iBATIS SQL Maps

Tutorial

Für SQL Maps Version 2.0

18. Februar 2006
Deutsche Übersetzung: Juni 2007

Deutsche Übersetzung von Joachim Rohde (webmaster@joachimrohde.de)

Einleitung

Dieses kurze Tutorial ist eine Schritt-für-Schritt Anleitung, die Ihnen den typischen Gebrauch von SQL Maps aufzeigt. Die Details zu den einzelnen Themen, die hier besprochen werden, finden Sie im SQL Maps Developer's Guide unter http://ibatis.apache.org.

Vorbereitung zur Nutzung von SQL Maps

Das SQL Maps Framework ist sehr tolerant gegenüber schlechten Datenbank-Modellen und sogar gegenüber schlechten Objekt-Modellen. Dennoch ist es empfehlenswert, dass Sie sich an die Best Practices halten, wenn Sie ihre Datenbank-Schemata (entsprechende Normalisierung) und Ihre Objekt-Modelle entwerfen. Damit erhalten Sie eine gute Performance und ein sauberes Design.

Am einfachsten fängt man damit an, zu analysieren, womit Sie arbeiten. Was sind Ihre Geschäfts-Objekte? Was sind Ihre Datenbank-Tabellen. Wie sind diese miteinander verbunden? Als ein erstes Beispiel nehmen wir folgende einfache Klasse Person, welche dem typischen JavaBeans Pattern entspricht.

Person.java

```
package examples.domain;

//benötigte imports….

public class Person {
   private int id;
   private String firstName;              //Vorname
   private String lastName;               //Nachname
   private Date birthDate;                //Geburtsdatum
   private double weightInKilograms;  //Gewicht
   private double heightInMeters;         //Größe

   public int getId () {
      return id;
   }
   public void setId (int id) {
      this.id = id;
   }

   //…um Platz zu sparen nehmen wir an, dass wir die anderen Setter und Getter haben…
}
```

Wie wird diese Klasse Person auf unsere Datenbank abgebildet? SQL Maps schränkt Sie bezüglich der Relationen, etwa Tabelle-pro-Klasse oder Multiple-Tabellen-pro-Klasse oder Multiple-Klassen-pro-Tabelle, nicht ein. Da Sie sämtliche Möglichkeiten haben, die Ihnen SQL bietet, gibt es nur sehr wenige Einschränkungen. Für unser Beispiel nehmen wir folgende einfache Tabelle, welche für eine Tabelle-pro-Klasse Beziehung geeignet ist:

Person.sql

```
CREATE   TABLE   PERSON(
   PER_ID              NUMBER        (5, 0)    NOT NULL,
   PER_FIRST_NAME      VARCHAR       (40)      NOT NULL,
   PER_LAST_NAME       VARCHAR       (40)      NOT NULL,
   PER_BIRTH_DATE      DATETIME                ,
   PER_WEIGHT_KG       NUMBER        (4, 2)    NOT NULL,
```

```
PER_HEIGHT_M        NUMBER          (4, 2)      NOT NULL,
PRIMARY KEY (PER_ID))
```

Die SQL Map Konfigurationsdatei

Sobald wir uns über die Klassen und Tabellen im Klaren sind, mit denen wir arbeiten möchten, ist die SQL Map Konfigurationsdatei, der richtige Startpunkt. Diese Datei dient als Basis-Konfiguration für unsere SQL Map Implementierung.

Die Konfigurationsdatei ist eine XML-Datei. In dieser werden wir Eigenschaften, JDBC DataSources und SQL Maps konfigurieren. Sie bietet die bequeme Möglichkeit, Ihre DataSource zentral zu konfigurieren, welche durch eine Vielzahl verschiedener Implementierungen realisiert worden sein kann. Das Framework kann mit verschiedenen DataSource Implementierungen umgehen, inklusive iBATIS SimpleDataSource, Jakarta DBCP (Commons) und jede andere DataSource Implementierung, welche über einen JNDI Kontext (z.B. aus einem Applikations-Server heraus) ermittelt werden kann. Darauf wird im Developer's Guide näher eingegangen. Die Struktur ist einfach und könnte für das obige Beispiel wie folgt aussehen:

Das Beispiel wird auf der nächsten Seite fortgesetzt…

SqlMapConfigExample.xml

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE sqlMapConfig
    PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<!-- Achten Sie immer darauf, den korrekten XML Header, wie oben verwendet, zu verwenden! -->

