# xRoad

An Application Data Model Abstraction Framework

## 1. Version History

| Version | Date | By | Description |
|---|---|---|---|
| DRAFT | 21 July 2016 | Ben Brand | Revamp of all documentation |
| | 11 November 2017 | Ben Brand | Rebuild |
| 0.1 | October 2020 | Ben Brand | Rebuilt as xRoad and moved to maven-based application models, a complete redo of this document |
| 1.11.1 | March 2023 | Ben Brand | Complete redo.Updated to Tomcat 10 and JDK11 compliance and expanded this document to make building from scratch easier. |
| 1.11.2 | April 2023 | Ben Brand | Edits around features and the parsing of the XML file |

Nothing in this document should be taken for granted as "truth". All claims are personal and/or opinions and should be viewed as such. All software must be tested based on your acceptance criteria, and you should never trust someone writing about their software. I have done my utmost to substantiate claims, but this is IT, and all sizes do not fit all questions.

Test & verify … or die.

## 2. What is this TL;DR?

An enterprise software application aims to automate processes, and all business processes generate and use data. This framework forms a single simple methodology for managing the data that the application needs/generates and how/where it is stored.

Over the years, I have developed dozens of enterprise applications from the ground up. I have always come back to the principle that a strong foundation is a baseline for any practical application. That baseline is how an application manages the data streams it generates and needs. Typically, a lot of effort will go into the front-end GUI and processes (rightfully so), but the data processing is often treated as an afterthought. The impact is that the application's day-to-day maintenance becomes very complicated and, thus, time-consuming. Building a framework that addresses simplicity and ease of use to take care of the data-oriented tasks (and add a few side benefits) makes a lot of sense.

I have become tired of reinventing myself and building another back-end. I felt it was about time to design and develop a simple-to-use framework based on timeless design principles: abstract business rules from data structures and abstract the data structure from the requirements of the underlying database flavor.

<rant> Let me expand on one pet peeve about complexity, which is one reason I built this framework (indulge me here for a moment). There are many frameworks out there, and adding one does not make it easier. Still, everywhere I looked, I needed help finding one that was easy to understand, easy to maintain, and did not have a multi-week learning curve. Come on; most frameworks need you to buy & read a book to grok … fully? Complexity is the death of software development. I have always tried to build software that could be understood within an hour. You will notice this framework does not contain hundreds of classes and methodologies, maybe not even all the sexiest stuff that Java can bring to bear, but it works … it is quick …. and it is reliable (because you own it).</rant>

# 3. Architecture

## Overall Design

How a well-designed application should look has evolved over the years (not to be confused with how it is developed, a separate discussion). The most widely accepted and, in my view, the most logical design approach is the MVC model.

As per Wikipedia:

- A model stores data retrieved according to commands from the controller and displays it in the view.
- A view generates new output to the user based on changes in the model.
- A controller can send commands to the model to update its state (e.g., by editing a document). It can also send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document).

The model is stored in the database, the view is whatever GUI we use to represent the process, and the controller is what will be described here. The view (user interactions) initiates the controller in a model-independent fashion; the controller translates the request between the view and the model. This is called "Data Model Abstraction" here or DMA for short.

For example: "I need a list of customers sorted by name for the sales region Germany." The controller translates this high-level query (query abstraction) into a DB-specific SQL statement that will query the database and return the rows that match the query's criteria.

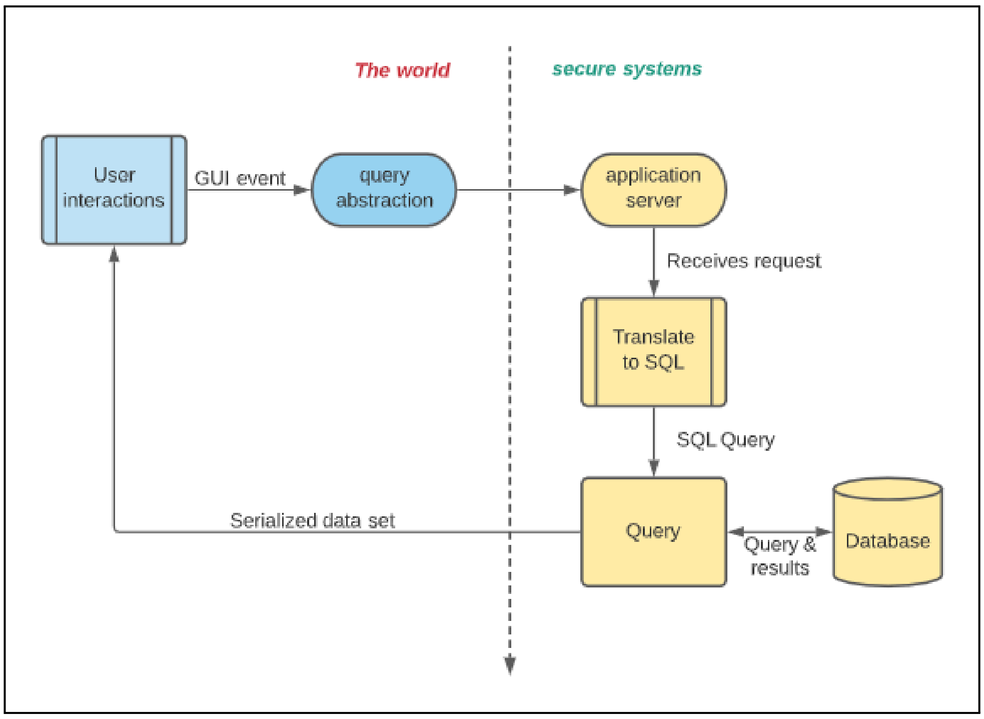
This is the most crucial component of this framework from a development and maintenance perspective. Applications are implementations of business rules and processes, creating and using data. Much too often, this is implemented so that it is challenging, if not impossible, to review and maintain without much effort. Maintaining and debugging someone else's code can be difficult; comments are often unused, and a single process can be mangled into many (sub)classes and functions. Encapsulating and abstracting data queries and updates under a logical name and removing that logic from the code to a centralized self-documenting XML makes (parallel and agile) development, testing, and maintenance much easier.

Databases form the heart of any application. They range from expensive commercial systems to lightweight open-source database management systems. The data layer depends on the application type and how the data will be managed. High levels of fluid transactional data require a different database type than static, read-only data.

The main problem is that design decisions concerning the data model taken when creating the application will probably influence it throughout its lifetime. It is challenging to extract all data-relevant structures from an existing application and port them to another one if we move to a new database vendor or into the cloud. This is almost a complete rewrite and needs to be redone each time the data storage layer changes.

It is best practice to design applications that do not have to take implementation decisions (like database vendors) into account in the design phase. A data abstraction layer (DAL) is used to achieve this.

Business processes and corresponding business rules change frequently. This will impact the application, of course, and possibly (probably) the underlying data model. This framework's architecture, based on an abstraction between business and data models (the Data Abstraction Layer, OR DAL), is set up to cater to multiple data models per object, allowing for an incremental and agile approach to business changes with minimal impact on an application.



## Technology Overview

Basic and off-the-shelf, well-documented technologies have been used for this framework:

- Java-based back-end, supplemented with widely used utility classes.
- Maven
- A multi-tier, i.e., mid-tier, is an application service (Tomcat, for example) followed by a database service (PostgreSQL, for example, but not exclusively).
- XML (DB abstraction) and web services
- JDBC
- Java-based serialization and reflection

- Role-based application and data security

## Data Model Abstraction

Querying a database is done through an SQL statement. SQL syntax is not self-documenting, is tedious to maintain, and will differ from one RDMS provider to the next. If a SQL statement is hard-coded into the GUI-view (front-end), then changing the RDBMS provides results in:

- performing a scan of **all** program code, identifying all the SQL statements
- translating these into the new SQL syntax
- testing everything again

This kind of change is a humongous and challenging process that will introduce many issues, bugs, and failures (and drive many programmers to find greener pastures). Allowing hard-coded SQL statements to infiltrate into the code base will also lead to difficult testing as the data model, capacity, application, and implementation need to be tested in a single go and cannot be separated into discrete chunks as they should.

Of course, we also need to mention SQL injection. SQL queries based on the GUI, which will be passed to the application server for execution, open the door for a hacker to intercept a URL and add their SQL to be passed on and executed. SQL injection is one of the most fundamental security flaws in any application.

This framework comes up with the following answer to this problem.

Data queries are referred to at a GUI-view level (abstracted) through a meta-name, for example, "CUSTOMER-REGION" representing a query of the CUSTOMER table sorted by REGION. The GUI-view programmer refers to the pre-defined query (CUSTOMER-REGION) and does not need to know the underlying model's structure. This will be translated in the data abstraction layer to the actual SQL statement: "SELECT * FROM CUSTOMER ORDER BY REGION". The link between meta-name and actual SQL is centralized in an XML container, allowing for swift and easy changes to the SQL syntax without recompiling the application. This translation happens close to the application server behind any firewalls preventing SQL injection; any SQL injected will be ignored as the abstraction layer will disregard it.

