

# Quarkus - Using the event bus

Quarkus allows different beans to interact using asynchronous events, thus promoting loose-coupling. The messages are sent to *virtual addresses*. It offers 3 types of delivery mechanism:

- point-to-point - send the message, one consumer receives it. If several consumers listen to the address, a round robin is applied;
- publish/subscribe - publish a message, all the consumers listening to the address are receiving the message;
- request/reply - send the message and expect a response. The receiver can respond to the message in an asynchronous-fashion

All these delivery mechanism are non-blocking, and are providing one of the fundamental brick to build reactive applications.



The asynchronous message passing feature allows replying to messages which is not supported by Reactive Messaging. However, it is limited to single-event behavior (no stream) and to local messages.

## Installing

This mechanism uses the Vert.x EventBus, so you need to enable the `vertx` extension to use this feature. If you are creating a new project, set the `extensions` parameter are follows:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.CR1:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=vertx-quickstart \
  -Dextensions="vertx,resteasy-mutiny"
cd vertx-quickstart
```

If you have an already created project, the `vertx` extension can be added to an existing Quarkus project with the `add-extension` command:

```
./mvnw quarkus:add-extension -Dextensions="vertx,resteasy-mutiny"
```

Otherwise, you can manually add this to the dependencies section of your `pom.xml` file:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-vertx</artifactId>
</dependency>
```

# Consuming events

To consume events, use the `io.quarkus.vertx.ConsumeEvent` annotation:

```
package org.acme.vertx;

import io.quarkus.vertx.ConsumeEvent;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class GreetingService {

    @ConsumeEvent                               ①
    public String consume(String name) {        ②
        return name.toUpperCase();
    }
}
```

- ① If not set, the address is the fully qualified name of the bean, for instance, in this snippet it's `org.acme.vertx.GreetingService`.
- ② The method parameter is the message body. If the method returns *something* it's the message response.

By default, the code consuming the event must be *non-blocking*, as it's called on the Vert.x event loop. If your processing is blocking, use the `blocking` attribute:



```
@ConsumeEvent(value = "blocking-consumer", blocking =
true)
void consumeBlocking(String message) {
    // Something blocking
}
```

Asynchronous processing is also possible by returning either an `io.smallrye.mutiny.Uni` or a `java.util.concurrent.CompletionStage`:

```

package org.acme.vertx;

import io.quarkus.vertx.ConsumeEvent;

import javax.enterprise.context.ApplicationScoped;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;
import io.smallrye.mutiny.Uni;

@ApplicationScoped
public class GreetingService {

    @ConsumeEvent
    public CompletionStage<String> consume(String name) {
        // return a CompletionStage completed when the processing
is finished.
        // You can also fail the CompletionStage explicitly
    }

    @ConsumeEvent
    public Uni<String> process(String name) {
        // return an Uni completed when the processing is finished.
        // You can also fail the Uni explicitly
    }
}

```



#### *Mutiny*

The previous example uses Mutiny reactive types, if you're not familiar with them, we recommend reading the [Getting Started with Reactive guide](#).

## Configuring the address

The `@ConsumeEvent` annotation can be configured to set the address:

```

@ConsumeEvent("greeting")
public String consume(String name) {
    return name.toUpperCase();
}

```

① Receive the messages sent to the `greeting` address

## Replying

The `return` value of a method annotated with `@ConsumeEvent` is used as response to the incoming message. For instance, in the following snippet, the returned `String` is the response.

```
@ConsumeEvent("greeting")
public String consume(String name) {
    return name.toUpperCase();
}
```

You can also return a `Uni<T>` or a `CompletionStage<T>` to handle asynchronous reply:

```
@ConsumeEvent("greeting")
public Uni<String> consume2(String name) {
    return Uni.createFrom().item(() ->
        name.toUpperCase()).emitOn(executor);
}
```



You can inject an `executor` if you use the Context Propagation extension:

```
@Inject Executor executor;
```

## Implementing fire and forget interactions

You don't have to reply to received messages. Typically for a *fire and forget* interaction, the messages are consumed and the sender does not need to know about it. To implement this, your consumer method just returns `void`

```
@ConsumeEvent("greeting")
public void consume(String event) {
    // Do something with the event
}
```

## Dealing with messages

As said above, this mechanism is based on the Vert.x event bus. So, you can also use `Message` directly:

```
@ConsumeEvent("greeting")
public void consume(Message<String> msg) {
    System.out.println(msg.address());
    System.out.println(msg.body());
}
```

