Java VAMDC-TAP Node software

Document Information

Editors: M. Doronin
Authors: M. Doronin
Contributors: Yaye Awa Ba
Type of document: standards documentation
Status: draft
Distribution: public
Work package: WP6
Version: 12.07r2
Date: 28/10/2015
Document code: 

Abstract: This document is a guide for the installation and use of the Java VAMDC-TAP Node software implementation.
Version History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Modified By</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0.1</td>
<td>31/10/2011</td>
<td>M.Doronin</td>
<td>first draft</td>
</tr>
<tr>
<td>V12.07</td>
<td>29/08/2012</td>
<td>M.Doronin</td>
<td>Release documentation for 12.07 Java node software</td>
</tr>
<tr>
<td>V12.07r2</td>
<td>28/10/2015</td>
<td>M.Doronin</td>
<td>Updates for 12.07r2 release of the Java Node Software</td>
</tr>
</tbody>
</table>

Disclaimer

The information in this document is subject to change without notice. Company or product names mentioned in this document may be trademarks or registered trademarks of their respective companies.

License

This document is protected by the Creative Commons license CC BY-ND 4.0.

For the license detail, please visit: http://creativecommons.org/licenses/by-nd/4.0/

You are free to:

• **Share** - copy and redistribute the material in any medium or format for any purpose, even commercially.

Under the following terms:

• **Attribution** - You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

• **NoDerivatives** - If you remix, transform, or build upon the material, you may not distribute the modified material.

All rights reserved

The document is proprietary of the VAMDC consortium members.

This document reflects only the authors’ view. The European Community is not liable for any use that may be made of the information contained herein.

Acknowledgements

VAMDC is funded under the “Combination of Collaborative Projects and Coordination and Support Actions” Funding Scheme of The Seventh Framework Program. Call topic: INFRA-2008-1.2.2 Scientific Data Infrastructure. Grant Agreement number: 239108.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Preface</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Web-service description</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Web-service endpoints</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>3.1</td>
<td>Used software</td>
<td>3</td>
</tr>
<tr>
<td>3.2</td>
<td>VAMDC common components</td>
<td>3</td>
</tr>
<tr>
<td>3.3</td>
<td>Node-specific components</td>
<td>4</td>
</tr>
<tr>
<td>3.4</td>
<td>Comparison with the python/django node software</td>
<td>4</td>
</tr>
<tr>
<td>3.5</td>
<td>Node implementation</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Node architecture</td>
<td>6</td>
</tr>
<tr>
<td>4.1</td>
<td>Request processing</td>
<td>6</td>
</tr>
<tr>
<td>4.2</td>
<td>Plugin and Framework Interfaces</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>Database plugin testing</td>
<td>10</td>
</tr>
<tr>
<td>5.1</td>
<td>Eclipse project setup / Screenshots</td>
<td>11</td>
</tr>
<tr>
<td>5.2</td>
<td>Maven Integration</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>Database Object Model</td>
<td>15</td>
</tr>
<tr>
<td>6.1</td>
<td>Step-by-step guide</td>
<td>15</td>
</tr>
<tr>
<td>6.2</td>
<td>Notable Cayenne features used</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>XSAMS tree building</td>
<td>24</td>
</tr>
<tr>
<td>7.1</td>
<td>Example constructor class</td>
<td>24</td>
</tr>
<tr>
<td>7.2</td>
<td>Attaching objects to the XSAMS Document tree</td>
<td>26</td>
</tr>
<tr>
<td>7.3</td>
<td>Identifiers generation</td>
<td>26</td>
</tr>
<tr>
<td>7.4</td>
<td>XSAMS JAXB convenience extensions</td>
<td>27</td>
</tr>
<tr>
<td>7.5</td>
<td>Case-By-Case generic builders</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>VSS query parsing and mapping</td>
<td>29</td>
</tr>
<tr>
<td>8.1</td>
<td>Query keywords tree</td>
<td>29</td>
</tr>
<tr>
<td>8.2</td>
<td>QueryParser library interface</td>
<td>30</td>
</tr>
<tr>
<td>8.3</td>
<td>Filtering logic</td>
<td>31</td>
</tr>
<tr>
<td>8.4</td>
<td>Query mapping scenarios</td>
<td>31</td>
</tr>
<tr>
<td>8.5</td>
<td>Query Mapping Library</td>
<td>33</td>
</tr>
<tr>
<td>9</td>
<td>Query metrics support</td>
<td>34</td>
</tr>
<tr>
<td>9.1</td>
<td>getMetrics(...) method</td>
<td>34</td>
</tr>
<tr>
<td>9.2</td>
<td>Sample implementation</td>
<td>35</td>
</tr>
<tr>
<td>10</td>
<td>VAMDC-TAP node deployment</td>
<td>36</td>
</tr>
<tr>
<td>10.1</td>
<td>Install a Java application server</td>
<td>36</td>
</tr>
</tbody>
</table>
This document describes the implementation and installation of a VAMDC-TAP node using the Java Node Software. It is organized to follow the node development path. Each chapter corresponds to an individual development task.

In the Introduction, a brief description of the components employed in the java node software implementation is given.

Node architecture chapter presents the architecture of the Java Node Software and gives the details on the interaction interfaces between the node software parts.

Setting up the development environment with the TAPValidator program is described in the Database plugin testing chapter.

The first development task is to access the data in the database. Apache Cayenne library is used to perform this task. The process is briefly described in the chapter Database Object Model. Once the access to the data is established, one may proceed to building the XSAMS elements from the database objects, as described in the chapter XSAMS tree building. The last development task is to define the mapping between the incoming VSS2 queries and the internal database queries. The chapter VSS query parsing and mapping describes that mapping in detail.

By this moment, node plugin should be capable of producing the XSAMS documents in response to incoming queries. Final development task is to implement special queries used to estimate if the database contains data for a given VSS2 request. It is described in the Query metrics support chapter.

The last chapter and last task is to install the node web service (VAMDC-TAP node deployment).
WEB-SERVICE DESCRIPTION

Java implementation of VAMDC-TAP node software is a web-service application implementing a RESTful interface according to VAMDC-TAP standard specification.

It is accessible using the HTTP protocol, with URLs like http://node.name.tld/vamdc-tap/12_07/VOSI/capabilities or http://node.name.tld/vamdc-tap/12_07/TAP/sync?query=select%20species&lang=VSS2&format=XSAMS. Here http://node.name.tld/vamdc-tap/12_07/ is the node base url, /VOSI/capabilities and /TAP/sync are endpoint addresses, and ?query=select%20species&lang=VSS2&format=XSAMS are HTTP request parameters.

2.1 Web-service endpoints

Java implementation of VAMDC-TAP node software implements the following webservice endpoints:

- /VOSI/capabilities Virtual Observatory Support Interface - service capabilities endpoint. Reports the service capabilities and available IVOA and VAMDC-TAP addresses, as well as their mirrors. Accept no parameters.

- /VOSI/availability Virtual Observatory Support Interface - service availability endpoint. Reports the service availability and internal checks status. Accepts no parameters.

- /TAP/sync VAMDC-TAP synchronous query interface, accepting HEAD and GET requests. Mandatory parameters are query, lang and request. Refer to the VAMDC-TAP standard documentation for more details.

- /config Returns the web-service configuration options

- /clear_cache Forces the purge of Apache Cayenne object caches. May be called on the database update to force the node software to reload the data from the database.
INTRODUCTION

Java implementation of the VAMDC-TAP node software was created in parallel with the Python version to pursue several objectives:

- Insure that standards are complete and contain no implementation-specific elements
- Give a choice of an implementation to node maintainers
- Employ the schema-based DOM XML generator to ensure the document validity. It presents an alternative approach to the keywords dictionary as used in Python/Django node software - see below the Differences section

3.1 Used software

The following open-source libraries and components are used:

- JAXB Reference Implementation (RI) for XML schema mapping and XSAMS document output
- Apache Cayenne ORM framework for database access
- MySQL database (any relational database can be used via Apache Cayenne)
- ANTLR generated query parser with slightly modified SQLite syntax to transform the incoming query into the tree of objects
- Oracle Jersey JAX-RS implementation to support the VAMDC-TAP webservice interfaces
- Apache Tomcat application server to run the Java VAMDC-TAP web application

3.2 VAMDC common components

Additional libraries were developed within the VAMDC project are part of Java VAMDC-TAP node software.

