Guided Tour Of Reladomo

A hands-on guide for developers

Guided Tour Of Reladomo: A hands-on guide for developers

Table of Contents

Chapter 1. Preface

Table of Contents

Introduction

The goal of this book is to introduce you to object-relational mapping using Reladomo. It provides a broad overview of various Reladomo features. The goal is to provide a feel and flavor for what is possible with Reladomo.

Audience

This book is written for Java programmers who want to use Reladomo for object-relational mapping. This book assumes that the reader is familiar with the basic concepts behind object-relational mapping (such as domain modeling) and SQL.

How to read this book

This book is intentionally light on commentary and heavy on Java code examples and snippets. The reader is encouraged to consult and run the complete examples, which are distributed with this book.

Examples

The examples are structured as multiple Maven modules. These modules are referenced in various chapters in the book.

To run the examples :

1. Clone `https://github.com/goldmansachs/reladomo-kata`
2. Import the top level `reladomo-tour` as a Maven module
3. Explore and run the main classes and tests in the various child Maven modules.

Conventions

The following conventions have been used:

- Example references - When a program/example listing refers to a source file, the title of the listing is formatted as `<name of maven module>/<name source file>`.
- Elided examples - Elided examples contain the text `// elided for brevity`. For the full example, consult the source file as indicated by the example's title.

Additional Documentation

This book is meant to be a practical guide. It is not reference documentation. Documentation that explores various Reladomo features in more detailcan be found at the *Reladomo Github project page [https://github.com/goldmansachs/reladomo]*. Some of these documents have also been linked in various chapters in this book.

Table of Contents

Chapter 2. Domain Modeling

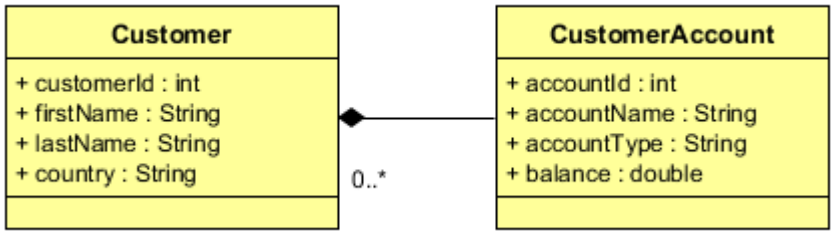
Table of Contents

This chapter introduces domain modeling in Reladomo. We start by building a domain model of an application and then use Reladomo to generate classes and DDL statements from the domain model.

Domain Model

A domain model represents the various entities in a system and their relationships. Consider a backend application for a bank. A simple domain model of this application can be expressed in UML as follows.

Figure 2.1. Bank Domain Model



We have two entities: `Customer` and `CustomerAccount`. We start by defining the attributes of each entity, along with some metadata in a `MithraObject` XML file.

Example 2.1. tour-examples/simple-bank/Customer.xml

```
<MithraObject objectType="transactional"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="mithraobject.xsd">

    <PackageName>simplebank.domain</PackageName>
    <ClassName>Customer</ClassName>
    <DefaultTable>CUSTOMER</DefaultTable>

    <Attribute name="customerId" javaType="int"
            columnName="CUSTOMER_ID"  primaryKey="true"/>
    <Attribute name="firstName" javaType="String"
            columnName="FIRST_NAME"  nullable="false" maxLength="64"/>
    <Attribute name="lastName" javaType="String"
            columnName="LAST_NAME"  nullable="false" maxLength="64"/>
    <Attribute name="country"  javaType="String"
            columnName="COUNTRY"  nullable="false" maxLength="48"/>

</MithraObject>
```

Example 2.2. tour-examples/simple-bank/CustomerAccount.xml

```
<MithraObject objectType="transactional"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="mithraobject.xsd">

    <PackageName>simplebank.domain</PackageName>
    <ClassName>CustomerAccount</ClassName>
    <DefaultTable>CUSTOMER_ACCOUNT</DefaultTable>

    <Attribute name="accountId" javaType="int"
            columnName="ACCOUNT_ID" primaryKey="true"/>
    <Attribute name="customerId" javaType="int"
            columnName="CUSTOMER_ID" nullable="false"/>
    <Attribute name="accountName"  javaType="String"
            columnName="ACCOUNT_NAME" nullable="false" maxLength="48"/>
    <Attribute name="accountType" javaType="String"
            columnName="ACCOUNT_TYPE" nullable="false" maxLength="16"/>
    <Attribute name="balance" javaType="double"
            columnName="BALANCE"/>

</MithraObject>
```

These XML files contain metadata that allows Reladomo to map entities to Java classes (e.g. `ClassName`, `javaType`) and to database tables (e.g. `DefaultTable`, `columName`). The XML schema definition (`mithraobject.xsd`) explains the various components in detail.

Primary Keys

In Reladomo, each object requires a primary key. A primary key might be composed of a single attribute or multiple attributes. For the `Customer` object, the `customerId` attribute is unique. So we simply include a `primaryKey` in the `Attribute` tag.

```
<Attribute name="customerId" javaType="int" columnName="CUSTOMER_ID" primaryKey="true"/>
```

A primary key can also be composed of multiple attributes. If the full name (first name and last name) of a `Customer` are unique, we can create a primary key by adding a `primaryKey` to both these attributes.

```

<Attribute name="firstName" javaType="String"
        columnName="FIRST_NAME" maxLength="16" primaryKey="true"/>
<Attribute name="lastName" javaType="String"
        columnName="LAST_NAME" maxLength="24" primaryKey="true"/>
```

In some cases, a natural primary key does not exist. We can ask Reladomo to generate a primary key by using a `primaryKeyGeneratorStrategy`. Primary key generation is discussed in other Reladomo documentation.

Relationships

In the bank, a `Customer` is related to `CustomerAccounts`. More specifically, a customer can have one or more accounts.

```
<Relationship name="accounts"
            relatedObject="CustomerAccount"
            cardinality="one-to-many"
        this.customerId = CustomerAccount.customerId
</Relationship>
```

The relationship definition contains the expression that relates the two objects. In this case, `Customer` and `CustomerAccount` are related by the `customerId` attribute. The relationship also influences Reladomo's code generation. In particular, the `Customer` class will have a `getAccounts` method. This is because the name of the relationship is `accounts`.

Sometimes it is desirable to navigate relationships in the reverse direction. For example, given a `CustomerAccount`, we may want to navigate to the `Customer` that owns it. We can implement this by adding another `Relationship` in the XML for `CustomerAccount` linking it to `Customer`. A better approach is to use a single `Relationship` in `Customer`, but add the `reverseRelationshipName` attribute.

```
<Relationship name="accounts"
            relatedObject="CustomerAccount"
            cardinality="one-to-many"
            reverseRelationshipName="customer">
        this.customerId = CustomerAccount.customerId
</Relationship>
```

Using the `reverseRelationshipName` attribute provides the benefit of a single definition but with the code generated for both the classes.

Here is the full listing of all the `MithraObject`s.

Example 2.3. tour-examples/simple-bank/Customer.xml

```
<MithraObject objectType="transactional"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="mithraobject.xsd">

    <PackageName>simplebank.domain</PackageName>
    <ClassName>Customer</ClassName>
    <DefaultTable>CUSTOMER</DefaultTable>

    <Attribute name="customerId"  javaType="int"
            columnName="CUSTOMER_ID" primaryKey="true"/>
    <Attribute name="firstName"  javaType="String"
            columnName="FIRST_NAME"  nullable="false" maxLength="64"/>
    <Attribute name="lastName"  javaType="String"
            columnName="LAST_NAME"  nullable="false" maxLength="64"/>
    <Attribute name="country"  javaType="String"
            columnName="COUNTRY"  nullable="false" maxLength="48"/>

   <Relationship name="accounts" relatedObject="CustomerAccount" cardinality="one-to-
many" reverseRelationshipName="customer" relatedIsDependent="true">
        this.customerId = CustomerAccount.customerId
   </Relationship>

</MithraObject>
```

Example 2.4. tour-examples/simple-bank/CustomerAccount.xml

```
<MithraObject objectType="transactional"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="mithraobject.xsd">

    <PackageName>simplebank.domain</PackageName>
    <ClassName>CustomerAccount</ClassName>
    <DefaultTable>CUSTOMER_ACCOUNT</DefaultTable>

    <Attribute name="accountId"  javaType="int"
            columnName="ACCOUNT_ID" primaryKey="true"/>
    <Attribute name="customerId"  javaType="int"
            columnName="CUSTOMER_ID" nullable="false"/>
    <Attribute name="accountName" javaType="String"
            columnName="ACCOUNT_NAME" nullable="false" maxLength="48"/>
    <Attribute name="accountType" javaType="String"
            columnName="ACCOUNT_TYPE" nullable="false" maxLength="16"/>
    <Attribute name="balance"  javaType="double"
            columnName="BALANCE"/>
</MithraObject>
```

Code Generation

To generate classes, Reladomo has to be told about the `MithraObject`s that it should generate code for. This is done via a `MithraClassList`.

Example 2.5. tour-examples/simple-bank/SimpleBankClassList.xml

```
<Mithra>
    <MithraObjectResource name="Customer"/>
    <MithraObjectResource name="CustomerAccount" />
</Mithra>
```

With the class list created, we can generate classes by invoking an Ant task with the class list as input. Here is a snippet showing the Ant task being invoked from Maven.

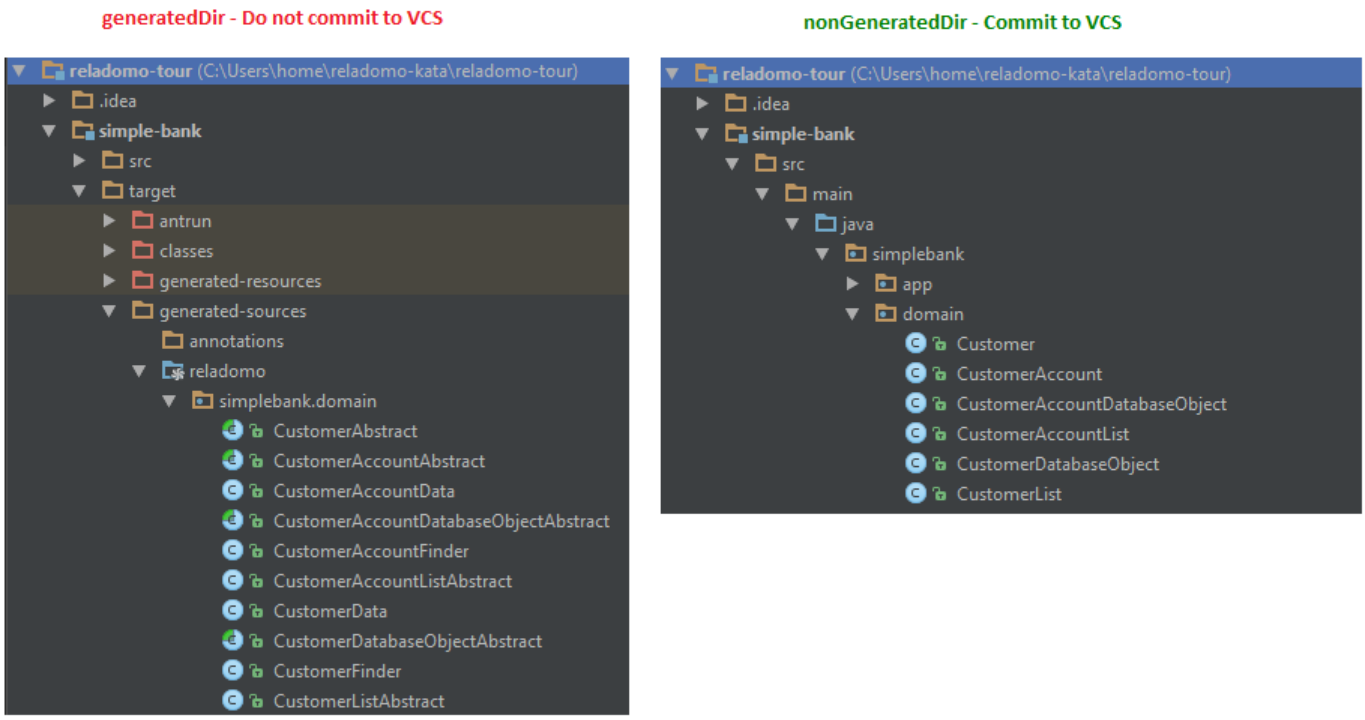
Example 2.6. tour-examples/simple-bank/pom.xml

```
<taskdef name="gen-reladomo" classpath="plugin_classpath"
            classname="com.gs.fw.common.mithra.generator.MithraGenerator"/>
<gen-reladomo
    xml="${project.basedir}/src/main/reladomoxml/SimpleBankClassList.xml"
    generateGscListMethod="true"
    generatedDir="${project.build.directory}/generated-sources/reladomo"
    nonGeneratedDir="${project.basedir}/src/main/java"/>

    // elided for brevity
```

The code generator generates two sets of classes. These are sent to different source directories via the `generatedDir` and `nonGeneratedDir` attributes. The classes in `generatedDir` (below figure on the left) are Reladomo framework classes, which are always generated. (i.e on every run of the code generator). These should never be committed to a version control system.

