**UnionVms**

**Developing new Database Modules**

Joakim Johansson

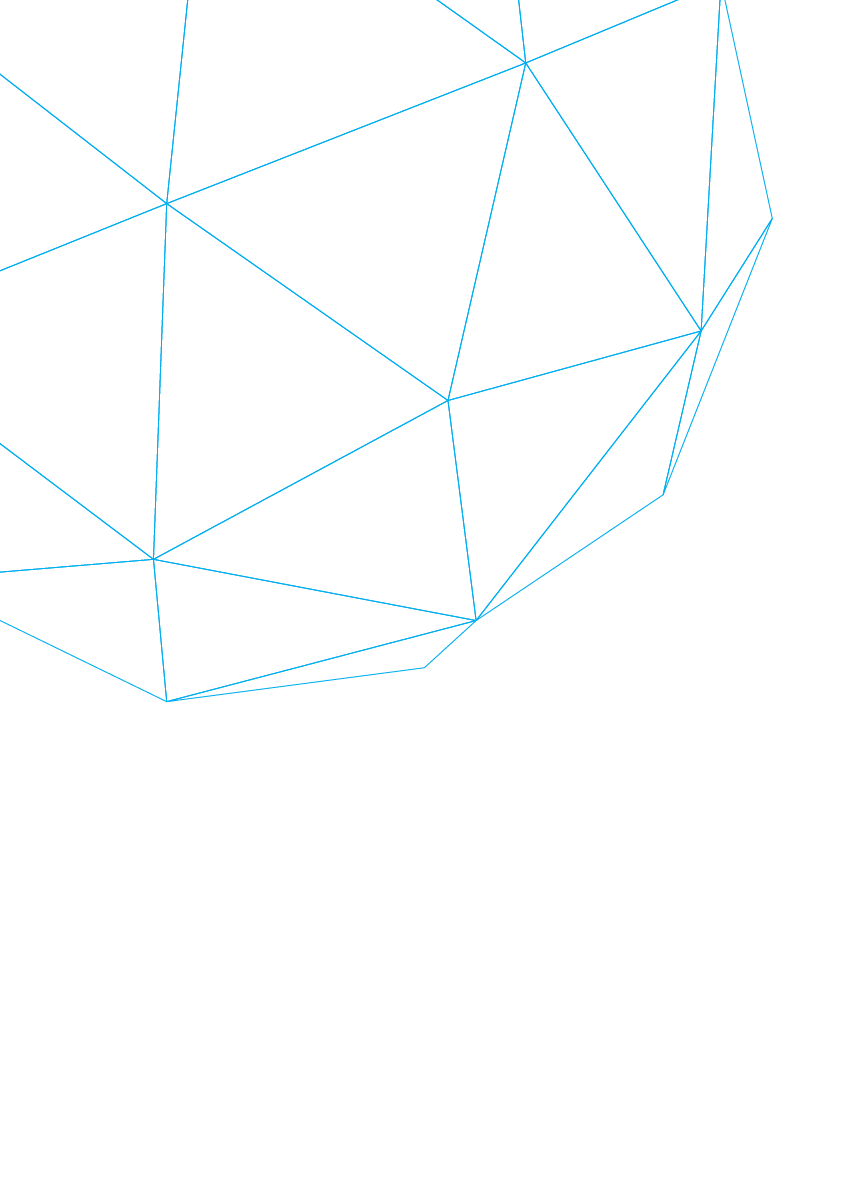


Table of contents

[Document history 3](#_Toc446512385)

[Intended readers 4](#_Toc446512386)

[Purpose of the Proxy module 4](#_Toc446512387)

[Developing a proxy module 4](#_Toc446512388)

[Generating proxy module from archetype 4](#_Toc446512389)

[Package overview 5](#_Toc446512390)

[Handle messages ( message package ) 5](#_Toc446512391)

[Proxy bean package 5](#_Toc446512392)

[The mapper package 6](#_Toc446512393)

[Mock package 6](#_Toc446512394)

[The port initiator class 6](#_Toc446512395)

[Start developing a Proxy module 6](#_Toc446512396)

[Communicating with the proxy module 7](#_Toc446512397)

[Responding to the main module 8](#_Toc446512398)

|  |  |  |
| --- | --- | --- |
| Author | Changes | Date |
| Joakim Johansson | Created document | 2016-04-14 |
|  |  |  |

# Document history

# Intended readers

This document is intended for developers wanting to develop a new proxy module that integrates to the UnionVms system. The text assumes that you as a reader is comfortable with coding and understanding Maven and the JavaEE technology.

This document can also be used as a generic description of the plugin technology for UnionVms but if the reader has no experience regarding JavaEE and Maven some parts can be hard to understand.