- Dictionaries of standard keywords, used in a query;
- Query parsing library, providing object-oriented view on a query string;
- Query mapping library, providing basic support for mapping of incoming queries to the node database queries using Apache Cayenne objects.
- XSAMS helper library, providing convenience methods for output XML generator implementation;
- Web application, .war archive integrating all libraries
- TAPValidator, VAMDC-TAP service validation tool that may be used for node plugin testing on all phases of development.
3.3 Node-specific components

In Java node software each node installation requires creating two libraries:

1. **Database Access Objects using Apache Cayenne.** This task is well described in the Apache Cayenne documentation \[CAYDOC\]. A graphical tool is provided allowing to define the database structure and relations. Java classes corresponding to table entities are generated using this tool.

2. **Node plugin,** A piece of software that implements the node plugin interfaces and integrates with Java VAMDC Node Software application. Node plugin is responsible for a query translation into the database-specific queries and for building the appropriate XSAMS tree from the fetched database access objects.

3.4 Comparison with the python/django node software

The paragraph provides a comparison between the Java-Implementation and the Python/Django node software

3.4.1 Common features

Both Java and Python node software implementations

- Work as a web application behind a web server
- Use object-relational mapping to access the database
- Provide the implementation of the parts that are common for all nodes
- Node-specific part works as a plugin, that means that no modification to the common components is needed during the node-specific part development.

3.4.2 Differencies

- **The main architectural difference** between the the Java implementation and the Python/Django one is the XML generator.

  Java version uses Document Object Model (DOM) mapping of the XML, Java node plugin needs to build XSAMS blocks as trees of objects.

  Python/Django version provides the XML generator with a defined and limited set of loops and anchors(“returnables”). Node developer needs to study not only the XSAMS documentation, but also to look through a plane list of keywords to correctly pin the data to the XML document structure.

  The use of DOM XML mapping has some advantages and disadvantages:
  - On a positive side, it gives more flexibility for the document generation. Any attribute of the output document can be accessed and set without the XML generator modification.
  - The output document is kept error-free because of the compile-time type checks and runtime XML special symbols filtering.
  - XML DOM mapping provided is complete: even if node developer wishes to put the data in a rarely used element of XSAMS, he can do it without the need to output XML blocks as the plain text.
  - On a bad side, the task of building a document tree requires an additional amount of node-specific code. Parts of XSAMS-extra library provide helper methods to simplify this task.
  - Memory consumption is higher due to the need to keep the whole document tree in memory.
• Document output is started after the construction of the XML tree, an additional delay is introduced, compared to the immediate streaming of the output by the Python/Django node software.

For the task of implementing XSAMS blocks builder, existing builders of KIDA, BASECOL and VALD may be used as examples.

• **Java implementation does not support document streaming.** The whole document tree is built in memory before producing the output XSAMS response.

  This approach allows to generate the document in the arbitrary order, i.e. export some species and states, then export processes, while exporting some more species and states.

• Java implementation does not provide any import tool from ASCII files into a relational database

  The node developer is himself responsible for creating and maintaining the node database structure and administration tool.

• Java implementation provides a sophisticated query parsing and mapping support

### 3.5 Node implementation

Implementing a node with the Java Node Software would require the following steps:

• The database model and classes should be created, as described in the Database Object Model section.

  After completing this step you will be able to access your database in a convenient way from any Java software you develop. For the details, see the Apache Cayenne documentation. [CAYDOC]

• The development environment for the plugin should be deployed. The query process and the interaction of the node and the plugin should be understood. See the Node architecture section for the details.

• XSAMS tree builders should be created, as described in the XSAMS tree building section. This work should be performed in collaboration with a person responsible for the scientific content of the database to establish the correspondence of the XSAMS elements and the database content.

  At this step the node plugin may be tested according to the procedure described in the section Database plugin testing. XSAMS generation should be verified and all the validation errors should be eliminated at this step.

• The supported restrictables and corresponding mappings should be defined, as described in the VSS query parsing and mapping section.

  Once this step is accomplished, it becomes possible to send different queries to the node. It should be checked if all the produced XSAMS documents are valid against the schema.

• The last development step would be to implement the query metrics that are used to estimate quickly if the node has the data for a particular query or not. See the Query metrics support section for the implementation details.

• Once the node plugin is tested and working, it may be deployed on the web server, as described in the VAMDC-TAP node deployment section. The node should be tested again with the VAMDC-TAP Validator working in the network mode.
Java implementation of VAMDC-TAP node software is a web-service application implementing a RESTful interface according to VAMDC-TAP standard specification.

It is accessible using the HTTP protocol, with URLs like `http://node.name.tld/vamdc-tap/12_07/VOSI/capabilities` or `http://node.name.tld/vamdc-tap/12_07/TAP/sync?query=select%20species&lang=VSS2&format=XSAMS`. Here `http://node.name.tld/vamdc-tap/12_07/` is the node base url, `/VOSI/capabilities` and `/TAP/sync` are endpoint addresses, and `?query=select%20species&lang=VSS2&format=XSAMS` are HTTP request parameters.

For the complete list of endpoint addresses implemented please refer to section **Web-service endpoints**.

### 4.1 Request processing

During normal operation, node software receives **HTTP GET** and **HTTP HEAD** requests to the `/TAP/sync` endpoint. As a response to a request a valid XSAMS document and/or the HTTP response headers are sent.

The figure below presents the request flow, highlighting the components involved in the request processing.

During the request processing the few steps are performed. Java VAMDC-TAP implementation (**Framework**) handles the steps that are common for all VAMDC-TAP nodes. Node **Plugin** is responsible for the steps that are...
essentially node-specific. To access the database, the Apache Cayenne object-relational mapping framework is used.

- On the query reception, the framework asks the plugin for the list of supported keywords (Restrictables).
- The Framework parses the incoming query. The validity of the query is checked. A tree of objects is constructed, representing the logical structure of the query. For more details please refer to the Query section.
- Framework asks Plugin to construct the document by calling the DatabasePlugin interface buildXSAMS(RequestInterface incomingRequest) method. All the query information is accessible via the RequestInterface object passed as a parameter.
- Plugin maps the incoming query logical tree into one or more Database queries, as described in the Query mapping scenarios section.
- Plugin queries the database and retrieves Apache Cayenne data objects. XSAMS elements are built from those data objects and are attached to the XSAMS document tree. See the XSAMS tree building section for the details.
- When the document tree is built, Plugin returns the control to the Framework.
- The Framework does the final checks on the document tree and calculates the accurate metrics for the document.
- The Framework converts the document tree into XML stream and sends it to the client.

4.2 Plugin and Framework Interfaces

Interaction between the database plugin and the Java node software is performed through two interfaces:

- ref DatabasePlug
defined in the org.vamdc.tapservice.api.DatabasePlugin java interface and implemented by the node developer. Methods of this interface are called by the node software upon the arrival of a VAMDC-TAP request or during the initial startup and operation checks.

- ref RequestInterface
defined in the org.vamdc.tapservice.api.RequestInterface java interface and implemented in the web-service framework. This interface is available to the node plugin to get the request information and feed the XSAMS blocks constructed.

4.2.1 DatabasePlugin interface

Each and every node plugin must implement the org.vamdc.tapservice.api.DatabasePlugin interface, defining the following methods:

- public abstract Collection<Restrictable> getRestrictables();
  Must return a collection of org.vamdc.dictionary.Restrictable dictionary keys that are supported by the node. This method is called once per each request to the /TAP/sync and /VOSI/capabilities endpoints.

- public abstract void buildXSAMS (RequestInterface userRequest);
  Build the XSAMS document tree corresponding to the incoming request. Object implementing RequestInterface interface is passed as a parameter. No return is expected. This method is called every time the node software is receiving an HTTP GET request to the /TAP/sync? endpoint.

  WARNING! Node plugin class is instantiated only once when the node is started, all calls to buildXSAMS should be thread-safe to handle concurrent requests correctly.

