DEV

EJB 3.0 Persistence

Standaard voor Java/database-communicatie

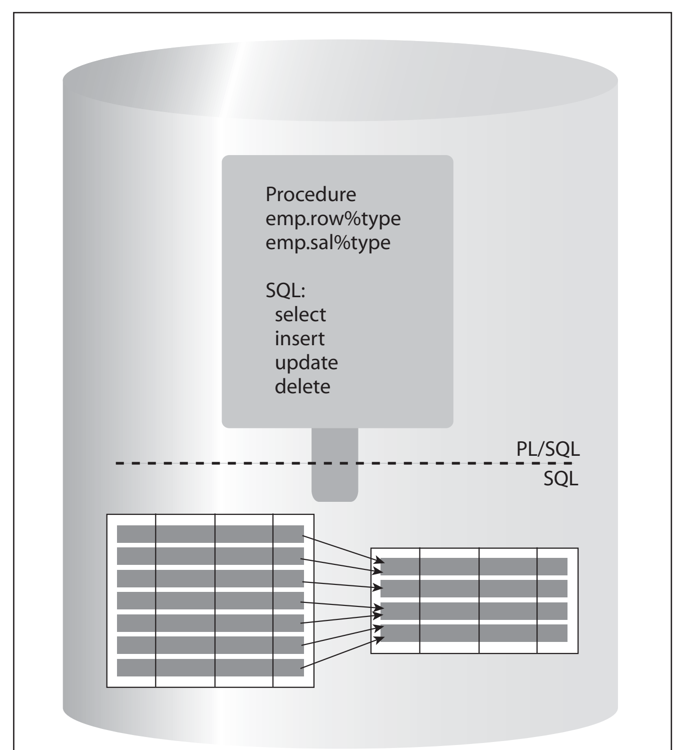
EJB 3.0 Persistence komt eraan: een nieuwe standaard voor Java-applicaties om met relationele databases te communiceren. EJB 3.0 Persistence wordt gesteund door alle belangrijke partijen die zich bezig houden met de mapping tussen Java-applicaties en relationele databases. EJB 3.0 Persistence is onderdeel van de later dit jaar te publiceren JEE 5 standaard – de opvolger van J2EE 1.4. Dit artikel geeft een introductie van EJB 3.0 Persistence.

De komende periode zullen veel Java-ontwikkelaars die met databases te maken hebben in aanraking komen met EJB 3.0. In artikelen, discussies en ook in alledaagse praktijk van de ontwikkeling van Java/J2EE-applicaties wordt veel aandacht besteed aan (Object) Persistence en Object/Relational Mapping. Hierbij gaat het in grote lijnen over de communicatie van de Java-applicatie met de achterliggende relationele database. Waarom is dat nu zo'n bijzonder onderwerp en waarom wordt daar door die Java-ontwikkelaars en J2EE-architecten zo moeilijk over gedaan? In traditionele Oracle-ontwikkelomgevingen zoals PL/SQL en Oracle Forms houden we ons eigenlijk helemaal niet zo expliciet met dit onderwerp bezig. Vanwaar dat verschil?

Services

Allereerst zouden we moeten kijken naar de begrippen Persistence en Object/Relational Mapping (ORM). Waar staan die voor? In vogelvlucht gaat het om het aan een client application beschikbaar stellen van gegevens uit een relationele database. Dat is mogelijk in een vorm die de client-application aankan. In het geval van een Java-applicatie zijn dat objecten. Daarnaast dienen de vluchtige client-objecten - die alleen bestaan in het geheugen of JVM van de client - persistent of permanent vastgelegd te worden in een database. Iets verder uitgewerkt zou je kunnen stellen dat een Java-applicatie behoefte heeft aan 'services' die:

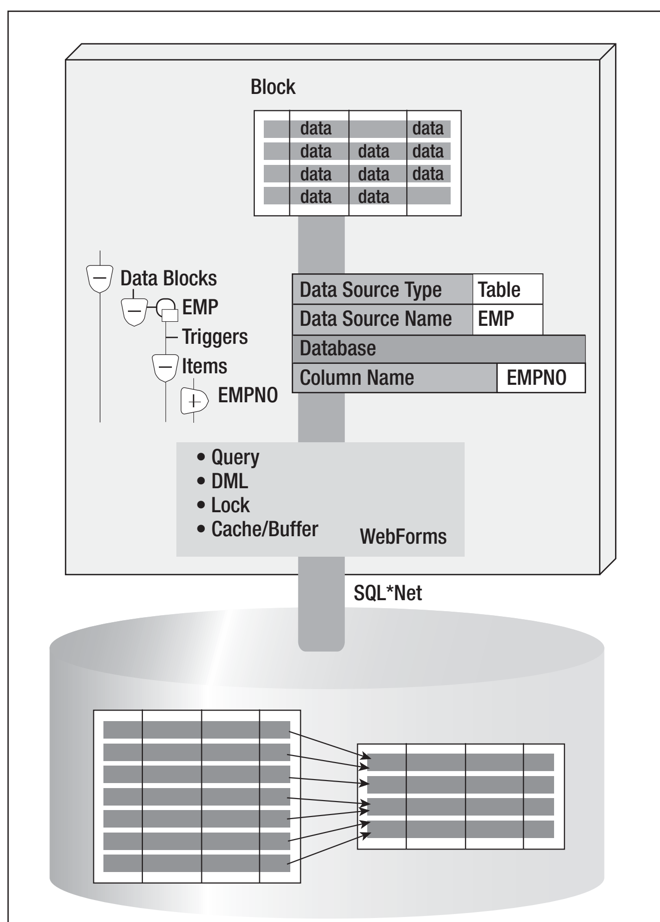
- Zoek-operaties bieden die specifieke relationele data in de vorm van objecten opleveren.
- Manipulatie-operaties verzorgen die objecten via insert, update of delete operaties 'persisteren' in een database.
- Data cachen of bufferen – zodat niet iedere keer dat een record als object benodigd is dit opnieuw uit de database hoeft te worden opgehaald en opnieuw omgezet in een object.
- Database records locken – zodat een gebruiker niet uitgebreide aanpassingen van data hoeft te maken, terwijl bij update blijkt dat een andere gebruiker hem is voor geweest met aanpassingen – of dat dit wel is gebeurd, maar niet blijkt!
- Synchroniseren van objecten – de meest recente toestand uit de database ophalen en toepassen op de mogelijk verouderde objecten.



Afbeelding 1. Voor PL/SQL- en Oracle Forms zijn ontwikkelaarspersistentie en mapping geen punt van zorg.

Een belangrijk aspect komt nadrukkelijk naar voren: het vertalen van objecten in relationele data en andersom. Dit overbruggen van de OO/R kloof – vaak aangeduid als de impedance mismatch tussen het OO-perspectief van de Java-applicatie en het relationele denken van de database – is een belangrijke taak van O/R frameworks. Het is overigens goed op te merken dat het instantiëren – en later door garbage collection weer laten afvoeren van objecten – een naar verhouding dure operatie is, in termen van performance. Dit verklaart direct waarom voor PL/SQL – en Oracle Forms ontwikkelaarspersistentie en mapping geen punt van zorg zijn: zowel PL/SQL als Forms maakt gebruik van dezelfde data-types als de database. Er hoeft geen transformatie plaats te vinden. Het enige gevalletje van 'impedance mismatch' in PL/SQL is het omgaan met Booleans: SQL en de database kennen geen Boolean-datatype, PL/SQL wel. Hier dient een 'mapping' plaats te vinden, van bijvoorbeeld een J/N of 1/0 kolom in de database naar een boolean-variabele in PL/SQL en andersom.

Verder kan in stored PL/SQL door middel van records en collections van records die gebaseerd zijn op tabellen – bijvoorbeeld EMP%ROW_TYPE – of cursoren eenvoudig een 'client-side' opslagstructuur voor relationele database data worden gecreëerd. Hier spelen caching, verversen en synchronisatie overigens vrijwel geen rol aangezien PL/SQL-objecten veelal 'stateless' worden gebruikt – er wordt niet tussen aanroepen allerlei data onthouden in het package. Aangezien de directe communicatie vanuit PL/SQL met de SQL-engine en de relationele data eenvoudig te programmeren en zeer snel uit te voeren is, is de noodzaak voor caching zeer beperkt.



*Afbeelding 2. Records worden uit de database gequeried in een Block en daarna omgezet via SQL*Net.*

Arme Java-ontwikkelaar

Oracle Forms handelt alle onderdelen van de bovengenoemde onderdelen van persistence en client/database-mapping min of meer transparant af. Records worden uit de database gequeried in een Block met zijn intrinsieke Form-PL/SQL recordstructuur. De omzetting van de data die SQL*Net aflevert in structuren waar Forms en PL/SQL bij kunnen, is volledig onzichtbaar. Data kunnen worden gebufferd in een Form – je hoeft niet record voor record op te halen, noch alle records die aan de query-criteria voldoen in één keer op te halen. Zodra een record wordt ge-edit zal Forms een lock proberen te verkrijgen in de database. Verversen van de data kan met requery eenvoudig gerealiseerd worden. Mapping wordt vastgelegd door het Data Source property van een Forms Block en het Column Name property van Items.

