

# Iniciación a Apache Kafka y Quarkus



build passing

Ejemplo simple de una conexión de Quarkus con Kafka.

Contenido  
Visión General  
Paso 1: Generar un proyecto  
Paso 2: Crear el modelo  
Paso 3: Crear un productor  
Paso 4: Crear un consumidor  
Paso 5: Configurar Kafka y los canales  
Paso 6: Endpoint REST endpoint para enviar mensajes  
Paso 7: Instalar Kafka  
Paso 8: Correr la aplicación  
Resumen  
Referencias

## Visión General

El marco ideal para la entrega continua y una mayor capacidad de recuperación lo proporciona la arquitectura orientada a microservicios. Además de impulsar la eficiencia de los desarrolladores y mejorar la escalabilidad en tiempo real, fomentan una innovación más rápida para responder a las cambiantes condiciones del mercado.

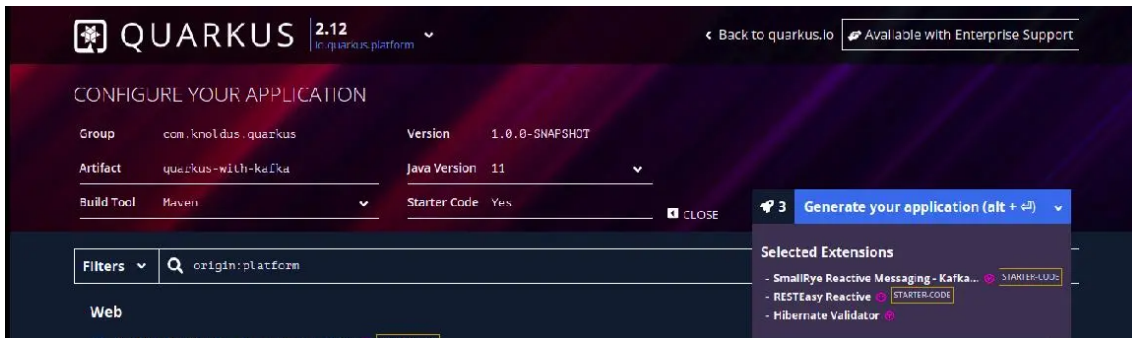
Para construir arquitecturas escalables, necesitamos una integración asíncrona y basada en eventos entre microservicios. Disponemos de múltiples opciones para la integración asíncrona. Kafka es la plataforma más popular que ofrece características como retención de mensajes, tolerancia a fallos y grupos de consumidores.


En este artículo, aprenderás cómo empezar con Apache Kafka y Quarkus framework. Aquí utilizaremos mensajería reactiva para construir microservicios basados en eventos, pero también puedes utilizar las API de Kafka para tu implementación.

## Paso 1: Generar un proyecto

Dirígete a <https://code.quarkus.io>, y rellena los detalles requeridos del proyecto como el id de grupo y el id de artefacto. Selecciona las siguientes dependencias de la lista

- SmallRye reactive messaging
- Resteasy reactive jackson
- Hibernate validator



Haga clic en Generar su aplicación, descargue el proyecto como un archivo zip, descomprímalo y cárguelo en su IDE favorito. La estructura del proyecto se verá así en el IDE.  Descripción de la imagen

## Paso 2: Crear el modelo

En Kafka, producimos y consumimos registros. Un registro contiene una clave y un valor. Crearemos un objeto Empleado, que será el objeto a producir y a consumir.

```
package org.acme.quarkus.model;

import lombok.*;

import javax.validation.constraints.NotBlank;

@Data
@AllArgsConstructor
@RequiredArgsConstructor
public class Employee {

    @NotBlank(message = "Primer apellido no puede ser nulo")
    private String firstName;

    private String lastName;

    @NotBlank(message = "Codigo empresa no puede ser nulo")
    private String empCode;

}
```

## Paso 3: Crear un productor

Añade una nueva clase EmployeeProducer según el siguiente código

```
package org.acme.quarkus.events.producer;
```

```

import io.smallrye.reactive.messaging.kafka.Record;
import org.acme.quarkus.model.Employee;
import org.eclipse.microprofile.reactive.messaging.Channel;
import org.eclipse.microprofile.reactive.messaging.Emitter;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

@ApplicationScoped
public class EmployeeProducer {

    @Inject
    @Channel("employee-out")
    Emitter<Record<String, Employee>> empEmitter;

    public void createEmployee(Employee employee) {
        empEmitter.send(Record.of(employee.getEmpCode(), employee));
    }
}

```

Vamos a decodificar esta clase. Hemos añadido la anotación `@ApplicationScoped` en la clase, lo que significa que una instancia se creará sólo una vez durante toda la aplicación. Dentro de la clase, hemos inyectado un Emisor que es responsable de enviar mensajes a un canal. El emisor se adjunta al canal `employee-out` y enviará los mensajes a Kafka. Tenemos un método `createEmployee` que enviará objetos `Record` de Empleado como par clave-valor.

