



OSGiTM Alliance

Messaging

Draft

29 Pages

*Text in Red is here to help you. Delete it when you have followed the instructions.
The <RFC Title> can be set from the File>Properties:User Defined menu. To update it onscreen, press F9. To update all of the fields in the document Select All (CTRL-A), then hit F9. Set the release level by selecting one from: Draft, Final Draft, Release. The date is set automatically when the document is saved.*

Abstract

Asynchronous communication is an important factor in today's business applications. Especially in the IoT domain but also for distributed infrastructures the communication over publish/subscribe protocols are common mechanisms. Whereas the existing OSGi Event Admin specification already describes an asynchronous event model within an OSGi framework, this RFP addresses the interaction of an OSGi environment with third-party communication protocols using a common interface.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the "Distribution") in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback ("Feedback") on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

Draft

September 1, 2020

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at <https://github.com/osgi/design>. The public can provide feedback about this document by opening a bug at <https://www.osgi.org/bugzilla/>.

0.4 Table of Contents

0 Document Information.....	2
0.1 License.....	2
0.2 Trademarks.....	3
0.3 Feedback.....	3
0.4 Table of Contents.....	3
0.5 Terminology and Document Conventions.....	4
0.6 Revision History.....	4
 1 Introduction.....	 4
 2 Application Domain.....	 5
 3 Problem Description.....	 5
 4 Requirements.....	 5
 5 Technical Solution.....	 5
 6 Data Transfer Objects.....	 6
 7 Javadoc.....	 6
 8 Considered Alternatives.....	 6

9 Security Considerations.....7**10 Document Support.....7**

10.1 References.....7

10.2 Author's Address.....7

10.3 Acronyms and Abbreviations.....7

10.4 End of Document.....7

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in Fehler: Verweis nicht gefunden.

Source code is shown in this typeface.

0.6 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	26 Nov 2019	Mark Hoffmann, initial content copied from RFP
1.0	<i>January 2020</i>	<i>Mark Hoffmann, some initial API</i>
1.1	<i>April 2020</i>	<i>Mark Hoffmann, Subscriptions, Reply-To, Ack, Builders</i>
1.2	<i>August 2020</i>	<i>Mark Hoffmann, Publisher, Subscription Rework</i>

1 Introduction

In the past there have already been some efforts to bring asynchronous messaging into the OSGi framework. There was the Distributed Eventing RFC-214 and the MQTT Adapter RFC-229. In addition to that there are further available specification like OSGi RSA, Promises and PushStreams, that focus on remote events, asynchronous processing and reactive event handling. Promises and PushStreams are optimal partners to deal with the asynchronous programming model, that comes with the messaging pattern.

Because of the growing popularity of the IoT domain, it is important to enable OSGi to be connected with the common services of the IoT world. This RFP is meant to provide an easy to use solution in OSGi, to connect to

and work with messaging systems. It is not meant to provide access to the full feature set and service guarantees of Enterprise class messaging solution like MQSeries or TIBCO EMS or massively scalable solutions like Kafka. Service guarantees and configuration details will be designed as configuration hints. The implementation will be optional and depend on the binding.

Protocols like AMQP, the Kafka Protocol or JMS are heavily used in back-end infrastructures. This RFP tries to address the use-case for connecting to those protocols with a subset of their functionality. For a seamless use of OSGi as well in IoT infrastructures and cloud-infrastructures, it is important to provide an easy to use and also seamless integration of different communication protocols in OSGi.

With the Event Admin specification, there is already an ease to use approach, for in-framework events. Distributed events often needs additional configuration parameters like quality of service, time-to-live or event strategies that needs to be configured at connection time or set at message publication time. This RFP is seen for standalone use but also as an extension to the Event Admin to provide the possibility for a Remote Event Admin.

2 Application Domain

Messaging is a pattern to reliably transport messages over an inherent unreliable network from a produce to one or more receivers. The process is to move messages from one system to another or many. The messaging system uses channels to do that. Because it is never clear, if the network or the receiver system is available, it is the task of the messaging system to handle that.

There are also different concepts in moving messages:

1. Send and Forget – Guarantees successful send
2. Send and Forward – Guarantees eventual successful receive

The following communication patterns exist:

1. Point-to-point
2. Publish-Subscribe
3. Guaranteed delivery
4. Temporary/transient channel
5. Dead-letter
6. Monitoring, error and administration channels

The use cases above cover Reply-To style communication and handling of common error conditions.

Another important fact is the structure of the messages. They usually consist of a header and body. The body contains the payload that is an array of bytes. The header provides additional properties for the message. There are some common properties that are used for handling Reply-To, sequencing/ordering of messages, time-synchronization, filtering and others. This is important because messaging decouples communication and has different demands regarding assumptions that are made to the process compared to a local call. Thus there is additional semantics for e.g. time-outs, retry-counts, address-spaces.

Messaging in general induces an asynchronous programming model. Therefore Promises and PushStreams are already existing specifications that are an optimal solution for data-handling as well as scaling of actors over more than one thread. These specifications allowing flexible message transformation into internal data formats. Further Promises provide the possibility to realize patterns like message construction or aggregation. [1]

2.1 Terminology + Abbreviations

2.1.1 Message

A message is a data structure that holds the payload and additional meta-information about the content that is submitted of a channel

2.1.2 Channel

Channels are named resources to connect programs and interchange the messages.

2.1.3 Sender / Publisher

A sender writes a message into a channel.

2.1.4 Receiver / Subscriber

One or more receivers read messages from a channel.

2.1.5 Reply-To Messaging

Sending a request and receiving a response are two separate atomic operations. Thus waiting for a response is not a blocking operation in the underlying implementation., A special message information, the correlation identifier, is used to assign a request to a response. Sometimes the reply-to address can be generated from the messaging system and is also submitted as property with the request message.

2.1.6 Message Context

A message context defines the meta-data that are needed to describe the message or/and the way the message has to be handled on sender or/and receiver side.

2.1.7 Message Builder

The Message Builder can create message objects with a corresponding message contexts or alternatively message contexts alone.