Omdat Forms de data-types van de database kent en ondersteunt, omvat deze mapping nauwelijks transformatie. Overigens ondersteunt WebForms 10g ook database Object Types – deze worden automatisch aan de client in afzonderlijke velden uiteengerafeld, een vorm van R/OO mapping waarbij de Database het object kent en de Client in termen van relationele objecten werkt. De arme Java-ontwikkelaar moet al die dingen die PL/SQL en zeker Oracle Forms transparant, automatisch bieden allemaal zelf doen. Daar komt dan nog eens die vertaling van SQL Data Types naar Java Types bij. Ook voor de meeste Java-applicaties geldt dat de onderliggende database essentieel is voor het succes van de applicatie. Vandaar dat er zoveel aandacht is voor Persistentie en OO/R-mapping.

Praten met de database?

De Java-ontwikkelaar gebruikt uiteindelijk – direct of indirect – JDBC om met zijn of haar relationele database te praten. JDBC is een standaard-API, een interface die Sun heeft beschreven en die door iedere database-leverancier op zijn eigen manier wordt geïmplementeerd. JDBC stelt de ontwikkelaar in staat om SQL-statements uit te voeren en zelfs PL/SQL-aanroepen te doen. De resultaten van de query's worden in termen van JDBC-objecten, zoals ResultSet, beschikbaar gesteld. Het is aan de ontwikkelaar zelf om alle SQL te schrijven en om de ResultSets om te zetten in de Java Objecten waar zijn applicatie

mee uit de voeten kan. Bijvoorbeeld: een query tegen de tabel EMP levert een ResultSet object op, terwijl de HRM-applicatie in termen van Employee en Department-objecten is ontwikkeld. Nadat de query is voltooid, moet extra code worden uitgevoerd die van de ResultSet velden echte Employee- en Department-objecten maakt. Uiteraard moet na wijziging van die objecten een omgekeerd traject worden bewandeld om uit de gewijzigde objecten Insert, Update en Delete statements los te peuteren en via JDBC uit te voeren.

Het gebruik van alleen maar JDBC bleek nogal wat nadelen te hebben:

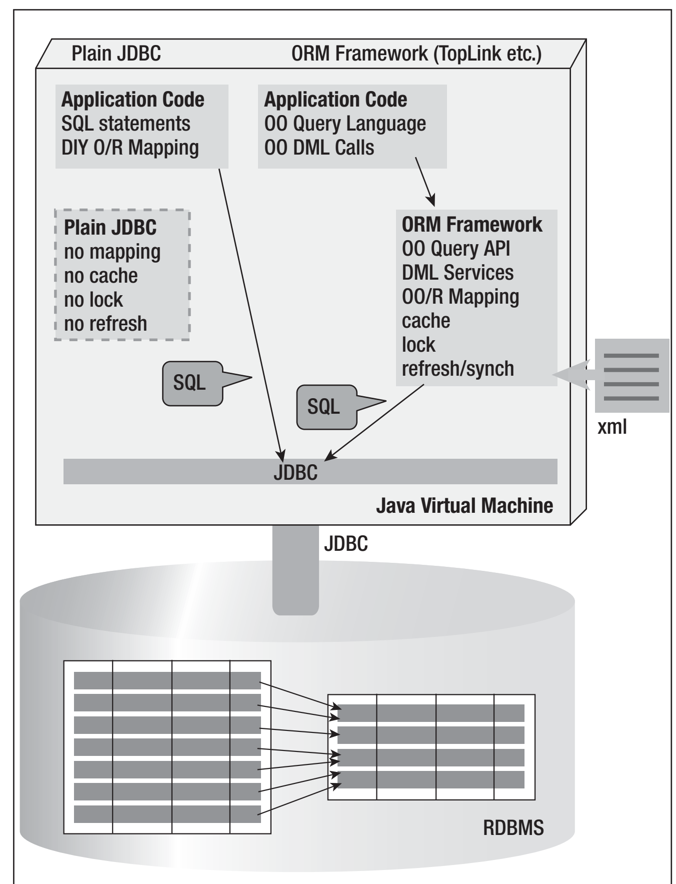
- de SQL-statements worden onderdeel van de Java-code, een overtreding van allerlei ontwerppatronen en regels voor goed ontwerp.
- Java-ontwikkelaars moeten opeens ook SQL leren.
- de SQL-statements worden direct tegen de RDBMS uitgevoerd en moeten dus ook in het specifieke SQL-dialect van de onderliggende database worden geschreven; hierdoor zijn applicaties minder overdraagbaar tussen verschillende databases.
- JDBC biedt geen faciliteiten voor caching, locking of refresh van data.
- JDBC-statements zijn vrij omslachtig in verband met resource-beheer en exceptie-afhandeling – soms is vijf keer zoveel code nodig voor de generieke afhandeling als voor de functionele operatie zelf; bovendien moet telkens opnieuw vrijwel dezelfde code worden geschreven vaak terugkerende operaties, zoals simpele Update- of Insert-statements.
- JDBC doet geen OO/R-mapping: de ontwikkelaar moet zelf de ResultSet omschrijven tot domein-objecten – de business-objecten zoals die in het Class Model ontworpen zijn.

