EUROPEAN COMMISSION

DIGIT
Connecting Europe Facility

Access Point

Domibus 4.2.3

Plugin Cookbook

Version [4.4]

Status [Final]

Date: 04/08/2021

Document Approver(s):

Approver Name	Role
Bogdan DUMITRIU	Project Manager

Document Reviewers:

Reviewer Name	Role
Caroline AEBY and Chaouki BERRAH	Technical Writers
Cosmin BACIU	Domibus Technical Leader
Ioana DRAGUSANU	CEF Technical Officer

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
2.08	07/09/2018	Cosmin BACIU	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
2.4	16/04/2019	Caroline AEBY	Domibus 4.1 updates
2.5	17/04/2019	Cosmin BACIU	Domibus 4.1 updates: added payloadSubmittedEvent & payloadProcessedEvent
2.6	07/05/2019	Cosmin BACIU	Domibus 4.1-RC1 updates
2.7	15/07/2019	Caroline AEBY	4.1-RC1 => 4.1
2.8	09/09/2019	Caroline AEBY	4.1.1 + reference to the SAD for Multitenancy
2.9	11/10/2019	Ioana Dragusanu	Domain handling in Multitenancy environment.
3.0	05/11/2019	Cosmin Baciu	Password encryption documented
3.1	06/11/2019	Cosmin Baciu	Notification listener information added
3.2	04/02/2020	Caroline AEBY	Domibus 4.1.3
3.3	14/04/2020	Caroline AEBY	Remark on common classloader and spring context between standard and custom plugins
3.4	27/05/2020	Ion PERPEGEL Caroline AEBY	Domibus 4.2. Section §3.10 added for Domibus 4.2
3.5	24/06/2020	Caroline AEBY Ioana Dragusanu	Domibus 4.2 Update plugin API deprecated methods
3.6	02/07/2020	Catalin Enache	Domibus 4.2 Added Section 3.8 - Ehcache documentation
3.7	28/07/2020	Cosmin BACIU	Domibus 4.2 Documented Command Service
3.8	20/08/2020	Arun Raj Venugopal	Added section 6.1.6 setting the domain MDC in logs
3.9	21/09/2020	Caroline AEBY	Domibus 4.2 RC release
4.0	18/11/2020	Caroline AEBY Cosmin BACIU	Domibus 4.2 FR - new API for plugin implementation, updated message submission and reception diagrams, deprecated API and how to migrate to the new API, plugin properties
4.1	24/02/2021	Caroline AEBY	Domibus 4.2.1 version
4.2	18/05/2021	Caroline AEBY	Domibus 4.2.2 version
4.3	25/06/2021	Caroline AEBY Cosmin BACIU	Receive notifications
4.3	04/08/2021	Caroline AEBY	Domibus 4.2.3 version

Table of Contents

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 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 .

Important remark: Developers of custom plugins should ensure that their plugins can be deployed alongside the standard Domibus Default Plugins, given that they share a common classloader and Spring context.

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

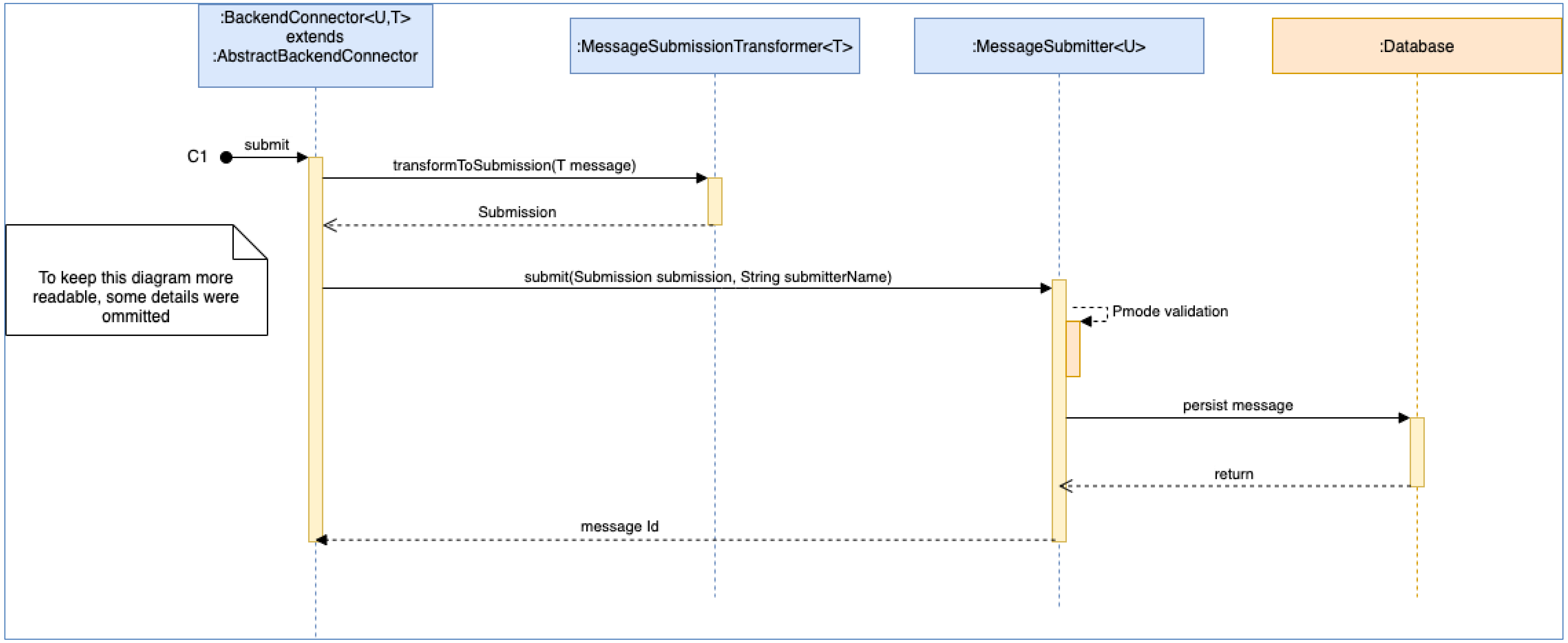

:BackendConnector<U,T> extends :AbstractBackendConnector
:MessageSubmissionTransformer<T>
:MessageSubmitter<U>
:Database
C1
submit
transformToSubmission(T message)
Submission
submit(Submission submission, String submitterName)
Pmode validation
persist message
return
message Id
To keep this diagram more readable, some details were ommitted