This method also abstracts business rules from the actual data implementation. Take, for example, the business rule "ACTIVE ORDERS". What an "active order" means to the business has been defined and implemented into the functional design via the data model. If we define a meta-query called "ACTIVE-ORDERS" and translate this via this framework into a set of queries, then we have enabled the application (GUI) builder to implement a view without necessarily understanding the underlying business or data model.

Data normalization in a data model leads to a relational data model. For example, a customer and project have a relationship based on the customer referred to in the project. You do not want to maintain the customer's name in the project table AND the customer table; this is redundant data. The project must refer back to the customer via a unique identifier. There is, in this case, a 1:n relationship between the project and the customer. The data abstraction layer can add the customer's name to any query results of the project table. This can be repeated to any depth within the data model, and all related data will be returned to the GUI within the same resulting data set via the orchestration file. This removes the need to set up chains of SQL to be executed.

In short:

- ***all SQL is maintained in one place, and any changes are reflected throughout the complete application, making for easy adoption***
- ***translation is close to the execution of the query; it is not possible to inject SQL and hack/hijack the data model***
- ***self-documenting***

## The DAL XML

The translation of the abstract query command, for example, “CUSTOMERS-BY-REGION” is orchestrated through an XML file containing all the necessary data. This also opens the possibility of maintaining multiple SQL flavors that reflect various RDBMS vendors that can be orchestrated at a runtime level. All kinds of runtime opportunities (logical versus physical instances, data security, etc.) arise based on this approach; see further what we have implemented.

## The Wrapper Class

A table in the database is built up around several columns, for example. Ignore the specific columns for the moment.

```
CREATE TABLE public.project
(
    project_id integer NOT NULL,
    description character varying COLLATE pg_catalog."default",
    capability_id integer,
    hours numeric,
    status integer,
    probability numeric,
    xrlastdate timestamp with time zone DEFAULT now(),
    xrlastuser character varying COLLATE pg_catalog."default",
    xrrowstate character varying COLLATE pg_catalog."default",
    CONSTRAINT project_pkey PRIMARY KEY (project_id)
)
```

This data definition of a table is accessible through the meta-data contained in the system tables the RDBMS uses to maintain the database. Once a new "project" has been created, this data is kept as a row in the project-table with primary key "project_id".

The wrapper class has a set of get/set methods, once for each column in the database and data from neighboring tables in the data models (related).

The wrapper class is a standard Java class corresponding to each table the framework needs to access. It matches up to the XML orchestration file for that table.

Looking at the wrapper class, you will see several lists passed to the wrapper constructor. This is done when the wrapper is built, as it pre-parses the XML orchestration file. The wrapper instance is serialized and used to communicate with the DAL. This pre-parsed instance also alleviates the need for the DAL to parse the XML orchestration class speeding things up. Each instance of the wrapper is aware of what the table looks like and can be used at any point of the framework. Note that the result is that any changes to the XML orchestration file are not reflected in any wrapper instances in flight.

```
public Project(ArrayList wrapperList) {
        if (wrapperList != null) {
            metadata = (ArrayList) wrapperList.get(0);
            columnList = (ArrayList) wrapperList.get(1);
            dataTypeList = (ArrayList) wrapperList.get(2);
            queryList = (ArrayList) wrapperList.get(3);
            helperList = (ArrayList) wrapperList.get(4);
```

The naming convention is the logical table name (case-sensitive) and is part of the common classes group under the “xrdbwrapper” package. The name of the class must correspond to the name to the name of the XML orchestration file. Java has the convention to start a class name with an uppercase letter; hence, I have stuck to that generic naming convention.

### Build a wrapper example

Table name: Project; NOT case-sensitive; SQL does not care about this, but better to stick to one naming convention.

- XML orchestration: Project.xml
- Java wrapper: Project.java

The wrapper requires the following structure:

- Constructor
    - Constructs an instance of the class that accepts an ArrayList that contains:
        - A java.list of metadata of each column
        - A java.list of the column names.
        - A java.list of the data types
        - A java.list of query
        - A java.list of helpers
    - The class is either constructed in the back end by the XRDBGet class (using a Constructor class) or via a common help class XRGetWrapper (factory). It is possible to create an instance of the class outside the framework, but not

recommended. The portal class returns a usable instance of any valid wrapper class using the XML data orchestration file as a baseline.
    - Each of these lists can be reviewed via a “get$….” class.
- set/get methods
    - Each column has a corresponding set$<logical column name> and get$<logical column name> that is used to contain and retrieve the value of that column for that row.

There is a utility class “XRBuildWrapper” that will build a stub for the wrapper class of a specific table. The XML orchestration file must be available and used as a baseline for the wrapper class. This utility class “as is” and the user needs to tweak the table's name within the code before building a template.

### DB Model Extensions

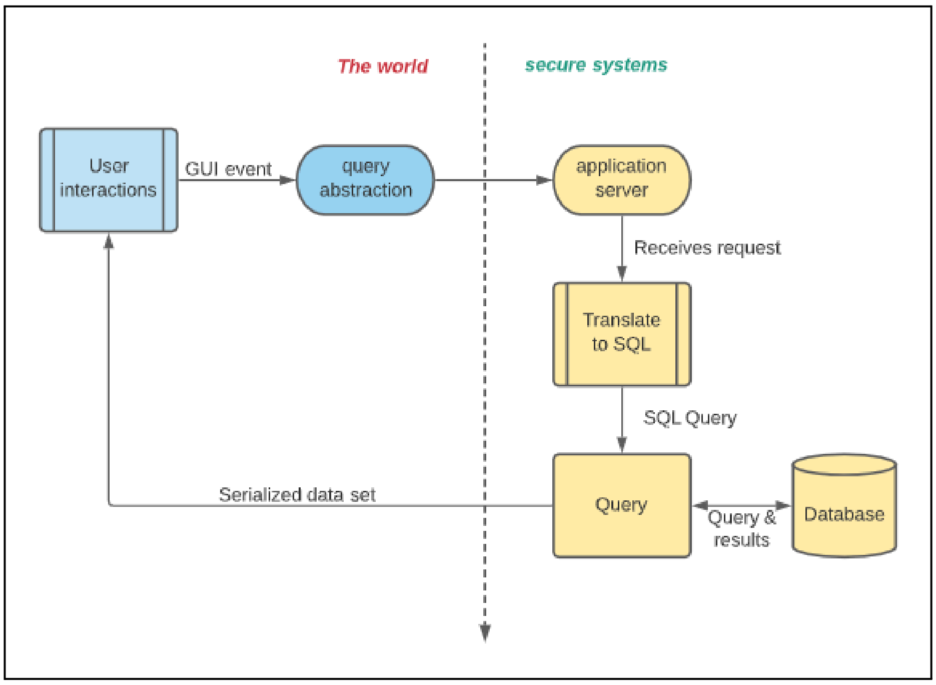
Btw, you will notice three columns with "xr ….", these are columns specific to the framework and are used to trace and track rows:

- xrlastDate
    - Part of the basic audit trail, the last date the row was created or updated.
- xrlastUser
    - Part of the basic audit trail, the last user that created or updated the row.
- xrrowState
    - Typically in a relational database model, it is challenging to eliminate rows. The application first needs to check if the referential integrity will be maintained before a row in a table can be deleted. For example, based on the above model (customer/project), if we want to remove a customer row, we first need to make sure there are no projects in the system for that customer or we will end up creating "orphans" within the data model (projects with no valid customer).
    - This orphaning can be prevented by not deleting records outright. When a delete request is received, we park the row into a predefined record state (for example, "to be deleted" or "D"). These records can then be filtered out of any data query but remain in the system until a final removal occurs.
    - A clean-up can occur regularly by having a job scan if the model's referential integrity can be guaranteed and the record can be deleted. This job will, of course, have to clean up the table in the correct order, i.e. referring-to before referring from.
    - As an additional benefit, records can be "undeleted" so long as the definitive clean-up routine has not run.

User interactions will continuously drive data across the framework. This data is not sent raw; for each row in the data set, there will be one corresponding wrapper class instance. This will be explained in more detail; in short, each row is captured into a wrapper class, and each column of data can be accessed through "get" and "set" methods. This set of wrapper classes is then serialized and passed via HTTP protocol across the framework.

# 4. The Data Flow

## Logical Overview



## Technical Overview

The framework is built around 3 tiers, each mapping to a source library.