Enterprise Java Beans

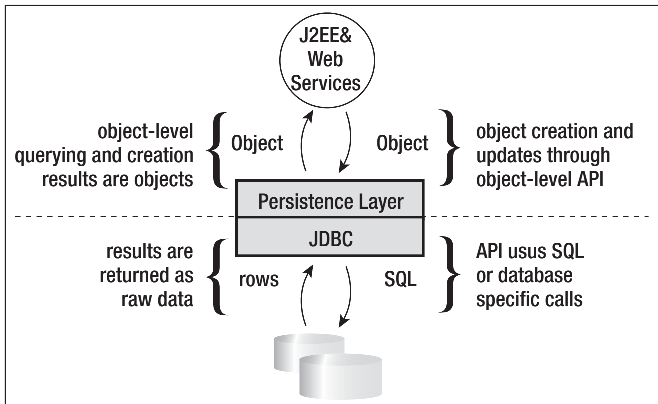
In 1997 kwam Sun met de specificatie voor Enterprise Java Beans (EJB). Dat was nog ruim twee jaar voor de eerste J2EE-specificatie. EJB beloofde een standaardoplossing voor de mapping- en persistence-problematiek waar Java-ontwikkelaars mee worstelden. EJB's boden daarnaast een mechanisme voor het distribueren van applicaties – met een Business Tier op basis van EJB's in een andere applicatieserver dan de Web Tier of Swing Client applicatie. Ook werden via de EJB-infrastructuur – de zogenaamde EJB Container, later onderdeel van J2EE-applicatieservers – geavanceerde services als Transactie en Security Management, Resource Pooling en JNDI Directory Services beschikbaar gesteld. Binnen EJB's zijn er verschillende smaken met betrekking tot database-communicatie:

- BMP ofwel Bean Managed Persistence waar je als ontwikkelaars alle database-interactie zelf programmeert, bijvoorbeeld met kale JDBC of één van de genoemde ORM frameworks.
- CMP ofwel Container Managed Persistence waar je als ontwikkelaar in EJB QL – de EJB Query Language – en standaard EJB service-aanroepen je persistentie-functionaliteit aanspreekt die vervolgens door de EJB Container wordt omgezet in SQL die via JDBC wordt uitgevoerd.

Dat de meningen over EJB verdeeld zijn, is een understatement. Hoewel er – zeker een aantal jaren geleden – veel en ook in zekere mate met succes gebruik is gemaakt van EJB zijn er toch ernstige bezwaren aan te voeren: EJB's zijn ingewikkeld. Ze lijken voorbehouden aan J2EE-goeroes en super-architecten, en schijnen niet voor de gemiddelde ontwikkelaar te zijn weggelegd. De productiviteit lijdt ernstig onder het grote aantal configuratie-files en code-componenten dat een EJB-applicatie vereist. Daarnaast is voor het runnen van een EJB-gebaseerde applicatie een J2EE-applicatieserver nodig – veelal de enterprise-editie – en kan niet met een gewone servlet-container zoals Tomcat worden volstaan. De overhead van EJB wordt in veel niet gerechtvaardigd door de voordelen, zeker omdat gedistribueerde applicaties veel minder voorkomen dan wel werd verondersteld. Daar komt bij dat EJB-implementaties onderling lang zo uitwisselbaar niet zijn als de bedoeling was. Ook is de functionaliteit van met name de EJB QL-querytaal nogal



Afbeelding 3. ORM Frameworks, tools en library's nemen een flink deel van het loodgieterswerk in de communicatie met de database over van de Java-ontwikkelaar.