Figure 1 - Message Submission from the backend

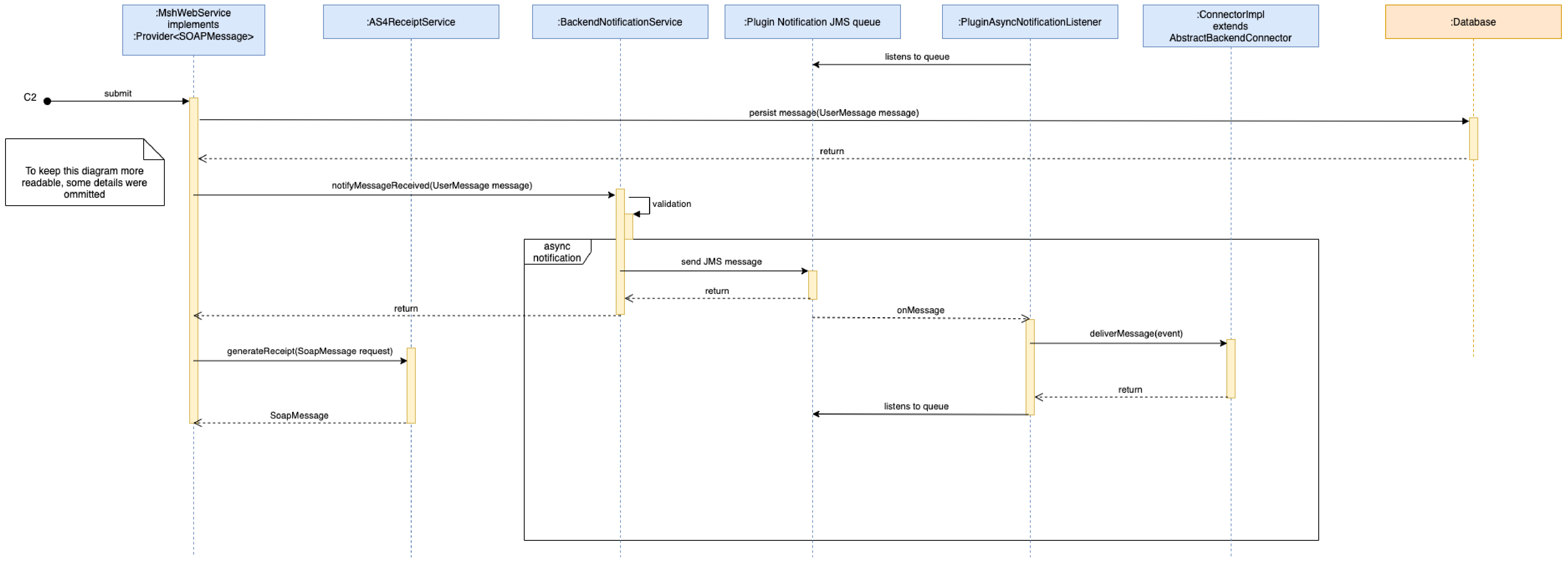

:MshWebService implements :Provider<SOAPMessage>
:AS4ReceiptService
:BackendNotificationService
:Plugin Notification JMS queue
:PluginAsyncNotificationListener
:ConnectorImpl extends AbstractBackendConnector
:Database
listens to queue
C2
submit
persist message(UserMessage message)
return
To keep this diagram more readable, some details were ommitted
notifyMessageReceived(UserMessage message)
validation
async notification
send JMS message
return
return
onMessage
deliverMessage(event)
generateReceipt(SoapMessage request)
return
listens to queue
SoapMessage

Figure 2 - Message reception by C3 from C2

3. Implementing a Plugin

3.1. Pull and Push plugins

There are two different ways of communicating with the backend.

- *eu.domibus.plugin.BackendConnector.Mode.PUSH*
- *eu.domibus.plugin.BackendConnector.Mode.PULL*

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, and the plugin is configured to receive notifications asynchronously, the notification will be retried according to the backend queue definition as described in Section B.3, "Plugin configuration and deployment". All three default Domibus plugins are configured in PUSH mode.

Second option 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).

A PULL plugin can get the status of a message by calling the *eu.domibus.common.MessageStatus getStatus(java.lang.String messageId)* method. In order to get more information in case of errors, the plugin can use a dedicated method *java.util.List< eu.domibus.common.ErrorResult> getErrorsForMessage(java.lang.String messsageId).*

3.2. Extending 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. *Mode.PUSH*

• getMessageSubmissionTransformer()
• getMessageRetrievalTransformer()
• deliverMessage(eu.domibus.common.DeliverMessageEvent)
• messageReceiveFailed(eu.domibus.common.MessageReceiveFailureEvent)
• messageSendFailed(eu.domibus.common.MessageSendFailedEvent)
• messageSendSuccess(eu.domibus.common.MessageSendSuccessEvent)
•messageStatusChanged(eu.domibus.common.MessageStatusChangeEvent event)
• payloadSubmittedEvent (eu.domibus.common.PayloadSubmittedEvent): Notifies the plugins for every payload that has been submitted to C2 but not yet saved

• payloadProcessedEvent (eu.domibus.common.PayloadProcessedEvent): Notifies the plugins for every payload that has been saved by C2

3.2.2. Mode.PULL

- getMessageSubmissionTransformer()
- getMessageRetrievalTransformer()
- payloadSubmittedEvent (eu.domibus.common.PayloadSubmittedEvent): Notifies the plugins for every payload that has been submitted to C2 but not yet saved
- payloadProcessedEvent (eu.domibus.common.PayloadProcessedEvent): Notifies the plugins for every payload that has been saved by C2

PULL plugins must configure an instance of eu.domibus.plugin.MessageLister. The instantiation of the MessageLister can be done in a Spring configuration class. First, we inject the ObjectProvider that will help to create the MessageLister instance:

```
@Autowired
protected ObjectProvider<QueueMessageLister> queueMessageListerObjectProvider;
```

and then we create the MessageLister instance passing as arguments the plugin JMS notification queue and the plugin name:

```
queueMessageListerObjectProvider.getObject(backendNotificationQueue,
getBackendName())
```

listPendingMessages() is only callable from *Mode.PULL* plugins. Additional details on these methods can be found in the Javadoc (see chapter 5). To download a message a combination of *listPendingMessages()* and *downloadMessage(java.lang.String, T)* should be used.

