**Sinds de release van de Java Persistence API (JPA) als onderdeel van Enterprise JavaBeans 3.0 (EJB3) in mei 2006 heeft het gebruik ervan een grote vlucht genomen. Al snel na het uitkomen van de standaard verschenen er verschillende implementaties. In dit artikel kijken we naar EclipseLink, één van de implementaties. We kijken vooral naar de extra features, die verder gaan waar de standaard ophoudt.**

# EclipseLink: Moet je ook eens proberen

**Bart Kummel**
is consultant en trainer bij Transfer Solutions
Email: bkummel@transfer-solutions.com

Het enthousiasme over EJB3 in het algemeen en JPA in het bijzonder is terecht. Bij de ontwikkeling van EJB versie 3 heeft de nadruk vooral gelegen op de bruikbaarheid vanuit het perspectief van ontwikkelaars, en dat is te merken.
Het meest in het oog springt het gebruik van annotaties als alternatief voor XML-configuratiebestanden[1]. Al met al biedt de JPA eindelijk een gestandaardiseerde oplossing voor Object/Relational Mapping (ORM) die ook praktisch bruikbaar is. Toch blijkt - zoals vaak met standaarden - dat de standaard soms net te beperkt is. De verschillende implementaties kennen daarom hun eigen uitbreidingen op de standaard. Die kunnen we hier niet allemaal behandelen, maar voor de volledigheid wel een overzicht van een aantal belangrijke JPA-implementaties:

- TopLink Essentials, de referentie-implementatie[2] van JPA 1.0. Onderdeel van het Glassfish-project. Zie https://glassfish.dev.java.net/. Glassfish is het open source project waarbinnen de Sun Java System Application Server wordt ontwikkeld. Glassfish doet tevens dienst als referentie-implementatie van de JEE standaard.
- Hibernate is onderdeel van de JBoss applicatieserver van Red Hat. Ten tijde van de oudere EJB versies - die voor ontwikkelaars een stuk lastiger in gebruik waren - was Hibernate een van de meestgebruikte alternatieven. Sinds versie 3.2[3] voldoet het ook aan de JPA-standaard. Zie http://www.hibernate.org/397.html.
- OpenJPA, de JPA-implementatie van het Apache-project. Gebaseerd op de sourcecode van Kodo, die BEA Systems in 2006 doneerde aan Apache[4]. Zie http://openjpa.apache.org/. De interessante vraag is natuurlijk wat er met Kodo en OpenJPA gaat gebeuren nu BEA Systems is overgenomen door Oracle. Het lijkt in ieder geval niet waarschijnlijk dat Oracle op de lange termijn twee JPA-implementaties gaat onderhouden...
- TopLink Essentials is een subset van het commerciële product Oracle TopLink[5]. Het commerciële product TopLink bevat al een heleboel nuttige extensies op de JPA-standaard. Hoewel TopLink een zeer goed, volwassen persistence framework is, wordt het - zelfs binnen de Oracle-wereld - relatief weinig gebruikt. Dit komt onder andere doordat Oracle binnen zijn Application Development Framework (ADF) standaard kiest voor ADF Business Components [2] als persistence Framework [3].

Buiten de Oracle-wereld zal ook een rol spelen dat er voldoende goede persistence frameworks zijn die gratis beschikbaar zijn. Misschien is dat wel de reden geweest dat Oracle in maart 2007 bekend maakte[6] dat het volledige TopLink-product open source gemaakt zou worden.
Oracle heeft ervoor gekozen de sourcecode van TopLink onder te brengen bij de Eclipse Foundation [6, `Oracle's Commitment to Eclipse']. TopLink zal daar verder worden ontwikkeld onder de naam EclipseLink. (In september 2007 schreven Jeroen van Wilgenburg en Peter Ebell al een artikel over EclipseLink in Java Magazine, dat een goed overzicht geeft van EclipseLink en zijn voorgeschiedenis. Binnenkort wordt de eerste productierelease van EclipseLink verwacht.
Dat is een mooi moment om eens even te kijken wat EclipseLink te bieden heeft naast de functionaliteit die standaard onderdeel is van de JPA. Daarbij kijken we alleen naar functies op het gebied van ORM, hoewel EclipseLink verder gaat dan dat. Omdat we vooral kijken naar uitbreidingen ten opzichte van de standaard JPA-functionaliteit, gaan we uit van enige basiskennis van de JPA.
In dit artikel zullen we enkele uitbreidingen van

EclipseLink op de JPA bekijken aan de hand van eenvoudige voorbeelden. Achtereenvolgens kijken we naar Pagination, Scrollable cursors, DatabasePlatforms, Relationship mappings en Converters. Tenslotte werpen we nog even een blik in de toekomst.