2.1.8 Messaging Instance

A messaging instance is bound to a certain implementation and connection settings. It provides subscriptions, publishers and a Message Builder instance.

message builders.

3 Problem Description

The OSGi Alliance already has a successful specification for messaging within an OSGi framework. The EventAdmin specification is well defined and widely used. The same is for the RSA specification that provides a good ground for synchronous calls. Also asynchronous remote services are supported in the RSA.

In the domains of IoT there are standardized protocols to connect remote devices and submit data over a broker based messaging system from remote clients. But also in cloud-based infrastructures, messaging systems are often used for de-coupling of services or functions.

Today, to interact with such systems the implementer has to deal with messaging protocol specifics and operational conditions, that are not covered, by existing specifications. With OSGi Promises and the PushStream specification there are already major parts available to deal with an asynchronous programming model. This is a requirement when using messaging.

The missing piece is a standardized way to send and receive data that supports the messaging patterns. Consuming and producing data using common protocols like AMQP, MQTT or JMS using OSGi services, would integrate an OSGi application into more systems.

Also other specifications could benefit from this specification. It should be possible to layer RSA remote calls over messaging. It should also be possible to provide a remote Event Admin service.

3.1 Intents

Messaging systems vary widely in their capabilities and are configurable with regard to guarantees of delivery. We do not want to expose this complexity the user of this solution. The RSA specification uses intent for that purpose.

4 Requirements

4.1.1 General

- MSG010 – The solution **MUST** be technology, vendor and messaging protocol independent. MSG030 – The solution **MUST** be configurable (address-space, timeouts, quality of service guarantees)
- MSG050 – The solution **MUST** announce their capabilities/intents to service consumers
- MSG060 – The solution **MUST** provide information about registered channels, client connection states, if available
- MSG070 – The solution **MUST** support the asynchronous programming model

- MSG080 – The solution **MUST** support a client API
- MSG090 – The solution **MUST** respect requested intents
- MSG095 – The solution **MUST** announce its supported intents
- MSG100 – The solution **MUST** fail when encountering unknown or unsupported intents.

4.1.2 Channels

- MSG100 – It **MUST** be possible to asynchronously send messages to a channel.
- MSG120 – The solution **MUST** support systems that support point-to-point channel type
- MSG130 – The solution **MUST** support systems that support the publish-subscribe channel type
- MSG140 – The solution **MUST** support quality of service
- MSG150 – The solution **MUST** support send-and-forget and send-and-forward semantics
- MSG160 – The solution **SHOULD** support Reply-To calls, if possible. For that the solution **MUST** act as caller (publish and subscribe) as well as Reply-To receiver (subscribe on publish)
- MSG170 – The solution **SHOULD** support filter semantics like exchange / routing-key and wildcards for channels
- MSG180 – The solution **MAY** support a do-autocreate as well as do-not-autocreate

Messages

- MSG200 – Messages bodies **MUST** support sending of byte-data
- MSG205 – The implementation **MAY** place limits on the size of the messages that can be send. The existence of a message size limitation for an implementation **SHOULD** be signaled.
- MSG210 – It **SHOULD** be possible to support additional message properties like sequencing and correlation. The implementation **SHOULD** provide access to properties when available.
- MSG220 – The solution **MAY** define a content encoding
- MSG230 – The solution **MAY** support message time-to-live information
- MSG240 – The solution **MAY** support manual acknowledge/reject support for messages
- MSG250 – The solution **MAY** have a journalling support
- *MSG260 – It **MUST** be possible to identify the channel the message was received on*

5 Technical Solution

5.1 Messaging Implementation

An implementer of this specification creates a certain messaging implementation. This implementation is defined using two properties:

- `osgi.messaging.provider` – defining the provider for this implementation (String)
- `osgi.messaging.protocol` – defining the protocols that are supported (String+)

An implementation itself is no service itself. It provides the infrastructure, to create several messaging instances, depending on their configuration. It provides the messaging instances with unique names using the `osgi.messaging.name` property.

To reflect special features, that are supported within this implementation, but not in other implementations, there is also a String+ property, that reflects supported features. Values must be the mandatory features provided, by the specification, but also additional implementation specific features. This property is named: `osgi.messaging.feature`. Possible values are describe with in each chapter of this RFC.

In addition to that, the implementation is responsible for the correct setup of all services and the service properties of messaging instances in respect to the features/intents provided in the instance configurations.

The OSGi Configuration Admin Service must be supported to provide configurations for instances of an implementation.

5.2 Messaging Instance

A messaging instance represents a set of services, that belong to certain parameters. These can reflect a certain connection setup like different user credentials or different client settings regarding caching, worker-thread-behavior.

This is why a connection URI to a broker cannot be seen as unique property for an messaging instance. Therefore, in addition to the provider and protocol properties, an instance must provide the mandatory property `osgi.messaging.name` property for all the provided services.

The messaging instance is represented in via certain services, that are a minimum:

- Messaging Runtime Service – Introspection into the messaging instance
- Subscription – Service to subscribe to channels
- Publisher – Services to publish messages over channels
- Message Context Builder Service

Implementation, that create these services must define the following service properties:

- `osgi.messaging.name` – The unique name of the messaging instance / configuration
- `osgi.messaging.protocol` – The array of protocol strings that are supported by this instance
- `osgi.messaging.feature` – The array of supported feature-strings for the services. The `MessagingRuntimeService` and the `MessageContextBuilder` services must provide all features, that are supported by the implementation. Publisher and Subscription can contain just a subset of the supported features, that are relevant for their corresponding case, these services cover.

