ABIS NIEUWSBRIEF VOOR DB2 PROFESSIONALS

JAARGANG 1 - NUMMER 7, APRIL 2003

OPEN CURSOR

Dit is een eerste speciaal nummer in de zin dat we DB2 willen bekijken vanuit een andere invalshoek. Een invalshoek waar we allemaal op korte dan wel langere termijn mee zullen te maken krijgen: Java. Meer specifiek pogen we in dit nummer stil te staan bij de verschillende technieken die ter beschikking staan van de applicatieontwikkelaar om data - en dus DB2 - te integreren met Java-applicaties.
We zetten een aantal veel gehoorde kreten uiteen - SQLJ, JDBC, EJB, ... - en we lichten hun impact toe op databases. Uiteraard komt ook de DBA aan het woord.
Dit nummer hebben we uitzonderlijk eveneens toegestuurd naar Javageïnteresseerden. We hopen dat ook zij geboeid zijn door de behandelde onderwerpen - zij het vanuit een ander gezichtspunt.

Veel leesgenot - het DB2/Java team.

IN DIT NUMMER:

- Een inleidend overzicht van de mogelijke technieken voor database toegang - *Java en DB2: een uitdagende combinatie*.
- Het verschil tussen JDBC en SQLJ wordt uiteengezet in *APIs voor de ontwikkelaar*.
- In *Mapping en persistentie* gaan we in op architecturele oplossingen voor database toegang: één artikel over EJB en één artikel over JDO.
- En zoals het hoort: de DBA heeft het laatste woord, in *Java en DB2 - bedenkingen van een DBA*.
- *Dossier 7*: UNIONS.
- *Cursusplanning apr-jun 2003.*

CLOSE CURSOR

Volgende maand richten we ons terug volledig op DB2. Aan de orde zijn SQL PL en de performance voordelen van de CASE-instructie. In dossier 7 bekijken we het belang van 2 nieuwe kolommen in sysibm.sysindexpart.

Tot dan!

Java en DB2, een uitdagende combinatie

Eric Seynaeve (ABIS)

De laatste jaren is het gebruik van Java in bedrijven fors toegenomen. Javatoepassingen interageren meer en meer met complexe, back-end, legacy systemen. Grote hoeveelheden data worden in de database, in de applicatieserver en in de gebruikersinterface behandeld. Javaontwikkelaars moeten niet enkel oog hebben voor Java, ze moeten ook stilstaan bij data en dus bij databases.

Hierdoor is ook de interactie gegroeid tussen de Javaontwikkelaars en de DBA's. Daarom is het interessant voor DBA's de verschillende technieken te onderkennen die Javaontwikkelaars gebruiken om contact op te nemen met de database.

In dit artikel worden vier veel voorkomende technieken ingeleid; we plaatsen ze binnen de context van databasetoegang. Verder bespreken we het verschil in objectstructuur en databasestructuur. Voor meer details over de verschillende technieken verwijzen we graag naar de vervolgartikels in deze 'Exploring DB2'.

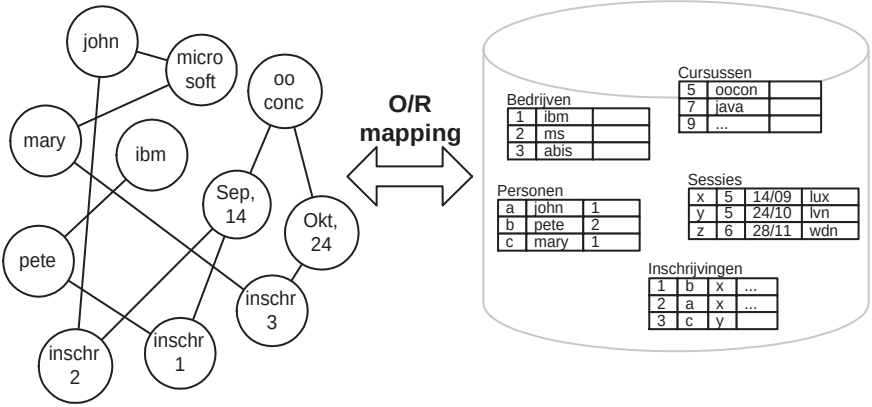
Objecten versus databases: 'O/R mapping'

In een Javatoepassing worden gegevens in objecten bijgehouden in tijdelijk geheugen. Om deze gegevens permanent te bewaren, worden ze gepersisteerd: dat wil zeggen weggeschreven naar een permanente geheugenstructuur, b.v. een database.

Bij het maken van objectgeoriënteerde (b.v. Java) toepassingen die zich naar een database richten, botsen we op een probleem: de logische datastructuur van een OO-toepassing komt gewoon niet overheen met de structuur van relationele databases! Of meer specifiek: Java structureert data in klassen die met elkaar verbonden zijn door overerving, compositie, e.d.; databases structureren data in tabellen, die met elkaar verbonden worden via relaties. Bovendien komen de attributen van klassen niet steeds overeen met de kolommen uit een tabel. Het overbruggen van dit structuurprobleem is meestal de grootste uitdaging bij de ontwikkeling van omvangrijke, schaalbare en onderhoudbare datagerichte Javatoepassingen.

Het 'Object-Relational mapping' probleem wordt geïllustreerd in figuur 1.

Figuur 1: Object-relational mapping



Gebruikte technieken

Binnen Java zijn er vier vaak terugkomende technieken om een connectie te maken met de database:

- JDBC (Java DataBase Connectivity) biedt de mogelijkheid aan de Javaontwikkelaar om dynamische SQL-statements uit te voeren op een database en de resultaatsets op te vragen. JDBC is onderdeel van de J2EE-standaard. Het is de standaardtechniek waar de hieronder opgesomde technieken gebruik van maken.

- SQLJ kan omschreven worden als het klassieke 'embedded SQL' gebruik. Het biedt de mogelijkheid om statische SQL te gebruiken in Javacode. SQLJ is geen onderdeel van de J2EE-standaard.