The classes under `nonGeneratedDir` (below figure on the right) are the main domain classes. Your can add your business logic to these classes and committ them to a version control system. Because you could have custom business logic in these classes, the code generator will not re-generate these classes if they already exist in the target directory.

Figure 2.2. Simple Bank - Generated Classes



Database DDL Generation

Reladomo not only generates Java classes but can also generate DDL statements for bootstrapping the application database. These include scripts to create tables, indices, and foreign keys. DDL generation is an optional feature of Reladomo. You can always choose to manually create database tables that map to the various `MithraObject`s.

Similar to code generation, the DDL generator is invoked with the `MithraClassList` containing the `MithraObject`s for which you want DDLs generated. It also has to be told the type of database for which DDLs have to be generated.

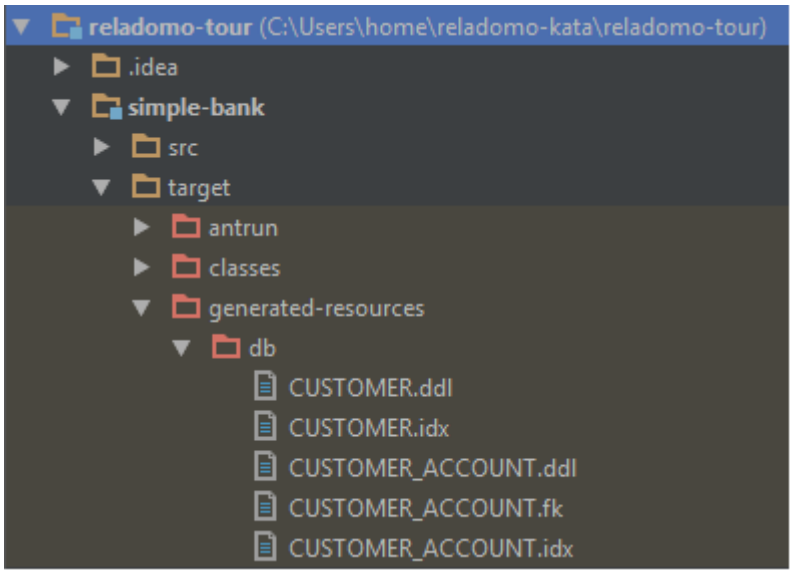
Here is a snippet showing the DDL generator Ant task being invoked from Maven.

Example 2.7. tour-examples/simple-bank/pom.xml

```
<tasks><property name="plugin_classpath" refid="maven.plugin.classpath"/>
    <taskdef name="gen-reladomo-db" classpath="plugin_classpath"

  classname="com.gs.fw.common.mithra.generator.dbgenerator.MithraDbDefinitionGenerator"/>
    <gen-reladomo-db
        xml="${project.basedir}/src/main/reladomoxml/SimpleBankClassList.xml"
        generatedDir="${project.build.directory}/generated-resources/db"
        databaseType="UDB82"/>

        // elided for brevity
</tasks>
```

Running the DDL generator, produces the following files. `.ddl` files contain the create table statements. `.idx` and `.fk` files contain the statements for creating indices and foreign keys.

Figure 2.3. Simple Bank - Generated DDL



Optimal DDLs

The DDLs generated by Reladomo are not meant to be used as is. They are merely meant to be a starting point for further customization. There are two common reasons to customize the generated DDLs. One is to add database specific clauses for specifying table spaces, schemas etc. Another is to add additional indices based on usage patterns seen in the application.

References

- *Object Definition Reference [https://goldmansachs.github.io/reladomo/xsddoc/index.html]*
- *Primary Key Generation [https://goldmansachs.github.io/reladomo/primaryKeyGenerator/PrimaryKeyGenerator.html]*
- *Database Definition Generators [https://goldmansachs.github.io/reladomo/mithraddl/ReladomoDdlGenerator.html]*
- *Object XML Generator [https://goldmansachs.github.io/reladomo/objectxmlgenerator/Generator.html]*
- *Visualize Domain Model Using Reladomo Metadata [https://goldmansachs.github.io/reladomo/visualization/ReladomoVisualization.html]*

Chapter 3. Wiring A Reladomo Application

Table of Contents

This chapter demonstrates the basic template for initializing Reladomo for use in an application.

Connection Manager

Clearly, an application that persists to a database needs a mechanism to acquire a database connection. The job of acquiring a connection is delegated to a connection manager class. How the connection is obtained is immaterial, as long as the connection is associated with a database transaction.

Sourceless Connection Manager

The `SourceLessConnectionManager` is used when all Reladomo objects are persisted in a single database. In other words, if all objects come from a single source database, use a `SourceLessConnectionManager`.

The snippet below shows a simple implementation that uses a direct host/port connection to the database. It also uses a Reladomo utility class, `XAConnectionManager` to acquire a transactional database connection.

Example 3.1. tour-examples/simple-bank/BankConnectionManager.java

```
public class BankConnectionManager implements SourcelessConnectionManager
{
    private XAConnectionManager xaConnectionManager;

    protected SimpleBankConnectionManager() {
        this.createConnectionManager();
    }

    private void createConnectionManager() {
        this.xaConnectionManager = new XAConnectionManager();
        xaConnectionManager.setDriverClassName("com.ibm.db2.jcc.DB2Driver");
        xaConnectionManager.setHostName("my.db.host");
        xaConnectionManager.setPort(12345);
        xaConnectionManager.setJdbcUser("user1");
        xaConnectionManager.setJdbcPassword("password1");
        xaConnectionManager.setMaxWait(500);
        xaConnectionManager.setPoolName("pet store connection pool");
        xaConnectionManager.setPoolSize(10);
        xaConnectionManager.setInitialSize(1);
        xaConnectionManager.initialisePool();
    }

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

    @Override
    public DatabaseType getDatabaseType() {
        return Udb82DatabaseType.getInstance();
    }

    @Override
    public TimeZone getDatabaseTimeZone() {
        return TimeZone.getTimeZone("America/New York");
    }

    //this uniquely identifies the database from which the connection is acquired
    @Override
    public String getDatabaseIdentifier() {
        return xaConnectionManager.getHostName() + ":" + xaConnectionManager.getPort();
    }
    // elided for brevity
}
```

Source Connection Manager

In some cases, Reladomo objects come from multiple source databases. One common case where this can happen is if the database is sharded. In this case, the connection manager's job is to return a shard-specific database connection.

Sharding is discussed in more detail in other Reladomo documentation.

Runtime Configuration

Reladomo's runtime is configured via a `MithraRuntime` XML file. This is similar to the `MithraClassList` in that it lists all the `MithraObjects` that are being used by the application. It is different in that it specifies runtime concerns such as connection management and caching. The connection manager class to be used is indicated via the `ConnectionManager` XML tag.

The following snippet shows a runtime configuration where all the objects are partially cached. This means that Reladomo will cache an object as long as there is sufficient memory. When there is a memory crunch, the cached objects will be garbage-collected.

Example 3.2. tour-examples/simple-bank/SimpleBankClassList.xml

```
<MithraRuntime>
    <ConnectionManager className="simplebank.util.BankConnectionManager">
        <MithraObjectConfiguration className="simplebank.domain.Customer" cacheType="partial" />

  <MithraObjectConfiguration className="simplebank.domain.CustomerAccount" cacheType="partial" />
    </ConnectionManager>
</MithraRuntime>
```

Wiring it all together

With the connection manager and runtime configuration ready, we can initialize Reladomo by initializing the `MithraManager`. As the name suggests, this class manages all Reladomo operations.

Example 3.3. tour-examples/simple-bank/SimpleBankApp.java

```
public class SimpleBankApp
{
    public static void main(String[] args) throws Exception
    {
        new SimpleBankApp().start();
    }

    public SimpleBankApp() throws Exception
    {
        this.initReladomo();
    }

    private void initReladomo() throws Exception
    {
        MithraManager mithraManager = MithraManagerProvider.getMithraManager();
        mithraManager.setTransactionTimeout(60 * 1000);
        InputStream stream = loadReladomoXMLFromClasspath("SimpleBankRuntimeConfiguration.xml");
        MithraManagerProvider.getMithraManager().readConfiguration(stream);
        stream.close();
    }

    private void start()
    {
        //implement app logic
    }
    // elided for brevity
}
```

This simple application does not do anything useful other than initializing Reladomo. In the next chapter, we will extend this application to build a simple REST API for the bank.

Chapter 4. CRUD Operations

Table of Contents

This chapter demonstrates basic CRUD (Create, Read, Update, Delete) operations using Reladomo.

Imagine that we want to build a REST API for our bank. The API for `Customer` could look like this.

```
GET    /customer              Get all Customers
GET    /customer/{id}         Get a Customer by id
GET    /customer/{lastName}   Get all Customers by last name.
POST   /customer              Create a Customer with the POST payload
PUT    /customer/{id}         Update the details of a Customer
DELETE /customer/{id}         Delete a Customer
```

To implement each of the above listed APIs, we need to use Reladomo to create, fetch and modify objects in the database.

Object Creation (C)

Creating objects is straightfoward in Reladomo. For each `MithraObject`, Reladomo generates a no-arg constructor. Simply invoke the constructor and the generated setter methods. These methods correspond to the attributes defined in the `MithraObject` XML. Then insert the object by invoking the generated `insert` method. The implementation of the `POST` API for the `Customer`, will invoke the following code.

```
Customer mickey = new Customer();
mickey.setCustomerId(123);
mickey.setFirstName("Mickey");
mickey.setLastName("Mouse");
```

Invoking multiple setters is clunky. This can be improved by adding a custom constructor to the generated `Customer` class.

```
public Customer(int customerId, String firstName, String lastName)
{
    super();
    this.setCustomerId(customerId);
    this.setFirstName(firstName);
    this.setLastName(lastName);
}
```

The generated classes are meant to be modified. However, the default no-arg constructor has to be preserved because it is used internally by Reladomo. When adding a custom constructor make sure to call the no-arg constructor.

Cascaded Insert

Consider the case where we want to create (insert) a `Customer` and his/her `CustomerAccount` . The naive approach is to instantiate and insert each object individually. Another option is to wire up the object graph and let Reladomo cascade the inserts.

```
Customer mickey = new Customer();
mickey.setCustomerId(123);
mickey.setFirstName("Mickey");
mickey.setLastName("Mouse");

CustomerAccount savingsAccount = new CustomerAccount();
savingsAccount.setAccountId(1000);
savingsAccount.setBalance(100);
savingsAccount.setAccountType("savings");

mickey.getAccounts().add(savingsAccount);

mickey.cascadeInsert();
```

When using cascading inserts, make sure the `relatedIsDependent="true"` attribute is enabled on the `Relationship` definition in the `MithraObject` XML.

