# MicroProfile Reactive Messaging Specification

James Roper, Clement Escoffier, Gordon Hutchison, Emily Jiang

3.1-SNAPSHOT, September 06, 2023: Draft

# Table of Contents


MicroProfile Reactive Messaging . . . 1
Copyright . . . 2
Eclipse Foundation Specification License . . . 2
Disclaimers . . . 2
Rationale . . . 4
Reactive Systems . . . 4
On JMS and Message Driven Beans . . . 5
Use cases . . . 5
Architecture . . . 7
Concepts . . . 7
Overall architecture . . . 7
Channel . . . 7
Message . . . 8
Message consumption with @Incoming . . . 9
Message production with @Outgoing . . . 9
Method consuming and producing . . . 10
Connectors . . . 10
Message stream operation . . . 11
Supported CDI scopes . . . 11
Supported method signatures . . . 12
Methods producing data . . . 13
Methods consuming data . . . 14
Methods processing data . . . 17
Examples of simple method streams . . . 23
Examples of methods using Reactive Streams or MicroProfile Reactive Streams Operators types . . . 24
Message acknowledgement and negative acknowledgement . . . 25
Positive acknowledgement . . . 25
Acknowledgement Examples . . . 29
Negative acknowledgement . . . 31
Connector . . . 33
Connector concepts . . . 33
Configuration . . . 34
Connector attribute . . . 36
Acknowledgement . . . 36
Metrics . . . 38
Injecting a Publisher or PublisherBuilder . . . 39
Publishing messages to a channel from imperative code . . . 40

Assembly and validation . . . 43
Release Notes for MicroProfile Reactive Messaging 3.0 . . . 44
Incompatible Changes . . . 44
API/SPI Changes . . . 44
Other Changes . . . 44
Release Notes for MicroProfile Reactive Messaging 2.0 . . . 45
Functional Changes . . . 45
Incompatible Changes . . . 45
Other Changes . . . 45

# MicroProfile Reactive Messaging

Specification: MicroProfile Reactive Messaging Specification

Version: 3.1-SNAPSHOT

Status: Draft

Release: September 06, 2023

# Copyright



# Eclipse Foundation Specification License



## Disclaimers

DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders. :sectnums:

# Rationale

State-of-the-art systems must be able to adapt themselves to emerging needs and requirements, such as market change and user expectations but also fluctuating load and inevitable failures. Leading-edge applications are capable of dynamic and adaptive capabilities aiming to provide *responsive* systems. While microservices aim to offer this agility, HTTP-based connecting tissue tends to fail to provide the required runtime adaptations, especially when facing failures.

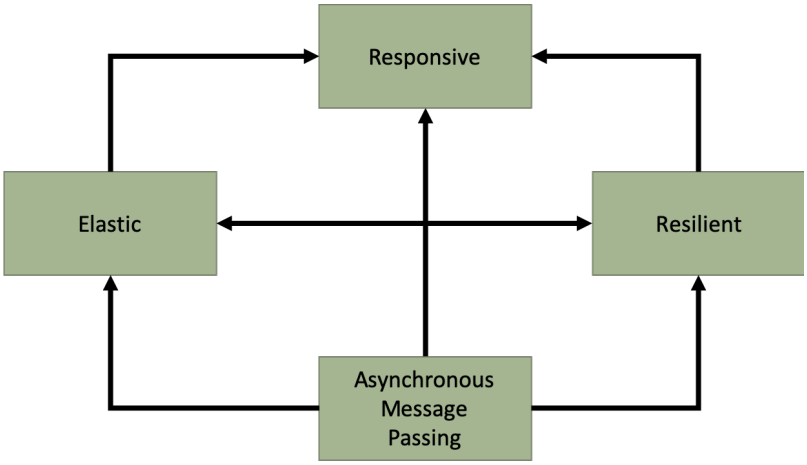
Asynchronous communication allows temporal decoupling of services in a microservice based architecture. This temporal decoupling is necessary if communication is to be enabled to occur regardless of when the parties involved in the communication are running, whether they are loaded or overloaded, and whether they are successfully processing messages or failing.

In contrast, synchronous communication couples services together, binding their uptime, failure, and handling of the load to each other. In a chain of synchronous interactions, the entire conversation can only be successful if all parties in the chain are responsive - if they are all running, processing messages successfully, and not overloaded. If just one party has a problem, all effectively exhibit the same problem. Therefore, systems of microservices relying on synchronous HTTP or relying on synchronous protocols tend to be *fragile*, and failures limit their availability. Indeed, in a microservice-based architecture, temporal coupling results in a fragile system, with resilience and scaling properties that are worse than a monolith, hence, it is essential for microservice based architectures to embrace asynchronous communication as much as possible.

The role of the MicroProfile Reactive Messaging specification is to deliver a way to build systems of microservices promoting both location transparency and temporal decoupling, enforcing asynchronous communication between the different parts of the system.

# Reactive Systems

Reactive Systems provide an architecture style to deliver *responsive* systems. By infusing asynchronous messaging passing at the core of the system, applications enforcing the reactive system's characteristics are inherently resilient and become more elastic by scaling up and down the number of message consumers.




Microservices as part of reactive systems interact using *messages*. The location and temporal decoupling, promoted by this interaction mechanism, enable numerous benefits such as:

- Better failure handling as the temporal decoupling enables message brokers to resend or reroute messages in the case of remote service failures.
- Improved elasticity as under fluctuating load the system can decide to scale up and down some of the microservices.
- The ability to introduce new features more easily as components are more loosely coupled by receiving and publishing messages.

The MicroProfile Reactive Messaging specification aims to deliver applications embracing the characteristics of reactive systems.

# On JMS and Message Driven Beans

Jakarta EE offers JMS and Message Driven Beans for handling asynchronous communication; however, there are some problems with these specifications:

- Both are designed for a technology landscape where messaging was typically on the edge of the system to hand control of a transaction from one system to another; consequently, these technologies can appear heavyweight when used between microservices.
- It is assumed in their design that consistency is handled using distributed transactions. However, many message brokers, popular in microservice deployments, such as Apache Kafka, Amazon Kinesis and Azure Event Hubs, do not support XA transactions, rather, message acknowledgment is handled using offsets with at least once delivery guarantees.
- They do not have support for asynchronous IO; it is assumed that message processing is done on a single thread, however, many modern specifications are moving to asynchronous IO.

Hence a lighter weight, reactive solution to messaging is desirable for MicroProfile to ensure microservices written using MicroProfile are able to meet the demands required by the architecture.

# Use cases

MicroProfile Reactive Messaging aims to provide a way to connect event-driven microservices. The key characteristics of the specification make it versatile and suitable for building different types of architecture and applications.

First, asynchronous interactions with different services and resources can be implemented using Reactive Messaging. Typically, asynchronous database drivers can be used in conjunction with Reactive Messaging to read and write into a data store in a non-blocking and asynchronous manner.

When building microservices, the CQRS and event-sourcing patterns provide an answer to the data sharing between microservices. Reactive Messaging can also be used as the foundation to CQRS and Event-Sourcing mechanism, as these patterns embrace message-passing as core communication pattern.

IOT applications, dealing with events from various devices, and data streaming applications can also be implemented using Reactive Messaging. The application receives events or messages,

process them, transform them, and may forward them to another microservices. It allows for more fluid architecture for building data-centric applications.

# Architecture

The Reactive Messaging specification defines a development model for declaring CDI *beans* producing, consuming and processing messages. The communication between these components uses Reactive Streams.

This specification relies on Eclipse MicroProfile Reactive Streams Operators and CDI.

