iBATIS SQL Maps

Tutorial

For SQL Maps Version 2.0

February 18, 2006

Introduction

This brief tutorial will take you through a walkthrough of a typical use of SQL Maps. The details of each of the topics below can be found in the SQL Maps developer guide available from http://ibatis.apache.org

---

Preparing to Use SQL Maps

The SQL Maps framework is very tolerant of bad database models and even bad object models. Despite this, it is recommended that you use best practices when designing your database (proper normalization) and your object model. By doing so, you will get good performance and a clean design.

The easiest place to start is to analyze what you're working with. What are your business objects? What are your database tables? How do they relate to each other? For the first example, consider the following simple Person class that conforms to the typical JavaBeans pattern.

Person.java

```
package examples.domain;

//imports implied….

public class Person {
   private int id;
   private String firstName;
   private String lastName;
   private Date birthDate;
   private double weightInKilograms;
   private double heightInMeters;

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

   //…let's assume we have the other getters and setters to save space…
}
```

How does this Person class map to our database? SQL Maps doesn't restrict you from having relationships such as table-per-class or multiple-tables-per-class or multiple-classes-per-table. Because you have the full power of SQL available to you, there are very few restrictions. For this example, let's use the following simple table which would be suitable for a table-per-class relationship:

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)
)
```

The SQL Map Configuration File

Once we're comfortable with the classes and tables we're working with, the best place to start is the SQL Map configuration file. This file will act as the root configuration for our SQL Map implementation.

The configuration file is an XML file. Within it we will configure properties, JDBC DataSources and SQL Maps. It gives you a nice convenient location to centrally configure your DataSource which can be any number of different implementations. The framework can handle a number of DataSource implementations including iBATIS SimpleDataSource, Jakarta DBCP (Commons), and any DataSource that can be looked up via a JNDI context (e.g. from within an app server). These are described in more detail in the Developer's Guide. The structure is simple and for the example above, might look like this:

Example is continued on the next page…

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">

<!-- Always ensure to use the correct XML header as above! -->

<sqlMapConfig>

  <!-- The properties (name=value) in the file specified here can be used placeholders in this config
       file (e.g. "${driver}".  The file is usually relative to the classpath and is optional.  -->

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

  <!-- These settings control SqlMap configuration details, primarily to do with transaction
       management.  They are all optional (see the Developer Guide for more).  -->

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

  <!-- Type aliases allow you to use a shorter name for long fully qualified class names. -->

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

  <!-- Configure a datasource to use with this SQL Map using SimpleDataSource.
       Notice the use of the properties from the above resource -->

  <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>

  <!-- Identify all SQL Map XML files to be loaded by this SQL map.  Notice the paths
       are relative to the classpath.  For now, we only have one… -->

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

</sqlMapConfig>
```

SqlMapConfigExample.properties

```
# This is just a simple properties file that simplifies automated configuration
# of the SQL Maps configuration file (e.g. by Ant builds or continuous
# integration tools for different environments… etc.)
# These values can be used in any property value in the file above (e.g. "${driver}")
# Using a properties file such as this is completely optional.

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

The SQL Map File(s)

Now that we have a DataSource configured and our central configuration file is ready to go, we will need to provide the actual SQL Map file which contains our SQL code and the mappings for parameter objects and result objects (input and output respectively).

Continuing with our example above, let's build an SQL Map file for the Person class and the PERSON table. We'll start with the general structure of an SQL document, and a simple select statement:

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>
```

The above example shows the simplest form of SQL Map. It uses a feature of the SQL Maps framework that automatically maps the columns of a ResultSet to JavaBeans properties (or Map keys etc.) based on name matching. The #value# token is an input parameter. More specifically, the use of "value" implies that we are using a simple primitive wrapper type (e.g. Integer; but we're not limited to this).

Although very simple, there are some limitations of using the auto-result mapping approach. There is no way to specify the types of the output columns (if necessary) or to automatically load related data (complex properties), and there is also a slight performance implication in that this approach requires accessing the ResultSetMetaData. By using a resultMap, we can overcome all of these limitations. But, for now simplicity is our goal, and we can always change to a different approach later (without changing the Java source code).