<sqlMapConfig>

  <!-- Die Eigenschaften (Name=Wert), die in der folgenden Datei spezifiziert sind, können als
       Platzhalter in dieser Konfigurations-Datei verwendet werden (z.B. "${driver}"). Die Datei liegt
       normalerweise relativ zum Klassenpfad und ist optional.  -->

  <properties resource="examples/sqlmap/maps/SqlMapConfigExample.properties" />

  <!-- Diese Einstellungen beeinflussen SqlMap Konfigurations Details, hauptsächlich solche, die
       etwas mit Transaktions-Management zu tun haben. Diese sind alle optional (mehr
       Informationen sind im Developers Guide zu finden).  -->

  <settings
   cacheModelsEnabled="true"
   enhancementEnabled="true"
   lazyLoadingEnabled="true"
   maxRequests="32"
   maxSessions="10"
   maxTransactions="5"
   useStatementNamespaces="false"
  />

  <!-- Type Aliases ermöglichen Ihnen, einen kürzeren Namen anstatt des vollen Klassennamens zu
       verwenden. -->

  <typeAlias alias="order" type="testdomain.Order"/>

  <!-- Konfiguration einer DataSource, die mit der SQL Map mittels SimpleDataSource verwendet
       werden soll.  -->

  <transactionManager type="JDBC" >
    <dataSource type="SIMPLE">
      <property name="JDBC.Driver" value="${driver}"/>
      <property name="JDBC.ConnectionURL" value="${url}"/>
      <property name="JDBC.Username" value="${username}"/>
      <property name="JDBC.Password" value="${password}"/>
    </dataSource>
  </transactionManager>

  <!-- Hier kommen alle SQL Map XML-Dateien hin, die von dieser SQL Map geladen werden sollen.
       Beachten Sie, dass die Pfade relativ zum Klassenpfad sind. Zur Zeit haben wir nur eine
       einzige... -->

  <sqlMap resource="examples/sqlmap/maps/Person.xml" />

</sqlMapConfig>
```

SqlMapConfigExample.properties

```
# Dies ist eine einfache Eigenschafts-Datei, welche die automatische Konfiguration der SQL Maps
# Konfigurations-Datei erleichtert (z.B. durch Ant oder Continuous Integrations Tools für andere
# Umgebungen, etc.)
# Diese Werte können in jedem Eigenschafts-Wert in der obigen Datei verwendet werden (z.B.
# "${driver}")
# Eine solche Eigenschafts-Datei wie diese hier, ist komplett optional.

driver=oracle.jdbc.driver.OracleDriver
url=jdbc:oracle:thin:@localhost:1521:oracle1
username=jsmith
password=test
```

Die SQL Map Datei(en)

Nun, da wir eine DataSource konfiguriert haben und unsere zentrale Konfigurationsdatei bereit ist, müssen wir uns um die eigentliche SQL Map Datei kümmern, welche unseren SQL Code und die Abbildung auf die Parameter- und Ergebnis-Objekte enthält (Eingabe respektive Rückgabe).

Wir fahren mit dem obigen Beispiel fort und erzeugen eine SQL Map Datei für die Klasse Person und die Tabelle PERSON. Wir fangen mit der üblichen Struktur eines SQL Dokuments und einem einfachem Select-Statement an.

Person.xml

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE sqlMap
   PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
   "http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Person">

    <select id="getPerson" resultClass="examples.domain.Person">
        SELECT
                PER_ID                  as id,
                PER_FIRST_NAME          as firstName,
                PER_LAST_NAME           as lastName,
                PER_BIRTH_DATE          as birthDate,
                PER_WEIGHT_KG           as weightInKilograms,
                PER_HEIGHT_M            as heightInMeters
        FROM PERSON
        WHERE PER_ID = #value#
    </select>

</sqlMap>
```

Das obige Beispiel zeigt die einfachste Form von SQL Map. Es benutzt ein Feature des SQL Maps Framework, welches automatisch die Spalten eines ResultSets auf die JavaBeans Eigenschaften abbildet (oder auf die *keys* eines Map-Objektes, etc.), basierend auf den entsprechenden Namen. Der Token #value# ist ein Eingabeparameter. Genauer gesagt, die Benutzung von "value" impliziert, dass wir einen einfachen Wrapper für einen primitiven Datentyp verwenden (z.B. Integer, wobei wir allerdings nicht auf diesen Datentyp beschränkt sind).