- De EJB (Enterprise JavaBeans) architectuur gaat een hele stap verder dan JDBC. Het definiëren van SQL-vragen kan nu ook gerealiseerd worden via een externe beschrijving, buiten de toepassing. Deze beschrijving wordt door de uitvoeringsomgeving (EJB-container) gebruikt voor het realiseren van de databasetoegang. Bovendien worden een aantal extra diensten zoals resource pooling, transactioneel beheer, life-cycle management, beveiliging, ... verzorgd door de EJB-container. Voor de databaseontwikkelaar zijn binnen het EJB-framework vooral de entity beans het meest interessant. Het is met behulp van deze entity beans dat, normaal gezien, contact opgenomen wordt met de database. EJB is onderdeel van de J2EE-standaard.

- JDO (Java Data Objects) zorgt weeral voor een framework, dat het de Javaontwikkelaar mogelijk maakt om transparant zijn gebruikte Javaobjecten te persisteren op een database. JDO is geen onderdeel van de J2EE-standaard, maar is wel door Sun gestandaardiseerd. JDO laat toe om op een eenvoudige manier de objecten te mappen op de databasestructuur. De extra diensten van

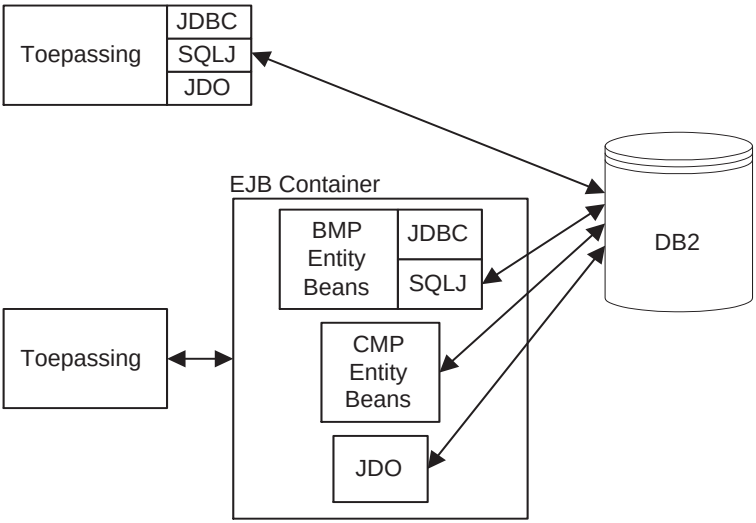
het JDO-framework zijn beperkt tot transactioneel beheer en resource pooling.

Inzetten van de verschillende technieken

Zoals in figuur 2 is aangeduid, kunnen de boven geschetste technieken op verschillende plaatsen gebruikt worden.

Voor kleinere toepassingen die niet zwaar belast worden en voornamelijk enkel gegevens lezen, volstaat het om de connectie met de database te voorzien via JDBC, SQLJ of JDO.

Figuur 2: Inzetten van de juiste technieken



Voor grotere enterprisetoepassingen waar gebruik gemaakt wordt van load-balancing, een 3- of n-tier architectuur, beveiliging, ... is het beter gebruik te maken van een EJB-container die deze diensten aanbiedt. Binnen de EJB-container kan bijvoorbeeld gebruik gemaakt worden van bean managed persistence (BMP) entity beans waar de ontwikkelaars zelf de code schrijven om de gegevens uit de database op te halen of in de database weg te schrijven. Het voordeel voor de ontwikkelaars om gebruik te maken van een EJB-container ligt op het gebied van de door de EJB-container aangeboden diensten: transacties, distributie van diensten, load-balancing, life-cycle management en beveiliging. Om de SQL te schrijven, maakt men meestal gebruik van JDBC en in mindere mate van SQLJ.

Indien men beslist om de SQL niet zelf te schrijven, maar te laten genereren, kan men gebruik maken van container managed persistence (CMP) entity beans.

Uiteraard kan men ook beslissen helemaal geen gebruik te maken van entity beans en deze te vervangen door JDO. Deze mogelijkheid is reeds voorzien in JDO en de gegenereerde code zal automatisch gebruik maken van de extra diensten die geleverd worden door de EJB-container.

Een voordeel bij het laten genereren van de SQL-code (bij CMP entity beans of JDO) is dat men minder last zal hebben van het zogenoemde object-relationele mappingprobleem.

Welke techniek men ook kiest, onderliggend wordt telkens JDBC gebruikt om de database te benaderen.

Conclusie

Java biedt verschillende mogelijkheden om te werken met gegevens uit een database. Indien de ontwikkelaars zelf de SQL zullen schrijven waarmee met de database gecommuniceerd wordt, wat in de praktijk het meeste voorkomt, zal men gebruik maken van JDBC of SQLJ. Afhankelijk van de grootte van de toepassing, de belasting van het systeem en het gemak waarmee toekomstige wijzigingen doorgevoerd moeten worden, kan men beslissen om gebruik te maken van een EJB-container of niet. In dat geval kan men de SQL laten genereren door gebruik te maken van CMP entity beans ofwel zelf de SQL te schrijven in BMP entity beans. Een andere optie is het gebruik van JDO waarbij men echter nog steeds in meer of mindere mate de mogelijkheid heeft om de gegenereerde SQL te sturen.

Een verklarende woordenlijst.

- *klasse*: een soort van type definitie (template) van gegevens en functies, die op deze gegevens inwerken. Bijvoorbeeld: een bedrijf heeft een naam, een adres, datum van oprichting, ... en kan o.a; een werknemer aannemen.
- *object*: op basis van een klasse worden concrete objecten aangemaakt. Bijvoorbeeld: ABIS is een bedrijf uit Leuven.
- *overerving*: de techniek waarbij een bepaalde klasse gedrag en kennis overerft van een generiekere klasse. Bijvoorbeeld: de klasse bank erft van de klasse bedrijf. Bank krijgt automatisch alle kennis die deel uitmaakt van bedrijf mee, zoals naam, adres, datum van oprichting, ..., en ook al het gedrag, zoals het aannemen van een werknemer, ...
- *compositie*: de techniek waarbij een bepaalde klasse deel uitmaakt van een andere klasse. Bijvoorbeeld: een bank heeft verschillende afdelingen.
- *persisteren*: complexe Javaobjecten (al dan niet inclusief broncode) bewaren in een database of direct toegankelijk bestand.