# Sending messages

Ok, we have seen how to receive messages, let's now switch to the *other side*: the sender. Sending and publishing messages use the Vert.x event bus:

```
package org.acme.vertx;

import io.smallrye.mutiny.Uni;
import io.vertx.mutiny.core.eventbus.EventBus;
import io.vertx.mutiny.core.eventbus.Message;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/async")
public class EventResource {

    @Inject
    EventBus bus;                                ①

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @PathParam("{name}")
    public Uni<String> greeting(@PathParam String name) {
        return bus.<String>request("greeting", name)    ②
            .onItem().transform(Message::body);
    }
}
```

① Inject the Event bus

② Send a message to the address **greeting**. Message payload is **name**

The **EventBus** object provides methods to:

1. **send** a message to a specific address - one single consumer receives the message.
2. **publish** a message to a specific address - all consumers receive the messages.
3. **send** a message and expect reply

```
// Case 1
bus.sendAndForget("greeting", name)
// Case 2
bus.publish("greeting", name)
// Case 3
Uni<String> response = bus.<String>request("address", "hello, how
are you?")
    .onItem().transform(Message::body);
```

## Putting things together - bridging HTTP and messages

Let's revisit a greeting HTTP endpoint and use asynchronous message passing to delegate the call to a separated bean. It uses the request/reply dispatching mechanism. Instead of implementing the business logic inside the JAX-RS endpoint, we are sending a message. This message is consumed by another bean and the response is sent using the *reply* mechanism.

First create a new project using:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.CR1:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=vertx-http-quickstart \
  -Dextensions="vertx"
cd vertx-http-quickstart
```

You can already start the application in *dev mode* using `./mvnw compile quarkus:dev`.

Then, creates a new JAX-RS resource with the following content:

```
package org.acme.vertx;

import io.smallrye.mutiny.Uni;
import io.vertx.mutiny.core.eventbus.EventBus;
import io.vertx.mutiny.core.eventbus.Message;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/async")
public class EventResource {

    @Inject
    EventBus bus;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @PathParam("{name}")
    public Uni<String> greeting(@PathParam String name) {
        return bus.<String>request("greeting", name)
            .onItem().transform(Message::body);
    }
}
```

① send the **name** to the **greeting** address and request a response

② when we get the response, extract the body and send it to the user

If you call this endpoint, you will wait and get a timeout. Indeed, no one is listening. So, we need a consumer listening on the **greeting** address. Create a **GreetingService** bean with the following content:

*src/main/java/org/acme/vertx/GreetingService.java*

```
package org.acme.vertx;

import io.quarkus.vertx.ConsumeEvent;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class GreetingService {

    @ConsumeEvent("greeting")
    public String greeting(String name) {
        return "Hello " + name;
    }

}
```

This bean receives the name, and returns the greeting message.

Now, open your browser to <http://localhost:8080/async/Quarkus>, and you should see:

```
Hello Quarkus
```

To better understand, let's detail how the HTTP request/response has been handled:

1. The request is received by the `hello` method
2. a message containing the `name` is sent to the event bus
3. Another bean receives this message and computes the response
4. This response is sent back using the reply mechanism
5. Once the reply is received by the sender, the content is written to the HTTP response

This application can be packaged using:

```
./mvnw clean package
```

You can also compile it as a native executable with:

```
./mvnw clean package -Pnative
```

## Using codecs

The [Vert.x Event Bus](#) uses codecs to *serialize* and *deserialize* objects. Quarkus provides a default codec

for local delivery. So you can exchange objects as follows:

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("/{name}")
public Uni<String> greeting(@PathParam String name) {
    return bus.<String>request("greeting", new MyName(name))
        .onItem().transform(Message::body);
}

@ConsumeEvent(value = "greeting")
Uni<String> greeting(MyName name) {
    return Uni.createFrom().item(() -> "Hello " + name.getName());
}
```

If you want to use a specific codec, you need to explicitly set it on both ends:

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("/{name}")
public Uni<String> greeting(@PathParam String name) {
    return bus.<String>request("greeting", name,
        new
        DeliveryOptions().setCodecName(MyNameCodec.class.getName())) ①
        .onItem().transform(Message::body);
}

@ConsumeEvent(value = "greeting", codec = MyNameCodec.class)
②
Uni<String> greeting(MyName name) {
    return Uni.createFrom().item(() -> "Hello "+name.getName());
}
```

1. Set the name of the codec to use to send the message
2. Set the codec to use to receive the message