Obwohl der Ansatz sehr einfach ist, gibt es bei der Benutzung des automatischen Abbildens des Ergebnisses einige Einschränkungen. Es gibt keine Möglichkeit den Datentyp der Ausgabe-Spalten zu spezifizieren

(soweit das nötig ist) oder um zugehörige Daten (bei komplexen Objekten) automatisch mitzuladen und es gibt darüber hinaus einen geringfügigen Einfluss auf die Performance, da dieser Ansatz auf ResultSetMetaData zugreifen muss. Indem man eine resultMap verwendet, kann man diese Einschränkungen allerdings umgehen. Aber zur Zeit wollen wir es so einfach wie möglich halten und wir können später immer noch zu einem anderen Ansatz umschwenken (ohne den Java Sourcecode ändern zu müssen).

Die meisten Datenbank-Applikationen lesen nicht ausschließlich aus der Datenbank, sie müssen die Daten in der Datenbank auch verändern. Wir haben bereits ein Beispiel gesehen, wie ein einfaches SELECT-Statement aussieht, aber was ist mit INSERT, UPDATE und DELETE? Die gute Nachricht ist, dass es keinen Unterschied gibt. Folgend werden wir unsere Person SQL Map mit weiteren Statements komplettieren, um einen kompletten Satz an Statements zu haben, um Daten abzufragen und zu manipulieren.

Person.xml

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE sqlMap
   PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
   "http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Person">

    <!-- Benutze einen primitiven Wrapper-Typ (e.g. Integer) als Parameter und erlaube, dass die
         Rückgabewerte automatisch auf die (JavaBean) Eigenschaften des Person Objektes
         abgebildet werden. -->
    <select id="getPerson" parameterClass=”int” resultClass="examples.domain.Person">
        SELECT
                PER_ID                  as id,
                PER_FIRST_NAME          as firstName,
                PER_LAST_NAME           as lastName,
                PER_BIRTH_DATE          as birthDate,
                PER_WEIGHT_KG           as weightInKilograms,
                PER_HEIGHT_M            as heightInMeters
        FROM PERSON
        WHERE PER_ID = #value#
    </select>

    <!-- Benutze die (JavaBean) Eigenschaften des Objekts Person als Parameter beim Einfügen.
        Jeder Parameter zwischen den #Rauten# Symbolen ist eine JavaBean-Eigenschaft.  -->
    <insert id="insertPerson" parameterClass="examples.domain.Person">
        INSERT INTO
        PERSON  (PER_ID, PER_FIRST_NAME, PER_LAST_NAME,
                  PER_BIRTH_DATE, PER_WEIGHT_KG, PER_HEIGHT_M)
        VALUES (#id#, #firstName#, #lastName#,
                  #birthDate#, #weightInKilograms#, #heightInMeters#)
    </insert>

    <!-- Benutze die (JavaBean) Eigenschaften des Objekts Person als Parameter beim Aktualisieren.
        Jeder Parameter zwischen den #Rauten# Symbolen ist eine JavaBean-Eigenschaft.  -->
    <update id="updatePerson" parameterClass="examples.domain.Person">
        UPDATE PERSON
        SET PER_FIRST_NAME = #firstName#,
            PER_LAST_NAME = #lastName#, PER_BIRTH_DATE = #birthDate#,
            PER_WEIGHT_KG = #weightInKilograms#,
            PER_HEIGHT_M = #heightInMeters#
        WHERE PER_ID = #id#
```

```
    </update>

    <!-- Benutze die (JavaBean) Eigenschaft “id” des Objekts Person als Parameter beim Löschen.
         Jeder Parameter zwischen den #Rauten# Symbolen ist eine JavaBean-Eigenschaft.  -->
    <delete id="deletePerson" parameterClass="examples.domain.Person">
          DELETE PERSON
          WHERE PER_ID = #id#
    </delete>

</sqlMap>
```

Mit dem SQL Map Framework programmieren

Nun, da wir alles konfiguriert und die Abbildungen definiert haben, müssen wir nur noch unsere Java Applikation programmieren. Als ersten Schritt müssen wir die SQL Map konfigurieren. Das ist eine einfache Angelegenheit, da wir lediglich die SQL Map Konfigurations-XML-Datei, die wir erstellt haben, laden müssen. Um das Laden der XML-Datei zu vereinfachen, können wir von der Resources Klasse Gebrauch machen, die im Framework enthalten ist.