## Pagination / Scrollable cursors

Als in een (web)applicatie gebruik wordt gemaakt van tabellen om data weer te geven en/of gebruikers een record te laten selecteren, is een veel voorkomend probleem dat er teveel records in de database zijn om in één keer weer te geven.
Ook uit performanceoogpunt is het vaak niet wenselijk alle records uit een tabel in één keer op te halen. JPA biedt een eenvoudige oplossing hiervoor, die gebruik maakt van pagination [8, pagina 178]. Een eenvoudig voorbeeld van het ophalen van een page met medewerkergegevens is weergegeven in listing 1.

```
Listing 1: Eenvoudig pagination voorbeeld
1 Query q = em.createQuery("SELECT e FROM Employee
  e");
2 q.setFirstResult(firstResult);
3 q.setMaxResults(pageSize);
4 List<Employee> result = q.getResultList();
```

In dit voorbeeld zijn pageSize en firstResult respectievelijk het aantal op te halen records en het eerste record van de page. Deze code zouden we bijvoorbeeld kunnen gebruiken in een stateful bean die bijhoudt wat de huidige page is en methodes heeft om naar de volgende en vorige page te gaan. Deze aanpak is simpel en lijkt doeltreffend. Helaas wordt deze vorm van pagination niet door alle databases op een efficiënte wijze ondersteund. Het is ook mogelijk dat de database wel pagination op effciënte wijze ondersteunt, maar dat de JPA-implementatie niet weet hoe hij hier gebruik van kan maken.
Wat dit laatste betreft is goed te merken dat TopLink - en dus EclipseLink - uit de Oracle-stal komt. TopLink kan heel effciënt pagineren met een Oracle database, waarbij gebruik wordt gemaakt van specifieke Oracle-features die dit mogelijk maken. Voor andere databases is deze functionaliteit echter (nog) niet geïmplementeerd. Het is overigens wel mogelijk om dit zelf te implementeren, door een subklasse te maken van DatabasePlatform (zie verderop). Het valt natuurlijk te verwachten dat de ondersteuning van andere databases verder geoptimaliseerd zal worden nu EclipseLink een open source product is geworden.
Een andere oplossing voor het geschetste probleem is het gebruik van een scrollable cursor. Scrollable cursors worden door veel databases ondersteund. Helaas heeft de JPA (nog) geen voorzieningen voor het gebruik van scrollable cursors. EclipseLink heeft deze voorziening wel. Listing 2 geeft een voorbeeld dat dezelfde medewerkersgegevens als in het vorige voorbeeld ophaalt met een scrollable cursor.

```
Listing 2: Het openen van een cursor
1 ReadAllQuery query = new ReadAllQuery(Employee.
class);
2 query.useScrollableCursor();
3 ScrollableCursor cursor =
4 (ScrollableCursor) session.executeQuery(query);
```

Merk op dat gebruik wordt gemaakt van een speciaal ScrollableCursor object. ScrollableCursor is een onderdeel van de EclipseLink API. De klasse ScrollableCursor implementeert de ListIterator interface uit de Java API, dus itereren over de lijst van medewerkers gaat vrij simpel, zoals we zien in listing 3.

```
Listing 3: Itereren met een cursor
1 while (cursor.hasNext()) {
2 System.out.println(cursor.next().toString());
3 }
```

In dit voorbeeld halen we omwille van de eenvoud alle records op. Maar dit is natuurlijk de plek waar we normaal gesproken alleen het aantal records ophalen dat op een pagina past. De cursor houdt zelf bij waar hij gebleven is. Na gebruik moeten we de cursor nog sluiten. (listing 4.)

```
Listing 4: Het sluiten van de cursor
1 cursor.close();
```