For both PUSH and PULL plugins, the *submit(U)* implementation of *eu.domibus.plugin.AbstractBackendConnector* should be used to submit a message to Domibus.

3.3. Implementing Message transformers

The implementations of the transformer classes are responsible for transformation between the native backend formats and *eu.domibus.plugin.Submission*:

eu.domibus.plugin.transformer.MessageSubmissionTransformer and
eu.domibus.plugin.transformer.MessageRetrievalTransformer

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. Receiving notifications

Domibus sends notifications to the plugins when different message events occur. Domibus notifies the plugins either synchronously, via callbacks, or asynchronously, via a JMS queue. More information about which callbacks are available can be found in *eu.domibus.plugin.BackendConnector* interface.

For example, the WS Plugin receives notifications on the *jms/domibus.notification.webservice* queue.

The notification types available are *eu.domibus.common.NotificationType*:

```
public enum NotificationType {
   MESSAGE_RECEIVED, MESSAGE_FRAGMENT_RECEIVED, MESSAGE_SEND_FAILURE,
MESSAGE_FRAGMENT_SEND_FAILURE, MESSAGE_RECEIVED_FAILURE,
MESSAGE_FRAGMENT_RECEIVED_FAILURE, MESSAGE_SEND_SUCCESS,
MESSAGE_FRAGMENT_SEND_SUCCESS, MESSAGE_STATUS_CHANGE,
MESSAGE_FRAGMENT_STATUS_CHANGE;
}
```

MESSAGE_RECEIVED – C3 receives a UserMessage from C2

MESSAGE_FRAGMENT_RECEIVED – C3 receives a UserMessage fragment from C2

MESSAGE_SEND_FAILURE - C2 fails to send a UserMessage to C3

MESSAGE_FRAGMENT_SEND_FAILURE - C2 fails to send a UserMessage fragment to C3

MESSAGE_RECEIVED_FAILURE – C3 fails to receive a UserMessage

MESSAGE_FRAGMENT_RECEIVED_FAILURE - C3 fails to receive a UserMessage

MESSAGE_SEND_SUCCESS – C2 sends successfully a UserMessage to C3

MESSAGE_FRAGMENT_SEND_SUCCESS - C2 sends successfully a UserMessage fragment to C3

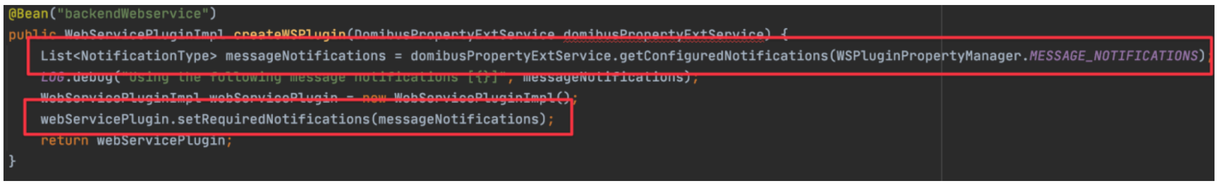
MESSAGE_STATUS_CHANGE – UserMessage status changes

MESSAGE_FRAGMENT_STATUS_CHANGE – UserMessage fragment status changes

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

A plugin can override the default list of notifications for which it wants to be notified. This can be done by setting the list of notifications via the method *eu.domibus.plugin.AbstractBackendConnector#setRequiredNotifications.*

For example, the WS Plugin configure the list of notifications in the plugin property file and overrides its list of notifications:



```
@Bean("backendWebservice")
public WebServicePluginImpl createWSPlugin(DomibusPropertyExtService domibusPropertyExtService) {
    List<NotificationType> messageNotifications = domibusPropertyExtService.getConfiguredNotifications(WSPluginPropertyManager.MESSAGE_NOTIFICATIONS);
    LOG.debug("Using the following message notifications [{}]", messageNotifications);
    WebServicePluginImpl webServicePlugin = new WebServicePluginImpl();
    webServicePlugin.setRequiredNotifications(messageNotifications);
    return webServicePlugin;
}
```