API's voor de ontwikkelaar

Pieter Bedert, Gie Indesteege (ABIS)

Java API's voor de ontwikkelaar

Bedrijfstoepassingen werken meestal met gegevens uit een database. Iedere ontwikkelomgeving gebruikt daarvoor zijn eigen specifieke mogelijkheden om zo efficiënt mogelijk om te gaan met die gegevens. Java als taal en platform biedt een uitgebreide waaier aan mogelijkheden om DB2 gegevens te benaderen. De Java API's (Application Programming Interfaces) die we hier zullen bespreken zijn JDBC (Java Data Base Connectivity) en SQLJ (Java - Relationele Database technologie).

JDBC

JDBC is zowat de standaard API om in Javaprogramma's en -applets SQL-instructies te gebruiken. JDBC vindt zijn oorsprong in het bekende ODBC (Open Data Base Connectivity), een open standaard om data (bases) te benaderen. JDBC werkt op basis van dynamische SQL-vragen.

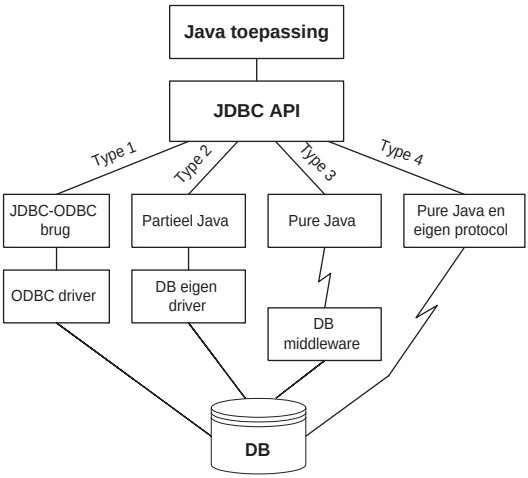
JDBC - connectie naar de database

Vooraleer men met behulp van JDBC SQL vragen kan stellen aan de database, moet men eerst duidelijk aangeven van welke database men gebruik wil maken. Daarom moet eerst een connectie aangemaakt worden met een specifieke database.

Het creëren van de connectie met de database gebeurt met behulp van een ‘driver’. Op dit moment worden 4 types JDBC-drivers ondersteund die elk op een eigen manier de database benaderen (voor een overzicht, zie figuur 1):

- Een JDBC-ODBC driver die eigenlijk een koppeling naar alle ODBC-ondersteunende databases kan maken (type 1).
- Een ‘app’-driver (type 2), deze Javadriver is voor een specifieke database ontwikkeld (b.v. DB2) en veronderstelt dat op de lokale machine nog een database-eigen toegansmodule voorzien is (b.v. DB2 client software).
- Het derde type, een ‘net’-driver, is een pure Javadriver (d.w.z. dat er lokaal geen databasesoftware vereist is) die connectie maakt naar middleware, die de vraag dan op zijn beurt doorspeelt naar de database. Type 3 wordt typisch door applets gebruikt die via een server naar de database connecteren.
- Het laatste type (type 4) is opnieuw een pure Javadriver, maar deze driver gaat rechtstreeks (via de DRDA Application Requester) naar de database connecteren. Type 4 is vanaf DB2 V8 ondersteund en zou de meest efficiënte toegang tot de database bieden.

Figuur 1: JDBC drivers



Voor meer informatie in verband met JDBC-drivers in DB2 verwijzen we naar volgende URL:
http://www-3.ibm.com/software/data/db2/udb/ad/v8/java/.

De driver activeren enerzijds, en de connectie naar de database ('*Connection*') creëren anderzijds, gebeurt dan door ofwel de '*DriverManager*' klasse (JDBC 1.0) ofwel door een '*DataSource*' object (vanaf JDBC 2.0) te gebruiken.

```
ourConnection =
DriverManager.getConnection
  (databaseURL ,
   userID,
   password)
```

Bij gebruik van een dergelijk '*DataSource*' object wordt de database dynamisch opgezocht (en dus niet mee hard gecodeerd) op de applicatieserver en tegelijk wordt ook de nodige driver geladen. Het gebruik van datasources biedt naast zijn flexibiliteit nog extra voordelen zoals connection pooling, transactiebeheer, beveiliging, ...

JDBC - SQL

JDBC SQL is dynamische SQL die in grote lijnen overeenkomt met de dynamische SQL die u kent uit andere programmeertalen (vb. COBOL, C,...). In voorbeeld 1 proberen we dit te verduidelijken.

Het SQL 'statement' wordt in een tekstvariabele (String) weggeschreven en eventueel in de loop van het programma nog gewijzigd of aangevuld. Wanneer het 'statement' volledig is afgewerkt, laat men het uitvoeren door de database. Dit gebeurt altijd in twee stappen.

Voorbeeld 1

```
...
String sql = "UPDATE rekeningen SET saldo = ? WHERE rekening_nr = ? " ;
PreparedStatement ourStatement = ourConnection.prepareStatement(sql) ;
ourStatement.setInt( 1 , 10000) ;
// Vervang het eerste vraagteken door een nieuw saldo van 10000.
ourStatement.setString( 2 , "123-1234567-12") ;
// Vervang het tweede vraagteken door rekeningnummer "123-1234567-12".
...
ourStatement.executeUpdate()
...
ourConnection.commit() ;  /
```

In een eerste stap wordt een '*Statement*' aangemaakt voor een bepaalde '*Connection*'. De beschikbare methodes hiervoor zijn:

- *createStatement* voor niet-geparametriseerde SQL;
- *prepareCall* voor het oproepen van een stored procedure;
- *prepareStatement* voor geparametriseerde SQL.