# Purpose of the Proxy module

The sole purpose of the database module is to act as a proxy between a main module and a database. The database module should map the interface from the modules data source contract ( Read more about this in the “Developing\_New\_Main\_Module.docx” ). The contract on the modules services are well defined and the proxy acts like a mapper between the extrernal webservice and Module

External Database

Database module

Main module

# Developing a proxy module

## Generating database module from archetype

The proxy module in already defined in a Maven Archetype located in the SVN repository at [*/unionvms-maven/archetypes/raw-archetypes/unionvms-domain-archetype* ]*.* To create a new databaseproject from the archetype do the following.

If you don´t have access to a public nexus/mvn repo with this archetype you can release it locally and create your modules from this archetype.

1. In the archetype root, open cmd and type [ mvn archetype:create-from-project ]
2. cd into target\generated-sources\archetype ( from the archetype root )
3. Type [ *mvn install* ]

Now your archetype is released to your local .m2 repository

To create a project from archetype do as follows

1. Create a new folder where you want the project to be
2. open cmd and cd to that folder
3. type [ *mvn archetype:generate -DarchetypeCatalog=local* ]
4. You will be presented with options from your local artifact repo. Chose the one that have the namespace "eu.europa.ec.fisheries.uvms.component:component-archetype"
5. Define value for property 'groupId': : eu.europa.ec.fisheries.uvms.proxy.YOUR\_COMPONENT\_NAME
6. Define value for property 'artifactId': : YOUR\_COMPONENT\_NAME
7. Define value for property 'version': 1.0-SNAPSHOT: : LEAVE\_THIS\_EMPTY ( Just hit ENTER )
8. Define value for property 'package': eu.europa.ec.fisheries.uvms.proxyYOUR\_COMPONENT\_NAME: : eu.europa.ec.fisheries.uvms.YOUR\_COMPONENT\_NAME
9. Select Y and Enter and you’re done!

Open the generated component in your ide and mvn clean build to ensure that the component is correctly configured

## \\hav.havochvatten.se\hav\root\users\jojoha\Desktop\FOLDERS\Pictures\databasePackageJPG.JPGPackage overview

The generated artifact contains all a developer needs to start developing the database module.

### Handle messages ( producer & consumer package )

All messages between the main module and the database module are sent through JMS. All messages sent to the database module is handled in the ConsumerBean.java class. The ConsumerBean.java class is an MDB that listens to the specified queue configured in the Constants.java class. The ConsumerBean har 2 events implemented that fires CDI events based on the outcome of the logic in the bean. @DomainSendEvent in a happy flow and @DomainErrorEvent in an error flow.

When returning a message back to the sender the JMSProducer.java class is used. This class listens for 2 CDI events, @DomainSendEvent and @DomainErrorEvent.

### Dao package

This package contains stub dao classes for inserting, deleting, updating etc for JPA entities.

In this class the developer can build up whatever he or she wants as a client. Some project may want to build an external JAXB client and include that as a dependency in the POM and then implement the web service interface. The developer can also create a REST interface or client. The developer has totally free hands in these classes.

All beans with business logic should end up in this package.

### The mapper package

The mapper package includes two classes out of the box. Of course more can be added but the initial two mappers are there for a reason. RequestMapper.java is intended to map the incoming objects from the requesting module and map them to the web service object on the outgoing side.

Main module

Proxy module

External webservice

Request ( Incoming object )

Map to outgoing object in RequestMapper

Request ( Outgoing request )

The same applies for the ResponseMapper.java class. The difference is that the mapper maps in the opposite direction

Main module

Proxy module

External webservice

Response ( Returning object )

Map to returning object in ResponseMapper

Response ( Incoming response )

It is important that the developer tries to use the convention proposed in this document regarding the naming of the mapper. This is because all other modules uses Response and Request in the naming and the same logic applies for all those mappers throughout the all types of modules.

### Mock package

This package and content are intended to be completely replaced or removed as it only acts as a mocking interface with DTOs SOAP port types etc.

### The port initiator class

This class is used as a onetime initiation of a SOAP port type. The intention of this class is to act as a singleton for initiation of web service connections. This is of course optional but can be used and is a part of the mocked archetype.

## Start developing a Proxy module

The first thing to decide is what the proxy module should achieve and what module it should communicate with. If you want to use the proxy module as a connection to a database this is NOT the kind of module to use. Instead use the Database module.

### Communicating with the proxy module

All communication between the main module and the proxy module is achieved through JMS. To ease the integration a well-defined contract is needed. This contract or interface is provided by the main module ( see the document “*initial reading.docx”* , the model section ). The contract that is to be used in the proxy module is the DataSourceService ( See the document *“Devloping\_New \_Main\_Module.docx”* the model section ).

The first step to integrate the main modules model is to add that import in the service EJB projects .pom file in proxy project.

For one of the proxy projects for the vessel module that import looks like below.