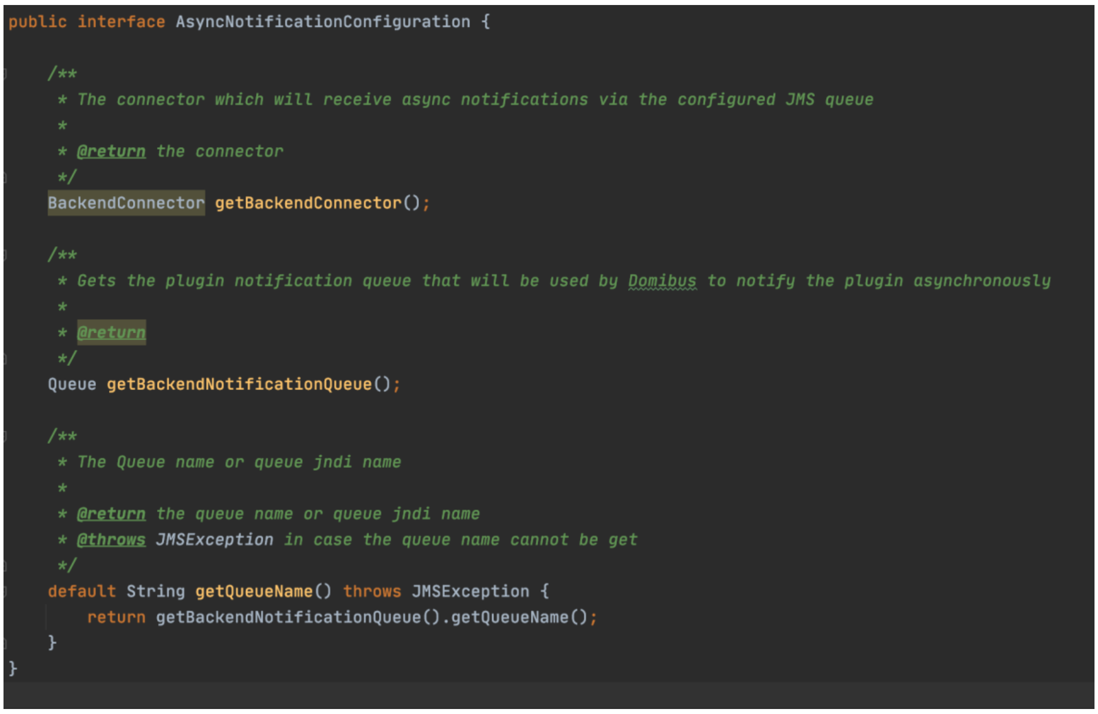
3.4.1. Synchronous notifications

A plugin receives notifications synchronously if no *eu.domibus.plugin.notification.PluginAsyncNotificationConfiguration* is configured for the plugin. This means that Domibus calls in the same thread the plugin notification method, for instance when a message is sent successfully it will call *eu.domibus.plugin.BackendConnector#messageSendSuccess(eu.domibus.common.MessageSendSuccessEvent).*

3.4.2. Asynchronous notifications

A plugin can be configured to receive notifications asynchronously from Domibus. In this mode, Domibus sends a JMS message to the plugin notification queue. The plugin JMS listener consumes JMS notification messages from the notification queue and then it will notify the plugin.

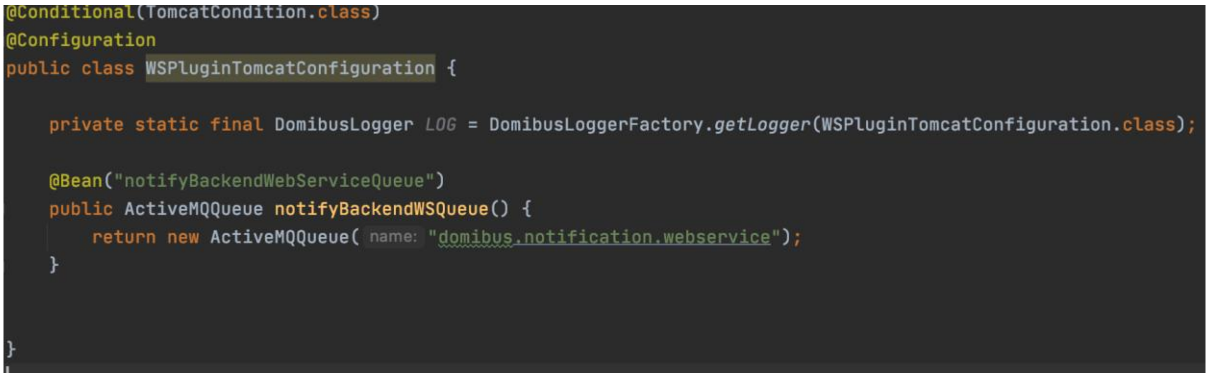
In order to receive asynchronous notifications, a plugin must create a *eu.domibus.plugin.notification.AsyncNotificationConfiguration* Spring bean. The class *eu.domibus.plugin.notification.PluginAsyncNotificationConfiguration* implements the *eu.domibus.plugin.notification.AsyncNotificationConfiguration* interface and it can be used to create the async configuration.



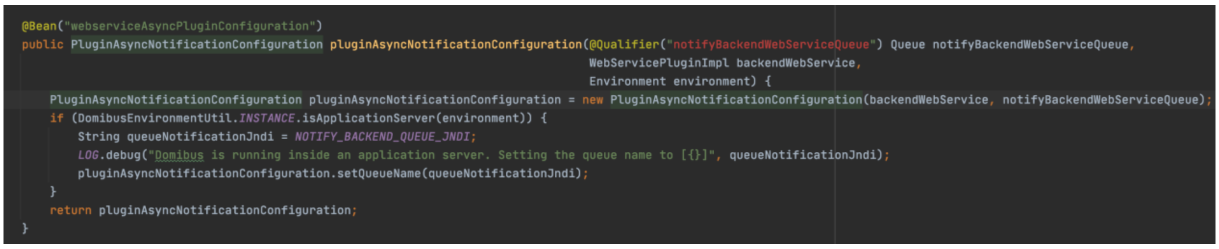
```
public interface AsyncNotificationConfiguration {

    /**
     * The connector which will receive async notifications via the configured JMS queue
     *
     * @return the connector
     */
    BackendConnector getBackendConnector();

    /**
     * Gets the plugin notification queue that will be used by Domibus to notify the plugin asynchronously
     *
     * @return
     */
    Queue getBackendNotificationQueue();

    /**
     * The Queue name or queue jndi name
     *
     * @return the queue name or queue jndi name
     * @throws JMSException in case the queue name cannot be get
     */
    default String getQueueName() throws JMSException {
        return getBackendNotificationQueue().getQueueName();
    }
}
```

The *AsyncNotificationConfiguration* must be configured with the plugin *BackendConnector* instance, the plugin JMS notification queue where Domibus will send JMS messages to notify the plugin and the JMS queue name.

For instance, please find below how the WS Plugin configures the *AsyncNotificationConfiguration* Spring bean instance for Tomcat using Java configuration:



```
@Conditional(TomcatCondition.class)
@Configuration
public class WSPluginTomcatConfiguration {

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

    @Bean("notifyBackendWebServiceQueue")
    public ActiveMQQueue notifyBackendWSQueue() {
        return new ActiveMQQueue( name: "domibus.notification.webservice");
    }


}
```



```
@Bean("webserviceAsyncPluginConfiguration")
public PluginAsyncNotificationConfiguration pluginAsyncNotificationConfiguration(@Qualifier("notifyBackendWebServiceQueue") Queue notifyBackendWebServiceQueue,
                                                                                 WebServicePluginImpl backendWebService,
                                                                                 Environment environment) {
    PluginAsyncNotificationConfiguration pluginAsyncNotificationConfiguration = new PluginAsyncNotificationConfiguration(backendWebService, notifyBackendWebServiceQueue);
    if (DomibusEnvironmentUtil.INSTANCE.isApplicationServer(environment)) {
        String queueNotificationJndi = NOTIFY_BACKEND_QUEUE_JNDI;
        LOG.debug("Domibus is running inside an application server. Setting the queue name to [{}]", queueNotificationJndi);
        pluginAsyncNotificationConfiguration.setQueueName(queueNotificationJndi);
    }
    return pluginAsyncNotificationConfiguration;
}
```