Door gebruik te maken van de '*Connection*' is meteen duidelijk op welke database de SQL zal uitgevoerd worden. Bovendien hebben de twee 'prepare'-methodes het extra voordeel dat ze onmiddellijk geoptimaliseerd worden.

De evidente tweede stap is eerst de vraagtekens (JDBC parameters) invullen en het uitvoeren van het SQL-statement. De methodes voor het uitvoeren van het statement die hierbij kunnen gebruikt worden zijn: *executeQuery*, *executeBatch*, *executeUpdate* of kortweg *execute*.

JDBC - foutenafhandeling e.a.

Naast het pure functionele uitvoeren van SQL denkt de ontwikkelaar ook onmiddellijk aan transactiebeheer binnen zijn applicaties: commit processing, lockingopties en uiteraard de nodige foutenafhandeling. Binnen JDBC zijn hiervoor de nodige methodes en klasses (vb: klasse '*SQLException*') voorzien.

SQLJ

In tegenstelling tot het pure Java programmeren met JDBC, is SQLJ in feite embedded SQL binnen de Javacode. SQLJ zal dan ook zeer herkenbaar zijn voor een klassieke databaseontwikkelaar en biedt de mogelijkheid om met statische SQL te werken. Zie ook: http://www.sqlj.org/.

SQLJ-toepassingen worden geschreven in Java maar de SQL-instructies worden Java-loos ingebed in de applicatie.

```
#sql { UPDATE rekeningen
   SET saldo = :saldo
  WHERE rekening_nr = :reknr }
```

De SQL-instructies worden via een SQLJ precompiler (of translator) vertaald naar standaard Java broncode met JDBC. Er

wordt ook een database toegangsmodule (profile) gegenereerd. Deze module kan gebruikt worden om de SQL instructies te 'bind'-en in een package (SQLJ profile customizer).

De gelijkenis met Embedded of Static SQL in klassieke programmeertalen (COBOL, C,...) is natuurlijk overduidelijk, maar we mogen een belangrijk verschil toch niet uit het oog verliezen. De klassieke precompiler zal SQL van programmacode scheiden en het programma kan slechts uitgevoerd worden wanneer de SQL gebind wordt in de database. De SQLJ-code daarentegen wordt tijdens de precompilatie herschreven tot een volwaardig JDBC-programma die dynamisch toegang tot de database kan krijgen. Daarnaast wordt de pure SQL ook weggeschreven in een profile-bestandje (vergelijkbaar met een DBRM). Dit bestand moet dan gebruikt worden om de applicatie in een package te binden. Alleen wanneer we de bind uitvoeren wordt de SQL statisch uitgevoerd; indien de bind vergeten wordt, gaat het programma niet in de fout en wordt alle SQL dynamisch uitgevoerd.

SQLJ - connectie naar de database

Bovenop een connectie naar de database moet in dit geval ook een '*ConnectionContext*' gemaakt worden. De connectie naar de database gebeurt op dezelfde manier als bij JDBC. Het creëren van de context gebeurt op basis van de connectie. De context wordt gebruikt om elk SQL-statement per connectie te groeperen.

```
#sql  [myContext] { UPDATE
 rekeningen
 SET saldo = :saldo
 WHERE rekening_nr = :reknr
} ;
#sql  [yourContext] { INSERT
 INTO audit(...) VALUES ...}
```

Voor verschillende contexten, dus mogelijk verschillende connecties naar de database, gaat de precompiler dan verschillende 'profile'-bestandjes aanmaken. Nadien kan elke groep SQL-statements (dus elk 'profile' bestand) dan apart aan de juiste database ge-'bind' worden.

SQLJ - SQL

Het schrijven van SQL is nu volledig gescheiden van de Java code, de embedded SQL is nu database afhankelijk en dus net hetzelfde als embedded SQL in elke andere programmeertaal.

SQLJ - foutenafhandeling e.a.

Ook in het geval van SQLJ zijn de nodige Java-klassen en methodes voor foutenafhandeling, locking-beheer,... voorzien.

SQLJ - opmerking

SQLJ is dus geen garantie voor het feit dat je statische SQL gebruikt. Zoals eerder gezegd zullen bij het vergeten van de bind, de SQL statements dynamisch uitgevoerd worden. Maar ook indien de applicatie gebind is, kan nog altijd dynamisch SQL uitgevoerd worden. Het is namelijk mogelijk SQLJ- en JDBC-SQL-statements in één Java-programma te mengen. Enkel de SQLJ-SQL kan dan gebinded worden. De JDBC-SQL blijft altijd dynamische SQL.

Mapping en persistentie met EJB

Gie Indesteege (ABIS)

Een goede SQL-ontwikkelaar kan vlot de juiste SQL-instructies formuleren om gegevens in de database te benaderen. Maar hoe zit het met de andere aspecten van een toepassing: verwerken van gegevens, controleren van gegevens, toegang tot de gegevens, synchronisatie van gegevens, ...

Bij het persisteren van gegevens in de database vanuit de objecten in de uitvoeringsomgeving is een van de grootste problemen het eenduidig afbeelden (mappen) van deze objecten op de database structuur. Attributen van objecten laten overeenstemmen met kolommen uit één of meerdere tabellen kan rechtstreeks in de SQL-instructie aangeduid worden, of via een mappingschema vastgelegd worden.

Het beheersen van gegevens kan best gecoördineerd verlopen aan de hand van een duidelijke architectuur. Java biedt hiervoor een aantal mogelijkheden, waarvan we in dit artikel de belangrijkste twee architecturen willen bespreken: Enterprise Java Beans (EJB) en Java Data Objects (JDO).

EJB - J2EE architectuur

Enterprise Java Beans zijn server-side componenten, die onder controle van een *EJB-container* de nodige functionaliteit en datatoegang kunnen realiseren voor de toepassing.

