Messaging

Draft

22 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,

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

0.5 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	*January 2020*	*Mark Hoffmann, some initial API*
1.1	*April 2020*	*Mark Hoffmann, Subscriptions, Reply-To, Ack, Builders*

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 are 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.

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.

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 RFP. 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 Message and MessageContext

The concept of the messaging specification relies on a message object and a corresponding message context. A message is bound to a message context. This pattern is similar to the EventAdmin specification, where you get the topic and properties from the Event, you can get the context from a message. The difference is that the MessageContext contains more and typed information, than just a topic and a properties map.

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

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

        /**
         * Returns <code>true</code>, if the message is acknowledged. In case there is no acknowledgment
         * handling available, the method always returns <code>true</code>
         * @return <code>true</code>, if the message is acknowledged, <code>false</code> if rejected
         */
        public boolean isAcknowledged();

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

}
```

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.

```
/**
 * 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
         */
        ChannelDTO 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 ChannelDTO getReplyToChannel();

        /**
         * Returns <code>true</code>, if auto-acknowledgment is activated.
         * @return <code>true</code>, if auto-acknowledgment is activated.
         */
```

```
        public boolean isAutoAcknowledge();

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

        /**
         * Returns the handler, to enable manual acknowledgement or rejection.
         * This handler is always available when auto-acknowledge is not active, which is the default.
         * It may be possible that {@link
MessageContextBuilder#handleAcknowledge(org.osgi.util.function.Consumer)} is
         * set, in this case calling the {@link AcknowledgeHandler#acknowledge()} or {@link
AcknowledgeHandler#reject()}
         * methods will cause an {@link IllegalStateException}
         * @return the handler or <code>null</code>, if auto-acknowledge is active
         */
        public AcknowledgeHandler getAcknowledgeHandler();

}
```

5.2 MessageBuilder and MessageContextBuilder

Because the Message and MessageContext objects are fundamental for this specification, we need a way to create them. When receiving a message, the instances for *Message* and *MessageContext* are created by the implementation. When publishing a message, the instances need to be created by the user. Therefore corresponding builder interfaces exist.

Where the implementations of the service interfaces can be specific to a certain protocol, the message context implementation can be as well protocol specific and hold implementation-specific properties.

```
/**
 * Builder for the {@link MessageContext} to configure publish or subscription properties
 */
public interface 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 MessageContextBuilder} instance
         */
        public MessageContextBuilder handleAcknowledge(Consumer<Message> acknowledgeHandler);

        /**
         * Defines a {@link Predicate} to test receiving message to either acknowledge or reject.
         * This filter only effects, 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 MessageContextBuilder} instance
         */
```

```
        public MessageContextBuilder filterAcknowledge(Predicate<Message> ackFilter);

        /**
         * 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 MessageContextBuilder} instance
         */
        public MessageContextBuilder 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 MessageContextBuilder} instance
         */
        public MessageContextBuilder 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 MessageContextBuilder} instance
         */
        public MessageContextBuilder autoAcknowledge(boolean autoAck);

        /**
         * Defines a reply to address when submitting a reply-to request. So the receiver will
         * knows, 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 routingKey the special key for routing a message
        * @return the {@link MessageContextBuilder} instance
        */
       public MessageContextBuilder channel(String channelName, String routingKey);

       /**
        * 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 optionEntry(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 options(Map<String, Object> options);

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

}
```

This builder can be used to create a *MessageContext* instance. To create a Message instance to corresponding *MessageBuilder* can be used. It inherits from the *MessageContext*, to make the configuration seamless, without having two builders.

```
/**
 * Builder for the {@link Message} to configure content
 */
public interface MessageBuilder {

       /**
        * 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 MessageBuilder withContext(MessageContext context);

        /**
         * Adds the content to the message
         * @param byteBuffer the content
         * @return the {@link MessageBuilder} instance
         */
        public MessageBuilder 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> MessageBuilder content(T object, Function<T, ByteBuffer> contentMapper);

        /**
         * Builds a message instance
         * @return the message instance
         */
        public Message buildMessage();

}
```

It

5.3 Message Runtime Service

To get runtime information about a certain connection of an implementation, a *MessageRuntimeService* exists that exposes connection information.

```
/**
 * The MessageServiceRuntime service represents the runtime information of a
 * Message Service implementation.
 * <p>
 * It provides access to DTOs representing the current state of the connection.
 * <p>
 * The MessageServiceRuntime service must be registered with the
 * {@link MessageConstants#MESSAGE_CONNECTION_URI} service
 * property.
 *
 */
@ProviderType
public interface MessageServiceRuntime {

        /**
         * Return the runtime DTO containing the connection state
         *
         * @return the runtime DTO
         */
        RuntimeDTO getRuntimeDTO();

}
```

TBD: Write more text here

5.4 Subscription

Service interfaces for subscribing on channels and publishing of messages are separated into own interfaces. That makes it possible just to implement either subscription or publishing.

The proposed interface defines a message subscription. I covers two major cases:

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

```
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 topic);

	/**
	 * 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);