3.5. Conditional Spring bean creation

Some Spring beans must be created conditionally depending on the server on which Domibus is deployed. When using Java configuration, a plugin can take advantage of the server conditions that are already existing in the Plugin API:

- ApplicationServerCondition: true when Domibus is deployed on WebLogic or WildFly
- TomcatCondition: true when Domibus is deployed on Tomcat
- WebLogicCondition: true when Domibus is deployed on WebLogic
- WildFlyCondition: true when Domibus is deployed on WildFly

For instance, the following Spring bean will be created when Domibus is deployed on Tomcat:

```
@Conditional(TomcatCondition.class)
@Configuration
public class FSPluginTomcatConfiguration
```

3.6. 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:

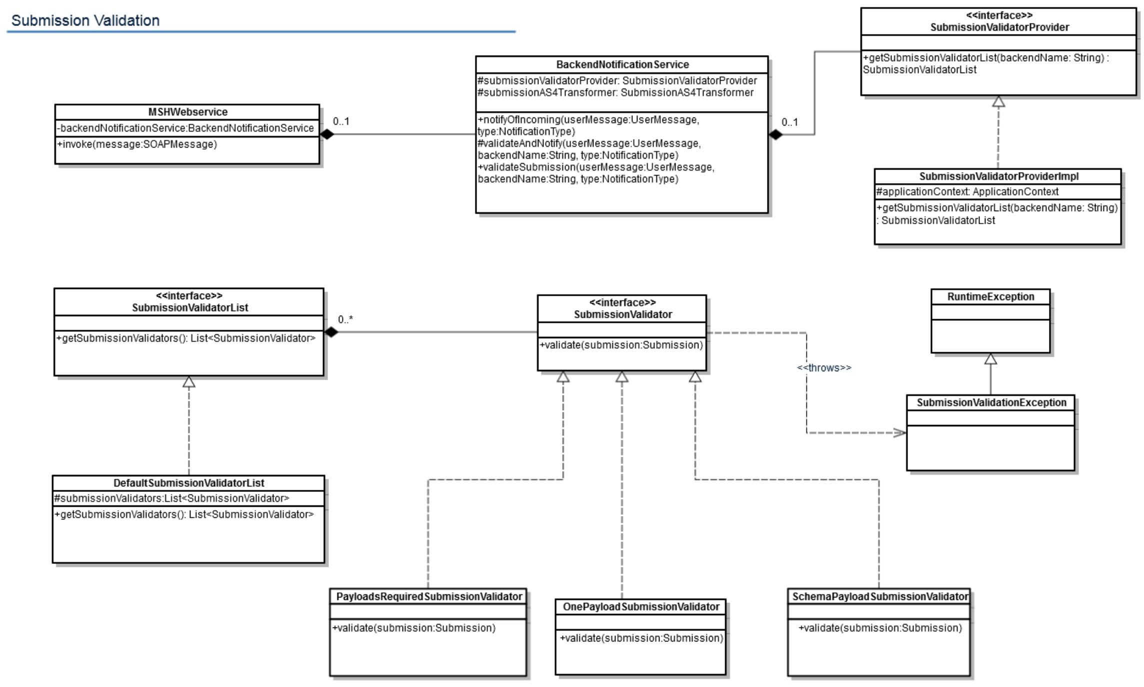

Submission Validation
MSHWebservice
-backendNotificationService:BackendNotificationService
+invoke(message:SOAPMessage)
0..1
BackendNotificationService
#submissionValidatorProvider: SubmissionValidatorProvider
#submissionAS4Transformer: SubmissionAS4Transformer
+notifyOfIncoming(userMessage:UserMessage, type:NotificationType)
#validateAndNotify(userMessage:UserMessage, backendName:String, type:NotificationType)
+validateSubmission(userMessage:UserMessage, backendName:String, type:NotificationType)
0..1
<<interface>> SubmissionValidatorProvider
+getSubmissionValidatorList(backendName: String) : SubmissionValidatorList
SubmissionValidatorProviderImpl
#applicationContext: ApplicationContext
+getSubmissionValidatorList(backendName: String) : SubmissionValidatorList
<<interface>> SubmissionValidatorList
+getSubmissionValidators(): List<SubmissionValidator>
0..*
<<interface>> SubmissionValidator
+validate(submission:Submission)
<<throws>>
RuntimeException
SubmissionValidationException
DefaultSubmissionValidatorList
#submissionValidators:List<SubmissionValidator>
+getSubmissionValidators(): List<SubmissionValidator>
PayloadsRequiredSubmissionValidator
+validate(submission:Submission)
OnePayloadSubmissionValidator
+validate(submission:Submission)
SchemaPayloadSubmissionValidator
+validate(submission:Submission)

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>
<!-- 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>
```

3.7. 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.

3.7.1. 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 three 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 **3.7.2 Authorization.**

3.7.2. 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**.**

3.8. 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 (SAD)** (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);

3.9. Ehcache

Domibus uses EhCache implementation for caching using @Cacheable annotations or programmatically.

At plugin level it is possible to add two configuration files for Ehcache in order to define theri own cache names. However, please note that the caches will be merged to Domibus Ehcache manager.

3.9.1. Default Ehcache file (classpath file)

- The expected format is *-plugin-default-ehcache.xml e.g ws-plugin-default-ehcache.xml
- The expected location (in a Java project) is: /src/main/resources/config/ehcache/*-plugin-default-ehcache.xml which will go to .jar file of the plugin into e.g. /config/ehcache/ws-plugin-default-ehcache.xml location
- The cache names defined in this file should be prefixed with the plugin name in order to avoid collision with Domibus cache names. Otherwise, Domibus will thrown an exception and will not deploy:

