In zijn artikel *Enterprise JavaBeans 3.0 komt eraan!* vertelt Bert Ertman de historie en achtergronden van de totstandkoming van EJB 3.0. Een belangrijk onderdeel van EJB 3.0 is de Persistence API, die zowel binnen als buiten EJB containers het contract beschrijft volgens welk Java-applicaties met databases kunnen communiceren. In dit artikel beschrijft Lucas Jellema in detail hoe je concreet met die EJB 3.0 Persistence API aan de slag kunt gaan.

Aan de slag met de EJB 3.0 Persistence API

Toepassing in een J2SE-applicatie

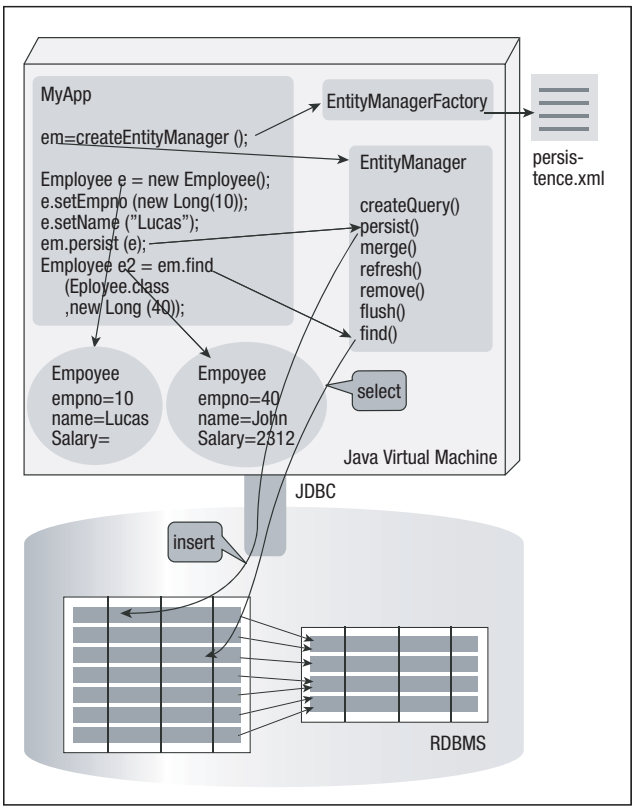
Aan de hand van de Glassfish Open Source Reference Implementatie wordt een toepassing in een J2SE-applicatie geschetst, maar alle voorbeelden en beschrijvingen gaan onverkort op voor Oracle Toplink (dat aan de basis ligt van Glassfish Persistence), JBoss Hibernate 3.0 en BEA (sinds december 2005) Kodo – de drie belangrijkste implementaties van EJB 3.0 Persistence.

JSR 220 - EJB 3.0 De EJB 3.0 Persistence specificatie bestaat uit drie hoofdcomponenten:

- Object Relational Mapping Meta Data – via annotations en/of een XML-file
- De EJB QL 3,0 Query Language
- De PersistenceManager – de centrale component om ORM Services aan te roepen

Om de Persistentie-services te benaderen moet een Java-applicatie allereerst een EntityManager instance in handen krijgen. Voor een J2SE-applicatie buiten de EJB-container gaat dat als volgt: EntityManager is een interface, via een EntityManagerFactory en een klein XML-configuratiebestand – persistence.xml – waarin ondermeer de database-connectie staat gespecificeerd – wordt een concrete implementatie, bijvoorbeeld GlassFish, TopLink of Hibernate, verkregen.

Op deze EntityManager kunnen vervolgens query's worden aangevraagd of Entities gecreëerd of geupdate. De applicatie werkt met POJOs als Entities, gewone Java Beans, die vrij zijn van afhankelijkheden van EJB 3.0 classes of interfaces. Hiervoor is het alleen vereist dat de EntityManager weet hoe de POJOs en hun properties en relaties vertaald kunnen worden in tabellen met kolommen en foreign keys. Deze informatie wordt vastgelegd met de Object Relational Mapping meta-data, normaalgesproken door middel van Annotaties in de POJOs.