5.3 Messaging Runtime Service

Each messaging instance is represented through a runtime service instance. This service allows introspection using DTO's. This enables interested parties to take a look into certain information about the underlying protocol implementation. So it could show a connection status as well as all subscriptions

- `MessagingRuntimeDTO`
- `ChannelDTO`
- `SubscriptionDTO`

Each instance of an messaging implementation has to provide a corresponding Messaging Runtime Service instance, that reflects the state of this instance, with all its subscriptions.

```
/**
 * The MessageServiceRuntime service represents the runtime information of a
 * Message Service instance of an implementation.
 * <p>
 * It provides access to DTOs representing the current state of the connection.
 * <p>
 */
public interface MessageServiceRuntime {

/**
 * Return the messaging instance DTO containing the connection state and subscriptions
 *
 * @return the runtime DTO
 */
MessagingRuntimeDTO getRuntimeDTO();
}
```

5.4 Subscription

When we think about messaging in general, one important part is subscribing to a channel to receive messages. A message subscription can cover different use-cases like

1. expecting a stream of data/messages for a certain channel
2. expecting answers for a reply-to request

The second case converts the reply-to behavior and is described in an own section later in this document. An messaging implementation must at least provide a Subscription Services for the supported protocols and instance.

A message subscription service using the interface below.

```
public interface MessageSubscription {  
  
    /**  
     * Subscribe the {@link PushStream} to the given topic  
     * @param topic the topic string to subscribe to  
     * @return a {@link PushStream} instance for the subscription  
     */  
    public PushStream<Message> subscribe(String channel);  
  
    /**  
     * Subscribe the {@link PushStream} to the given topic with a certain quality of service  
     * @param topic the message topic to subscribe  
     * @param context the optional properties in the context  
     * @return a {@link PushStream} instance for the given topic  
     */  
    public PushStream<Message> subscribe(MessageContext context);  
}
```

This provided methods of this service provide a common use-case for messaging. Both methods return a reactive PushStream of messages for the subscription. It is possible to provide the channel name, you want to subscribe to, directly. Otherwise there is a method to provide the *MessageContext*, which is a subscription configuration with additional properties, that may needed for the subscription of the underlying implementation.

The implementations of theses two methods are always expected to work, no matter how complex the underlying implementation is. The implementation should then provide appropriate default configuration or semantics, for e.g. definition of a channel name

```
@Reference  
private MessageSubscription subscription;  
@Reference  
private MessageContextBuilder mcb;  
  
...  
  
PushStream<Message> ps = subscription.subscribe("foo-topic");  
ps.forEach((m) -> doSomething(m));  
  
MessageContext ctx = mcb.channel("foo-topic").buildContext();  
PushStream<Message> ps = subscription.subscribe(ctx);  
ps.forEach((m) -> doSomething(m));
```

5.5 Message Publishing

To publish messages a messaging instance provides a service using the following interface.

```
public interface MessagePublisher {  
  
    /**  
     * Publish the given {@link Message} to the given topic  
     * contained in the message context of the message  
     * @param message the {@link Message} to publish  
     */  
    public void publish(Message message);  
}
```

```
/**
 * Publish the given {@link Message} to the given topic
 * @param message the {@link Message} to publish
 * @param channel the topic to publish the message to
 */
public void publish(Message message, String channel);

/**
 * Publish the given {@link Message} using the given {@link MessageContext}.
 * The context parameter will override all context information, that come
 * with the message's Message#getContext information
 * @param message the {@link Message} to send
 * @param context the {@link MessageContext} to be used
 */
public void publish(Message message, MessageContext context);

}
```

There are various method signature to publish a message. Like in the message subscription, the *MessageContext* can be used to define additional properties that may needed to publish a message, like e.g. quality of service or also implementation specific options.

```
@Reference
private MessagePublisher publisher;
@Reference
private MessageContextBuilder mcb;

/*
 * Using an existing MessageBuilder, with a channel name
 */
Message message = mcb.content(ByteBuffer.wrap("Foo".getBytes())).buildMessage();
publisher.publish(message, "bar-topic");

/*
 * Using an existing MessageBuilder, define the channel name
 * using the builder.
 */
Message message = mcb.content(ByteBuffer.wrap("Foo".getBytes()))
    .channel("bar-topic")
    .message();
publisher.publish(message);

/*
 * Publish a message with an maybe existing
 * MessageContext that can be a default
 * context provided as service.
 */
@Reference
private MessageContext ctx;
@Reference
private MessagePublisher publisher;
@Reference
private MessageContextBuilder mcb;

Message message = mcb.content(ByteBuffer.wrap("Foo".getBytes())).buildMessage();
publisher.publish(message, ctx);
```

In the latter example the message context parameter will override any information that are eventually set in the messages's context instance.

5.6 Message, Message Context and Message Context Builder

The concept of messaging relies on a message object that hold the payload and a corresponding message context. This pattern is similar to the EventAdmin specification, where you get the topic and properties from the Event. The difference is that the message context contains pre-defined and typed information, than just a topic-string and a properties map.

The Message holds the payload. The Message Context represents the additional messaging properties or header. The Message Context builder is a convenient way to create Message instances or MessageContext instances.

5.6.1 Message

The message is represented using the following interface.

```
/**
 * An message object
 */
public interface Message {

    /**
     * Returns the payload of the message as {@link ByteBuffer}
     * @return the payload of the message as {@link ByteBuffer}
     */
    public ByteBuffer payload();

    /**
     * Returns the message context. This must not be <code>null</code>.
     * @return the {@link MessageContext} instance of this message
     */
    public MessageContext getContext();
}
```

On the publishing side, the creator of the message may want to define some payload and meta data for publishing:

```
@Reference
private MessagePublisher publisher;
@Reference
private MessageContextBuilder mcb;

Message message = mcb.content(ByteBuffer.wrap("Foo".getBytes())) //message payload
    .channel("foo-topic") //message context information
    .contentType("text/plain") //message context information
    .createMessage();
publisher.publish(message);
```

This example creates a message with a containing message context. Where the payload is part of the message instance and the channel definition and content type are part of the message context instance inside the message object:

```
message.getContext().getContentType();//text/plain
```

On the subscriber side, the message instance is created by the subscription service implementation. The consumer of the message may want to know some meta data about the payload:

```
PushStream<Message> ps = subscription.subscribe("foo-topic");
ps.forEach((m)->{
    String contentType = m.getContext().getContentType();
    if ("application/json".equals(contentType)) {
        // do Json stuff
    }
});
```

5.6.2 Message Context

The message context provides meta-information about the message. It can be seen like HTTP Request or Response-Header. These context information are either created from the implementation, when subscribing a message. On the publishing side, these information can be provided by the creator of a message. It is an object that contains common options like content type or quality of service.