```
String resource = "com/ibatis/example/sqlMap-config.xml";
Reader reader = Resources.getResourceAsReader (resource);
SqlMapClient sqlMap =  SqlMapClientBuilder.buildSqlMapClient(reader);
```

Das SqlMapClient Objekt ist ein langlebiges, Thread-sicheres Service Objekt. Es muss im Lebenszyklus einer Applikation nur einmal instantiiert / konfiguriert werden. Dadurch haben wir einen guten Kandidaten für eine statische Member-Variable innerhalb einer Basis-Klasse (z.B. der Basis DAO Klasse) oder, wenn Sie es lieber zentraler konfiguriert und global verfügbar haben möchten, können Sie es in einer eigenen Wrapper-Klasse erzeugen. Hier ein Beispiel, wie so eine Klasse aussehen könnte:

```
public MyAppSqlConfig {

   private static final SqlMapClient sqlMap;

   static {
      try {
         String resource = "com/ibatis/example/sqlMap-config.xml";
         Reader reader = Resources.getResourceAsReader (resource);
         sqlMap =  SqlMapClientBuilder.buildSqlMapClient(reader);
       } catch (Exception e) {
         // Wenn hier ein Fehler auftritt, ist es egal, was für einer es war. Wir können nichts daran
         // ändern und wollen, dass die Applikation wirklich aussteigt, damit wir das Problem auch
         // wirklich mitbekommen. Sie sollten solche Fehler immer protokollieren und erneut so
         // werfen, dass Sie unverzüglich auf das Problem aufmerksam werden.
         e.printStackTrace();
         throw new RuntimeException ("Error initializing MyAppSqlConfig class.  Cause: " + e);
       }
   }

   public static SqlMapClient getSqlMapInstance () {
      return sqlMap;
   }
}
```

Objekte aus der Datenbank lesen

Nun, da wir eine SqlMap Instanz initialisiert haben und einfach darauf zugreifen können, können wir Gebrauch davon machen. Als erstes laden wir ein Person-Objekt aus der Datenbank. (Für dieses Beispiel nehmen wir an, dass es 10 PERSON Einträge in der Datenbank, mit PER_ID von 1 bis 10 gibt).

Um ein Person Objekt aus der Datenbank zu laden, benötigen wir lediglich die SqlMap Instanz, den Namen des Abbildungs-Statements und eine Person ID. Wir wollen Person Nummer 5 laden.

```
…
SqlMapClient sqlMap = MyAppSqlMapConfig.getSqlMapInstance(); // wie oben programmiert
…
Integer personPk = new Integer(5);
Person person = (Person) sqlMap.queryForObject ("getPerson", personPk);
…
```

Objekte in die Datenbank schreiben

Wir haben das Person Objekt aus der Datenbank geladen. Nun ändern wir einige Daten. Und zwar ändern wir die Höhe und das Gewicht der Person.

```
…
person.setHeightInMeters(1.83);          // person wurde oben ausgelesen
person.setWeightInKilograms(86.36);
…
sqlMap.update("updatePerson", person);
…
```

Wenn wir die Person löschen wollen, ist es genauso einfach.

```
…
sqlMap.delete ("deletePerson", person);
…
```

Und das Einfügen einer neuen Person gestaltet sich ähnlich.

```
Person newPerson = new Person();
newPerson.setId(11);        // normalerweise würde die ID eine fortlaufende Nummer sein
newPerson.setFirstName("Clinton");
newPerson.setLastName("Begin");
newPerson.setBirthDate (null);
newPerson.setHeightInMeters(1.83);
newPerson.setWeightInKilograms(86.36);
…
sqlMap.insert ("insertPerson", newPerson);
…
```

Das ist alles, was es zu sagen gibt!

Die nächsten Schritte…

Dies ist das Ende dieses kurzen Tutorials. Bitte besuchen Sie http://ibatis.apache.org für den kompletten SQL Maps 2.0 Developers Guide, sowie JPetStore 4, welches ein Beispiel einer kompletten funktionalen Web-Applikation ist, die auf Jakarta Struts, iBATIS DAO 2.0 und SQL Maps 2.0 basiert.