- Frontend
    - This is where user interactions are captured, and the appropriate query abstractions are mapped to them. For example, the “onPress” event for the “Save”-button is captured, and the “SAVE-PROJECT” query-abstraction is mapped. The data set to be passed to the database is first wrapped into a class of the corresponding table and passed on to the next tier in the framework, the DTL.
- DTL (data transport layer, common files)
    - The DTL contains all classes and files, for example, wrapper classes and XML orchestration files needed to facilitate the transportation of the data sets across the framework. A portal class takes the information passed on, builds the correct URL, (de) serializes the data wrapper set, and passes it on to the DAL. The portal class also listens for serialized data sets being passed back to the front-end.
- DAL
    - The DAL is built around a set of servlets in a container (Tomcat, for example) that interacts with the database. The DAL captures data sets and query

abstractions, checks the XML orchestration files for what needs to be done for that abstraction, and executes corresponding actions on the database (CRUD). If the result is a data set, this is then encapsulated into a corresponding set of wrapper classes and passed back to the port class.

## Source libraries

The xRoad framework is contained in three "libraries" that need to be added to the application development stream.

| project | package | description |
|---|---|---|
| xroad | | The top-level, user-facing, set of classes that need to be integrated in the GUI |
| | brand.xrutils | Set of classes to help test the framework and build the wrapper and xml entities. |
| xroad-common | | The DTC, a set of classes that<br>● facilitate the bi-direction communication using HTTP with the DAL<br>● contains the wrapper and xml classes<br>The JAR built from this set of (common) classes is a dependency for both the "xroad" and "xroad-dal". |
| | brand.xrdbwrapper | ● The wrapper classes for the tables<br>● Helper classes for utilizing the wrappers |
| | brand.xrportal | ● XRPortal: The key class that facilitates the communication (see further on) between the GUI and the DAL.<br>● XRParser: common class used to parse the XML DAL files |
| | brand.xrutils | Utility classes used to build and test the wrapper and XML artifacts. |
| | brand.xrdbxml | The XML artifacts are kept here. |
| xroad-dal | | This is deployed onto the application server and contains the servlets needed for the Tomcat server that facilitate communication with the database server. |
| | brand.xrhelper | A set of helper classes used by all servlets. |
| | brand.servlets | A set of CRUD servlets to act on the data<br>XRDBDelete, delete a row from a table<br>XRDBInstert, add a row to a table<br>XRDBSet, change the contents of a row<br>XRDBGet, query the database and return a set of rows. |

| | | |
|---|---|---|
| | brand.xrutils | Some utility classes to help debug and implement the framework. |

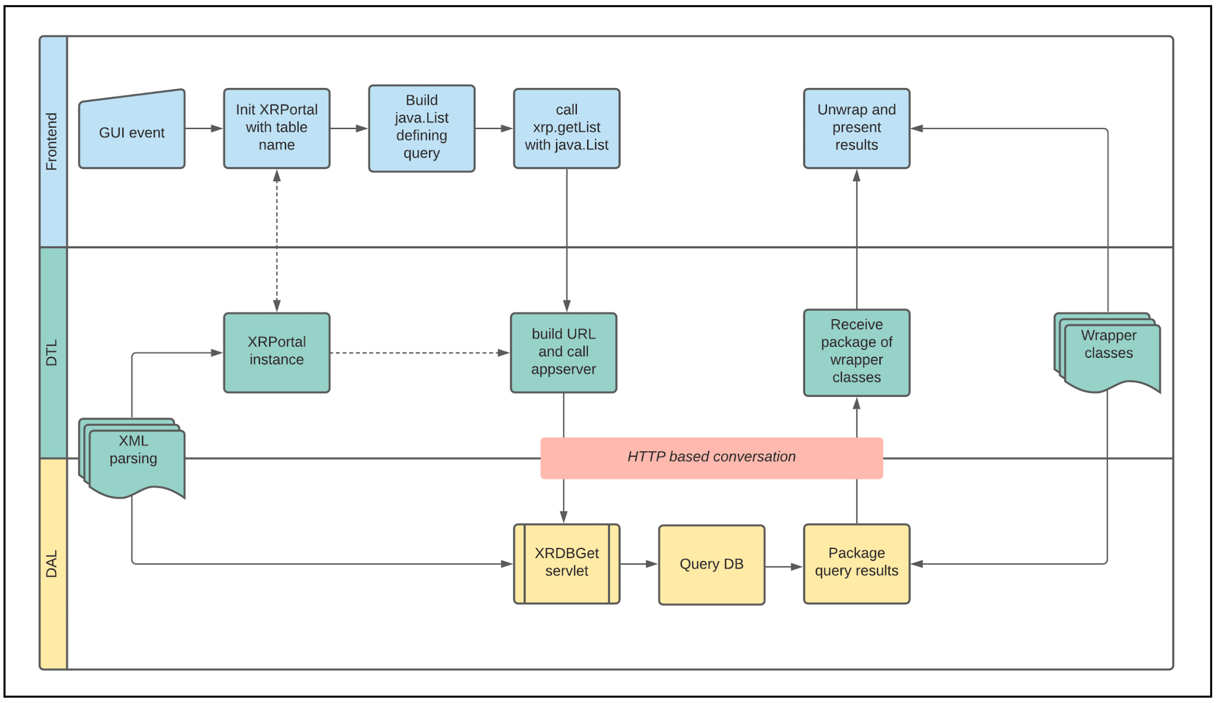
## XRPortal Class How To

The XRPortal class, which resides in the DTC model, is the main access method to and from the front end to the application server (hence the term 'portal').

| **stage / method** | usage |
|---|---|
| **create instance** | `xrp = new XRPortal(<table name>)`<br>1. note that <table name> is case sensitive and needs to reflect the method used for both the XML orchestration file and wrapper class.<br>2. the class constructor parses the XML orchestration file |
| **getData(List)** | Method used to send a query to the application server and save the results in a java.List, each item containing a wrapper class of the record returned. List.size() = 0 = no rows found for this query.<br><br>`List<Object> projectList = xrPortalProject.getList(pList);`<br><br>The method is called with a java.List containing information about the query that maps to the <parameters> section of the XML orchestration file.<br><br>For example, Projects.xml:<br><br>`    <parameters>`<br>`        <p>ACTION-ITEM</p>`<br>`        <p>ID</p>`<br>`        <p>METADATA</p>`<br>`        <p>QUERY-DEPTH</p>`<br>`        <p>FILTER</p>`<br>`    </parameters>`<br><br>1. ACTION-ITEM: the name of the query abstraction in the XML file; may not be empty<br>2. ID: The ID (key) of a unique record; 0 - n/a<br>3. METADATA: add the metadata of the table to the results? "Y" or "N"<br>4. QUERY-DEPTH: how deep we want to extract data in the relational model; 0 = top level, 1 = n:1, 2 = n:n:1 enz.<br>5. FILTER: optional filter, for example, used on a query "where <field> ge <FILTER>"<br><br>Example:<br><br>`pList.add("GET-ONE");`<br>`pList.add(2);`<br>`pList.add("N");`<br>`pList.add(0);`<br>`pList.add(0);`<br>`List<Object> projectList = xrPortalProject.getList(pList);` |

| | |
|---|---|
| **setData(wrapper, option)** | A wrapper containing the updated contents of the row to be changed. All columns in the wrapper are updated to the database, not just the changes. This means that typically a row needs to be acquired from the backend (getData method), changes logged into the wrapper and the wrapper passed to the application server, and the table updated.<br>The user option is optional; passing something on to the application server can be abused. |
| **insertRow(wrapper, option)** | The insertRow function is much the same as the setRow function. A wrapper is built, sent to the DAL, and the row is inserted. |
| **deleteRow(wrapper, option)** | The DELETE will delete a record outright or use the framework functionality to deactivate a row and leave the actual removal of the row after a referential check has been completed. |