De Java 2 Enterprise Edition (J2EE) biedt niet alleen een aantal API's aan voor het bouwen van enterprisetoepassingen, maar levert ook de nodige voorzieningen om de uitvoering van deze toepassingen gecontroleerd te laten verlopen. De J2EE-architectuur beschrijft de componenten om

- toepassingen in productie te zetten, het zogenaamde deployment, aan de hand van configuratieparameters;
- de uitvoeringsomgeving of runtime te controleren. Dit gebeurt op basis van de configuratie vastgelegd bij deployment en de voorziene infrastructuur van de toepassingsserver.

Er zijn drie soorten enterprise beans vastgelegd in de EJB-specificatie:

- *session beans*: controleren de conversatie of sessie met de gebruiker van de toepassing.
- *entity beans*: zorgen voor de persistentie met de database (of datasource).

- *message (driven) beans*: verwerken asynchroon boodschappen van allerhande messageleveranciers.

De EJB-container zorgt, alweer volgens de EJB-specificatie, voor allerhande beheersaspecten zoals:

- resource pooling naar o.a. databases
- transactioneel beheer
- beveiliging
- netwerkdistributie van remote clients
- verdeling van de werkbelasting
- schaalbaarheid

EJB's - databases

Vermits wij hier voornamelijk geïnteresseerd zijn in de gegevenstoegang, zetten wij even de verschillende mogelijkheden die EJBs hiertoe bieden, op een rijtje.

Vanuit een *session bean* kunnen gegevens rechtstreeks benaderd worden d.m.v. JDBC- of SQLJ-instructies. Dit biedt het voordeel dat de ontwikkelaar zelf de 'meest optimale' SQL-instructie schrijft, maar toch beroep doet op de beheersaspecten van de EJB-container.

Entity beans, die bedoeld zijn om de logische objecten te persisteren, bieden echter de meeste mogelijkheden voor het definiëren van database toegang.

- *Container Managed Persistence* (CMP) entity beans worden volledig gecontroleerd door de EJB-container; zowel voor wat betreft het genereren van de SQL-vraag, aan de hand van een mappingschema, alsook de volledige automatische synchronisatie met de database. De te persisteren informatie wordt gedefinieerd via CMP-velden (~= tabelkolommen) en Container Managed Relaties (CMR) (~= tabelrelaties). De persistentie code wordt gegenereerd wanneer de entity bean wordt gedeployed in de EJB-server.

- *Bean Managed Persistence* (BMP) entity beans staan dan weer zelf in voor de SQL-vraag en het controleren van de synchronisatie met de database, maar laten toe om complexere SQL-vragen te formuleren.

- *Message beans* worden niet rechtstreeks gebruikt voor database toegang.

Het gebruik van JDBC en SQLJ werd in een vorig artikel reeds uit de doeken gedaan. Hier zullen we wat dieper ingaan op de mogelijkheden van CMP entity beans die, transparant voor de ontwikkelaar, de databasetoegang zullen realiseren. Dit gebeurt aan de hand van een mappingschema dat vastgelegd wordt via de zogenaamde *deployment descriptor*, en door middel van de specifieke vraagtaal *EJBQL* (EJB Query Language - vastgelegd in de EJB 2.0 specificatie).

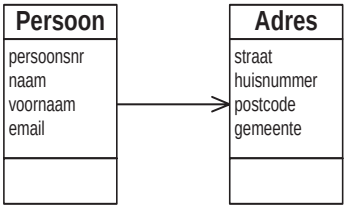
Het schrijven van de EJBQL kan handmatig gebeuren,of gebruik makend van J2EE-hulpmiddelen die door verschillende leveranciers van EJB-servers (IBM WebSphere, Oracle Internet Application Server) aangeboden worden.

Vermits de EJB-container instaat voor het (transparant) opzoeken van de gegevens uit de database en het creëren en/of opvullen van de overeenkomende objecten, volstaat het om de juiste logische zoekopdrachten te formuleren. Bij het definiëren van de EJB-container moet er natuurlijk een koppeling gedefinieerd worden naar de juiste database(s) d.m.v. een datasource-definitie.

De EJBQL definieert zoek- en selectiemethodes, alsook de navigatie over entity bean relaties heen (*Container Managed Relations*).

EJB - een voorbeeld

Figuur 1: Een personenregistratiesysteem



Het beschrijven van de EJB-structuur en het gebruik ervan door een Java-applicatie zou ons veel te ver leiden. Daarom proberen we de werking ervan te illustreren aan de hand van een klein praktisch voorbeeld, gebaseerd op een personenregistratiesysteem (zie figuur 1 en voorbeeld 1).

Voorbeeld 1: Container-managed relatie

```
package be.abis.ejb.db2exploring;
public abstract class PersoonBean implements javax.ejb.EntityBean {
private javax.ejb.EntityContext myEntityCtx;

public abstract int getPersoonsNr();
public abstract void setPersoonsNr(int newPersoonsNr);
public abstract String getNaam();
public abstract void setNaam(String newNaam);
 ...
```

De entity bean 'Persoon' heeft een associatie met een entity bean 'Adres', die als ondergeschikt ('dependency') wordt vastgelegd via een container-managed relatie. Alle beans worden volgens een *abstract persistentieschema* gedefinieerd Naast de pure datastructuur kan er natuurlijk ook nog bedrijfslogica worden gedefinieerd in de entity beans.

De concretisering van de container-managed persistentie en de container-managed relatie gebeurt via de *XML-deployment descriptor*, bij het deployen van de entity beans - zie voorbeeld 2.

