**UnionVms**

**Developing new Plugins**

Joakim Johansson

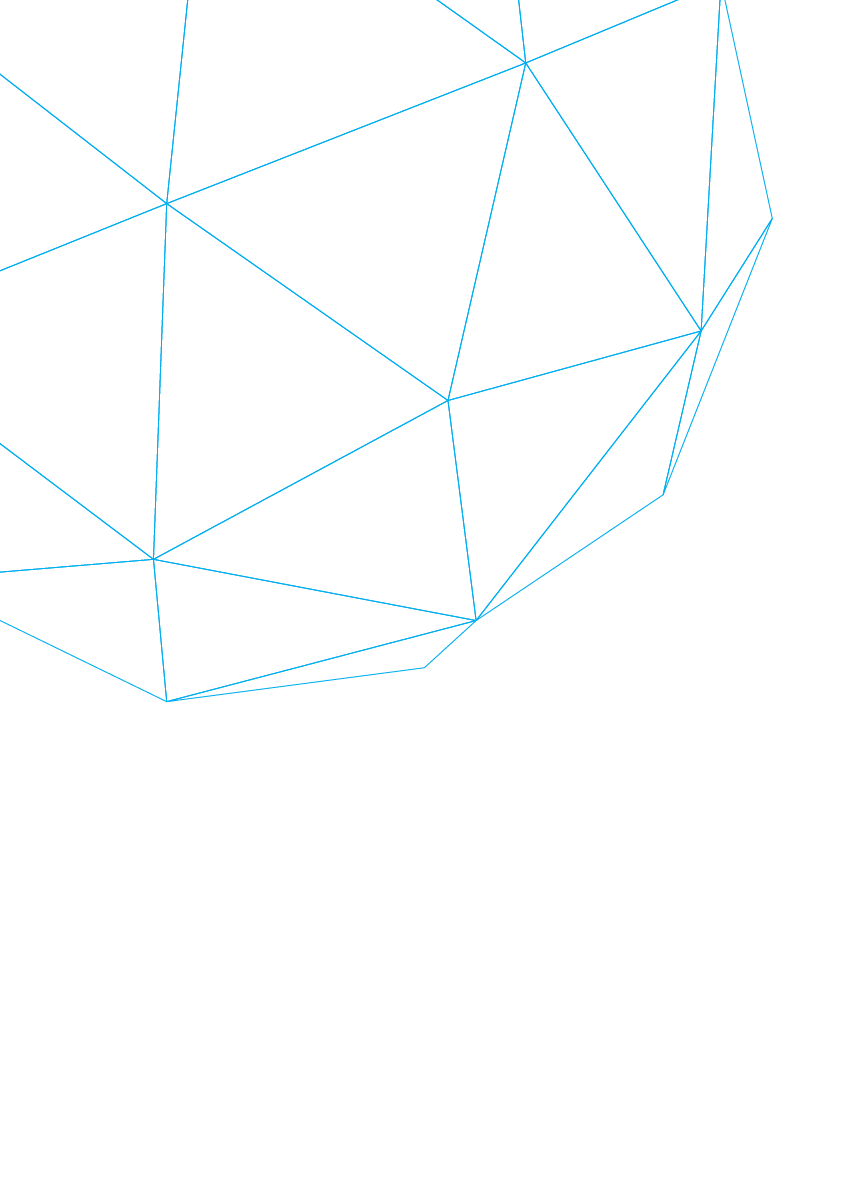


Table of contents

[Document history 3](#_Toc447034118)

[Intended readers 4](#_Toc447034119)

[Purpose of the plugin 4](#_Toc447034120)

[Developing plugins for the Exchange module 4](#_Toc447034121)

[Overview of the plugin 4](#_Toc447034122)

[Plugin configurations 5](#_Toc447034123)

|  |  |  |
| --- | --- | --- |
| Author | Changes | Date |
| Joakim Johansson | Created document | 2016-03-08 |
|  |  |  |

# Document history

# Intended readers

This document is intended for developers wanting to develop a new plugin that integrates to the UnionVms system. The text assumes that you as a reader are 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.

The reader is assumed to have read the document “Initial\_Reading” to have some basic understanding of the modular architecture of UVMS.

# Purpose of the plugin

The sole purpose of the plugin is to act as a mapper and interface between the exchange module external systems.

External system

External system

Exchange module

Plugin

Plugin

The interface on the UnionVms side is very strict and defined whilst the interface on the interacting system can be of any type. Any system should be able to interact with UnionVms therefore it is the plugin that sets the boundry for what type of communication and interfaces that UnionVms can interact with.

# Developing plugins for the Exchange module

* + This plugin type is a placeholder for a

### Out of the box features of the plugin

The Exchange module acts an interface and proxy between UnionVms and other systems that want to communicate with UnionVms. To enable modularity and flexibility a generic plugin and registry technique is implemented in the exchange module. Any developer can develop their own plugin to enable communication to and from the Exchange module.

The plugin is already defined in a maven archetype. After creating a project from the archetype a developer should able to do the following out of the box without writing a single line of code.

