

# Quarkus - Funqy Knative Events

Quarkus Funqy [Knative Events](#) builds off of the [Funqy HTTP](#) extension to allow you to route and process Knative Events within a Funqy function.

The guide walks through quickstart code to show you how you can deploy and invoke on Funqy functions with Knative Events.

## Prerequisites

To complete this guide, you need:

- 60+ minutes
- Read about [Funqy Basics](#). This is a short read!
- JDK 1.8+ installed with [JAVA\\_HOME](#) configured appropriately
- Apache Maven 3.6.3
- Have gone through the [Knative Tutorial](#), specifically [Brokers and Triggers](#)

## Setting up Knative

Setting up Knative locally in a Minikube environment is beyond the scope of this guide. It is advised to follow [this](#) Knative Tutorial put together by Red Hat. It walks through how to set up Knative on Minikube or Openshift in a local environment.



Specifically you should run the [Brokers and Triggers](#) tutorial as this guide requires that you can invoke on a Broker to trigger the quickstart code.

## Read about Cloud Events

The Cloud Event [specification](#) is a good read to give you an even greater understanding of Knative Events.

## The Quickstart

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an [archive](#).

The solution is located in the `funqy-knative-events-quickstart` directory.

## The Quickstart Flow

The quickstart works by manually sending an HTTP request containing a Cloud Event to the Knative

Broker using `curl`. The Knative Broker receives the request and triggers the startup of the Funqy container built by the quickstart. The event triggers the invocation of a chain of Funqy functions. The output of one function triggers the invocation of another Funqy function.

## Funqy and Cloud Events

When living within a Knative Events environment, Funqy functions are triggered by a specific Cloud Event type. You can have multiple Funqy functions within a single application/deployment, but they must be triggered by a specific Cloud Event Type. The exception to this rule is if there is only one Funqy function in the application. In that case, the event is pushed to that function irregardless of the Cloud Event type.

Currently, Funqy can only consume JSON-based data. It supports both Binary and Structured mode of execution, but the data component of the Cloud Event message must be JSON. This JSON must also be marshallable to and from the Java parameters and return types of your functions.

## The Code

Let's start looking at our quickstart code so that you can understand how Knative Events map to Funqy. Open up [SimpleFunctionChain.java](#)

The first function we'll look at is `defaultChain`.

```
import io.quarkus.funqy.Funq;

public class SimpleFunctionChain {
    @Funq
    public String defaultChain(String input) {
        log.info("*** defaultChain ***");
        return input + "::" + "defaultChain";
    }
}
```

As is, a Funqy function has a default Cloud Event mapping. By default, the Cloud Event type must match the function name for the function to trigger. If the function returns output, the response is converted into a Cloud Event and returned to the Broker to be routed to other triggers. The default Cloud Event type for this response is the function name + `.output`. The default Cloud Event source is the function name.

So, for the `defaultChain` function, the Cloud Event type that triggers the function is `defaultChain`. It generates a response that triggers a new Cloud Event whose type is `defaultChain.output` and the event source is `defaultChain`.

While the default mapping is simple, it might not always be feasible. You can change this default mapping through configuration. Let's look at the next function:

```
import io.quarkus.funqy.Funq;

public class SimpleFunctionChain {
    @Funq
    public String configChain(String input) {
        log.info("*** configChain ***");
        return input + "::" + "configChain";
    }
}
```

The `configChain` function has its Cloud Event mapping changed by configuration within `application.properties`.

```
quarkus.funqy.knative-
events.mapping.configChain.trigger=defaultChain.output
quarkus.funqy.knative-events.mapping.configChain.response-
type=annotated
quarkus.funqy.knative-events.mapping.configChain.response-
source=configChain
```

In this case, the configuration maps the incoming Cloud Event type `defaultChain.output` to the `configChain` function. The `configChain` function maps its response to the `annotated` Cloud Event type, and the Cloud Event source `configChain`.

