

De JDBC (Java DataBase Connectivity) API is het onderdeel van de Java SE dat definieert hoe je vanuit Java via SQL met de database communiceert. Het bestaat uit een set interfaces en classes in de packages `java.sql` en `javax.sql` en is databaseonafhankelijk. Database-vendors als Oracle en IBM, maar ook partijen als DataDirect leveren databasespecifieke drivers met de implementatie voor de low-level communicatie en vertaling van datastructuren, deels uitgebreid met bijvoorbeeld met databasespecifieke functionaliteit.

JDBC 4.0

Nieuwe functionaliteit en meer gebruiksgemak

Met het uitkomen van Java SE 6, eind 2006, is ook JDBC bij een nieuwe versie aangekomen. De doelstelling van JSR-221 die aan deze versie 4.0 ten grondslag ligt was 'to provide an improved developer experience while working with relational SQL data stores from the Java platform'. Dit vertaalt zich enerzijds naar een betere ondersteuning voor databasefeatures en anderzijds naar een hoger gebruiksgemak voor de programmeur. Niet onbelangrijk is het feit dat het volledig backwards compatible is. In dit artikel wordt ingegaan op een aantal van de nieuwe features van JDBC 4.0. Ook zal er kort aandacht besteed worden aan JavaDB, de relationele database die samen met Java SE gedistribueerd wordt.

JDBC Drivers

De DriverManager ondersteunt het nieuwe Java SE Service Provider-mechanisme. Hierdoor hoeft de JDBC-driver niet meer expliciet geladen te worden met de `Class.forName()` method, maar wordt deze automatisch geladen wanneer een connectie opgevraagd wordt:

```
Connection conn =
    DriverManager.getConnection(jdbcUrl, user, password);
```

Een vendor configureert zijn driver door in de jar-file de file `META-INF/services/java.sql.driver` op te nemen met daarin de naam van de implemen-

tatie-class, bijvoorbeeld `org.apache.derby.jdbc.EmbeddedDriver` of `oracle.jdbc.OracleDriver`. Vanzelfsprekend moet de driver-library in het classpath staan. Voor backwards compatibility kan de `forName` methode nog steeds gebruikt worden. Sowieso blijft de aanbeveling om, indien mogelijk, een datasource te gebruiken.

De nieuwe `javax.sql.Wrapper` interface is bedoeld voor driver-vendors om databasespecifieke functionaliteit af te schermen en gestandaardiseerd beschikbaar te stellen. De interface wordt onder andere geïmplementeerd door de `ResultSet`, `DataSource`, `Statement` en `DatabaseMetaData`.

Connections en Statements

JDBC 4.0 biedt ook een aantal uitbreidingen dat niet direct voor de programmeur van belang is, maar meer van toepassing is voor driver-vendors en ORM-frameworks. Aan de interfaces `Connection` en `Statement` zijn nieuwe methoden toegevoegd, als `isValid` (al dan niet met timeout), `isPoolable`, `setPoolable` en `isClosed`, waarmee deze beter gemanaged kunnen worden, met name in pooled condities. Via de interface `java.sql.DatabaseMetaData` kan allerlei informatie over de database (en de driver) opgehaald worden, zoals de databasenaam, de versie, de JDBC-versie maar ook de databasecatalogus, de naam daarvan, of de database in read-only mode is, of de database kolom aliasing ondersteund, het maximum aantal

Aino Andriessen

is Software Architect bij AMIS Services BV in Nieuwegein.

Hij is bereikbaar via aino.andriessen@amis.nl en blogt regelmatig op <http://technology.amis.nl/blog>.

kolommen in een select-statement et cetera. Deze interface is met name bruikbaar voor Framework-developers en voor applicaties die met verschillende databases werken. De interface is uitgebreid met nieuwe methoden als *getSchemas*, *getFunctions*, *getRowIdLifetime*, *supportsStoredFunctionsUsingCallSyntax*.

De nieuwe get- en setClientInfo-methoden op de Connection-interface zijn van belang om specifieke client-informatie aan de database door te geven zodat deze in de database uniek geïdentificeerd kan worden voor bijvoorbeeld logging- en tracing-toepassingen. Er zijn drie standaard properties, *ApplicationName*, *ClientUser*, *ClientHostName* en daarnaast eventueel nog databasespecifieke properties. Deze feature wordt niet door Derby ondersteund, wat trouwens voor de embedded mode (zie hierna) toch niet zo van belang is.