In the snippet above, the `cascadeInsert` inserts both the objects. However, the inserts are not atomic. If atomic inserts are needed, simply wrap the insert in a transaction.

```
MithraManagerProvider.getMithraManager().executeTransactionalCommand((tx) -> {
    Customer mickey = new Customer();
    mickey.setCustomerId(123);
    mickey.setFirstName("Mickey");
    mickey.setLastName("Mouse");

    CustomerAccount savingsAccount = new CustomerAccount();
    savingsAccount.setAccountId(1000);
    savingsAccount.setBalance(100);
    savingsAccount.setAccountType("savings");

    mickey.getAccounts().add(savingsAccount);

    mickey.cascadeInsert();
    return null;
});
```

Object Read (R)

Objects are read using generated classes called `Finders`. There are two parts to a finder. The first is the `Operation`. An operation corresponds to an expression in a SQL `where` clause. For each attribute in the `MithraObject`, Reladomo generates a method named after the attribute. These methods return `Attribute` objects which in turn expose methods that return `Operations`.

```
StringAttribute<Customer> firstNameAttribute = CustomerFinder.firstName();
Operation firstNameOperation = firstNameAttribute.eq("Mickey");

IntegerAttribute<Customer> idAttribute = CustomerFinder.customerId();
Operation idOperation = idAttribute.in(IntSets.immutable.of(123, 456, 789));
```

Operations can also be chained. Once an operation is built, one of the generated `findMany`, `findOne` methods is used to actually fetch the objects.

```
CustomerList mickeys = CustomerFinder.findMany(firstNameOperation);
```

Here are some snippets that are used to implement the `GET` API for `Customer`.

```
public Customer getCustomerById(int customerId)
{
    return CustomerFinder.findByPrimaryKey(customerId);
}

public CustomerList getCustomersByLastName(String lastName)
{
    return CustomerFinder.findMany(CustomerFinder.lastName().eq(lastName));
}
```

Caching

When a `Finder` is invoked, Reladomo normally constructs a SQL query and executes it against the database. However, this is not always the case. If the object has been cached in memory, Reladomo will return the cached object instead of hitting the database.

Cursors

When a finder that returns a list is invoked, the entire list is forced into memory. This is not ideal if the list is huge. In such cases, it is better to iterate over the list with a cursor. The snippet below iterates over the first hundred Customers. The cursor is terminated when the `DoWhileProcedure` lambda returns false.

```
CustomerList customers = ...;
MutableIntSet ids = IntSets.mutable.empty();
customers.forEachWithCursor(o -> {
    Customer p = (Customer)o;
    ids.add(p.getCustomerId());
    return ids.size() < 100;
});
```

Deep Fetch

Consider the case of fetching related objects. Let's say we have a customer and are interested in all the accounts owned by this customer. A naive approach to implementing this looks as follows.

```
CustomerList customers = CustomerFinder.findMany(CustomerFinder.firstName().eq("mickey"));
MutableList<Object> mickeysAccounts = Lists.mutable.empty();
for (Customer customer : customers)
{
    for (CustomerAccount account : customer.getAccounts())
    {
        mickeysAccounts.add(account.getAccountId());
    }
}
```

If there is a single customer with first name "mickey" and he has five accounts, the above snippet results in six database hits. The first hit is to fetch the record for mickey (line 1). There are five more hits to the database, one per account (line 7). In ORM literature, this anti-pattern is referred to as the *N+1 query problem*.

Clearly this can be very inefficient for large lists. This problem is fixed by using a "deep fetch". When a deep fetch is performed, Reladomo will not only fetch the object in question, but will also traverse the relationship graph to fetch related objects. The above example using deep fetch is as follows.

```
CustomerList customers = CustomerFinder.findMany(CustomerFinder.firstName().eq("mickey"));

customers.deepFetch(CustomerFinder.accounts());  // configure deep fetch

MutableList<Object> mickeysAccounts = Lists.mutable.empty();
for (Customer customer : customers)
{
    for (CustomerAccount account : customer.getAccounts())
    {
        mickeysAccounts.add(account.getAccountId());
    }
}
```

All of the code remains the same except for line 2 where we configure the deep fetch. For each `Relationship` defined in the `MithraObject`, Reladomo generates a related finder instance which can be used to fetch related objects. In this case, the `accounts` related finder is because of the `accounts` relationship between `Customer` and `CustomerAccount`.

```
<Relationship name="accounts"
    relatedObject="CustomerAccount"
    cardinality="one-to-many"
    reverseRelationshipName="customer">
        this.customerId = CustomerAccount.customerId
   </Relationship>
```

Now with the deep fetch configured, Reladomo issues a single query against the database.

Object Modification (U)

Objects retrieved via finders are "live" objects. Any changes made to these objects via the generated setters are reflected in the database immediately.

In the following snippet, execution of the setter in line 2 results in a SQL update statement being executed against the database.

```
Customer mickey = CustomerFinder.findOne(CustomerFinder.firstName().eq("mickey"));
mickey.setFirstName("SuperMickey");
```

In some cases this is not desirable. For example, you might want a set of updates to one or more objects to be executed atomically. In such cases, the solution is to wrap the updates in a transaction.

```
MithraManagerProvider.getMithraManager().executeTransactionalCommand((tx) -> {
    Customer mickey = CustomerFinder.findOne(CustomerFinder.firstName().eq("mickey"));
    mickey.setFirstName("Mickey");
    mickey.setLastName("Mouse");
    return null;
});
```

Detached Objects

As described previously, an object fetched via a Finder is a "live" object ; changes to it are immediately flushed to the database. To disable these flushes, simply detach the object from the database.

Imagine you are building a UI that allows users to change multiple attributes of an account. In this case, you do not want each change to be flushed to the database, because the user could decide not apply his changes. Detached objects are handy in use cases like this, where there is a good chance that changes made to an object might have to be discarded.

```
 1 public void testDetached()
 2 {
 3     Customer mickey = CustomerFinder.findByPrimaryKey(1);
 4     assertEquals("mickey", mickey.getFirstName());
 5
 6     p.setFirstName("Mickey");
 7     assertEquals("Mickey", CustomerFinder.findByPrimaryKey(1).getFirstName());
 8
 9     Customer detached = p.getDetachedCopy();
10     detached.setFirstName("SuperMickey");
11     assertEquals("Mickey", CustomerFinder.findByPrimaryKey(1).getFirstName());
12 }
```

In the above snippet, the set of the first name in line 6 is flushed to the database right away. In line 9 the object is detached via the `getDetachedCopy` method. After it has been detached, the set of the first name in the subsequent line is not flushed to the database.

Detaching an object not only detaches that object, but the entire object graph attached to the object.

Database flush

In general a setter invocation results in the change being flushed to the database. But this is not always the case. When inside a transaction, all updates are flushed only when the transaction is committed.

Object Deletion (D)

Deleting an object involves invoking the generated delete method.

```
Operation idOp = CustomerAccountFinder.accountId().eq(100);
CustomerAccount account = CustomerAccountFinder.findOne(idOp);
account.delete();
```

As with the insert operation, a cascading delete operation is available as well.

Example 4.1. simple-bank/CustomerResource.java

```
@DELETE
@Path("/{id}")
public Response deleteCustomer(@PathParam("id") int customerId)
{
    Customer customer = CustomerFinder.findOne(CustomerFinder.customerId().eq(customerId));
    customer.cascadeDelete();
    return Response.ok().build();
}
```

And, as with create and modification operations, deletes can be wrapped in a transaction as well.

Chapter 5. A Complete Reladomo Application

Table of Contents

This chapter does not introduce any new Reladomo concepts. It uses all the concepts introduced in the previous chapters to build a REST API for the bank domain model introduced in Chapter 1.

It is assumed you have a basic familiarity with REST and building REST APIs with Jersey.

Jersey Boilerplate

REST Resource

The first step is to build a Jersey resource class that implements the REST API. The implementation of the API uses the Reladomo CRUD APIs that were reviewed in the previous chapter.

The `CustomerResource` class implements the `Jersey` resource for `Customer.`

Example 5.1. tour-examples/simple-bank/CustomerResource.java

```
@Path("/api/customer")
public class CustomerResource
{
    @POST
    public Response createCustomer(
            @FormParam("customerId") int customerId,
            @FormParam("firstName") String firstName,
            @FormParam("lastName") String lastName)
    {
        Customer customer = new Customer();
        customer.setCustomerId(customerId);
        customer.setFirstName(firstName);
        customer.setLastName(lastName);
        customer.insert();
        return Response.ok().build();
    }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Customer getCustomerById(@PathParam("id") int customerId) throws
 JsonProcessingException
    {
        return CustomerFinder.findByPrimaryKey(customerId);
    }

    // elided for brevity
}
```

REST Server Main Class

The `SimpleBankServer` class initializes Reladomo with the `MithraRuntime` XML and starts the web server with the Jersey resource class.

Example 5.2. tour-examples/simple-bank/SimpleBankServer.java

```
public class SimpleBankServer
{
    private ResourceConfig config;

    public SimpleBankServer(String runtimeConfigXML) throws Exception
    {
        this.initReladomo(runtimeConfigXML);
    }

    protected void initReladomo(String runtimeConfigXML) throws Exception
    {
        MithraManager mithraManager = MithraManagerProvider.getMithraManager();
        mithraManager.setTransactionTimeout(60 * 1000);
        InputStream stream = loadReladomoXMLFromClasspath(runtimeConfigXML);
        MithraManagerProvider.getMithraManager().readConfiguration(stream);
        stream.close();
    }

    private InputStream loadReladomoXMLFromClasspath(String fileName) throws Exception
    {
        InputStream stream = SimpleBankServer.class
                                .getClassLoader().getResourceAsStream(fileName);
        if (stream == null)
        {
            throw new Exception("Failed to locate " + fileName + " in classpath");
        }
        return stream;
    }

    protected void initResources()
    {
        this.config = new ResourceConfig(CustomerResource.class);
        config.register(JacksonFeature.class);
        config.register(SimpleBankJacksonObjectMapperProvider.class);
    }

    public void start() throws IOException
    {
        initResources();
        URI baseUri = UriBuilder.fromUri("http://localhost/").port(9998).build();
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(baseUri, config);
        server.start();
    }

    public static void main(String[] args) throws Exception
    {
        String runtimeConfigXML = "reladomoxml/SimpleBankRuntimeConfiguration.xml";
        new SimpleBankServer(runtimeConfigXML).start();
    }

    // elided for brevity
}
```

JSON Serialization

The REST API's endpoints consume JSON input and produce JSON output. In the `CustomerResource` class, we have used the Reladomo generated classes like `Customer` as the data transfer POJOs.

But the generated classes are not pure POJOs (in the Java sense). Therefore we need to add some extra plumbing to help with the JSON serialization and deserialization. This can be seen in the `initResources` method of the `SimpleBankServer` class.

Note that in Reladomo it doesn't matter how you choose to implement JSON serialization and deserialization.

Chapter 6. Testing With Reladomo

Table of Contents

One of the features that distinguishes Reladomo from other ORM frameworks is its first class support for testing. This chapter demonstrates writing simple and fast tests for a Reladomo application.

Reladomo's testing philosophy is very simple. You should be able to test your application without the usual ceremony of mocks and other test helpers. Reladomo facilitates this by starting and populating an in-memory H2 database with test data. All of you have to do is to point your application against this in-memory database and exercise your application's API/interface.

Test Connection Manager

Chapter 2 introduced the `ConnectionManager` which is responsible for obtaining a database connection. Since we are going to be connecting to an in-memory test database, we cannot simply use the same connection manager as we would in production.