	/**
	 * Subscribe for single answer on a request
	 * @param requestMessage the request message
	 * @return the {@link Promise} that is resolved, when the answer message has arrived
	 */
	public Promise<Message> subscribeForRequest(Message requestMessage);

	/**
	 * Subscribe for single answer on a reply-to request
	 * @param message the request message
	 * @param subscriptionContext the optional properties in the context for the subscription
	 * @return the {@link Promise} that is resolved, when the answer has arrived
	 */
	public Promise<Message> subscribeForRequest(Message requestMessage, MessageContext
subscriptionContext);

	/**
	 * Subscribe for multiple answers on a reply-to request. This call is similar to the simple
subscription on a
	 * topic. This request message contains parameters for the subscription
	 * @param requestMessage the request message
	 * @param subscriptionContext the optional properties in the context for the subscription
	 * @return the {@link PushStream} for the answer stream
	 */
	public PushStream<Message> subscribeStreamForRequest(Message requestMessage, MessageContext
subscriptionContext);

}
```

5.4.1 Simple Subscription

The most common use-case for messaging is a simple subscription to a channel. The first two `subscribe` methods cover this case. The subscription configuration can be handed over by providing the *MessageContext* instance.

```
MessageSubscription subscription = ...;
PushStream<Message> ps = subscription.subscribe("foo-topic");
ps.forEach((m)→doSomething(m));
```

```
MessageContextBuilder mcb = ...;
MessageContext ctx = mcb.channel("foo-topic").buildContext();

MessageSubscription subscription = null;
PushStream<Message> ps = subscription.subscribe(ctx);
ps.forEach((m)→doSomething(m));
```

5.4.2 Reply-To Subscription

Another use-case for a subscription is the reply-to scenario. Following a asynchronous request-response pattern. Because there are two sides in the reply-to scenario, the provided methods cover the side that provides the request and waits for at least one answer.

The second side, where a worker receives an request and has to create the answer is covered in an own section.

```
MessageContextBuilder mcb = ...;
MessageContext ctx = mcb.channel("foo-topic")
               .correlationId("1234")
               .replyTo("bar-topic")
               .buildContext();

MessageBuilder mb = ...;
Message request = mb.content(ByteBuffer.wrap("Foo".getBytes())).buildMessage();;
MessageSubscription subscription = ...;
Promise<Message> ps = subscription.subscribeForRequest(request, ctx);
ps.onSuccess((m)->doSomething(m));
```

The example above send the given request message to the *foo-topic* using the correlation id. The response is expected to arrive on the *bar-topic*. The returned Promise will resolve as soon as the answer arrives on *bar-topic*. The implementation is responsible to subscribe on the reply-to channel and publish the message to the defined channel.

It is also expected that the implementation un-subscribes from the reply-to channel as soon as the answer arrives or an error occurs.

There is a second reply-to behavior. Where the request is sent to the channel, but the response is expected to be a stream. This is like subscribing on a channel with parameters:

```
MessageContextBuilder mcb = ...;
MessageContext ctx = mcb.channel("foo-topic")
               .correlationId("1234")
               .replyTo("bar-topic")
               .buildContext();

MessageBuilder mb = ...;
Message request = mb.content(ByteBuffer.wrap("Foo".getBytes())).buildMessage();;
MessageSubscription subscription = ...;
Promise<Message> ps = subscription.subscribeForRequest(request, ctx);
ps.onSuccess((m)→doSomething(m));
```

5.4.3 Reply-To Response Handler

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

There must be a service that subscribes on a certain channel and listens for incoming request. In that case it is expected to handle the request and provide a response, that is then published back to the *reply-to* channel, provided in the request message context. In addition to that the correlation must be handled correct as well.

There must be a service registered, that is configured in a way, that is capable to listen for requests. This service must implement the corresponding interface:

```
public interface ReplyToResponseHandler {

	/**
	 * Creates a response {@link Message} for the 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 Message}, must not be null
	 */
	public Message handleResponse(Message requestMessage, MessageBuilder responseBuilder);

	/**
	 * Creates {@link PushStream} of response {@link Message} for the 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);

}
```

Implementations have to provide a *ReplyToWhiteboard* runtime that binds a *ReplyToResponseHandler*. This runtime is responsible for subscribing to a certain channel to receive the requests and also delegating them to the bound response handler. After that the response message or messages have to published to the correct reply-to address.

```
public interface ReplyToWhiteboard {

	// Add a runtime DTO here?!