Afbeelding 4. ORM Frameworks bieden standaardservices voor het creëren, wijzigen of verwijderen van domeinobjecten.

beperkt en niet te vergelijken met SQL. Tenslotte is unit-testen van EJB's zeer complex, aangezien de EJB's uitsluitend binnen een EJB-container kunnen bestaan.

Object Relational Mapping Frameworks

In reactie op de tekortkomingen van kaal JDBC en de complexiteit van EJB's ontstond er een aantal Object Relational Mapping frameworks, tools en library's die een flink deel van het 'loodgieterswerk' in de communicatie met de database overnamen van de Java-ontwikkelaar. Frameworks als TopLink (in 2002 door Oracle overgenomen), Hibernate (een open source-product van de JBoss-groep) en minder geavanceerde producten als Castor, iBatis en Apache OJB bieden ondersteuning bij het overbruggen van de 'Object/Relational gap'. In de meeste gevallen specificeert de ontwikkelaar in een XML-bestand de mapping tussen tabellen, kolommen en foreign keys in de database naar classes, property's en relaties in de Java-applicatie. Er kan een echt Class Model – een ERD van Java Objecten – worden ontworpen met werkelijke domein-objecten. Het domeinmodel bestaat uit POJO's – plain old java objects ofwel eenvoudige Java Beans met zogenaamde getters en setters (methodes die de waarde van een property opvragen of aanpassen). De ORM Frameworks bieden standaardservices voor het creëren, wijzigen of verwijderen van domeinobjecten en daarmee impliciet voor manipulatie van de onderliggende database-records.

Verschillende van de ORM-frameworks – zoals TopLink en Hibernate – bieden een eigen, objectgeoriënteerde query-taal. Hiermee kunnen ontwikkelaars een zoekvraag stellen op een puur functionele manier, in termen van de domeinobjecten – property's en relaties – die door het framework wordt omgezet in een SQL-query. Hiermee wordt de Java-applicatie portable tussen databases, er staat immers geen database-specifieke SQL meer in de applicatie. Bovendien zorgt het framework ervoor dat het zoekresultaat in de vorm van domeinobjecten wordt teruggegeven. De ontwikkelaar hoeft dus ook niet meer handmatig code te schrijven die de mapping doet van de ResultSet naar zijn eigen classes. Daarnaast hebben de geavanceerde ORM-frameworks faciliteiten voor het cachen van data, het correct implementeren van locking-strategieën, zowel Pessimistic Locking als Optimistic Locking, en het eenvoudig verversen of synchroniseren van Java-objecten op basis van database-gegevens.

Standaardisatie

Ondanks het grote succes van deze ORM-frameworks zijn er nog wel enkele kanttekeningen te plaatsen. Er is geen sprake van standaardisatie. Hoewel alle frameworks vergelijkbaar zijn, zijn ze toch allemaal anders. Ontwikkelaars moeten voor ieder ORM-framework opnieuw de schoolbanken in. Applicaties zijn niet overdraagbaar tussen de ORM-frameworks van verschillende leveranciers. Ook voor de ORM-frameworks geldt dat er – weliswaar minder dan met kaal JDBC – ook vrij veel generieke, repeterende code nodig is. Daarnaast wordt in de meeste gevallen met POJO's gewerkt – volledig standaard Java Beans – maar die zijn in de praktijk vaak toch net niet helemaal onafhankelijk van het ORM Framework: er moeten specifieke types worden gebruikt voor de property's en de collections, of de POJO's moeten framework-specifieke interfaces implementeren. Daarnaast was de ondersteuning door ontwikkelhulpmiddelen (IDE's) in veel gevallen maar matig. Dat maakte de ORM Frameworks voor de gemiddelde ontwikkelaar minder toegankelijk dan wenselijk was.

Java Data Objects (JDO)

Sinds 2002 probeert een aantal leveranciers binnen JSR-12 een vorm van standaardisatie te bereiken op het gebied van ORM. JDO is echter nooit echt doorgebroken, ondermeer door halfslachtige tot afwezige ondersteuning van JDO door TopLink en Hibernate. Dit ondanks aansprekende resultaten en succesvolle implementaties. In het begin van 2005 werd de nieuwe 2.0 versie van JDO (JSR-243) door IBM, BEA en Oracle tegengehouden, formeel vanwege de verwarring met EJB 3.0. Het moge duidelijk zijn dat hiermee JDO waarschijnlijk een doodlopende weg zal blijken. Ongetwijfeld hebben politieke overwegingen een grote rol gespeeld. De grote leveranciers van non-JDO ORM frameworks hebben hun JDO-concurrenten hiermee effectief de pas afgesneden.