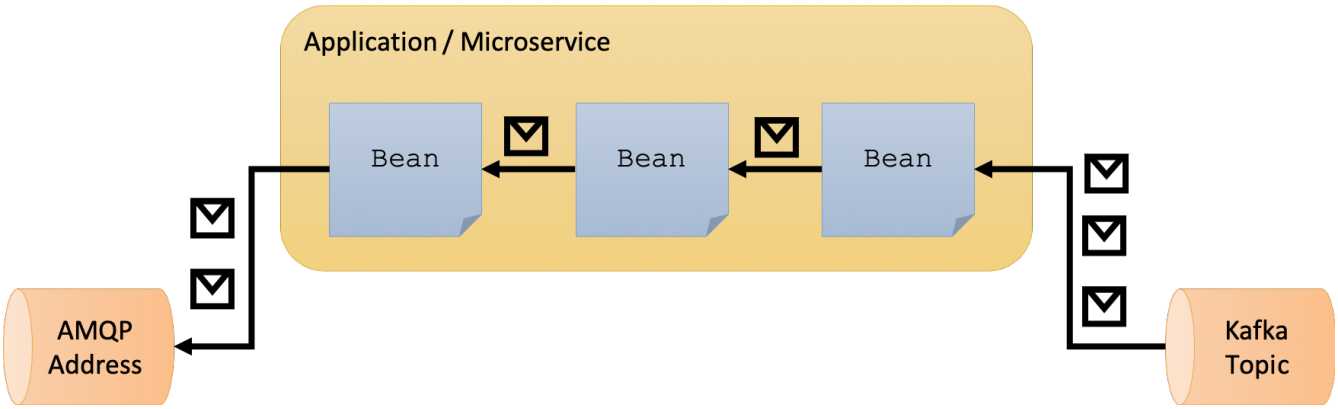
# Concepts

This section describes the different concepts introduced by the Reactive Messaging specification

## Overall architecture

An application using Reactive Messaging is composed of CDI beans consuming, producing and processing *messages*.

These messages can be wholly *internal* to the application or can be sent and received via different message brokers.




Application's beans contain methods annotated with `@Incoming` and `@Outgoing` annotations. A method with an `@Incoming` annotation consumes messages from a *channel*. A method with an `@Outgoing` annotation publishes messages to a *channel*. A method with both an `@Incoming` and an `@Outgoing` annotation is a message processor, it consumes messages from a *channel*, does some transformation to them, and publishes messages to another *channel*.

## Channel

A *channel* is a name indicating which source or destination of messages is used. Channels are opaque `Strings`.

There are two types of channel:

- Internal channels are local to the application. They allows implementing multi-step processing where several *beans* from the same application form a chain of processing.
- Channels can be *connected* to remote brokers or various message transport layers such as Apache Kafka or to an AMQP broker. These channels are managed by *connectors*.

If a component receives messages from a channel, we call this channel an *upstream* channel. If a component produces messages to a channel, we call this channel a *downstream* channel. Messages flow from upstream to downstream until it reaches a final consumer.

## Message

At the core of the Reactive Messaging specification is the concept of *message*. A *message* is an envelope wrapping a *payload*. A message is sent to a specific channel and, when received and processed successfully, *acknowledged*.

Reactive Messaging application components are addressable recipients which await the arrival of messages on a channel and react to them, otherwise lying dormant.

Messages are represented by the `org.eclipse.microprofile.reactive.messaging.Message` class. This interface is intentionally kept minimal. The aim is that *connectors* will provide their own implementations with additional metadata that is relevant to that connector. For instance, a `KafkaMessage` would provide access to the *topic* and *partition*.

The `org.eclipse.microprofile.reactive.messaging.Message#getPayload` method retrieves the wrapped payload. The `org.eclipse.microprofile.reactive.messaging.Message#ack` method acknowledges the message. The `org.eclipse.microprofile.reactive.messaging.Message#nack` method reports a negative acknowledgement. Note that the `ack` and `nack` methods are asynchronous as acknowledgement is generally an asynchronous process.

*Plain* messages are created using:

- `org.eclipse.microprofile.reactive.messaging.Message#of(T)` - wraps the given payload, no acknowledgement
- `org.eclipse.microprofile.reactive.messaging.Message#of(T, java.util.function.Supplier<java.util.concurrent.CompletionStage<java.lang.Void>>)` - wraps the given payload and provides the acknowledgment logic
- `org.eclipse.microprofile.reactive.messaging.Message#of(Supplier<CompletionStage<Void>> ack, Function<Throwable, CompletionStage<Void>> nack)` - wraps the given payload and provides the acknowledgment and negative acknowledgment logic

You can also create new `Message` instances by copying the content from an original message using the `withX` methods:

```
// Create a new message with a new payload, but using the ack and nack functions from
`message`
Message<T> newMessage = message.withPayload(newPayload);
// Create a new message with a new ack and nack logic, but using the original payload
Message<T> another = message
    .withNack(...)
    .withAck(...);
```

## Message consumption with @Incoming

The `org.eclipse.microprofile.reactive.messaging.Incoming` annotation is used on a method from a CDI bean to indicate that the method consumes messages from the specified channel:

```
@Incoming("my-channel")                                          ①
public CompletionStage<Void> consume(Message<String> message) {  ②
  return message.ack();
}
```

1. `my-channel` is the channel
2. the method is called for every message sent to the `my-channel` channel

Reactive Messaging supports various forms of method signatures. This is detailed in the next section.

Remember that Reactive Messaging interactions are assembled from Reactive Streams. A method annotated with `@Incoming` is a Reactive Streams *subscriber* and so consumes messages that fit with the message signature and its annotations. Note that the handling of the Reactive Streams protocol, such as subscriptions and back pressure, is managed by the Reactive Messaging implementation. The MicroProfile Reactive Streams specification used as a foundation for this version of Reactive Messaging is a single subscriber model where a stream `Publisher` is connected to a single `Subscriber` which controls back pressure. This implies that a Reactive Messaging channel should appear in a single `@Incoming` annotation. The annotation of more than one `@Incoming` method to be associated with the same channel is not supported and will cause an error during deployment.

Uncaught exceptions in the messaging methods are not propagated as errors to downstream, nor they are canceling the upstream.

From the user perspective, whether the incoming messages comes from co-located beans or a remote message broker is transparent. However, the user may decide to consume a specific subclass of `Message` (e.g. `KafkaMessage` in the following example) if the user is aware of this characteristic:

```
@Incoming("my-kafka-topic")
public CompletionStage<Void> consume(KafkaMessage<String> message) {    ①
  return message.ack();
}
```

1. Explicit consumption of a `KafkaMessage`

## Message production with @Outgoing

The `org.eclipse.microprofile.reactive.messaging.Outgoing` annotation is used to annotate a method from a CDI bean to indicate that the method publishes messages to a specified channel:

```
@Outgoing("my-channel")                                          ①
```

```
public Message<String> publish() {                                    ②
  return Message.of("hello");                                         ③
}
```

1. `my-channel` is the targeted channel
2. the method is called for every consumer *request*
3. you can create a *plain* `org.eclipse.microprofile.reactive.messaging.Message` using `org.eclipse.microprofile.reactive.messaging.Message#of(T)`

Reactive Messaging supports various forms of method signatures. This is detailed in the next section.

A method annotated with `@Outgoing` is a Reactive Streams *publisher* and so publishes messages according to the requests it receives. The downstream `@Incoming` method or outgoing connector with a matching channel name will be linked to this publisher. Only a single method can be annotated with `@Outgoing` for a particular channel name. Having the same channel name in more than one `@Outgoing` annotated method is not supported and will result in an error during deployment.

