EUROPEAN COMMISSION

DIGIT
Connecting Europe Facility

# Domibus 3.2.4

# Plugin Cookbook

## Usage and implementation manual

Date: 10/04/2017

Document Approver(s):

| Approver Name | Role |
|---|---|
| Adrien FERIAL | CEF eDelivery Technical team leader |
| | |
| | |

Document Reviewers:

| Reviewer Name | Role |
|---|---|
| Federico MARTINI | Software Developer |
| | |
| | |

Summary of Changes:

| Version | Date | Created by | Short Description of Changes |
|---|---|---|---|
| 0.1 | 21.06.2016 | Christian KOCH | Initial version |
| 0.2 | 14.07.2016 | Christian KOCH | Changes according to comments |
| 0.3 | 21.07.2016 | Christian KOCH | Added link to administration guide |
| 0.4 | 29.07.2016 | Christian KOCH | Changes according to comments |
| 0.5 | 11.08.2016 | Christian KOCH | Changes according to comments |
| 1.0 | 11.08.2016 | Cosmin BACIU | Describe how perform the validation of the submission. First version published |
| 1.1 | 15.09.2016 | Cosmin BACIU | Updated the document for 3.2-RC1 |
| 1.2 | 04.10.2016 | Ioana DRAGUSANU | Added authentication details for 3.2.0 |
| 1.3 | 10.10.2016 | Adrien FERIAL | Finalization of the document |
| 1.4 | 19.01.2017 | Cosmin BACIU | Documented the new *messageReceiveFailed* method |
| 1.5 | 22.03.2017 | Cosmin BACIU | Upgrade the version to 3.2.3 |
| 1.6 | 10.04.2017 | Cosmin BACIU | Upgrade the version to 3.2.4 |

# Table of Contents

# 1. INTRODUCTION

This document describes the Domibus plugin architecture and plugin API

## 1.1. Objectives

After reading this document the reader should be aware of the capabilities provided by the Domibus plugin system. Additionally a developer familiar with the AS4 protocol will be able to implement a plugin integrating an existing back office application into Domibus

## 1.2. Users

This document is intended for:

- The Directorate Generals and Services of the European Commission, Member States (MS) and also companies of the private sector wanting to set up a connection between their backend system and the Access Point. In particular:
    - Business Architects will find it useful for determining how to best exploit the Access Point to create a fully-fledged solution.
    - Analysts will find it useful to understand the Use-Cases of the Access Point.
    - Architects will find it useful as a starting point for connecting a Back-Office system to the Access Point.
    - Developers will find it essential as a basis of their development concerning the Access Point services.
    - Testers can use this document in order to test the use cases described.

# 2. Backend Integration

## 2.1. General Overview

The purpose of Domibus is to facilitate B2B communication. To achieve this goal it provides a very flexible plugin model which allows the integration with nearly all back office applications.

There are two default plugins available with the Domibus distribution, the domibus-default-jms-plugin and the domibus-default-ws-plugin. Further documentation for those plugins can be found at https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus

## 2.2. Plugin Structure

A plugin is only dependent on the *domibus-plugin-api* module which is released together with the main Domibus application. Any changes to previous API versions will be addressed in a migration guide. While the default plugins use the Spring framework [https://spring.io/] for dependency injection this is not mandatory.

A plugin consists of implementations of at least two interfaces and one abstract class. The extension of eu.domibus.plugin.AbstractBackendConnector and an implementation of both eu.domibus.plugin.transformer.MessageSubmissionTransformer and eu.domibus.plugin.transformer.MessageRetrievalTransformer.

This way multiple plugins can share the same data formats while using different transport protocols or enforcing different security policies. It also is possible to implement transport handlers for protocols while keeping the actual data format pluggable as those classes are not necessarily coupled and can be reused independently from each other.