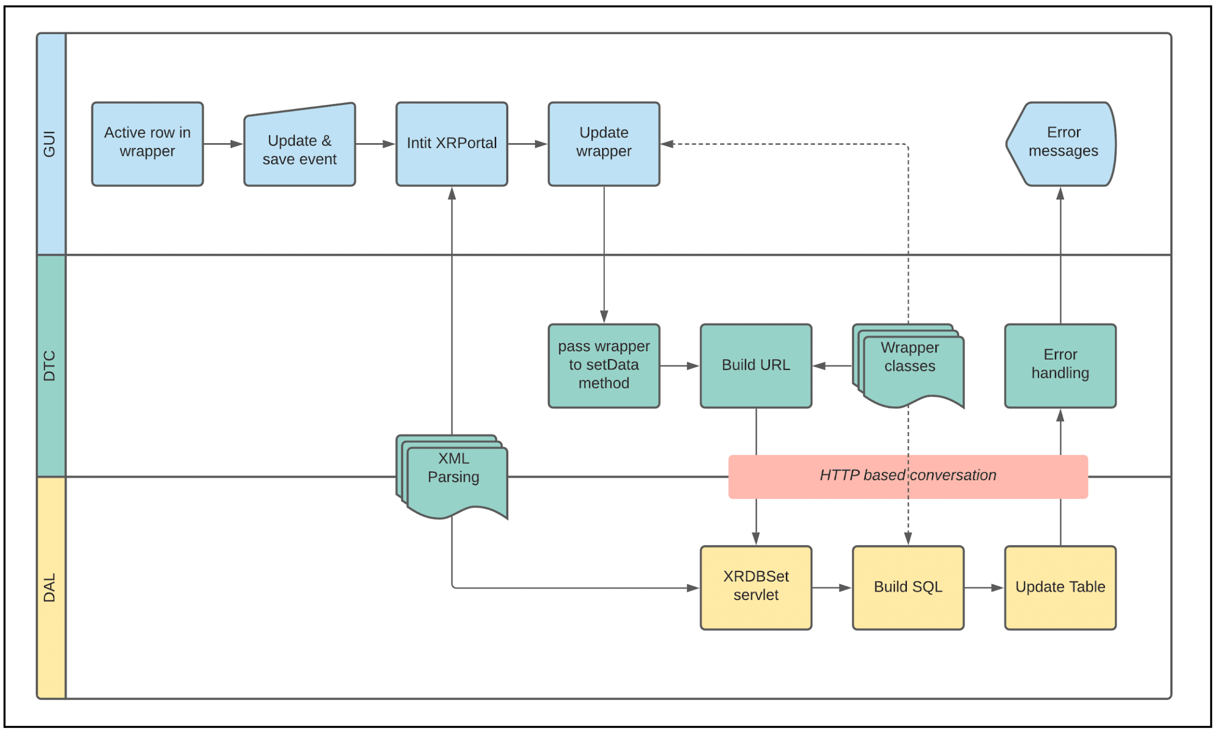
## Get (Read) Roundtrip


Frontend
DTL
DAL
GUI event
Init XRPortal with table name
Build java.List defining query
call xrp.getList with java.List
Unwrap and present results
XRPortal instance
build URL and call appserver
Receive package of wrapper classes
Wrapper classes
XML parsing
HTTP based conversation
XRDBGet servlet
Query DB
Package query results

| | |
|---|---|
| GUI Event | The user interacts with the GUI, an event is fired, and this triggers an action request. |
| Init XRPortal with table name | An instance of the XRPortal class is created, and the table name is passed on; this will analyze the XML orchestration file. |
| Build java.List defining query | The java.List is defined and the prerequisites are added to the list. See the XML of the table to understand what is needed. An incomplete List will generate an error message. |

| | |
|---|---|
| call xrp.getList with java.List | The XRPortal.getList method is called |
| build URL and call appserver | The method builds the URL for the application server servlet XRDBGet/ The method pairs the parameter names with the list's contents and added them to the URL.<br>The connection is established with the application server using the URL. |
| XRDBGet servlet | The servlet is called, the parameters passed via the URL are parsed out, and the appropriate SELECT SQL statement is built using the XML orchestration XML. |
| Query DB | The SELECT SQL statement is fired at the database, and error handling is performed. |
| Package query results | Each row in the result set is transposed into a wrapper class and added to a new list. |
| Receive resultsReceive package of wrapper classes | The query results have been wrapped up and serialized onto the HTTP link back to the calling application. The list of wrappers is deserialized and is now ready for consumption |
| Error management | The results from the database might not be as expected and an error state has been triggered. This is passed back to the GUI to inform the user. |
| Present results | The list of wrapper classes can now be traversed, unpacked (set#<> methods) and be presented to the user via the GUI. |

## Set (Write) Roundtrip

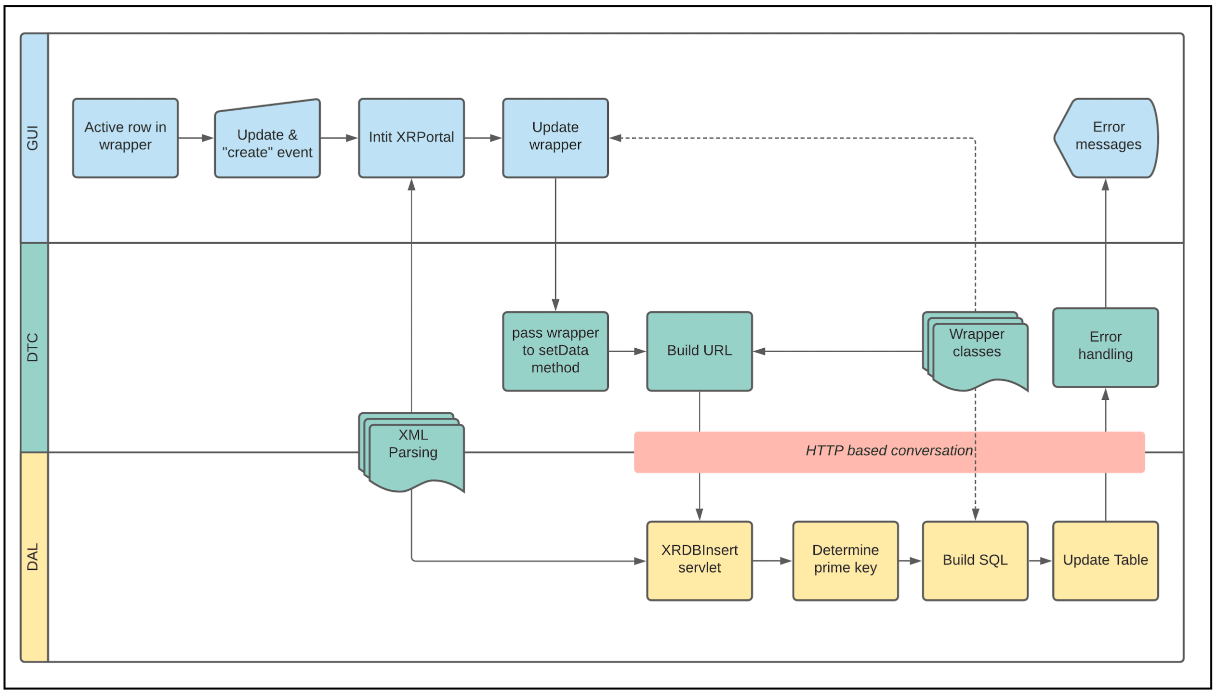


| | |
|---|---|
| Active row in wrapper | A row has been read from a table and is saved in a wrapper clause. |
| GUI Event | The data in the wrapper is updated due to an event caused by the user; the user hits a “save” button, for example. |
| Update wrapper | The updated data is written back into the wrapper (get$<> methods). The wrapper is now ready to be sent to the application server and the table to be updated.<br>NOTE: All fields in the wrapper (= columns in the table) are updated on the table! |
| Pass wrapper to setData method | The wrapper is now passed to the XRPortal.setData method |
| Build URL | The URL for the application server is built, a connection with the server is established using this URL and the wrapper is passed to the DAL via the HTTP connection established with the URL (serialization). |
| XRDBSet servlet | The XRDBSet servlet has been reached, the wrapper is now received (deserialized). |
| Build SQL | The SET SQL statement is built using the data in the wrapper and the data dict in the XML orchestration file. |

| | |
|---|---|
| Update table | The SET statement is fired on the database, error codes are trapped and sent back for notification to the user and log files. |
| Error handling | The return status of the database is sent back to be checked. Application based messages are logged locally as well. |
| Error messages | Something went wrong, the user needs to be informed. |

## Insert (Create) Roundtrip

The INSERT (creating) a new row into the database is very similar to the SET mode, other than we now need to take into account the prime key (if this is a framework auto-number).



| | |
|---|---|
| Active row in wrapper | A row has been read from a table and is saved in a wrapper clause. |
| GUI Event | The data in the wrapper is updated due to an event caused by the user; the user hits, for example, a “save” button. |
| Update wrapper | The updated data is written back into the wrapper (get$<> methods). The wrapper is now ready to be sent to the application server, and the table is to be updated.<br>NOTE: All fields in the wrapper (= columns in the table) are updated on the table! |
| Pass wrapper to insertData method | The wrapper is now passed to the XRPortal.insertData method |
| Build URL | The URL for the application server is built, a connection with the |

| | |
|---|---|
| | server is established using this URL, and the wrapper is passed to the DAL via the HTTP connection established with the URL (serialization). |
| XRDBInsert servlet | The XRDBInsert servlet has been reached; the wrapper is now received (deserialized). |
| Determine prime key | An efficient referential data model utilizes prime keys. The framework allows for several ways to determine the keys.<br>● 1 = auto sequence ID field<br>○ This results in do-nothing in the framework; the auto-sequence number is part of the database functionality.<br>● 2 = ID derived from system table<br>○ The framework takes care of determining the numbers of the prime key via a system table<br>● 3 = manual. the prime key is provided |
| Build SQL | The INSERT SQL statement is built using the data in the wrapper and the data dict in the XML orchestration file. |
| Update table | The INSERT statement is fired on the database, error codes are trapped and sent back for notification to the user and log files. |
| Error handling | The return status of the database is sent back to be checked. Application-based messages are logged locally as well.<br>There is no error handling around duplicate prime keys, which will generate a database error,  this logic is to be provided by the user. |
| Error messages | Something went wrong, the user needs to be informed. |