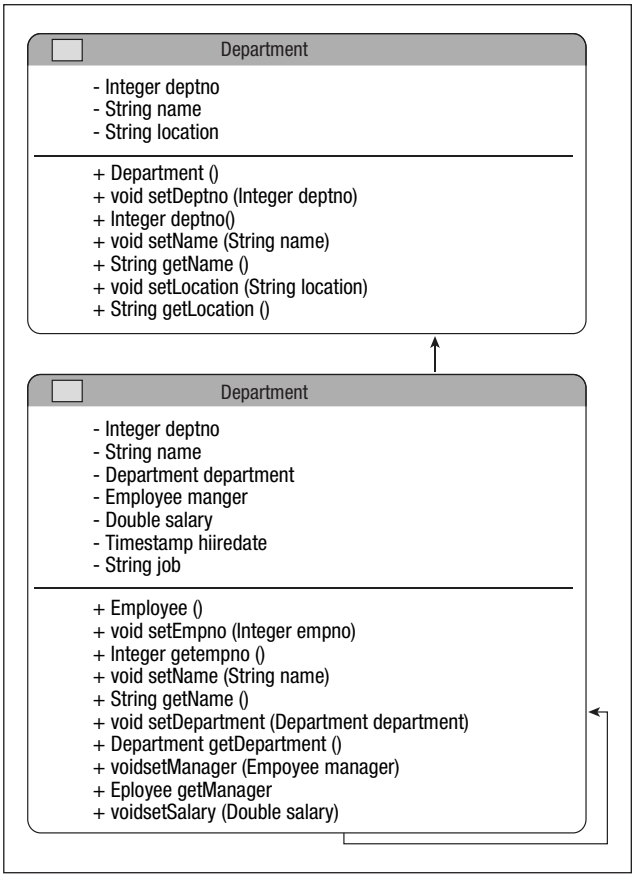
Afbeelding 1. EJB 3.0 Persistence API.

EJB 3.0 Persistence in J2SE-applicatie

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 Entities, Employee en Department geheten. De Employee Entity is een gewoon Java Object, met properties als Empno, Name, Job, Manager en Department en bijbehorende getter en setter methoden:

Als je naar de code kijkt voor deze Java Classes wordt opeens duidelijk hoe Annotations aan de EntityManager aangeven hoe deze Entities gerelateerd zijn aan database-objecten:

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

import ...

@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;
    }
}
```



Afbeelding 2. De Employee Entity is een Java-object met property's en bijbehorende getter en setter-methoden

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, pesistent'-property's zijn die aan een database-column zijn gekoppeld. Als de naam van het property afwijkt van de naam van de kolom kan met (name="COLUMN_NAME") de juiste koppeling worden vastgelegd. De annotatie @Id tenslotte 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. Employee.java luidt:

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

import ...

@Entity
@Table(name="EMP")
```

```
public class Employee {

    private Integer empno;
    private String name;
    private Department department;
    private Employee manager;
    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 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 enzovoorts. 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 object verzameling. 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 dit property wordt gespecificeerd als een OneToMany, een referentie naar een verza-

meling van nul, een 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 code fragment:

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

import ...

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.createEntity
Manager();

  // voorbeeld van het creeren van nieuw
Department
  Department dept = new Department();
  dept.setDeptno(60);
  dept.setName("JAVA MAGAZINE");
  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 van standaard services van de EntityManager: persist() en merge(). Andere services zijn refresh(), remove(), find() – op primary key – flush() en createQuery(). 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', 'JAVA MAGAZINE')
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 configuratiefile – 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 Entities 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 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 entities aan de applicatie teruggegeven. Hier volgt 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());
  }
```

Uiteraard kan dat nog wat interessanter. Bijvoorbeeld een query waar het gebruik van beans, property's en relaties aardig naar voren komt: 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();
```

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();
```

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. Query's kunnen dynamisch worden geconstrueerd, zoals hierboven gedemonstreerd. Maar query's kunnen ook voorgedefinieerd worden, met gebruik van Annotaties. Bijvoorbeeld de volgende annotatie in de Employee class die alle Employees met die de job Salesman hebben 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").getResultList();
}
```

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 we kunnen spreken van een krachtige query-taal – met de mogelijkheid van Native Queries 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 entities in relationele data.

Conclusie 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: browse naar http://technology.amis.nl/blog/index.php?s=ejb+3.0+download+jdeveloper en kies een van de weblog-artikelen die voor je favoriete IDE beschrijven hoe je met de Glassfish Reference Implementatie een project configureert en met een helloEJB30PersistenceWorld applicatie op stoom kunt komen. 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.

Bronnen

- JSR-220 EJB 3.0 Specification - http://www.jcp.org/en/jsr/detail?id=220
- Project GlassFish Homepage – de JEE 5 Referentie Implementatie – https://glassfish.dev.java.net
- Verscheidene Weblog-artikelen over EJB 3.0 Persistence op de AMIS Technology Weblog, over ondermeer Relaties, Version en Optimistic Locking, Geavanceerd EJB QL etc.: 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.