The following interface defines a message context:

```
/**
 * Context object that can be used to provide additional properties that
 * can be put to the underlying driver / connection.
 * The context holds meta-information for a message to be send or received
 */
public interface MessageContext {

    /**
     * Returns a channel definition
     * @return a channel definition
     */
    String getChannel();

    /**
     * Returns the content type like a mime-type
     * @return the content type
     */
    public String getContentType();

    /**
     * Returns the content encoding
     * @return the content encoding
     */
    public String getContentEncoding();

    /**
     * Returns the correlation id
     * @return the correlation id
     */
    public String getCorrelationId();

    /**
     * Returns the reply to channel
     * @return the reply to channel
     */
    public String getReplyToChannel();

    /**
     * Returns the options map for additional configurations. The returning map can not be
     * modified anymore
     * @return the options map, must no be null
     */
    public Map<String, Object> getExtensions();
}
```

5.6.3 Message Context Builder

The MessageContextBuilder is a service provided by the instance of an messaging implementation using the following interface:

```
/**
 * Builder for building a {@link Message} or {@link MessageContext} to configure publish or subscription
 * properties
 */
public interface MessageContextBuilder {
```

```
/**
 * Sets the provided {@link MessageContext} instance. If this context is set,
 * calling message context builder functions on this builder will no override
 * the values from the given context.
 * @param context an existing context
 * @return the {@link MessageBuilder} instance
 */
public MessageContextBuilder withContext(MessageContext context);

/**
 * Adds the content to the message
 * @param byteBuffer the content
 * @return the {@link MessageBuilder} instance
 */
public MessageContextBuilder content(ByteBuffer byteBuffer);

/**
 * Adds typed content to the message and maps it using the provided mapping function
 * @param <T> the content type
 * @param object the input object
 * @param contentMapper a mapping function to map T into the {@link ByteBuffer}
 * @return the {@link MessageBuilder} instance
 */
public <T> MessageContextBuilder content(T object, Function<T, ByteBuffer> contentMapper);

/**
 * Defines a reply to address when submitting a reply-to request. So the receiver will
 * know, where to send the reply.
 * @param replyToAddress the reply address
 * @return the {@link MessageContextBuilder} instance
 */
public MessageContextBuilder replyTo(String replyToAddress);

/**
 * Defines a correlation id that is usually used for reply-to requests.
 * The correlation id is an identifier to assign a response to its corresponding request.
 * This options can be used when the underlying system doesn't provide the generation of these
 * correlation ids
 * @param correlationId the correlationId
 * @return the {@link MessageContextBuilder} instance
 */
public MessageContextBuilder correlationId(String correlationId);

/**
 * Defines a content encoding
 * @param content the content encoding
 * @return the {@link MessageContextBuilder} instance
 */
public MessageContextBuilder contentEncoding(String contentEncoding);

/**
 * Defines a content-type like the content mime-type.
 * @param contentType the content type
 * @return the {@link MessageContextBuilder} instance
 */
public MessageContextBuilder contentType(String contentType);

/**
 * Defines a channel name and a routing key
 * @param channelName the channel name
 * @param channelExtension the special key for routing a message
 * @return the {@link MessageContextBuilder} instance
 */
public MessageContextBuilder channel(String channelName, String channelExtension);

/**
```

```
    * Defines a channel name that can be a topic or queue name
    * @param channelName the channel name
    * @return the {@link MessageContextBuilder} instance
    */
    public MessageContextBuilder channel(String channelName);

    /**
     * Adds an options entry with the given key and the given value
     * @param key the option/property key
     * @param value the option value
     * @return the {@link MessageContextBuilder} instance
     */
    public MessageContextBuilder extensionEntry(String key, Object value);

    /**
     * Appends the given options to the context options
     * @param options the options map to be added to the options
     * @return the {@link MessageContextBuilder} instance
     */
    public MessageContextBuilder extensions(Map<String, Object> extension);

    /**
     * Builds the message context
     * @return the message context instance
     */
    public MessageContext buildContext();

    /**
     * Builds the message with a containing context
     * @return the message instance
     */
    public Message buildMessage();
}
```

The service for the message context builder must be registered as a prototype-scoped service, to allow the creation of multiple instance and also releasing them.

Messaging is not only about just sending data over a distributed network. It may happen, that you need certain connection information when the message has arrived. On the other hand, you just want to send/publish one special message, with additional connection information. The message context object is meant for that.

5.6.4 Implementation-specific Message Context Builder

To allow provider specific builder implementations and methods, it is possible to additionally register these instances as a service as well. There must be at least a service for the Message Context Builder.

5.7 Reply-To Behavior

Beside the common publishing and subscriptions there is also a common use-case, the reply-to-behavior.

The following cases can exist:

- *Reply-To-Publisher* - A request message is sent to a certain channel and expects at least one answer-message on a reply-channel.
- *Reply-To-Subscription* - Request-messages are expected on a certain channel and at least one answer message is created by a handler and published back to a reply-channel

Many protocols support these cases, but not all allowing such features out-of the box. So, its not expected that all implementations support these features.

In case that an implementation supports reply-to behavior, it has to provide the services for publishing reply-to-request messages, as well as the infrastructure component to deal with reply-to-subscriptions. The services that must be provided from an implementation are:

- ReplyToPublisher Service – provides interface to submit messages and expect at least one message as response
- ReplyToWhiteboard Service – provides an infrastructure that binds Reply-To-Subscription Services

To find out, if the implementation supports reply-to handling, the MessagingRuntimeDTO can be inspected if it supports the feature named **“replyTo”**. In that case publishing as well subscribing/handling of request must be supported.