#### Method consuming and producing

A method can combine the `@Incoming` and `@Outgoing` annotation and will then act as a Reactive Streams processor:

```
@Incoming("my-incoming-channel")                                      ①
@Outgoing("my-outgoing-channel")                                      ②
public Message<String> process(Message<String> message) {
  return message.withPayload(message.getPayload().toUpperCase());  ③
}
```

1. The incoming channel
2. The outgoing channel
3. Use the `withPayload` method to preserve the `ack` and `nack` functions

Having the same channel appear in the `@Outgoing` and `@Incoming` annotations of a processor is not supported and will result in an error during deployment.

## Connectors

The application can receive and forward messages from various message brokers or transport layers. For instance, an application can be connected to a Kafka cluster, an AMQP broker or an MQTT server.

Reactive Messaging *Connectors* are extensions managing the communication with a specific transport technology. They are responsible for mapping a specific *channel* to remote sink or source of messages. This mapping is configured in the application configuration. Note that an implementation may provide various ways to configure the mapping, but support for MicroProfile

Config as a configuration source is mandatory.

Connector implementations are associated with a name corresponding to a messaging transport, such as Apache Kafka, Amazon Kinesis, RabbitMQ or Apache ActiveMQ. For instance, an hypothetical Kafka connector could be associated with the following name: `acme.kafka`. This name is indicated using a qualifier on the connector implementation.

The user can associate a channel with this connector using the associated name:

```
mp.messaging.incoming.my-kafka-topic.connector=acme.kafka  ①
```

1. the name associated with the connector.

The configuration format is detailed later in this document.

The Reactive Messaging implementation is responsible for finding the connector implementation associated with the given name in the user configuration. If the connector cannot be found, the deployment of the application must be failed.

The Reactive Messaging specification provides an SPI to implement connectors.

# Message stream operation

Message stream operation occurs according to the principles of reactive programming. The back pressure mechanism of reactive streams means that a publisher will not send data to a subscriber unless there are outstanding subscriber requests. This implies that data flow along the stream is enabled by the first request for data received by the publisher. For methods that are annotated with `@Incoming` and `@Outgoing` this data flow control is handled automatically by the underlying system which will call the `@Incoming` and `@Outgoing` methods as appropriate.

Although `@Incoming` and `@Outgoing` methods remain callable from Java code, calling them directly will not affect the reactive streams they are associated with. For example, calling an `@Outgoing` annotated method from user code will not post a message on a message queue and calling an `@Incoming` method cannot be used to read a message. Enabling this would bypass the automatic back pressure mechanism that is one of the benefits of the specification. The `@Incoming` and `@Outgoing` method annotations are used to declaratively define the stream which is then run by the implementation of MicroProfile Reactive Messaging without the user's code needing to handle concerns such as subscriptions or flow control within the stream.

# Supported CDI scopes

Implementations of the Reactive Messaging specification must support at least the following CDI scopes:

- `@ApplicationScoped` beans

- `@Dependent` beans

The following code gives an example of a bean annotated with `@ApplicationScoped`:

```
@ApplicationScoped
public class ApplicationScopeBeans {

  @Outgoing("source")
  public Publisher<Integer> source() {
    return ReactiveStreams.of(id).buildRs();
  }

  @Incoming("source")
  @Outgoing("output")
  public int process(int i) {
    return i + 1;
  }

  @Incoming("output")
  public void sink(int v) {
    System.out.println(v);
  }
}
```

Implementations can provide support for other scopes. However the behavior is not defined.

# Supported method signatures

The signature of message stream methods can have a number of different distinct types, offering differing levels of power and simplicity to application developers. Different shapes are supported depending on whether the method is a publisher, subscriber or processor, for example, a publishing stream supports returning MicroProfile Reactive Streams `PublisherBuilder`, but not `SubscriberBuilder`, the inverse is true for a subscribing stream.

This section lists the methods signatures that must be supported by the Reactive Messaging implementation. Implementations must validate that the stream shape matches the `@Outgoing` and `@Incoming` annotations, if they don't, a CDI definition exception should be raised to the CDI container during initialization.

It's important to remember that users must not call these methods directly. They are invoked by the Reactive Messaging implementation following the Reactive Streams protocol.

Also the method must be implemented in a non-blocking fashion. For blocking transformations, asynchronous variants can be used.

*assembly* time is when the Reactive Messaging implementation initializes itself and creates the different bean instances and connects them together. If the implementation cannot find a incoming of outgoing channel, a `DeploymentException`

must be thrown when the application starts.

In the following lists, `Message` can be an implementation of the `Message` interface.

## Methods producing data

| Signature | Behavior | Invocation |
|---|---|---|
| `@Outgoing("name")`<br>`Publisher<Message<O>> method()` | Returns a stream of `Message` associated with the channel `name`. | Method called once at assembly time. |
| `@Outgoing("channel")`<br>`Publisher<O> method()` | Returns a stream of *payload* of type `O` associated with the channel `channel`. Produced payloads are mapped to `Message<O>` by the Reactive Messaging implementation. | Method called once at assembly time. |
| `@Outgoing("channel")`<br>`PublisherBuilder<Message<O>> method()` | Returns a stream of `Message` associated with the channel `channel`. | Method called once at assembly time. |
| `@Outgoing("channel")`<br>`PublisherBuilder<O> method()` | Returns a stream of *payload* associated with the channel `channel`. Produced payloads are mapped to `Message<O>` by the Reactive Messaging implementation. | Method called once at subscription time. |
| `@Outgoing("channel")`<br>`Message<O> method()` | Produces an infinite stream of `Message` associated with the channel `channel`. | This method is called for each *request* made by the subscriber. |
| `@Outgoing("channel")`<br>`O method()` | Produces an infinite stream of *payload* associated with the channel `channel`. Produced payloads are mapped to `Message<O>` by the Reactive Messaging implementation. | This method is called for each *request* made by the subscriber. |

| Signature | Behavior | Invocation |
| --- | --- | --- |
| `@Outgoing("channel")`<br>`CompletionStage<Message<O>> method()` | Produces an infinite stream of `Message` associated with the channel `channel`. The result is a `CompletionStage`. The method should not be called by the reactive messaging implementation until the `CompletionStage` returned previously is completed. | This method is called for each *request* made by the subscriber. |
| `@Outgoing("channel")`<br>`CompletionStage<O> method()` | Produces an infinite stream of *payload* associated with the channel `channel`. Produced payloads are mapped to `Message<O>` by the Reactive Messaging implementation. The result is a `CompletionStage`. The method should not be called by the reactive messaging implementation until the `CompletionStage` returned previously is completed. | This method is called for each *request* made by the subscriber. |

## Methods consuming data

| Signature | Behavior | Invocation |
| --- | --- | --- |
| `@Incoming("channel")`<br>`Subscriber<Message<I>> method()` | Returns a `Subscriber` that receives the `Message` objects transiting on the channel `channel`. | The method is called only once to retrieve the `Subscriber` object at assembly time. This subscriber is connected to the matching channel. |

| Signature | Behavior | Invocation |
| --- | --- | --- |
| `@Incoming("channel")`<br>`Subscriber<I> method()` | Returns a `Subscriber` that receives the *payload* objects transiting on the channel `channel`. The payload is automatically extracted from the inflight messages using `Message.getPayload()`. | The method is called only once to retrieve the `Subscriber` object at assembly time. This subscriber is connected to the matching channel. |
| `@Incoming("channel")`<br>`SubscriberBuilder<Message<I>, Void>`<br>`method()` | Returns a `SubscriberBuilder` that receives the `Message` objects transiting on the channel `channel`. | The method is called only once at assembly time to retrieve a `SubscriberBuilder` that is used to build a `CompletionSubscriber` that is subscribed to the matching channel. |
| `@Incoming("channel")`<br>`SubscriberBuilder<I, Void> method()` | Returns a `SubscriberBuilder` that is used to build a `CompletionSubscriber<I>` that receives the *payload* of each `Message`. The payload is automatically extracted from the inflight messages using `Message.getPayload()`. | The method is called only once at assembly time to retrieve a `SubscriberBuilder` that is used to build a `CompletionSubscriber` that is subscribed to the matching channel. |
| `@Incoming("channel")`<br>`void method(I payload)` | Consumes the *payload*. | This method is called for every `Message<I>` instance transiting on the channel `channel`. The payload is automatically extracted from the inflight messages using `Message.getPayload()`. The user method is never called concurrently and so must return before being called with the next payload. |

