Reladomo Tutorial

July 11, 2006

Table of Contents

1. How Do I Start a Reladomo Project? 1
2. Create The Reladomo Object XML Files 1
3. Create the Reladomo class list 4
4. Create the Reladomo Connection Manager 5
5. Create the Reladomo Runtime Configuration file 6
6. Initialization and startup 7
7. Transaction Management 9
8. Build your project 12

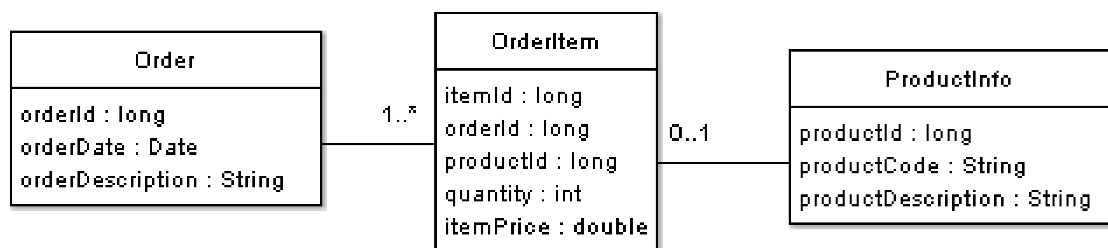
1. How Do I Start a Reladomo Project?

This tutorial demonstrates the necessary steps to get your Reladomo project started. Since this example is for illustration purposes, we will keep it as simple as possible. Other examples will be provided to illustrate more advanced features of Reladomo.

Let's assume that we want to create an application that can manipulate orders. An order contains information about its creation date, the user which created the order, and a description of the order. An order has a collection of items. An item has price information, quantity and the product specific information (product code, and product description).

After the analysis process we came up with the domain model shown inFigure 1, “Domain Model”:

Figure 1. Domain Model



For this example, our database will contain three tables: ORDERS, ORDER_ITEM, and PROD_INFO.

The first thing we need to do is create the Reladomo Object XML files that will map the Reladomo objects with the corresponding database table.

2. Create The Reladomo Object XML Files

Reladomo maps database tables with Java objects. These Java objects are generated by Reladomo using a set of configuration and Reladomo Object XML (mapping) files provided by the user.

It is required to create one XML file for every Reladomo object. The Reladomo Object XML file is the metadata definition of the Reladomo object. The Reladomo Object XML defines:

- **objectType** - Specifies if the object is read-only or transactional.
- **packageName** - The package to which the generated Reladomo classes belong to.
- **className** - The name of the Reladomo object class. This class name will be used as the base name for the generated classes.
- **tableName** - Name of the database table to which the Reladomo object is mapped to.
- List of object attributes. For each attribute the meta-data will define:
  - **name** - Name of the attribute.
  - **javaType** - The Java type of the attribute. Using the Java type specified and the attribute name, the generator will create the attribute's getters and setters. Valid JavaTypes are primitive types, String, Date, Timestamp, and byte[] (for blobs).
  - **columnName** - The name of the column in the mapped database table.
  - **primaryKey** - Indicates that the attribute is part of the primary key. Every Reladomo object must contains at least one primaryKey attribute.
  - primaryKeyGeneratorStrategy - Valid only when primaryKey is set to "true". It indicates the strategy used for automatic primary key generation. Valid values are: Max or SimulatedSequence. Max strategy is more suitable for legacy systems.
  - **nullable**- True if the column is nullable.
  - **readOnly** - Indicates that this attribute is read-only (cannot be modified).
- Relationships to another classes - Relationships link objects of various types

Refer to the xml schema definition (mithraobject.xsd) for a detailed explanation of the components of the Reladomo Object XML file.

For this example we need to create three Reladomo objects: Order, OrderItem, and ProductInfo.

Order.xml:

```
<MithraObject objectType="transactional">
    <PackageName>com.gs.fw.common.mithra.test.domain</PackageName>
    <ClassName>Order</ClassName>
    <DefaultTable>ORDERS</DefaultTable>

    <Attribute name="orderId" javaType="long" columnName="ORDER_ID"
  primaryKey="true"/>
    <Attribute name="orderDate" javaType="Timestamp"
  columnName="ORDER_DATE"/>
    <Attribute name="userId" javaType="int" columnName="USER_ID"/>

  <Attribute name="description" javaType="String" columnName="DESCRIPTION" maxLeng
>

  <Relationship name="items" relatedObject="OrderItem" cardinality="one-
to-many" >
```

```
        OrderItem.orderId = this.orderId
    </Relationship>
</MithraObject>
```