Instead we use `com.gs.fw.common.mithra.test.ConnectionManagerForTests`, which is included in the Reladomo test jars. This class is a convenience class with factory methods for creating a connection manager. You don't have to do much other than to simply use this class.

Test Runtime Configuration

Chapter 2 introduced the `MithraRuntime` XML file. This file ties together the various Reladomo domain model classes along with the `ConnectionManager` to fetch and persist these objects.

The `MithraRuntime` XML for testing differs from production in two ways. First, it uses `com.gs.fw.common.mithra.test.ConnectionManagerForTests`. Second, it has an extra `Property` tag. This tag is used to simulate multiple databases in the same in-memory H2 database server.

The snippet below shows a testing `MithraRuntime` with entities being loaded from two databases.

```
<MithraRuntime>
    <ConnectionManager className="com.gs.fw.common.mithra.test.ConnectionManagerForTests">
        <Property name="resourceName" value="test_db"/>
        <MithraObjectConfiguration className="com.gs.fw.myapp.Foo" cacheType="partial"/>
        <MithraObjectConfiguration className="com.gs.fw.myapp.Bar" cacheType="partial"/>
        ...
    </ConnectionManager>
    <ConnectionManager className="com.gs.fw.common.mithra.test.ConnectionManagerForTests">
        <Property name="resourceName" value="desk_db"/>
        <MithraObjectConfiguration className="com.gs.fw.myapp.Product" cacheType="partial"/>
        <MithraObjectConfiguration className="com.gs.fw.myapp.Account" cacheType="partial"/>
        ...
    </ConnectionManager>
 </MithraRuntime>
```

The snippet below shows the `SimpleBankTestRuntimeConfiguration` XML in its entirety.

Example 6.1. tour-examples/simple-bank/SimpleBankTestRuntimeConfiguration.xml

```
<MithraRuntime
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="mithraruntime.xsd">
    <ConnectionManager className="com.gs.fw.common.mithra.test.ConnectionManagerForTests">
        <Property name="testResource" value="mithra_db"/>
        <MithraObjectConfiguration cacheType="partial" className="simplebank.domain.Customer"/>

  <MithraObjectConfiguration cacheType="partial" className="simplebank.domain.CustomerAccount"/>
    </ConnectionManager>
</MithraRuntime>
```

Test Data

Reladomo will not only start a in-memory H2 database, but also populate it with data. All you have to do is provide a test data file. The following snippet describes the data file's format.

```
class <classname>
<attribute name 1>, <attribute name 2>... <attribute name N>
<value 1, 1>, <value 1, 2> ... <value 1, N>
<value 2, 1>, <value 2, 2> ... <value 2, N>
...
...
<value M, 1>, <value M, 2> ... <value M, N>
```

The snippet below shows the `SimpleBankTestData.txt` file that will be used to test the REST API we built in chapter 4.

Example 6.2. tour-examples/simple-bank/SimpleBankTestData.txt

```
class simplebank.domain.Customer
customerId, firstName, lastName, country
1, "mickey", "mouse", "USA"
2, "minnie", "mouse", "USA"
3, "peter", "pan", "Neverland"

class simplebank.domain.CustomerAccount
accountId, customerId, accountName, accountType, balance
100, 1, "mickey mouse club", "savings", 5000
101, 2, "retirement", "savings", 10000
102, 3, "fun stuff", "checking", 3000
```

Test Resource

The `MithraTestResource` wires everything together. It is used to load the `MithraClassList`, and initialize the test database with test data. The following is a snippet from `SimpleBankAPITest`, showing a complete test.

Example 6.3. tour-examples/simple-bank/SimpleBankAPITest.java

```
public class SimpleBankAPITest
{
    private String testRuntimeConfigXML = "testconfig/SimpleBankTestRuntimeConfiguration.xml";

    @Before
    public void setup() throws Exception
    {
        intializeReladomoForTest();
        initializeApp();
    }

    private void intializeReladomoForTest()
    {
        MithraTestResource testResource = new MithraTestResource(testRuntimeConfigXML);
        ConnectionManagerForTests connectionManager =
 ConnectionManagerForTests.getInstance("test_db");
        testResource.createSingleDatabase(connectionManager, "testconfig/
SimpleBankTestData.txt");
        testResource.setUp();
    }

    private void initializeApp() throws Exception
    {
        new SimpleBankServer(testRuntimeConfigXML).start();
    }

    @Test
    public void testGetCustomer()
    {
        WebTarget target = webTarget("/api/customer/1");
        Response response = target.request(MediaType.APPLICATION_JSON_TYPE).get();

        assertEquals(Response.Status.OK.getStatusCode(), response.getStatus());
        Customer mickey = response.readEntity(Customer.class);
        assertEquals(1, mickey.getCustomerId());
        assertEquals("mickey", mickey.getFirstName());
        assertEquals("mouse", mickey.getLastName());

        CustomerAccountList mickeysAccounts = mickey.getAccounts();
        assertEquals(1, mickeysAccounts.size());
        CustomerAccount clubhouseAccount = mickeysAccounts.get(0);
        assertEquals(100, clubhouseAccount.getAccountId());
        assertEquals("mickey mouse club", clubhouseAccount.getAccountName());
        assertEquals("savings", clubhouseAccount.getAccountType());
        assertEquals(5000, clubhouseAccount.getBalance(), 0);
    }
    // elided for brevity
}
```

Debugging

When using an ORM, it is useful to inspect the SQL statements that are being executed. In Reladomo, SQL logging is enabled by setting the log level of the SQL loggers to DEBUG.

The following snippet shows a Logback configuration file that enables SQL logging. The same can be implemented with other logging frameworks such as Log4j.

Example 6.4. simple-bank/logback.xml

```
<configuration>
    <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <logger name="com.gs.fw.common.mithra.sqllogs" level="DEBUG"></logger>
    <logger name="com.gs.fw.common.mithra.batch.sqllogs" level="DEBUG"></logger>
    <logger name="com.gs.fw.common.mithra.test.sqllogs" level="DEBUG"></logger>
    <!--
    <logger name="com.gs.fw.common.mithra.test.multivm.SlaveVm" level="DEBUG"></logger>
    -->

    <root level="INFO">
        <appender-ref ref="stdout"/>
    </root>
</configuration>
```

The following is a snippet of running a test with SQL logging enabled. Logs are generated not only for main code (i.e SQL executed by the generated Reladomo classes), but also for test code like the `MithraTestResource` when it populates the in-memory H2 database.

The logs also include the number of objects affected and timing information.

```
// in-memory database setup

H2DbServer - Starting H2 database Server
H2DbServer - H2 database Server Started
Customer - executing statement drop table CUSTOMER
Customer - executing statement create table CUSTOMER ( CUSTOMER_ID integer not null,FIRST_NAME
 varchar(64) not null,LAST_NAME varchar(64) not null,COUNTRY varchar(48) not null )
Customer - executing statement CREATE UNIQUE INDEX I_CUSTOMER_PK ON CUSTOMER (CUSTOMER_ID)
CustomerAccount - executing statement drop table CUSTOMER_ACCOUNT
CustomerAccount - executing statement create table CUSTOMER_ACCOUNT ( ACCOUNT_ID integer not
 null,CUSTOMER_ID integer not null,ACCOUNT_NAME varchar(48) not null,ACCOUNT_TYPE varchar(16) not
 null,BALANCE double )
CustomerAccount - executing statement CREATE UNIQUE INDEX I_CUSTOMER_ACCOUNT_PK ON
 CUSTOMER_ACCOUNT (ACCOUNT_ID)
Customer - executing statement truncate table CUSTOMER
CustomerAccount - executing statement truncate table CUSTOMER_ACCOUNT

// inserting test data into the in-memory database

Customer - executing statement INSERT INTO CUSTOMER (CUSTOMER_ID,FIRST_NAME,LAST_NAME,COUNTRY)
 values (1,'mickey','mouse','USA')
Customer - executing statement INSERT INTO CUSTOMER (CUSTOMER_ID,FIRST_NAME,LAST_NAME,COUNTRY)
 values (2,'minnie','mouse','USA')
Customer - executing statement INSERT INTO CUSTOMER (CUSTOMER_ID,FIRST_NAME,LAST_NAME,COUNTRY)
 values (3,'peter','pan','Neverland')
CustomerAccount - executing statement INSERT INTO CUSTOMER_ACCOUNT
 (ACCOUNT_ID,CUSTOMER_ID,ACCOUNT_NAME,ACCOUNT_TYPE,BALANCE) values (100,1,'mickey mouse
 club','savings',5000.0)
CustomerAccount - executing statement INSERT INTO CUSTOMER_ACCOUNT
 (ACCOUNT_ID,CUSTOMER_ID,ACCOUNT_NAME,ACCOUNT_TYPE,BALANCE) values
 (101,2,'retirement','savings',10000.0)
CustomerAccount - executing statement INSERT INTO CUSTOMER_ACCOUNT
 (ACCOUNT_ID,CUSTOMER_ID,ACCOUNT_NAME,ACCOUNT_TYPE,BALANCE) values (102,3,'fun
 stuff','checking',3000.0)

// SQL executed to satisfy a finder query

Customer - connection:1690573857 find with: select
 t0.CUSTOMER_ID,t0.FIRST_NAME,t0.LAST_NAME,t0.COUNTRY from CUSTOMER t0 where  t0.CUSTOMER_ID = 1
Customer - retrieved 1 objects, 13.0 ms per
CustomerAccount - connection:1690573857 find with: select
 t0.ACCOUNT_ID,t0.CUSTOMER_ID,t0.ACCOUNT_NAME,t0.ACCOUNT_TYPE,t0.BALANCE from CUSTOMER_ACCOUNT t0
 where  t0.CUSTOMER_ID = 1
CustomerAccount - retrieved 1 objects, 1.0 ms per
```

Too much logs!

Depending on your application, SQL logging can be very verbose and spam your logs. In general, SQL logging should not be enabled by default in production.

References

- *Reladomo Test Resource [https://goldmansachs.github.io/reladomo/mithraTestResource/ReladomoTestResource.html]*

Table of Contents

Chapter 7. Chaining

Table of Contents

This chapter introduces chaining and it's variants. It also introduces a motivating example that will be used to illustrate the various types of chaining in subsequent chapters.

Types of Chaining

Chaining refers to how data is stored in a relational database such that changes to the data can be tracked.

Changes can be tracked along two time dimensions:

- Business Time - This is when the change actually occurred in the world
- Processing Time - This is when the change actually was recorded in the database

Note

Business time need not always be the time when the change actually occurred. It can also be the time we want it to be. For example, a bank might balance its books every day at 6.30 PM even though some banking activity might happen after 6.30 PM.

There are three types of chaining :

- Audit-Only - In this type of chaining, changes are recorded with *only the processing-time*
- Business-Only - In this type of chaining, changes are recorded with *only the business-time*
- Bitemporal - In this type of chaining, changes are recorded with *both the business-time and processing-time*

Motivating Example - The SlowBank

Consider the case of the "SlowBank". The bank has several ATMs but the ATM's connectivity to the bank's backend systems is flaky. When an ATM is not able to communicate with the bank's backend, ATM transactions are recorded locally. Later someone is sent to fetch these local transactions which are manually applied to the bank's database.

This delay in updating the bank's database means that your bank balance is not always up-to-date. Sometimes the bank says you have less money than you actually have ; sometimes it says you have more money than you actually have! All of this is very frustrating. But the bank happily adjusts your balance everytime you report a discrepancy.

However, the bank has a new problem. They now need to be able to reason about how the account's balance changed over time.

As we will see in the following chapters, each of the various chaining models allows us to reason about changes to the account in different ways.

Chapter 8. Audit-Only Chaining

Table of Contents

This chapter introduces the concept of audit-only chaining for relational databases.