| Signature | Behavior | Invocation |
| --- | --- | --- |
| `@Incoming("channel")`<br>`CompletionStage<Void> method(Message<I> msg)` | Consumes the `Message` | This method is called for every `Message<I>` instance transiting on the channel `channel`. The user method is never called concurrently. The reactive messaging implementation must wait until the completion of the previously returned `CompletionStage` before calling the method again with the next `Message`. Note that `@Incoming("channel") void method(Message<I> msg)` is not allowed as message acknowledgement is asynchronous. |
| `@Incoming("channel")`<br>`CompletionStage<?> method(I payload)` | Consumes the *payload* asynchronously | This method is called for every `Message<I>` instance transiting on the channel `channel`. The payload is automatically extracted from the inflight messages using `Message.getPayload()`. The user method is never called concurrently. The reactive messaging implementation must wait until the completion of the previously returned `CompletionStage` before calling the method again with the next *payload*. |

### Methods processing data

| Signature | Behavior | Invocation |
|---|---|---|
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Processor<Message<I>, Message<O>>`<br>`method()` | Returns a Reactive Streams processor consuming incoming `Message` instances and produces `Message` instances. | This method is called once; at assembly time. |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Processor<I, O> method();` | Returns a Reactive Streams processor consuming incoming *payload* instances and produces *payload* instances. | This method is called once; at assembly time. |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`ProcessorBuilder<Message<I>, Message<O>>`<br>`method();` | Returns a `ProcessorBuilder` consuming incoming `Message` instances and produces `Message` instances. | This method is called once; at assembly time. |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`ProcessorBuilder<I, O> method();` | Returns a Reactive Streams processor that consuming incoming *payload* instances and produces *payload* instances. | This method is called once; at assembly time. |

| Signature | Behavior | Invocation |
| --- | --- | --- |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Publisher<Message<O>> method(Message<I> msg)` | Returns a Reactive Streams `Publisher` for each incoming `Message`. The returned `Publisher` can be empty or emits multiple `Message` instances. If the returned `Publisher` emits several elements, these elements are *flattened* in the outgoing stream as a concatenation of elements. The *flattening* follows the same semantics as the `flatMap` operator from the MicroProfile Reactive Streams specification. | This method is called for every incoming message. Implementations must not call the method subsequently until the stream from the previously returned `Publisher` is completed. |

| Signature | Behavior | Invocation |
| --- | --- | --- |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Publisher<O> method(I payload)` | Returns a Reactive Streams `Publisher` for each incoming *payload*. The returned `Publisher` can be empty or emits multiple *payload* instances. If the returned `Publisher` emits several elements, these elements are *flattened* in the outgoing stream as a concatenation of elements. The *flattening* follows the same semantics as the `flatMap` operator from the MicroProfile Reactive Streams specification. The Reactive Messaging implementation must create new `Message` instances for each emitted *payload* as well as extracing the payload for each incoming `Message`. | This method is called for every incoming message. Implementations must not call the method subsequently until the stream from the previously returned `Publisher` is completed. |

| Signature | Behavior | Invocation |
| --- | --- | --- |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`PublisherBuilder<Message<O>> method (Message<I> msg)` | Returns a `PublisherBuilder` for each incoming `Message`. The stream resulting from the built `Publisher` can be empty or emits multiple `Message` instances. If the stream emitted from the built `Publisher` emits several elements, these elements are *flattened* in the outgoing stream as a concatenation of elements. The *flattening* follows the same semantics as the `flatMap` operator from the MicroProfile Reactive Streams specification. | This method is called for every incoming message. Implementations must not call the method subsequently until the stream built from the previously returned `PublisherBuilder` is completed. |

| Signature | Behavior | Invocation |
| --- | --- | --- |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`PublisherBuilder<O> method(I payload)` | Returns a `PublisherBuilder` for each incoming *payload*. The stream resulting from the built `Publisher` can be can be empty or emits multiple *payload* instances. If the stream emitted from the built `Publisher` emits several elements, these elements are *flattened* in the outgoing stream as a concatenation of elements. The *flattening* follows the same semantics as the `flatMap` operator from the MicroProfile Reactive Streams specification. The Reactive Messaging implementation must create new `Message` instances for each emitted *payload* as well as extracing the payload for each incoming `Message`. | This method is called for every incoming message. Implementations must not call the method subsequently until the stream built from the previously returned `PublisherBuilder` is completed. |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Message<O> method(Message<I> msg)` | Returns a `Message` for each incoming `Message`. | This method is called for every incoming message. Implementations must not call the method subsequently until the previous call must have returned. |

| Signature | Behavior | Invocation |
| --- | --- | --- |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`O method(I payload)` | Returns a *payload* for each incoming *payload*. *The Reactive Messaging implementation is responsible for unwrapping the _payload* from the incoming `Message` and creating a `Message` from the returned *payload*. | This method is called for every incoming message. Implementations must not call the method subsequently until the previous call must have returned. |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`CompletionStage<Message<O>> method (Message<I> msg)` | Produces a `Message` for each incoming `Message`. This method returns a `CompletionStage` that can redeem the `Message` instance asynchronously. The returned `CompletionStage` must not be completed with `null`. | This method is called for every incoming message. Never concurrently. The implementations must wait until the completion of the previously returned `CompletionStage` before calling the method again with the next `Message`. |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`CompletionStage<O> method(I payload)` | Produces a *payload* for each incoming *payload*. This method returns a `CompletionStage` that can redeem the *payload* instance asynchronously. The returned `CompletionStage` must not be completed with `null`. | This method is called for every incoming *payload*. Never concurrently. The implementations must wait until the completion of the previously returned `CompletionStage` before calling the method again with the next *payload*. |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Publisher<Message<O>> method(Publisher <Message<I>> pub)` | Applies a transformation to the incoming stream of `Message`. This method is used to manipulate streams and apply stream transformations. | This method is called once, at *assembly* time. |

| Signature | Behavior | Invocation |
| --- | --- | --- |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`PublisherBuilder<Message<O>> method`<br>`(PublisherBuilder<Message<I>> pub)` | Applies a transformation to the stream represented by the `PublisherBuilder` of `Message`. This method is used to manipulate streams and apply stream transformations. | This method is called once, at *assembly* time. |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Publisher<O> method(Publisher<I> pub)` | Applies a transformation to the incoming streams of *payloads*. This method is used to manipulate streams and apply stream transformations. | This method is called once, at *assembly* time. |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`PublisherBuilder<O> method`<br>`(PublisherBuilder<I> pub)` | Applies a transformation to the stream represented by the `PublisherBuilder` of *payloads*. This method is used to manipulate streams and apply stream transformations. | This method is called once, at *assembly* time. |

## Examples of simple method streams

The simplest shape that an application may use is a simple method. This is a method that accepts an incoming message, and possibly publishes an outgoing message:

```
@Incoming("in")
@Outgoing("out")
public Message<O> process(Message<I> msg) {
  return convert(msg);
}
```

In the above example, the stream is both a publishing and subscribing stream, with a 1:1 mapping of incoming to outgoing messages. Asynchronous processing may also be used, by returning a `CompletionStage`:

```
@Incoming("in")
@Outgoing("out")
```

```
public CompletionStage<Message<O>> process(Message<I> msg) {
  return asyncConvert(msg);
}
```

If the method is not `@Outgoing` annotated, then the returned value is ignored - however, note that for asynchronous methods, the returned `CompletionStage` is still important for determining when message processing has completed successfully, for the purposes of message acknowledgement. When there is no `@Outgoing` annotation, `void` may also be returned.

