Enterprise

'Never a dull moment' in de wereld van Java. Op die regel zal 2009 ook zeker geen uitzondering zijn. Eén van de ontwikkelingen die al enige tijd zijn schaduw vooruit werpt en in de loop van 2009 tot voorlopig hoogtepunt komt is JEE 6, de nieuwste release van de Java Enterprise Edition. Java Magazine besteedt dit hele jaar aandacht aan de belangrijke componenten in JEE 6 - met in dit nummer de schijnwerper op JPA 2.0, de tweede release van de Java Persistence Architecture.

Het Jaar van JEE 6 (2)

JPA 2.0: Rentree in de tweede versnelling

Lucas Jellema
is Java & SOA specialist bij AMIS.
(lucas.jellema@amis.nl)

JPA als specificatie kent een historie sinds pakweg 2005. De ideeën achter en concepten onder JPA zijn al veel ouder en dateren zelfs van voor het ontstaan van Java zelf. Het terrein waar JPA over gaat is Java Persistence met als kernvraag: 'Hoe werkt Java - een Object geOriënteerde taal - samen met databases, over het algemeen niet OO maar relationeel van karakter'. Data in Java objecten is vaak afkomstig uit een bron die deze data persistent - over de levensduur van de Java applicatie en de JVM heen - vasthoudt en bovendien ook aan andere afnemers dan de Java-applicatie geeft. Oorspronkelijk was die bron meestal een relationele database die vanuit Java via JDBC werd benaderd - overigens is die bron de laatste jaren in toenemende mate ook een WebService waarmee een meer XML-gebaseerde interactie wordt aangegaan.

De communicatie met de database via JDBC kent verschillende vertaalslagen met elk hun eigen uitdaging - bijvoorbeeld van de wereld van de Java-programmeertaal naar de wereld van SQL en bijbehorende data types, van Objecten naar Relationele records en van vluchtig in vele parallelle sessies naar persistent en single-point-of-truth. Voor die vertaalslagen zijn door de jaren heen allerlei hulpmiddelen ontwikkeld - de Object Relational Mapping frameworks waarvan TopLink de eerste en Hibernate waarschijnlijk de meest bekende is, naast pragmatische deeloplossingen als iBatis en Spring JDBC. Hoewel al die verschillende oplossingen voor een gemeenschappelijke uitdaging elk een eigen bijdrage leverden die in specifieke omstandigheden de grootste meerwaarde had, was het ongelukkig dat ontwikkelaars zich voor elk volgend project een nieuw ORM persistence framework eigen moesten maken. De overkoepelende J(2)EE oplossing - EJB - werd door de groeiende complexiteit in steeds minder gevallen toegepast.

JEE 5 zorgde in 2006 voor een ommekeer - zowel voor de complexiteit van EJB 2.x als voor de wildgroei in uiteenlopende ORM frameworks. De EJB 3.0 specificaties werden op een veel pragmatischer manier ingericht, werden de vriend van de ontwikkelaar in plaats van haar tegen te werken. Het simpele adagium 'configuration by exception' - alleen configureren wat afwijkt van de default én zorgen voor simpele logische defaults - maakte alleen al een wereld van verschil. De inzet van Java 5 Annotaties maakt de nog resterende configuratie nog eenvoudiger. De insteek was ook sterk pragmatisch; ervaring en best practices met allerlei frameworks werd nadrukkelijk ingebracht bij de ontwikkeling van de nieuwe specificaties. De inbreng van de teams achter Hibernate en TopLink bij de totstandkoming van Java Persistence API (JPA) - de ORM motor binnen EJB 3.0 - is een goed voorbeeld daarvan. Omdat er voor gekozen is om de EJB specificatie te scheiden van de JPA specificatie ontstond - bijna als bijproduct van de herinrichting van EJB 3.0 - een specificatie voor Java Persistence die niet alleen binnen JEE containers voor EJBs kon worden gebruikt, maar die ook buiten de applicatie server in stand-alone Java applicaties of web applicaties kon worden ingezet.

JPA (1.0) bracht de vele persistency (Object Relationele) frameworks bij elkaar achter een generieke interface. Ontwikkelaars die leren werken met JPA op basis van EclipseLink kunnen hun kennis ook toepassen bij JPA op basis van de implementaties van bijvoorbeeld Hibernate of Kodo. Bovendien is met JPA de link gelegd tus-

sen de complexe wereld van Enterprise Java Beans inclusief gedistribueerde Entities en Container Managed Persistence en de meer lichtgewicht Bean Managed Persistence die database persistency biedt voor Servlet/JSP gebaseerde toepassingen of stand alone Java applicaties.