<dependency>

<groupId>eu.europa.ec.fisheries.uvms.asset</groupId>

<artifactId>asset-model</artifactId>

<version>${asset.model.version}</version>

<type>jar</type>

</dependency>

All main modules should have the mappers for the request and response objects to and from the main module already implemented. So the first thing is to implement the data source method you want to implement in the proxy module from the defined methods in the model package. This is done in the ProxyMessageReciever.java class.

All Main modules always inherit a base method that defines the method signature to handle in all modules that implements that interface. In the ProxyMessageReciever ( hereafter called MDB ) you first have to unmarshall the incoming message from the JMS queue. In the VesselProxy module this is done by unmarshalling the AssetDataSourceRequest. If it has been another module the Object would be called “AnothermoduleDataSourceRequest”. The JAXBMarshaller is always a part of the Main modules model package.

AssetDataSourceRequest request **=** JAXBMarshaller**.**unmarshallTextMessage**(**textMessage**,** AssetDataSourceRequest**.**class**);**

From this “BaseRequest” we can decide what method the caller wants to invoke.

In the proxy module for the code looks as follows. The method is simply an enumeration that we can do a switch case on.

**switch** **(**request**.**getMethod**())** **{**

**case** GET**:**

GetAssetRequest getRequest **=** JAXBMarshaller**.**unmarshallTextMessage**(**textMessage**,** GetAssetRequest**.**class**);**

AssetId vesselId **=** getRequest**.**getId**();**

getVessel**(**textMessage**,** vesselId**.**getType**(),** vesselId**.**getValue**());**

**break;**

**case** PING**:**

**case** CREATE**:**

**case** DELETE**:**

**case** GROUP\_CREATE**:**

**case** GROUP\_DELETE**:**

**case** GROUP\_GET**:**

**case** GROUP\_LIST**:**

**case** GROUP\_UPDATE**:**

**case** HISTORY\_GET**:**

**case** HISTORY\_LIST**:**

**case** LIST**:**

**case** LIST\_GET\_BY\_GROUP**:**

**case** UPDATE**:**

**case** UPSERT**:**

**default:**

errorEvent**.**fire**(new** ProxyResponse**(**message**,** "Method " **+** request**.**getMethod**().**name**()** **+** " not implemented!"**));**

LOG**.**error**(**"[ Error, method {} not implemented. ]"**,** request**.**getMethod**().**name**());**

**throw** **new** VesselProxyException**(**"Method " **+** request**.**getMethod**().**name**()** **+** " not implemented!"**);**

**}**

We can simply choose what methods we want to implement from the interface and simply return an error on all other non-implemented methods instantly to the requesting module.

The next step in parsing the message is to extract the actual message from the request. In the previous step we only decided what method we wanted to use. If we also want the object that we know is a part of that message we must parse it. In the example above we must know that the method “GET” should deliver an Object GetAssetRequest.

The GetAssetRequest is unmarshalled with the help of the JAXBMarshaller provided in the main modules model.

GetAssetRequest getRequest **=** JAXBMarshaller**.**unmarshallTextMessage**(**textMessage**,** GetAssetRequest**.**class**);**

As you probably noticed the “AssetDataSourceRequest” and “GetAssetRequest” uses the same unmarshaller as if they had the same root level when parsing. A request always inherits from a “Base request”. So the GetAssetRequest inherits from the AssetDataSourceRequest. All request objects should do the same in the requests for all modules in all requests.

### Responding to the main module

After the request is unmarshalled it is up to the developer to implement the logic that he or she wants to apply. When the logic is processed it is time to return the response to the main module. Then mapping to the response object should already be implemented in the imported model jar from the module. Communication via the JMS queue is always done with the jms TextMessage object. In the request that object will already be used but it is important that the developer uses that object in the response because the module will try to unparse a string value in the TextMessage object ( getText() ).

For each request, for example “GetAssetRequest” there is alwawys a object with the same name but with the response ending in the name. So for “GetVesselRequest” there is a object named “GetVesselResponse”. For that response there should be a mapper already implemented in a datasource mapper in the imported model.jar.

All mappers should be static classes so it should be easy for the developer to find the appropriate method without the need to instantiate the mapper classes.

The mapper should return a String that is a marshalled objects in an XML representation of the returning Object. See more about this in the chapter “*Sending messages on the JMS queue”* in the document “*Initial reading.“*

When the object has gone through JAXB processing and transformation to XML the marshalled object is set as the payload in the TextMessage object. When responding the ProxyMessageSender.java class should be used. The following 2 subjects are VERY important when returning a JMS message using the ProxyMessageSender.

* ReplyQueue
  + The Reply queue should be the JMSReplyTo in the TextMesage object request received in the proxy module.
* JMSCorrelationId
  + The correlationIs should be the JMSMessageId in the TextMesage object request received in the proxy module.

The reply queue is the queue that the requesting module decides it wants the answer to.

The correlation id is the id that the requesting module will listen to when waiting for the response from the proxy module. When the JMS message is initially created a JMS id is automatically created and sent with the message. That id is then used to verify that the response is correlating with the correct request.