iBATIS SQL Maps

Tutoriel

Pour SQL Maps Version 2.0

15 Mai 2006

Traduction française de Julien Lafontaine (julienlafontaine2006@gmail.com)

Introduction

Ce tutoriel vous guidera à travers les différentes étapes d'une utilisation classique de SQL Maps. Vous trouverez plus d'informations concernant les sujets abordés ci dessous dans le Guide du développeur SQL Maps, disponible à l'adresse suivante : http://ibatis.apache.org

---

Préalables à l'utilisation de SQL Maps

Le framework SQL Maps est très tolérant vis à vis des modèles de base de données ou même des modèles objet mal conçus. Il est malgré tout recommandé de concevoir le modèle de sa base de données (normalisation) ainsi que son modèle objet dans les règles de l'art. Cela vous garantira de bonnes performances et une conception propre.

Commençons par nous pencher sur les éléments avec lesquels nous allons travailler. Quels sont les objets métiers? Quelles sont les tables de la base de données? Quelles relations existe-t-il entre eux? Pour notre premier exemple, considérons la classe Person obéissant à la convention JavaBean.

Person.java

```
package examples.domain;

//imports sous-entendus….

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;
   }
   //…les autres getters et setters ont été omis par manque de place…
}
```

Comment mapper cette classe sur notre base de données? SQL Maps n'impose aucune relation particulière entre classes java et tables de la base de données. Il est tout à fait possible d'avoir un mapping du type une table par classe, plusieurs tables par classe ou encore plusieurs classes par table. Puisque vous disposez de toute la puissance d'expression du SQL, il y a très peu de restrictions. Pour cet exemple, nous utiliserons la table suivante qui est tout à fait adaptée à un mapping du type une classe par table:

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)
)
```

Le fichier de configuration SQL Maps

Maintenant que nous nous sommes familiarisés avec les classes et les tables avec lesquelles nous allons travailler, intéressons nous au fichier de configuration SQL Maps. Ce fichier fait office de configuration de base pour notre implémentation de SQL Maps.

Le fichier de configuration est un fichier XML. Par l'intermédiaire de ce fichier nous allons configurer les propriétés internes du framework, les DataSources JDBC et le mapping SQL. Cela permet de centraliser la configuration des DataSources quelque soit leur nombre ou leur implémentation. Le framework gère différentes implémentations de DataSource : SimpleDataSource d'iBATIS , Jakarta DBCP (Commons), ainsi que tous les DataSources auxquels on peut accéder via un contexte JNDI (typiquement au sein d'un serveur d'application). Tout cela est décrit en plus détail dans le Guide du développeur. La structure de ce fichier est très simple, dans le cas de l'exemple ci-dessus le fichier de configuration pourrait ressembler à cela:

Exemple sur la page suivante…

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">

  <!-- Assurez vous de toujours utiliser l'en-tête XML ci dessus! -->

<sqlMapConfig>

  <!-- On peut faire référence aux propriétés (nom=valeur) du fichier ci-dessous depuis le fichier de
configuration (ex: “${driver}”.  Ce fichier est optionnel. Le chemin est relatif au classpath.  -->

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

  <!-- Ces paramètres permettent de configurer SqlMaps, ils concernent principalement la gestion
des transactions. Ils sont optionnels (reportez vous au guide du développeur pour plus
d'informations).  -->

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

  <!-- Les alias vous permettent de faire référence à une classe en utilisant un nom court à la place
du nom complet de la classe (nom + package). -->

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

  <!-- Configure un datasource utilisé par SQL Map, basé sur l'implémentation SimpleDataSource.
Notez l'utilisation des propriétés contenues dans le fichier de propriétés ci-dessus -->

  <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>

  <!-- Identifie tous les fichiers XML SQL Maps à charger. Notez que les chemins sont relatifs au
classpath. Pour le moment il n'y en a qu'un… -->

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

</sqlMapConfig>
```

SqlMapConfigExample.properties

```
# Ceci est un fichier de propriétés qui permet de simplifier la configuration automatique du fichier
# de configuration de SQL Maps (ex: via l'utilisation d'Ant, d'un outil d'intégration continue… etc.)
# Les valeurs suivantes peuvent être utilisées comme valeurs des propriétés dans le fichier ci-
# dessus (ex: "${driver}")
# L'utilisation d'un fichier de propriétés comme celui-ci est complètement optionnelle.

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

Le(s) fichier(s) SQL Maps

Maintenant que nous avons un DataSource et que notre fichier de configuration est prêt, il nous faut écrire le fichier SQL Maps lui même. Il contient le code SQL ainsi que les mappings pour les objets paramètre et résultat (entrée et sortie respectivement).

Continuons avec notre exemple, et construisons un fichier SQL Map pour la classe Person et la table PERSON. Nous commencerons par la structure générale d'un document SQL avec une simple requête de type SELECT:

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>
```

L'exemple ci-dessus est en fait le type de fichier SQL Map le plus simple qui soit. Il tire partie d'une des fonctionnalités du framework SQL Maps qui consiste à mapper les colonnes d'un ResultSet sur les propriétés d'un JavaBean (ou les entrées d'une Map etc.) lorsque elles ont le même nom. La chaîne #value# est un paramètre d'entrée. L'utilisation de “value” implique que l'on utilise un objet enveloppe associé à un type primitif (ex: Integer pour int, Long pour long, etc...).