NOTE: The new prime key (record id) will be returned to the GUI front end. The portal gathers the returning data and adds that to an ArrayList with one member, a comma delimtted string:

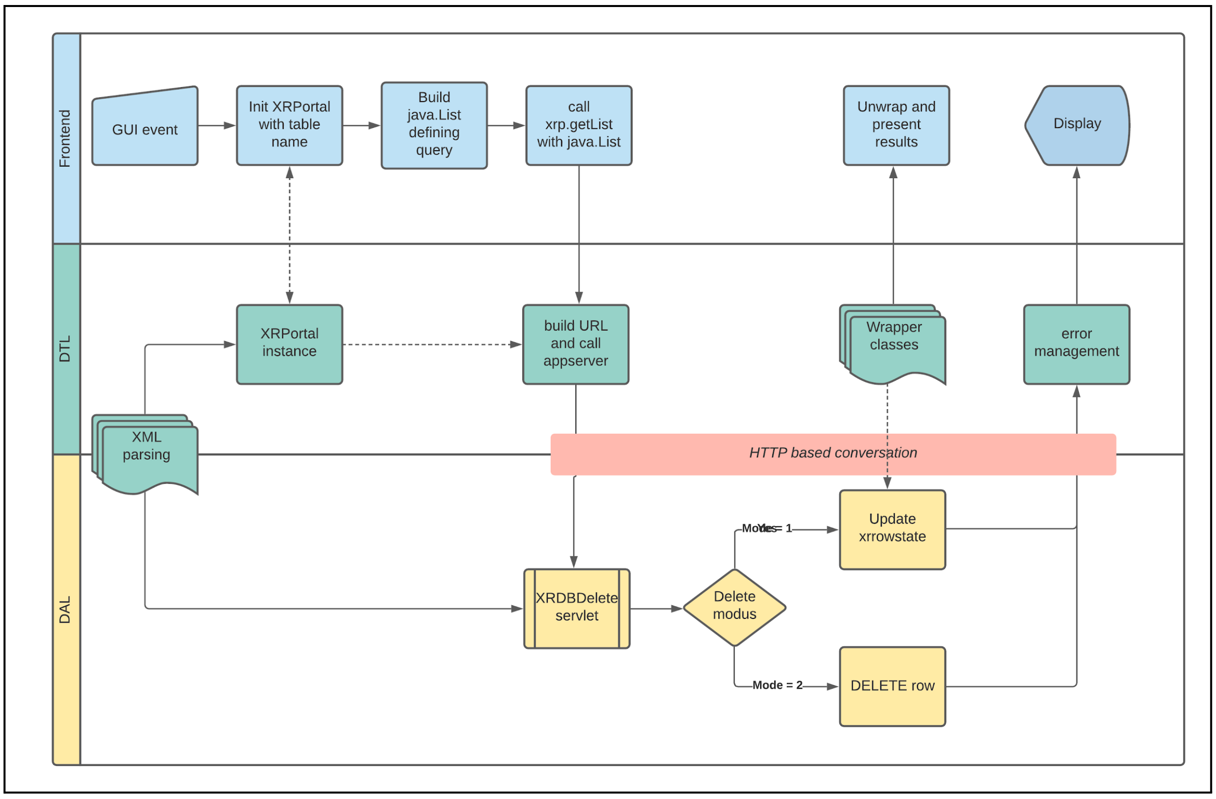
- return string
- the newly created record id
- an error message

A java “split” an then be used to retrieve the record id, “prj” is the wrapper here.

```
ArrayList returnCode = xrpProject.insertRow(prj, "");
String[] d = returnCode.get(0).toString().split(",");
projectid = new Integer(d[1]);
```

## Delete Roundtrip

The DELETE will delete a record outright or use the framework functionality to deactivate a row and leave the actual removal of the row after a referential check has been completed.



## XML Data Orchestration

### What is this?

As described in the architecture chapter, the Data Orchestration is managed via an XML file. Each table in the database has a corresponding XML file that contains all relevant information about how to use the table within the framework. The goal is that the XML file can be adapted on the fly, thus abstracting all database and table-related activities from the application code.

### Locating the XML file

The XML file must be reached from both ends of the application and thus resides in the "common" library under the folder "xrdbxml."

### The XML Schema

The XML orchestration file forms the heart of this framework and fulfills several functions. This file is read and parsed for every database-related action requested by the user (CRUD) and is table specific.

First, to help understand the XML Orchestration File, we will use an example of two tables, a “project” table and a “customer” table. Both contain master data and are related to each other; a customer can have 0:n projects. Use the following SQL commands to create these tables. NOTE: This example is meant for tables in a POSTGRESQL database but can be easily adapted for another database vendor.

```
CREATE TABLE customer ( customer_id integer NOT NULL, name character varying,
address character varying, city character varying, country character varying,
salesregion character varying, xrlastDate date, xrlastUser character varying,
xrrowState character varying, CONSTRAINT "customerID" PRIMARY KEY (customer_id) )
WITH ( OIDS=FALSE );

CREATE TABLE project ( project_id integer NOT NULL, customer_id integer,
startdate date, enddate date, description character varying, xrlastDate date,
xrlastUser character varying, xrrowState character varying, CONSTRAINT project_id
PRIMARY KEY (project_id) ) WITH ( OIDS=FALSE );
```

We will walk through the sections for the XML file using the PROJECT table as an example (as stated, EACH table used in the application must have an orchestration file).

**NOTE: The helpers class XRParser is used to parse the db-xml file. This is a hard coded rather rigid class and assumes that the layout of the db-xml follows the predefined layout as documented below 100%. Add or changing the layout of the XML file must be done for all db-xml files and this class must reflect that global change.**

The XML defines which parameters are passed to the (get) servlet. This is key; these parameters “catch” what is passed on via the URL and pair them up with the runtime parameters.

This works as follows:

1. In the GUI build an arraylist with the correct parameter values and in the same order as stated in this section.
2. The portal class is called, checking if the correct number of parameters have been passed on (prevents null exceptions on the application server). The portal class then builds a string with the parameters paired up (name & passed value) [parameter]=[value]&. This string is then used to build the URL.

## <parameters>

| Element | Parent/Child | Description |
|---|---|---|
| Naming convention | Rule | The naming convention for the XML orchestration file is [table name].xml; table name in case sensitive case and must correspond to the name of the wrapper class. |

| | | |
|---|---|---|
| | | As the XML file is needed by both the front and back end, it must be part of the common set of files and classes. NOTE: table name here is the logical table name; the physical in the database can be different. |
| &lt;root&gt; | parent | Everybody needs a root, am I right? |
| &lt;parameters/&gt; | parent | A list of parameters that is included on the URL when the servlet is called. This is the default list that every XML file should have.<br>`<parameters>`<br>`<p>ACTION-ITEM</p>`<br>`<p>ID</p>`<br>`<p>METADATA</p>`<br>`<p>QUERY-DEPTH</p>`<br>`<p>FILTER</p>`<br>`</parameters>` |
| &lt;p/&gt; | child | The first in the list must be the type of request which maps to the abstracted query name and is mapped to ACTION-ITEM parameter. Optionally a parameter can be added that maps to the parameter name QUERY-DEPTH. The query depth allows how deep into the data model a specific query needs to return data. See further on for more on this important parameter.<br>Finally, a parameter METADATA can be added; this means that ONLY the metadata of the table and columns will be returned. This is a "toggle", "Y" or "N". |

We build the ArrayList as follows:

- ArrayList testList = new ArrayList();
- testList.add("FULL-LIST"); // action-item
- testList.add("0"); // id
- testList.add("N"); // metadata
- testList.add("0"); // drill-depth
- testList.add("ABC"); // filter

This should result in an in the URL:

```
http://localhost:8080/xRoadv1engine/XRDBGet?TABLE=project&ACTION-ITEM=FULL-LI
ST&ID=0&METADATA=N&QUERY-DEPTH=0&FILTER=ABC
```

## <datalocator>

The datalocator-section provides a link between the logical and physical data models.

> The meta-data of a table (physical data model) is often built up around rather esoteric naming conventions that a data modeler thought was the bees-knees at some point but to others, makes it horribly complicated to understand. The meta-data is often impossible to change as the impact downstream to other applications and query-builders cannot be easily identified.

The framework allows us to use logical naming conventions (columns, tables, databases) that are easier to understand. For example, we can refer to the column as "CUSTOMER_ID" while the actual physical name might be "C100IID".