In addition to `Message`, implementations must allow:

- payloads (the content wrapped in a `Message`)
- implementation of the `Message` interface

### Examples of methods using Reactive Streams or MicroProfile Reactive Streams Operators types

For more power, developers may use Reactive Streams instances. Reactive Streams shaped methods accept no parameters, and return one of the following:

- `org.eclipse.microprofile.reactive.streams.operators.PublisherBuilder`
- `org.eclipse.microprofile.reactive.streams.operators.SubscriberBuilder`
- `org.eclipse.microprofile.reactive.streams.operators.ProcessorBuilder`
- `org.reactivestreams.Publisher`
- `org.reactivestreams.Subscriber`
- `org.reactivestreams.Processor`

Implementations may optionally support other types, such as JDK9 Flow publishers, subscribers and processors, or other representations of Reactive Streams. Application developers are recommended to use the MicroProfile Reactive Streams Operators *builders* in order to allow for the highest level of portability.

For example, here's a message processor:

```
@Incoming("in")
@Outgoing("out")
public PublisherBuilder<Message<I>, Message<O>> process() {
  return ReactiveStreams.<Message<I>>builder()
    .map(this::convert);
}
```

Implementations must support implementations of the `Message` interface.

# Message acknowledgement and negative acknowledgement

Acknowledgement is an important part of message processing. Acknowlegement indicates that a message has been processed correctly. A negative-acknoledgement indicates that a message was not processed correctly.

Messages are either acknowledged (positively or negatively) explicitly, or implicitly by the implementation.

All messages must be acknowledged either positively or negatively. What happens when a message is acknowledged depends on the source of the message, often on the connector.

# Positive acknowledgement

Acknowledgement for the `@Incoming` messages is controlled by the `org.eclipse.microprofile.reactive.messaging.Acknowledgment` annotation. The annotation allows configuring the acknowledgement strategy among:

- `MANUAL` - the user is responsible for the acknowledgement, by calling the `Message#ack()` method, so the Reactive Messaging implementation does not apply implicit acknowledgement
- `PRE_PROCESSING` - the Reactive Messaging implementation acknowledges the message before the annotated method or processing is executed
- `POST_PROCESSING` - the Reactive Messaging implementation acknowledges the message once:
  1. the method or processing completes if the method does not emit data
  2. when the emitted data is acknowledged

Each method signature type has different defaults and can implement different acknowledgement policies. If the `Acknowledgment` annotation is not set, the default policy is applied.

Method only annotated with `@Outgoing` do not support acknowledgement as they don't receive an input `Message`.

When a method annotated with `@Incoming` defines its acknowledgement policy to be `PRE_PROCESSING` or `POST_PROCESSING`, the Reactive Messaging implementation is responsible for the acknowledgement of the message. When the `POST_PROCESSING` policy is used, the incoming message is acknowledged when the outgoing message is acknowledged. Thus, it creates a chain of acknowledgements, making sure that the messages produced by an `IncomingConnectorFactory` are only acknowledged when the dispatching of the messages has been completed successfully.

The `MANUAL` strategy indicates that the incoming message acknowledgement is managed by the user code. The `MANUAL` strategy is often used to acknowledge incoming messages when the produced messages are acknowledged. For example, in the next snippet, the received `KafkaMessage` is acknowledged when the produced message is acknowledged.

```
@Incoming("data")
```

```
@Outgoing("sink")
@Acknowledgment(Acknowledgment.Strategy.MANUAL)
public Message<Integer> process(KafkaMessage<String, Integer> input) {
  return Message.of(processThePayload(input.getPayload(), () -> input.ack()));
}
```

Specifying `NONE` as acknowledgment strategy allows for this method to complete processing and return without handling acknowledgment and for this to be considered valid behavior of the method. However, if messages were never acknowledged this would result in a build-up of unacknowledged messages in the system as no automatic acknowledgment will be done when `NONE` is specified. When 'NONE' is specified, each message object's `ack()` method should still be invoked once as part of the overall processing but this is considered valid behavior either before, during, or after this method's execution.

The following table indicates the defaults and supported acknowledgement for each supported signature:

| Signature | Default Acknowledgement Strategy | Supported Strategies |
|---|---|---|
| `@Incoming("channel")`<br>`Subscriber<Message<I>> method()` | Manual | None, Pre-Processing, Post-Processing (when the `onNext` method returns), Manual |
| `@Incoming("channel")`<br>`Subscriber<I> method()` | Post-Processing | None, Pre-Processing, Post-Processing (when the `onNext` method returns) |
| `@Incoming("channel")`<br>`SubscriberBuilder<Message<I>, Void>`<br>`method()` | Manual | None, Pre-Processing, Post-Processing (when the `onNext` method returns), Manual |
| `@Incoming("channel")`<br>`SubscriberBuilder<I, Void> method()` | Post-Processing | None, Pre-Processing, Post-Processing (when the `onNext` method returns) |
| `@Incoming("channel")`<br>`void method(I payload)` | Post-Processing | None, Pre-Processing, Post-Processing (when the method returns) |
| `@Incoming("channel")`<br>`CompletionStage<?> method(Message<I>`<br>`msg)` | Manual | None, Pre-Processing, Post-Processing (when the returned `CompletionStage` is completed), Manual |

| Signature | Default Acknowledgement Strategy | Supported Strategies |
|---|---|---|
| `@Incoming("channel")`<br>`CompletionStage<?> method(I payload)` | Post-Processing | None, Pre-Processing, Post-Processing (when the returned `CompletionStage` is completed) |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Processor<Message<I>, Message<O>>`<br>`method()` | Manual | None, Pre-Processing, Manual |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Processor<I, O> method();` | Pre-Processing | None, Pre-Processing Post-Processing can be optionally supported by implementations, however it requires a 1:1 mapping between the incoming element and the outgoing element. |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`ProcessorBuilder<Message<I>, Message<O>>`<br>`method();` | Manual | None, Pre-Processing, Manual |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`ProcessorBuilder<I, O> method();` | Pre-Processing | None, Pre-Processing Post-Processing can be optionally supported by implementations, however it requires a 1:1 mapping the incoming element and the outgoing element. |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Publisher<Message<O>> method(Message<I>`<br>`msg)` | Manual | None, Manual, Pre-Processing |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Publisher<O> method(I payload)` | Pre-Processing | None, Pre-Processing |

| Signature | Default Acknowledgement Strategy | Supported Strategies |
| --- | --- | --- |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`PublisherBuilder<Message<O>> method`<br>`(Message<I> msg)` | Manual | None, Manual, Pre-Processing |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`PublisherBuilder<O> method(I payload)` | Pre-Processing | None, Pre-Processing |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Message<O> method(Message<I> msg)` | Manual | None, Manual, Pre-Processing |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`O method(I payload)` | Post-Processing | None, Pre-Processing, Post-Processing (when the message wrapping the produced payload is acknowledged) |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`CompletionStage<Message<O>> method`<br>`(Message<I> msg)` | Manual | None, Manual, Pre-Processing |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`CompletionStage<O> method(I payload)` | Post-Processing | None, Pre-Processing, Post-Processing (when the message wrapping the produced payload is acknowledged) |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Publisher<Message<O>> method(Publisher`<br>`<Message<I>> pub)` | Manual | None, Manual, Pre-Processing |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`PublisherBuilder<Message<O>> method`<br>`(PublisherBuilder<Message<I>> pub)` | Manual | None, Manual, Pre-Processing |

| Signature | Default Acknowledgement Strategy | Supported Strategies |
|---|---|---|
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`Publisher<O> method(Publisher<I> pub)` | Pre-Processing | None, Pre-Processing |
| `@Incoming("in")`<br>`@Outgoing("out")`<br>`PublisherBuilder<O> method`<br>`(PublisherBuilder<I> pub)` | Pre-Processing | None, Pre-Processing |

Invalid acknowledgement policies must be detected and a `DeploymentException` raised when the application is deployed.