## 2.3. Message Flow

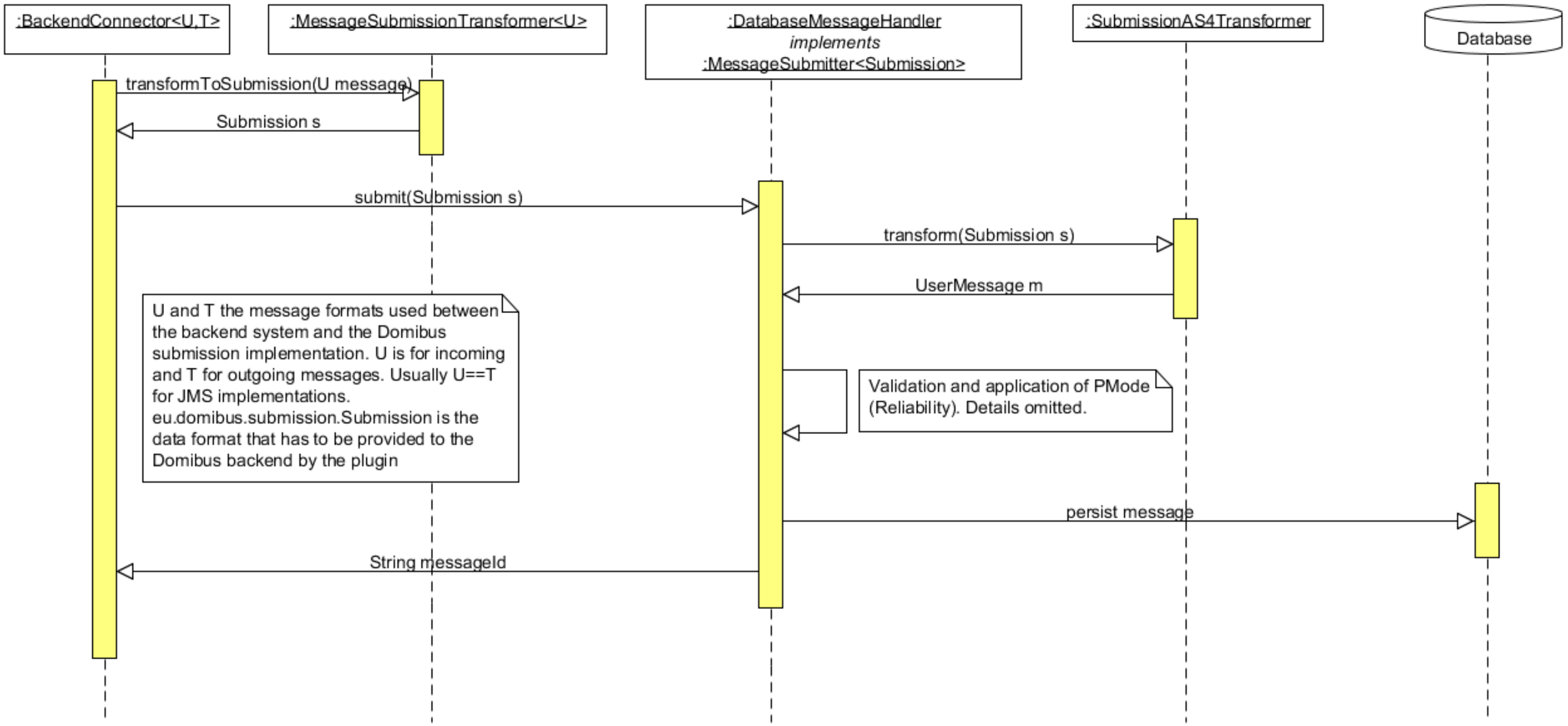



**Figure 1: Message Submission from the back end**

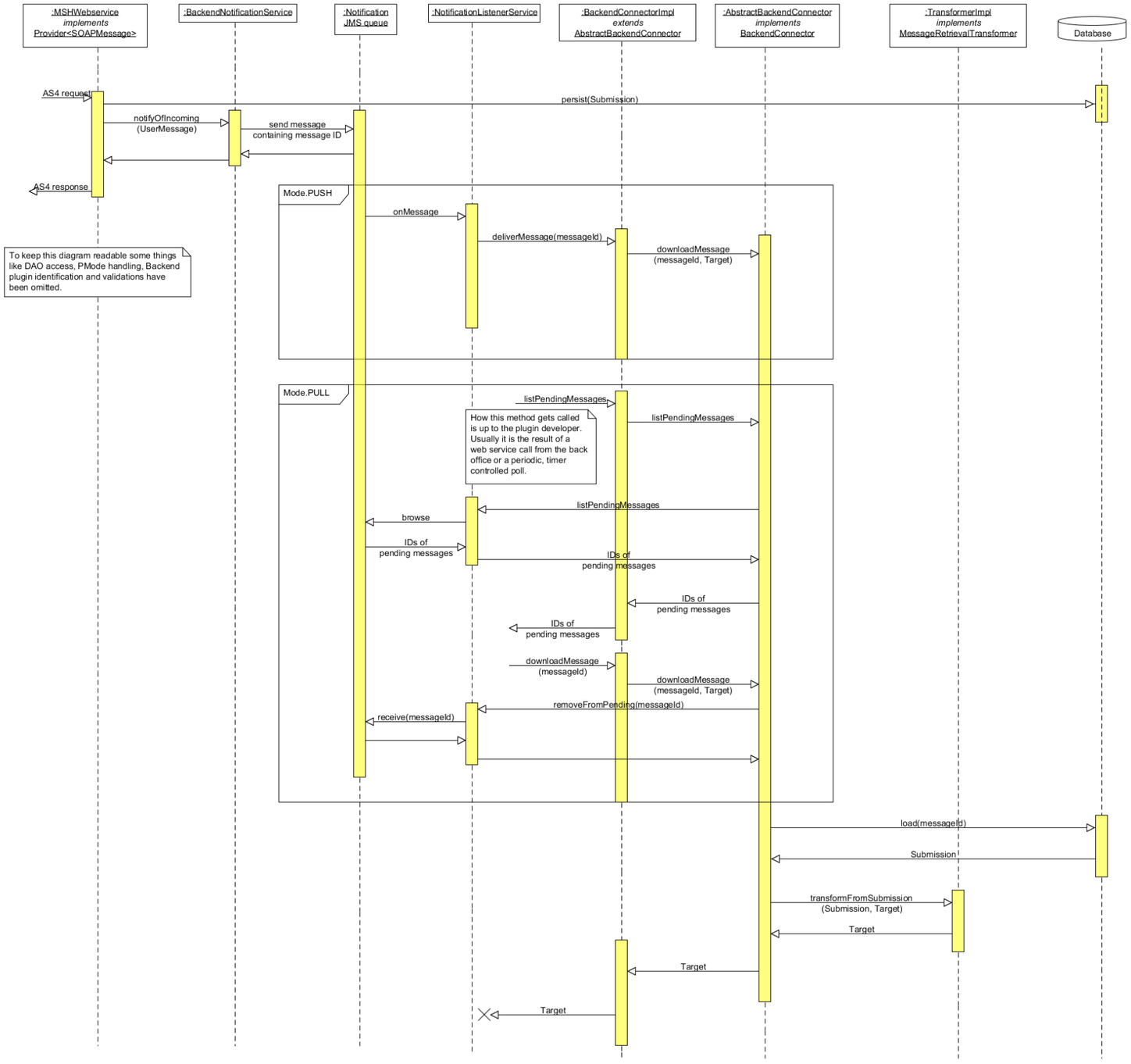



**Figure 2: Message reception by the back end and delivery to the plugin (PUSH/PULL mode)**

# 3. Implementing a Plugin

## 3.1. Pull and Push plugins

There are two different ways of communicating with the back end. The first one is *eu.domibus.plugin.BackendConnector.Mode.PULL*. A plugin operation under this mode initiates all communications with *Domibus MSH* by itself and is never called from *Domibus* itself. This mode is intended mostly for backend applications that are not always online (e.g. mobile devices).

The default *Webservice* plugin bundled with *Domibus* is an example of such a plugin. The major disadvantage of this mode is that there is no way for *Domibus* to report processing errors back to the plugin, relying on calls to *getMessageStatus(java.lang.String)* by the back end to be informed about any error conditions.

