EUROPEAN COMMISSION

DIGIT
Connecting Europe Facility

Access Point

Domibus 4.0.2

Plugin Cookbook

Version [2.3]

Status [Final]

© European Union, 2018

Reuse of this document is authorised provided the source is acknowledged. The Commission's reuse policy is implemented by Commission Decision 2011/833/EU of 12 December 2011 on the reuse of Commission documents.

Date: 11/02/2019

Document Approver(s):

Approver Name	Role
Adrien FERIAL	CEF eDelivery Technical team leader

Document Reviewers:

Reviewer Name	Role
Federico MARTINI	Software Developer
Caroline AEBY and Chaouki BERRAH	Technical Writers

Summary of Changes:

Version	Date	Created by	Short Description of Changes
0.01	21.06.2016	Christian KOCH	Initial version
0.02	14.07.2016	Christian KOCH	Changes according to comments
0.03	21.07.2016	Christian KOCH	Added link to administration guide
0.04	29.07.2016	Christian KOCH	Changes according to comments
0.05	11.08.2016	Christian KOCH	Changes according to comments
1.00	11.08.2016	Cosmin BACIU	Describe how perform the validation of the submission. First version published
1.01	15.09.2016	Cosmin BACIU	Updated the document for 3.2-RC1
1.02	04.10.2016	Ioana DRAGUSANU	Added authentication details for 3.2.0
1.03	10.10.2016	Adrien FERIAL	Finalization of the document
1.04	19.01.2017	Cosmin BACIU	Documented the new *messageReceiveFailed* method
1.05	22.03.2017	Cosmin BACIU	Upgrade the version to 3.2.3
1.06	10.04.2017	Cosmin BACIU	Upgrade the version to 3.2.4
1.07	22/05/2017	Yves ADAM	Upgrade the version to 3.2.5
1.08	20/06/2017	Ioana DRAGUSANU, Cosmin BACIU	Add 3.3-RC1 changes, ext-services, domibus-logging, getStatus API
1.09	03/10/2017	Cosmin BACIU	Documented the new messageStatusChanged method
2.00	05/10/2014	Caroline AEBY	Document review and corrections.
2.01	09/10/2017	CEF Support	List of reviewers updated.
2.02	24/11/2017	CEF Support	Update with Domibus 3.3.1
2.03	08/02/2018	CEF Support	Update with Domibus 3.3.2
2.04	20/03/2018	CEF Support	Reuse notice added, update with Domibus 3.3.3
2.05	16/04/2018	CEF Support	Update with Domibus 3.3.4
2.06	01/08/2018	Caroline AEBY	Updated for Domibus 4.0. Added info on notification listener. Plugin services documented.
2.07	04/09/2018	Caroline AEBY	Removed references to release candidate version
1. 2.08	2. 07/09/2018	3. Cosmin BACIU	4. Updated for multi-tenancy

2.09	14/09/2008	Chaouki BERRAH	Minor Update from Ioana DRAGUSANU
2.10	26/09/2018	Caroline AEBY	Contact information update
2.20	27/11/2018	Caroline AEBY	Domibus 4.0.1 update
2.3	11/02/2019	Caroline AEBY	Domibus 4.0.2 update

Table of Contents

1. INTRODUCTION 4
1.1. Purpose 4
1.2. Users 4
2. BACKEND INTEGRATION 5
2.1. General Overview 5
2.2. Plugin Structure 5
2.3. Message Flow 6
3. IMPLEMENTING A PLUGIN 8
3.1. Pull and Push plugins 8
3.2. Extending eu.domibus.plugin.AbstractBackendConnector 8
3.2.1. eu.domibus.plugin.BackendConnector.Mode.PULL 8
3.2.2. eu.domibus.plugin.BackendConnector.Mode.PUSH 8
3.3. Implementing eu.domibus.plugin.transformer.MessageSubmissionTransformer and eu.domibus.plugin.transformer.MessageRetrievalTransformer 9
3.4. Notification Listener 9
3.5. Validation of the submission 10
3.6. Plugin Security 13
3.6.1. Authentication 13
3.6.2. Authorization 14
3.7. Logging 15
3.8. Plugin Services 15
3.8.1. Message acknowledgement service 15
3.8.2. Monitoring service 16
4. PLUGIN CONFIGURATION AND DEPLOYMENT 17
5. API DOCUMENTATION 18
6. LIST OF FIGURES 19
7. LIST OF TABLES 19
8. CONTACT INFORMATION 20

1. Introduction

This document describes the Domibus plugin architecture and plugin API.