There are additional use-cases, reply-to-many publishing and subscription that are optional and provide their own features:

- Reply-To Many Publisher Feature: **“replyToManyPublish”**
- Reply-To Many Subscription Handler: **“replyToManySubscribe”**

5.7.1 Reply-To Publisher

Submitting a reply-to request can be handled using the following interface:

```
public interface ReplyToPublisher {  
  
    /**  
     * Subscribe for single response for a reply-to request  
     * @param requestMessage the request message  
     * @return the {@link Promise} that is resolved, when the answer message has arrived  
     */  
    public Promise<Message> publishAndReply(Message requestMessage);  
  
    /**  
     * Subscribe for single response for a reply-to request  
     * @param message the request message  
     * @param replyToContext the optional properties in the context for the request and response channels  
     * @return the {@link Promise} that is resolved, when the answer has arrived  
     */  
    public Promise<Message> publishAndReply(Message requestMessage, MessageContext replyToContext);  
  
}
```

If an implementation supports the reply-to behavior it must provide a service with this interface. It is expected that the implementation un-subscribes from the reply-to channel as soon as the answer arrives or an error occurs.

The service must provide the `osgi.messaging.feature` service property that defines, if there are certain built-in features available. Possible features and so values for this property in the reply-to context are:

- automatically generate correlation identifiers: `generateCorrelationId`
- automatically generate a response channel: `generateReplyChannel`

If given, the implementation supports these functionalities. That means that it is not necessary, to manually provide these additional properties, when publishing request messages:

```
@Reference(target = "(&(osgi.messaging.feature=generateCorrelationId)  
(osgi.messaging.feature=generateReplyChannel))")  
private ReplyToPublisher replyToPub;
```

```
@Reference
private MessageContextBuilder mcb;

...

Message request = mcb.content(ByteBuffer.wrap("Foo".getBytes()))
    .channel("foo-topic")
    .buildMessage();
Promise<Message> ps = replyToPub.publishAndReply(request);
ps.onSuccess((response) -> doSomething(response));
```

The example above sends the given request message to the *foo-topic* using a correlation id, that is generated by the underlying implementation. The same applies to the reply-to channel.

The response is expected to arrive on the generated reply-to channel. The returned Promise will resolve as soon as the answer arrives on the reply-to channel. The implementation is responsible to subscribe on the reply-to channel and publish the message to the defined channel.

When there are no such features provided by the implementation, all needed information can be additionally provided:

```
@Reference
private ReplyToPublisher replyToPub;
@Reference
private MessageContextBuilder mcb;

...

Message request = mcb.content(ByteBuffer.wrap("Foo".getBytes()))
    .channel("foo-replyTo")
    .correlationId("1234")
    .replyTo("bar-replyToResponse")
    .buildMessage();
Promise<Message> ps = replyToPub.publishAndReply(request);
ps.onSuccess((m) -> doSomething(m));
```

This examples shows, how correlation id and/or reply-channel information can be provided using the messaging context.

5.7.2 Reply-To Many Publisher

An optional extension to the Reply-To Publisher is this kind of publishing, where many response messages are expected for one request message instead of just one.

If an implementation supports the feature “**replyToManyPublish**”, it must provide a service with the following interface:

```
public interface ReplyToManyPublisher {

    /**
     * Subscribe for multiple responses on a reply-to request. This call is similar to the simple
     * subscription to a topic. This request message contains payload and parameters,
     * like e.g. correlation id or response channel for the request, response setup.
     * @param requestMessage the request message
     * @return the {@link PushStream} for the answer stream
     */
    public PushStream<Message> publishAndReplyMany(Message requestMessage);

    /**
     * Publish a request and await multiple answers for that request.
     * This call is similar to the simple subscription on a
     * topic. This request message contains payload and parameters, like e.g. correlation id
     */
}
```

```

* or response channel for the request, response setup.
* @param requestMessage the request message
* @param replyToContext the properties in the context for the request and response setup
* @return the {@link PushStream} for the answer stream
*/
public PushStream<Message> publishAndReplyMany(Message requestMessage, MessageContext replyToContext);
}

```

It follows the same rules, like for the ordinary reply-to publishing mechanism. The implementation is responsible to provide the correct handling of the push stream regarding error handling and closing of the stream.

This feature may need additional setup for the implementation, e.g. in form of special message headers. This must be hidden from the user by the implementation so that no further user-interaction is needed.

5.7.3 Reply-To Subscription Whiteboard

Implementations that support reply-to behavior have to provide a *ReplyToWhiteboard* runtime that binds *ReplyToSingleSubscriptionHandler* and, if supported *ReplyToManySubscriptionHandler*.

This runtime is responsible for subscribing to a certain channel, provided as services property of the bound handlers. It then receives the requests and also delegates them to the bound response reply-to handler services. After that the response message or messages have to be published to the correct reply-to address. This whiteboard is provided using the marker interface:

```

public interface ReplyToWhiteboard {

    // Add a runtime DTO here?!

}

```

This whiteboard implementation must also take care to pre-configure a response message context builder instance, that is later provided to the registered reply-to subscriptions handlers as parameter.

A simple example of such a whiteboard, that is provided from the implementer of an protocol, could look like this:

```

@Component(property = {"messaging.name=myFooBarWhiteboard", "foo=bar"})
public class FooBarReplyToWhiteboardImpl implements ReplyToWhiteboard {

    @Reference(target = "(osgi.messaging.protocol=mqtt)")
    private MessageSubscription fooSubscription;
    @Reference(target = "(osgi.messaging.protocol=mqtt)")
    private MessagePublisher barPublisher;
    @Reference
    private MessageContextBuilder mcb;

    private ReplyToSingleSubscriptionHandler handler;

    @Reference(policy = ReferencePolicy.DYNAMIC, target = "(messaging.name=myFooBarHandler)")
    public void setSubscriptionHandler(ReplyToSingleSubscriptionHandler handler, HashMap<String, Object>
properties) {
        this.handler = handler;
        String subChannel = (String) properties.get("osgi.messaging.replyTo.channel");
        fooSubscription.subscribe(subChannel).map(this::handleResponse).forEach(barPublisher::publish);
    }

    private Message handleResponse(Message request) {
        MessageContext requestCtx = request.getContext();
        String channel = requestCtx.getReplyToChannel();
        String correlation = requestCtx.getCorrelationId();
        MessageContextBuilder responseCtxBuilder = mcb.channel(channel).correlationId(correlation);
        try {