Voorbeeld 2: Deployment descriptor

```
<ejb-jar id="ejb-jar_ID">
   <display-name>TestPersoonEJB</display-name>
   <enterprise-beans>
     <entity id="Persoon">
       <ejb-name>Persoon</ejb-name>
       ...
       <persistence-type>Container</persistence-type>
       <prim-key-class>be.abis.ejb.db2exploring.PersoonKey
       </prim-key-class>
       <cmp-version>2.x</cmp-version>
       <abstract-schema-name>Persoon</abstract-schema-name>
       <cmp-field><field-name>persoonsNr</field-name></cmp-field>
       <cmp-field><field-name>naam</field-name></cmp-field>
     </entity>
     ...
   </enterprise-beans>
   <relationships>
     <ejb-relation>
       <description>persoon heeft adres</description>
       <ejb-relation-name>Persoon-Adres</ejb-relation-name>
       <ejb-relationship-role>
        <ejb-relationship-role-name>adres</ejb-relationship-role-name>
          <multiplicity>One</multiplicity>
          <relationship-role-source>
             <ejb-name>Persoon</ejb-name>
          </relationship-role-source>
          <cmr-field>
             <cmr-field-name>adres</cmr-field-name>
          </cmr-field>
       </ejb-relationship-role>
     </ejb-relation>
   </relationships>
</ejb-jar>
```

Bij het deployen worden de concrete Javaklassen gegenereerd op basis van de abstracte entity bean definities en de deployment descriptor. Deze concrete klassen zorgen voor de databasetoegang en synchronisatie bij uitvoering, alsook voor het bewaken van de relaties tussen de klassen. Dit alles in een robuuste transactionele context, gecontroleerd door de EJB-container in samenwerking met de persistentiemanager.

EJB - EJBQL

Het opzoeken en gebruiken van entity bean objecten gebeurt aan de hand van find-methodes die in de home-interface van de bean worden vastgelegd - voorbeeld 3 geeft u een kort overzicht en wat meer details. Op basis van deze interface kan dan de implementatie gebeuren door middel van de EJBQL, die ook wordt opgenomen in de deployment descriptor.

Voorbeeld 3: Find-methode

```
public interface PersoonHome extends javax.ejb.EJBHome {
      ...
      public Persoon findByPrimaryKey(Integer persoonsNr)
                  throws RemoteException, CreateException;
      public Collection findByPostNummer(String postNummer)
                  throws RemoteException, CreateException;
}
```

Voor het opzoeken via de persoonsNr, die als primaire sleutel gedefinieerd is in de DB2-tabel, moet er vanzelfsprekend geen definitie gemaakt worden. Voor het opzoeken via de postcode daarentegen kunnen we volgende EJBQL definiëren.

```
FROM adres
 WHERE adres.postCode = ?1
```

Dit wil zoveel zeggen als: "selecteer alle Persoon beans met een postcode gelijk aan het postCode-argument". Zoals u ziet wordt er geen SELECT-clausule geschreven in de find-methodes.

Naast de find-methodes kunnen er ook **select-methodes** geïmplementeerd worden om, binnen de context van de entity bean, informatie op te halen voor de bedrijfslogica.

Tenslotte moet er natuurlijk nog een client-toepassing gebouwd worden, die gebruikt maakt van deze entity beans en hun methodes. Maar dat is een 'puur' Javaverhaal waar geen database meer aan te pas komt.

EJB - besluit

Dit eenvoudige voorbeeld is op het eerste gezicht niet spectaculair. Toch toont het aan dat het definiëren van bedrijfslogica kan losgekoppeld worden van de SQL-vragen. Of met andere woorden: een EJB-programmeur kan zich concentreren op de Javatoepassing, en de deployer, c.q. DBA, is verantwoordelijk voor de koppeling van de businessobjecten aan de bedrijfseigen database. Het definiëren van deze koppeling gebeurt in de beschrijving van de CMP- en CMR-velden in de deployment descriptor, en in het vastleggen van de EJBQL find- en select-methodes.

De generatie van de SQL en het synchroniseren van de databasetoegang gebeurt volledig onder controle van de EJB-container tijdens de uitvoering.

Verdere info over EJB CMP en EJBQL is te vinden op volgend URL: *http://java.sun.com/products/ejb/*.

Mapping en persistentie met JDO

Eric Seynaeve (ABIS)

Het opzetten van een persistentieframework is een hele klus. En misschien is de J2EE-infrastructuur met EJB's wel te zwaar voor de beoogde toepassing. Een alternatief wordt geboden door JDO (Java Data Objects).

JDO is een architectuur die, net zoals CMP entity beans, voorziet in een standaardmogelijkheid om transparant Javaobjecten te persisteren in een database. JDO kan ingezet worden in allerlei soorten Javatoepassingen: J2SE, web, servers. Maar JDO voorziet zelf geen gevorderde mogelijkheden voor gedistribueerde toepassingen. Transacties en connection pooling worden daarentegen wel voorzien. Hierdoor is JDO een veel lichtere technologie die voor minder zware toepassingen een uitkomst kan bieden. Verder kan JDO geïntegreerd worden met een EJB-container, die dan de diensten die JDO niet of beperkt voorziet, voor zijn rekening neemt. Binnen een EJB-container kan JDO gebruikt worden als vervanging van CMP entity beans, als aanvulling op BMP entity beans of rechtstreeks binnen session beans, dus zonder gebruik te maken van entity beans.

JDO is vooral bedoeld voor de persistentie van lokale, uitgewerkte objectmodellen. Het grote voordeel van deze technologie is dat de Javacode niet, of zeer weinig, aangepast moet worden om gebruikte objecten te kunnen persisteren.

JDO - mapping

De belangrijkste opdracht voor JDO is het afbeelden van de Javaobjecten op databasetabellen en -kolommen. Hiervoor worden definities aangemaakt in de vorm van XML-metadata in een .jdo-bestand. De JDO-implementatie voorziet in de mogelijkheid van een 'class file enhancement tool' dat op basis van het .jdo-bestand de klassefiles aanpast zodanig dat deze gepersisteerd kunnen worden. Voor deze mapping zijn er onder andere volgende XML-tags voorzien:

- <class> voor elke Javaklasse
- <field> voor elk veld van een Javaklasse
- <extension> voor vendor-specifieke uitbreidingen

In voorbeeld 1 herbruiken we het persoonsregistratiesysteem van het vorige artikel. De code van de Persoon- en Adresklasse loopt zeer gelijk met deze van de CMP entity beans. De Persoonklasse kent een veld 'persoonsNr' dat overeenkomt met de primaire sleutel. De Javaapplicatie moet deze sleutel zelf beheren. Voor de Adresklasse laten we de primaire sleutel genereren en beheren door JDO. Deze klassen kunnen op de gekende object-georiënteerde manier gebruikt worden.