  Implementation details are covered in the XSAMS tree building section.
• **public abstract Map<Dictionary.HeaderMetrics,Integer> getMetrics(RequestInterface userRequest);**

Get the metrics corresponding to a query. This method is called for every incoming HEAD request to the `/TAP/sync/` endpoint. `RequestInterface userRequest` parameter is identical to the one passed to `buildXSAMS` method. This method should return a map of VAMDC-COUNT-* HTTP header names and their estimate values. For header names and meaning, see the [VAMDC-TAP] documentation section `DataAccessProtocol`.

• **public abstract boolean isAvailable();**

Do some really node-specific availability checks. This method is called periodically from the availability monitor. First call is initiated after the first request to the `/VOSI/availability` service endpoint. Method may be used to temporary shutdown the node during the database maintenance, or to do some integrity checks on the database. Availability check interval may be set in the `Node Configuration File` option `self-check_interval`.

**WARNING!** this method should not be used for doing periodic maintenance since it is never called before the first request to the `/VOSI/availability` service endpoint.

### 4.2.2 RequestInterface interface

When the incoming request arrives to Java Node Software, methods of the `DatabasePlugin` interface are called, as described earlier. Methods of that interface accept as a parameter an object implementing the `org.vamdc.tapservice.api.RequestInterface`. That interface provides the access to the request information and the Java Node Software facilities.

Following methods are part of this interface:

• **public abstract boolean isValid();** this method returns `true` if the incoming request is valid and should be processed. If the returned value is `false`, the request should be considered as invalid, no processing is required.

  Query string may be obtained by calling the `getQueryString` method and saved by the node plugin for the logging purposes.

• **public abstract Query getQuery();** This method returns the base object of the QueryParser library, implementing the Query interface.

  That interface is described in detail in the `Query` section of this document.

• **public abstract LogicNode getQueryTree();**

  This is the shortcut method returning the root element of the logic tree corresponding to the incoming query.

• **public abstract Collection<RestrictExpression> getQueryKeywords();** This is another shortcut method, allowing to get a collection of the keywords used in the incoming query. The keywords relation logic is omitted.

  **WARNING!** This method should not be used as the main source of data for the query mapping since it completely losses the query relation logic. Imagine the query:

  ```sql
  SELECT * WHERE AtomSymbol='Ca' OR AtomSymbol='Fe'
  ```

  If this method is used for the query mapping, this query would produce the same result as the query:

  ```sql
  SELECT * WHERE AtomSymbol='Ca' AND AtomSymbol='Fe'
  ```

  which is obviously incorrect.

• **public abstract String getQueryString();** The shortcut method to get the incoming query string.

• **public abstract boolean checkBranch(Requestable branch);** The shortcut method for the `Query.checkBranch()`, returns true if the result document is requested to contain a certain branch of XSAMS, specified by the `org.vamdc.dictionary.Requestable` name.
This method should be called in all builders to verify if a certain branch should be built, before even executing or mapping the queries.

The behaviour of the keywords is described in the VAMDC Dictionary documentation [VAMDCDict], the section Requestables.

- **public abstract ObjectContext getCayenneContext();** Returns the Apache Cayenne object context. That is the main endpoint of the Cayenne ORM library. For more information on using the Apache Cayenne look in the sections Database Object Model and Query mapping scenarios.

- **public abstract XSAMSManager getXSAMSManager();** Get an instance of the XSAMS document manager. This manager contains several helper methods for building the XSAMS document. All XSAMS branches built by the node plugin should be attached to it. Check the XSAMS tree building section for more details.

- **public abstract void setLastModified(Date date);**

  Sets the last-modified header of the response. This method may be called several times during the request processing. If called more than once, the last modification date is updated only if the subsequent date is newer than the value communicated before.
CHAPTER
FIVE

DATABASE PLUGIN TESTING

To test the node plugin, the VAMDC-TAP Validator software may be used. It implements the same DatabasePlugin interface call logic and RequestInterface methods operation as the VAMDC-TAP node webservice. All the dependency libraries are bundled in a single TAPValidator.jar archive.

Such an approach facilitates the node plugin development: there is no need to install the application server and deploy web service framework on the development machine.

To use the TAPValidator for the plugin development, few steps are required:

• The most recent version of TAPValidator.jar should be added to the project library path.

• New Java Application run/execution configuration should be defined, indicating the org.vamdc.validator.ValidatorMain as the main class.

• TAPValidator should be configured to operate in the plugin development mode. In the Settings window, plugin mode radiobutton should be selected; The class name implementing the DatabasePlugin interface should be indicated, including the java package name.

If everything is set up correctly, the list of supported restrictables should appear in the right-top text area. Plugin buildXSAMS(...) method is called every time the Query button is pressed.

For more information on the VAMDC-TAP Validator user interface and features, please refer to the [TAPValidator] documentation.
5.1 Eclipse project setup / Screenshots

The screenshots corresponding to the Eclipse project setup are presented below.

5.1.1 Adding VAMDC-TAP Validator to the build path

Open the project properties of your database plugin.

Add the latest VAMDC-TAP Validator JAR to the build path by clicking on the “Add external JARs” button.
5.1.2 Managing run configurations

Setup a new run configuration by clicking the “New...” button
5.1.3 Creating run configuration

Create a new run configuration with the `org.vamdc.validator.ValidatorMain` Main class path
5.2 Maven Integration

All Java software developed as a part of VAMDC is available at VAMDC Maven repository 
http://nexus.vamdc.org/nexus/content/repositories/releases/

To use Maven for dependency management of your plugin, a following sample POM.xml may be used:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.vamdc.%databasename%</groupId>
  <artifactId>plugin</artifactId>
  <name>databasename plugin for java node software</name>

  <repositories>
    <repository>
      <id>nexus</id>
      <name>VAMDC Releases</name>
      <url>http://nexus.vamdc.org/nexus/content/groups/public/</url>
    </repository>
  </repositories>

  <parent>
    <groupId>org.vamdc.tap</groupId>
    <artifactId>vamdctap-plugin</artifactId>
    <version>12.07r2</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.vamdc.%databasename%</groupId>
      <artifactId>database_dao</artifactId>
      <version>0.0.1-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.vamdc</groupId>
      <artifactId>TAPValidator</artifactId>
      <version>12.07r2</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

All the required dependencies are included in the parent project, `vamdctap-plugin`
CHAPTER SIX

DATABASE OBJECT MODEL

Java VAMDC-TAP node software implementation suggests and supports the use of Apache Cayenne Object-Relational Mapping (ORM) library.

Apache Cayenne supports various database engines, such as MySQL, PostgreSQL, SQLite, Oracle, DB2, Microsoft SQL Server.

Creation of database objects is made simple thanks to the graphical modeler application, provided as a part of Cayenne.

Process of creating and using a database model is well described in the project official documentation [CAYDOCU] and there is no need to repeat it in this document. Reading the Cayenne documentation is a MUST for the understanding and creating efficient query mapper routines and high performance database access classes.

6.1 Step-by-step guide

Here is a small illustrated guide on creating database mappings. We will need Cayenne modeler application, it can be downloaded from http://cayenne.apache.org/download.html as a part of binary distribution.

Version 3.1 is recommended for use with the 12.07r2 version of the Java VAMDC-TAP Node Software.

6.1.1 Create maven project

First we need to generate a Maven project:

```bash
mvn archetype:generate \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DgroupId=org.vamdc.database \
-DartifactId=daoclasses
```

and create folder src/main/resources where we will put cayenne modeler files.