## Acknowledgement Examples

Transiting data may be wrapped in a `Message`, which can be used to supply metadata, and also allows messages to be acknowledged. The contract for acknowledging messages is anything that accepts a `Message` is required to acknowledge it. So, if the application receives an incoming message wrapped in `Message`, it is responsible for invoking `Message.ack()`, and if the application publish an outgoing message wrapped in `Message`, then the spec implementation is responsible for invoking `Message.ack()`.

For example, the following application code is incorrect, since it accepts a message wrapped in `Message`, but does not acknowledge the messages:

```
@Incoming("in")
@Acknowledgment(Acknowledgment.Strategy.MANUAL)
public void process(Message<I> msg) {
  System.out.println("Got message " + msg.getPayload());
}
```

Here is a correct implementation:

```
@Incoming("in")
@Acknowledgment(Acknowledgment.Strategy.MANUAL)
public CompletionStage<Void> process(Message<I> msg) {
  System.out.println("Got message " + msg.getPayload());
  return msg.ack();
}
```

This implementation is also correct, since the application receives a payload wrapped in a message. It's the implementations responsibility to invoke `ack()` on the incoming message:

```
@Incoming("in")
public void process(I payload) {
  System.out.println("Got payload " + payload);
}
```

When dealing with payloads, the `POST_PROCESSING` strategy is the default strategy. In the following snippet, the incoming payload is transported into a message and unwrapped before calling the method. The produced result is wrapped into another `Message`. Following the `POST_PROCESSING` strategy, the incoming message must only be acknowledged when the output message is acknowledged. The implementation is responsible to chain the acknowledgements.

```
@Incoming("in")
@Outgoing("out")
public O process(I payload) {
  ...
}
```

The acknowledgment strategy can be changed. For instance, using the `PRE_PROCESSING` strategy, the incoming message is acknowledged before the method is called. It also means that the acknowledgment of the outgoing message would not acknowledge the incoming message anymore, as it's already acknowledged.

```
@Incoming("in")
@Outgoing("out")
@Acknowledgment(Acknowledgment.Strategy.PRE_PROCESSING)
public O process(I payload) {
  ...
}
```

`PRE_PROCESSING` can also be used with `Messages`:

```
@Incoming("in")
@Outgoing("out")
@Acknowledgment(Acknowledgment.Strategy.PRE_PROCESSING)
public Message<O> process(Message<I> msg) {
  return Message.of(convert(msg.getPayload()));
}
```

In this case, the message `msg` is acknowledged before the method being called. The outgoing message (returned by the method) does not have to chain the acknowledgment.

The `NONE` strategy indicates that the incoming message is not acknowledged and the acknowledgment of the outgoing message would not acknowledge the incoming message anymore. The `NONE` strategy may be used when the incoming messages are acknowledged in another location, or a different mechanism..

```
@Incoming("in")
@Outgoing("out")
@Acknowledgment(Acknowledgment.Strategy.NONE)
public O process(I payload) {
  ...
}
```

The `MANUAL` strategy indicates that the acknowledgment is managed by the user code. The following snippet is particularly useful for processing messages that are also being sent to a destination, as the implementation must not invoke `ack` until after the outgoing message has been sent to the destination:

```
@Incoming("in")
@Outgoing("out")
@Acknowledgment(Acknowledgment.Strategy.MANUAL) // Default strategy
public Message<O> process(Message<I> msg) {
  return Message.of(convert(msg.getPayload()), msg::ack);
}
```

The implementation is responsible for enforcing the acknowledgement strategy defined by the user when the `@Acknowledgement` policy is used. If the annotation is not used, the default policy must be enforced.

## Negative acknowledgement

Negative acknowledgement indicates a processing failure. As for the `POST_PROCESSING` positive acknowledgement strategy, a *nack* is propagated from a message to its source if the messages are chained. When the *nack* reaches the top of the chain, the source, generally a connector, can handle the failure and act accordingly. Typically, sending the invalid message to a dead-letter-queue or retrying the delivery are common strategies.

Negative acknowledgement happens automatically if a method annotated with `@Incoming` receives a single payload and uses `POST_PROCESSING` acknowledgement and throws an exception. The incoming message is *nacked* and the thrown exception is passed as reason. The following method would automatically trigger a `nack` if the incoming payload is `b`:

```
@Incoming("data")
@Outgoing("out")
public String process(String s) {  ①
    if (s.equalsIgnoreCase("b")) {
        throw new IllegalArgumentException("b"); ②
    }
    return s.toUpperCase();
}
```

1. `POST_PROCESSING` is the default strategy for this signature

2. throwing an exception *nacks* the incoming message

If a method handles payload asynchronously (returns a `CompletionStage`), the incoming message is nacked if:

- the method throws an exception
- the produced `CompletionStage` is completed exceptionally

```
@Incoming("data")
@Outgoing("out")
public CompletionStage<String> process(String s) {   ①
    if (s.equalsIgnoreCase("b")) {
        throw new IllegalArgumentException("b");     ②
    }

    if (s.equalsIgnoreCase("f")) {
        return null;                                 ③
    }

    if (s.equalsIgnoreCase("c")) {
        CompletableFuture<String> cf = new CompletableFuture<>();
        cf.completeExceptionally(new IllegalArgumentException("c"));  ④
        return cf;
    }
    return CompletableFuture.completedFuture(s.toUpperCase());
}
```

1. `POST_PROCESSING` is the default strategy for this signature
2. throwing an exception *nacks* the incoming message
3. returning `null` nacks the incoming message, as it's illegal to return `null` as `CompletionStage`
4. returning a `CompletionStage` failed exceptionally also nacks the incoming message

If the method receives a single `Message`, the user is responsible for calling the `nack` function:

```
@Incoming("data")
@Outgoing("out")
public Message<String> process(Message<String> m) {  ①
    String s = m.getPayload();
    if (s.equalsIgnoreCase("b")) {
        // we cannot fail, we must nack explicitly.
        m.nack(new IllegalArgumentException("b"));  ②
        return null;
    }
    return m.withPayload(s.toUpperCase());
}
```

1. `MANUAL` is the default strategy for this signature

2. throwing an exception would not *nack* the message, as it may have already been explicitly *acked* or *nacked*. Before the exception happens, the user must call the `nack` method.

Methods using the `NONE` acknowledgement strategy do not *nack* the incoming message. Methods using the `PRE_PROCESSING` acknowledgement strategy, the incoming message is not *nacked* either, as it was already acknowledged.

# Connector

Reactive Messaging connects matching `@Incoming` and `@Outgoing` stream elements running inside the same application. Additionally, it maps specific *channels* to *external* technologies such as Apache Kafka, MQTT, Web Sockets, AMQP, or JMS. This means that Reactive Messaging can receive messages from virtually any messaging technology and dispatch messages to any messaging technology. This bridging to an external messaging technology is done using a reactive messaging *connector*.

## Connector concepts

Each *connector* is responsible for a specific technology. A connector can:

- act as a *Publisher*, meaning it retrieves or receives messages from an external messaging technology and publishes them to a reactive stream. The messages will then be sent to a method annotated with `@Incoming`.
- act as a *Subscriber*, meaning it subscribes to a reactive stream and dispatches messages to an external messaging technology. The messages are received from a method annotated with `@Outgoing`.
- handle both directions.

It's essential that connectors implement the back-pressure protocol defined by the Reactive Streams specification.