JPA 1.0 werd in 2006 vrijgegeven compleet met een referentie implementatie: Toplink Essentials, gebaseerd op het toen nog commerciële (Oracle) TopLink product dat al in 1993 ontwikkeld was als ORM-framework voor Smalltalk-omgevingen. Bekende ORM-producten kwamen snel met hun eigen implementaties van JPA, aangevuld met specifieke functionaliteit voor ondermeer tuning en tracing, caching, functies aggregatie en andere verrijkingen voor JPQL - de querytaal. JPA implementaties zijn ondermeer Hibernate, TopLink - inmiddels omgedoopt tot EclipseLink en gedoneerd aan de open source gemeenschap, OpenJPA (gebaseerd op Kodo, voorheen van BEA, nu dus Oracle), Apache Cayenne en DataNucleus (gebaseerd op JPOX voor JPA).
Het succes van JPA 1.0 was enorm. In een paar jaar tijd is het in veel projecten en organisaties vanzelfsprekend geworden dat de communicatie tussen de Java-applicatie en de database verloopt via JPA. Al wordt vastgehouden aan de al langer gebruikte frameworks zoals Hibernate of TopLink, ook dan is het een 'best practice' om de JPA API te hanteren. Bijvoorbeeld om een eventuele overstap naar een ander onderliggende JPA-implementatie te vergemakkelijken, maar ook om ontwikkelaars met JPA-kennis in het project te kunnen inzetten zonder inwerktijd in het specifieke framework en te profiteren van andere hulpmiddelen die voor JPA beschikbaar zijn zoals Dali - IDE plugins voor declaratief, visueel en met wizards ondersteund JPA mappings te definiëren -, OpenXava - een JPA gebaseerde applicatie generator - en het JPA Query Tool - een JPQL query editor.
Ondanks het succes en de razendsnelle adoptie van JPA is er flinke ruimte voor verbetering van de specificatie. De JPQL query-taal is bijvoorbeeld lang niet zo uitgebreid als de vergelijkbare querytalen in ORM frameworks. Ook vroegen de leveranciers van JPA implementaties om meer aanknopingspunten voor bijvoorbeeld hun oplossingen voor caching en locking faciliteiten. De flexibiliteit in de mapping tussen tabellen en objecten liet nogal wat te wensen over - nogal wat OO concepten zoals embedded objects- konden eigenlijk niet goed binnen JPA worden toegepast of vroegen kunstgrepen die het domeinmodel of het database ontwerp vervuilden.

JPA 2.0 alias JSR 317

Binnen de scope van JEE 6 ging een nieuwe werkgroep aan de slag met de JPA 2.0 specificatie onder de noemer JSR-317. Het doel van de Java Persistence 2.0 API is vooral om de reikwijdte van de API te vergroten, op bovenstaande verbeterpunten in te gaan en daarmee features in te bouwen waar de Java-community in de afgelopen jaren om heeft gevraagd. De non-portabiliteit moet worden teruggedrongen, ondermeer door standaardisering van de optionele (door leverancier inplugbare functionaliteit).

Al sinds JSpring 2007 en JavaOne 2007 wordt gepresenteerd en gesproken over de nieuwe functionaliteit van JPA 2.0. In 2008 ontstond steeds meer duidelijkheid en op 31 oktober 2008 verscheen de Public Review Draft van de specificatie en op 15 december eindigde de Public Ballot. Op dit moment is ondermeer de implementatie gaande van de JPA 2.0 features in de referentie implementatie - EclipseLink 2.0 - die beschikbaar moet zijn alvorens de specificatie definitief wordt.

O/R Mapping en Domain Modeling

JPA 1.0 kende een flink aantal vormen van mapping - inclusief bijvoorbeeld de vrij geavanceerde ManyToMany mapping - maar had tegelijkertijd een aantal beperkingen en tekortkomingen. Op dat vlak brengt JPA 2.0 veel verbeteringen, waarvan je er ook een heleboel waarschijnlijk nooit zal gaan gebruiken.

