Quarkus Reactive Messaging Streams

Quarkus relies on MicroProfile Reactive Messaging spec to implement reactive messaging streams.

This cheat sheet covers the integrations between Quarkus and Messaging systems like Apache Kafka, AMQP, and MQTT protocols.

CREATING THE PROJECT

```
mvn "io.quarkus:quarkus-maven-plugin:1.4.2.Final:create" \
  -DprojectGroupId="org.acme" \
  -DprojectArtifactId="greeting" \
  -DprojectVersion="1.0-SNAPSHOT" \
  -DclassName="org.acme.GreetingResource" \
  -Dextensions="reactive-messaging, mutiny" \
  -Dpath="/hello"
```

Tip You can generate the project in https://code.quarkus.io/

PRODUCING MESSAGES

There are two ways:

Declarative annotation-based approach: `@org.eclipse.microprofile.reactive.messaging.Outgoing`. This is perfect for the reactive code approach.

Programmatic-based approach: injecting `org.eclipse.microprofile.reactive.messaging.Emitter` interface. This is perfect for linking the imperative code to reactive code.

Declarative

```
@ApplicationScoped
public class PriceMessageProducer {

    @Outgoing("prices") // (1)
    public Multi<Double> generate() {
        return
Multi.createFrom().ticks().every(Duration.ofSeconds(
1))
                .map(x -> random.nextDouble());
    }

}
```

1. It sets `prices` as a channel.

By default, messages are only dispatched to a single consumer. By using `@io.smallrye.reactive.messaging.annotations.Broadcast` the message is dispatched to all consumers.

```
@Outgoing("out")
@Broadcast(2) // (1)
```

1. Sets the number of consumers. If not set then all consumers receive the message.

Programmatic

```
import
org.eclipse.microprofile.reactive.messaging.Channel;

@ApplicationScoped
public class PriceMessageProducer {

    @Channel("prices") // (1)
    Emitter<Double> emitter;

    public void send(double d) {
        emitter.send(d);
    }
}
```

1. It configures `Emitter` channel to `prices`.

You can use `org.eclipse.microprofile.reactive.messaging.OnOverflow` to configure back pressure on Emitter.

```
@Channel("prices")
@OnOverflow(value = OnOverflow.Strategy.BUFFER,
bufferSize = 256) // (1)
Emitter<Double> emitter;
```

1. Overflow strategy.

The possible strategies are: `BUFFER`, `UNBOUNDED_BUFFER`, `DROP`, `FAIL`, `LATEST` and `NONE`.

Messages

If you want to send more information apart from the payload, you can use `org.eclipse.microprofile.reactive.messaging.Message` interface instead of directly the body content.

```
@Channel("prices") Emitter<Message<Double>> emitter;

MyMetadata metadata = new MyMetadata();
emitter.send(Message.of(d, Metadata.of(metadata)));
```

The framework automatically acknowledges messages, but you can change that with `@org.eclipse.microprofile.reactive.messaging.Acknowledgment` annotation or/and with Message instance.

```
@Outgoing("out")
@Acknowledgment(Acknowledgment.Strategy.PRE_PROCESSING)
public String process(String input) {}
```

Possible values are:.

`POST_PROCESSING`
: It is executed once the produced message is acknowledged.

`PRE_PROCESSING`
: It is executed before the message is processed by the method.

`MANUAL`
: It is done by the user.

`NONE`
: No acknowledgement is performed.

```
@Outgoing("out")
public Message<Integer> processAndProduceNewMessage(Integer in) {
    return Message.of(in,
        () -> {
            return in.ack();
        });
}
```

CONSUMING MESSAGES

There are two ways:

Declarative annotation-based approach: `@org.eclipse.microprofile.reactive.messaging.Incoming`. This is perfect for the reactive code approach.

Programmatic-based approach: injecting `io.smallrye.mutiny.Multi` or `org.reactivestreams.Publisher` interface.

Declarative

```
@ApplicationScoped
public class PayloadProcessingBean {

    @Incoming("prices") // (1)
    public void process(String in) {
        System.out.println(in.toUpperCase());
    }
}
```

1. Consumes messages from `prices` channel.

By default, having multiple producers to the same channel is considered as an error, but you can use `@io.smallrye.reactive.messaging.annotations.Merge` annotation to support it.

```
@Incoming("in1")
@Outgoing("out")
public int inc (int i) {}

@Incoming("in2")
@Outgoing("out")
public int mult (int i) {}

@Incoming("out")
@Merge
public void getAll(int i) {}
```

The following strategies are supported in `Merge` annotation: `ONE` to pick the first source only, `CONCAT` to concat the sources and `MERGE` (the default) to merge the different sources.