```

```

        return handler.handleResponse(request, responseCtxBuilder).getValue();
    } catch (Exception e) {
        Message error = responseCtxBuilder.content(ByteBuffer.wrap(e.getMessage().getBytes()))
            .buildMessage();
        return error;
    }
}

```

This very simple implementation injects a MQTT subscriber and publisher. The handler *ReplyToSingleSubscriptionHandler* was provided by the user and is then also injected into this component. As soon as the handler is injected the subscription to the request topic is established. In the PushStream pipeline the mapping is done using the *handleResponse* method, that delegates to the injected subscription handler that creates the response message. This response message is then forwarded to the publisher, that submits the response.

5.7.4 Reply-To Subscription

The previous section covered the “sending a request” side of the reply-to scenario. This section will cover the “handle request” part.

The reply-to subscriptions have to be registered. An implementation must provide a component/whiteboard, that provides an infrastructure to subscribe on certain *request* channels and listens for incoming requests. In that case it is expected, that the request are handled and provide response messages, that is/are then published back to the *reply-to* channel, provided in the request message context. In addition to that the correlation must be handled correctly as well.

This whiteboard consumes reply-to subscription handler interfaces:

- ReplyToSingleSubscriptionHandler
- ReplyToManySubscriptionHandler

```

public interface ReplyToSingleSubscriptionHandler {

    /**
     * Creates a {@link Promise} for response {@link Message} for the incoming request
     * {@link Message}.
     * The promise will be resolved, as soon as the execution completed successfully.
     * Errors during the handling are delegated using the promise fail, behavior
     * The response message context builder is pre-configured.
     * Properties like the channel and correlation are already set correctly to the builder.
     * @param requestMessage the {@link Message}
     * @param responseBuilder the builder for the response message
     * @return the response {@link Message}, must not be null
     */
    public Promise<Message> handleResponse(Message requestMessage, MessageContextBuilder responseBuilder);
}

```

If a system wants to act as party that listens to message request and handles the response, it must register an implementation of these interfaces. To determine the correct reply-to subscription whiteboard a LDAP filter style target filter must be provided as service property `osgi.messaging.replyToSubscription.target`. Beside that the channels to listen on for this handler have to be defined using the String+ service property `osgi.messaging.replyToSubscription.channel`.

```
Dictionary<String, Object> properties = new Hashtable<String, Object>();
```

```
properties.put("osgi.messaging.replyToSubscription.target", "(&(osgi.messaging.name=foo)
(osgi.messaging.protocol=mqtt)(osgi.messaging.feature=replyTo))");
properties.put("osgi.messaging.replyToSubscription.channel", "foo-request-topic");

ReplyToSingleSubscriptionHandler handler = new MySingleSubscriptionHandler();

ServiceRegistration<ReplyToSingleSubscriptionHandler> myHandlerReg
    = context.registerService(ReplyToSingleSubscriptionHandler.class, handler, properties);
```

The above example registers a single subscription handler, that accepts request, that come in over the *foo-request-topic*. Each time a message arrives, the *handleResponse* method is called with the incoming message instance. The provided message context builder is already configured in respect to certain properties like correlation id and reply-channel.

When providing the property `osgi.messaging.replyToSubscription.replyChannel`, like in the following example, the handler would only be called, if channel and replyChannel service properties are the same like the definition in the message context. This allows to place handlers for a very special setup.

Each registered handler is reflected as `ReplyToSubscriptionDTO` in the `MessagingRuntimeDTO`.

5.7.5 Reply-To-Many Subscriptions

This kind of subscription is a special use-case. One request message is answered by multiple response messages. This case must work, if the feature “**replyToManySubscribe**” is provided. The subscription handler interface is:

```
public interface ReplyToManySubscriptionHandler {

    /**
     * Creates {@link PushStream} of response {@link Message}s for an incoming request {@link Message}.
     * The response builder is pre-configured. Properties like the channel and correlation
     * are already set correctly to the builder.
     * @param requestMessage the {@link Message}
     * @param responseBuilder the builder for the response message
     * @return the response {@link PushStream}, must not be null
     */
    public PushStream<Message> handleResponses(Message requestMessage, MessageBuilder responseBuilder);
}
```

The handling of the use-case works like the one for a single response. So the services must be provided by the user. In addition to that, the implementation is also responsible to handle the push stream events for closing a stream or error correctly

5.8 Acknowledgment

Messaging implementations also support acknowledgment and rejection of messages. This can lead to the situation, that a rejection of a message will cause re-delivery of the message in the system. There may be different reasons to influence the decision, if or when to acknowledge a message or not.

If an implementation acknowledgment, the feature “**acknowledge**” must be provided.