OrderItem.xml:

```
<MithraObject objectType="transactional">

    <PackageName>com.gs.fw.common.mithra.test.domain</PackageName>
    <ClassName>OrderItem</ClassName>
    <DefaultTable>ORDER_ITEM</DefaultTable>


  <Attribute name="itemId" javaType="long" columnName="ITEM_ID" primaryKey="true"/
>

  <Attribute name="orderId" javaType="long" columnName="ORDER_ID" primaryKey="true
>
    <Attribute name="productId" javaType="long" columnName="PROD_ID"/>
    <Attribute name="quantity" javaType="int" columnName="QTY"/>

  <Attribute name="itemPrice" javaType="double" columnName="ITEM_PRICE"/
>


  <Relationship name="productInfo" relatedObject="ProductInfo" cardinality="one-
to-one" >
        ProductInfo.productId = this.productId
    </Relationship>
</MithraObject>
```

ProductInfo.xml:

```
<MithraObject objectType="transactional">
    <PackageName>com.gs.fw.common.mithra.test.domain</PackageName>
    <ClassName>ProductInfo</ClassName>
    <DefaultTable>PROD_INFO</DefaultTable>


  <Attribute name="productId" javaType="long" columnName="PROD_ID" primaryKey="tru
>

  <Attribute name="productCode" javaType="String" columnName="PROD_CODE"/
>

  <Attribute name="productDescription" javaType="double" columnName="PROD_DESC"/
>
</MithraObject>
```

Reladomo Relationships

The relationship tag defines how the Reladomo object is related to another Reladomo object. In our example, the Order object has a one-to-many relationship with OrderItem. In the database, these two tables are related by the ORDER_ID column. A relationship between Order and OrderItem can be defined like this:

```
<Relationship name="items" relatedObject="OrderItem" cardinality="one-
to-many">
    OrderItem.orderId = this.orderId
</Relationship>
```

- **name** - This name is used by Reladomo to generate convenience methods to retrieve related objects.
- **relatedObject** - This attribute specifies the related object's name
- **cardinality** - This attribute specifies the type of relationship between the objects. It could be on of "one-to-one", "one-to-many", "many-to-many"
- **relationship expression** - An expression that indicates a set of rules that an one object needs to satisfy to become related to another object.

This relationship definition instructs the Reladomo code generator to create a relationship attribute in the Finder object (Finders are generated classes used to access Reladomo object attributes and to create Operation objects, more on Finders, Attributes, and Operations later). In addition, a method to access the related objects is generated in the Reladomo object. For example, in this case a getItems() method is generated. This method will return all the OrderItem objects for an Order based on the relationship expression (OrderItem.orderId = this.orderId).

Sometimes is necessary to traverse the relationship in the opposite direction, for example, it will be useful to know the details of an Order based on an OrderItem. You can add a relationship tag in the OrderItem.xml that defines the relationship from OrderItem to Order. This will cause the generation of similar methods and relationship attributes in the OrderItem generated classes. Another way to do this is to use the reverseRelationshipName attribute in the Relationship tag definition.

```
<Relationship name="items" relatedObject="OrderItem" cardinality="one-
to-many" reverseRelationshipName="order">
      OrderItem.orderId = this.orderId
</Relationship>
```

By using the reverseRelationship attribute, the Relationship needs to be defined only once (in this case in Order.xml). Reladomo will take care of generating the necessary relationship attributes and methods on both related set of classes.

Will be discussing in details how relation works and how to build more complex relationships in subsequent documents.

Once the Reladomo Object XML files are created, we need to create the Reladomo class list.

3. Create the Reladomo class list

The Reladomo class list is an xml file used during the generation of the Reladomo classes. The file lists the name of the Reladomo objects that need to be generated.

MithraClassList.xml:

```
<Mithra>
    <MithraObjectResource name="Order"/>
    <MithraObjectResource name="OrderItem"/>
    <MithraObjectResource name="ProductInfo"/>
```

```
</Mithra>
```

4. Create the Reladomo Connection Manager

The Reladomo Connection Manager is an object that Reladomo will use during runtime to obtain a database connection.