Maven pom.xml can be replaced with the following contents:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.vamdc.databaseName</groupId>
  <artifactId>node_dao</artifactId>
  <version>12.07r2</version>
  <name>databaseName database objects</name>
  <parent>
    <groupId>org.vamdc.tap</groupId>
    <artifactId>cayenne_dao</artifactId>
  </parent>
</project>
```
6.1.2 Create Cayenne classes

Let’s open the cayenne modeler application and create a new project:

![Cayenne Modeler Application](image)

Now we need a dataNode describing the database connection.
And a dataMap that will contain all mapping tables and classes

After the DataMap is created, we need to import the database schema:
!Note the “Meaningful PK” flag is set to preserve the getter methods for the primary key fields.
As a result a separate Java class is generated for each database table, containing attribute fields and relationship references.

For many-to-many relations relationships need to be created manually: create a new relationship, press on the violet I button, indicate the path in the bottom of the window.

The last step is to generate class objects:
As a destination directory, src/main/java of Maven project needs to be specified. Cayenne project itself needs to be saved next to it, in src/main/resources.

6.2 Notable Cayenne features used

The use of an ORM framework provides several benefits:
  • Object-oriented view of the database;
  • Automatic relations traversal using the object methods (explained below);
  • Simplified translation of the incoming queries (see VSS query parsing and mapping)

6.2.1 Relations traversing

If properly defined, database model contains information about all table relations by means of the foreign keys. While constructing the query, those relations can be automatically traversed to form the correct query with desired selection criterias.

As an example, let us have a look at the case of two tables, ‘artists’ and ‘albums’, with a one-to-one mapping of albums to artists using the foreign key ‘albumArtist’ pointing to the ‘id’ field of the ‘artists’ table.

<table>
<thead>
<tr>
<th>artists</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

and

<table>
<thead>
<tr>
<th>albums</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

For the table ‘albums’ we have one-to-one relation with the ‘artists’ table, called ‘albumArtist’ and for artists the reverse one-to-many relationship ‘artistAlbums’

So, if we want to get all artists that released albums in 1997, we would create an Expression containing the path from the ‘artists’ table to the ‘year’ field of ‘albums’ table and the expression type ‘match’

```
Expression exp = ExpressionFactory.matchExp("artistAlbums.year", 1997);
SelectQuery query = new SelectQuery(Artists.class, exp);
List<Artists> artists = context.performQuery(query);
```

To add another constraint on a query, we may redefine the Expression:

```
exp = exp.andExp(ExpressionFactory.likeExp("name", "%Cocker%"));
```

Once we received the artists list, we may get the name of each artist, or have a look at all his albums:

```
for (Artists artist:artists){
    System.out.println("name: "+artist.Name+":");
    for (Albums album:artist.getArtistAlbums()){}
    System.out.println(" Album: "+album.Name+"("+album.Year+")");
}
```

This code should print something like:

Elton John:
Album: The Big Picture(1997)
Album: Goodbye Yellow Brick Road(1973)
Joe Cocker:
   Album: Across from Midnight(1997)
   Album: Respect Yourself(2002)

For the case of VAMDC node plugin, none of the expressions would require ‘manual’ construction: they will be translated from the incoming queries. Query translation is described in a separate chapter *Query mapping scenarios*

### 6.2.2 Path aliases

Imagine that we have the scenario of many-to-many relation through a separate table. For the previous example, let’s add a table ‘albumartist’ with three columns, ‘id’, ‘artistId’ and ‘albumId’ Table ‘albums’ does not contain the ‘artistID’ column any more, but both forward and reverse relations are still called ‘albumArtists’ and ‘artistAlbums’

Let us imagine that we need to select the artists that released their albums both in 1973 and 1997.

Joining expressions neither with exp.andExp nor exp.orExp would give us appropriate queries.

exp.andExp() would produce a query:

```sql
SELECT DISTINCT t0.name, t0.id
FROM artist t0
JOIN albumartist t1 ON (t0.id = t1.idArtist)
JOIN album t2 ON (t1.idAlbum = t2.id)
WHERE (t2.year = 1973) AND (t2.year = 1997)
```

that obviously never returns the data since the WHERE sub-conditions are mutually exclusive.

exp.orExp() would produce a query:

```sql
SELECT DISTINCT t0.name, t0.id
FROM artist t0
JOIN albumartist t1 ON (t0.id = t1.idArtist)
JOIN album t2 ON (t1.idAlbum = t2.id)
WHERE (t2.year = 1973) OR (t2.year = 1997)
```

that returns all the artists that released albums either in 1973 or 1997.

To resolve the problem, Apache Cayenne provides the aliases mechanism:

```java
Expression e1 = ExpressionFactory.match("artistAlbumsAlias1.year", 1997);
Expression e2 = ExpressionFactory.match("artistAlbumsAlias2.year", 1973);
Expression e = e1.andExp(e2);
SelectQuery q = new SelectQuery(Artists.class, e);
q.aliasPathSplits("artistAlbums", "artistAlbumsAlias1", "artistAlbumsAlias2");
```

That last command tells the select query how to interpret the alias. Because the aliases are different, the SQL generated will have two completely separate set of joins:

```sql
SELECT DISTINCT t0.name, t0.id
FROM artist t0
JOIN albumartist t1 ON (t0.id = t1.idArtist)
JOIN album t2 ON (t1.idAlbum = t2.id)
JOIN albumartist t3 ON (t0.id = t3.idArtist)
JOIN album t4 ON (t3.idAlbum = t4.id)
WHERE (t2.year = 1997) AND (t4.year = 1973)
```

This is called “split path” in the Apache Cayenne terms.

If applied to the case of VAMDC databases, this approach may be used for handling the cases where the data is linked through a separate join table, such as articles by author or year, or reactants in chemical reaction databases.

### 6.2. Notable Cayenne features used

23
XSAMS TREE BUILDING

XSAMS is an XML schema, adopted within VAMDC for data exchange. Java node software implementation uses JAXB library to establish the mapping between Java objects and XML elements and to read/write XSAMS documents.

Each node plugin is responsible for building object trees corresponding to the XSAMS document branches and for attaching them to the main tree, managed by the node software. Once the node plugin has finished building the XSAMS document and returned from the buildXSAMS() method, the node software outputs the XML document to the client.

Java objects corresponding to every element of XSAMS schema are generated using JAXB library and are a part of org.vamdc.xsams package, included in xsams and xsams-extra libraries.

XSAMS objects can be constructed by extension of Jaxb mapping classes with convenience methods, provided within xsams-extra library. Constructors can receive Cayenne mapping objects as argument and initialize appropriate mapping XML fields. Such an approach allows to instantly apply an arbitrary processing to any field/element value or their combinations.

7.1 Example constructor class

As an example let’s look at the BASECOL Source element constructor:

```java
package org.vamdc.basecol.xsams;

import org.vamdc.basecol.dao.RefsArticles;
import org.vamdc.basecol.dao.RefsGroups;
import org.vamdc.xsams.XSAMSManager;
import org.vamdc.xsams.schema.SourceCategoryType;
import org.vamdc.xsams.schema.SourceType;
import org.vamdc.xsams.util.IDs;

public class Source extends SourceType{
    public Source(RefsArticles article){
        setSourceID(IDs.getSourceID(article.getArticleID().intValue()));
        setCategory(SourceCategoryType.fromValue(article.getJournalRel().getCategory()));
        setSourceName(article.getJournalRel().getSmallName());

        //Year
        setYear(article.getYear().intValue());

        //Authors
        setAuthors(new Authors(article.getFlatAuthorRel()));

        //Title
        setTitle(article.getTitle());
    }
}
```
The full source is available in `org.vamdc.basecol.xsams.Source` class.

Here, `RefsArticles` is the BASECOL Cayenne mapping object identifying one source record in the database, and `SourceType` is a root element of XSAMS Sources branch. `SourceType` is defined by the class `org.vamdc.xsams.schema.SourceType`.

Collection of `RefsArticles` objects is retrieved automatically through the Cayenne model relation. For each `Source` element we need to check if it is already attached to the XSAMS Document tree. If no existent object is found, the aforementioned builder is called and the generated object is attached to the document tree:

```java
    public static List<SourceType> getSources(
            List<RefsGroups> referenceRel,
            XSAMSManager document,
            boolean filterSource) {
        ArrayList<SourceType> result = new ArrayList<SourceType>();
        /*always add the database self-reference*/
        result.add(document.getSource(IDs.getSourceID(0)));
        if (referenceRel==null) return result;
        /*Add all sources that are stated as 'isSource'*/
        for (RefsGroups myref:referenceRel) {
           RefsArticles article = myref.getArticleRel();
            if (article!=null && (myref.getIsSource() || !filterSource)){
                //Check if the source with this ID is already referenced:
                SourceType source = document.getSource(
                        IDs.getSourceID(article.getArticleID().intValue()));
                if (source == null){ //If not, create and add it:
                    source = new Source(article);
                    document.addSource(source);
                }
                //Now, add source record to the list of source references
                result.add(source);
            }
        }
        return result;
    }
```

This list of the `SourceType` objects should be passed as the argument to the objects requiring the bibliographic references. For example, if a new `DataType` object is created, references may be attached to it in the following manner:

```
7.1. Example constructor class
```
```java
DataType quantity = new DataType(table.value, table.units);
quantity.addSources(Source.getSources(table.sourceRelation, request, true));
```

Here, “table” is an object of the database model, providing value and units fields plus the relation to the sources.

### 7.2 Attaching objects to the XSAMS Document tree

The interface `RequestInterface` provides the access to the XSAMS Document tree through the `XSAMSManager` interface. Node plugin may obtain the XSAMSManager by calling the `getXsamsManager()` method of the `RequestInterface`.

The `org.vamdc.xsams.XSAMSManager` interface provides a handful of methods to add different branches to the XSAMS tree, getting them by known ID or iterating through all of them. For a full list of methods, consult the JavaDoc of the JAXB XSAMS library [XSAMSJavaDoc].

Most commonly used methods are:

- `String addSource(SourceType source);` returning the source identifier
- `String addElement(SpeciesInterface species);` returning the species identifier
- `int addStates(String speciesID, Collection<? extends StateInterface> states);` returning the number of added atomic or molecular states
- `boolean addProcess(Object process);` returning true if the process element was attached to the tree for adding the sources, species, states and processes respectfully.

For each XSAMS block that has an identifier, a convenience method is implemented allowing to fetch the root element of the block by its string identifier. For example, for the sources block there exists a method `SourceType getSource(String sourceID).`

A complete list of methods is available in the JavaDoc of the XSAMSManager interface [XSAMSJavaDoc].

### 7.3 Identifiers generation

Each major block of an XSAMS document has an unique identifier, an arbitrary alphanumeric string starting with a block-specific symbol.

To assure VAMDC-wide uniquiness of those identifiers, a node-specific prefix is inserted into every identifier string. For instance, that allows to merge the documents coming from the different databases.

In Java node software the identifiers are managed by a special class, `org.vamdc.xsams.IDs`. That class provides several static constants and methods:

- `String getID(char prefix, String suffix)` Most generic method, allowing to generate an arbitrary identifier. The allowed prefix values are enumerated as `public final static char` constants:
  - `IDs.SOURCE`
  - `IDs.ENVIRONMENT`
  - `IDs.SPECIE`
  - `IDs.FUNCTION`
  - `IDs.METHOD`
  - `IDs.STATE`
  - `IDs.MODE`
  - `IDs.PROCESS`
- `String getSourceID(int idSource);`

---

**7.2. Attaching objects to the XSAMS Document tree**
• String getEnvID(int idEnv) for the environment identifiers
• String getFunctionID(int idFunction);
• String getMethodID(int idMethod);
• String getStateID(int EnergyTable, int Level);
• String getModeID(int molecule, int mode);
• String getSpecieID(int idSpecies);
• String getProcessID(char group, int idProcess);

All those ID generation methods automatically add the configured node-specific ID prefix.

### 7.4 XSAMS JAXB convenience extensions

For convenience, all the XSAMS object classes were extended and grouped by the schema block into different packages:

- `org.vamdc.xsams.common` for elements used all around the schema
- `org.vamdc.xsams.environments` for elements from the Environments branch
- `org.vamdc.xsams.functions` for elements from the Functions branch
- `org.vamdc.xsams.methods` for elements from the Methods branch
- `org.vamdc.xsams.process` for elements from the Processes (collisions, transitions) branch
- `org.vamdc.xsams.sources` for elements from the Sources branch
- `org.vamdc.xsams.species` for elements from the Species (atoms, molecules, particles, solids) branch

Few value constructors were added:

- class `org.vamdc.xsams.species.molecules.ReferencedTextType`
  ```java
  public ReferencedTextType(String value);
  ```
  Creates a ReferencedTextType element with the defined value

- class `org.vamdc.xsams.sources.AuthorsType`
  ```java
  public AuthorsType(Collection<String> authors)
  public AuthorsType(String concatAuthors, String separator)
  ```
  First constructor creates Authors element with all authors from the passed collection, second one splits the first argument using the separator from the second one and puts the resulting strings into distinct Author records.

- class `org.vamdc.xsams.sources.AuthorType`
  ```java
  public AuthorType(String name)
  ```
  Creates a single Author element with the name from the argument.

- class `org.vamdc.xsams.common.TabulatedDataType`
  ```java
  public TabulatedDataType(String... CoordsUnits);
  public TabulatedDataType(Collection<String> columns);
  ```
  Constructors, defining multi-dimensional tables. Parameters passed define the units of axes, the last element of the collection or the last string define the units for Y (values). The `org.vamdc.xsams.common.TabulatedDataType` class contains a full set of methods for the XSAMS tables manipulation, so if you need to use them it is worth reading the XSAMS library JavaDoc [XSAMSJavaDoc]

- class `org.vamdc.xsams.common.DataType`
You will certainly use DataType objects, since almost any quantity in XSAMS is represented by them. Two constructors are provided, with parameter names speaking for themselves. Source references may be attached to created object later by calling the `addSource()` or `addSources()` methods.

- class `org.vamdc.xsams.common.ValueType`:
  ```java
  public ValueType(Double value, String units);
  ```
  ValueType, used as often as the DataType, supports no source reference and is a simple extension of the Double type, providing the `units` attribute. Convenience constructor is also provided for it.

- class `org.vamdc.xsams.common.ChemicalElementType`:
  ```java
  public ChemicalElementType(int charge, String symbol);
  ```
  Used in Atoms and Solids branches, ChemicalElementType has a convenience constructor consuming the atom nuclear charge and its chemical element symbol.

### 7.5 Case-By-Case generic builders

Molecular state quantum numbers in XSAMS are represented as additional XML sub-schemas, defining an element QNs with the ordered child quantum number elements. Each case has its own separate namespace, that means that Java JAXB mapping of each case would be in a separate package and the user would either require a generic builder using Java Reflection or have a builder for each case.

Since all cases are just combinations of roughly 30 quantum numbers, it was chosen to create an intermediate structure to contain them, together with the case identifier. The class name is `org.vamdc.xsams.util.StateCore`. It holds a collection of quantum numbers and other important state-related information.

Each quantum number is represented by the `org.vamdc.xsams.util.QuantumNumber` object. That object contains the value, optional label and mode index, plus the mandatory quantum number type.

Each case package is complemented with its own builder. The general case builder `org.vamdc.xsams.cases.CaseBuilder` accepts the `StateCore` object as a single parameter. Depending on the caseID parameter value, a specific case builder is called, returning the built tree. The identifier Case ID is the same as defined in the case-by-case documentation.

The following code illustrates the use of the `CaseBuilder`:

```java
StateCore statedata = new BasecolStateCore(myetable, level);
MolecularStateType molecularState = new MolecularStateType();
// filling in other MolecularStateType fields is omitted
if (myrequest.checkBranch(Requestable.MoleculeQuantumNumbers))
    molecularState.getCases().add(CaseBuilder.buildCase(statedata));
```

Here, `BasecolStateCore` is a custom class that extends `StateCore`, providing a custom constructor to fill the fields from the Basecol Cayenne objects.

`MolecularStateType` is the autogenerated XSAMS JAXB mapping class that should be fed directly to the XSAMS library by calling the:

```java
RequestInterface.getXsamsroot().addState(speciesID, molecularState);
```

Please note that the element corresponding to the given speciesID should be attached to the XSAMS tree before attaching the state.
VSS QUERY PARSING AND MAPPING

Vamdc nodes are queried using a common SQL-like keyword-based query language. A typical query looks like:

```
SELECT * WHERE (keyword1=value or keyword2=value2) and keyword3='value3'
```

where the keywords correspond to the different parts of the XSAMS document and are defined in the VAMDC Dictionary [VAMDCDict] as Restrictable keywords. Since every VAMDC node has a specific database structure, the incoming query keywords need to be mapped to the database fields in order to fetch the data corresponding to the query. Java Node software tries to make the process of query mapping as simple and sophisticated as possible.

### 8.1 Query keywords tree

As a first step, on query reception the Java Node Software parses the query into a tree of objects using the `vamdcTap-queryparser` library. Leaf nodes of that tree of objects represent the individual keyword expressions. The root node and the branches are reflecting the boolean relations between the leaves. For instance, a query:

```
SELECT * WHERE reactant1.AtomSymbol = 'C' AND reactant2.AtomSymbol = 'O'
AND (CollisionCode='inel' or CollisionCode='elas')
```

would map into the tree:

![Query Keywords Tree Diagram]

**LogicNode**

**AND**

**RestrictExpression**

- Pref: reactant 1
- Kw: AtomSymbol
- Op: equals
- Val: C

**RestrictExpression**

- Pref: reactant 2
- Kw: AtomSymbol
- Op: equals
- Val: O

**LogicNode**

**OR**

**RestrictExpression**

- Pref: null
- Kw: Operation
- Op: equals
- Val: Restrictable

**RestrictExpression**

- Pref: null
- Kw: Operation
- Op: equals
- Val: value

**RestrictExpression**

- Pref: null
- Kw: Operation
- Op: equals
- Val: CollisionCode

**RestrictExpression**

- Pref: null
- Kw: Operation
- Op: equals
- Val: CollisionCode

**RestrictExpression**

- Pref: null
- Kw: Operation
- Op: equals
- Val: CollisionCode

**RestrictExpression**

- Pref: null
- Kw: Operation
- Op: equals
- Val: CollisionCode

**RestrictExpression**

- Pref: null
- Kw: Operation
- Op: equals
- Val: CollisionCode
8.2 QueryParser library interface

The QueryParser library provides an interface org.vamdc.tapservice.vss2. Query providing several methods to access the incoming query elements.

8.2.1 Query

The methods to access the incoming query elements are:

- **public LogicNode getRestrictsTree()** is the main method, returning the root of the query tree. Accompanying are two methods, getFilteredTree and getPrefixedTree, returning the subsets.
- **public LogicNode getFilteredTree(Collection<Restrictable> allowedKeywords)** returns a subtree containing only the keywords present in the collection that is passed as a parameter.
- **public LogicNode getPrefixedTree(VSSPrefix prefix, int index)** returns a subtree containing only the keywords having the prefix and index that are passed to the method. If null is passed as a prefix, returned tree would only contain nodes that have a null prefix.
- **public Collection<Prefix> getPrefixes()** returns a collection of prefixes that are present in the query
- **public List<RestrictExpression> getRestrictsList()** returns a list of all the keywords that are present in the query. The logic of relations between the elements is lost.

8.2.2 LogicNode

The **LogicNode** interface represents a node of the query tree. **getOperator()** method provides access to the node operator (OR, AND, NOT), **getValues()** returns a collection of child nodes.

For the single-value operators like **NOT**, the **getValue()** method may be used to obtain the child element directly.

8.2.3 RestrictExpression

The leaf nodes of the logic tree are implementing the **RestrictExpression** interface, representing the query expressions.

The **RestrictExpression** interface is the extension of the **LogicNode** interface providing the **getOperator()**, **getValues()** and **getValue()** methods, as well as few additional methods:

- **public Prefix getPrefix()** a method returning the prefix used in the expression.
- **public Restrictable getColumn()** a method returning the Restrictable keyword used by the expression.

8.2.4 Prefix

Prefix is a simple class, keeping the **VSSPrefix** keyword of the dictionary, plus an integer index of the prefix.

- **int getIndexOf()** method provides access to the index, and
- **VSSPrefix getPrefix()** gives access to the prefix name.
8.3 Filtering logic

The algorithm implemented for the `getFilteredTree()` and `getPrefixedTree()` methods of the `Query` interface works the following way: It removes the irrelevant `RestrictExpression` objects from `LogicNodes`, than removes the Nodes of the tree that have no child expression elements.

For example, in the case of filtering by the prefix:

- Original query:
  ```sql
  select ALL where reactant1.AtomSymbol='C' and reactant1.AtomIonCharge=1 and reactant2.AtomSymbol='H' and reactant2.AtomIonCharge=-1 and EnvironmentTemperature > 100
  ```

- Effective query for `getPrefixedTree(VSSPrefix.REACTANT, 1)`:
  ```sql
  select ALL where reactant1.AtomSymbol='C' and reactant1.AtomIonCharge=1
  ```

- Effective query for `getPrefixedTree(VSSPrefix.REACTANT, 2)`:
  ```sql
  select ALL where reactant2.AtomSymbol='H' and reactant2.AtomIonCharge=-1
  ```

- Effective query for `getPrefixedTree(null, 0)`:
  ```sql
  select ALL where EnvironmentTemperature > 100
  ```

- Effective query for `getFilteredTree(Collection<Restrictable>{AtomSymbol})` with a collection containing only the `AtomSymbol` element:
  ```sql
  select ALL where reactant1.AtomSymbol='C' and reactant2.AtomSymbol='H'
  ```

8.4 Query mapping scenarios

To perform queries on the actual database, the incoming VAMDC-TAP query needs to be adapted to the database structure. In the case of Apache Cayenne, `org.apache.cayenne.exp.Expression` object needs to be constructed, representing the query logic relative to the database primary table.

In this chapter the way to construct the query expressions from the incoming query is described.

8.4.1 Mapping of the LogicTree

Mapping of the LogicTree nodes is simple with one-to-one mapping of AND, OR and NOT operators to Cayenne `Expression.andExp()`, `Expression.orExp()`, `Expression.notExp()`, see the Cayenne Javadoc [CAYJAVADOC]. Usable example of such mapper is provided in `org.vamdc.tapservice.query.QueryMapper` class (vamdctap-querymapper library), that is bundled both with the TAPValidator and the node software.

8.4.2 Mapping of RestrictExpression elements

Mapping of the leaf nodes of the query tree, represented by the `RestrictExpression` elements, may be a bit more tricky. A lot of the information is contained in leaf nodes that should be mapped into the query logic:

- Keyword prefix
- Prefix index
- VAMDC dictionary Restrictable keyword
- comparison operator
- value or a set of values
VAMDC Restrictable keyword may correspond to one or several database fields. For example, let us have a database where all the species: particles, atoms and molecules are stored in a single table, where the inchikeys are defined for atoms and molecules, and there is a field indicating the species type. In this case the MoleculeInchiKey keyword in a query will map to two internal Expression constraints:

- a constraint on the inchikey field of the table,
- and a constraint indicating that the species is actually a molecule.

To correctly handle such a keyword we will need to join two Cayenne Expressions with the Expression.andExp operator, then add them to the mapped query tree.

Prefix and prefix index may also impose a check for a certain field, like if element is a reactant or product in a chemical reaction.

To handle the prefixes, it is possible to loop through the prefixes present in the query by using Query.getPrefixes() method. The second step would be to filter the incoming query tree by the prefix using the Query.getPrefixedTree(...) method, then map it to Cayenne Expressions, and finally join the obtained expressions with the Expression.andExp() method.
8.5 Query Mapping Library

As a part of the Java Node software, a Query Mapper library (vamdctap-querymapper) is provided. It is able to map the incoming query trees into cayenne Expression objects. Two interfaces and generic implementations within a package org.vamdc.tapservice.querymapper are provided:

- **KeywordMapper** defining an interface of a RestrictExpression mapper;
- **KeywordMapperImpl** a generic implementation, providing one-to-one mapping of Restrictable keywords to the database fields. No value transformation is performed. The node plugin may extend this class to implement the value translation, one-to-many fields mapping, or even the prefix-conditional mapping.
- **QueryMapper** defining the library main interface;
- **QueryMapperImpl** providing a generic implementation. That implementation keeps the references to all the defined KeywordMapper instances. It is capable of mapping the parsed query trees to Cayenne Expressions. Boolean logic of LogicNode tree is translated one-to-one by using the Apache Cayenne Expression andExp(), orExp() and notExp(). For leaf nodes the registered KeywordMapper instances are called to translate the RestrictExpression into the Cayenne Expression objects.

8.5.1 Using the querymapper library

From the plugin point of view the mapper library is used in the following way:

- A static instance of QueryMapper is initialized. For each supported Restrictable keyword a new mapper is added:

  ```java
  public final static QueryMapper queryMapper= new QueryMapperImpl(){
      this.addMapper(
          new KeywordMapperImpl(Restrictable.IonCharge)
          .addNewPath("symelementRel.elementRel.charge")
          .addNewPath("partyRel.elementRel.charge")
          );
  });
  ```

  To define the database relations paths originating from different primary tables, subsequent calls to addNewPath method are used. Here for example the first path originates from the Species table, the second one originates from the Processes.

- If the advanced mapping is necessary, the class KeywordMapperImpl may be extended. Extension classes may add for example a possibility to map keywords to multiple fields, translate values from query units to database units.

- The QueryMapper implementation automatically handles a list of Restrictable keywords supported by the node. That list may be fetched by the public Collection<Restrictable> getRestrictables() method.

- The mapping of the incoming query trees or filtered subtrees is performed using the QueryMapper methods mapAliasedTree(...) or mapTree(...). Cayenne Expression objects are produced as the output.
QUERY METRICS SUPPORT

To give the estimation of the volume of data returned by the node for a query, the node software supports HEAD request queries. The parameter set of the HEAD request is the same as for the GET requests.

The VAMDC-specific HTTP headers that may be present in the response are the following:

- **VAMDC-COUNT-SPECIES** Total count of the atomic Ion and Molecule records with distinct SpecieID attribute.
- **VAMDC-COUNT-ATOMS** Count of the atomic Ion records with distinct SpecieID attribute.
- **VAMDC-COUNT-MOLECULES** Count of the Molecule records with distinct SpecieID attribute.
- **VAMDC-COUNT-SOURCES** Count of distinct Source records.
- **VAMDC-COUNT-STATES** Count of distinct State records, both AtomicState and MolecularState combined.
- **VAMDC-COUNT-COLLISIONS** Count of the CollisionalTransition elements of the Processes branch of XSAMS.
- **VAMDC-COUNT-RADIATIVE** Count of the RadiativeTransition elements of the Processes branch of XSAMS.
- **VAMDC-COUNT-NONRADIATIVE** Count of the NonRadiativeTransition elements of the Processes branch of XSAMS.
- **Last-Modified** HTTP header can also be sent to client to indicate the time when the extracted data was modified last time.

### 9.1 getMetrics(...) method

HTTP HEAD response may be produced without building the XSAMS document tree. For this purpose a dedicated plugin method is called by the node software for each incoming HEAD request.

:: public abstract Map<Dictionary.HeaderMetrics, Object> getMetrics(RequestInterface userRequest);

The **userRequest** parameter has the same interface as the parameter of the plugin buildXSAMS method. Since the HEAD response will not transmit the XSAMS document body, there is no need to access the XSAMSManager object.

Typical logic of the method implementation would be the following:

- Translate the query using the same translation strategy as the builders use;
- Convert the query into the `SELECT Count(*) WHERE ...` using the `org.vamdc.tapservice.query.QueryUtil.countQuery(DataContext, SelectQuery)` static method;
- Add the obtained count() value along with the corresponding HTTP header to the map of result headers;
• If the obtained count() value is greater than zero, set the value of the Last-Modified header by using the RequestInterface setLastModified(...) method and org.vamdc.tapservice.query.QueryUtil.lastTimestampQuery(...) translator to obtain the value from the database last-modified date column.

• iterate over all primary tables if more than one primary table is used to query the database.

9.2 Sample implementation

Here follows the sample implementation from BASECOL database plugin

```java
@override
public Map<HeaderMetrics, Integer> getMetrics(RequestInterface request) {
    if (request.isValid() && checkRequest(request)){
        return Metrics.estimate(request);
    }
    return null;
}
```

and Metrics.estimate has the following implementation:

```java
public static Map<HeaderMetrics, Object> estimate (RequestInterface request){
    Map<HeaderMetrics, Object> estimates = new HashMap<HeaderMetrics, Object>();

    //Estimate collisions
    Expression colExpression = CollisionalTransitionBuilder.getCayenneExpression(request);
    SelectQuery query=new SelectQuery(Collisions.class,colExpression);
    Long collisions = QueryUtil.countQuery((DataContext) request.getCayenneContext(), query);
    if (collisions>0){
        estimates.put(HeaderMetrics.VAMDC_COUNT_COLLISIONS, collisions.intValue());
        request.setLastModified(QueryUtil.lastTimestampQuery(
            (DataContext) request.getCayenneContext(),
            query,
            "modificationDate"));
    }

    //Estimate species
    Expression spExpression = ElementBuilder.getExpression(request);
    SelectQuery spQuery=new SelectQuery(EnergyTables.class,spExpression);
    Long etables = QueryUtil.countQuery((DataContext) request.getCayenneContext(), spQuery);
    if (etables>0){
        estimates.put(HeaderMetrics.VAMDC_COUNT_SPECIES, etables.intValue());
        request.setLastModified(QueryUtil.lastTimestampQuery(
            (DataContext) request.getCayenneContext(),
            spQuery,
            "modificationDate"));
    }

    return estimates;
}
```

Here, the getExpression(...) methods are the same translator methods as used in corresponding XSAMS builders.
10.1 Install a Java application server

The Java implementation of the VAMDC-TAP node software is provided as a web application archive (.war) and is intended to be run under a Java application server like Apache Tomcat. For the server installation instructions refer to the server documentation.

10.2 Deploy the node software

Once the node plugin is tested with TAPValidator and working, a few steps are needed to install(deploy) a VAMDC node using the Java Node software.

1. Download the latest version of Java Node Software from the VAMDC site: http://www.vamdc.org/software/
   What you need is a web application archive vamdctap-webservice-xxx.war

2. Deploy the downloaded .war file using your server default deploy method. (see the server documentation).
   The contents of the .war archive is unpacked to webapps/vamdctap.../ by the server.

3. The Apache Cayenne configuration files need to be copied to WEB-INF/classes/ subdirectory of the unpacked web application archive. The cayenne project file needs to be renamed into cayenne-project.xml
   Database credentials can be changed in WEB-INF/classes/cayenne-project.xml or in dbcp.properties if you are using Apache DBCP for the database connection pooling.

4. Load the address http://$BASEURL/config to see the default configuration. To alter the options, a key-value pairs should be saved in a text file WEB-INF/config/tapservice.conf. For a full list of options and their descriptions please refer to the section Node Configuration File.

5. The node plugin and DAO jar files should be copied to WEB-INF/lib/ directory of the deployed web application.
   !WARNING! Never try to copy those jars to the webserver or java system library paths. Such a configuration will never work, causing unexpected ClassCast exceptions.

6. The application or server should be reloaded to update the configuration and load the libraries.

7. Try to access the Capabilities and Availability endpoints of the service. They should be accessible through http://$BASEURL/VOSI/capabilities and http://$BASEURL/VOSI/availability. Availability status should be Service is currently available: true

8. !WARNING! Do a backup copy of all configuration files and .jar files that were put or adjusted in the deployed application folder, notably the node configuration files. All those files will be erased on every vamdctap-webservice war redeployment and should be recopied in place every time the Java Node Software web application is updated or relocated.

9. Once the deployment procedure is complete, the node operation should be tested using the TAPValidator application. In the TAPValidator configuration the network mode should be chosen and the capabilities endpoint address of the node should be indicated.
10.3 Node Configuration File

Java VAMDC-TAP node is configured through a single file, containing a set of key-value pairs.

The configuration may be altered by modifying the file WEB-INF/config/tapservice.conf in the web application root directory. Once the new parameters are set, the application should be reloaded to take the account of modifications.

The current node configuration may be retrieved by accessing the config endpoint: http://host.name:8080/tapservice/config. Here host.name:8080/tapservice is the url of the deployed web application.

10.3.1 Default configuration parameters

By default, the node parameters take the following values:

```plaintext
limits_enable=false
limit_states=-1
limit_processes=-1
selfcheck_interval=60
database_plug_class=org.vamdc.database.tap.OutputBuilder
dao_test_class=org.vamdc.database.dao.ClassName
xsams_id_prefix=DBNAME
baseurl=http://host.name:8080/tapservice#http://mirror.host/tapservice
xml_prettyprint=false
test_queries=select species where atomsymbol like '%';select * where inchkey='UGFAIRIUMAVXCW-UHFFFAOYSA-N'
returnables=keyword1;keyword2;...
processors=ivo://vamdc/processor_1#ivo://vamdc/processor2
```

10.3.2 Detailed parameters description

- **limits_enable** = (true|false) enable output document element count limits
- **limit_states** = N - limit maximum output states count in document to N, “-1” disables the limit.
- **limit_processes** = N - limit maximum output processes count in document to N, “-1” disables the limit.
- **selfcheck_interval** = N interval in seconds between the node availability self checks. First check is initiated upon the request to /VOSI/availability endpoint. Background checks allow the accurate tracking of “upSince”, “backAt” and “downAt” time attributes.
- **database_plug_class** = org.vamdc.database.tap.OutputBuilder The name of the class implementing the org.vamdc.tapservice.api.DatabasePlugin interface. An instance of that class is created on the web application startup. Methods of this class are invoked to construct a response for the incoming request.
- **dao_test_class** = fully.qualified.apache.cayenne.dao.Object The name of a Cayenne DAO class, used for the node availability checking. A table behind that class should have more than 10 records. This option may be omitted but in that case the availability endpoint will not work properly.
- **xsams_id_prefix** = DBNAME Prefix for XSAMS library id generator org.vamdc.xsams.util.IDs, used to produce all XML id/idRef references. Each node needs to have its own prefix to maintain global uniqueness of identifiers in XSAMS documents.
- **baseurl** = http://host.name:8080/tapservice Base url used in VOSI/capabilities output, must contain globally accessible URL pointing to the VAMDC-TAP service web application root. Multiple addresses may be specified to indicate mirror nodes, separated by # symbol. The first address must point to the node itself.
- **xml_prettyprint** = (true|false) Produce pretty-printed XML documents or output everything in a single line of text with no linefeeds. Defaults to false, enabling it increases the size of output XML document by ~20%
- **test_queries** = select species where atomsymbol like '%'; semicolon-separated list of valid test queries for the node. It must contain only valid queries that demonstrate the full functionality of the node. On the other
hand such queries must produce compact documents, since those queries would be used for periodic node testing.

- **returnables** a semicolon-separated list of Returnable keywords that will be shown in the Capabilities endpoint output. May be left empty.

- **processors** a list of IVOA identifiers of the preferred XSAMS processors. This list is used by the vamdc portal to suggest the processors. Processor identifiers may be obtained from the VAMDC registry using the TAPValidator application (starting from version 12.07r2).

To obtain the configuration parameter, open the TAPValidator, in the menu bar select Tools->Processors. A window will appear, containing the table of the registered processors. Select the lines while holding the ctrl key. Press the “Copy Selected” button. A text line containing the processors configuration option is copied to clipboard and may be pasted directly into the configuration file using your preferred text editor.

### 10.4 Cayenne configuration using DBCP

To avoid database connection time-out errors, Apache **commons-dbcp** library should be used in junction with Apache Cayenne. Configuration change for this case is simple and straight-forward. **vamdtap-webservice** application archive already comes with bundled **commons-dbcp** jar.

- in the file cayenne-project.xml the **factory** attribute of the **node** element should be changed to org.apache.cayenne.conf.DBCPDataSourceFactory.

- **dbcp.properties** configuration file should be put next to the cayenne-project.xml in the classes directory. the following parameters should be put into the file:

  ```
cayenne.dbcp.driverClassName=com.mysql.jdbc.Driver
cayenne.dbcp.url=jdbc:mysql://hostname:port/databasename
cayenne.dbcp.username=databaseUserName

cayenne.dbcp.password=databasePassword
```

Other DBCP parameters may be adjusted, please consult [http://commons.apache.org/dbcp/configuration.html](http://commons.apache.org/dbcp/configuration.html) for more information.

### 10.5 Database updates and cache

Java node software maintains its own database cache. If database fields are updated, this cache needs to be purged. To force node to purge its caches, request to $BASEURL/clear_cache resource needs to be sent. The full URL will be [http://host.name:8080/tapservice/clear_cache](http://host.name:8080/tapservice/clear_cache). Result document contains a single line of text, indicating the number of records that were contained in the cache.

This URL should be called every time the database contents is updated. It may be accessed either manually, included as an iframe in the node database administration interface, or accessed by a special script that tracks database modifications in some way.

### 10.6 Node mirroring

The best way to set up node mirrors is to configure database replication and deploy an instance of the node software on each mirror.

Deployment procedure of node software on master server and mirrors is the same. The only difference will be the order of the addresses in the **baseurl** configuration parameter on each of the mirrors.
10.6.1 Mysql master server configuration

On master server we need to enable binary logging in my.cnf:

```
[mysqld]:
server-id = 1
log-bin = /var/lib/mysql/mysql-bin
replicate-do-db = databasename
bind-address = 0.0.0.0
```

and add user with replication privileges in mysql console:

```
mysql@master> GRANT replication slave ON "databasename".* TO "replication"@"mirror.ip.or.hostname" IDENTIFIED BY "password";
```

mysql service needs to be restarted after that.

After server restart we need to create a database dump:

```
mysql@master> FLUSH TABLES WITH READ LOCK; mysql@master> SET GLOBAL read_only = ON; mysql@master> SHOW MASTER STATUS
#mysqldump -u root -p databasename | bzip2 -9 -c - > database_dump.sql.bz2
mysql@master> SET GLOBAL read_only = OFF;
```

Here we need to note File and Position values from the mysql command Show Master Status.

10.6.2 Mysql slave configuration

On a slave (replica) server we need to set up the following things:

import mysql dump for database:

```
#bzip2 -d -c database_dump.sql.bz2 | mysql -u root -p databasename
```

In my.cnf:

```
[mysqld]:
server-id = 2
relay-log = /var/lib/mysql/mysql-relay-bin
relay-log-index = /var/lib/mysql/mysql-relay-bin.index
replicate-do-db = testdb
```

restart mysql service and start replication:

```
mysql@replica> CHANGE MASTER TO MASTER_HOST = "master.ip.or.host", MASTER_USER = "replication", MASTER_PASSWORD = "password", MASTER_LOG_FILE = "mysql-bin.000003", MASTER_LOG_POS = 98;
mysql@replica> start slave;
```

where MASTER_LOG_FILE and MASTER_LOG_POS parameters we take from the SHOW MASTER STATUS G mysql command on master server.

Then we may see the slave server status by issuing mysql command:

```
mysql@replica> SHOW SLAVE STATUS\G
```

Following status parameters are important and indicated values show that replication is working properly:

- **Slave_IO_State**: Waiting for master to send event
- **Slave_IO_Running**: Yes
- **Slave_SQL_Running**: Yes
- **Seconds_Behind_Master**: 0
Changing `Read_Master_Log_Pos` parameter value may indicate that cache of node software on the mirror needs to be purged. A script can be set up to track this parameter, if no other mean of cache invalidation is used.


## 10.7 Node registration

To update the node registration to include the mirror nodes, two steps are needed:

- In the configuration of the main node and all mirrors the `baseurl` parameter should be set up to include the node and mirrors, separated with hash (#) symbol. On mirror nodes, the url of the mirror itself should be the first in that list. Java node web application should be reloaded to update the configuration.

- Registry record of the node should be updated with new Capabilities including new mirror.

In any case for the registry update please contact the VAMDC registry maintainer via support@vamdc.org.
BIBLIOGRAPHY

[XSAMSJavaDoc]  http://dev.vamdc.org/nexus/content/repositories/releases/org/vamdc/xml/
[MavenRepo]  http://dev.vamdc.org/nexus/content/repositories/releases/