JPA 2.0 beschrijft diverse nieuwe mappings van collections. De meest eenvoudige is de verzameling van basic types - String, Date, Integer - naar een bepaalde kolom in een afhankelijke tabel. Bijvoorbeeld met de mapping annotatie ElementCollection, ondersteund met CollectionTabel om de bron tabel van de collectiewaarden aan te duiden en Column om de kolom te specificeren waaruit de waarde wordt opgevraagd, in de Department entity wordt beschreven hoe de eigenschap staff wordt bepaald uit de waarde van de ENAME kolom in de tabel EMP voor alle rijen die via een DEPTNO gelijk hebben aan de primary key van het Department.

```
    @ElementCollection
          @CollectionTable(name="EMP", joinColumns = @
JoinColumn(name="DEPTNO"))
        @Column(name="ENAME")
    private List<String> staff = new ArrayList();
```

De mapping naar een ElementCollection hoeft niet een basis type te betreffen maar kan ook een over een verzameling Embeddable objecten gaan. Als de Employees een of meerdere vaardigheden kunnen hebben, vastgelegd in een detail tabel SKILLS met foreign key emp_id naar de primary key van Employee, zou de entity Employee de volgende inhoud kunnen bevatten:

```
public class Employee implements Serializable {
    @ElementCollection
```

Ondanks het succes en de snelle adoptie van JPA is er ruimte voor verbetering

Het direct uitvoeren van queries om entiteiten op te vragen is van grote betekenis

```
    @CollectionTable(name = "SKILLS",
                joinColumns = @JoinColumn(name = "EMP_
ID"))
       @AttributeOverride(name  =  "since",  column  =  @
Column(name = "YEAR_SINCE")
    )
    protected Set<Skill> skills;
```

Skill is een Class die niet zelfstandig als entity naar een tabel is gemapped. Wel kunnen Skill instances geinstantieerd worden in de context van een Employee. Skill is geannoteerd met de @Embeddable annotatie:

```
    @Embeddable
    public class Skill {
```

Met deze definities kunnen nieuwe Skill records in de database worden aangemaakt via de Skills set op de Employee entiteit:

```
          Employee   man  =  em.find(Employee.class,  new
Long(7698));
        Skill sk = new Skill();
        sk.setRating(new Long(4));
        sk.setSince(1981);
        sk.setSkill("LAWNMOWING");
        man.getSkills().add(sk);
        em.getTransaction().commit();
```

Het SQL Statement dat wordt uitgevoerd:

```
INSERT INTO SKILLS (EMP_ID, SKILL, YEAR_SINCE, RATING)
VALUES (?, ?, ?, ?)
        bind => [7698, LAWNMOWING, 1981, 4]
```

Embeddable classes kunnen ook zelf meer als state attributen hebben van classes die zelf embeddable zijn. Embeddable classes kunnen ook relaties hebben - alle vier types zijn toegestaan.

Een interessante vorm van collection mapping is ook nog de @ElementCollection annotatie op een Map die bijvoorbeeld entries heeft voor child records met een vanuit het kind gerefereerde entiteit als sleutel en een veld uit het kind als waarde. Stel dat Employees Allocaties hebben met een Project referentie en een rol attribute. De mapping kan er als volgt uitzien:

```
@Entity
public class Employee{
…
@ElementCollection
Map <Project, rol> allocaties
…
}

@Entity
@Table(name = "PROJECTS")
Public Class Project
```

In JPA 2.0 is ook meer aandacht voor de ordening van elementen. Zo kunnen we in geval van een List met detail records (op basis van een OneToMany mapping) met de nieuwe annotatie @OrderBy aangegeven op welk attribuut de child records gesorteerd moeten worden bij instantiatie van de lijst.

```
    @OneToMany(mappedBy = "department")
        @OrderBy("ename")
    private List<Employee> employeeList;
```

Met @OrderColumn gaan we nog een stap verder in het beheer van de volgorde van gegevens en kunnen we JPA een kolom laten bijhouden waarin de index-waarde voor een rij is vastgelegd. Als in de Collection een entity wordt verplaatst, toegevoegd of verwijderd moet de JPA provider zorgen voor een update van de onderliggende kolom voor alle betrokken rijen. Een nieuwe entity die vooraan in de lijst wordt geplaatst kan dus een update veroorzaken van een flink aantal rijen. Onderstaande OrderColumn annotatie laat JPA de volgorde van de employees in de list vertalen naar de kolom importance_in_department_index - waarden 0 of 1 en hoger.

```
    @OneToMany(mappedBy = "department")
        @OrderColumn("importance_in_department_index")
    private List<Employee> employeeList;
```

Andere uitbreidingen op het vlak van mappings zijn ook de eenzijdige One-to-Many mapping - met alleen een annotatie in de parent en niet in de child entiteit. Relaties kunnen deel uitmaken van de primary key van een entiteit. De optie Automatic Orphan Deletion kan worden gebruikt om bij verwijdering van een entiteit alle logisch, via one-to-one en one-to-many aanhangende entiteiten ook te laten verwijderen.