MithraTestAppConnectionManager.java:

```
import com.gs.fw.common.mithra.connectionmanager.XAConnectionManager;
import
 com.gs.fw.common.mithra.connectionmanager.SourcelessConnectionManager;
import com.gs.fw.common.mithra.databasetype.DatabaseType;
import com.gs.fw.common.mithra.databasetype.SybaseDatabaseType;
import java.util.TimeZone;
import java.sql.Connection;

public class MyProjConnectionManager implements
 SourcelessConnectionManager
{
    protected static MyProjConnectionManager instance;
    protected static final String MAX_POOL_SIZE_KEY = "maxPoolSize";
    protected final int DEFAULT_MAX_WAIT = 500;
    protected static final int DEFAULT_POOL_SIZE = 10;
    private static final TimeZone NEW_YORK_TIMEZONE =
 TimeZone.getTimeZone("America/New_York");
    private final String driverClassname = "...";
    private final String serverName = "MYSERVER";
    private final String resourceName = "MYPROJ";

    private XAConnectionManager xaConnectionManager;

    public static synchronized MyProjConnectionManager getInstance()
    {
        if (instance == null)
        {
            instance = new MyProjConnectionManager();
        }
        return instance;
    }
    protected MyProjConnectionManager()
    {
        this.createConnectionManager();
    }

    private XAConnectionManager createConnectionManager()
    {
        xaConnectionManager = new XAConnectionManager();
        xaConnectionManager.setDriverClassName(driverClassname);
        xaConnectionManager.setMaxWait(DEFAULT_MAX_WAIT);
        xaConnectionManager.setLdapName(serverName);
        xaConnectionManager.setDefaultSchemaName(resourceName);
        xaConnectionManager.setJdbcUser("sa");
        xaConnectionManager.setJdbcPassword("");
```

```
        xaConnectionManager.setPoolName("myproj connection pool");
        xaConnectionManager.setInitialSize(1);
        xaConnectionManager.setPoolSize(DEFAULT_POOL_SIZE);
        xaConnectionManager.setUseStatementPooling(true); // Only good
 for DB2
        xaConnectionManager.initialisePool();
        return xaConnectionManager;
    }

    public Connection getConnection()
    {
        return xaConnectionManager.getConnection();
    }

    public DatabaseType getDatabaseType()
    {
        return SybaseDatabaseType.getInstance();
    }

    public TimeZone getDatabaseTimeZone()
    {
        return NEW_YORK_TIMEZONE;
    }

    public BulkLoader createBulkLoader() throws BulkLoaderException
    {
        return this.getDatabaseType().createBulkLoader(
                    "sa",
                    "",
                    this.xaConnectionManager.getHostName(),
                    this.xaConnectionManager.getPort());
    }

    public String getDatabaseIdentifier()
    {
        return
 xaConnectionManager.getServerName()+":"+xaConnectionManager.getResourceName();
    }
}
```

5. Create the Reladomo Runtime Configuration file

The Reladomo configuration file is an xml file used during the initialization of the Reladomo service. The Reladomo configuration xml contains lists of Reladomo Connection Manager and its associated Reladomo object configuration. The Reladomo object configuration specifies the fully qualified Reladomo object class and the caching strategy that Reladomo will use for that specific object.

Reladomo supports two types of caching strategies:

- **partial** - This strategy makes Reladomo holds references that will be garbage collected if there is a memory crunch.

- **full** - This caching mode makes Reladomo load all the objects from database and keep them permanently in cache. All queries for these objects are returned from cache and all updates are write-through.

MithraRuntimeConfig.xml:

```
<MithraRuntime>

  <ConnectionManager className="com.gs.fw.common.MithraTestAppConnectionManager
 ">

  <MithraObjectConfiguration className="com.gs.fw.myProject.mithra.Order" cacheTyp
>

  <MithraObjectConfiguration className="com.gs.fw.myProject.mithra.OrderItem
 " cacheType="partial"/>

  <MithraObjectConfiguration className="com.gs.fw.myProject.mithra.ProductInfo
 " cacheType="partial"/>
    </ConnectionManager>
</MithraRuntime>
```

6. Initialization and startup

The runtime initialization process includes setting the transaction timeout and loading the MithraRuntimeConfig xml file(s).

