Accessing a Tamino database

Tamino is a native XML database by Software AG. Compared to a relational database, it has the disadvantage of being not very popular. However, if your data is structured (more structured than conveniently expressible by relational data structures, that is) you will soon find a lot of advantages and possibly prefer it over a traditional SQL database engine. From within JaxMe, Tamino may be accessed in either of three ways:

- Via the XML:DB API, implemented by the XmlDbPM. This is the recommended way if you want your application to be portable amongst various XML databases. The XML:DB API is being described in a separate document. (To be done.)
- Via native HTTP, implemented by the InoManager. This solution is recommended, if you need a very low memory profile, even for processing a large result set. In particular it offers a true streaming mode.
- Via the official Tamino Java API, called TaminoAPI4J. This is recommended for enterprise applications, as it allows to embed Tamino into the transactional context of an EJB container.

1. Preparations

Tamino is accessible via two different query languages. The elder variant is called *X-Query* and is best compared with *XPath*. The newer language is based on *XQuery*.

XPath and X-Query share an important problem when using namespaces: They have no mapping between namespace prefixes and namespace URIs. In other words, if you perform a query like

```
_xql=ad:Address
```

then the database *must know*, that the prefix `ad` is mapped to the namespace URI `http://ws.apache.org/jaxme/test/misc/address`.

Tamino and the JaxMe managers overcome the absence of a mapping in the query by storing the mapping from the schema and using that. In other words, if you are using namespaces, then you should:

1. Specify a prefix for the namespace in the schema.
2. Use the same schema (and thus the same prefix) for the database schema as well as the JaxMe generator. (Obviously this is recommended anyways.)

3. Use the same prefix for specifying queries.

2. Preparing the Tamino Resource Adapter

If you are not using the Tamino Resource Adapter (which is typically the case, if you are *not* using an EJB container like JBoss, WebSphere, or the like. In that case you may very well skip this section and proceed to the next section, which is about creating the schema file.

Adding the Tamino Resource Adapter is covered in the documentation of the TaminoAPI4J. However, we'll provide specific details for JBoss 3.2 here, because the docs are for JBoss 3.0 only and because we disagree with the recommendation to add the Tamino jar files to the JBoss lib directory. So here's what we've done, step by step:

- Add the jar file `TaminoAPI4J.jar` to the rar file, for example `TaminoJCA_localTx.rar`. Copy the RAR file to the JBoss deploy directory.
- Create a deployment descriptor file `tamino-service.xml` with the following contents and copy it to the JBoss deploy directory. (Of course you should adapt the database URL, user and password to your local settings. Most probably you would also want to change the JNDI name `MyTaminoLocalTxConnector`.

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <!-- ===================================================================== -->
  <!-- New ConnectionManager setup for Tamino -->
  <!-- ===================================================================== -->
  <mbean code="org.jboss.resource.connectionmanager.LocalTxConnectionManager"
         name="jboss.jca:service=LocalTxCM,name=MyTaminoLocalTxConnector">
    <attribute name="JndiName">MyTaminoLocalTxConnector</attribute>

    <depends optional-attribute-name="ManagedConnectionPool">
      <!--embedded mbean-->
      <mbean code="org.jboss.resource.connectionmanager.JBossManagedConnectionPool"
             name="jboss.jca:service=LocalTxPool,name=MyTaminoLocalTxConnector">
            <attribute name="MinSize">0</attribute>
        <attribute name="MaxSize">50</attribute>
            <attribute name="BlockingTimeoutMillis">5000</attribute>
            <attribute name="IdleTimeoutMinutes">15</attribute>
            <!-- criteria indicates if Subject (from security domain) or app supplied
                     parameters (such as from getConnection(user, pw)) are used to dist
                     connections in the pool. Choices are
                     ByContainerAndApplication (use both),
                     ByContainer (use Subject),
                     ByApplication (use app supplied params only),
                     ByNothing (all connections are equivalent, usually if adapter supp
                     reauthentication)-->
            <attribute name="Criteria">ByContainer</attribute>

            <depends optional-attribute-name="ManagedConnectionFactoryName">
              <!--embedded mbean-->
                  <mbean code="org.jboss.resource.connectionmanager.RARDeployment"
```

```
                              name="jboss.jca:service=LocalTxDS,name=MyTaminoLocalTxConn
                    <attribute name="ManagedConnectionFactoryProperties">
                      <properties>
                            <config-property name="TaminoURL" type="java.lang.String">h
                            <config-property name="UserName" type="java.lang.String">ej
                            <config-property name="Password" type="java.lang.String">xx
                      </properties>
                    </attribute>

                    <!--Below here are advanced properties -->
                    <depends optional-attribute-name="OldRarDeployment">jboss.jca:servi
                  </mbean>
            </depends>
          </mbean>
    </depends>
    <depends optional-attribute-name="CachedConnectionManager">jboss.jca:service=Cached
    <depends optional-attribute-name="JaasSecurityManagerService">jboss.security:servic
    <depends optional-attribute-name="TransactionManagerService">jboss:service=Transact

