregates[0];
    System.out.println("Average salary "+averageSalary+"; Maximum Salar
y"+(Double)salarayAggregates[1]);
```

Een voorbeeld van een NativeQuery, met gewoon SQL in plaats van EJB QL, op zoek naar de best verdienende Employee in ieder Department met gebruikmaking van Oracle Analytical functions:

```
    List<Employee> departmentChampions = em.createNativeQuery("select *
from (select emp.*, row_number() over (partition by deptno order by sal
desc) rn from emp) where rn =1  ", nl.amis.hrm.ejb30.Employee.class).
getResultList();

    for( Employee e:departmentChampions) {
        System.out.println("Employee "+e.getName()+"  Salary: "+e.get-
Salary());
    }
```

Hier zien we dat een gewone SQL-query expliciet wordt geassocieerd met de class Employee. Daarmee wordt de EntityManager geïnstrueerd om in de ResultSet op zoek te gaan naar 'kolommen' met de namen die via de @Column-annotaties gekoppeld zijn aan property's van de Employee-entity. Met behulp van de aldus gevonden waarden moeten Employees geïnstantieerd worden. Zoals hiervoor gedemonstreerd, kunnen query's dynamisch worden geconstrueerd. Maar query's kunnen ook voorgedefinieerd worden, met gebruik van Annotaties. Bijvoorbeeld de volgende annotatie in de Employee class die alle Employees met de job Salesman oplevert:

```
@Entity
@Table(name="EMP")
@NamedQuery(name="salesmen",queryString="SELECT e FROM Employee e WHERE
e.job='SALESMAN'")
public class Employee {
```

De in meta-data vastgelegde NamedQuery kan op de volgende wijze via de EntityManager uitgevoerd worden:

```
...
List<Employee> salesmen = em.createNamedQuery("salesmen").getResult-
List();
for( Employee e:salesmen) {
  System.out.println("Employee "+e.getName()+"  Job: "+e.getJob());
}
```

Uiteraard hebben we hier nog slechts het topje van de ijsberg voor wat betreft EJB QL 3.0 en de Query-interface in EJB 3.0 gezien. Maar hopelijk is duidelijk dat we kunnen spreken van een krachtige query-taal – met mogelijkheid van Native Query's als we er met EJB QL niet uitkomen en database-portabiliteit geen topprioriteit is – met een simpel aan te spreken interface en een krachtige wederzijdse vertaling van entity's in relationele data.

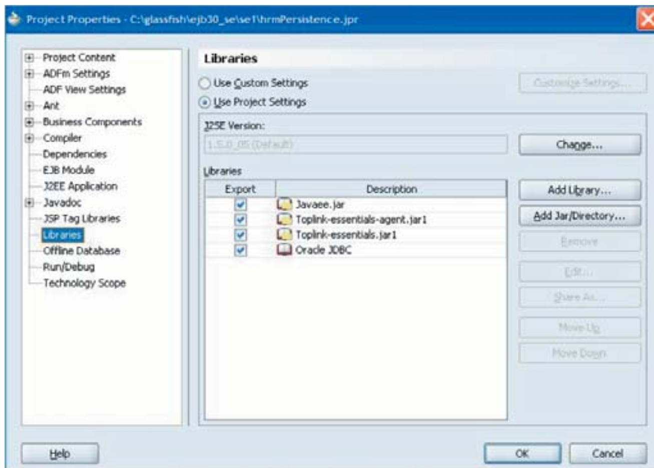
Aan de slag met EJB 3.0 Persistence in JDeveloper 10.1.3EA

Ik hoop dat nu je vingers jeuken om zelf eens aan de gang te gaan met EJB 3.0 Persistence. Dat kan ook heel eenvoudig, als je de volgende stappen volgt. Meer details over deze set-up vind je in het weblog-artikel Using GlassFish Reference Implementation of EJB 3.0 Persistence with JDeveloper 10.1.3EA http://technology.amis.nl/blog/?p=964.

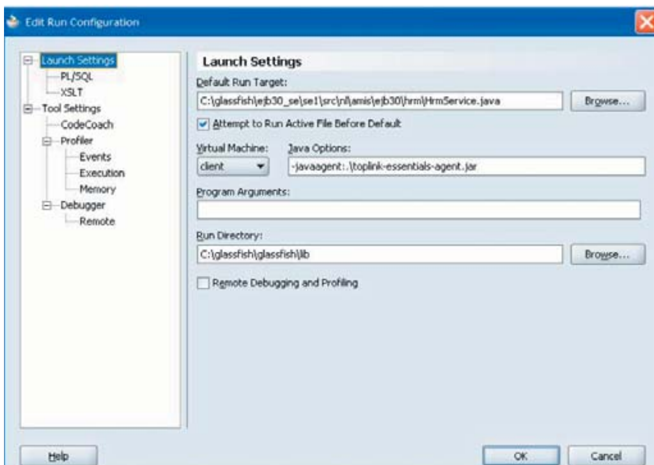
- Download JDeveloper 10.1.3EA van OTN en unzip
- Download GlassFish Reference Implementation van GlassFish Homepage en unzip
- Start JDeveloper 10.1.3EA en creëer een nieuwe Application Workspace en een nieuw Project, bijvoorbeeld ejb30
- Ga naar de project properties van het nieuwe project. Ga naar de Libraries tab. Voeg de jar-files Javaee.jar, Toplink-essentials-agent.jar en Toplink-essentials.jar uit de GlassFish download toe aan het project. Voeg ook de Oracle JDBC library toe – om de Oracle JDBC driver te kunnen gebruiken (zie afbeelding 2).
- Ga naar de node Run/Debug, selecteer de Default Configuration en klik op de Edit button. Zet de Virtual Machine op Client. Dit heeft te maken met de javaagent optie die we nu gaan zetten. Typ in het Java Options veld: -javaagent:.\toplink-essentials-agent.jar. Type in het veld Run Directory de $Glassfish_Home\lib directory.

Nu kun je aan de slag met het ontwikkelen van de Entities, persistence.xml file en de applicatie zelf die de EntityManager gaat gebruiken voor persistentie-services. Je kunt natuurlijk ook de wizards van JDeveloper gebruiken om razendsnel Entity's te laten genereren op basis van Tabel-definities. Gebruik daarvoor de wizard New, Business, EJB, CMP Entity Beans from Tables.

Deze wizard creëert de Entity's, compleet met Annotaties, op basis van Tabellen of Views. Je kunt zelfs een BusinessService class genereren door een Session Bean te laten aanmaken door de wizard. Het resultaat moet je wel enigszins verfijnen om buiten de EJB Container te gebruiken.



Afbeelding 2. In de Libraries tab worden de jar-files en de Oracle JDBC-library toegevoegd.



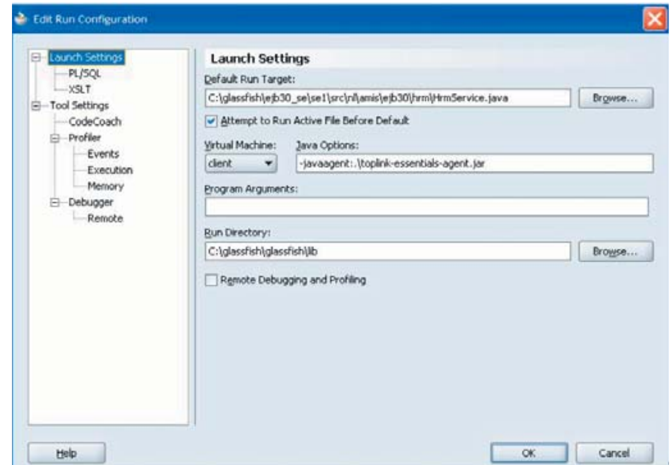
Afbeelding 3. De VM wordt op Client gezet in verband met de javaagent-option die we gebruiken.

Conclusies

Eendracht maakt macht. EJB 3.0 Persistence is in ogenschijnlijke grote eendracht ontstaan en heeft ook alles in zich om het veld van Object Relational Mapping samen te binden en te convergeren. EJB 3.0 Persistence is simpel, elegant, snel te leren en biedt vrijwel alle essentiële faciliteiten die we van ORM-frameworks hebben leren verwachten.

De EJB 3.0 Persistence API bevat nog niet alles wat we nodig hebben. Er zit geen voorziening in voor data caching en zoiets simpels als 'Refresh after Insert' om de effecten van database triggers en default values te synchroniseren is niet eenvoudig te realiseren. In veel gevallen zal dan ook de Referentie Implementatie GlassFish niet afdoende zijn en zal een leverancierspecifieke implementatie worden gekozen die onherroepelijk enige afhankelijkheid introduceert die de overdraagbaarheid van de applicatie aantast.

EJB 3.0 Persistence is meer dan veelbelovend. Daarnaast is het leuk om mee te werken. Ik zou dus iedere Java-ontwikkelaar willen aanraden er eens mee te gaan spelen. Wellicht dat de aanwijzingen en voorbeelden in dit artikel daarbij behulpzaam kunnen zijn.



Afbeelding 4. Via de CMP Entity Beans from Tables creëert de Entity's, compleet met Annotaties, op basis van Tabellen of Views.

Bronnen

- JSR-220 EJB 3.0 Specification - www.jcp.org/en/jsr/detail?id=220
- Project GlassFish Homepage – de JEE 5 Referentie Implementatie – https://glassfish.dev.java.net/
- Oracle JDeveloper 10.1.3EA met ondersteuning van EJB 3.0 - www.oracle.com/technology/tech/java/ejb30.html
- Using GlassFish Reference Implementation of EJB 3.0 Persistence with JDeveloper 10.1.3EA - http://technology.amis.nl/blog/?p=964
- Verscheidene weblog-artikelen over EJB 3.0 Persistence op de AMIS Technology Weblog, over onder meer Relaties, Version en Optimistic Locking en Geavanceerd EJB QL: http://technology.amis.nl/blog/index.php?s=ejb+3.0

Lucas Jellema (jellema@amis.nl) is sinds 2002 werkzaam bij AMIS Service in Nieuwegein, als Expertise Manager Technologie en Technisch Consultant. Daarvoor werkte hij ruim acht jaar bij Oracle, ondermeer binnen het iDevelopment Center of Excellence. Hij houdt zich onder meer bezig met Java, XML/XSLT en andere webtechnologie als ook de Oracle-database en tools voor applicatie-ontwikkeling.