The preferred way of implementing a plugin is *eu.domibus.plugin.BackendConnector.Mode.PUSH* This mode allows Domibus to push notifications and incoming messages to the backend. In case the backend is not reachable the notification will be retried according to the backend queue definition as described in Section B.3, “Plugin configuration and deployment”. The default bundled JMS plugin is an example of such a plugin.

## 3.2. Extending eu.domibus.plugin.AbstractBackendConnector

*eu.domibus.plugin.AbstractBackendConnector* provides implementations of most methods defined in *eu.domibus.plugin.BackendConnector. eu.domibus.plugin.AbstractBackendConnector* should be used as basis for every plugin.

### *3.2.1. eu.domibus.plugin.BackendConnector.Mode.PULL*

- getMessageSubmissionTransformer()
- getMessageRetrievalTransformer()

To submit a message to the MSH the *submit(U)* implementation of *eu.domibus.plugin.AbstractBackendConnector* should be used.

To download a message a combination of *listPendingMessages()* and *downloadMessage(java.lang.String, T)* should be used.

### *3.2.2. eu.domibus.plugin.BackendConnector.Mode.PUSH*

- getMessageSubmissionTransformer()
- getMessageRetrievalTransformer()
- deliverMessage(java.lang.String)

- messageReceiveFailed(java.lang.String, java.lang.String) – in 3.2.2 this method has been deprecated

- messageReceiveFailed (eu.domibus.common.MessageReceiveFailureEvent)

- messageSendFailed(java.lang.String)

- messageSendSuccess(java.lang.String)

Additionally *listPendingMessages()* is only callable from *Mode.PULL* plugins. To submit a message to the MSH the *submit(U)* implementation of *eu.domibus.plugin.AbstractBackendConnector* should be used. Additional details on these methods can be found in the Javadoc (see chapter 5).

## 3.3. Implementing eu.domibus.plugin.transformer.MessageSubmissionTransformer and eu.domibus.plugin.transformer.MessageRetrievalTransformer

The implementations of the transformer classes are responsible for transformation between the native backend formats and *eu.domibus.plugin.Submission*. As there are two different interfaces to implement it is possible to use different DTOs for message submission and reception. This is convenient when those tasks are handled by different backend applications.

As *eu.domibus.plugin.Submission* is able to represent all kinds of messages there are many parameters that must be set, with some of them unknown to the backend application. One approach is to statically set those values in the transformer classes. Another, more flexible approach is the usage of overridable default settings as used in the bundled default JMS plugin. For further details, see the documentation and implementation of the default JMS plugin.

## 3.4. Validation of the submission

There are uses cases when it is required that the Submission object is validated before it is being delivered to the plugin. For instance, the user might want to verify that one of all the payloads is valid against a custom XSD schema. In this case, it does not make sense to deliver the message to the plugin for processing if it is not valid.

In order to better understand why the current API is not sufficient for this use case we have to understand first how the Submission object is delivered to the plugin for processing.

There are two transactions involved in the Submission processing:

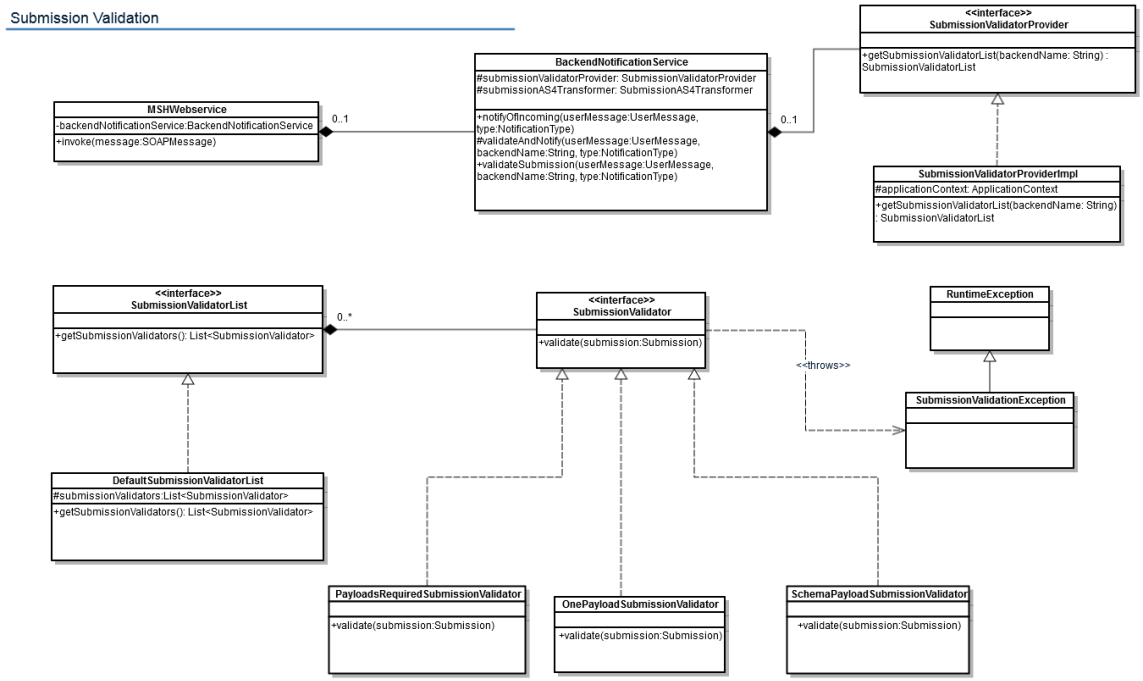
1. In the first transaction the message is stored in the database and a signal is sent internally via JMS to trigger the Submission processing.

2. A JMS listener is listening to Submission processing events and triggers the processing.

If the Submission is validated in the second step it would be too late because the Submission has been already saved and accepted for processing in the first step. This is the reason why we need to perform the Submission validation in the first step. If the Submission is not valid an exception will be raised and the processing will not be performed.

The API for Submission validation can be found in the plugin API under the package *eu.domibus.plugin.validation*

Hereunder you can find the class diagram of the classes involved in the submission validation.




In order to validate the Submission object one has to declare in the plugin Spring context a bean of type *eu.domibus.plugin.validation.SubmissionValidatorList* and the bean id needs to contain the plugin name. The core will automatically discover the bean of type *SubmissionValidatorList* and perform the validation by calling the *validate* method on each *SubmissionValidator* configured.

In the plugin API there is already a default implementation of the *SubmissionValidatorList* interface *DefaultSubmissionValidatorList* that has an *java.util.ArrayList* for maintaining the list of validators.

By default Domibus comes with 3 implementations of the *SubmissionValidator* interface. An example how to use them can be found in the next paragraph.

1. *eu.domibus.submission.validation.OnePayloadSubmissionValidator* – validates that there is at least one payload present in the Submission

2. *eu.domibus.submission.validation.PayloadsRequiredSubmissionValidator* – validates that there is only one payload present in the Submission

3. *eu.domibus.submission.validation.SchemaPayloadSubmissionValidator* – validates that the payloads are valid against a custom XSD schema

For example, below there is an extract of a custom plugin Spring context where we can see that a custom validator has been implemented and there are 3 validators used to validate the Submission:

```
<!-- custom validator -->
<bean id="customValidator"
class="eu.domibus.submission.validation.CustomSubmissionValidator"/>


<bean id="customJaxbContext" class="javax.xml.bind.JAXBContext" factory-
method="newInstance">
    <constructor-arg type="java.lang.String"
                     value="eu.domibus.plugin.custom.domain"/>
</bean>
<!-- schema validator -->
<bean id="customPayloadSchemaValidator"
class="eu.domibus.submission.validation.SchemaPayloadSubmissionValidator">
    <property name="jaxbContext" ref="customJaxbContext"/>
    <property name="schema" value="classpath:xsd/as4Payload.xsd"/>
</bean>

<!-- validators list -->
<bean id="customSubmissionValidatorList"
class="eu.domibus.plugin.validation.DefaultSubmissionValidatorList">
    <property name="submissionValidators">
        <list>
            <ref bean="onePayloadSubmissionValidator"/>
            <ref bean="customValidator"/>
            <ref bean="customPayloadSchemaValidator"/>
        </list>
    </property>
</bean>
```

## 3.5. Plugin Authorisation

Domibus can be configured to require authorisation by setting the following property to false in the config file *domibus-configuration.xml:*

*<prop key="domibus.auth.unsecureLoginAllowed">false</prop>*

The authorisation is performed at method level by using Spring, @PreAuthorize annotation.