Het inzetten van JDO in een toepassing is gebaseerd op een API met als belangrijkste component de 'PersistenceManager', in ons voorbeeld 'pm' genoemd. Deze biedt ons onder andere de mogelijkheid om met transacties te werken en om aan te geven welke objecten gepersisteerd moeten worden.

Een typische applicatie zal eerst de huidige transactie opvragen en de transactie beginnen. Daarna worden de te persisteren objecten aangemaakt en doorgegeven aan de PersistenceManager zodat deze ze kan markeren om te persisteren. Verrassend is dat het Adresobject niet aangeduid wordt. Dit zal toch gepersisteerd worden omdat het `a1` deel uitmaakt van het Persoonobject `p1`. Tot slot wordt de transactie gecommitted.

Voorbeeld 1: JDO

```
Transaction t = pm.currentTransaction();
t.begin();
...
Adres a1 = new Adres("Straat 1", "Nummer 1", "Postcde 1", "Gemeente 1");
Persoon p1 = new Persoon(1, "Naam 1", "Voornaam 1", "Email 1", a1);
pm.makePersistent(p1);
...
t.commit();
```

Om de mapping te kunnen maken, moeten we onderscheid maken tussen de twee richtingen die hier bestaan: Java naar relationeel versus relationeel naar Java.

- *Java naar relationeel*. Hierbij wordt een bestaande Javatoepassing met behulp van JDO uitgebreid om de mogelijkheid te bieden naar een database te persisteren. Aangezien er nog geen tabellen bestaan in de database, kan de specifieke JDO-implementatie zelf de tabellen genereren. De exacte naamgeving van de tabellen en kolommen wordt in de JDO-specificatie vrijgelaten en kan dus verschillen van vendor tot vendor. De generatie van de tabellen gebeurt aan de hand van de beschrijvingen in het .jdo-bestand. Omdat mengeen rekening moet houden met een reeds bestaand schema, kan het .jdo-bestand zeer eenvoudig gehouden worden.

```
<class name="Persoon"
identity-type="application"
>
<field name="persoonsNr"
primary-key="true" />
</class>
<class name="Adres" />
```

De specifieke JDO-implementatie gaat nu zelf de relaties tussen de tabellen leggen. Voor klassen waar geen primaire sleutel voor opgegeven is, zal de JDO-implementatie zijn eigen kolom aanmaken en volgens zijn eigen implementatie de primaire sleutelwaarden beheren.

- *Relationeel naar Java*. Voor deze richting vertrekken we vanuit een bestaand schema en worden de Javaklassen naar dit schema gemapt. De JDO 1.0 specificatie biedt deze mogelijkheid aan, maar enkel via de <extension> tag. De verschillende JDO-implementaties kunnen met behulp van deze tag hun eigen uitbreidingen op de JDO-

specificatie aanbieden. Hierdoor zal de specifieke manier waarop de mapping van relationeel naar gebeurt, vendor-afhankelijk zijn, als deze mogelijkheid al door de JDO-vendor aangeboden wordt. De meeste commerciële JDO-implementaties bieden deze mogelijkheid aan, samen met tools om de benodigde .jdo-bestanden, en eventueel ook Javaklassen, automatisch te genereren.

JDO - zoekopdrachten uitvoeren

Het ondervragen van de objecten gebeurt door middel van de *Java Data Objects Query Language (JDOQL)*, die toelaat om het objectenmodel te ondervragen i.p.v. de databasetabellen. De JDOQL wordt door het JDO-framework vertaald naar de juiste SQL voor de database. JDOQL is een vraagtaal die sterker aanleunt bij Javacode dan bij SQL. Voorbeeld 2 illustreert het gebruik van één parameter.

Voorbeeld 2: JDOQL

```
String filter =
"persoonsNr == nummer";
String parameterBeschrijving = "long nummer";
Long parameter = new Long(2);
Query q = pm.newQuery(Persoon.class, filter);
q.declareParameters
(parameterBeschrijving);
Collection c = (Collection)q.execute(parameter);
/* loop van voor naar achter door de Collection c en haal alle
geselecteerde objecten één voor één op */
```

Het is uiteraard ook mogelijk om meerdere parameters op te geven, alsook variabelen. Het bespreken van alle mogelijkheden valt echter buiten het bestek van dit artikel.

JDO - besluit

Deze eenvoudige voorbeelden laten maar een klein deel van de mogelijkheden zien van JDO. Toch is het duidelijk dat, aangezien de mapping in een extern .jdo-bestand gebeurt, veranderingen in de database doorgevoerd kunnen worden zonder de Javacode aan te hoeven passen. Het is echter jammer dat de JDO-standaard bij de mapping van relationeel naar Java veel overlaat aan de JDO-vendor. Hierdoor zal men zeer goed moeten opletten bij het aankopen van een specifieke JDO-implementatie.

Meer informatie over JDO is te vinden via deze URL:
http://java.sun.com/products/jdo/.

Java en DB2: bedenkingen van een DBA

Eric Venmans (ABIS)

Reacties van een conservatieve DBA

- Java applicatieontwikkelaars zijn 'cowboys'. Ze denken dat ze de database voor zich alleen hebben. Locking, concurrency, unit-of-work, ... daar hebben ze nooit van gehoord, zo lijkt het toch.

- Hebben we eindelijk DB2 getuned gekregen voor onze online-transacties die controleerbare 'static SQL' gebruiken en daar komen de Java boys af met 'dynamic SQL'. Aparte databases, daar moeten ze het mee stellen. Synchronisatie met productiegegevens? Een zorg voor later.