1.1. Purpose

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 three default plugins available with the Domibus distribution:

- the domibus-default-jms-plugin
- the domibus-default-ws-plugin.
- The domibus-default-fs-plugin.

Further documentation about those plugins can be found at https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus .

2.2. Plugin Structure

A plugin is 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.

In addition to this required module, another module is available (more information about this module can be found in the **Domibus Software Architecture Document**[1]):

- ***domibus-logging***: may be used to maintain a uniform logging style with the Domibus core.

A plugin consists of the implementation of at least two interfaces, eu.domibus.plugin.transformer.MessageSubmissionTransformer and eu.domibus.plugin.transformer.MessageRetrievalTransformer, and the extension of one abstract class, eu.domibus.plugin.AbstractBackendConnector.

This way multiple plugins can share the same data formats while using different transport protocols or enforcing different security policies. It is also 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.

[1] The document can be downloaded at https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus in the documentation section.

2.3. Message Flow

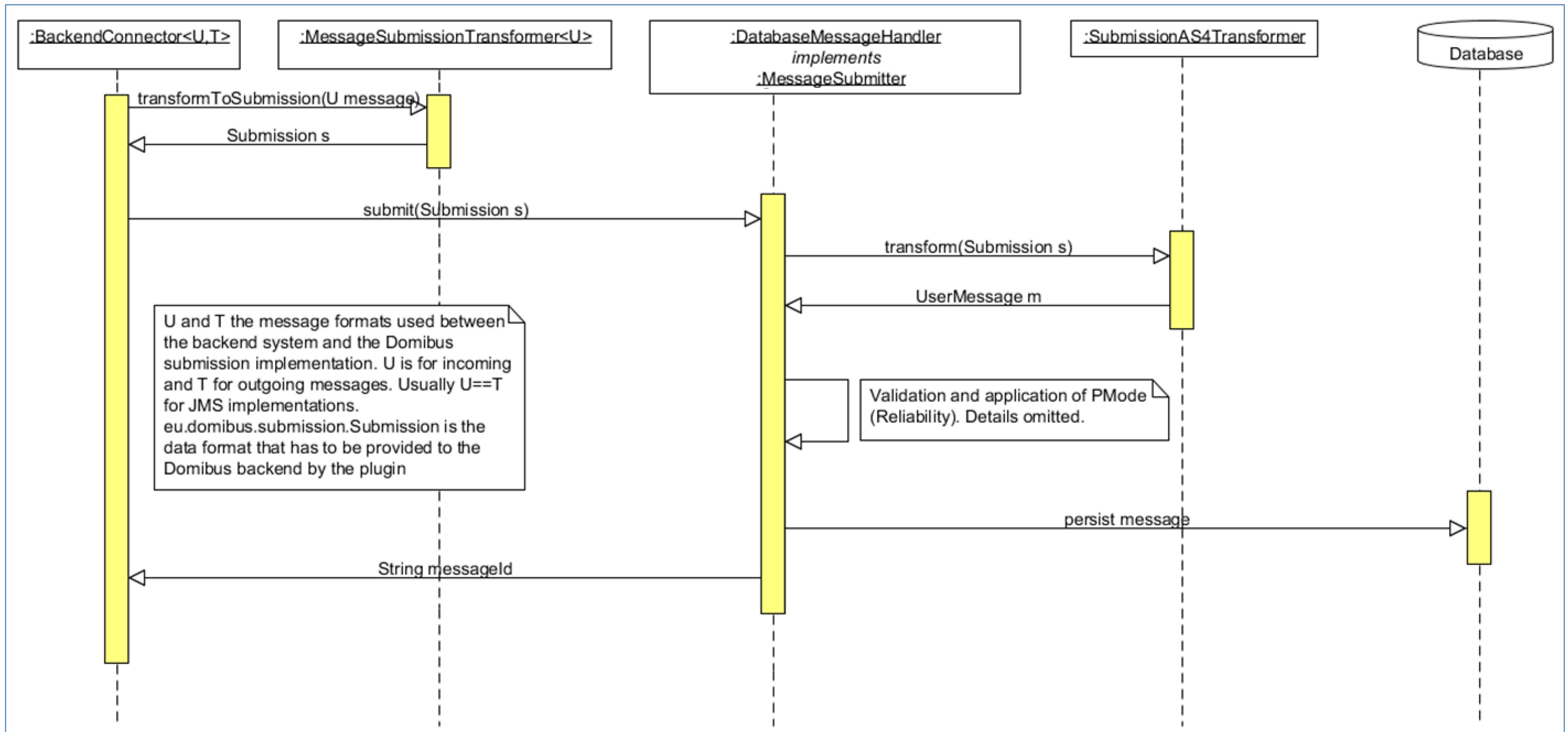

Figure 1 - Message Submission from the backend

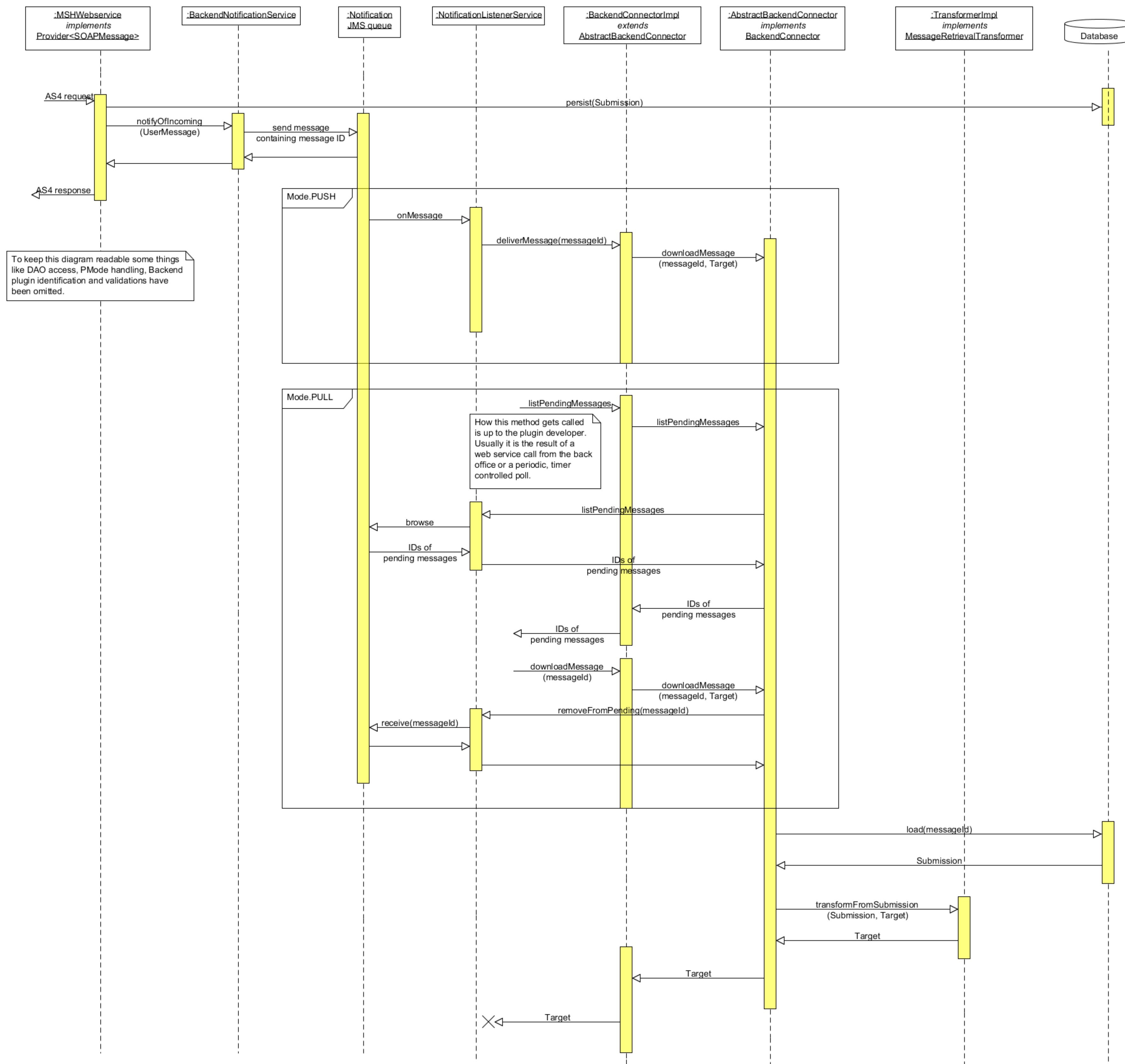

Figure 2 - Message reception by the backend and delivery to the plugin (PUSH/PULL mode)

3. Implementing a Plugin

3.1. Pull and Push plugins

There are two different ways of communicating with the backend. 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 *getStatus(java.lang.String)* by the backend 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(eu.domibus.common.MessageReceiveFailureEvent)
- messageSendFailed(java.lang.String)
- messageSendSuccess(java.lang.String)
- messageStatusChanged(eu.domibus.common.MessageStatusChangeEvent event)

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. Notification Listener

Domibus core sends notifications to the plugins on the occurrence of different events via the backend notification queue. For example, the WebService plugin receives notifications on the jms/domibus.notification.webservice queue.

The Notification Types are:

```
public enum NotificationType {