* Registration to the Exchange modules registry
* Set plugin configuration values
* Send position reports to the Exchange module
* Send position reports to the plugin
* Send commands to plugin
* Start the plugin
* Stop the plugin
* Ping the plugin

## Overview of the plugin

Exchange module

Registry

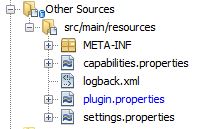
Plugin 1

Plugin 2

There are 3 separate steps in the plugin lifecycle, registry, communication and unregistering.

The Exchange module exposes an interface for a plugin to register. This procedure starts automaticly when the module is deployed in the application container. The plugin automaticly then registers to the Exchange module.

If the registration is successfull the Exchange module answers back with a OK on the plugin JMS Bus.



## Plugin configurations

The plugin has several types of configurations that will push to the exchange module when registering. These settings should be set in the correct corresponding properties file. More on the different properties files later.

### Configurations at register time

If the plugin has never registered with the exchange module all settings in the settings.properties file will be pushed to the exchange module.

If the Config module is deployed the Exchange module will push that settings to the config module. The user can then edit the plugin configuration values from the config modules GUI. The Exchange module and Config module will always be in sync regarding plugin settings.

If the plugin has already registered to the exchange module the plugin will not push its default settings on deployment. Instead the Exchange module will push the previous settings back to the registered plugin. This behavior is implemented for two reasons.

1. If the user has entered specific settings that settings will never be overwritten
2. If the system shuts down and is restarted the registration process of the plugin will start again. In that case we do not want our settings to be overwritten.

The plugin.properties file is only used as a properties file for internal use for the plugin. Plugin.properties is automatically generated when building the plugin. The logback.xml uses this file for automatic setup of the plugin logging. There are three important values in the plugin.properties file that is used when registering the plugin. All properties in the plugin.properties file are set from the pom.xml. So if you want to make changes to the pplugin.properties file you have to do them in the pom.xml, otherwise the settings will be overwritten the next time you compile the project.

* application.logfile
  + The value of this key is the path to the logfile
* application.error.logfile
  + The path to the error logfile
* application.name
  + This is the value that will be shown in the exchange module as the name of the plugin
* application.groupid
  + This is the unique identifier of the plugin. Be sure that your namespace for the plugin is unique in the system!
* application.responseTopicName
  + This value is concatenated with the value of the application.groupid. This is the name that the plugin will listen to when expecting responses on the JMS queue ( Message driven bean )

All plugins can have capabilities. The capabilities can be anything the developer chooses. The capabilities that you get out of the box in the plugin are specific for a plugin for a satellite receiver, but the key values ban be set to whatever the developer wants.

## StartupBean

The startup bean is the heart of the plugin. Startup bean is a singleton bean that holds all settings and properties for the plugin. All registering and unregistering is initiated in the Startup bean.

To use the Startup bean you have to inject it in a Java bean. In the startup bean you can access the following attributes.

|  |  |
| --- | --- |
| Name | Description |
| RegisterClassName | The unique identifier of the plugin. This is also the topic name the plugin will subscribe to in the event bus to receive messages to the plugin. |
| IsRegistered | Has the plugin successfully registered with the Exchange module |
| IsEnabled | The plugin can be “turned off\on”, this Is the attribute that tells what state the plugin is in. |
| PluginResponseSubscriptionName | When the plugin is actively waiting for a response for a request the plugin sent, this is the subscription name the plugin waits for on the message bus. |
| IsWaitingForResonse |  |
|  |  |
|  |  |

### Capabilities

The capabilities in the list below is the default capabilities. You can set any key value of your choice and they will be registered as is in the Exchange database. Capabilities are as of date ( 2016-04-19 ) only used by SATELITE\_RECEIVER plugin type. And the

|  |  |
| --- | --- |
| Name | Description |
| POLLABLE |  |
| ONLY\_SINGLE\_OCEAN |  |
| MULTIPLE\_OCEAN |  |
| CONFIGURABLE |  |
| SAMPLING |  |

### Plugin types

As of the time for this writing there are only a certain amount of plugin types you can implement. This can of course be changed and there is a ”OTHER” type that can be used if your plugin does not correspond the the current types. See ”StartupBean” chapter for more information. If you want to add more plugin types this has to be done in the interface ( plugin.xsd ) in the Exchange model.

* EMAIL
  + This is a plugin type that can send and recieve email.
* SATELLITE\_RECEIVER
  + This is a plugin that handles position reports ( movements ) to and from the Exhange module.
* FLUX
  + This type of plugin use the EU Commission FLUX network to send and recieve position reports.
* MANUAL
  + This type only acts as a placeholder and no plugins are imlemented or should be implemented using this type! This plugin type exists solely because the exhange module needs a placeholder for a plugin when a user creates a manual position report from the GUI.
* OTHER
  + If you cannot determine the type of your plugin and don´t want to add it to the model this is the type you should use.
* NAF
  + This type of plugin should use the NAF protocol to send and recieve position reports