Defines the extended data model:

A table rarely stands alone in the data model; it forms part of a larger set of tables that form a logical hierarchy that has been normalized. In our example, the PROJECT table contains a column called CUSTOMER_ID which points to a row in the CUSTOMER table that contains the NAME column.

| **Element** | **Parent/Child** | **Description** |
|---|---|---|
| <datalocator> | parent | This section |
| <database/> | child | Part of the <datalocator> section refers to the logical database name that is used for this set of queries. Only use this if the table where the database resides is not part of the default set up. This allows for "foreign" databases. The database mentioned here must be set up in context.xml and web.xml files of the Tomcat server.<br><br>NOTE: We have a separate chapter on connecting to databases. In this framework, databases are named in a hierarchy, i.e. logical name, followed by the environment: dev, test, prod etc. This name is the overall top logical name regardless of the state of the development and rollout. The physical name is defined at the point of actual connection. |
| <table> | child | This enables the link between the logical table name and the physical. The logical name is used when referring to the table in the application and maps to the name of the XML file.<br>If empty, the logical table name is used. (physical = logical) |
| <dbcolumns/> | parent | As stated before, this framework decouples the physical data model from the logical data model. This section forms the heart by linking the logical to the physical metadata. Each column of the table must have a row in this section.<br><br>This does mean any change to the physical model must also be reflected here. This then, of course, forms a point of failure if a solid change process is not put into place. Preferably to have the logical model and physical model cataloged in a repository and then generate this section together with the required DDL for the database. So there is room for improvement. |
| <c/> | child | Each column in the table is reflected here with a line in this section.<br>Each line has 5 comma-separated fields that refer to:<br>0 = logical column name<br>1 = physical column name<br>2 = data type<br>3 = to be included in '*' selection (0 = not, 1 = yes)<br>4 = enforce unique value across the table (only used for db "write" actions) (0/1); for example, column is "name",the value here is "1", that means this name must be unique across the contents of the table; do not confuse with a prime key which determines a physical |

| | | |
|---|---|---|
| | | uniqueness for a row while this value can drive a logical level of uniqueness.<br><br>Example:<br>`<c>project_id,project_id,integer,1,0</c>`<br>`<c>customer_id,customer_id,integer,1,0</c>`<br>`<c>startdate,startdate,date,1,0</c>`<br>`<c>enddate,enddate,date,1,0</c>`<br>`<c>description,description,string,1,0</c>`<br>`<c>xrlastdate,xrlastdate,date,1,0</c>`<br>`<c>xrlastuser,xrlastuser,string,1,0</c>`<br>`<c>xrrowstate,xrrowstate,string,1,0</c>` |
| <fkcolumns/> | parent | A data model reflects not just a simple flat one table view, but also refers to tables that contain data that relates to the table. An example would be project -> customer. Each project has one and only one customer (1:1) while a customer can have multiple projects (1:n).<br>In this case, the linking column would be the column "customer_id" that is used to find the master data of the customer and then typically something like the "name" column. We could also sort through the customer master data based on the customer name and then find the corresponding projects.<br>When using transactional data like the project, we typically want corresponding master data to be shown (the project -> customer.name example we just used). The framework allows for a complete data model to be integrated into any query allowing for quick, efficient programming access. In this fashion, one data interrogation delivers data from multiple tables.<br>Not all entities of the data model might be required for a simple query. See below by "querydepth" on how this is used to fine-tune a query. |
| <f/> | child | Each line has 5 comma-separated fields denoting a row that is used from a related table.<br>0 = logical column name<br>1 = physical/actual column name<br>2 = data type<br>3 = to be included in '*' selection<br>4 = enforce unique value across table (only used for db "write" actions)<br><br>Example:<br><br>`<f>customer$name,customer$name,string,0,0</f>`<br>`<f>region$description,region$description,string,0,0</f>` |

## <keyfields>

The prime key fields of the the table, typically one field.

| **Element** | **Parent/Child** | **Description** |
|---|---|---|
| <keyfields> | parent | This section |
| <k> | child | Each prime key field gets an entry |

### <get><base>

The "get" section contains all elements used to read data from the database.

- The baseline SQL syntax
- What data set is to be returned
- How to build a valid query out of the baseline syntax

| Element | Parent/Child | Description |
|---|---|---|
| <get> | parent | The "get" section contains all elements used to read data from the database. |
| <base> | parent | The "base" section defines the queries on the primary table. |
| <[logical query name]> | parent/child | This is the name that is used to refer to the query from the calling GUI class. It is a logical name that already unfolds what the returning dataset will be.<br><br>Examples:<br>&lt;GET-ONE&gt; = get one row from the table<br>&lt;FULL-LIST&gt; = get a full list of all rows<br>&lt;PROJECT-CUSTOMER&gt; = get list of projects sorted by customer |
| <sql/> | child | The SQL command used within this query. If a parameter is used, then a '?' Is used as a placeholder. The order of the placeholder corresponds to the parameter pointer below, ie 1 = first parameter, 3 = third, etc.<br>The reference to the table can be coded to a reference variable "$table$" or refer to an actual table in the database. If the reference variable is used this is then translated to the &lt;TABLE&gt;, if that is empty then the logical name of the XML file is used.<br><br>Example:<br>`<sql>`<br>`SELECT * FROM $table$`<br>`ORDER BY project_id`<br>`</sql>`<br>or<br>`<sql>`<br>`SELECT * FROM project`<br>`ORDER BY project_id`<br>`</sql>` |
| <returnset/> | child | The DAL returns a java.Arraylist with the rows that have been found by the query. Each entry in this Arraylist is either an instance of the wrapper class wrapped around a row or an Arraylist with a subset of the columns in the row.<br>Defines how many columns are to be returned:<br>'*' = all columns ie SELECT * FROM ...<br>'n' = where n is the number of data fields to be returned.<br>if not added, defaults to '*'<br>The '*' will map all of the columns into an instance of the wrapper class and return an ArrayList of wrapper-object (HTTP based serialization in action). |

| | | |
|---|---|---|
| | | If a query returns only a subset of the columns or other data the number of returning values is given, and they are added to a unique Arrylist and added to the Arraylist sent back.<br><br>For example: SELECT SUM(revenue), SUM(tax) FROM invoices WHERE … GROUP BY …<br>Enter the value 2 which will then return for each resulting row an Arraylist with at 1 (.get(0)) = sum of the revenue and at 2 (.get(1)) = sum of the tax. |
| <parameters/> | parent/child | The where clause (if used) will have a number of parameters that need to be completed. For example "where customer_id = ?", here the comparison needs to be completed. This section allows for this, each parameter is made up of three children: datatype, pointer, value (see below) |
| <p/> | parent/child | Delineates the parameters. |
| <datatype/> | child | The data type for the parameter. |
| <pointer/> | child | Each parameter points to a parameter "?" in the SQL. This pointer tells us which one. 1 = first parameter |
| <value/> | child | Can be:<br>a value passed to the servlet (see the parameter list up top); in that case use the name of the parameter enclosed in "valueof(xxxx)"; variable typing should be covered by the programmer; Example: valueof(ID) = value passed to the servlet with the parameter "ID" (via the URL).<br>"today" which translates to the current system date of the tomcat engine<br>the value of the element is used |
| <forcecase/> | child | If 'U' then forces both sides of the query to the same case (upper), 'L' (lower) or '0' = n/a. If not used defaults to '0'.<br>The user needs to ensure data typing, using this for a non-string will generate a runtime error. |
| <prewild/><br><postwild/> | child | At times a SQL statement will be used that contains wildcards to facilitate a LIKE based query. These values will be surrounded by a wildcard. Adding these wildcards here makes sure thy are treated as SQL literals  and not part of a string. |

## <get><related><set>

The data model with just one table is rather flat. Typically, a table has a "relationship" with other tables, for example, a project table has a column "customer_id" which forms a link to a row in the "customer" table. We can state that a project has one, and only one customer (project -> customer, 1:1). The other way around a customer has zero, one, or many projects (customer -> project, 1:n).

The model can have even more depth; the customer table can refer to for example a table with regions (via a column region_id). This is then the model: project -> customer -> region.  We can say the data model has a depth of 3.

This framework has the ability to add columns from a related table to the original query results relieving the programmer from the need for multiple database reads. For example, we would want to add the customer.name and the region.description to a certain query.

There might be use cases where diving down into the data model might not be needed, it would only slow up the read. The parameter “querydepth” can be used to define how deep into the data model a certain use case needs to return data.

NOTE: Per SQL only one column can be returned. If multiple columns need to be added then multiple SQL’s must be set up in the XML file.