Audit-Only Chaining

In audit-only chaining, all changes to a database are tracked along the processing-time dimension. This is the time at which a change is actually recorded in the database, irrespective of the time the change occurred in the world.

Audit-Only Updates To An Account

In this section we look at a sequence of transactions to an account and demonstrate how audit-only chaining is used to capture history.

Jan 1 - Open an account with $100

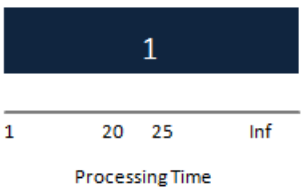
On Jan 1 you open a new bank account with a balance of $100. The bank updates it's database (table) with an entry for your account. Since audit-only chaining is being used, each row in the table has two timestamp columns:

- IN_Z - This is the time when the row was added to the table
- OUT_Z - Along with IN_Z, this defines the range in which this row is valid

The table looks as follows. (The number to the right of every row provides an easy way to refer to rows in this document.)

Figure 8.1. Bank Account On Jan 1

Account #	Balance	IN_Z	OUT_Z	
1	100	Jan 1	Infinity	1



Row 1 records the following facts:

- The account was created today (Jan 1). So IN_Z = Jan 1. This example will use dates (formatted as 'Jan 1' for simplicity) instead of timestamps.

- This is the only row for this account. And we mark this row as valid by setting OUT_Z to Infinity. Infinity is a magic timestamp, in the sense that it cannot possibly be a valid date in the system e.g. 9999/1/1.

The chart on the right (which is not drawn to scale), is a handy tool to visualize the progression of the chaining. The chart visualizes each row as a rectangle. In this case, there is only one row that extends to infinity along the processing-time dimension.

Jan 17 - Deposit $50

A couple of weeks later on Jan 17 you deposit $50. Because of a flaky connection to the bank, the ATM does not send your deposit to the bank right away. While you walk away thinking your account has $150, your account actually has only $100.

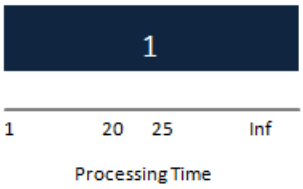
Missed Deposit !!

The table remains unchanged as the deposit was never posted to the account.

Figure 8.2. Bank Account On Jan 17

Account #	Balance	IN_Z	OUT_Z
1	100	Jan 1	Infinity

1



Jan 20 - Deposit $200

A few days later, on Jan 20 you deposit $200 at one of the ATMs.

The goal is to track changes along processing-time. So the bank cannot simply update the balance in Row 1. They cannot delete Row 1 and insert a new row either, as that loses history.

In general, making changes to a audit-only chained database is a two step process :

- Invalidate the row whose view of the world is incorrect
- Add a new row to reflect the new view of the world

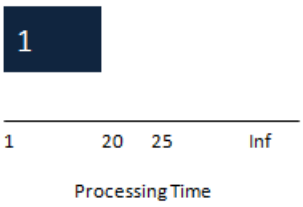
Invalidate row

Row 1 currently states that the balance is $0 from Jan 1 to Infinity. This is not true anymore as the bank just accepted a $200 deposit on Jan 20. So we invalidate Row 1 by setting its OUT_Z to today (Jan 20).

Figure 8.3. Bank Account On Jan 20 - After invalidating existing rows

Account #	Balance	IN_Z	OUT_Z
1	100	Jan 1	Jan 20

1



Adding new rows

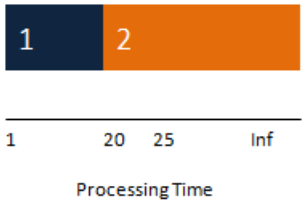
Our new view of the world is as follows :

- From Jan 20 to Infinity, balance = $300 (opening balance + $200 deposit)

So we add Row 2 to capture this fact. The IN_Z and OUT_Z of this row captures the fact that this change was made today and that this row represents the latest state of the account (i.e OUT_Z is Infinity)

Figure 8.4. Bank Account On Jan 20 - After adding new rows

Account #	Balance	IN_Z	OUT_Z	
1	100	Jan 1	Jan 20	1
1	300	Jan 20	Infinity	2



Jan 25 - Correct the missing $50

On Jan 25 you check your bank account and realize that the $50 deposit on Jan 17 has not been posted to the account. Furious, you call the bank to complain. They are apologetic and are willing to update the account.

Just as before, the bank follows this approach :

- Invalidate rows whose view of the world is incorrect
- Add new rows to reflect the new view of the world

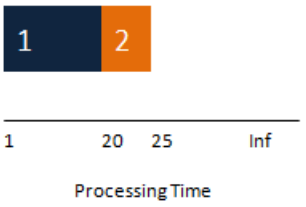
Invalidate row

Row 1 is already invalid. It does not need to be updated.

Row 2 is invalid . So we invalidate it by setting its OUT_Z to today (Jan 25).

Figure 8.5. Bank Account On Jan 25 - After invalidating existing rows

Account #	Balance	IN_Z	OUT_Z	
1	100	Jan 1	Jan 20	1
1	300	Jan 20	Jan 25	2



Add new row

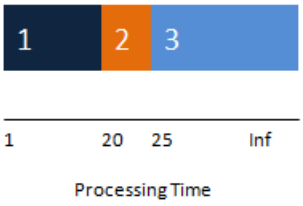
Our new view of the world is as follows :

- From Jan 25 to Infinity, balance = $350 (opening balance + deposit of $200 on Jan 20 + correction deposit of $50 on Jan 25)

Since we are adding this row today (Jan 25), the IN_Z of the newly added row = Jan 25.

Figure 8.6. Bank Account On Jan 25 - After adding new row

Account #	Balance	IN_Z	OUT_Z	
1	100	Jan 1	Jan 20	1
1	300	Jan 20	Jan 25	2
1	350	Jan 25	Infinity	3



To be more explicit, the table as it stands now, simply states that the latest balance is $350. While this is true, we have also lost history. In particular, the table does not tell us that the $50 adjustment made on Jan 25 (processing date) was really an adjustment for Jan 17 (business date).

This is a fundamental limitation of audit-only chaining. Since it only tracks when changes were made (processing-time), there is no information about when these changes actually occurred in the world.

If all you are trying to to is implement an audit trail of changes, audit-only chaining is sufficient.

References

- *Developing Time-Oriented Database Applications in SQL, Richard T. Snodgrass [http://www2.cs.arizona.edu/~rts/tdbbook.pdf]*

Chapter 9. Audit-Only Chaining API

Table of Contents

This chapter introduces various Reladomo APIs that are used to update a audit-only chained table. We will use the SlowBank example from the previous chapter and write code for each of the updates.

Domain Model

Audit-only chaining requires two additional timestamp columns. Also, Reladomo needs to be told that the domain objects are audit-only chained. All of this means that we need to update the `MithraObject` XML definitions.

When temporal chaining is used, the entire object graph is temporally chained. In this case, it means that both the `Customer` and `CustomerAccount` objects are chained. It is very unusual to have only parts of an object graph chained.

Mixing temporal and non temporal objects

In some cases temporal and non temporal objects can be mixed. For example, a temporal object could refer to a non-temporal object. But a non-temporal object cannot refer to a temporal one (as there can be multiple versions of the object).

The snippet below shows the `Customer` `MithraObject` with the new attribute to enable audit-only chaining.

Example 9.1. tour-examples/auditonly-bank/Customer.xml

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

<MithraObject objectType="transactional"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="mithraobject.xsd">

    <PackageName>auditonlybank.domain</PackageName>
    <ClassName>Customer</ClassName>
    <DefaultTable>CUSTOMER</DefaultTable>

    <AsOfAttribute name="processingDate" fromColumnName="IN_Z" toColumnName="OUT_Z"
                   toIsInclusive="false"
                   isProcessingDate="true"

  infinityDate="[com.gs.fw.common.mithra.util.DefaultInfinityTimestamp.getDefaultInfinity()]"

  defaultIfNotSpecified="[com.gs.fw.common.mithra.util.DefaultInfinityTimestamp.getDefaultInfinity()]"
    />

    // elided for brevity
</MithraObject>
```

Audit-only chaining is enabled by adding a `AsOfAttribute` element to the object. The `AsOfAttribute` declares that

- The name of the attribute is `processingDate` and that Reladomo should call the corresponding columns `IN_Z` and `OUT_Z`
- This attribute is for the processing date (`isProcessing=true`)

Reladomo needs to be told what is the value of Infinity. To recap, infinity can be any date that can be reasonably expected to not be part of normal chaining history. The `infinityDate` attribute points to a simple helper class which returns a `java.sql.Timestamp` object to be used as Infinity.

Other than this attribute, the rest of the `MithraObject` definition remains the same as before.

Generated Code

Code generation is the same with non-temporal objects. But because the `MithraObjects` are audit-only chained, the generated classes have a few additions.

First, each generated class's constructor accepts a processing date timestamp.

Example 9.2. tour-examples/auditonly-bank/Customer.java

```
public class Customer extends CustomerAbstract
{
    public Customer(Timestamp processingDate)
 {
  super(processingDate);
  // You must not modify this constructor. Mithra calls this internally.
  // You can call this constructor. You can also add new constructors.
 }

 public Customer()
 {
  this(auditonlybank.util.TimestampProvider.getInfinityDate());
 }
}
```

Second, the generated `Finder` classes expose attribute methods that are used to set the processing date when working with the objects.

```
CustomerFinder.processingDate();
```

Putting It All Together

This section will demonstrate chaining in action by implementing each of the events from the previous chapter's motivating example.

Note

The following sections map the code in `AuditOnlyChainingInAction.java` to the motivating example discussed before.

To ensure that the behavior of the program matches the example discussed here, the program explicitly sets the processing time as needed. You should not have to do this in production code. Processing time by default is set to when the database operation is performed.

Example 9.3. tour-examples/auditonly-bank/AuditOnlyChainingInAction.java

```
@Test
public void run() throws Exception
{
    int accountId = 12345;

    createAccount("2017-01-01", accountId);

    fetchLatestAccount(accountId);

    updateBalance("2017-01-20", accountId, 200);

    dumpCustomerAccount(accountId);

    updateBalance("2017-01-25", accountId, 50);

    dumpCustomerAccount(accountId);
}
```

Jan 1 - Open an account with $100

To open an account, we create the `Customer` and `CustomerAccount` and insert them. However, because the objects are audit-only chained, the constructor must be supplied with a processing date. In this case, we are using the overloaded constructor which sets the date to Infinity.

Example 9.4. tour-examples/auditonly-bank/AuditOnlyChainingInAction.java

```
private void createAccount(String date, int accountId)
{
    MithraManagerProvider.getMithraManager().executeTransactionalCommand(tx -> {

        // Simulate the db change happening on a specific date - Do not do this in production
        doNotDoThisInProduction(date, tx);

        Customer customer = new Customer();
        customer.setFirstName("mickey");
        customer.setLastName("mouse");
        customer.setCustomerId(1);
        customer.setCountry("usa");

        CustomerAccount account = new CustomerAccount();
        account.setAccountId(accountId);
        account.setBalance(100);
        account.setAccountType("savings");
        account.setAccountName("retirement");
        customer.getAccounts().add(account);
        customer.cascadeInsert();
        return null;
    });
}
```

Jan 1 - Fetch the account

Here we use the generated `Finder` but we have to specify an additional operation for the business date.

Example 9.5. tour-examples/auditonly-bank/AuditOnlyChainingInAction.java

```
private void fetchLatestAccount(int accountId)
{
    Operation idOp = CustomerAccountFinder.accountId().eq(accountId);
    CustomerAccount account = CustomerAccountFinder.findOne(idOp);
    assertEquals(100, (int) account.getBalance());
}
```

When the code above is run, Reladomo generates the following SQL.