A *connector* is implemented as a CDI Bean, generally *application* scoped implementing:

- the `org.eclipse.microprofile.reactive.messaging.connector.IncomingConnectorFactory` interface to receive messages from an external source;
- the `org.eclipse.microprofile.reactive.messaging.connector.OutgoingConnectorFactory` interface to dispatch messages to an external sink

Depending on the integrated technology, the *connector* can implement one of the interface or both.

The bean is a factory called by the Reactive Messaging implementation to create `PublisherBuilder` or `SubscriberBuilder` objects. These objects are then connected to methods annotated with `@Incoming` or `@Outgoing`.

Beans implementing the `IncomingConnectorFactory` or `OutgoingConnectorFactory` must use the `org.eclipse.microprofile.reactive.messaging.spi.Connector` qualifier. This qualifier defined the name associated with the connector.

The `@Connector` qualifier is used as follows:

```
package org.eclipse.reactive.sample.kafka;

import org.eclipse.microprofile.reactive.messaging.spi.*;

@ApplicationScoped
@Connector("acme.kafka")
public class KafkaConnector implements IncomingConnectorFactory,
OutgoingConnectorFactory {
    // ...
}
```

Once defined, the user can, in the configuration, refer to this connector using the given name (`acme.kafka` in this example). When the Reactive Messaging implementation processes the configuration, it determines the connector to be used based on the `connector` attribute.

## Configuration

Reactive Messaging connectors are configured using MicroProfile Config. The implementation processes the global configuration and determines:

- which channels are defined
- which connectors are used (using the `connector`) attribute
- the configuration for each channel

The builder methods defined in the `IncomingConnectorFactory` and `OutgoingConnectorFactory` receive a `org.eclipse.microprofile.config.Config` as parameter. The `Config` object contains key-value pairs to configure the connector. The configuration is specific to the connector. For example, a Kafka connector expects a *bootstrap.servers* entry as well as a *topic* entry.

The Reactive Messaging implementation reads the global application configuration and must support the following format:

- `mp.messaging.incoming.[channel-name].[attribute]=[value]`
- `mp.messaging.outgoing.[channel-name].[attribute]=[value]`
- `mp.messaging.connector.[connector-name].[attribute]=[value]`

For each extracted `channel-name`:

1. The `connector` attribute of the channel is read, and the connector implementation identified. If no loadable connector implementation matches, the deployment must be failed with a `DeploymentException`;
2. Relevant attributes are those matching either the `channel-name` or the resolved `connector-name`.
3. Relevant attributes are processed to generate a `Config` object containing only `attribute=value` entries. If is valid to have an attribute specified at a connector level and also for a specific channel. If an attribute appears for both a channel and its relevant connector, the channel

specific value will be used. In the example below, the `acme.kafka` default value for `bootstrap.servers` is overridden for `my-channel` to be `9096`.

The following snippet gives an example for a hypothetical Kafka connector:

```
mp.messaging.incoming.my-channel.connector=acme.kafka
mp.messaging.incoming.my-channel.bootstrap.servers=localhost:9096
mp.messaging.incoming.my-channel.topic=my-topic
mp.messaging.connector.acme.kafka.bootstrap.servers=localhost:9092
```

For properties that have a `mp.messaging.incoming.` or `mp.messaging.outgoing` prefix, this prefix is stripped off the property name and the remainder of the property name up to the first occurrence of `.` is treated as the channel name. Channel names may not include the `.` character.

For properties that have a `mp.messaging.connector.` prefix, this prefix is stripped off the property name and the longest remaining prefix that matches any configured `connector` is treated as a connector name. The remainder of the property name, minus the expected initial `.` separator, is taken as the name of an attribute for this connector. For example `bootstrap.servers` appears as a default attribute for all channels that use the `acme.kafka` connector.

The Reactive Messaging implementation:

1. Reads the configuration
2. Identifies that a `my-channel` source needs to be managed
3. Searches for the `connector` attribute and finds `acme.kafka`
4. Looks for a bean implementing the `IncomingConnectorFactory` interface qualified with `@Connector("acme.kafka")`. If the configuration had contained a `mp.messaging.outgoing.my-channel…` entry, a bean implementing the `OutgoingConnectorFactory` interface would have been searched for.
5. Creates a new `Config` object with just the relevant `key=value` pairs:

   ```
   bootstrap.servers=localhost:9096
   topic=my-topic
   ```

6. Calls the `PublisherBuilder<? extends Message> getPublisherBuilder(Config config)` method with the created `Config` object. If the configuration is invalid, the connector can throw:
   - a `NoSuchElementException` if a mandatory attribute is missing in the configuration
   - an `IllegalArgumentException` if the initialization of the connector fails for any other reasons.

   The Reactive Messaging implementation catches these exceptions and wraps them into a `DeploymentException`, failing the deployment of the application.
7. The built `PublisherBuilder` is connected to a method using the `@Incoming("my-stream")` annotation. The implementation of the connector must map every received message to an `org.eclipse.microprofile.reactive.messaging.Message`. Optionally, it can provide its own

implementation of `org.eclipse.microprofile.reactive.messaging.Message` providing additional metadata.

The configuration passed to the `IncomingConnectorFactory` and `OutgoingConnectorFactory` contains at least the:

- `channel-name` attribute indicating the name of the channel being configured,
- `connector` attribute must match the name given to the `@Connector` qualifier.

## Connector attribute

To help tools (IDEs, documentation generator) to extract the configuration of each connector, the specification provides the `org.eclipse.microprofile.reactive.messaging.spi.ConnectorAttribute` annotation for implementations to create a good ecosystem with the tools. However, the support for this SPI is not mandatory.

When used, each attribute supported by the connector should be documented using this annotation. The following snippet provides an example of a connector using the annotation:

```
@ConnectorAttribute(name = "bootstrap.servers", alias = "kafka.bootstrap.servers",
type = "string",
   defaultValue = "localhost:9092", direction = Direction.INCOMING_AND_OUTGOING,
    description = "...")
@ConnectorAttribute(name = "topic", type = "string", direction = Direction
.INCOMING_AND_OUTGOING,
    description = "...")
@ConnectorAttribute(name = "value-deserialization-failure-handler", type = "string",
direction = Direction.INCOMING,
    description = "...")
@ConnectorAttribute(name = "merge", direction = OUTGOING, type = "boolean",
defaultValue = "false",
    description = "...")
@Connector("my-connector")
public class MyConnector implements  IncomingConnectorFactory,
OutgoingConnectorFactory {
  // ...
}
```

As a result, tools can extract the supported attributes and improve the user experience.

## Acknowledgement

The connector is responsible for the acknowledgment (positive or negative) of the incoming and outgoing messages:

- An incoming connector must only acknowledge the received message when the produced `org.eclipse.microprofile.reactive.messaging.Message` is acknowledged.
- An incoming connector must handle received negative acknowledgment.

- An outgoing connector must acknowledge the incoming `org.eclipse.microprofile.reactive.messaging.Message` once it has successfully dispatched the message.
- An outgoing connector must acknowledge negatively the incoming `org.eclipse.microprofile.reactive.messaging.Message` if it cannot be dispatched.

# Metrics

When MicroProfile Reactive Messaging is used in an environment where MicroProfile Metrics is enabled, the Reactive Messaging implementation automatically produces metrics.