Spring Framework

Een andere ontwikkeling om de omslachtigheid en relatieve complexiteit van de verschillende ORM-frameworks te lijf te gaan, is te vinden in het Spring Framework. Binnen Spring worden verschillende Container-achtige services als Transactie Management beschikbaar gesteld. Daarnaast kent Spring voor onder meer kaal JDBC, Hibernate, iBatis en Toplink zogenaam-

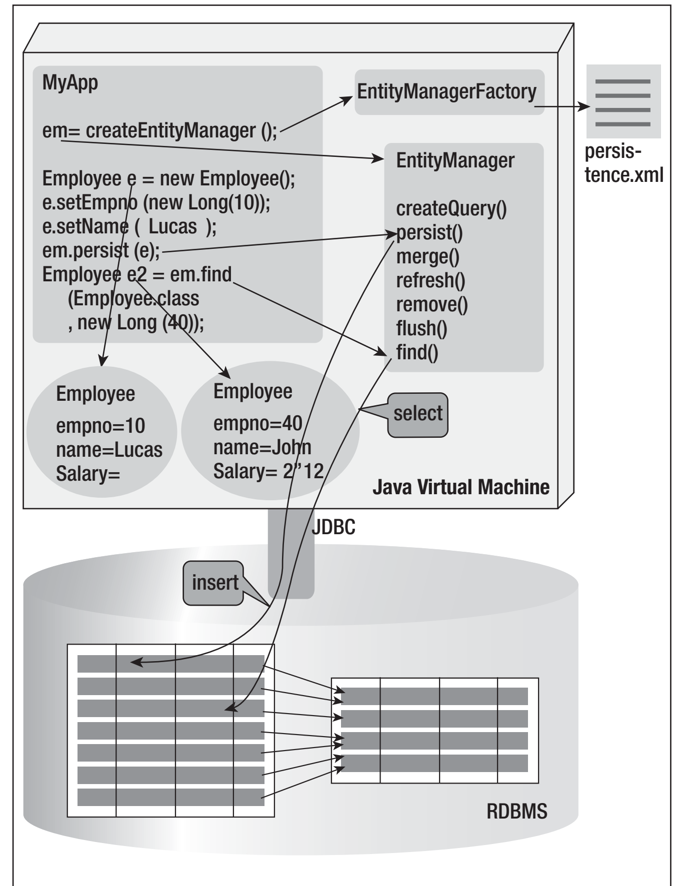
de Template-classes. Deze maken het voor de ontwikkelaar veel eenvoudiger om de persistentie-services aan te spreken: alle infrastructurele code en exceptie-afhandeling is in het Spring Framework ingebouwd. De ontwikkelaar kan zich volledig richten op de business-logica. Code is echter ook met Spring-templates niet overdraagbaar tussen ORM-frameworks; voor ieder framework gebruik je afzonderlijke templates.

Oracle Business Components for Java (ADF BC voorheen BC4J)
In 1999 kwam Oracle binnen JDeveloper 2.0 met de eerste versie van het Business Components for Java framework. Ook BC4J is een ORM-framework met faciliteiten als caching, locking en services voor query- en data-manipulatie. BC4J was veel 'relationeler en meer SQL-gericht' dan de eerder besproken ORM-frameworks. BC4J is zeer database-aware en biedt voor de Oracle-database faciliteiten als Refresh After Insert of Update om de effecten van Database Triggers en Default Values op te pakken. Binnen BC4J schrijft de ontwikkelaar directe SQL-statements – niks geen OO-query taal voor database-agnostische Java-ontwikkelaars. Een BC4J-applicatie werkt ook niet met een zuiver POJO-model; de domeinobjecten zijn allemaal BC4J-specifieke classes en interfaces. De productiviteit die BC4J aan ontwikkelaars biedt, heeft wel een prijs: vervlechting van de applicatie met het framework.
Door deze onvervalste (Oracle-) database-oriëntatie en zeer productieve integratie met de JDeveloper IDE is BC4J populair geworden bij de grote groep Oracle-ontwikkelaars die naast Designer en Forms ook met Java aan de slag ging. BC4J is ook de motor onder ADF en JHeadstart. Tegelijk hebben diezelfde eigenschappen ervoor gezorgd dat buiten de Oracle-gemeenschap veelal met enige achterdocht en wantrouwen naar BC4J wordt gekeken.