Cette approche est très simple mais elle comporte des limitations. Il est par exemple impossible de spécifier le type des colonnes (si cela est nécessaire) ou de charger automatiquement des données associées (propriétés complexes). Cette approche a également un impact sur les performances dans la mesure où elle nécessite de manipuler l'objet ResultSetMetaData. L'utilisation d'un resultMap permet de s'affranchir de ces limitations. Mais pour le moment nous privilégierons la simplicité. Il est toujours possible de changer d'approche plus tard (sans avoir à changer le code java).

La plupart des applications utilisant une base de données ne se contentent pas de lire dans cette base de données, elles doivent aussi la modifier. Nous venons de voir à quoi ressemble le mapping d'une simple requête de type SELECT, mais qu'en est il des requêtes INSERT, UPDATE et DELETE? La bonne nouvelle c'est que ce n'est pas très différent. Ajoutons, dans le fichier SQL Map de la classe Person, les requêtes manquantes pour disposer d'un jeu complet de commandes nous permettant de charger et de modifier les données.

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">

    <!-- Utilisation d'un objet enveloppe associé au type primitif (ex: Integer pour int , Long pour long
         etc...) comme paramètre, et mapping automatique du résultat de la requête sur les
         propriétés de l'objet Person -->
    <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>

    <!-- Utilisation des propriétés de l'objet Person (JavaBean) comme paramètres de la requête.
          Chaque paramètre entouré par le caractère dièse (ex: #id#) est une propriété du Bean.-->
    <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>

    <!-- Utilisation des propriétés de l'objet Person (JavaBean) comme paramètres de la requête.
          Chaque paramètre entouré par le caractère dièse (ex: #id#) est une propriété du Bean.-->
    <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>

    <!-- Utilisation de la propriété “id” de l'objet Person (JavaBean) comme paramètre de la requête.
          Chaque paramètre entouré par le caractère dièse (ex: #id#) est une propriété du Bean.-->
    <delete id="deletePerson" parameterClass="examples.domain.Person">
          DELETE PERSON
          WHERE PER_ID = #id#
    </delete>

</sqlMap>
```

Programmation avec le Framework SQL Map

Maintenant que nous en avons fini avec la configuration et le mapping, il ne nous reste plus qu'à écrire le code Java. La première étape est la configuration de SQL Map. C'est très simple, il suffit de charger le fichier XML de configuration que nous venons de créer. On peut utiliser la classe Ressource fournie avec le framework afin de simplifier le chargement de ce fichier.

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

L'objet SqlMapClient est un objet service à durée de vie longue et sûr vis a vis des threads (*thread safe*). Pour chaque application s'exécutant vous n'avez besoin de l'instancier/configurer qu'une seule fois. Il peut donc être judicieux d'en faire un membre statique d'une classe de base (classe DAO de base par exemple), ou si vous préférer qu'il soit configurer de manière plus centrale et accessible plus globalement, vous pouvez le stocker dans une classe utilitaire que vous aurez développée vous même. Voici à quoi cette classe pourrait ressembler:

```
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) {
         // Si une erreur survient à cet endroit, quelle qu'en soit la raison elle ne sera pas récupérable.
         // Il faut donc que l'application signale clairement le problème.
         // Il est important de tracer ces exceptions et de les lever de nouveau afin de mettre
         // immédiatement le problème en évidence.
         e.printStackTrace();
         throw new RuntimeException(“Erreur durant l'initialisation de MyAppSqlConfig. Cause: ”+e);
       }
   }

   public static SqlMapClient getSqlMapInstance () {
      return sqlMap;
   }

}
```

Chargement d'objets depuis la base de données

Maintenant que l'instance de SqlMap est initialisée et facilement accessible, nous pouvons enfin l'utiliser. Commençons par charger un objet Person depuis la base de données. (Pour cet exemple supposons que nous ayons 10 enregistrements dans la table PERSON dont les identifiants PER_ID vont de 1 à 10).

Pour charger un objet Person depuis la base de données, nous avons seulement besoin d'une instance de SqlMap, du nom du *mapping* de la requête et de l'identifiant de l'enregistrement que nous voulons récupérer. Essayons de charger l'objet Person numéro 5.

```
…
SqlMapClient sqlMap = MyAppSqlMapConfig.getSqlMapInstance(); // définie ci-dessus
…
Integer personPk = new Integer(5);
Person person = (Person) sqlMap.queryForObject (“getPerson”, personPk);
…
```

Ecriture d'objets dans la base de données

Maintenant que nous disposons d'un objet Person chargé depuis la base de données. Essayons de modifier quelques informations. Nous allons changer la taille et le poids de cette Person.

```
…
person.setHeightInMeters(1.83);          // objet Person chargé plus haut
person.setWeightInKilograms(86.36);
…
sqlMap.update("updatePerson", person);
…
```

Il est tout aussi simple de supprimer une Person

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

Insérer un nouvelle Person ce fait de manière similaire

```
Person newPerson = new Person();
newPerson.setId(11);        // Il est généralement préférable d'obtenir un identifiant depuis une
                            // séquence ou un table spécifique
newPerson.setFirstName("Clinton");
newPerson.setLastName("Begin");
newPerson.setBirthDate (null);
newPerson.setHeightInMeters(1.83);
newPerson.setWeightInKilograms(86.36);
…
sqlMap.insert ("insertPerson", newPerson);
…
```

Et voilà ce n'est pas plus compliqué que ça!

Etapes suivantes…

C'est la fin de ce petit tutoriel. Le guide du développeur SQL Maps 2.0, ainsi que JPetStore 4 (un exemple complet d'application basée sur Jakarta Struts, iBATIS DAO 2.0 et SQL Maps 2.0) sont disponibles à l'adresse suivante http://ibatis.apache.org.