The following metrics are produced for each channel declared by the application and are added to the `base` scope.

| Name | Type | Unit | Description |
|---|---|---|---|
| `mp.messaging.message.count{channel="<channelname>"}` | Counter | None | The number of messages sent on the named channel. |

# Injecting a Publisher or PublisherBuilder

You can receive messages from a channel by injecting either a `Publisher` or `PublisherBuilder` and using the `@Channel` qualifier to specify the channel name:

```
@ApplicationScoped
public class BeanInjectedWithAPublisherOfPayloads {

    private final Publisher<String> constructor;
    @Inject
    @Channel("hello")
    private Publisher<String> field;

    @Inject
    public BeanInjectedWithAPublisherOfPayloads(@Channel("bonjour") Publisher<String>
constructor) {
        this.constructor = constructor;
    }

    public List<String> consume() {
        return Flowable
                .concat(
                        Flowable.fromPublisher(constructor),
                        Flowable.fromPublisher(field))
                .toList()
                .blockingGet();
    }

}
```

You can then inject `BeanInjectedWithAPublisherOfPayloads` to JAX-RS resources.

The value *hello* in the above example indicates the name of the channel.

For a payload type `X`, the following types can be injected:

- `Publisher<X>`
- `PublisherBuilder<X>`
- `Publisher<Message<X>>`
- `PublisherBuilder<Message<X>>`

# Publishing messages to a channel from imperative code

Traditionally, the reactive world and imperative world are separated and operate in parallel. Reactive Messaging deals with streams of data in the reactive world, while the imperative world is pretty much point to point and synchronous communication. However, imperative programme sometimes needs to connect to reactive streams so that responses can be emitted to a destination service. Bridging the two worlds is very valuable thing to do, so that one microservice can use technologies from both environments. For an instance, a JAX-RS resource might want to publish messages to a Reactive Messaging channel. This section is about enabling imperative code to publish messages to a Reactive Messaging channel, so that it can be consumed by a consumer.

You can inject an `Emitter` and use it to send either payloads (`X`) or messages (`Message<X>`) to a channel as demonstrated below.

```
@Inject @Channel("myChannel")
private Emitter<String> emitter;

public void publishMessage() {
  emitter.send("a");
  emitter.send("b");
  emitter.complete();
}
```

When sending payload, the `send` method returns a `CompletionStage`. This `CompletionStage` is completed when the emitted message is acknowledged. If the processing of the emitter message fails, the returned `CompletionStage` is completed exceptionally.

```
@Inject
@Channel("foo")
private Emitter<String> emitter;

public void run() {
    emitter.send(Message.of("a"));
    emitter.send(Message.of("b"));
    emitter.send(Message.of("c"));
}
```

When sending a `Message`, the `send` message does not return `CompletionStage`. However, you can create a `Message` configured with custom `ack` and `nack` functions.

When injecting an `Emitter` (e.g. `@Inject Emitter<T>`), you must specify the target channel name using the `@Channel` qualifier. You can then configure how the back pressure is handled via `@OnOverflow` annotation, for the situation where emitting messages/payloads faster than the consumption of the messages.

```
@Inject @Channel("myChannel")
@OnOverflow(value=OnOverflow.Strategy.BUFFER, bufferSize=300)
private Emitter<String> emitter;

public void publishMessage() {
  emitter.send("a");
  emitter.send("b");
  emitter.complete();
}
```

In the above snippet, the buffer size is set to 300 elements. If `@OnOverflow` is absent, the buffer strategy `OnOverflow.Strategy.BUFFER` will be used.

If the `bufferSize` is not specified, the size will be the value of the config property `mp.messaging.emitter.default-buffer-size`. If the property does not exist, the default value will be 128 elements. If the buffer is full, an error will be propagated.

The *value* attribute on `OnOverflow` is shown below:

- `OnOverflow.Strategy.BUFFER` - use a buffer, whose size will be determined by the value of `bufferSize` if set. Otherwise, the size will be the value of the config property `mp.messaging.emitter.default-buffer-size` if it exists. Otherwise, 128 will be used. If the buffer is full, an exception will be thrown from the `send` method.
- `OnOverflow.Strategy.UNBOUNDED_BUFFER` - use an unbounded buffer. The application may run out of memory if values are continually added faster than they are consumed.
- `OnOverflow.Stragegy.THROW_EXCEPTION` - throws an exception from the `send` method if the downstream can't keep up.
- `OnOverflow.Strategy.DROP` - drops the most recent value if the downstream can't keep up. It means that new value emitted by the emitter are ignored.
- `OnOverflow.Strategy.FAIL` - propagates a failure in case the downstream can't keep up. No more value will be emitted.
- `OnOverflow.Strategy.LATEST` - keeps only the latest value, dropping any previous value if the downstream can't keep up.
- `OnOverflow.Strategy.NONE` - ignores the back pressure signals letting the downstream consumer to implement a strategy.

Below are some examples:

```
@Inject
@Channel("myChannel")
@OnOverflow(value = OnOverflow.Strategy.BUFFER) // Buffer strategy using the buffer
size specified by _mp.messaging.emitter.default-buffer-size_ if exists. Otherwise, 128
will be used.
Emitter<String> emitter;

@Inject
```

```
@Channel("myChannel") // Buffer strategy will be used. It behaves as if
@OnOverflow(value = OnOverflow.Strategy.BUFFER) is present.
Emitter<String> emitter;

@Inject
@Channel("myChannel")
@OnOverflow(value = OnOverflow.Strategy.DROP) // Drop the most recent values
Emitter<String> emitter;
```

Since the `@Channel("myChannel")` is used to produce messages, a consumer with the `@Incoming("myChannel")` should be specified to consume the messages transiting on the channel *myChannel*.

# Assembly and validation

When the application starts, the Reactive Messaging implementation:

- connect the methods annotated with `@Incoming`, `@Outgoing`, `Emitter`, `@Channel` and connectors
- verify the validity of the resulting graph

Implementations must throw a `DeploymentException` when the application starts for any of the following conditions:

- A method with `@Incoming` has no *upstream* channel
- A method with `@Outgoing` has no *downstream* channel
- A method with `@Incoming` has multiple *upstream* channels
- A method with `@Outgoing` has multiple *downstream* channels
- An `Emitter` has no *downstream* channel
- An `Emitter` has multiple *downstream* channels
- An injected `@Channel` has no *upstream* channel
- An injected `@Channel` has multiple *upstream* channels
- The application configures a missing connector
- An incoming connector has no *downstream* channels
- An incoming connector has multiple *downstream* channels
- An outgoing connector has no *upstream* channels
- An outgoing connector has multiple *upstream* channels

# Release Notes for MicroProfile Reactive Messaging 3.0

A full list of changes delivered in the 3.0 release can be found at MicroProfile Reactive Messaging 3.0 Milestone.

### Incompatible Changes

This release aligns with Jakarta EE 9.1 (137), so it won't work with earlier versions of Jakarta or Java EE.

### API/SPI Changes

- Acknowledgement doesn't work with default Message.of(T,ack, nack)(#143)

## Other Changes

- Wrong assertion in ThrowExceptionOverflowStrategyOverflowTest (#145)
- DefaultOverflowStrategyOverflowTest expects downstream error (#147)
- AsynchronousPayloadProcessorAckTest expects stream to continue after nack (#148)
- Make the TCK tests CDI 4 compatible (#151)

# Release Notes for MicroProfile Reactive Messaging 2.0

A full list of changes delivered in the 2.0 release can be found at MicroProfile Reactive Messaging 2.0 Milestone.

## Functional Changes

- Add support for unmanaged stream injection using `@Inject @Channel(…)` (#3)
- Add support for emitters allowing emitting messages from *imperative* code (#70)
- Add support for metrics (#31)
- Add support for negative acknowledgement (#98)
- Update default acknowledgement strategy when dealing with `Message` (#97)
- Move metrics to the base scope (#93)
- Assembly validation on application start (#119)
- Add `@ConnectorAttribute` to allow connector configuration discovery (#94)
- Add negative acknowledgement support (#98])

## Incompatible Changes

- Spec dependencies marked as "provided" (#88)

## Other Changes

- Update to Jakarta EE8 APIs for MP 4.0 (75)