Most database applications don't simply read from the database, they also have to modify data in the database. We've already seen an example of how a simple SELECT looks in a mapped statement, but what about INSERT, UPDATE and DELETE? The good news is that it's no different. Below we will complete our Person SQL Map with more statements to provide a complete set of statements for accessing and modifying data.

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">

    <!-- Use primitive wrapper type (e.g. Integer) as parameter and allow results to
         be auto-mapped results to Person object (JavaBean) properties -->
    <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>

    <!-- Use Person object (JavaBean) properties as parameters for insert.  Each of the
         parameters in the #hash# symbols is a JavaBeans property.  -->
    <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>

    <!-- Use Person object (JavaBean) properties as parameters for update. Each of the
         parameters in the #hash# symbols is a JavaBeans property.  -->
    <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>

    <!-- Use Person object (JavaBean) "id" properties as parameters for delete. Each of the
         parameters in the #hash# symbols is a JavaBeans property.  -->
    <delete id="deletePerson" parameterClass="examples.domain.Person">
        DELETE PERSON
        WHERE PER_ID = #id#
    </delete>

</sqlMap>
```

Programming with the SQL Map Framework

Now that we are all configured and mapped, all we need to do is code it in our Java application. The first step is to configure the SQL Map. This is very simply a matter of loading our SQL Map configuration XML file that we created before. To simplify loading the XML file, we can make use of the Resources class included with the framework.

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

The SqlMapClient object is a long-lived, thread safe service object. For a given run of an application, you only need to instantiate/configure it once. This makes it a good candidate for a static member of a base class (e.g. base DAO class), or if you prefer to have it more centrally configured and globally available, you could wrap it up in a convenience class of your own. Here is an example of a convenience class you could write:

```
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) {
         // If you get an error at this point, it doesn't matter what it was.  It is going to be
         // unrecoverable and we will want the app to blow up hard so we are aware of the
         // problem.  You should always log such errors and re-throw them in such a way that
         // you can be made immediately aware of the problem.
         e.printStackTrace();
         throw new RuntimeException ("Error initializing MyAppSqlConfig class.  Cause: " + e);
       }
   }

   public static SqlMapClient getSqlMapInstance () {
      return sqlMap;
   }

}
```

Reading Objects from the Database

Now that the SqlMap instance is initialized and easily accessible, we can make use of it. Let's first use it to get a Person object from the database. (For this example, let's assume there's 10 PERSON records in the database ranging from PER_ID 1 to 10).

To get a Person object from the database, we simply need the SqlMap instance, the name of the mapped statement and a Person ID. Let's load Person #5.

```
…
SqlMapClient sqlMap = MyAppSqlMapConfig.getSqlMapInstance(); // as coded above
…
Integer personPk = new Integer(5);
Person person = (Person) sqlMap.queryForObject ("getPerson", personPk);
…
```

Writing Objects to the Database

We now have a Person object from the database. Let's modify some data. We'll change the person's height and weight.

```
…
person.setHeightInMeters(1.83);          // person as read above
person.setWeightInKilograms(86.36);
…
sqlMap.update("updatePerson", person);
…
```

If we wanted to delete this Person, it's just as easy.

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

Inserting a new Person is similar.

```
Person newPerson = new Person();
newPerson.setId(11);       // you would normally get the ID from a sequence or custom table
newPerson.setFirstName("Clinton");
newPerson.setLastName("Begin");
newPerson.setBirthDate (null);
newPerson.setHeightInMeters(1.83);
newPerson.setWeightInKilograms(86.36);
…
sqlMap.insert ("insertPerson", newPerson);
…
```

That's all there is to it!

Next Steps…

This is the end of this brief tutorial. Please visit http://ibatis.apache.org for the complete SQL Maps 2.0 Developer Guide, as well as JPetStore 4, which is an example of a complete working web application based on Jakarta Struts, iBATIS DAO 2.0 and SQL Maps 2.0