    <!--make the rar deploy! hack till better deployment-->
    <depends>jboss.jca:service=RARDeployer</depends>
  </mbean>
</server>
```

- The Tamino driver is internally using the method `XMLReaderFactory.createXMLReader()`. Unfortunately this method is using `Class.forName(String)` internally to load the SAX driver class. This won't work in an environment with complex class loaders. In order to make sure, that the latest SAX version is used, I did the following:
  1. Downloaded the jar file `sax2r2.jar` and extracted the jar file `sax.jar` from it.
  2. Moved that file to the directory `jre/lib/endorsed` in my Java SDK directory.

3. An example schema for TaminoAPI4J

As an example, we'll reuse the schema from the marshaller examples, `Address.xsd`:

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

<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:ad="http://ws.apache.org/jaxme/test/misc/address"
    xmlns:inoapi="http://ws.apache.org/jaxme/namespaces/jaxme2/TaminoAPI4J"
    xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition"
    jaxb:extensionBindingPrefixes="xjc inoapi"
    xml:lang="EN"
    targetNamespace="http://ws.apache.org/jaxme/test/misc/address"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
  <xs:annotation>
```

```
    <xs:documentation>
      A simple JaxMe example: Personal address collection.
    </xs:documentation>
    <xs:appinfo>
      <jaxb:globalBindings>
        <xjc:serializable/>
        <inoapi:raDetails collection="adr" jndiReference="java:MyTaminoLocalTxConnector
        <!-- If you are not using the Tamino Resource Adapter, then the following
             will fit for you:
          <inoapi:dbDetails collection="adr" url="http://127.0.0.1/tamino/adrDb"
            user="me" password="MySecretPassword"/>
        -->
      </jaxb:globalBindings>
      <tsd:schemaInfo name="Address">
        <tsd:collection name="adr"></tsd:collection>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>

  <xs:element name="Address">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name">
                  ...
                </xs:element>
          </xs:sequence>
        </xs:complexType>
  </xs:element>
</xs:schema>
```

Ignoring the details of the actual `Address` type, we only note the differences in the schema header:

- A namespace prefix `ad` is specified for the target namespace. In the previous section we have discussed, that this is a precondition.
- The element `xjc:serializable` element requests, that the generated classes implement the interface `java.io.Serializable`. This is required for use within an EJB container, as the objects wouldn't be serializable otherwise. The `xjc:serializable` is a vendor extension from the JAXB RI and supported by JaxMe too.
- The element `tsd:schemaInfo` fixes the schema and collection name. This element is read by Tamino when creating the schema. It is ignored by JaxMe.
- The element `inoapi:raDetails` specifies the same collection name and a JNDI name. The latter name is used to lookup the Tamino resource adapter.
- The JAXB specification requires, that the element `jaxb:globalBindings` contains no elements from other namespace than `jaxb`. To add vendor extensions like `xjc:serializable` and `inoapi:raDetails`, we need to add the attribute `jaxb:extensionBindingPrefixes="jaxb inoapi"` to `xs:schema`.

4. Build your own JaxMe distribution

For licensing reasons, we cannot add the files `TaminoAPI4J.jar` and `TaminoJCA.jar` to the JaxMe SVN repository. In particular we cannot offer compiled classes based on these files in the JaxMe distribution. Unfortunately that means, that you have to build your own distribution. Fortunately, this is quite simple:

- Download the JaxMe source distribution (to be distinguished from the JaxMe binary distributon) and extract it.
- Download the TaminoAPI4J distribution and install it. Copy the files `TaminoAPI4J.jar` and `TaminoJCA.jar` to the subdirectory `prerequisites` in the JaxMe directory.
- Change to the JaxMe directory and run `ant`. The build script will automatically detect the presence of the Tamino API files.

> **Note:**
> As of this writing, there is no official JaxMe distribution available, which includes the Tamino support. In other words, rather than downloading the sources you have to extract them from the JaxMe SVN repository.

5. Creating an Ant task

To invoke the JaxMe generator, use an Ant task like the following:

```
  <target name="generate">
<taskdef name="xjc" classname="org.apache.ws.jaxme.generator.XJCTask">
  <classpath>
            <fileset dir="lib" includes="jaxme*.jar"/>
            <fileset dir="lib" includes="log4j-1.2.8.jar"/>
  </classpath>
</taskdef>
    <mkdir dir="${build.src}"/>
<xjc target="${build.src}">
  <schema dir="${etc}" includes="*.xsd"/>
  <produces dir="${build.src}" includes="org/apache/ws/jaxme/test/misc/address/**
  <sgFactoryChain className="org.apache.ws.jaxme.generator.ino.api4j.TaminoAPI4JS
  <schemaReader className="org.apache.ws.jaxme.generator.sg.impl.JaxMeSchemaReade
</xjc>
  </target>
```

6. Using the native HTTP API

Not yet documented. (To be done.)