Query faciliteiten

Naast mapping is het direct kunnen uitvoeren van specifieke queries om entiteiten op te vragen van grote betekenis. JPA gebruikt daarvoor de JPQL taal, die wel wat wegheeft van SQL maar strict Object (Entity) georiënteerd is. Een paar leuke uitbreidingen van JPQL zijn ondermeer:

Het kunnen gebruiken van time en date literals in queries:

```
@NamedQuery(name = "Employee.experiencedStaff", query =
"select o from Employee o where o.hiredate  < '1981-05-
25'")
```

Ondersteuning van de non-polymorphic query waarbij je het subtype van een entiteit als query criterium kan meenemen:

```
SELECT e FROM Employee e
WHERE CLASS(e) = FullTimeEmployee
OR e.wage = "SALARY"
```

Het gebruik van CASE expressies in queries en DML operaties:

```
update Employee e
Set    e.salary = e.salary *
                  case e.job
                  when 'CLERK' then 2
```

```
                when 'SALESMAN' then 1.5
                when 'MANAGER' then 1.01
                else 1.1
                end
```

Naast CASE zijn ook NULLIF en COALESCE nieuw ondersteunde functies - ook bekend uit de ANSI SQL standaard.

Het toepassen van een Collection als Named Parameter:

```
@Entity
@NamedQueries({
  @NamedQuery(name = "Employee.findAll", query = "select
o from Employee o")
, @NamedQuery(name = "Employee.findInJobs", query =
"select o from Employee o where o.job in :jobs")
})
public class Employee {
```

Op basis van bovenstaande query definitie kunnen we met onderstaande code de query gebruiken om bijvoorbeeld alle medewerkers te verkrijgen die als Job de waarde CLERK of SALESMAN hebben:

```
Query findEmpInJobQuery = em.createNamedQuery("Employee.
findInJobs");
findEmpInJobQuery.setParameter("jobs", new String[]
{"CLERK","SALESMAN"});
List<Employee> emps = findEmpInJobQuery.getResultList();
```

Query Criteria API

Een zeer veel gevraagde uitbreiding voor JPA 2.0 is de ondersteuning van een Query Criteria API - een methode om dynamisch queries op te bouwen zonder string manipulatie te hoeven doen, in kleine, door de compiler gecontroleerde stappen, met gebruik van gestructureerde, herbruikbare, objecten als bouwstenen voor de query. Zowel EclipseLink (voorheen TopLink) als Hibernate hebben een Criteria API en nu wordt deze ook in JPA opgenomen. Een Query object wordt opgebouwd vanuit een QueryBuilder factory. Een QueryDefinition object wordt voor een bepaalde root-entiteit aangemaakt. Op dat object kunnen vervolgens de onderdelen van de query worden aangemaakt, zowel de select items als de where clause en ook de having, group by en order by onderdelen. Daarnaast kunnen join condities - zowel een gewone join als een left (outer) join - aan het Query Definition object worden toegevoegd.

Dit voorbeeld toont het opbouwen van een query om alle employees te verzamelen met als Job de waarde SALESMAN.

```
                QueryBuilder queryBuilder = em.
getQueryBuilder();
              DomainObject employee = queryBuilder.
createQueryDefinition(Employee.class);

    employee.select(employee).where(employee.get("job").
equal("SALESMAN"));

        Query allSalesmen = em.createQuery(employee);
        List<Employee> salesmen = allSalesmen.
getResultList();
```

Deze query komt overeen met de JPQL query: select e from employee e where e.job = 'SALESMAN'.

Run Time APIs

Een betrekkelijk klein punt maar wel een voorbeeld van de portabiliteit die wordt nagestreefd is het gebruik van generieke properties zoals javax.persistence.jdbc.url, javax.persistence.jdbc.user en javax.persistence.logging.level in persistence.xml, in plaats van implementatie-specifieke properties als toplink.jdbc.url, hibernate.connection.password en eclipselink.persistence.logging.

Met em.getSupportedProperties() kan van de EntityManager een lijst van alle ondersteunde properties en hun exacte spelwijze worden opgevraagd - een feature dat mij in het verleden veel tijd had kunnen schelen!

De JPA 2.0 specificatie stelt dat er integratie dient te zijn met de Validation API (JSR-303) voor bedrijfsregel validatie. De details van deze integratie zijn, zoals de EclipseLink Website stelt, "to be determined".