```
<ehcache>
  <cache name="wsplugin.policyCache"
      maxBytesLocalHeap="5m"
      timeToLiveSeconds="3600"
      overflowToDisk="false">
  </cache>
</ehcache>
```

3.9.2. External file

- The expected format is *-plugin-ehcache.xml e.g ws-plugin-ehcache.xml
- The expected location is: ${domibus.config.location}/plugins/config
- The cache names defined here could override the one defined in the classpath file (see above), they must not override/collide with Domibus cache names

```
<ehcache>
  <cache name="wsplugin.policyCache"
```

```
        maxBytesLocalHeap="5m"
        timeToLiveSeconds="360"
        overflowToDisk="false">
    </cache>
    <cache name="wsplugin.crlByCert"
        maxBytesLocalHeap="5m"
        timeToLiveSeconds="3600"
        overflowToDisk="false">
    </cache>
</ehcache>
```

In the above file, wsplugin.policycache overrides the one defined in the classpath file.

The following algorithms will be used:

A. Domibus parses default-ehcache.xml (classpath) and config/internal/ehcache.xml (which may override the caches declared in the first file).

B. Domibus parses all plugin default files from classpath (.jar) - -default-ehcache.xml and all caches to a plugin cache manager (in memory).

C. Domibus parses all plugin non default files - from /plugins/config folders - the caches are added to the same plugins manager and they may override the caches defined at step B.

D. Domibus merges the plugins cache manager (step B+C) to the Domibus cache manager (step A). If there is a plugin cache which is overriding a Domibus cache -> a DomibusCoreException is thrown and the deployment is stopped.

3.10. 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.

3.10.1. *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()))
```

3.10.2. 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

3.10.3. Command service

This service is used to execute commands in a cluster.

A plugin can execute a cluster command using the following steps:

1. Implement the command logic in a Spring service bean which implements the interface eu.domibus.ext.services.CommandExtTask. The command task must use a unique command name and must indicate that it can handle the command in the **canHandle** method. Eg:

```
@Service
public class CustomCommandTask implements CommandExtTask {

  public static final String COMMAND_NAME = "mycommand";

  @Override
  public boolean canHandle(String command) {
    return COMMAND_NAME.equals(command);
  }

  @Override
  public void execute(Map<String, String> properties) {
    //custom logic
  }
}
```

2. Trigger the execution of the command using the method eu.domibus.ext.services.CommandExtService#executeCommand(String commandName, Map<String, Object> properties). The command name is the name of the command defined at the above point 1. Custom parameters can be passed to the command using the properties parameter.

3.11. Password encryption

Passwords configured in the plugin configuration files are stored by default in clear text.

Domibus gives the possibility to the plugins to encrypt passwords configured in the plugin configuration file using symmetric encryption with *AES/GCM/NoPadding* algorithm.

The *eu.domibus.plugin.encryption.PluginPropertyEncryptionListenerinterface* interface should be implemented by the custom who need to encrypt their own passwords. The plugin passwords can be configured in property files or in other sources, like a database.

If password encryption is active (*domibus.password.encryption.active=true*), Domibus will generate the secret keys that are used to encrypt the passwords. Domibus will notify afterwards via the *PluginPropertyEncryptionListenerinterface* interface the subscribed plugin listeners to encrypt their own passwords. Custom plugins can use the password encryption services, *eu.domibus.ext.services.PasswordEncryptionExtService*, to handle the password encryption.

The password encryption has already been implemented in the Default File System Plugin, which can serve as an example on how to implement password encryption in your own custom plugin.

For instance, the property *fsplugin.authentication.password=test123* will be encrypted to *fsplugin.authentication.password=ENC(4DTXnc9zUuYqB0P/q7RtRHpG9VJLs3E=).*

At the API level, the password properties are decrypted automatically at runtime by the eu.domibus.ext.services.DomibusPropertyExtService service when they are retrieved, so there are no additional actions that must be performed by a plugin.

4. Plugin properties

A plugin will need at some point to define its own set of properties, apart from the Domibus properties. Typically the plugin properties are configured in a property file, for instance ws-plugin.properties.

Domibus gives the possibility to the plugins to use integrated property management. The plugin must first define its properties into the plugin property file and then it must create properties metadata, property metadata manager and property manager to manage how and where these properties are retrieved and set.

4.1. Property files

A plugin can configure properties in two locations: in the classpath or on the file system.

Domibus introspects the classpath for files under **config** location which end in **-plugin-default.properties.** When using Maven, the plugin can configure the default property file under src/main/resources/config. Once a default property file is found, Domibus adds all the properties in the central org.springframework.context.support.PropertySourcesPlaceholderConfigurer. A plugin **must not** configure another PropertySourcesPlaceholderConfigurer, neither via Java configuration nor XML configuration(context:property-placeholder).

A plugin can configure also an external property file located under ${domibus.config.location}/plugins/config. The properties from the external property file override the default properties defined in the classpath.

4.2. Property metadata

In order to integrate into the property management, one needs to create a property manager. This can be achieved by creating a class that implements the *DomibusPropertyManagerExt* interface or by extending the abstract class *DomibusPropertyExtServiceDelegateAbstract.* If the properties are intended to be managed using its own property bag then one should implement directly the interface. If the properties are to be stored in the Domibus property bag then one should extend the abstract class.

In case of extending the abstract class (like default Jms plugin does), one should implement only the

```
public abstract Map<String, DomibusPropertyMetadataDTO> getKnownProperties();
```

method, and provide the map of properties metadata.

`DomibusPropertyMetadataDTO` class is used to store the attributes of a property.

The most important attributes are:

- *name*- the name of the property, the same as the one stored in the file with the value.
- *type*- NUMERIC/BOOLEAN/CONCURRENCY/EMAIL/CRON/STRING- used for validation when changing them at runtime.

- *writable*- specifies if the property can be changed at runtime.
- *usage*- can be GLOBAL , DOMAIN or SUPER or a combination of them, as described in SAD document (Domibus Software Architecture (SAD) document can be downloaded at https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus in the documentation section).
- *withFallback* – falls back to the global value for the domain property.
- *storedGlobally* – must be false if the plugin manages its value or true if domibus property provider does it.

In case the plugin manages the properties internally, one must implement all the methods of the *DomibusPropertyManagerExt* interface. Besides providing the map of property metadata as described above, one should also implement the get and set methods to store and retrieve the property value from the bag. In this case, the *storedGlobally* attribute must be set to false.