   MESSAGE_RECEIVED, MESSAGE_SEND_FAILURE, MESSAGE_RECEIVED_FAILURE,
MESSAGE_SEND_SUCCESS, MESSAGE_STATUS_CHANGE;
}
```

Each plugin may configure its own list of notification types for which it expects to be notified. This list is optional. By default, plugins that use PULL mode receive notifications for ***MESSAGE_RECEIVED***, ***MESSAGE_SEND_FAILURE***, ***MESSAGE_RECEIVED_FAILURE*** while the PUSH plugins receive notification for all.

This list is passed as a constructor to the notification listener bean, in the *-plugin.xml file.

Example:

```
<util:list id="requiredNotificationsList" value-type="eu.domibus.common.NotificationType">

    <value>MESSAGE_RECEIVED</value>

    <value>MESSAGE_SEND_FAILURE</value>

    <value>MESSAGE_STATUS_CHANGE</value>

</util:list>
```

```
<bean id="webserviceNotificationListenerService"

    class="eu.domibus.plugin.NotificationListenerService"

    c:queue-ref="notifyBackendWebServiceQueue" c:mode="PULL"

    p:backendConnector-ref="backendWebservice"/>

    p:backendConnector-ref="backendWebservice">

  <constructor-arg ref="requiredNotificationsList"/>

</bean>
```

3.5. 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:

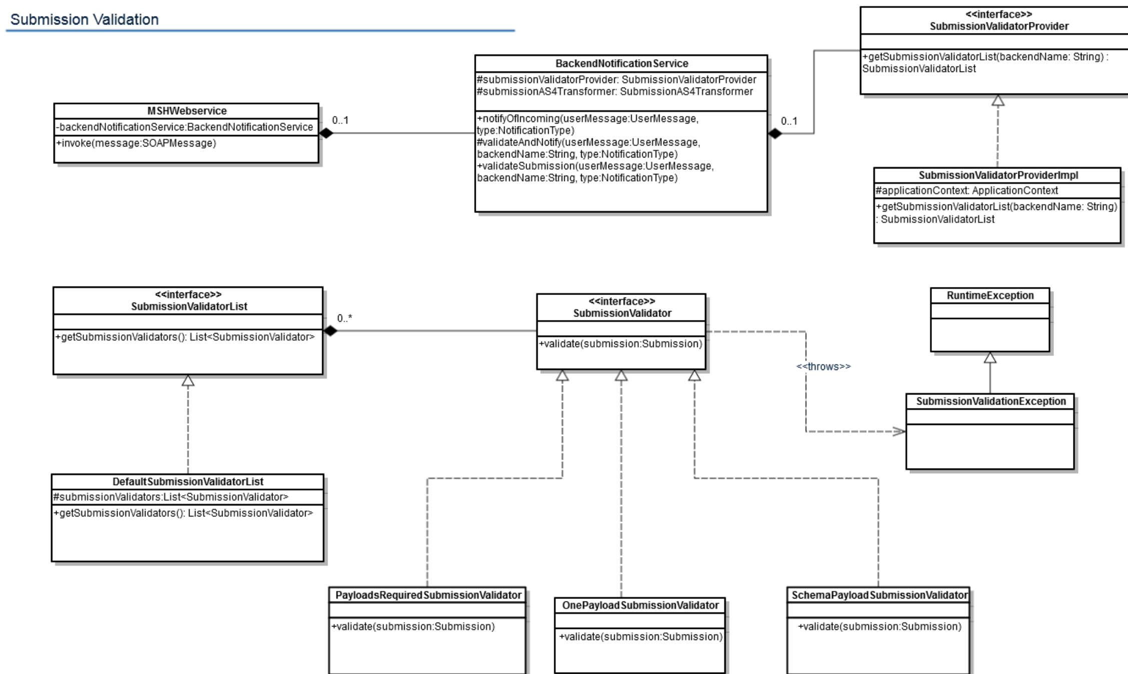

Figure 3 – Submission validation class diagram

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.* 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 configured *SubmissionValidator*.

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

Below 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>
```

```xml
<!-- 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>
```

3.6. Plugin Security

By default the plugins security is disabled. Domibus can be configured to require authorization by setting the following property to **false** in the ***domibus.properties*** configuration file*:*