NOTE: The SQL statement are built and executed sequentially as set up in the SQL. The “fkcolumns” section must reflect the same order.

| **Element** | **Parent/Child** | **Description** |
|---|---|---|
| <related> | parent | The section header |
| <r/> | child/parent | Just a numbering, groups each related table |
| <tablename/> | child | The logical name of the related table. This is used to determine which wrapper object is to be used (serialization).<br><br>Example:<br>`<tablename>customer</tablename>` |
| <querydepth/> | child | This attribute must exceed the overall parameter to be part of the return set; allows for tuning of queries and update response times.<br>Parameter value:<br>0 = included all<br>6 = the DRILL-DOWN parameter must be 7+ for this to be included.<br>99 = skip all |
| <sql/> | child | The SQL statement used to retrieve the data.<br><br>Example:<br><br>`<sql>`<br>`SELECT customer.name FROM customer`<br>`WHERE customer.customer_id = ?`<br>`</sql>` |
| <setters/> | parent/child | This is a list of parameter that are used to “set up” the SQL statement, drives the “where clause” of the SQL. |
| <s/> | parent/child | 1:n childeren |
| <setType/> | child | The data type to be set (“string” in the example) |
| <setPointer/> | child | This is for which parameter (“?”) in the where clause. |
| <setValue/> | child | Can be:<br>valueof(parameter): set with value from passed parameter |

| | | |
|---|---|---|
| | | parent(column): set with value in column in parent; column must be part of result set<br>sysvalue(): system based value<br>o today = date stamp of system date (tomcat container) |
| <getters/> | parent/child | Once the SQL for the related rows has been executed extract the data from the resultset. |
| <g/> | parent/child | For each column that needs to be retrieved a “g”. This must match the SQL statement set up above, the framework does not check is the parameters/columns matches the syntax. |
| <getType/> | child | The data type of the column retrieved |
| <getPointer/> | child | The value of which of the column in order of the select-clause to place here. 0 = first column, 1 = second column, 2 ... |
| <getColumn/> | child | Points to the value of a specific column number, ie 2 = the second column. If getPoint and getColumn are used then getPointer prevails. |
| <getDefault> | child | If NO rows could be found then this is the default value of the column, for example, “unknown” or “ERROR” |

### <create>

This section drives the behavior when adding rows to the table.

| **Element** | **Parent/Child** | **Description** |
|---|---|---|
| <create/> | parent | Section header |
| <primekeymode/> | child | 1 = auto sequence ID field<br>2 = ID derived from system table<br>3 = manual |
| <primekeytable/> | | Mode = 2, name of the system table containing the ID values. If Not provided “systemidvars” used. |
| <primekeycolumn/> | | Name of the column that contains the prime key, only relevant for primekeymode 1 or 2, if not provided then “ID” assumed. |

### <delete>

This section drives the behavior when deleting rows from the table.

| **Element** | **Parent/Child** | **Description** |
|---|---|---|
| <delete/> | parent | Section header |

| | | |
|---|---|---|
| <deletemode/> | child | 1 = use a field to mark the row as "inactive, to be deleted"<br>2 = delete row outright. NOTE: this could damage the relational model. |
| <deletecolumn/> | | If deletemode = 1, this column marks which column is used to mark the status of a row. |

**Setting it Up**

The XML file needs to reside in the common libraries as it is used by both the front and back ends.

It can be rather tedious to build the orchestration file (you need 1 per table!), and each programmer will have his/her shortcuts to do this. The framework includes a utility class "XRBuildXML" that builds a stub XML that can be used as a basis.

The utility class harbors a bit of "intelligence" as it interprets the data model following the conventions I use for a data model. These conventions are:

1. The prime key of any table is a sequence number with the naming convention "<table name>_id". If the utility class encounters this combination, it treats this column as the prime key. For example, the "project" table has the column "project_id"; the utility class treats this as the prime key and makes some assumptions based on this.
2. If a column ending with "_id" that does not contain the table name is found then it treats this as a foreign key to a related table. The project table has a column "customer_id", the assumption is that there is a relation between the project and customer, and the customer_id column is used.

Check the output for what these assumptions lead to.

## XML Parser Class

The helper class XRParser has  the following methods:

| Method | Input | Returns | Description |
|---|---|---|---|
| getParameters | | Arraylist | returns a list containing the <parameters/> |
| getDataBase | | String | Returns a string representing the database used. At this version this is not used.<br><br>**<datalocator/> ↪ <database>** |
| getTableName | | String | Returns a string representing the physical database name.<br><br>**`<datalocator/> ↪ <table>`** |
| getDBColumns | | ArrayList | Return a list of Arrays, each representing the DB columns<br>(0) = logical columns names<br>(1) = physical columns names |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | (2) = data types<br>(3) = returned in query * (1 = true, 0 = false) [NOT TESTED]<br>(4) = unique row (1 = true, 0 = false) [NOT TESTED}<br><br>`<datalocator/> ↳ <dbcolumns>` |
| getFKColumns | | ArrayList | | | Same as getDBCOlumns, but now for any columns from using a related table.<br><br>`<datalocator/> ↳ <fkcolumns>` |
| getBase (get) | | Hashmap | | | This returns a hasmap that contains all provided elements for each logical query. The GET data portion of the DAL.<br><br>`<get/> ↳ <base>`<br><br>The key for the map is the name of the query found via the first element of the list.<br><br>The following elements are maintained in a list that is the data portion of the hashmap entry. |
| | | | 0 | | A string representing what data the query returns, either all (*) or a number of fields (integer); defaults to all.<br><br>`<get/> ↳ <base> ↳ <returnset/>` |
| | | | 1 | | A trimmed string representing the SQL statement, if not found throw an error.<br><br>`<get/> ↳ <base> ↳ <sql/>` |
| | | | 2 | | An arraylist of parameters passed used in the query, to resolved to addres the '?' in the query.<br><br>`<get/> ↳ <base> ↳ <parameters/>` |
| | | | | 0 | String; representing the datatype<br><br>`<datatype>` |
| | | | | 1 | String; resolve to this parameter, 1 = first parameter<br><br>`<pointer>` |
| | | | | 2 | String; the value of the parameter<br><br>`<value>` |
| | | | | 3 | String; is the case of the value is to forces to lower case, 0 = false. optional<br><br>`<forcecase>` |
| | | | | 4 | String; wildcard that can be added to the value-string at the start; optional<br><br>`<prewild>` |
| | | | | 5 | String; wildcard that can be added to the value-string at the end of the string; optional<br><br>`<postwild>` |

| | | | | | |
|---|---|---|---|---|---|
| | | | 3 | | Returns and String if the query should drill down to the related tables. This is used as not all queries may return the prime key of the related tables, which would result in a SQL error during runtime.<br><br>Defaults to “1” = yes<br><br>**`<get/> ↳ <base> ↳ <drilldown/>`** |
| getRelations | | ArrayList | The db-xml notes and “related” table that can also be reads to simplify querying inot the related mode. These are noted in an Arraylist where each element is a arraylist representing one related table.<br><br>If no related tables then an empty ArrayList is returned.<br><br>**`<get/> ↳ <related>`** | | |
| | | | 0 | | A trimmed string representation of the SQL to query the related table.<br><br>**`<sql>`** |
| | | | 1 | | An ArrayList of the setters parameters |
| | | | | 0 | String; the data type<br><br>**`<setType>`** |
| | | | | | String, the sequential number of the parameter (?) in the SQL statement.<br><br>**`<setPointer>`** |
| | | | | | The value<br><br>**`<setValue>`** |
| | | | 2 | | An arraylist of the getter parameters |
| | | | | 0 | String; the data type<br><br>**`<getType>`** |
| | | | | | String, the sequential number of the parameter (?) in the SQL statement.<br><br>**`<getPointer>`** |
| | | | | | The column(s) returned in the data set, comma delimitted<br><br>**`<getColumn>`** |
| | | | | | The default value to be returned, basically an error message of some sorts.<br><br>**`<getDefault>`** |
| | | | 3 | | The querydepth; see description in the manual<br><br>**`<querydepth/>`** |
| | | | 4 | | The name of the table. |

| | | | | |
|---|---|---|---|---|
| | | | | **`<tablename/>`** |
| getKeyFields | | Arraylist | Returns and ArrayL:ist for the prime key columns for this table.<br><br>**`<keyfields/>`** | |
| getCreateMode | | ArrayList | Returns an ArrayList of how the prime key when adding a new row is defined. | |
| | | | 0 | The prime key mode, integer<br><br>**`<primekeymode/>`** |
| | | | 1 | If autonumbered, the table where these counters reside, string<br><br>**`<primekeytable/>`** |
| | | | 2 | What is the primekeycolumn indicator<br><br>**`<primekeycolumn/>`** |

## Setting up a development environment

There is no proscriptive IDE to be used. The current baseline framework resides on GitHub in three repos and are NetBeans projects. These can easily be ported to any other IDE.

| | | |
|---|---|---|
| xroad-ui | A stump project that contains some classes to demo/test the framework from the client side. | |
| | `XRTestDALConnection` | Test if a roundtrip can be made from the UI to the DAL. |
| | `XRSerializeTest` | This will test if the four DAL servlets work. |
| | `XRXMLTest` | Test if the XML DAL file has been correctly formed. |
| | `XRBuildAllCommon` | Building the backed files (serialization classes and XML files) can be a bit tedious. This utility class will build a set of stub classes and XML files that can be used as a baseline. |
| xroad-lib | `XRGetWrapper` | ● called with a table name<br>● parses the XML wrapper file -> XRParser<br>● returns an object containing the wrapper class of the corresponding table |
| | `XRGetProp` | Get system-wide environment vars and return them as a variable |
| | `XRParser` | Set of methods to analyze the XML file |
| | `XRPortal` | Set of methods to query the DAL and return results. |
| | `XRBuildDefaultProps` | Build a default config.properties |
| | `XRBuildWrapper` | A utility class that can be called to build a wrapper and XML stump for a table. |
| | `XRSetSystemProps` | Set environment-wide properties |
| xroad-dal | `XRDBConnect` | Set up a connection to the database |
| | `XRDBGetID` | A master file containing the prime key values of each table can be set up and maintained here. The methods getNextVal and saveNextVal are then used. If this method is used is defined in the tables XML DAL file. |
| | `XRDBDelete, XRDBGet, XRDBInsert, XRDMSet` | The servlets to update rows in the database. |
| | `XRBuildXML` | A servlet that can be called to build a default XML DAL stub |

| | | |
|---|---|---|
| | `XRTestDBConnection` | The if a connection can be made with the database by the backend. IT requires a test table to be set up in the target database. |
| | | |

## config.properties

All solutions require some ability to adapt to the environment. Hard coding would lock into one environment. A properties file allows the ability to adapt the solution to the environment on the fly.