```
select t0.ACCOUNT_ID,t0.CUSTOMER_ID,t0.ACCOUNT_NAME,t0.ACCOUNT_TYPE,t0.BALANCE,t0.IN_Z,t0.OUT_Z
    from CUSTOMER_ACCOUNT t0
    where  t0.ACCOUNT_ID = 12345 and
    t0.OUT_Z = '9999-12-01 23:59:00.000'
```

In other words, Reladomo is looking for the latest state of the account (i.e OUT_Z=Infinity).

Jan 17 - Deposit $50

In the narrative of the SlowBank, this deposit is lost. So we simulate that by not updating the account on Jan 17.

Jan 20 - Deposit $200

To deposit $200, we have to fetch the account and increment its balance by 200. However, updating a row in chained table can result in the creation of new rows. Therefore the update must be wrapped in a transaction for atomicity.

Example 9.6. tour-examples/auditonly-bank/AuditOnlyChainingInAction.java

```
private void updateBalance(String date, int accountId, int deposit)
{
    MithraManagerProvider.getMithraManager().executeTransactionalCommand(tx ->
    {
        // Simulate the db change happening on a specific date - Do not do this in production
        doNotDoThisInProduction(date, tx);

        Operation id = CustomerAccountFinder.accountId().eq(accountId);
        CustomerAccount account = CustomerAccountFinder.findOne(id);
        account.setBalance(account.getBalance() + deposit);
        return null;
    });
}
```

Running this method results in the following SQL statements being executed. Reladomo is invalidating the row (update OUT_Z) with business date (Jan 1 to Infinity) and adding new rows for business date (Jan 1 to Jan 20) and (Jan 20 to Infinity).

```
update CUSTOMER_ACCOUNT
            set OUT_Z = '2017-01-20 00:00:00.000'
            where ACCOUNT_ID = 12345 AND OUT_Z = '9999-12-01 23:59:00.000'

insert into CUSTOMER_ACCOUNT(ACCOUNT_ID,CUSTOMER_ID,ACCOUNT_NAME,ACCOUNT_TYPE,BALANCE,IN_Z,OUT_Z)
    values (12345,1,'retirement','savings',300.0,
        '2017-01-20 00:00:00.000','9999-12-01 23:59:00.000')
```

Jan 20 - Fetch the account (dump history)

This fetch is identical to the fetch on Jan 1. We simply set the `processingDate` on the finder to Jan 20. Sometimes it is useful to be able to fetch the entire history of an object. Example use cases include debugging, generating reports etc.

History along a time dimension can be fetched by using the `equalsEdgePoint()` operation.

Example 9.7. tour-examples/auditonly-bank/AuditOnlyChainingInAction.java

```
private Operation computeHistoryOperation(int accountId)
{
    Operation idOp = CustomerAccountFinder.accountId().eq(accountId);
    Operation processingDateOp = CustomerAccountFinder
            .processingDate()
            .equalsEdgePoint();
    return idOp.and(processingDateOp);
}
```

The `edgePoint` operations instruct Reladomo to generate the following query. As you can see, the where clause does not include any of the temporal columns, resulting in a full dump of the database.

```
select t0.ACCOUNT_ID,t0.CUSTOMER_ID,t0.ACCOUNT_NAME,t0.ACCOUNT_TYPE,t0.BALANCE,t0.IN_Z,t0.OUT_Z
    from CUSTOMER_ACCOUNT t0 where  t0.ACCOUNT_ID = 12345
```

This `Operation` can now be used with the `Finder`. To make this more interestig, let's use Reladomo's `DbExtractor` which can extract the data from a table and among other things dump it to a file.

Example 9.8. tour-examples/auditonly-bank/AuditOnlyChainingInAction.java

```
private String dumpCustomerAccount(int accountId) throws Exception
{
    Operation historyOperation = computeHistoryOperation(accountId);
    Path tempFile = Files.createTempFile(System.nanoTime() + "", "");
    DbExtractor dbExtractor = new DbExtractor(tempFile.toFile().getAbsolutePath(), false);
    dbExtractor.addClassToFile(CustomerAccountFinder.getFinderInstance(), historyOperation);

    String data = Files.readAllLines(tempFile).stream().collect(Collectors.joining("\n"));
    System.out.println(data);
    return data;
}
```

The `DbExtractor`'s output is not only a listing of all the rows in the table. The output is also formatted for copying and pasting into a Reladomo `MithraTestResource` file for use in tests. See `AuditOnlyChainingInActionTestData.txt`.

Jan 25 - Correct the missing $50

Here we need to adjust the balance for Jan 17. Since we are not tracking business-time, we cannot update the balance for Jan 17. The best we can do is fetch the latest state of the account and adjust for the missing $50. As before Reladomo takes care of invalidating existing rows and adding new rows.

Example 9.9. tour-examples/auditonly-bank/AuditOnlyChainingInAction.java

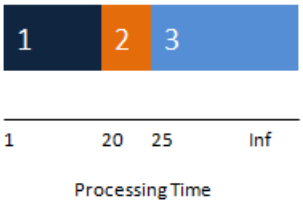
```
private void updateBalance(String date, int accountId, int deposit)
{
    MithraManagerProvider.getMithraManager().executeTransactionalCommand(tx ->
    {
        // Simulate the db change happening on a specific date - Do not do this in production
        doNotDoThisInProduction(date, tx);

        Operation id = CustomerAccountFinder.accountId().eq(accountId);
        CustomerAccount account = CustomerAccountFinder.findOne(id);
        account.setBalance(account.getBalance() + deposit);
        return null;
    });
}
```

Querying a chained table

Consider the visualization from Chapter 7. Each colored rectangle corresponds to a row in the table.

Figure 9.1. Bank Account On Jan 1

Account #	Balance	IN_Z	OUT_Z	
1	100	Jan 1	Jan 20	1
1	300	Jan 20	Jan 25	2
1	350	Jan 25	Infinity	3



Each of these rows can be queried by setting the processing date as required. The following snippet shows querying for the account balance for processing date of Jan 17.

Example 9.10. tour-examples/auditonly-bank/AuditOnlyChainingInAction.java

```
@Test
private void balanceAsOfJan17(int accountId)
{
    Timestamp date = DateUtils.parse("2017-01-17");
    Operation idOp = CustomerAccountFinder.accountId().eq(accountId);
    Operation dateOp = CustomerAccountFinder.processingDate().eq(date);
    CustomerAccount account = CustomerAccountFinder.findOne(idOp.and(dateOp));

    /*
        This balance is incorrect !!
        Even though we adjusted for the missed deposit on Jan 17, there is no way to query for
 the balance as of Jan 17
    */
    assertEquals(100, (int) account.getBalance());
}
```

Running the above code results in Reladomo executuing the following query.

```
select t0.ACCOUNT_ID,t0.CUSTOMER_ID,t0.ACCOUNT_NAME,t0.ACCOUNT_TYPE,t0.BALANCE,t0.IN_Z,t0.OUT_Z
    from CUSTOMER_ACCOUNT t0
    where  t0.ACCOUNT_ID = 5678
    and t0.IN_Z <= '2017-01-17 00:00:00.000' and t0.OUT_Z > '2017-01-17 00:00:00.000'
```

The drawback of audit-only chaining is that the above query cannot be used to query what balance as of a business date.

Changing history

Say on March 1, we fetch the account with a processing date of Jan 17 and update it's balance. Will this add a new row for Jan 17 ?

No. There is no way to change history in audit-only chaining. Reladomo will just add a new row to the new change from March 1 to Infinity.

```
# Before update on March 1

class auditonlybank.domain.CustomerAccount
accountId,customerId,accountName,accountType,balance,processingDateFrom,processingDateTo
5678,2,"retirement","savings",100,"2017-01-01 00:00:00.000","2017-01-20 00:00:00.000"
5678,2,"retirement","savings",300,"2017-01-20 00:00:00.000","2017-01-25 00:00:00.000"
5678,2,"retirement","savings",350,"2017-01-25 00:00:00.000","9999-12-01 23:59:00.000"

# After updating on March1, the account that was fetched with a processing date of Jan 17
# Account balance was increased by $150