5.8.1 Acknowledge Message Context and Builder

The `MessageContextBuilder` must be adaptable to the following interface:

```
public interface AcknowledgingMessageContextBuilder extends MessageContextBuilder{
```

```
/**
 * A consumer that is called to do custom logic. It gets the {@link Message} as parameter.
 * Using the {@link MessageContext#getAcknowledgeHandler()} from the message, enables implementers
 * to manually trigger
 * This handler is protocol implementation specific.
 * Using this handler, the user can decide whether to acknowledge or reject the message.
 *
 * This handler is only called, when the autoAcknowledge is set to <code>false</code>
 * @param acknowledgeHandler the consumer that gets the {@link Message} as parameter
 * @return the {@link AcknowledgingMessageContextBuilder} instance
 */
public AcknowledgingMessageContextBuilder handleAcknowledge(Consumer<Message> acknowledgeHandler);

/**
 * Defines a {@link Predicate} to test receiving message to either acknowledge or reject.
 * This filter only takes effect, if auto-acknowledge is set to <code>true</code>.
 * If the test of the predicate is <code>true</code>, the message will be acknowledged, otherwise
rejected using the
 * implementation specific logic.
 * If postAcknowledge or/and postReject consumers are set, they will be called after that.
 * @param ackFilter the predicate to test the message
 * @return the {@link AcknowledgingMessageContextBuilder} instance
 */
public AcknowledgingMessageContextBuilder filterAcknowledge(Predicate<Message> ackFilter);

/**
 * Defines a LDAP style target filter for {@link Predicate} that are registered as an OSGi service.
 * This filter only takes effect, if auto-acknowledge is set to <code>true</code>.
 * If the test of the predicate is <code>true</code>, the message will be acknowledged, otherwise
rejected using the
 * implementation specific logic.
 * If postAcknowledge or/and postReject consumers are set, they will be called after that.
 * @param ackFilter the predicate to test the message
 * @return the {@link AcknowledgingMessageContextBuilder} instance
 */
public AcknowledgingMessageContextBuilder filterAcknowledgeTarget(String ackFilterTarget);

/**
 * A consumer that is called to do the custom logic of rejected messages.
 * That this consumer is called, either an acknowledge filter needs to be set or otherwise the
acknowledge
 * handler needs to be defined. Only in this cases a decision can be made, to reject a message.
 *
 * If auto-acknowledge is active and the acknowledge filter is set, this consumer is called after
calling the internal
 * protocol specific message rejection logic.
 * If auto-acknowledge is not active and the acknowledge handler is defined, this consumer is called
after
 * invoking the {@link AcknowledgeHandler#reject()} method.
 * An acknowledge filter needs to be set and test a message as rejected to invoke this
 * consumer after the implementation specific rejection.
 * @param rejectConsumer
 * @return the {@link AcknowledgingMessageContextBuilder} instance
 */
public AcknowledgingMessageContextBuilder postReject(Consumer<Message> rejectConsumer);

/**
 * A consumer that is called to do custom logic of acknowledged messages. If a acknowledge filter is
set and
 * auto-acknowledgement is not active, this method will be called to invoke the acknowledge process.
 *
 * Usually the implementations will provide their default behavior to execute the acknowledgement.
Providing
 * this consumer will not override this behavior. Instead the consumer is called afterwards.
 * @param ackConsumer the consumer
 * @return the {@link AcknowledgingMessageContextBuilder} instance

```

```

*/
public AcknowledgingMessageContextBuilder postAcknowledge(Consumer<Message> ackConsumer);

/**
 * Set to <code>true</code>, to take profit of the underlying functionality
 * of automatically acknowledging messages that are received.
 * In case an acknowledge filter is provided, the messages are automatically
 * acknowledged, when the predicate test returns <code>true</code>. Otherwise the
 * message will be automatically rejected.
 *
 * This flag is defaulted to <code>false</code>.
 * @param autoAck the auto-acknowledge flag
 * @return the {@link AcknowledgingMessageContextBuilder} instance
 */
public AcknowledgingMessageContextBuilder autoAcknowledge(boolean autoAck);
}

```

The builder can be used to define custom handling for acknowledgment and rejection.

Implementations must also provide a message context that also implements the following interface:

```

public interface AcknowledgeMessageContext {

    /**
     * Returns the acknowledge handler
     * @return the acknowledge handler instance
     */
    AcknowledgeHandler getAcknowledgeHandler();
}

```

5.8.2 Acknowledge Filter Predicate

Acknowledgement usually happens directly when the implementation receives the message. So this logic is called, before messages where published to the PushEventSource.

```

MessageContextBuilder mcb = ...;
AcknowledgingMessageContextBuilder amcb = mcb.adapt(AcknowledgingMessageContextBuilder.class);

MessageContext ctx = amcb.autoAcknowledge(true)
    .filterAcknowledge(this::isGoodMessage)
    .postAcknowledge((m)->System.out.println("Store Acknowledged Message"))
    .postReject((m)->System.out.println("Log Rejected Message"))
    .channel("foo-topic")
    .buildContext();

MessageSubscription subscription = ...;
PushStream<Message> ps = subscription.subscribe(ctx);

```

This configuration uses an automatically acknowledgment/rejection of messages, depending on the result of the provided filter result of *isGoodMessage* . Right after doing the internal acknowledgment or rejection, the provided *postAcknowledge* or *postReject* consumer callbacks are executed. These post-methods are useful, to determine, which messages are acknowledged or rejected. At least you can log these messages.

5.8.3 Acknowledge Predicate Target

Like in the example before, the used predicate implementation can also be registered as an OSGi service. Taken this service component:

```

@Component(property = "foo=bar")
public class JsonPredicate implements Predicate<Message> {

    @Override

```



```

public boolean test(Message m) throws Exception {
    return m.getContext().getContentType().equals("application/json");
}
}

```

The acknowledge message context can be created like this, to use the registered predicate:

```

MessageContext ctx = amcb.autoAcknowledge(true)
    .filterAcknowledgeTarget("(foo=bar)")
    .postAcknowledge((m)->System.out.println("Store Acknowledged Message"))
    .postReject((m)->System.out.println("Log Rejected Message"))
    .channel("foo-topic")
    .buildContext();

```

```

MessageSubscription subscription = ...;
PushStream<Message> ps = subscription.subscribe(ctx);

```

5.8.4 Programmatic Acknowledgment

There is also a possibility to manually do an acknowledgment or rejection. For this there is this interface:

```

/**
 * Handler interface to acknowledge or reject a message. Messaging provider implementations use this
 * interface to provide logic to acknowledge a message for their protocol.
 *
 * This interface is therefore not meant to be implemented by users.
 */
public interface AcknowledgeHandler {

    /**
     * Acknowledge the message, that is returned by the getMessage method
     */
    public void acknowledge();

    /**
     * Reject the message, that is returned by the getMessage method
     */
    public void reject();

}

```

This interface needs to be provided by the implementation. It delegates acknowledge or reject calls, to the implementation specific logic.

```

MessageContextBuilder mcb = ...;
AcknowledgingMessageContextBuilder amcb = mcb.adapt(AcknowledgingMessageContextBuilder.class);
MessageContext ctx = amcb.handleAcknowledge((m, h)->{
    MessageContext context = m.getContext();
    if (isGoodMessage(m)) {
        h.acknowledge();
    } else {
        h.reject();
    }
})
    .postAcknowledge((m)->System.out.println("Store Acknowledged Message"))
    .postReject((m)->System.out.println("Log Rejected Message"))
    .channel("foo-topic")
    .buildContext();

MessageSubscription subscription = ...;
PushStream<Message> ps = subscription.subscribe(ctx);

```

This configuration uses the programmatic way to decide for acknowledging or rejecting messages for channel *foo-topic*. The *handleAcknowledge* method provides the message and the *AcknowledgeHandler* instance as parameters. The latter can be used to manually trigger the acknowledgement/rejection.