- Zie je het al gebeuren: een zware select met Repeatable Read; enkele minuten een productietabel gelocked; en alle transacties die updates willen doen knallen eruit met een time-out. Ik moet er niet aan denken.

- Distributed transacties? Mooie theorie dat two-phase commit protocol, maar de 'indoubts' en hun 'recovery' zijn toch maar mooi voor de DBA.

- Performance problemen? 't Zal weer de schuld van de database zijn. Mogen wij weer indexen bijmaken die het onderhoud van onze data en databases alleen maar verzwaren terwijl de echte oorzaak 'slechte SQL' en 'communicatie-overhead' nauwelijks ter sprake komen.

Reacties van een voorzichtige DBA

- Lezen met 'Uncommitted Read' kan zonder problemen

- Als de applicatielogica 'updates met autocommit' toelaat, kunnen we ze aanvaarden

- Als een 'unit-of-work' met meerdere updates kan verpakt worden als een 'stored procedure', kan die gemaakt en gebruikt worden op voorwaarde dat de call gebeurt met 'commit-on-return'

- Het gebruik van 'static SQL' via SQLJ gaan we aanmoedigen; we kunnen dan de EXPLAIN-functie gebruiken om toegangspaden te controleren.

DOSSIER 7

Overal unions of change all subselect fullselect

Waarom moest men tot en met V6 van DB2 UDB voor OS/390 steeds weer het verschil tussen een subselect en een fullselect voor ogen houden? Omdat een fullselect zowat alleen maar bruikbaar was in het select statement zelf. De andere SQL statements zoals een CREATE VIEW, UPDATE, DELETE, of subqueries in predicaten en nested table expressions moesten het stellen met een subselect,... dus geen union of union all keyword.

Dit is een voorbeeld van een view gebaseerd op een fullselect:

```
create view unione as
select * from personsa
where psex = 'F'
union
select * from personsb
where psex = 'F' ;

select count(*) from unione
```

Dezelfde query kan ook uitgevoerd worden als een nested table expression:

```
select count(*) from
( select *  from personsa
  where psex = 'F'
  union
  select *  from personsb
  where psex = 'M')
  as tab
```

Fullselects kunnen vanaf V7 gebruikt worden in basic predicaten, quantified predicaten, 'exists' en 'in' predicaten, alsook in 'insert', 'delete' en 'update' statements. Denk er wel aan dat zo'n views niet updatable zijn. In het voorbeeld ziet men ook een count functie die op de view uitgevoerd wordt. Op zich niets wonderbaarlijks, maar het toont aan dat problemen bij het berekenen van kolomfuncties over meerdere tabellen opgelost zijn.

Net zoals bij een gewone view zal DB2 de predicaten uit het CREATE view statement en diegene geformuleerd bij het selecteren van data uit de view combineren. Wanneer een view bestaat uit een union (all) van twee subselects, dan zal DB2 deze selects afzonderlijk uitvoeren, het resultaat samenvoegen en afhankelijk van de query nog bijkomende operaties uitvoeren op het tussenresultaat (e.g. sorteren). Dit alles is zichtbaar in de EXPLAIN output. Of bij dit proces materialisatie plaatsheeft, is dankzij de nieuwe kolom TABLE_TYPE in de plan_tabel rechtstreeks afleesbaar: 'W' duidt materialisatie aan, bij 'Q' niet. Om duidelijk de verschillende fazes in de uitvoering en hun volgorde te zien, werd de kolom PARENT_QBLOCKNO toegevoegd en zijn er nieuwe waarden voor de kolom QBLOCK_TYPE (TABLEX, UNIONA, UNION). In een volgend nummer zal verder ingegaan worden op het gebruik van UNION in de create view syntax en zal dit vergeleken worden met het gebruik van partities.

Katrien Platteborze (ABIS)

CURSUSPLANNING APR - MEI - JUN 2003

DB2 for OS/390, een totaaloverzicht	1625 EUR	07-11/04 (W), 22-28.05.2003 (W), 02-06/06 (L)
DB2 UDB, een totaaloverzicht	1625 EUR	22-23/5 & 02-04/06 (W)
RDBMS concepten	325 EUR	24/04 (W), 22/05 (W), 02/06 (L)
Basiskennis SQL	325 EUR	25/04 (L), 23/05 (W), 03/06 (L)
DB2 for OS/390 basiscursus	975 EUR	28-30/04 (L), 26-28/05 (W), 04-06/06 (L)
DB2 UDB basiscursus	975 EUR	28-30/4 (L), 02-04/06 (W)
DB2 UDB concepten	375 EUR	05/06 (L)
SQL workshop	700 EUR	22-23/4 (W), 12-13/06 (W), 16-17/06 (L)
DB2 for OS/390 programmering voor gevorderden	1050 EUR	19-21/05 (L)
DB2 for OS/390: SQL performance	1200 EUR	11-13/06 (L)
Fysiek ontwerp van relationele databases	700 EUR	28-29/04 (L)
DB2 for OS/390 database administratie	1600 EUR	12-15/05 (L)
DB2 UDB database administratie	1600 EUR	23-26/06 (L)
DB2 UDB systeembeheer en performance	400 EUR	09/05 (L)
DB2 for OS/390 V7 upgrade voor ontwikkelaars	375 EUR	06/06 (L)
DB2 UDB en zijn extenders: XML en text search	200 EUR	16/05 (L)
DB2 UDB integratie met MQSeries	200 EUR	16/05 (L)

Plaats: L = Leuven; W = Woerden
Details, andere data en bijkomende cursussen: www.abis.be

Postbus 220
Diestsevest 32
BE-3000 Leuven
Tel. 016/245610
Fax 016/245691
training@abis.be

abis
TRAINING & CONSULTING

Postbus 122
Pelmolenlaan 1-K
NL-3440 AC Woerden
Tel. 0348-435570
Fax 0348-432493
training@abis.be