If a property is marked as writable, it must be possible to change it at runtime and it will appear in the admin console properties page (and users will be able to change it), so the set methods need to be implemented accordingly. If some custom code needs to be executed while changing it, one should create a class that implements the *PluginPropertyChangeListener* interface, and the method *public boolean handlesProperty(String propertyName)* must return true if the *propertyName* parameter is equal to the property name for which the custom code must be executed.

If a property has domain usage, the property manager needs to manage a different value for each domain. In this case, the methods

String getKnownPropertyValue(String propertyName);

void setKnownPropertyValue(String propertyName, String propertyValue);

need to get and set the value for the current domain of the Domibus.

5. 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

6. API Documentation

Standard Javadoc documentation for the API can be downloaded on the https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus.

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

7. MULTITENANCY

To understand how Multitenancy works in Domibus, please refer to Domibus Software Architecture Document (SAD), chapter “Multitenancy” and especially in the section “Plugins”.

After reading the SAD, the following aspects should be clear in a multitenant environment:

- Authentication is always required to submit a message (C1 to C2),
- Every plugin user is associated to one and only one domain,
- Every domain has its own schema in the database,
- A general schema exists to match users to domains,
- Default plugins already work in multitenant environment; they can be used as examples.

Any custom plugin should work in a Domibus multitenant environment.

Domibus Plugin API offers the possibility to handle the domains and access the database schema specific to each domain. Examples of domain handling using the Domibus plugin API are presented below.

7.1.1. Check if Domibus runs in Multitenancy mode

```
@Autowired
protected DomibusConfigurationExtService domibusConfigurationExtService;
boolean isMultiTenantAware = domibusConfigurationExtService.isMultiTenantAware();
```

7.1.2. Get the current domain

```
@Autowired
private DomainContextExtService domainContextExtService;
DomainDTO domainDTO = domainContextExtService.getCurrentDomainSafely();
```

7.1.3. Set the current domain for an asynchronous execution

When the plugin uses asynchronous execution, the domain code is required to be configured manually.

Let’s take the scenario where a message is put in a JMS queue and a worker picks it up and sends it.

Along with the message, the domain code must be added in the JMS message. The consumer should first get the domain based on the domain code, set it as current domain, and afterwards execute the job.

Finally, the plugin is responsible to clear the current domain.

Get the domain:

```
@Autowired
protected DomainExtService domainExtService;

final DomainDTO = domainExtService.getDomain(domainCode)
```

Set current domain, execute job and clear domain:

```
@Autowired
protected DomainContextExtService domainContextExtService;

try {
   domainContextExtService.setCurrentDomain(currentDomain);

   // … executeJob

} finally {
   domainContextExtService.clearCurrentDomain();
}
```

7.1.4. Access the database

Every domain has its own schema in the database. In case the plugin needs a new table to persist plugin specific data, it is recommended to keep the same segregation as the Domibus core and add a new table in the database schema of each domain.

The recommended naming convention is **<plugin_name>_TB_<name>** for tables storing plugin data and **<plugin_name>_QRTZ_<name>** for quartz tables.

See section §7.1.5 to see how to create a quartz job.

Below example shows how to save message info in a new table:

a) Create the tables in the database (to be performed by the database admin)

   Table: WS_PLUGIN_TB_MESSAGE

   - ID_PK - primary key (auto increment)
   - DESCRIPTION

b) Create the entity class:

```

import javax.persistence.*;

@Entity
```

```
@Table(name = "WS_PLUGIN_TB_MESSAGE")
public class MessageInfoEntity {

    @Id
    @Column(name = "ID_PK")
    private long entityId;

    @Column(name = "DESCRIPTION")
    private String description;

    /**
     * @return the primary key of the entity
     */
    public long getEntityId() {
        return this.entityId;
    }

    public void setEntityId(long entityId) {
        this.entityId = entityId;
    }

    public MessageInfoEntity(final String description){
        this.description = description;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

c) Create a basic DAO implementation providing the standard CRUD operations. The class reuses the Domibus EntityManager.

```

public abstract class BasicDao<T> {

    protected final Class<T> typeOfT;

    @PersistenceContext(unitName = "domibusJTA")
    protected EntityManager em;

    /**
     * @param typeOfT The entity class this DAO provides access to
     */
    public BasicDao(final Class<T> typeOfT) {
        this.typeOfT = typeOfT;
    }
    public <T> T findById(Class<T> typeOfT, String id) {
        return em.find(typeOfT, id);
    }

    @Transactional(propagation = Propagation.REQUIRED)
    public void create(final T entity) {
        em.persist(entity);
    }

    @Transactional(propagation = Propagation.MANDATORY)
    public void delete(final T entity) {
        em.remove(em.merge(entity));
    }

    public T read(final long id) {
        return em.find(this.typeOfT, id);
    }

    @Transactional(propagation = Propagation.MANDATORY)
    public void updateAll(final Collection<T> update) {
        for (final T t : update) {
            this.update(t);
        }
    }

    @Transactional(propagation = Propagation.MANDATORY)
    public void deleteAll(final Collection<T> delete) {
        for (final T t : delete) {
            this.delete(t);
        }
    }

    @Transactional(propagation = Propagation.MANDATORY)
    public void update(final T entity) {
        em.merge(entity);
    }

    public void flush() {
        em.flush();
    }
}
```

Note that the EntityManager is already configured in Domibus core to be multitenant aware and persist the data in the right table, of the current domain.

d) Create the MessageInfoDao class, that extends the BasicDao:

```
@Repository
public class MessageInfoDao extends BasicDao<MessageInfoEntity> {

   public MessageInfoDao() {
      super(MessageInfoEntity.class);
   }
}
```

e) Save the message info:

```
@Autowired
MessageInfoDao messageInfoDao;

String description = “New message messageId = " + messageInfo.getMessageId();
messageInfoDao.create(new MessageInfoEntity(description));
```

7.1.5. Create a quartz job

To create a new Quartz job that executes plugin specific tasks, it is recommended to extend the abstract class DomibusQuartzJobExtBean and override the method executeJob() with the specific actions. By doing this, the domain configuration is handled internally by Domibus.