MithraTestApp.java:

```
public class MithraTestApp
{
    private Logger logger =
 LoggerFactory.createLogger(MithraTestApp.class.getName());

    private static int maxTransactionTimeout = 120;

    public Logger getLogger()
    {
        return logger;
    }

    public void setLogger(Logger logger)
    {
        this.logger = logger;
    }

    public static int getMaxTransactionTimeout()
    {
        return maxTransactionTimeout;
    }

    public static void setMaxTransactionTimeout(int
 maxTransactionTimeout)
    {
```

```
        MithraTestApp.maxTransactionTimeout = maxTransactionTimeout;
    }

    public static void main(String[] args)
    {
        MithraTestApp app = new MithraTestApp();
        try
        {
            app.initialiseMithra();

app.loadMithraConfigurationXml("MithraRuntimeConfig1.xml");
            // optionally load multiple config files:

app.loadMithraConfigurationXml("MithraRuntimeConfig2.xml");

            //Do your thing!!

        }
        catch(Exception e)
        {
            System.out.println(e);
        }

    }

    private void loadMithraConfigurationXml(String mithraXml)
            throws Exception
    {
        logger.info("Mithra configuration XML is " + mithraXml);
        InputStream is =
MithraTestApp.class.getClassLoader().getResourceAsStream(mithraXml);
        if (is == null) throw new Exception("can't find file: " +
mithraXml + " in classpath");

MithraManagerProvider.getMithraManager().readConfiguration(is);
        try
        {
            is.close();
        }
        catch (java.io.IOException e)
        {
            logger.error("Unable to initialise Reladomo!", e);
            throw new Exception("Unable to initialise Reladomo!", e);
        }
        logger.info("Reladomo configuration XML " + mithraXml+" is now
loaded.");
    }

    private void initialiseMithra() throws Exception
    {
        try
        {
            logger.info("Transaction Timeout is "
+ this.getMaxTransactionTimeout());
```

```
            MithraManager mithraManager =
 MithraManagerProvider.getMithraManager();

 mithraManager.setTransactionTimeout(this.getMaxTransactionTimeout());
        }
        catch (Exception e)
        {
            logger.error("Unable to initialise Reladomo!", e);
            throw new Exception("Unable to initialise Reladomo!", e);
        }
        logger.info("Reladomo has been initialised!");
    }
}
```

7. Transaction Management

Objects retrieved via a finder operations are "live" objects meaning that any calls to setter methods are immediately reflected in the database. For efficiency, a series of setters should be called without each call resulting in a trip to the database. Also, usually the requirement is for a set of changes to be applied to one or more objects atomically so that they are committed together or rollback together. For both of these scenarios, Reladomo transactional commands are used.

There is a TransactionalCommand interface for which developers provide an implementation class. The key method of the interface is:

```
public Object executeTransaction(MithraTransaction tx) throws
 Throwable
```

Implementers provide a class which implements this method. Inside this method, the various setters, insert, and/or delete methods of Reladomo objects are invoked. The executeTransaction method forms the transactional boundary which begins when the method is entered and commits when the method returns. The changes made to the various Reladomo objects modified inside the executeTransaction method are committed atomically in a single transaction. To rollback all the changes at any point, an exception is thrown from this method.

To have a transaction performed, an initialized instance of the class which implements the TransactionalCommand interface is passed to the executeTransactionalCommand method of the MithraManagerProvider via this call:

```
MithraManagerProvider.getMithraManager().executeTransactionalCommand(myTransaction
```

The following example illustrate Reladomo transaction management. They are based on the Order processing objects used in earlier sections.

Example 1: Create a new order with 3 order items and corresponding product descriptions. This update will require the insert of new 1 Order object, 3 OrderItem objects, and 3 ProductDescription objects. The steps to perform inside the executeTransaction method would be:

- Create a new Order object and fill in the details via setter methods
- Create 3 new OrderItem objects and fill in the details vai setter methods
- Create 3 new ProductDescription objects and fill in the details via setter methods
- Associate each ProductDescription with their respective OrderItem objects via the OrderItem.setProductDescription(ProductDescription) method

- Assign the 3 OrderItem objects to the Order object by calling Order.getOrderItems.add(OrderItem) method (Note: Order.getOrderItems() returns a OrderItemList upon which the add(OrderItem) method is called to add the new OrderItem to the list.)
- Commit all the changes by calling cascadeInsert() on the Order object.

Since all the objects are related either directly or indirectly to the Order object, Reladomo will automatically insert all the new objects so it is unnecessary to call insert() on each of them. However, as an alternative, you could call insert() on each newly created object individually rather than the cascadeInsert() call. This might be easier depending on the situation.