JSR 220 - EJB 3.0

Begin 2005 werd JSR-220 aangekondigd: de EJB Release 3.0 specificatie. Een belangrijk onderdeel van deze specificatie is de nieuwe Persistence API. Tot de belangrijkste doelstellingen van deze nieuwe API horen:

- een industriebrede standaard voor Object Relational Mapping – zodat ontwikkelaars slechts één ORM-aanpak hoeven te leren en applicaties overdraagbaar worden tussen verschillen ORM-implementaties.
- een vorm van ORM die zowel binnen EJB Containers kan worden gebruikt als voor J2SE-applicaties – standaard Java-applicaties zoals Swing Client of JSP/Servlet webapplicaties.
- een schone lei, gebaseerd op jarenlange ervaring met 'the good, the bad and the ugly' in Java Object/Relational Mapping, ondersteund door alle prominente spelers: Tot de partijen die vooraan lopen bij EJB 3.0 horen naast Sun ondermeer JBoss/Hibernate, Oracle TopLink en SolarMetric Kodo – de belangrijkste implementatie van JDO. Overigens werd eind 2005 bekend dat SolarMetric wordt overgenomen door BEA.
- een simpele, elegante en rijke interface die eenvoudig door ontwikkelaars opgepakt kan worden en die ook goed door IDE's ondersteund kan worden; een van de motto's van EJB 3.0 is: 'Configuration by Exception'. Daarmee wordt bedoeld dat je alleen maar hoeft te configureren wat afwijkt van de default. Dat scheelt enorm, tot zeventig procent, in de omvang van de configuratie-definities.



Afbeelding 5. De EJB 3.0 Persistence API.

Ook wordt voor EJB 3.0 een referentie-implementatie aangeboden die gratis, open source en van productiekwaliteit is. Deze referentie-implementatie is onderdeel van GlassFish, de JEE 5 Referentie Implementatie Applicatie Server . De GlassFish Persistence-implementatie is een uitgeklede variant van Oracle Toplink. Die code is nu gratis als open source beschikbaar! Om de concurrentie tussen de ORM-leveranciers aan te moedigen zonder de uitwisselbaarheid in gevaar te brengen biedt EJB 3.0-faciliteiten om extra ORM-functionaliteit beschikbaar te stellen binnen het standaardraamwerk, ondermeer door leverancier-specifieke hints en extra annotaties die genegeerd mogen worden.
Voor alle duidelijkheid: EJB 3.0 is veel meer dan alleen deze Persistence API. Deze 3.0 release biedt sowieso volledige back-

wards compatibility, ofwel alle componenten die voor EJB 2.1 werden ondersteund, kan je ook met een EJB 3.0 container gebruiken. Local en Remote interfaces, aanroepen via webservices en RMI of RMI/IIOP en Message Driven Beans bestaan ook nog steeds binnen EJB 3.0. Deze is zowel beschikbaar binnen EJB-containers als daarbuiten. De laatste is van eminent belang in de brede toepasbaarheid van de EJB 3.0 Persistence API.

Ondersteuning in IDE's: Dali en JDeveloper

Onder leiding van Oracle wordt voor de Eclipse IDE een plugin ontwikkeld om ontwikkeling van EJB 3.0-applicaties, met name Entity's met bijbehorende ORM mapping meta-data, te ondersteunen. Deze plugin heet Dali en kan worden gedownload van de Dali-homepage: www.eclipse.org/dali/main.html. Daarnaast bouwt Oracle in JDeveloper een rijke set wizards waarmee op vergelijkbaar productieve wijze als met BC4J op basis van database-objecten – tabellen en views – de entity's kunnen worden gegenereerd.

Wat is de EJB 3.0 Persistence API?

Veel van de J(2)EE-technologieën zijn gespecificeerd in de vorm van API's of interfaces. Zo ook EJB 3.0 Persistence. De specificatie legt vast welke functionaliteit op welke manier kan worden aangeroepen, maar zegt niet hoe die functionaliteit moet worden geïmplementeerd. Zo stelt de Persistence API dat er een refresh (Object) methode is, die een Entity synchroniseert met de achterliggende database. Hoe een leverancier die methode wil implementeren is niet beschreven. Zo kunnen implementaties met elkaar concurreren door betere, slimmere, snellere implementaties te bouwen van de methodes die in de interface zijn voorgeschreven.

De EJB 3.0 Persistence-specificatie bestaat uit drie hoofdcomponenten:

- Object Relational Mapping Meta Data – via annotations 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 EJB QL query's – of NativeQueries in puur SQL - worden aangevraagd of Entities gecreëerd of ge-update. De applicatie werkt met POJO's 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 POJO's en hun property's en relaties vertaald kunnen worden in tabellen met kolommen en foreign keys. Deze informatie wordt vastgelegd met de Object Relational Mapping meta-data, normaal gesproken door middel van Annotaties in de POJO's.

