Accessing databases via JDBC

Using a relational database via JDBC has a major disadvantage: For obvious reasons, you are limited to a flat XML structure. In other words, you may choose an XML element, and map its atomic child elements and/or attributes to the columns of a table. Of course, you are free to extend the XML structure with additional attributes or child elements, even complex child elements. However, these cannot be saved or read by the generated persistence managers (PM's). To overcome this problem, you might like to use manually written subclasses of the generated PM's.

1. Creating a schema

The first step when working with JDBC is to read the database schema. The database details must be specified in the schema. As an example, we quote the file session.xsd from the JaxMe distribution:

```
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://ws.apache.org/jaxme/test/pm/session"
    xmlns:jm="http://ws.apache.org/jaxme/namespaces/jaxme2/schema"
    xmlns:jdm="http://ws.apache.org/jaxme/namespaces/jaxme2/jdbc-mapping"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
  <xs:element name="Session">
    <xs:complexType>
      <xs:annotation><xs:appinfo>
        <jdm:table name="httpSession"/>
      </xs:appinfo></xs:annotation>
      <xs:sequence>
        <xs:element name="IpAddress" type="xs:string">
          <xs:annotation>
            <xs:documentation>
              A default for the spelling of "IpAddress". Without this default,
              the spelling would be choosen as provided by JDBC, typically
              uppercased.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="Cookie">
        <xs:annotation>
          <xs:documentation>
            Just to demonstrate, that a database column can also
```

```
            be mapped to an attribute.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The most important thing to note is the element jdm:table below xs:schema/xs:element/xs:complexType/xs:annotation/xs:appinfo: It contains the name of a table. The JDBC generator takes this as an advice to do the following:

1. Connect to the database via JDBC and read the tables columns via JDBC metadata.
2. For any column in the table: If there already exists an atomic child element or attribute with the same name, create a mapping between this column and the element or attribute. Note, that matching of the column name is case insensitive.
3. For any column in the table: A new attribute is created, if there was no matching child element or attribute.

Note:

Column names are mostly in upper case letters when returned by JDBC. In other words: If your column is called "NAME", then your generated methods will be called getNAME() and setNAME(), which is typically not what you want. To get rid of the uppercase letters, either create an explicit attribute called "name" or use a syntax like CREATE TABLE "Address" ("name" VARCHAR(20) ...)

2. Running the JDBC generator

As usual, the schema reader must be used to read your schema and create sources. However, in this case the invocation looks slightly different:

```
<xjc target="${build.src}">
  <schema dir="src/test/pm" includes="*.xsd"/>
  <sgFactoryChain className="org.apache.ws.jaxme.pm.generator.jdbc.JaxMeJdbcSG"/>
  <schemaReader className="org.apache.ws.jaxme.generator.sg.impl.JaxMeSchemaReader"
  <produces dir="${build.src}" includes="org/apache/ws/jaxme/test/pm/session/*"/>
  <property name="jdbc.driver" value="${jdbc.driver}"/>
  <property name="jdbc.url" value="${jdbc.url}"/>
  <property name="jdbc.user" value="${jdbc.user}"/>
  <property name="jdbc.password" value="${jdbc.password}"/>
</xjc>
```

The differences to the usual schema reader invocation are:

- A so-called SGFactoryChain is used to customize the schema binding. In this particular case it is an instance of JaxMeJdbcSG. This object will connect to the database and populate the original schema with columns read from the database. It will also create specific persistence managers, one for any table.

- Some parameters are required to connect to the database. In our case they are specified as properties. The required parameters are:
  jdbc.driver
  The JDBC driver class, for example org.hsqldb.jdbcDriver
  jdbc.url
  The JDBC URL, for example jdbc:hsqldb:directoryname/filename
  jdbc.user
  The database user name, for example sa
  jdbc.password
  The users password
  Driver and JDBC URL are required, user name and password are optional.

If you do not like to specify the connection details on the command line, you may instead choose to specify them as a part of the jdm:table element, like this:

```
<jdm:table name="httpSession" driver="org.hsqldb.jdbcDriver"
           url="jdbc:hsqldb:directoryname/filename" user="sa"
           password=""/>
```

This approach has the obvious advantage, that you may specify different parameters for any table, combining tables from various databases. The obvious disadvantage is, that you must specify parameters for any table. :-)

To omit the disadvantage, you have a third option: Specifying the database details as a part of the schema bindings, like this:

```
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://ws.apache.org/jaxme/test/pm/session"
    xmlns:jm="http://ws.apache.org/jaxme/namespaces/jaxme2/schema"
    xmlns:jdm="http://ws.apache.org/jaxme/namespaces/jaxme2/jdbc-mapping"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
  <xs:annotation><xs:appinfo>
    <jdm:connection driver="org.hsqldb.jdbcDriver"
           url="jdbc:hsqldb:directoryname/filename" user="sa"
           password=""/>
  </xs:appinfo></xs:annotation>
  ...
</xs:schema>
```

Note:

The order of precedence is as follows: Parameters specified on the command line will always win. If a parameter is not specified on the command line, then it will be read from a jdm:table element. If such an element is missing or the respective attribute is missing, then the jdm:connection element will be consulted.

3. Inserting objects into the database

To insert an object into the database, you might like to use the following code:

```
            import javax.xml.bind.JAXBContext;
            import org.apache.ws.jaxme.PM;
            import org.apache.ws.jaxme.impl.JAXBContextImpl;
            import org.apache.ws.jaxme.test.pm.session.ObjectFactory;
            import org.apache.ws.jaxme.test.pm.session.Session;

          Session session = new ObjectFactory().createSession();
          session.setIpAddress("127.0.0.1");
          session.setCookie("56ghs2398723sjkjl");
          JAXBContextImpl factory = (JAXBContextImpl)
            JAXBContext.newInstance("org.apache.ws.jaxme.test.pm.session");
          PM pm = factory.getJMPM();
          pm.insert(session);
```

It really is that simple!

The question arises: What magic tells the persistence manager to what database it should be connect? This is specified in the file Configuration.xml, which was also generated by the JDBC generator in the directory org/apache/ws/jaxme/test/pm/session:

```
          <Configuration xmlns="http://ws.apache.org/jaxme/namespaces/jaxme2/configuratio
            <Manager validatorClass="org.apache.ws.jaxme.test.pm.session.impl.SessionType
                     qName="{http://ws.apache.org/jaxme/test/pm/session}Session"
                     pmClass="org.apache.ws.jaxme.test.pm.session.impl.SessionTypePM"
                     marshallerClass="org.apache.ws.jaxme.test.pm.session.impl.SessionTyp
                     handlerClass="org.apache.ws.jaxme.test.pm.session.impl.SessionHandle
                     elementInterface="org.apache.ws.jaxme.test.pm.session.Session"
                     elementClass="org.apache.ws.jaxme.test.pm.session.impl.SessionImpl">
              <Property value="org.hsqldb.jdbcDriver" name="jdbc.driver" />
              <Property value="jdbc:hsqldb:directoryname/filename" name="jdbc.url" />
              <Property value="sa" name="jdbc.user" />
              <Property value="" name="jdbc.password" />
            </Manager>
          </Configuration>
```

Note:

If you have different databases for production and development, then you have to edit the generated file Configuration.xml. The recommended way to do so is using the Ant task "filter".

Note:

As of this writing, the insert method cannot generate a primary key for you. The recommended way to do so is via identity or autoincrement columns. However, there is still no way to return the primary key. We'll look into that as soon as possible.

4. Reading objects from the database

Reading objects from the database is also straightforward. Assume for now, that we want to fetch the HTTP session with Cookie "56ghs2398723sjkjl". Here's how we do that:

```
            import javax.xml.bind.JAXBContext;
            import org.apache.ws.jaxme.PM;
            import org.apache.ws.jaxme.PMParams;
            import org.apache.ws.jaxme.impl.JAXBContextImpl;
            import org.apache.ws.jaxme.test.pm.session.Session;

                JAXBContextImpl factory = (JAXBContextImpl)
          JAXBContext.newInstance("org.apache.ws.jaxme.test.pm.session");
        PM pm = factory.getJMPM();
        String whereClause = "COOKIE = ?";
        PMParams params = new PMParams();
        params.addParam("56ghs2398723sjkjl");
        Iterator iter = pm.select(whereClause, params);
        if (!iter.hasNext()) {
          throw new NullPointerException("No matching cookie found");
        }
        Session session = (Session) iter.next();
```

5. Deleting objects from the database

Deleting an object requires the objects primary key. Assuming that the cookie attribute is an HTTP sessions primary key, one might do the following:

```
            import javax.xml.bind.JAXBContext;
            import org.apache.ws.jaxme.PM;
            import org.apache.ws.jaxme.PMParams;
            import org.apache.ws.jaxme.impl.JAXBContextImpl;
            import org.apache.ws.jaxme.test.pm.session.Session;

        Session session = new ObjectFactory().createSession();
        session.setCookie("56ghs2398723sjkjl");
                JAXBContextImpl factory = (JAXBContextImpl)
          JAXBContext.newInstance("org.apache.ws.jaxme.test.pm.session");
        PM pm = factory.getJMPM();
        pm.delete(session);
```

6. Updating objects in the database

Updating an object works much like deleting it. In particular it requires the presence of a primary key. In other words, the following example would change an HTTP sessions IP address:

```
    import javax.xml.bind.JAXBContext;
    import org.apache.ws.jaxme.PM;
    import org.apache.ws.jaxme.PMParams;
    import org.apache.ws.jaxme.impl.JAXBContextImpl;
    import org.apache.ws.jaxme.test.pm.session.Session;

        JAXBContextImpl factory = (JAXBContextImpl)
  JAXBContext.newInstance("org.apache.ws.jaxme.test.pm.session");
PM pm = factory.getJMPM();
String whereClause = "COOKIE = ?";
PMParams params = new PMParams();
params.addParam("56ghs2398723sjkjl");
Iterator iter = pm.select(whereClause, params);
if (!iter.hasNext()) {
  throw new NullPointerException("No matching cookie found");
}
Session session = (Session) iter.next();
session.setIpAddress("192.168.1.1");
pm.update(session);
```