class auditonlybank.domain.CustomerAccount
accountId,customerId,accountName,accountType,balance,processingDateFrom,processingDateTo
5678,2,"retirement","savings",100,"2017-01-01 00:00:00.000","2017-01-20 00:00:00.000"
5678,2,"retirement","savings",300,"2017-01-20 00:00:00.000","2017-01-25 00:00:00.000"
5678,2,"retirement","savings",350,"2017-01-25 00:00:00.000","2017-03-01 00:00:00.000"
5678,2,"retirement","savings",500,"2017-03-01 00:00:00.000","9999-12-01 23:59:00.000"
```

Chapter 10. Bitemporal Chaining

Table of Contents

This chapter introduces the concept of bitemporal chaining for relational databases. Continuing the bank example, it shows how by using bitemporal chaining, it is easy to make corrections to historical data without losing any history.

Bitemporal Chaining

In bitemporal chaining, all changes to a database are tracked along two dimensions.

- Business Time - This is when the change actually occurred in the world
- Processing Time - This is when the change actually was recorded in the database

While these two times are often the same, they can be different.

Bitemporal Updates To An Account

In this section we look at a sequence of transactions to an account and demonstrate how bitemporal chaining is used to capture history.

Jan 1 - Open an account with $100

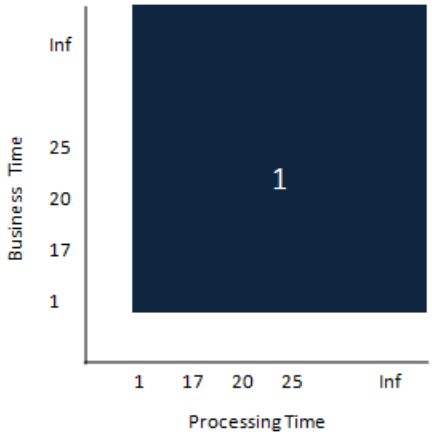
On Jan 1 you open a new bank account with a balance of $100. The bank updates it's database (table) with an entry for your account. Since bitemporal chaining is being used, each row in the table has four timestamp columns:

- FROM_Z and THRU_Z track the validity of the row along the business-time dimension
- IN_Z and OUT_Z track the validity of the row along the processing-time dimension

The table looks as follows. (The number to the right of every row provides an easy way to refer to rows in this document.)

Figure 10.1. Bank Account On Jan 1

Account #	Balance	FROM_Z	THRU_Z	IN_Z	OUT_Z	
1	100	Jan 1	Infinity	Jan 1	Infinity	1



Row 1 records the following facts:

- The account was created on today (Jan 1). So FROM_Z = Jan 1. This example will use dates (formatted as 'Jan 1' for simplicity) instead of timestamps.
- The account was added to the database today (Jan 1). So IN_Z = Jan 1
- This is the only row for this account. And we mark these rows as valid by setting the THRU_Z and OUT_Z to Infinity. Infinity is a magic timestamp, in the sense that it cannot possibly be a valid date in the system e.g. 9999/1/1.

The chart on the right (which is not drawn to scale), is a handy tool to visualize the progression of the chaining. The chart visualizes each row as a rectangle. In this case, there is only one row that extends to infinity along both the time dimensions.

Jan 17 - Deposit $50

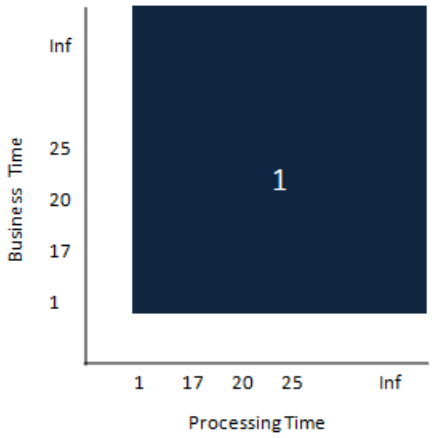
A couple of weeks later on Jan 17 you deposit $50. Because of a flaky connection to the bank, the ATM does not send your deposit to the bank right away. While you walk away thinking your account has $150, your account actually has only $100.

Missed Deposit !!

The table remains unchanged as the deposit was never posted to the account.

Figure 10.2. Bank Account On Jan 17

Account #	Balance	FROM_Z	THRU_Z	IN_Z	OUT_Z	
1	100	Jan 1	Infinity	Jan 1	Infinity	1



Jan 20 - Deposit $200

A few days later, on Jan 20 you deposit $200 at one of the ATMs.

The goal of bitemporal milestoning is to track changes along both dimensions. So the bank cannot simply update the balance in Row 1. They cannot delete Row 1 and insert a new row either, because that loses history.

In general, making changes to a bitemporally-chained database is a two step process :

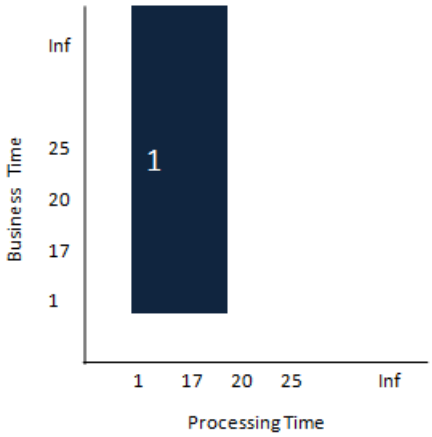
- Invalidate rows whose view of the world is incorrect
- Add new rows to reflect the new view of the world

Invalidating Rows

Row 1 currently states that the balance is $100 from Jan 1 to Infinity. This is not true anymore as the bank just accepted a $200 deposit on Jan 20. So we invalidate Row 1 by setting its OUT_Z to today (Jan 20).

Figure 10.3. Bank Account On Jan 20 - After invalidating existing rows

Account #	Balance	FROM_Z	THRU_Z	IN_Z	OUT_Z	
1	100	Jan 1	Infinity	Jan 1	Jan 20	1



Adding new rows

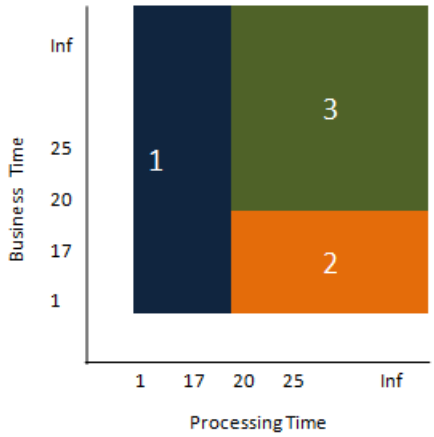
Our new view of the world is as follows :

- From Jan 1 to Jan 20, balance = $100 (opening balance)
- From Jan 20 to Infinity, balance = $300 (opening balance + $200 deposit)

So we add Rows 2 and 3 to capture these facts. The IN_Z and OUT_Z of these rows captures the fact that these changes were made today and that these rows represent the latest state of the account (i.e OUT_Z is Infinity)

Figure 10.4. Bank Account On Jan 20 - After adding new rows

Account #	Balance	FROM_Z	THRU_Z	IN_Z	OUT_Z	
1	100	Jan 1	Infinity	Jan 1	Jan 20	1
1	100	Jan 1	Jan 20	Jan 20	Infinity	2
1	300	Jan 20	Infinity	Jan 20	Infinity	3



Jan 25 - Correct the missing $50

On Jan 25 you check your bank account and realize that the $50 deposit on Jan 17 has not been posted to the account. Furious, you call the bank to complain. They are apologetic and are willing to update the account.

Just as before, the bank wants to preserve history in both dimensions. They follow the same approach :

- Invalidate rows whose view of the world is incorrect
- Add new rows to reflect the new view of the world

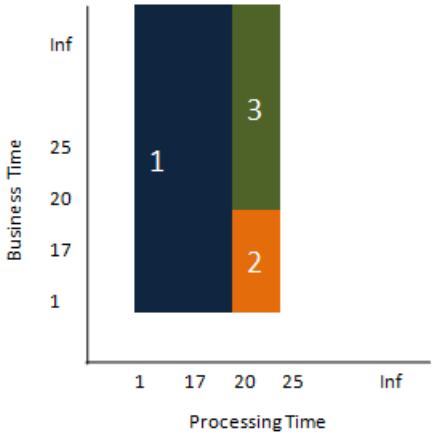
Invalidate rows

Row 1 is already invalid. It does not need to be updated.

Rows 2 and 3 are invalid along the business-time dimension. However, we want to preserve the fact that they are invalid. So we invalidate them by setting their OUT_Z to today (Jan 25).

Figure 10.5. Bank Account On Jan 25 - After invalidating existing rows

Account #	Balance	FROM_Z	THRU_Z	IN_Z	OUT_Z	
1	100	Jan 1	Infinity	Jan 1	Jan 20	1
1	100	Jan 1	Jan 20	Jan 20	Jan 25	2
1	300	Jan 20	Infinity	Jan 20	Jan 25	3



Add new rows

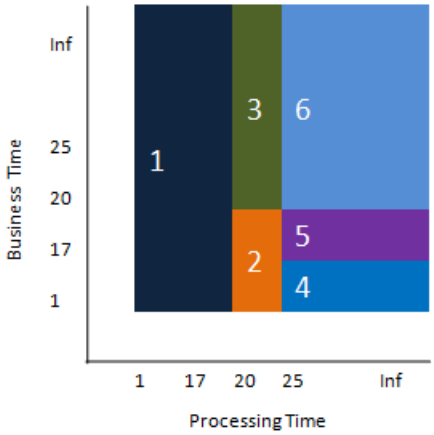
Our new view of the world is as follows :

- From Jan 1 to Jan 17, balance = $100 (opening balance)
- From Jan 17 to Jan 20, balance = $150 (opening balance + deposit of $50 on Jan 17)
- From Jan 20 to Jan Infinity, balance = $1350 (opening balance + deposit of $50 on Jan 17 + deposit of $200 on Jan 20)

Because we are adding these rows today (Jan 25), the IN_Z of these newly added rows = Jan 25.

Figure 10.6. Bank Account On Jan 25 - After adding new rows

Account #	Balance	FROM_Z	THRU_Z	IN_Z	OUT_Z	
1	100	Jan 1	Infinity	Jan 1	Jan 20	1
1	100	Jan 1	Jan 20	Jan 20	Jan 25	2
1	300	Jan 20	Infinity	Jan 20	Jan 25	3
1	100	Jan 1	Jan 17	Jan 25	Infinity	4
1	150	Jan 17	Jan 20	Jan 25	Infinity	5
1	350	Jan 20	Infinity	Jan 25	Infinity	6



To be more explicit about what we just did, we were able to go back in time and record an update for Jan 17 (business-time), along with recording the fact that this update was made today on Jan 25 (processing-time). This is the beauty of bitemporal chaining.

As you can see the visualization now shows a rectangle for each of the rows in the table. It should be clear that because rows are never deleted, we never lose history.

References

- *Developing Time-Oriented Database Applications in SQL, Richard T. Snodgrass [http://www2.cs.arizona.edu/~rts/tdbbook.pdf]*

Chapter 11. Bitemporal Chaining API

Table of Contents

This chapter introduces various Reladomo APIs that are used to update a bitemporally-chained table. We will use the SlowBank example from the previous chapter and write code for each of the updates.

Domain Model

Bitemporal chaining requires us to add four additional timestamp columns and declare the domain objects to be bitemporal. This means that we need to update the `MithraObject` XML definitions.

When temporal chaining is used, the entire object graph is temporally chained. In this case, it means that both the `Customer` and `CustomerAccount` objects are chained. It is very unusual to have only parts of an object graph chained.

Mixing temporal and non temporal objects

In some cases temporal and non temporal objects can be mixed. For example, a temporal object could refer to a non-temporal object. But a non-temporal object cannot refer to a temporal one (because there can be multiple versions of the object).

The snippet below shows the `Customer MithraObject` with the new attributes to enable bitemporal chaining.

Example 11.1. tour-examples/bitemporal-bank/Customer.xml

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

<MithraObject objectType="transactional"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="mithraobject.xsd">

    <PackageName>bitemporalbank.domain</PackageName>
    <ClassName>Customer</ClassName>
    <DefaultTable>CUSTOMER</DefaultTable>

    <AsOfAttribute name="businessDate" fromColumnName="FROM_Z" toColumnName="THRU_Z"
                   toIsInclusive="false"
                   isProcessingDate="false"
                   infinityDate="[bitemporalbank.util.TimestampProvider.getInfinityDate()]"
                   futureExpiringRowsExist="true"
    />
    <AsOfAttribute name="processingDate" fromColumnName="IN_Z" toColumnName="OUT_Z"
                   toIsInclusive="false"
                   isProcessingDate="true"
                   infinityDate="[bitemporalbank.util.TimestampProvider.getInfinityDate()]"

  defaultIfNotSpecified="[bitemporalbank.util.TimestampProvider.getInfinityDate()]"
    />

    // elided for brevity
</MithraObject>
```

Bitemporal chaining is enabled by adding two `AsOfAttribute` elements to the object. The first `AsOfAttribute` declares business date.

- The name of the attribute is `businessDate` and that Reladomo should call the corresponding columns `FROM_Z` and `THRU_Z`
- This attribute is for the business date (`isProcessing=false`)

The second `AsOfAttribute` declares the transaction date.

- The name of the attribute is `processingDate` and that Reladomo should call the corresponding columns `IN_Z` and `OUT_Z`
- This attribute is for the processing date (`isProcessing=true`)

For both these attributes, you will need to set the value of Infinity. To recap, infinity can be any date that can be reasonably expected to not be part of normal chaining history. The `infinityDate` attribute points to a simple helper class which returns a `java.sql.Timestamp` object to be used as Infinity.

Other than these two attributes, the rest of the `MithraObject` definition remains the same as with non-temporal objects.

Generated Code

Code generation is the same with non-temporal-objects. But because the `MithraObjects` are bitemporally-chained, the generated classes have a few additions.

First, each generated class's constructor accepts two timestamps.

Example 11.2. tour-examples/bitemporal-bank/Customer.java

```
public class Customer extends CustomerAbstract
{
 public Customer(Timestamp businessDate, Timestamp processingDate)
 {
  super(businessDate ,processingDate);
  // You must not modify this constructor. Mithra calls this internally.
  // You can call this constructor. You can also add new constructors.
 }

 public Customer(Timestamp businessDate)
 {
  super(businessDate);
 }
}
```

Second, the generated `Finder` classes expose attribute methods that are used to set the business and processing date when working with the objects.

```
CustomerFinder.businessDate();
CustomerFinder.processingDate();
```

Putting It All Together

This section will demonstrate chaining in action by implementing each of the events from the previous chapter's motivating example.

Note

The following sections map the code in `BitemporalChainingInAction.java` to the motivating example discussed before.

To ensure that the behavior of the program matches the example discussed here, the program explicitly sets the processing time as needed. You should not have to do this in production code. Processing time by default is set to when the database operation is performed.

Example 11.3. tour-examples/bitemporal-bank/BitemporalChainingInAction.java

```
@Test
public void run() throws Exception
{
    int accountId = 12345;

    createAccount("2017-01-01", accountId);

    fetchAccountWithBusinessDate("2017-01-01", accountId);

    updateBalance("2017-01-20", accountId, 200);

    dumpCustomerAccount(accountId);

    updateBalance("2017-01-17", "2017-01-25", accountId, 50);

    dumpCustomerAccount(accountId);

    balanceAsOfJan12_OnJan23(accountId);

    dumpCustomer(1);
}
```

Jan 1 - Open an account with $100

To open an account, we create the `Customer` and `CustomerAccount` and insert them. However, because the objects are bitemporal, the constructor must be supplied with a business and processing date. In this case, we are using the overloaded constructor which sets both the dates to Jan 1.