SQLXML

De laatste versie van de ANSI/ISO SQL-standaard, SQL:2003, bevat onder andere XML-gerelateerde features als de SQL/XML query-syntax en het XML-datatype. Data kunnen daarmee als XML opgevraagd, gemanipuleerd en opgeslagen worden. De standaard wordt tegenwoordig door vrijwel alle databases ondersteund. Voorbeeld van een SQL/XML-query:

```
select xmlelement
("employee"
, xmlelement ("name", e.ename)
, xmlelement ("job", e.job)
) empasxml
from emp e
where e.job = 'CLERK'
```

Deze query levert een XML-datatype op. Voorheen kon dit resultaat alleen via een omweg in Java gebruikt worden, namelijk door het te converteren naar String of Clob. De DataDirect-driver voert de conversie onder water uit, maar met de meeste andere drivers zal de conversie in de SQL-query moeten gebeuren, bijvoorbeeld:

```
select xmlserialize
(document xmlelement ( "name", e.ename) as
VARCHAR2(100))
from emp e
```

Met JDBC 4.0 hoeft er dus geen conversie meer uitgevoerd te worden, maar kan het resultaat van de XML-query direct uit de ResultSet opgehaald worden:

```
java.sql.SQLXML sqlxml = rs.getSQLXML(1);
```

De SQLXML-interface is feitelijk een logische pointer naar de XML-data in de database en is bovendien slechts valide gedurende een transactie. De XML zal dus opgehaald en 'geconverteerd' moeten worden in een formaat dat voor Java han-

teerbaar is. Daarvoor is de interface voorzien van methoden om de XML te benaderen als String, Reader, InputStream, Source, Writer en Result al dan niet in combinatie met een DOM, SAX of StAX parser of XSLT-transformatie. Het is van belang je te realiseren dat de methoden van het SQLXML-object eenmalig aangeroepen kunnen worden. Een nieuwe poging wordt beloond met een SQLException.

Voorbeeld om de XML te parsen met een DOM parser:

```
DocumentBuilder parser =
DocumentBuilderFactory.newInstance().newDocument-
Builder();
Document result = parser.parse(sqlxml.getBinaryS-
tream());
```

Vanwege de XML-implementatie in de database kan het gebruik van de *getSource*-methode performancevoordelen opleveren boven serialisatie naar een stream en het parsen van XML, het bovenstaande voorbeeld wordt dan:

```
DOMSource domSource = sqlxml.getSource(DOMSource.
class);
Document document = (Document) domSource.getNode();
```

Natuurlijk is het ook mogelijk om XML in de database op te slaan:

```
SQLXML sqlxml= connection.createSQLXML();
DOMResult domResult = sqlxml.setResult(DOMResult.
class);
domResult.setNode(node);
PreparedStatement pstmt = con.prepareStatement
("insert into xmldocuments (xmlidoc) values (?)");
pstmt.setSQLXML(1,sqlxml);
pstmt.executeUpdate();
```

Meer voorbeelden zijn te vinden in de javadoc van de `java.sql.SQLXML`-interface.

Hoewel vrijwel elke database het XML-datatype ondersteunt, moeten we helaas constateren dat momenteel nog vrijwel geen enkele driver de SQLXML-interface ondersteunt.

Datatypes

Nieuw in JDBC 4.0 is ondersteuning voor het rowid datatype, voor National Character Set datatypes, en uitgebreidere ondersteuning voor Blobs en Clobs.

Het rowid representeert het unieke (fysieke of logische) adres van een rij in een tabel. Ondersteuning hiervoor is aanwezig in de Oracle en DB2-database, maar niet in Derby en MySQL. Het rowid moet beschouwd worden als specifiek voor een datasource en is normaliter ook niet portable tussen verschillende datasources. Meer informatie over de geldigheidsduur van het rowid kan achterhaald worden met de methode *getRowIdLifetime* in de interface `DatabaseMetadata`.

Met deze methode kan ook bepaald worden of het rowid überhaupt ondersteund wordt. Nieuw zijn de `java.sql.RowId`-interface en de bijbehorende getters en setters in de `ResultSet` en `PreparedStatement`.

De National Character Set is een extra, Unicode, character set die op de database gedefinieerd kan worden als aanvulling op de bestaande (eventueel niet unicode) character set. Deze zijn als nieuwe `java.sql.Types` (onder andere `NCHAR`, `NCLOB`, `NVARCHAR`) en bijbehorende methoden in het `PreparedStatement`, `CallableStatement` en `ResultSet` beschikbaar.

De `CallableStatement`, `PreparedStatement`, `ResultSet` en `RowSet`-interfaces zijn uitgebreid met setters en getters voor `Clobs`, `Blobs`, `Streams` en `Readers`.

Hoewel niet nieuw in JDBC 4.0, is het toch de moeite waard om even de `javax.sql.Rowset` interfacehiërarchie aan te stippen. De `RowSet` is vergelijkbaar met een `ResultSet` en biedt rij-georiënteerde toegang tot tabulaire data en implementeert het `javaBean` componenten- en eventmodel. De subinterfaces bieden echter pas echt interessante functionaliteit. Waar de `ResultSet` alleen beschikbaar is gedurende een connectie kan de `CachedRowSet` ook `disconnected` bestaan. De data kan bovendien gewijzigd worden en weer teruggeschreven kunnen worden in de oorspronkelijke `datasource`, zoals in onderstaande code voorbeeld wordt gedemonstreerd:

```
PreparedStatement s =
conn.prepareStatement("select empno, ename from person");
ResultSet rs = s.executeQuery();
CachedRowSet crset = new
CachedRowSetImpl();
crset.populate(rs);
conn.close();
while (crset.next()) {
    System.out.println("crset.get-
String(2));
}
crset.first();
crset.updateString(2,"Jan Klaasen");
crset.updateRow();
conn = db.getConnection();
crset.setTableName("emp"); // required for
Oracle, not for Derby
crset.acceptChanges(conn);
crset.commit();
conn.close();
```

Andere subinterfaces van de `CachedRowSet` zijn de `WebRowSet` die XML-toegang tot de onderliggende data biedt en de `JoinRowSet` die een mechanisme biedt om verschillende `RowSets` te combineren.