There is a config.properties file that sets several defaults and can be expanded as needed.

The properties file resides in the common library's src/main/resources.

the properties are read and copied into system-wide system variables.

```
#Fri Mar 10 16:22:26 CET 2023
wrapperClass=com.twobees.xrdbwrapper.
dbRootName=cman
releaseLevel=0.1
runLevel=99
xmlDB=com/twobees/xrdbxml/
dalURL=http\://localhost\:8080/cman-dal/
releaseCycle=DEV
```

## Let's Build It

| | |
|---|---|
| 1. | Clone the three repos |
| 2. | Build a comparable environment for your application, 3 repos, and commit these to your GitHub. Note that the DAL must be run on an application server, for example, Tomcat. This impacts how you set up a project, it must build into an applicable war file and be exported to the test server.<br>The baseline repos are set up where the UI and DAL projects run on an application server, and the LIB has the common classes between the application server-based project. |
| 3. | Copy the classes that you will be using. The utilities are optional, but the DAL servlets and corresponding classes must be set up. |
| 4. | from LIB edit and run the class `XRBuildDefaultProps` to reflect your setup and then check the output. Move the config.properties file to the root folder of the UI. |
| 5. | Open XRBuildAllCommon from LIB<br>Edit the definition of *commonLibXML* and *commonLibWrapper* to reflect your environment.<br>Reflect a current table in the database. |
| 6. | Update the web.xml and context.xml files; initially, they are empty on the UI side. |
| 7. | Package the common library, make sure it is part of the other libraries (UI and DAL) |
| 8. | Build a wrapper and XML for an existing table (use the utility class **XRBuildAllCommon** to help you along) and package these |
| 9. | Package and publish the UI and DAL projects to your test environment. |
| 10. | Test the connection to the database using **XRTestDALConnection** |
| 11. | Build and use a test class to test all 4 database functions (see example test class). |

### DAL, web.xml

```
 <?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
    version="6.0"
>
    <servlet>
        <servlet-name>XRTestDBConnection</servlet-name>
        <servlet-class>com.twobees.xrutils.XRTestDBConnection</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>XRDBGet</servlet-name>
        <servlet-class>com.twobees.xrservlets.XRDBGet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>XRDBSet</servlet-name>
        <servlet-class>com.twobees.xrservlets.XRDBSet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>XRDBInsert</servlet-name>
        <servlet-class>com.twobees.xrservlets.XRDBInsert</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>XRDBDelete</servlet-name>
        <servlet-class>com.twobees.xrservlets.XRDBDelete</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>XRBuildXML</servlet-name>
        <servlet-class>com.twobees.xrutils.XRBuildXML</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>XRTestDBConnection</servlet-name>
        <url-pattern>/XRTestDBConnection</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>XRDBGet</servlet-name>
        <url-pattern>/XRDBGet</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>XRDBSet</servlet-name>
        <url-pattern>/XRDBSet</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>XRDBInsert</servlet-name>
        <url-pattern>/XRDBInsert</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>XRDBDelete</servlet-name>
        <url-pattern>/XRDBDelete</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>XRBuildXML</servlet-name>
        <url-pattern>/XRBuildXML</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

## DAL, context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/capman-dal">
    <WatchedResource>WEB-INF/web.xml</WatchedResource>
    <Resource
        name="cmanDEV"
        username="******"
        password="********"
        url="jdbc:postgresql://127.0.0.1:5432/cManDEV"
        auth="Container"
        driverClassName="org.postgresql.Driver"
        logAbandoned="true"
        maxActive="40"
        maxIdle="4"
        validationQuery="SELECT * FROM public.xrsyskey"
        removeAbandoned="true"
        removeAbandonedTimeout="60"
        type="javax.sql.DataSource"/>
</Context>
```

## Testing your setup

Here is the code for a Java class to test the set up of your system as well as the SQL documented in your XML file for this table (= Role)

```
package dbtest;


/*
 * Copyright (C) 2013-2023 Ben Brand
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
 */
import com.twobees.xrdbwrapper.Role;
import com.twobees.xrdbwrapper.XRGetWrapper;
import com.twobees.xrportal.XRPortal;
import com.twobees.xrutils.XRSetSystemProps;
import java.io.IOException;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;

/**
 *
 * @author brand
 *
 */
public class DBTest_role {

    public static void main(String[] args) throws IOException {
        //
        // get the propperties derived ofmr the config.properties file in the
        // common librabry. If these have not been declered yet then the
        // proporties are read and the system wide properties are declared.
        //
        XRSetSystemProps sp = new XRSetSystemProps();
        Properties p = System.getProperties();
        //
        //
        XRPortal xrp = new XRPortal("Role");
        //
        // The test methode reflects one of the four possible table actions
        // (and serveltes): get, set, insert, delete. Change this to test
        // each action.
        String testMethod = "get"; // what are we testing?
        //
        switch (testMethod) {
            case "get": { // read one or more row from the table
                System.out.println("TEST ROLE GET\n");
                ArrayList testList = new ArrayList();
                //
                // the action-item relfect which AQL is to be tested
                // change to rotate through each one in your XML file
                //
                testList.add("ROLE-LIST"); // action-item
                testList.add("1"); // id or 0
                testList.add("N"); // metadata
                testList.add("0"); // query-depth, n/a
                testList.add("");  // filter, n/a
                //
                // Do it.
                ArrayList rowList = xrp.getData(testList);
                if (rowList.isEmpty()) {
                    System.out.println("No result found for ROLE / get");
                } else {
                    for (int i = 0; i < rowList.size(); ++i) {
                        Role role = (Role) rowList.get(i);
                        StringBuffer dataDump = role.dumpData();
                        System.out.println(dataDump);
                    }
                }
                break;
            }
            case "set": { // read a row and update a field; reread the row
                System.out.println("TEST ROLE SET\n");
                ArrayList testList = new ArrayList();
                testList.add("ROLE"); // action-item
                testList.add("1"); // id
                testList.add("N"); // metadata
                testList.add("0"); // query-depth, n/a
                testList.add(""); // filter, n/a
                //
                //
                ArrayList rowList = xrp.getData(testList);
                if (rowList.isEmpty()) {
                    System.out.println("No result found for ROLE / set; initial
read");
                } else {
                    Role role = new Role();
                    role = (Role) rowList.get(0);
                    role.set$description("updated via test " +
LocalDate.now().format(DateTimeFormatter.ISO_DATE));
                    ArrayList returnCode = xrp.setData(role, "SET");
                    System.out.println("Update. return code " +
returnCode.toString());
                }
                break;
            }
            case "insert": {
                System.out.println("TEST ROLE INSERT\n");
                XRGetWrapper xrg = new XRGetWrapper();
                Role role = (Role) xrg.getWrapper("Role");
                role.initRow();
                role.set$description("test record insert " +
LocalDate.now().format(DateTimeFormatter.ISO_DATE));
                role.set$xrrowstate("T");
                ArrayList returnCode = xrp.insertRow(role, "");
                System.out.println("Insert. return code " +
returnCode.toString());
                break;
            }
            case "delete": {
                //
                // It seems a bit strange this one: why do we need to build a
                // wrapper by first reading the row before passing it on to
                // portal. If we construct a wrapper class and then pass
                // that on directly to the portal we get a nonseriazable
exception.
                // That does not make too much sense.
                //
                System.out.println("TEST ROLE DELETE\n");
                ArrayList pList = new ArrayList();
                //
                // get a row into a wrapper
                pList.add("ROLE");
                pList.add(1);
                pList.add("N");
                pList.add(0);
                pList.add(0);
                List<Object> roleList = xrp.getData(pList);
                XRGetWrapper xrg = new XRGetWrapper();
                Role role = (Role) xrg.getWrapper("Role");
                //
                // cast the  result into the wrapper
                role = (Role) roleList.get(0);
                //
                // delete it
                ArrayList returnCode = null;
                returnCode = xrp.deleteRow(role, "");
                System.out.println("Delete. return code " +
returnCode.toString());
                break;
            }
        }
        }
}
```