Example 11.4. tour-examples/bitemporal-bank/BitemporalChainingInAction.java

```
private void createAccount(String date, int accountId)
{
    Timestamp jan1 = DateUtils.parse(date);
    MithraManagerProvider.getMithraManager().executeTransactionalCommand(tx ->
    {
        // Simulate the db change happening on a specific date - Do not do this in production
        doNotDoThisInProduction(date, tx);

        Customer customer = new Customer(jan1);
        customer.setFirstName("mickey");
        customer.setLastName("mouse");
        customer.setCustomerId(1);
        customer.setCountry("usa");

        CustomerAccount account = new CustomerAccount(jan1);
        account.setAccountId(accountId);
        account.setBalance(100);
        account.setAccountType("savings");
        account.setAccountName("retirement");
        customer.getAccounts().add(account);
        customer.cascadeInsert();
        return null;
    });
}
```

Jan 1 - Fetch the account

Here we use the generated `Finder` but we have to specify an additional operation for the business date.

Example 11.5. tour-examples/bitemporal-bank/BitemporalChainingInAction.java

```
private void fetchAccountWithBusinessDate(String date, int accountId)
{
    Timestamp jan1 = DateUtils.parse(date);
    Operation idOp = CustomerAccountFinder.accountId().eq(accountId);
    Operation jan1Op = CustomerAccountFinder.businessDate().eq(jan1);
    CustomerAccount account = CustomerAccountFinder.findOne(idOp.and(jan1Op));
    assertEquals(100, (int)account.getBalance());
}
```

When the code above is run, Reladomo generates the following SQL.

```
select t0.ACCOUNT_ID,t0.CUSTOMER_ID,t0.ACCOUNT_NAME,t0.ACCOUNT_TYPE,t0.BALANCE,
            t0.FROM_Z,t0.THRU_Z,t0.IN_Z,t0.OUT_Z
    from CUSTOMER_ACCOUNT t0
    where
    t0.CUSTOMER_ID = 1 and
    t0.FROM_Z <= '2017-01-01 00:00:00.000' and t0.THRU_Z > '2017-01-01 00:00:00.000' and
    t0.OUT_Z = '9999-12-01 23:59:00.000'
```

In other words, Reladomo is looking for the latest state of the account (i.e OUT_Z=Infinity) and where the business date is >= Jan 1.

Jan 17 - Deposit $50

In the narrative of the SlowBank, this deposit is lost. So we simulate that by not updating the account on Jan 17.

Jan 20 - Deposit $200

To deposit $200, we have to fetch the account and increment its balance by 200. However, updating a row in bitemporal table can result in the creation of new rows. Therefore the update must be wrapped in a transaction for atomicity.

Example 11.6. tour-examples/bitemporal-bank/BitemporalChainingInAction.java

```
private void updateBalance(String date, int accountId, int balance)
{
    MithraManagerProvider.getMithraManager().executeTransactionalCommand(tx -> {

        Timestamp timestamp = DateUtils.parse(date);

        // Simulate the db change happening on a specific date - Do not do this in production
        doNotDoThisInProduction(date, tx);

        Operation ts = CustomerAccountFinder.businessDate().eq(timestamp);
        Operation id = CustomerAccountFinder.accountId().eq(accountId);
        CustomerAccount account = CustomerAccountFinder.findOne(ts.and(id));
        account.incrementBalance(balance);
        return null;
    });
}
```

Running this method results in the following SQL statements being executed. Reladomo is invalidating the row (update OUT_Z) with business date (Jan 1 to Infinity) and adding new rows for business date (Jan 1 to Jan 20) and (Jan 20 to Infinity).

```
update CUSTOMER_ACCOUNT set OUT_Z = '2017-01-20 00:00:00.000'
where ACCOUNT_ID = 12345 AND THRU_Z = '9999-12-01 23:59:00.000' AND OUT_Z = '9999-12-01
 23:59:00.000'

insert into CUSTOMER_ACCOUNT (...) values (?,?,?,?,?,?,?,?,?) for 2 objects

insert into CUSTOMER_ACCOUNT (...)
    values (12345,1,'retirement','savings',300.0,
        '2017-01-20 00:00:00.000','9999-12-01 23:59:00.000',
        '2017-01-20 00:00:00.000','9999-12-01 23:59:00.000')

insert into CUSTOMER_ACCOUNT (...)
    values (12345,1,'retirement','savings',100.0,
        '2017-01-01 00:00:00.000','2017-01-20 00:00:00.000',
        '2017-01-20 00:00:00.000','9999-12-01 23:59:00.000')
```

Jan 20 - Fetch the account (dump history)

This fetch is identical to the fetch on Jan 1. We simply set the `businessDate` on the finder to Jan 20. Sometimes it is useful to be able to fetch the entire history of an object. Example use cases include debugging, generating reports etc.

History along a time dimension can be fetched by using the `equalsEdgePoint()` operation. for example, `businessDate().equalsEdgePoint()`. In this case we will use the `equalsEdgePoint` operation on both dimensions to get the full history.

Example 11.7. tour-examples/bitemporal-bank/BitemporalChainingInAction.java

```
private Operation computeHistoryOperation(int accountId)
{
    Operation idOp = CustomerAccountFinder.accountId().eq(accountId);
    Operation processingDateOp = CustomerAccountFinder
            .processingDate().equalsEdgePoint();
    Operation businessDateOp = CustomerAccountFinder
            .businessDate().equalsEdgePoint();
    return idOp.and(processingDateOp).and(businessDateOp);
}
```

The `edgePoint` operations instruct Reladomo to generate the following query. As you can see, the where clause does not include any of the temporal columns, resulting in a full dump of the database.

```
select t0.ACCOUNT_ID,t0.CUSTOMER_ID,t0.ACCOUNT_NAME,t0.ACCOUNT_TYPE,
t0.BALANCE,t0.FROM_Z,t0.THRU_Z,t0.IN_Z,t0.OUT_Z
from CUSTOMER_ACCOUNT t0
where t0.ACCOUNT_ID = 100
```

This `Operation` can now be used with the `Finder`. To make this more interestig, let's use Reladomo's `DbExtractor` which can extract the data from a table and among other things dump it to a file.

Example 11.8. tour-examples/bitemporal-bank/BitemporalChainingInAction.java

```
private String dumpCustomerAccount(int accountId) throws Exception
{
    Operation historyOperation = computeHistoryOperation(accountId);
    Path tempFile = Files.createTempFile(System.nanoTime() + "", "");
    DbExtractor dbExtractor = new DbExtractor(tempFile.toFile().getAbsolutePath(), false);
    dbExtractor.addClassToFile(CustomerAccountFinder.getFinderInstance(), historyOperation);

    String data = Files.readAllLines(tempFile).stream().collect(Collectors.joining("\n"));
    System.out.println(data);
    return data;
}
```

The `DbExtractor`'s output is not only a listing of all the rows in the table. The output is also formatted for copying and pasting into a Reladomo `MithraTestResource` file for use in tests. See `BitemporalChainingInActionTestData.txt`.

Jan 25 - Correct the missing $50

Here we need to adjust the balance for Jan 17. All we have to do is fetch the account with a business date of Jan 17 and update the balance. As before Reladomo takes care of invalidating existing rows and adding new rows.

Example 11.9. tour-examples/bitemporal-bank/BitemporalChainingInAction.java

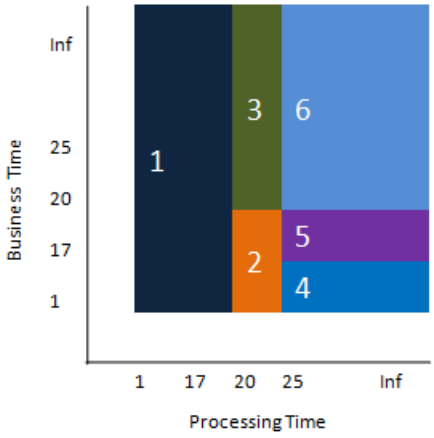
```
private void updateBalance(String businessDate, String processingDate, int accountId, int
 balance)
{
    MithraManagerProvider.getMithraManager().executeTransactionalCommand(tx -> {

        // Simulate the db change happening on a specific date - Do not do this in production
        doNotDoThisInProduction(processingDate, tx);

        Timestamp businessDateTs = DateUtils.parse(businessDate);
        Operation ts = CustomerAccountFinder.businessDate().eq(businessDateTs);
        Operation id = CustomerAccountFinder.accountId().eq(accountId);
        CustomerAccount account = CustomerAccountFinder.findOne(ts.and(id));
        account.incrementBalance(balance);
        return null;
    });
}
```

Querying a chained table

Consider the visualization from Chapter 6. Each colored rectangle corresponds to a row in the table.

Figure 11.1. Bank Account On Jan 1

Account #	Balance	FROM_Z	THRU_Z	IN_Z	OUT_Z	
1	100	Jan 1	Infinity	Jan 1	Jan 20	1
1	100	Jan 1	Jan 20	Jan 20	Jan 25	2
1	300	Jan 20	Infinity	Jan 20	Jan 25	3
1	100	Jan 1	Jan 17	Jan 25	Infinity	4
1	150	Jan 17	Jan 20	Jan 25	Infinity	5
1	350	Jan 20	Infinity	Jan 25	Infinity	6



Each of these rows can be queried by setting the business date and processing date as required. The following snippet shows querying for the account balance for business date Jan 12 but on processing date Jan 23.

Example 11.10. tour-examples/bitemporal-bank/BitemporalChainingInAction.java

```
@Test
public void balanceAsOfJan12_OnJan23()
{
    Timestamp jan12TS = DateUtils.parse("2017-01-12");
    Timestamp jan23TS = DateUtils.parse("2017-01-23");

    Operation id = CustomerAccountFinder.accountId().eq(100);
    Operation businessDate = CustomerAccountFinder.businessDate().eq(jan12TS);
    Operation processingDate = CustomerAccountFinder.processingDate().eq(jan23TS);
    CustomerAccount account = CustomerAccountFinder
                                    .findOne(id.and(businessDate).and(processingDate));
    assertEquals(100, (int)account.getBalance());
}
```

Running the above code results in Reladomo executuing the following query.

```
select t0.ACCOUNT_ID,t0.CUSTOMER_ID,t0.ACCOUNT_NAME,t0.ACCOUNT_TYPE,t0.BALANCE,
        t0.FROM_Z,t0.THRU_Z,t0.IN_Z,t0.OUT_Z
from CUSTOMER_ACCOUNT t0
where t0.ACCOUNT_ID = 12345
and t0.FROM_Z <= '2017-01-12 00:00:00.000' and t0.THRU_Z > '2017-01-12 00:00:00.000'
and t0.IN_Z <= '2017-01-23 00:00:00.000' and t0.OUT_Z > '2017-01-23 00:00:00.000'
```

The beauty of bitemporal chaining is that this query can be run at any point in time as long as the rows exist in the database. For example, five years from now, you could run this query to determine what we thought the balance was for business date Jan 12 but five years ago (Jan 23 2017).

Next steps

This section has barely scratched the surface of the what is possible with bitemporal chaining and other APIs that are relevant to chaining. Some interesting questions to consider :

- What happens if `futureExpiringRowsExist`is set to false in `MithraObject` XML ?
- What if you have to update a non-numeric column ? i.e we cannot use `incrementBalance`
- When we use `incrementBalance`, Reladomo automatically cascades the update to cover the active history. (TODO : rephrase this sentence) What if you want to increment the balance for a few specific dates ?
- What if you really really want to delete history ?

To answer these questions, consider completing one of the Reladomo Katas. You might also want to complete the REST API in the `bitemporal-bank` Maven module.