JPA 1.0 kent alleen Optimistic Locking, waarbij een exceptie wordt gegooid als op basis van Version attributes wordt vastgesteld dat de versie waarop een update wordt uitgevoerd niet meer de huidige versie in de database is. Pessimistic locking maakt gebruik van native database locking mechanismen die worden ingezet om een record te locken alvorens wijzigingen worden doorgevoerd. Deze vorm van locking geeft de locker meer zekerheid: de wijziging kan niet opeens tijdens commit alsnog vanwege locking-conflicten worden afgekeurd. Het nadeel is wel dat het aantal (mogelijke onnodige) database locks flink zou kunnen toenemen. JPA 2.0 geeft ons de gelegenheid om een Entity pessimistic te locken. In de simpelste vorm doen we dat met dit statement:

```
    Department d = em.find(Department.class, new
Long(10));
    em.refresh(d, LockModeType.PESSIMISTIC); // of: em.
lock(d, LockModeType.PESSIMISTIC)
```

Als het lock verkregen wordt kan alleen de huidige, open transactie een wijziging van de entiteit doorvoeren. Het lock vervalt na een commit of rollback van de transactie. Als het niet lukt om een lock te verkrijgen - omdat een andere transactie al een lock heeft - zal een PessimisticLockException worden gegooid. Als de entity die wordt gelocked ook onderhevig is aan optimistic locking policies - via het Version attribute - moet eerst gecontroleerd worden of de entity nog gesynchroniseerd is met de database alvorens het database lock mag worden uitgereikt. Als er geen synchronisatie is wordt een OptimisticLockException gegooid. De implementatie van het locking mechanisme kan verschillen per database en JPA implementatie. Met

Aan de slag

Je kunt al wat vingeroefeningen met JPA 2.0 doen met een nightly build van EclipseLink, te vinden op: http://www.eclipse.org/eclipselink/downloads/nightly.php. Ten tijde van het schrijven van dit artikel kon pakweg 40% van de nieuwe JPA 2.0 features al daadwerkelijk worden uitgeprobeerd. Aan de rest werd nog gesleuteld. De public draft van de JPA 2.0 specificatie kan je terugvinden op http://www.jcp.org/en/jsr/detail?id=317 - hierin vind je ook veel voorbeelden van toepassing van JPA 2.0 functionaliteit.

De code voorbeelden die in dit artikel zijn opgenomen kun je terugvinden om zelf ook te proberen op http://technology.amis.nl/blog/5131/first-trials-with-jpa-20-and-the-eclipselink-preview.

EclipseLink Essentials (JPA 2.0 Preview) tegen een Oracle RDBMS levert bovenstaand statement onderstaande SQL op:

```
SELECT DEPTNO, DNAME, LOC FROM DEPT WHERE (DEPTNO = ?)
FOR UPDATE
```

De hint javax.persistence.lock.timeout kan worden gebruikt om een timeout aan te geven - de periode waarbinnen het lock verkregen moet zijn. De specificatie kent nog een onduidelijk over de vraag of de timeout in seconden of miliseconden moet worden aangegeven.

Caching is een essentieel onderdeel van een schaalbare Java persistence oplossing. Niet alleen een sessie-cache - die door de EntityManager binnen JPA min of meer wordt geïmplementeerd - maar een cache over de threads en sessies heen, op het niveau van de JVM - ook wel een L(evel)2 cache genoemd. Een cache die ervoor zorgt dat een entity niet meerdere keren in het geheugen van de JVM hoeft te worden geladen, en waardoor een update op het moment van commit direct voor alle sessies beschikbaar is. JPA 2.0 voegt niet een L2 cache toe, maar houdt er wel rekening mee en beschrijft een basis API voor de cache als de gebruikte JPA implementatie een L2 cache bevat tussen persistence context en database. Deze API omvat de methodes contains(entityClass, primaryKey), evict(entityclass[, primaryKey]) en evictAll().

Aan de slag

De definitieve publicatie van JPA 2.0 wordt verwacht vlak voor JavaOne 2009 - begin juni. Tegen die tijd zal ook de EclipseLink referentie implementatie compleet zijn.

Conclusie

JPA 2.0 is geen revolutie, wel een zeer welkome evolutie van de JPA standaard. De portabiliteit van applicaties tussen verschillende JPA implementaties wordt verder vergroot door de uitbreiding van JPA met features die tot nu toe alleen op een leverancier specifieke manier toe te passen waren. De mappingfaciliteiten zijn sterk uitgebreid en bieden aanzienlijk meer flexibiliteit dan voorheen.
De nieuwe Query API lijkt erg op de manier van opbouwen van queries in ondermeer Hibernate en TopLink en is voor veel ontwikkelaars wellicht makkelijker en beheersbaarder te gebruiken dan JPQL. Al met al zijn er eigenlijk geen goede redenen meer te bedenken om niet-triviale Java persistency uitdagingen zonder JPA te lijf te gaan. Gebruik je Hibernate of TopLink/EclipseLink, zorg dan dat je die frameworks via de JPA interface benadert en niet rechtstreeks. «