SQLExceptions

Op het gebied van foutafhandeling zijn de nodige wijzigingen doorgevoerd. Het aantal

`SQLExceptions` (subclasses) is uitgebreid. De `SQL`-status, een nieuwe `SQL:2003` standaard returncode die de status van een `SQL`-statement aangeeft, wordt ondersteund. De `SQLExceptions` ondersteunen het `Java SE` chained exception-mechanisme waarmee verscheidene `SQLExceptions` in één `JDBC`-operatie afgehandeld kunnen worden. Dankzij de implementatie van de `Iterable` interface en de `enhanced for` loop kunnen deze snel doorlopen worden:

```
catch(SQLException ex) {
    for(Throwable e : ex) {
        System.out.println(e);
    }
}
```

De `SQLExceptions` zijn in drie categorieën (subclasses) verdeeld :

- **Transient Exceptions.** Betreft een foutsituatie waarbij het opnieuw uitvoeren van dezelfde actie wel succesvol zou kunnen zijn, bijvoorbeeld in het geval van een timeout of een database rollback.
- **Recoverable Exception.** Dit betreft ook een foutsituatie waarbij het opnieuw uitvoeren van de actie wel zou kunnen slagen. Hiervoor moet de applicatie wel wat recovery, minimaal het sluiten en openen van een connectie, uitvoeren.
- **Non Transient Exceptions.** Betreft de situatie waarbij dezelfde actie weer een exceptie zal opleveren, bijvoorbeeld als de `SQL`-syntax incorrect is of een constraint wordt overtreden.

JavaDB

Sinds `Java 6 SE` wordt `Apache Derby` onder de naam `JavaDB` meegeleverd met de `Java SE`. De relationele `JavaDB` is geschreven in `Java`, conformeert zich aan standaarden, kent een rijke functionaliteit, heeft een kleine footprint en is zowel embedded als in client-server mode te gebruiken. In client-server mode werkt het als een normale databaseserver. In de embedded mode vormt de database feitelijk een exclusief onderdeel van de applicatie voor de persistence van data via `JDBC`. Toegang tot de database is niet beperkt tot `Java`; er is een command-line `SQL`-client beschikbaar en voor de server mode zijn er ook administratieve tools.

Het gebruik van de embedded database in een `Java`-applicatie is erg eenvoudig. Voeg de library, `derby.jar` (<`JAVA_HOME`>\db\lib), toe aan het classpath en klaar is Kees. Vanuit `Java` wordt de database benaderd met `JDBC`. Er zijn in de embedded mode wel een paar speciale constructies om de database te creëren en te stoppen.

```
DriverManager.getConnection
("jdbc:derby:myDB;create=true", properties);
```

Hier een streamer rechts
Hier een streamer rechts
Hier een streamer rechts

Hiermee wordt in de current directory (of in de directory die gezet is met de systeemparameter `derby.system.home`) de directory `MyDB` aangemaakt met de Derby-database. Als de database al bestaat wordt er een connectie geopend.

```
try {
    DriverManager.getConnection("jdbc:derby::shutdown=true");
} catch (SQLException se) {
    System.out.println("Database shut down normally");
}
```

Typisch is dat er een exceptie gegooid wordt als de database succesvol gestopt is.

De JavaDB heeft al een lange geschiedenis, onder andere onder de naam Cloudscape. Het is een volwassen relationele database, gebaseerd op ANSI/ISO SQL-standaarden. Hij beschikt over vele moderne databasefeatures als (ACID) transacties, concurrency, online backup, export/import, unicode, Blobs and Clobs, procedures enzovoort en wordt (vanzelfsprekend) ondersteund ORM-frameworks als Hibernate en

Toplink. De embedded mode en de kleine footprint maken de JavaDB bovendien uitermate geschikt voor portable devices.

Samenvatting

JDBC 4.0 is een belangrijke major version. Hij implementeert nieuwe features van de SQL:2003-standaard, biedt meer controle en maakt het gebruik van JDBC voor de programmeur toegankelijker. Helaas is momenteel de ondersteuning door de drivers nog beperkt. De nieuwste driver-versies zijn al wel geschikt voor JDBC 4 via het Service Provider-mechanisme, maar ondersteunen nog slechts enkele features en in vrijwel geen enkel geval het SQLXML-datatype. Ook de ORM-frameworks als Hibernate en Toplink kennen nog geen ondersteuning. Met de toenemende inzet van Java 6 is de verwachting dat dit zal verbeteren.

Referenties:

JSR-221 - <http://jcp.org/en/jsr/detail?id=221>

Java Database technology - <http://java.sun.com/javase/technologies/database/>

Apache Derby - <http://db.apache.org/derby/>