There are two roles defined in the application, ROLE_ADMIN and ROLE_USER

```
@PreAuthorize("hasAnyRole('ROLE_USER', 'ROLE_ADMIN')")
public void hasUserOrAdminRole() {}

@PreAuthorize("hasAnyRole('ROLE_ADMIN')")
public void hasAdminRole() {}
```

**ROLE_ADMIN** has the right to call:

- *sendMessage* with any value for originalSender property
- *downloadMessage* (any message among messages notified to this plugin)
- *listPendingMessages* will list all pending messages for this plugin
- *getMessageStatus* and *getMessageErrors*

**ROLE_USER** has the right to call:

- *sendMessage* with originalSender equal to the originalUser
- *downloadMessage*, only if finalRecipient equals the originalUser
- *listPendingMessages*, pending messages filtered by the finalRecipient (equal to the originalUser)

The authentication object implements *org.springframework.security.core.Authentication* and is expected in the security context holder:

```
SecurityContextHolder.getContext().setAuthentication(authentication)
```

It is required that the method *getPrincipal()* of the authentication object returns the value of the original user which represents the authenticated end user (matching the "originalSender" or "finalRecipient" in the message properties).

## 3.6. WS plugin authentication example

The WS plugin provides an example of how to implement the authentication in the plugin. It supports:

- Basic Authentication
- X509Certificates Authentication
- Blue Coat Authentication

Note: Blue Coat is the name of the reverse proxy at the EC. It forwards the request in HTTP with the certificate details inside the request ("Client-Cert" header key).

The authentication is performed through a custom interceptor named `CustomAuthenticationInterceptor`. This interceptor is configured to intercept incoming requests on /backend interface, in the `ws-plugin.xml`:

```
<jaxws:endpoint id="backendInterfaceEndpoint"
implementor="#backendWebservice" address="/backend">
        …
    <jaxws:inInterceptors>
        <ref bean="customAuthenticationInterceptor"/>
    </jaxws:inInterceptors>
</jaxws:endpoint>
```

The interceptor will then call a custom authentication provider depending on the authentication type in the request.

Basic Authentication takes precedence on both **http** and **https**.

X509Certificates is expected on **https** when no Basic Authentication was found.

Blue Coat certificates are expected on **http** when no Basic Authentication was found.

For convenience reasons, the WS plugin uses, to store the users/passwords and certificate ids, exactly the same database as configured in Domibus core.

Two users are already inserted in the DB (TB_AUTHENTICATION_ENTRY table), "admin" and "user" both with passwords "123456". You need to change them for security reasons.

"admin" has the role ROLE_ADMIN and "user" has the role ROLE_USER, matching the originalUser "urn:oasis:names:tc:ebcore:partyid-type:unregistered:C1"

**Note: Other plugins may use their own custom authentication providers and perform different authentication as long as the SecurityContextHolder is set correctly as described in the "3.5 Plugin Authorisation" section.**

# 4. PLUGIN CONFIGURATION AND DEPLOYMENT

The documentation for configuration of the message routing and plugin deployment for all supported deployment platforms can be found in the administration guide. It can be downloaded from the release page of Domibus, section Documentation:
https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus

# 5. API Documentation

Standard Javadoc documentation for the API can be downloaded at https://ec.europa.eu/cefdigital/artifact/service/local/repositories/eDelivery/content/eu/domibus/domibus-plugin-api/3.2.4/domibus-plugin-api-3.2.4-javadoc.jar.

This documentation includes all necessary information required to implement the necessary methods.

# 6. CONTACT INFORMATION

CEF Support Team

By email: CEF-EDELIVERY-SUPPORT@ec.europa.eu

By phone: +32 2 299 09 09

- Standard Service: 8am to 6pm (Normal EC working Days)
- Standby Service*: 6pm to 8am (Commission and Public Holidays, Weekends)

* *Only for critical and urgent incidents and only by phone*