- `quarkus.funqy.knative-events.mapping.{function name}.trigger` sets the Cloud Event type that triggers a specific function
- `quarkus.funqy.knative-events.mapping.{function name}.response-type` sets the Cloud Event type of the response
- `quarkus.funqy.knative-events.mapping.{function name}.resource-source` sets the Cloud Event source of the response

The Funqy Knative Events extension also has annotations to do this Cloud Event mapping to your functions. Take a look at the `annotatedChain` method

```
import io.quarkus.funqy.Funq;
import io.quarkus.funqy.knative.events.CloudEventMapping;

public class SimpleFunctionChain {
    @Funq
    @CloudEventMapping(trigger = "annotated", responseSource =
"annotated", responseType = "lastChainLink")
    public String annotatedChain(String input) {
        log.info("*** annotatedChain ***");
        return input + "::" + "annotatedChain";
    }
}
```

If you use the `@CloudEventMapping` annotation on your function you can map the Cloud Event type trigger and the Cloud Event response. In this example the `annotatedChain` function will be triggered by the `annotated` Cloud Event type and the response will be mapped to a `lastChainLink` type and `annotated` Cloud Event source.

So, if look at all the functions defined within `SimpleFunctionChain` you'll notice that one function triggers the next. The last function that is triggered is `lastChainLink`.

```
import io.quarkus.funqy.Context;
import io.quarkus.funqy.Funq;

public class SimpleFunctionChain {
    @Funq
    public void lastChainLink(String input, @Context CloudEvent
event) {
        log.info("*** lastChainLink ***");
        log.info(input + "::" + "lastChainLink");
    }
}
```

There are two things to notice about this function. One, it has no output. Your functions are not required to return output. Second, there is an additional `event` parameter to the function.

If you want to know additional information about the incoming Cloud Event, you can inject the `CloudEvent` interface using the Funqy `@Context` annotation. The `CloudEvent` interface exposes information about the triggering event.

```
public interface CloudEvent {
    String id();
    String specVersion();
    String source();
    String subject();
    OffsetDateTime time();
}
```

## Maven

If you look at the `pom`, you'll see that it is a typical Quarkus pom that pulls in one funqy dependency

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-funqy-knative-events</artifactId>
</dependency>
```

# Dev mode and Testing

Funqy Knative Events support dev mode and unit testing using RestAssured. You can invoke on Funqy Knative Events functions using the same invocation model as [Funqy HTTP](#) using normal HTTP requests, or Cloud Event Binary mode, or Structured Mode. All invocation modes are supported at the same time.

So, if you open up the unit test code in [FunqyTest.java](#) you'll see that its simply using RestAssured to make HTTP invocations to test the functions.

Funqy also works with Quarkus Dev mode!

## Build the Project

First build the Java artifacts:

```
mvn clean install
```

Next, a docker image is required by Knative, so you'll need to build that next:

```
docker build -f src/main/docker/Dockerfile.jvm -t  
yourAccountName/funqy-knative-events-quickstart .
```

Make sure to replace **yourAccountName** with your docker or quay account name when you run **docker build**. The Dockerfile is a standard Quarkus dockerfile. No special Knative magic.

Push your image to docker hub or quay

```
docker push yourAccountName/funqy-knative-events-quickstart
```

Again, make sure to replace **yourAccountName** with your docker or quay account name when you run **docker push**.

## Deploy to Kubernetes/Openshift

The first step is to define a Kubernetes/Openshift service to points to your the docker image you created and pushed during build. Take a look at [funqy-service.yaml](#)

```

apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: funky-knative-events-quickstart
spec:
  template:
    metadata:
      name: funky-knative-events-quickstart-v1
      annotations:
        autoscaling.knative.dev/target: "1"
    spec:
      containers:
        - image: docker.io/yourAccountName/funqy-knative-events-quickstart

```

This is a standard Kubernetes service definition yaml file.



Make sure you change the image url to point to the image you built and pushed earlier!

For our quickstart, one Kubernetes service will contain all functions. There's no reason you couldn't break up this quickstart into multiple different projects and deploy a service for each function. For simplicity, and to show that you don't have to have a deployment per function, the quickstart combines everything into one project, image, and service.

Deploy the service yaml.