```
@DisallowConcurrentExecution // Only one Worker runs at any time on the same node
public class MyPluginWorker extends DomibusQuartzJobExtBean {

   @Override
   protected void executeJob(JobExecutionContext context, DomainDTO domain) {
      //do plugin specific tasks
   }
}
```

Create the quartz job:

```
<bean id="myPluginWorkerJob"
     class="org.springframework.scheduling.quartz.JobDetailFactoryBean">
   <property name="jobClass" value="eu.domibus.plugin.worker. MyPluginWorker "/>
   <property name="durability" value="true"/>
</bean>
```

Create the Quartz job trigger:

```
<bean id="MyPluginWorkerTrigger"
    class="org.springframework.scheduling.quartz.CronTriggerFactoryBean"
scope="prototype">
  <property name="jobDetail" ref=" myPluginWorkerJob "/>
  <property name="cronExpression" value=" 0 0/1 * * * ?"/>
  <property name="startDelay" value="20000"/>
</bean>
```

The default FileSystem plugin provides examples of how to create new Quartz jobs to purge sent, received and failed files that were archived.

7.1.6. Setting the domain MDC in logs

Refer section **Logging** in the Domibus Administration guide for details on configuring logging in Domibus (the document can be downloaded at https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus in the documentation section).

In multi-tenant scenarios, the domain name is logged in an MDC (Mapped Diagnostic Context) field represented in the logback configuration by **[%X{d_domain}]**. DomibusLogger class should be used for this purpose. Within the plugins, domain MDC can be set in the logs using the following code:

```
import eu.domibus.logging.DomibusLogger;
import eu.domibus.logging.DomibusLoggerFactory;
import static eu.domibus.logging.DomibusLogger.MDC_DOMAIN;
…
private static final DomibusLogger LOG = DomibusLoggerFactory.getLogger(….class);
…
LOG.putMDC(MDC_DOMAIN, domain.getCode());
```

8. Deprecated API and Migrating to the new API

Starting with Domibus 4.2 some classes and methods from the Plugin API have been deprecated. The deprecated API is still supported in Domibus 4.2 but it will be removed in Domibus 5.0.

In this chapter we will give more details about the deprecated Plugin API and how to migrate to the new API.

The following classes and methods have been deprecated:

- eu.domibus.plugin.NotificationListener interface

The *NotificationListener* interface was used by a plugin to receive async notifications. It has been replaced by *eu.domibus.plugin.notification.AsyncNotificationConfiguration* which is documented in the **3.4.2 Asynchronous notifications** section.

Please find below all methods of the *NotificationListener* interface and the new methods that are replacing the deprecated methods:

Deprecated method	New method
String getBackendName()	eu.domibus.plugin.BackendConnectorBackendConnector#getName()
BackendConnector.Mode getMode()	eu.domibus.plugin.BackendConnector#getMode()
List<NotificationType> getRequiredNotificationTypeList()	BackendConnector#getRequiredNotifications()
void deleteMessageCallback(String messageId)	BackendConnector#messageDeletedEvent(MessageDeletedEvent)
void notify(final String messageId, final NotificationType notificationType, final Map<String, Object> properties)	Domibus notifies the connector using the lifecycle methods eg messageSendSuccess, messageSendFailed, etc defined in eu.domibus.plugin.BackendConnector

- eu.domibus.plugin.NotificationListenerService class has been replaced by eu.domibus.plugin.notification.PluginAsyncNotificationConfiguration

The class NotificationListenerService is implementing eu.domibus.plugin.NotificationListener and eu.domibus.plugin.MessageLister. We have explained above how to migrate eu.domibus.plugin.NotificationListener.

The interface eu.domibus.plugin.MessageLister is used to access messages waiting to be pulled by plugins using Mode.PULL, as explained in Error! Reference source not found. Error! Reference source no t found.. Each PULL plugin must define its own instance of MessageLister that will replace the PULL specific implementation from NotificationListenerService.

- eu.domibus.submission.WeblogicNotificationListenerService class; use PluginAsyncNotificationConfiguration and set the JNDI name as the queue name

- the methods from eu.domibus.plugin.BackendConnector

    - deliverMessage(java.lang.String) replaced by deliverMessage(eu.domibus.common.DeliverMessageEvent)

    - messageSendSuccess(java.lang.String) replaced by messageSendSuccess(eu.domibus.common.MessageSendSuccessEvent)

    - messageSendFailed(java.lang.String) replaced by messageSendFailed(eu.domibus.common.MessageSendFailedEvent)

- the methods from eu.domibus.ext.services.DomibusPropertyManagerExt class:

    - getKnownPropertyValue(java.lang.String, java.lang.String) replaced by getKnownPropertyValue(java.lang.String)

    - setKnownPropertyValue(java.lang.String, java.lang.String, java.lang.String) replaced by setKnownPropertyValue(java.lang.String, java.lang.String)

- the method from eu.domibus.ext.services.PModeExtService

    - updatePModeFile(byte[], java.lang.String) was replaced by updatePModeFile(org.springframework.web.multipart.MultipartFile, java.lang.String)

Next we will give an example with a deprecated plugin configuration and we will describe how to migrate to the new plugin configuration. We will use the WS Plugin in this example.

Old configuration:

```
<amq:queue id="notifyBackendWebServiceQueue"
               physicalName="domibus.notification.webservice"/>


    <bean id="backendWebservice"
          class="eu.domibus.plugin.webService.impl.BackendWebServiceImpl">
        <constructor-arg value="backendWebservice"/>
    </bean>

    <bean id="webserviceNotificationListenerService"
          class="eu.domibus.plugin.NotificationListenerService"
          c:queue-ref="notifyBackendWebServiceQueue" c:mode="PULL"
          p:backendConnector-ref="backendWebservice"/>
```

New configuration:

```
    <bean id="backendWebservice"
          class="eu.domibus.plugin.webService.impl.WebServicePluginImpl">
        <constructor-arg value="backendWebservice"/>
    </bean>

    <bean id="webserviceNotificationListenerService"

class="eu.domibus.plugin.notification.PluginAsyncNotificationConfiguration"
          c:queue-ref="notifyBackendWebServiceQueue"
          c:backendConnector-ref="backendWebservice"/>
```

9. List of Figures

10. Contact Information

CEF Support Team

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

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