Behalve de ScrollableCursor zien we in dit voorbeeld nog een aantal andere niet-JPA zaken. Er wordt bijvoorbeeld gebruik gemaakt van een ReadAllQuery in plaats van een Query. Dit is vereist om een ScrollableCursor te kunnen gebruiken. Makkelijk is dat we geen JPQL4-query hoeven te schrijven. Het volstaat om een Entity klasse mee te geven aan de constructor van ReadAllQuery om een standaard read all query te genereren. EclipseLink kent meer typen Query's, die allemaal geoptimaliseerd zijn voor een bepaalde taak.
Nadeel is dat de JPA EntityManager uitsluitend JPA query-objecten kan creëeren. Als gevolg daarvan is er ook geen getResultList() methode. De query weet immers niks af van een EntityManager. Vandaar dat we een Session-object nodig hebben om de query te kunnen uitvoeren. In een volledig EclipseLinkgeoriënteerd project, zonder JPA, kunnen we het eigen sessiebeheer van EclipseLink gebruiken. In een meer EJB3-georiënteerd project kunnen we de klasse JpaHelper gebruiken. Deze bevat een aantal static methodes die het makkelijker maken om uitgebreide EclipseLink-functies te combineren met standaard JPA-functionaliteit. (In dit specifieke geval kunnen we bijvoorbeeld de methode getServerSession(EntityManagerFactory emf) gebruiken.)

**Heel efficiënt pagineren met een Oracle database**

## DatabasePlatform

Binnen EclipseLink zorgt de interne klasse DatabasePlatform5 voor de implementatie van alle database-specifieke zaken. Omdat er nogal wat verschillen zijn tussen de verschillende databases, wordt standaard een aantal subklasses van DatabasePlatform meegeleverd, waarin de specifieke zaken voor de meestgebruikte databases worden geregeld. Zoals genoemd is het in EclipseLink ook mogelijk om zelf een subklasse van DatabasePlatform te implementeren. Zodoende krijgen we maximale controle over de gegenereerde SQL. We kunnen bijvoorbeeld zorgen voor een effciënte toepassing van pagination voor het gebruikte databaseplatform. Dit kan door de methode printSQLSelectStatement() te overriden. Deze methode zet elke EclipseLink expressie om in een SQL statement dat naar de database gestuurd kan worden. Hier kunnen we dus specifieke features van het databaseplatform gebruiken. Op zich is het implementeren van DatabasePlatform niet heel moeilijk, maar het vereist wel behoorlijk wat regels code, omdat we de SQL-string die naar de database gaat grotendeels in code moeten opbouwen. Daarom is dat minder geschikt om hier verder uit te werken. In de discussie op het TopLink forum van Oracle zijn meer informatie en een voorbeeld te vinden. Het implementeren van een DatabasePlatform alleen is niet genoeg. EclipseLink moet nog wel weten dat het een eigen implementatie moet gebruiken. Het standaardgedrag is dat EclipseLink bij de actieve JDBC-driver automatisch de best passende DatabasePlatform-implementatie zoekt, waarbij EclipseLink alleen kijkt naar de standaard meegeleverde implementaties. Om dit gedrag te overriden, moeten we het configuratiebestand persistence.xml aanpassen. In de properties-sectie van dit bestand moeten we een regel toevoegen, zie listing 5.

```
Listing 5: Een DatabasePlatform-subklasse aanmelden
in persistence.xml
1   <property   name="eclipselink.target-database"
value="..."/>
```

Op de puntjes vullen we de fully qualified name van de gemaakte klasse in. Deze kan gewoon in het package van het betreffende project zitten en hoeft dus niet te beginnen met org.eclipse.persistence of iets van die strekking.

## Relationship mapping

Een veel voorkomend probleem is dat we records standaard moeten filteren. Een voorbeeld hiervan is een klantenbestand, waar nooit records uit verwijderd worden. Ook al gaat een klant naar de concurrent, het record blijft in de database. Dit wordt gedaan om de referentiële integriteit van de database te waarborgen. In het dagelijks gebruik is het echter niet handig dat medewerkers steeds records van oude klanten tegenkomen. Daarom wordt gewerkt met een zogenaamde soft delete. Als een klant verwijderd wordt, wordt het record niet verwijderd, maar wordt het veld deleted op Y gezet.