```
domibus.auth.unsecureLoginAllowed=false
```

Once the plugins security is activated, all the methods of the **eu.domibus.plugin.AbstractBackendConnector** class can only be called by authenticated users.

3.6.1. Authentication

The service **eu.domibus.ext.services.AuthenticationExtService** provided in the plugin API can be used by the plugins to authenticate the request.

It provides to the plugins two Java methods to authenticate:

1. **authenticate (HttpServletRequest httpRequest) throws AuthenticationExtException**

   This method supports the following authentication types:

   - 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 **authenticate** method evaluates the 3 supported authentication methods in the following order: Basic Authentication, X509Certificates, Blue Coat certificates. The first authentication method found will execute, and the remaining authentication methods will not be evaluated anymore.

2. **basicAuthenticate(String username, String password) throws AuthenticationExtException;**

   This method supports only basic authentication.

All the users configured in the **Plugin User** UI page can authenticate and call any operation of the **eu.domibus.plugin.AbstractBackendConnector**.

By default there are two plugin users defined:

- "**admin**" has the role ROLE_ADMIN
- "**user"** has the role ROLE_USER, configured with Original User "**urn:oasis:names:tc:ebcore:partyid-type:unregistered:C1**"

Custom plugins may use their own custom authentication providers and perform different types of authentication. In case of custom authentication, the Spring **SecurityContextHolder** has to set correctly the **authentication** parameter after a successful authentication:

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

It is mandatory that the method **getPrincipal()** of the **authentication** parameter set above returns the original user value associated to the authenticated user. This original user value is used to authorize the user to a specific message. More information on how it is implemented can be found in **3.6.2 Authorization.**

3.6.2. Authorization

The authorization for the method defined in **eu.domibus.plugin.AbstractBackendConnector** is performed at Java method level using Spring **@PreAuthorize** annotation.

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

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

There are three roles defined for the plugin users **ROLE_AP_ADMIN, ROLE_ADMIN** and **ROLE_USER**, described below.

A user with role **ROLE_AP_ADMIN** or **ROLE_ADMIN** has the right to call the following methods of **eu.domibus.plugin.AbstractBackendConnector**:

- ***submit***
- ***downloadMessage***
- ***listPendingMessages***
- ***getStatus***
- ***getMessageErrors***

A user with role **ROLE_USER** associated to an Original User has the right to call the following methods of **eu.domibus.plugin.AbstractBackendConnector**:

- ***submit*** when the value of the **originalSender** from the submitted message is equal to the Original User of the authenticated user
- ***downloadMessage***, only if the **finalRecipient** value from the message to be downloaded is equal the Original User of the authenticated user
- ***listPendingMessages***, pending messages for which the **finalRecipient** value is equal to the Original User of the authenticated user
- ***getStatus*** and getMessageErrors when the value of the **originalSender** or **finalRecipient** of the message is equal to the Original User of the authenticated user**.**

3.7. Logging

The logging service is provided in the ***domibus-logging*** module, which is released together with the main Domibus application. More information about ***domibus-logging*** module can be found in the **Domibus Software Architecture Document** (the document can be downloaded at https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus in the documentation section).

Example of use:

private static final DomibusLogger LOG = DomibusLoggerFactory.getLogger(BackendWebServiceImpl.class);

3.8. Plugin Services

The Plugin API offers several services like monitoring or message acknowledgment which are described below.

These services can be accessed in two ways:

- Java API

It can be used by the plugin implementers of the custom plugins.

- REST interface

The REST interface can be used directly by the C1/C4 backends if the network configuration allows it.

The documentation of the REST interface can be found on CEF Digital site.

3.8.1. Message acknowledgement service

This service is used to acknowledge when a message is:

- delivered from C3 to the backend
- processed by the backend

Here are the typical use cases for using the *MessageAcknowledgementService*:

- a message is received by C3 from C2: the plugin that handles the message registers an acknowledgment before delivering the message to the backend
- a message is processed by the backend and it informs C3 via the plugin; the plugin registers an acknowledgment that the message has been processed by the backend
- a message is processed by the backend and informs C3 directly via the REST service exposed by the core; a REST service is exposed containing the same signature as {@link MessageAcknowledgeService}

There are two ways of performing message acknowledgments between C3 and the backend:

- synchronous

C3(via the plugin) notifies the backend synchronously and the backend process the messages also synchronously. In this case, there is no need for the backend to send a separate message acknowledgement so the plugin at the C3 side registers the processing of the message by the backend.

```
Eg:
BackendResponse backendResponse = plugin.callBackendWS(message)
messageAcknowledgeService.acknowledgeMessageDelivered(message.getId(), new
Timestamp(System.currentTimeMillis()))
messageAcknowledgeService.acknowledgeMessageProcessed(message.getId(), new
Timestamp(System.currentTimeMillis()))
```

- asynchronous

C3 notifies the backend synchronously and the backend process the messages asynchronously. In this case, the backend will send a separate message acknowledgement when it manages to process the message successfully.

```
Eg:
plugin.sendMessageToTheBackend(message)
messageAcknowledgeService.acknowledgeMessageDelivered(message.getId(), new
Timestamp(System.currentTimeMillis()))
```

3.8.2. Monitoring service

This service is used to monitor failed messages and to restore them if necessary.

Assuming that "failed message" means failed to be sent by the sender access point and getting the status set to SEND_FAILURE, the service gives the possibility to:

- list all the failed messages
- restore a failed message
- restore all messages failed during a specific period
- know how long time a message has been failed
- get the history of all delivery attempts
- delete the message payload of a failed message

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/content/repositories/public/eu/domibus/domibus-plugin-api/4.0.2/domibus-plugin-api-4.0.2-javadoc.jar.

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

6. List of Figures

Figure 1 - Message Submission from the backend 6
Figure 2 - Message reception by the backend and delivery to the plugin (PUSH/PULL mode) 7
Figure 3 – Submission validation class diagram 11

7. List of Tables

No table of figures entries found.

8. Contact Information

CEF Support Team

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

Support Service: 8am to 6pm (Normal EC working Days)