The following is what the code might look like:

```
public class CreateNewOrder implements TransactionalCommand
{
    public void addOrderItem(OrderItemDetails item;)
    {
        myOrderDetailsList.add(item);  // Keep a list of
    }

    public void setOrderDescription(String orderDescription)
    {
        this.orderDescription = orderDescription;
    }

    public Object executeTransaction() throws Throwable
    {
        Order = new Order();
        order.setDescription(this.orderDescription)

        for (OrderItemDetails details : myOrderDetailsList)
        {
            OrderItem orderItem = new OrderItem();
            // Set values on orderItem from details object
            ....

            ProductInfo productInfo = new ProductInfo();
            // Set values on productInfo from details object
            ......

            // Associate ProductInfo Object with OrderItem
            orderItem.setProductInfo(productInfo);

            // Associate OrderItem to Order
            order.getOrderItems().add(OrderItem)
        }

        //Insert all the objects
        order.cascadeInsert();
    }
}
```

Example 1 illustrated what can be done when all the objects involved are new. When adding new objects that will be related to existing objects, the situation is much simpler. See example 2 below:

Example 2: Add a new OrderItem to an existing Order In this example, we are creating a new OrderItem object and associating it with an existing Order. The steps to perform inside the executeTransaction method would be:

- Create a new OrderItem object
- Associate it with the existing Order via the setOrderID(int orderID) method where orderID is the ID of the existing Order
- Call OrderItem() on the new ContactDetail object

NOTE: You do not have to retrieve the Order object if you already know the order ID to use. Simply assigning the value via the setter method on OrderItem is sufficient to establish the relationship such that the list returned by a subsequent call to Order.getOrderItems() would now include the newly created OrderItem.

The following is what the code might look like:

```
public class UpdateExistingOrder implements TransactionalCommand
{
    public void addOrderItem(OrderItemDetails item;)
    {
        myOrderDetailsList.add(item);  // Keep a list of or
    }

    public void setOrderID(long orderID)
    {
        // Record the ID of order to be updated
        this.orderID = orderID;
    }

    public Object executeTransaction() throws Throwable
    {
        for (OrderItemDetails details : myOrderDetailsList)
        {
            OrderItem orderItem = new OrderItem();
            //Set values on orderItem from details object
            ....

            ProductInfo productInfo = new ProductInfo();
            //Set values on productInfo from details object
            ......

            //Associate ProductInfo Object with OrderItem
            orderItem.setProductInfo(productInfo);

            //Associate new OrderItem with existing Order
            orderItem.setOrderID(this.orderID);

            //Save the new OrderItem
            orderItem.insert();
        }
    }
}
```

8. Build your project

Add Ant targets in the build.xml to generate and compile the Reladomo classes. A template driven code generator will generate all the necessary classes for Reladomo's use and generate empty concrete classes that will be modified by users to implement business logic. The code generator is available as an Ant task definition that can be integrated in an Ant target to generate the Reladomo classes that are specific to your project.

build.xml:

```
<project name="myproj" default="compile-myproj">
    <property name="myproj.home" location=<your project home> />
    <property name="myproj.source.dir" location="${myproj.home}/src"/>
    <property name="generated.src.dir" location="${myproj.home}/
generatedsrc"/>
    <property name="myproj.classes.dir" location="${myproj.home}/
classes"/>
    <property name="myproj.lib.dir" location="${myproj.home}/lib" />

    <path id="myproj.compile.classpath">
        <fileset dir="${myproj.lib.dir}">
              <include name="*.jar" />
        </fileset>
    </path>

    <target name="init-myproj">
        <mkdir dir="${myproj.classes.dir}" />
        <mkdir dir="${generated.src.dir}" />
    </target>

    <target name="clean-myproj">
        <delete quiet="true" dir="${myproj.classes.dir}"/>
        <delete quiet="true" dir="${generated.src.dir}"/>
    </target>

    <target name="generate-myproj-mithra-classes">
        <taskdef   name="mithra-gen"

  classname="com.gs.fw.common.mithra.generator.MithraGenerator"
                  loaderRef="mithraGenerator">
              <classpath refid="mithragen.classpath"/>
        </taskdef>
        <mithra-gen   xml="${myproj.home}/xml/mithra/
MithraClassList.xml"
                                generatedDir="${generated.src.dir}"
                                nonGeneratedDir="${myproj.source.dir}"
                                generateConcreteClasses="true"/>
    </target>

    <target name="compile-myproj"
               depends="clean-myproj, init-myproj, generate-myproj-
mithra-classes"
               description="Compile my project  classes">
```

```
        <javac destdir="${myproj.classes.dir}">
            <src path="${myproj.source.dir}"/>
            <src path="${generated.src.dir}"/>
            <classpath refid="myproj.compile.classpath"/>
        </javac>
    </target>
</project>
```