Multiple `@Incoming` annotations can be repeated to listen from more than one channel.

```
@Incoming("channel-1")
@Incoming("channel-2")
public void process(String s) {}
```

Programmative

```
@Channel("my-channel")
Multi<String> streamOfPayloads;

streamOfPayloads.map(s -> s.toUpperCase());
```

CONNECTORS

You need to set the mapping between the `channel` and the topic in the remote boker. The configuration parameters format is: `mp.messaging.[incoming|outgoing].[channel-name].[attribute]=[value].`

`incoming` or `outgoing` is to define if the channel is used as consumer or as producer.

`channel-name` is the name of the channel you've given in the annotation.

`attributes` are specific to the connector used.

Apache Kafka

```
./mvnw quarkus:add-extension -Dextensions="reactive-
messaging-kafka"
```

```
mp.messaging.outgoing.my-channel-
out.connector=smallrye-kafka
mp.messaging.outgoing.my-channel-out.topic=prices
mp.messaging.outgoing.my-channel-
out.bootstrap.servers=localhost:9092
mp.messaging.outgoing.my-channel-
out.value.serializer=org.apache.kafka.common.seriali
zation.IntegerSerializer

mp.messaging.incoming.my-channel-
in.connector=smallrye-kafka
mp.messaging.incoming.my-channel-
in.value.deserializer=org.apache.kafka.common.serial
ization.IntegerDeserializer
...
```

A complete list of supported properties are provided in the Kafka site. For the producer and for the consumer.

SmallRye Reactive Messaging Kafka provides `io.smallrye.reactive.messaging.kafka.KafkaRecord` as implementation of the `org.eclipse.microprofile.reactive.messaging.Message`.

```
OutgoingKafkaRecord<Integer, String>
outgoingKafkaRecord = KafkaRecord.of(s.id,
JsonbBuilder.create().toJson(s));

metadata =
OutgoingKafkaRecordMetadataBuilder.builder().withTim
estamp(Instant.now()).build();
outgoingKafkaRecord.withMetadata(metadata);
```

AMQP

```
./mvnw quarkus:add-extension -Dextensions="reactive-
messaging-kafka"
```

```
amqp-host=amqp
amqp-port=5672
amqp-username=quarkus
amqp-password=quarkus

mp.messaging.outgoing.my-channel-
out.connector=smallrye-amqp
mp.messaging.outgoing.my-channel-out.address=prices
mp.messaging.outgoing.my-channel-out.durable=true

mp.messaging.incoming.my-channel-
in.connector=smallrye-amqp
...
```

SmallRye Reactive Messaging AMQP provides `io.smallrye.reactive.messaging.amqp.IncomingAmqpMetadata` and `io.smallrye.reactive.messaging.amqp.OutgoingAmqpMetadata` to deal with AMQP metadata.

```
Optional<IncomingAmqpMetadata> metadata =
incoming.getMetadata(IncomingAmqpMetadata.class);

OutgoingAmqpMetadata metadata =
OutgoingAmqpMetadata.builder()
            .withAddress("customized-address")
            .withDurable(true)
            .withSubject("my-subject")
            .build();
incoming.addMetadata(metadata);
```

A complete list of supported properties for the AMQP integration is provided at the Reactive Messaging site.

MQTT

```
./mvnw quarkus:add-extension -Dextensions="reactive-
messaging-kafka"
```

```
mp.messaging.outgoing.my-channel-out.type=smallrye-
mqtt
mp.messaging.outgoing.my-channel-out.topic=prices
mp.messaging.outgoing.my-channel-out.host=localhost
mp.messaging.outgoing.my-channel-out.port=1883
mp.messaging.outgoing.my-channel-out.auto-generated-
client-id=true

mp.messaging.incoming.my-channel-in.type=smallrye-
mqtt
```

A complete list of supported properties for the MQTT integration is provided at the Reactive Messaging site.

Author Alex Soto
Java Champion, Working at Red Hat