```
kubectl apply -n knativetutorial -f src/main/k8s/funqy-service.yaml
```

The next step is to deploy Knative Event triggers for each of the event types. As noted in the code section, each Funqy function is mapped to a specific Cloud Event type. You must create Knative Event triggers that map a Cloud Event and route it to a specific Kubernetes service. We have 4 different triggers.

[defaultChain-trigger.yaml](#)

```

apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:
  name: defaultchain
spec:
  filter:
    attributes:
      type: defaultChain
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: funqy-knative-events-quickstart

```

The `spec:filter:attributes:type` maps a Cloud Event type to the Kubernetes service defined in `spec:subscriber:ref`. When a Cloud Event is pushed to the Broker, it will trigger the spin up of the service mapped to that event.

There's a trigger yaml file for each of our 4 Funqy functions. Deploy them all:

```

kubectl apply -n knativetutorial -f src/main/k8s/defaultChain-
trigger.yaml
kubectl apply -n knativetutorial -f src/main/k8s/configChain-
trigger.yaml
kubectl apply -n knativetutorial -f src/main/k8s/annoatedChain-
trigger.yaml
kubectl apply -n knativetutorial -f src/main/k8s/lastChainLink-
trigger.yaml

```

## Run the demo

You'll need two different terminal windows. One to do a curl request to the Broker, the other to watch the pod log files so you can see the messages flowing through the Funqy function event chain.

Make sure you have the `stern` tool installed. See the Knative Tutorial setup for information on that. Run `stern` to look for logs outputted by our Funqy deployment

```
stern funq user-container
```

Open a separate terminal. You'll first need to learn the URL of the broker. Execute this command to find it.

```
kubectl get broker default -o jsonpath='{.status.address.url}'
```

This will provide you a url like this (exactly like this if you followed the knative tutorial): <http://default-broker.knativetutorial.svc.cluster.local> Remember this URL.

Next thing we need to do is ssh into our Kubernetes cluster so that we can send a curl request to our broker.

```
kubectl -n knativetutorial exec -it curler -- /bin/bash
```

You will now be in a shell within the Kubernetes cluster. Within the shell, execute this curl command

```
curl -v "http://default-broker.knativetutorial.svc.cluster.local" \  
-X POST \  
-H "Ce-Id: 1234" \  
-H "Ce-Specversion: 1.0" \  
-H "Ce-Type: defaultChain" \  
-H "Ce-Source: curl" \  
-H "Content-Type: application/json" \  
-d '"Start"'
```

This posts a Knative Event to the broker, which will trigger the `defaultChain` function. As discussed earlier, the output of `defaultChain` triggers an event that is posted to `configChain` which triggers an event posted to `annotatedChain` then finally to the `lastChainLink` function. You can see this flow in your `stern` window. Something like this should be outputted.

```
funqy-knative-events-quickstart-v1-deployment-59bb88bcf4-9jwdx  
user-container 2020-05-12 13:44:02,256 INFO  
[org.acm.fun.SimpleFunctionChain] (executor-thread-1) ***  
defaultChain ***  
funqy-knative-events-quickstart-v1-deployment-59bb88bcf4-9jwdx  
user-container 2020-05-12 13:44:02,365 INFO  
[org.acm.fun.SimpleFunctionChain] (executor-thread-2) ***  
configChain ***  
funqy-knative-events-quickstart-v1-deployment-59bb88bcf4-9jwdx  
user-container 2020-05-12 13:44:02,394 INFO  
[org.acm.fun.SimpleFunctionChain] (executor-thread-1) ***  
annotatedChain ***  
funqy-knative-events-quickstart-v1-deployment-59bb88bcf4-9jwdx  
user-container 2020-05-12 13:44:02,466 INFO  
[org.acm.fun.SimpleFunctionChain] (executor-thread-2) ***  
lastChainLink ***  
funqy-knative-events-quickstart-v1-deployment-59bb88bcf4-9jwdx  
user-container 2020-05-12 13:44:02,467 INFO  
[org.acm.fun.SimpleFunctionChain] (executor-thread-2)  
Start::defaultChain::configChain::annotatedChain::lastChainLink
```