Nu wil men in de te bouwen Java-applicatie voorkomen dat oude klantrecords opduiken. Dat kunnen we doen door in alle JPQL-query's een extra voorwaarde op te nemen: deleted = 'N'. Als er veel verschillende query's zijn, kan dat lastig worden.[6] Nog veel vervelender wordt het als er sprake is van relaties. Stel dat er bijvoorbeeld een one to many relatie is tussen vertegenwoordigers en klanten. Om alle klanten van een bepaalde vertegenwoordiger op te halen, zou dan een aanroep van vertegenwoordiger.getKlanten() afdoende moeten zijn. Hoe zorgen we er nu voor dat de geretourneerde List alleen klanten bevat die niet soft deleted zijn? In standaard JPA is hier geen goede oplossing voor. Maar ook hier biedt EclipseLink een elegante oplossing. Het is namelijk mogelijk om de standaard joins, die automatisch gemaakt worden als er bijvoorbeeld een @OneToMany annotatie gebruikt wordt, aan te passen. Hiervoor moeten we de interface DescriptorCustomizer[7] implementeren. Deze interface schrijft één methode voor, met de naam customize. Met behulp van een ExpressionBuilder[8] kunnen we in deze methode een expressie bouwen die gebruikt wordt in de where clause van de automatisch gegenereerde join.[9] In het voorbeeld van de vertegenwoordiger met zijn klanten kan het er uitzien als in listing 6.

```
Listing 6: Het implementeren van
DescriptionCustomizer
1  public class VertegenwoordigerDescriptor
2  implements DescriptorCustomizer {
3
4  public void customize(ClassDescriptor
   descriptor) {
5  OneToManyMapping mapping = (OneToManyMapping)
6  descriptor.getMappingForAttributeName("klanten");
7  ExpressionBuilder builder = new
   ExpressionBuilder();
8  mapping.setSelectionCriteria(
9  builder.getField("klanten.vertwId")
10 .equal(builder.getParameter("id"))
11 .and(builder.getField("klanten.deleted")
12 .equal("N")));
13 }
14 }
```

We zien in dit voorbeeld dat een expressie wordt opgebouwd die overeenkomt met WHERE klanten. vertwId = vertegenwoordigers.id AND klanten. deleted = 'N'. Deze expressie vervangt dus de standaard expressie, die normaal gesproken overeen zal komen met alleen het gedeelte vóór de AND uit de voorgaande expressie. Om te zorgen dat de gemaakte customizer ook daadwerkelijk gebruikt wordt, moeten we deze nog 'aanmelden' in het persistence.xml configuratiebestand, zie listing 7.

```
Listing 7: Een customizer aanmelden
1 <property name="eclipselink.descriptor.
  customizer.Vertegenwoordiger"
2 value="mypackage.VertegenwoordigerCustomizer"/>
```

## Converters

Een andere handige toevoeging van EclipseLink op JPA is het automatisch converteren van data uit de database naar een ander type in de Entity. Dit kan bijvoorbeeld handig zijn voor booleans, die in databases vaak als tekstveld van lengte 1 worden opgeslagen, met bijvoorbeeld Y en N als toegestane waarden.

Op zich kunnen we deze conversie in de getters en setters uitvoeren. Maar als er veel van dit soort velden in de database zijn, kost dat een hoop dubbel werk. In EclipseLink kan de conversie op één plek worden vastgelegd en vervolgens steeds worden hergebruikt. Het definiëren van de conversie gaat met de @ObjectTypeConverter-annotatie, zoals te zien in listing 8. In dit voorbeeld is de hoeveelheid dubbele code die we uitsparen natuurlijk beperkt. Het mag duidelijk zijn dat deze methode bij complexere conversies veel meer voordeel biedt. (We kunnen bijvoorbeeld denken aan een Orderstatus veld met zes mogelijke waarden; dit zouden we mooi kunnen afbeelden op een enum-waarde in het Entity object.)

```
Listing 8: Een type converter definiëren
1 @ObjectTypeConverter (
2 name="booleanConverter",
3 dataType=java.lang.String.class,
4 objectType=java.lang.Boolean.class,
5 conversionValues={
6 @ConversionValue(dataValue="Y",
objectValue=Boolean.TRUE),
7 @ConversionValue(dataValue="N",
objectValue=Boolean.FALSE)}
8 )
```

