Messaging

Draft

26 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

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*
1.2	*August 2020*	*Mark Hoffmann, Publisher, Subscriptopn Rework*

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 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 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 Messaging Implementation

An implementor for this specification creates a messaging implementation. This implementation is defined using two properties:

- **`osgi.messaging.provider`** – defining the provider for this implementation
- **`osgi.messaging.protocol`** – defining the protocols that are supported

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.

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

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 a s 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();

}
```

The following DTO’s defining the state of an 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;

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

}
```



```java
/**
 * 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;

	/**
	 * <code>true</code>, if the channel is connected
	 */
	public boolean			connected;

}

/**
 * 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;

}
```

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.

```java
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);

}
```

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 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}.
```

```
         * 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 *MessagingContext* can be used to define additional properties that may needed to publish a message, like e.g. qualitiy 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 playload of the message as {@link ByteBuffer}
	 * @return the playload 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("application/json")       //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();//application/json
```

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 <code>null</code>
	 */
	public Map<String, Object> getOption();

}
```

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
     * 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 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 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();

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

        /**
         * Returns <code>true</code>, if this context builder can adapt to different builder, that
inherits
         * from {@link MessageContextBuilder}
         * @param <M> the concrete message context builder type
         * @param builderClass he concrete message context builder class
         * @return <code>true</code>, if this build can adapt to this type, <code>false</code>, if not
         */
        public <M extends MessageContextBuilder> boolean isAdaptable(Class<M> builderClass);

        /**
         * Returns the adapted message builder instance
         * @param <M> the concrete message context builder type
         * @param builderClass he concrete message context builder class
         * @return the builder instance
         */
        public <M extends MessageContextBuilder> M adapt(Class<M> builderClass);

}
```

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 Adaptable Message Context Builder

To to allow implementation specific builder methods, there is a need for an extensible Message Context Builder. The adapter pattern can be used for it, to provide more specialized, typed methods:

```
public class MyContextBuilder implements MessageContextBuilder {

        private String property;

        public MyContextBuilder configureMe(String withSomething) {
                this.property = withSomething;
                return this;
        }

        ...

        public Message buildMessage() {
                MyMessageContext mctx = new MyMessageContext(buildContext(), property);
                return new MyMessage(mctx);
        }

        @Override
```

```
	public <M extends MessageContextBuilder> boolean isAdaptable(Class<M> builderClass) {
		return MyContextBuilder.class.isAssignableFrom(builderClass);
	}

	@Override
	public <M extends MessageContextBuilder> M adapt(Class<M> builderClass) {
		if (isAdaptable(builderClass)) {
			return builderClass.cast(this);
		}
		return null;
	}

}

boolean adaptable = mcb.isAdaptable(MyContextBuilder.class);
if (adaptable) {
	MyContextBuilder amcb = mcb.adapt(MyContextBuilder.class);
	message = amcb.configureMe("atomatically")
				.content(ByteBuffer.wrap("Foo".getBytes()))
				.channel("foo-topic")
				.buildMessage();
	boolean isMyMessage = message instanceof MyMessage;//true
}
```

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 a protocol supports reply-to, it has to provide the services as well for publishing reply-to-request messages, as well as the infrastructure component to deal with reply-to-subscriptions.

5.7.1 Reply-To Publisher

Correlation id generated or not

5.7.2 Reply-To Subscription

5.8 Acknowledgment

TBD: Write more text here

5.9 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.

5.9.1 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));
```

5.9.2 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;
        }

}
```

5.9.3 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();
        }
});
```

5.10 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);

}
```

5.11 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