Conclusies

Eendracht maakt macht. EJB 3.0 Persistence is in grote ogenschijnlijke 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 vele implementaties stellen ontwikkelaars al in een vroeg stadium in staat met EJB 3.0 aan de slag te gaan en de vele aandacht op websites, in tijdschriften en op conferenties zoals JavaOne en vorige maand JavaPolis in Antwerpen laten zien dat er veel interesse is.
EJB 3.0 Persistence is niet hetzelfde als Enterprise Java Beans. EJB 3.0 is meer dan alleen Persistence, en EJB 3.0 Persistence is zowel minder als meer dan EJB 2.1. Meer omdat het ook buiten EJB-containers kan worden ingezet, voor gewone webapplicaties in servlet-containers zoals Tomcat en ook voor Java Client (Swing) Applicaties. Waar tot nu toe ORM-frameworks als TopLink, Hibernate, iBATIS gebruikt werden, zou het wenselijk zijn via de EJB 3.0 Persistence interface te gaan werken. De vele bezwaren die tegen EJB zijn ingebracht zijn niet relevant als het gaat om EJB 3.0 Persistence.
Met EJB 3.0 Persistence hebben de grote ORM-leveranciers – Oracle/TopLink, JBoss/Hibernate en BEA/Kodo de lat – bewust – erg hoog gelegd: zij kunnen een EJB 3.0-implementatie leveren op basis van hun bestaande producten, en bovendien daar bovenop nog concurrerende extra's bieden. Kleinere spelers als Apache OJB , iBATIS en Castor zullen het erg moeilijk krijgen snel met een goede EJB 3.0 implementatie te komen en dreigen daardoor uit de boot te vallen. De rol van JDO lijkt uitgespeeld, ook al zijn de JDO-leveranciers het daarmee hartgrondig oneens. Het feit dat Kodo, de voornaamste JDO-speler een van de eerste EJB 3.0 implementaties heeft is veelbetekenend. Ook de positie van BC4J of ADF Business Components wordt minder duidelijk. Ik vraag me af hoe lang Oracle volledig op ADF BC blijft inzetten. Ook al is dat vooralsnog nog wel het geval met betrekking tot de Fusion Tools, ik hoor ook geluiden binnen Oracle die suggereren dat de focus naar EJB 3.0 zou kunnen verschuiven.

Beperkingen en kanttekeningen

EJB 3.0 is er nog niet. Vlak voor JavaOne – mei 2006 – wordt de specificatie naar verwachting definitief vastgesteld, als onderdeel van het JEE 5 platform. Productieversies van JEE 5 applica-

tieservers en van J2SE-implementaties van EJB 3.0 van ondermeer TopLink en Hibernate zouden heel snel daarna beschikbaar moeten komen. Nu al zijn er preview-versies die vergaand de specificatie ondersteunen. Voordat zeker de grotere organisaties op Java 5 en JEE 5 zijn overgestapt zijn we makkelijk een paar jaar verder. Maar de (r)evolutie naar EJB 3.0 zal spoedig een aanvang nemen, zeker omdat buiten de container al spoedig de nieuwste versies van ondermeer TopLink en Hibernate op basis van EJB 3.0 gebruikt kunnen gaan worden.
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.

Al met al

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 - 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.

OraVision bouwt Oracle-oplossingen waarin documenten, transacties en bestaande systemen samenwerken.
OraVision staat bekend als *the mid-office company*.
OraVision bouwt vanuit haar geheel eigen visie: kwaliteit staat centraal.

Kwaliteit in kennis
OraVision beschikt over enorme ervaring in Oracle-, Java- en integratietechnologieën. Bij ons staat de techniek echter nooit op zichzelf. Juist bij mid-office en document-integratie toepassingen laten we de technologie tot volle bloei komen.

Kwaliteit in werk
Klanten geven OraVision al jaren het vertrouwen om geavanceerde ICT-toepassingen te realiseren die tegelijk gebruikersvriendelijk zijn. Onze mid-office oplossingen bevinden zich immers in het hart van elke bedrijfsvoering.

Kwaliteit in samenwerking
Bij OraVision staat niet alleen technische kwaliteit hoog in het vaandel, ook onze stijl is onderscheidend. Vanuit onze Limburgse basis investeren we nadrukkelijk in persoonlijke relaties en genieten van het goede leven.

Geïnteresseerd in de visie van OraVision op Oracle, Java, integratie en mid-office? Bezoek *www.oravision.com* en abonneer u gratis op de OraVisionair.