De @ObjectTypeConverter-annotatie zetten we bijvoorbeeld bovenaan de Entityklasse, waar ook de @Entity-annotatie staat. Vervolgens kunnen we met de @Convert-annotatie refereren aan de gedefinieerde converter, zie listing 9. We kunnen de gedefinieerde converter dus op verschillende plaatsen gebruiken.

```
Listing 9: Een converter gebruiken
1 @Convert("booleanConverter")
2 public Boolean isManager() {
3 return manager;
4 }
```

## De toekomst

Op het moment van schrijven is er nog geen officiële productierelease van EclipseLink. Die staat gepland voor juli 2008[12]. Hoewel dit een 1.0-versie zal zijn, kunnen we hoge kwaliteit verwachten, aangezien de meeste code al jaren in productie draait als onderdeel van TopLink. Ook de verdere toekomst ziet er interessant uit voor EclipseLink. Onlangs is bekend gemaakt dat EclipseLink de referentie-implementatie zal verzorgen van JPA 2.0 (JSR 317). Als referentie-implementatie zal EclipseLink onderdeel uitmaken van GlassFish v3. Dus het ziet er naar uit dat veel JEE-ontwikkelaars vanzelf met EclipseLink in aanraking gaan komen. De naam TopLink Essentials zal vermoedelijk snel van het toneel verdwijnen. Alle aandacht is nu gericht op EclipseLink en het lijkt niet zinvol om TopLink Essentials als apart project te blijven onderhouden. Zoals te lezen is in de EclipseLink FAQ [14, punt 8 en verder], zal Oracle het commerciële product TopLink blijven leveren.

## Conclusie

Met het open source maken van TopLink onder de naam EclipseLink is een zeer uitgebreid persistence framework vrij en gratis beschikbaar gekomen. Naast de standaardfuncties die voortvloeien uit de JPA-standaard, biedt EclipseLink vele extra's, die de gaten dichten die de JPA-standaard open laat. En dan is nog buiten beschouwing gelaten dat EclipseLink naast ORM-functionaliteit ook nog ondersteuning biedt voor XML-files (MOXy/JAXB), Service Data Objects (SDO) en legacy-systemen via EIS. Echt... je moet het ook eens proberen! «

### Referenties


1. Niet iedereen vindt het gebruik van annotaties overigens een verbetering ten opzichte van XML-configuratiebestanden. Dat is waarschijnlijk een kwestie van smaak. Voorstanders van XML boven annotaties hoeven niets te vrezen, want alles wat met annotaties kan worden geconfigureerd in EJB3, kan ook (nog steeds) in XML-files worden vastgelegd.
2. Net als TopLink is ADF Business Components - voorheen Business Components for Java (BC4J) - ooit door Oracle gekocht. Oracle profileert het als een persistence framework dat beter aansluit bij bestaande database-schema's.
3. Overigens is het in ADF wel degelijk mogelijk om TopLink als persistence framework te kiezen.
4. Java Persistence Query Language, de eigen query language die in de JPA standaard gedefinieerd is.
5. org.eclipse.persistence.platform.database.DatabasePlatform
6. Eventueel kunnen we ervoor kiezen om alle query's bij elkaar te groeperen, door gebruik te maken van named queries. Dat neemt niet weg dat dan nog steeds in al deze query's een extra voorwaarde moet worden opgenomen.
7. org.eclipse.persistence.internal.sessions.factories.DescriptorCustomizer
8. org.eclipse.persistence.expressions.ExpressionBuilder
9. Een algemene waarschuwing is hier misschien op zijn plaats. Het gebruik van @OneToMany annotaties kan soms vervelende gevolgen hebben voor de performance. In zulke gevallen kan het lonen om gebruik te maken van een JPQL JOIN-constructie. Zo kunnen we voorkomen dat elk record apart wordt opgehaald. Zie [8, hoofdstuk 7] voor een voorbeeld.