Right after that, the provided *postAcknowledge* or *postReject* consumer are called.

5.8.5 Inner-PushStream Acknowledgment

As described before, the acknowledgment/rejection happens, before the message enter the PushStream. Sometimes the point of acknowledging needs to be after executing some operations within the PushStream. The following configuration shows how consumers can do an acknowledgment at a later point:

```
MessageContextBuilder mcb = ...;
AcknowledgingMessageContextBuilder amcb = mcb.adapt(AcknowledgingMessageContextBuilder.class);
MessageContext ctx = amcb.postAcknowledge((m)->System.out.println("Store Acknowledged Message"))
    .postReject((m)->System.out.println("Log Rejected Message"))
    .channel("foo-topic")
    .buildContext();

MessageSubscription subscription = ...;
PushStream<Message> ps = subscription.subscribe(ctx);
ps.map(this::doSomething).forEach((m)->{
    AcknowledgeMessageContext amc = (AcknowledgeMessageContext) m.getContext();
    AcknowledgeHandler h = amc.getAcknowledgeHandler();
    if (isGoodMessage(m)) {
        h.acknowledge();
    } else {
        h.reject();
    }
});
```

The AcknowledgeHandler can also get from the message context. In the example the acknowledgment is done within the PushStream. After that the *postAcknowledge* / *postReject* methods are called.

6 Data Transfer Objects

6.1 Messaging Runtime DTO

The following DTO defines the state of an messaging instance:

```
/**
 * Represents the messaging instance DTO
 *
 */
public class MessagingRuntimeDTO extends DTO {

    /**
     * The DTO for the corresponding {@code MessageServiceRuntime}. This value is
     * never {@code null}.
     */
    public ServiceReferenceDTO serviceDTO;

    /**
     * The connection URI
     */
    public String connectionURI;

    /**
     * Implementation provider name
     */
}
```

```
*/
public String                providerName;

/**
 * The supported protocols
 */
public String[]              protocols;

/**
 * The instance id
 */
public String                instanceId;

/**
 * The set of features, that are provided by this implementation
 */
public String[]              features;

/**
 * DTO for all subscriptions
 */
public SubscriptionDTO[]      subscriptions;

/**
 * DTO for all reply-to subscriptions
 */
public ReplyToSubscriptionDTO[] replyToSubscriptions;
}
```

6.2 ChannelDTO

The ChannelDTO describes a channel, which can be a topic or queue. It additionally contains the possibility to define extensions, beside the channel name. This can be information like a routing-key.

```
/**
 * A {@link DTO} that defines a channel with the possibility to provide additional
 * channel information like routing keys.
 */
public class ChannelDTO extends DTO {

    /**
     * The name of the channel
     */
    public String                name;

    /**
     * A possible extension to a channel like a routing key
     */
    public String                extension;

    /**
     * true, if the channel is connected
     */
    public boolean               connected;
}
```

6.3 Subscription DTO

The subscription DTO represents the inner state of a subscription for a certain channel.

```
/**
```

```
* Represents a subscription instance DTO
*
*/
public class SubscriptionDTO extends DTO {

/**
 * The DTO for the corresponding {@code Subscription} service. This value is
 * never {@code null}.
 */
public ServiceReferenceDTO          serviceDTO;

/**
 * DTO that describes the channel for this subscription
 */
public ChannelDTO                  channel;

}
```

6.4 Reply-To Subscription DTO

This DTO represents a reply-to subscription. This DTO is specific for the reply-to behavior and contains the inner state and configuration.

```
/**
 * Represents a subscription for the reply to request DTO
 *
 */
public class ReplyToSubscriptionDTO extends DTO {

/**
 * The DTO for the corresponding {@code Subscription} service. This value is
 * never {@code null}.
 */
public ServiceReferenceDTO          serviceDTO;

/**
 * DTO that describes the channel for the request subscription
 */
public ChannelDTO                  requestChannel;

/**
 * DTO that describes the channel for publishing the response
 */
public ChannelDTO                  responseChannel;

/**
 * The DTO of the registered handler, that executes the logic for this subscription
 */
public ServiceReferenceDTO          handlerService;

/**
 * Flag that shows, if correlation id generation is active or not
 */
public boolean                     generateCorrelationId = false;

/**
 * Flag that shows, if reply channel generation is active or not
 */
public boolean                     generateReplyChannel = false;

}
```

7 Javadoc

Please include Javadoc of any new APIs here, once the design has matured. Instructions on how to export Javadoc for inclusion in the RFC can be found here: <https://www.osgi.org/members/RFC/Javadoc>

8 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10 Document Support

10.1 References

- [1]. Enterprise Integration Pattern: Designing, Building, and Deploying Messaging Solutions. Gregor Hohpe, Bobby Woolf. ISBN 0-133-06510-7.

*Add references simply by adding new items. You can then cross-refer to them by choosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. **STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

10.2 Author's Address

Name	Mark Hoffmann
Company	Data In Motion Consulting GmbH
Address	Kahlaische Strasse 4, 07745 Jena
Voice	
e-mail	m.hoffmann@data-in-motion.biz

10.3 Acronyms and Abbreviations

10.4 End of Document