}
```

The following code-snippes outline an example of a whiteboard implementation and its handler. At first a very simple whiteboard implementation:

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

	@Reference
	private MessageSubscription fooSubscription;
	@Reference
	private MessagePublisher barPublisher;
	@Reference(target = "(messaging.name=myFooBarHandler)")
	private ReplyToResponseHandler handler;
	private MessageContextBuilder mcb;
	private MessageBuilder mb;

	@Activate
	public void activate() {
		fooSubscription.subscribe("foo-
topic").map(this::handleResponse).forEach(barPublisher::publish);
	}

	private Message handleResponse(Message request) {
		MessageContext requestCtx = request.getContext();
		String channel = requestCtx.getReplyToChannel().name;
		String correlation = requestCtx.getCorrelationId();
		MessageContext responseCtx =
mcb.channel(channel).correlationId(correlation).buildContext();
		return handler.handleResponse(request, mb.withContext(responseCtx));
	}

}
```

The corresponding handler to creates the reponse is usually implemented by the user and could look like this:

```
@Component(property = {"messaging.name=myFooBarHandler"})
public class FooBarHandler implements ReplyToResponseHandler {

	@Override
	public Message handleResponse(Message requestMessage, MessageBuilder responseBuilder) {
		String t = new String(requestMessage.payload().array());
		t += "-bar";
		return responseBuilder.content(ByteBuffer.wrap(t.getBytes())).buildMessage();
	}

	@Override
	public PushStream<Message> handleResponses(Message requestMessage, MessageBuilder
responseBuilder) {
		return null;
	}

}
```

5.4.4 Acknowledgment and Rejection

Different implementations support different kinds of handling acknowledgment. There may be reasons to influence the decision, if or when to acknowledge or not. The following two examples show a configuration to add custom logic into the acknowledge or reject process.

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

```
MessageContextBuilder mcb = ...;
MessageContext ctx = mcb.channel("foo-topic")
	.autoAcknowledge(true)
```

```
        .filterAcknowledge(this::isGoodMessage)
        .postAcknowledge((m)->System.out.println("Store Acknowledged Message"))
        .postReject((m)->System.out.println("Log Rejected Message"))
        .buildContext();

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

This configuration uses the automatically acknowledgment or 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 are called.

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

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

This configuration uses the programmatic way to decide for acknowledgment or rejection of messages. Right after that, the provided *postAcknowledge* or *postReject* consumer are called.

Sometimes the point of acknowledging needs to be after executing some operation within the PushStream. The following configuration outlines this use case:

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

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

5.5 Message Publishing

To publish messages to a channel we use the following interface:

```
public interface MessagePublisher {

        /**
         * Publish the message created in the builder function to the configuration
         * @param builderFunction the function to create a {@link Message} from the builder
         */
```

```
        public void publish(Function<MessageBuilder, Message> builderFunction);

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

        /**
         * Publish the given {@link Message} using the given {@link MessageContext}
         * @param message the {@link Message} to send
         * @param context the {@link MessageContext} to be used
         */
        public void publish(Message message, MessageContext context);

}
```

5.6 Configuration

This specification is meant to provide a convenient way of using messaging within OSGi. The *Configurator* and *ConfigurationAdmin* specifications are perfect wing-man to achieve a user-friendly handling.

The implementation should to define a subscription or publisher using a certain default values for the *MessageContext.* A sample configuration could look like this.

```
{
  ":configurator:resource-version": 1,

        "MQTTMessaging~foosub":
        {
            "messaging.name": "foo-subscription",
            "messaging.connectUri”: "mqtt://my-host",
            "messaging.type" : "subscribe",
            "messaging.ctx.contentType" : "application/json",
            "messaging.ctx.channel" : "foo-topic",
            "messaging.ctx.acknowledgeFilter" : "(filterName=myfancyFilter)"
        },
        "MQTTMessaging~barbup":
        {
            "messaging.name": "bar-publish",
            "messaging.connectUri": "mqtt://my-host",
            "messaging.type" : "publish",
            "messaging.ctx.contentType" : "application/json",
            "messaging.ctx.channel" : "bar-topic"
        }
}
```

In this case, the values for the *channel* and *content-type* are predefined. The *Predicate* for the acknowledge filter is expected to be a service with the provided target binding:

```
@Component(property = {"filterName=myfancyFilter"})
public class MyFilter implements Predicate<Message> {

        @Override
        public boolean test(Message m) {
                return isGoodMessage(m);
        }

        ...
```

```
}
```

Then the *MessageSubscription* service can be injected like this:

```
@Reference(target = "(messaging.name=foo-subscription)")
private MessageSubscription fooSubscription;
```

It is then not necessary to provide a context, because the pre-configured setup is taken:

```
PushStream<Message> ps = fooSubscription.subscribe(null);
ps.forEach((m)→doSomething(m));
```

The similar behavior for the *MessagePublisher* it looks like this:

```
@Reference(target = "(messaging.name=bar-publish)")
private MessagePublisher barPublisher;

…

MessageBuilder mb = ...;
Message message = mb.content(ByteBuffer.wrap("Bar".getBytes())).buildMessage();
barPublisher.publish(message, null);
```

In case the context parameter for this method is used, the provide *MessageContext* will be taken. It will override the predefined values.

If the implementation has no appropriate, predefined *MessageContext* to be used, the subscribe method will return with an **TBD** exception.

6 Data Transfer Objects

6.1 RuntimeDTO

```
/**
 * Represents the message runtime DTO
 *
 */
public class RuntimeDTO extends DTO {

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

}
```

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;

}
```

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 chosing <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