## Paso 4: Crear un consumidor

Aquí, vamos a crear un consumidor `EmployeeConsumer` para el evento que publicamos anteriormente.

```

package org.acme.quarkus.events.consumer;

import org.acme.quarkus.model.Employee;
import io.smallrye.reactive.messaging.kafka.Record;
import org.eclipse.microprofile.reactive.messaging.Incoming;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class EmployeeConsumer {

    private final Logger logger = LoggerFactory.getLogger(EmployeeConsumer.class);

    @Incoming("employee-in")
    public void addEmployee(Record<String, Employee> employeeRecord) {
        logger.info("New employee {} {} joined with employee code {}",
            employeeRecord.value().getFirstName(),
            employeeRecord.value().getLastName(),

```

```

        employeeRecord.value().getEmpCode());
    }
}

```

Estamos utilizando la anotación `@Incoming` para pedir a Quarkus que llame al método `receive` por cada registro recibido del canal `employee-in`.

## Paso 5: Configurar Kafka y los canales

En la mensajería reactiva, enviamos y recibimos mensajes desde canales. Estos canales se asignan a la plataforma de mensajería subyacente mediante la configuración. En nuestra aplicación, hemos utilizado los canales `employee-in` y `employee-out` para la mensajería. Configuraremos estos canales en el archivo `application.properties`.

```

kafka.bootstrap.servers=localhost:9092

mp.messaging.incoming.employee-in.connector=smallrye-kafka
mp.messaging.incoming.employee-in.topic=employee
mp.messaging.incoming.employee-
in.key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
mp.messaging.incoming.employee-
in.value.deserializer=org.acme.quarkus.util.EmployeeDeserializer

mp.messaging.outgoing.employee-out.connector=smallrye-kafka
mp.messaging.outgoing.employee-out.topic=employee
mp.messaging.outgoing.employee-
out.key.serializer=org.apache.kafka.common.serialization.StringSerializer
mp.messaging.outgoing.employee-
out.value.serializer=org.acme.quarkus.util.EmployeeSerializer

```

Como tenemos un objeto para enviar como valor hemos creado un serializador y deserializador personalizado para enviar y recibir el mensaje.

## Paso 6: Endpoint REST endpoint para enviar mensajes

Para llamar a nuestro productor y enviar el objeto `Empleado` como un mensaje, vamos a crear un endpoint que llamará al método `createEmployee`.

```

package org.acme;

import org.acme.quarkus.events.producer.EmployeeProducer;
import org.acme.quarkus.model.Employee;

import javax.inject.Inject;
import javax.validation.Valid;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("/employee")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class EmployeeResource {

```

```

@Inject
EmployeeProducer employeeProducer;

@POST
public Response createEmployee(@Valid Employee employee) {
    employeeProducer.createEmployee(employee);
    return Response.accepted().build();
}
}

```

## Paso 7: Instalar Kafka

Cree un archivo docker-compose.yaml en la carpeta raíz y añada el siguiente contenido

```

version: '3'

services:

  zookeeper:
    image: strimzi/kafka:0.20.1-kafka-2.6.0
    command: [
      "sh", "-c",
      "bin/zookeeper-server-start.sh config/zookeeper.properties"
    ]
    ports:
      - "2181:2181"
    environment:
      LOG_DIR: /tmp/logs

  kafka:
    image: strimzi/kafka:0.20.1-kafka-2.6.0
    command: [
      "sh", "-c",
      "bin/kafka-server-start.sh config/server.properties --override
listeners=${KAFKA_LISTENERS} --override
advertised.listeners=${KAFKA_ADVERTISED_LISTENERS} --override
zookeeper.connect=${KAFKA_ZOOKEEPER_CONNECT}"
    ]
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      LOG_DIR: "/tmp/logs"
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181

```

## Paso 8: Correr la aplicación

En tu terminal, ejecuta el siguiente comando

```
docker-compose up -d
```

Una vez, el zookeeper y Kafka están en funcionamiento. Ejecute lo siguiente en el terminal

```
./mvnw quarkus:dev
```

La aplicación se iniciará en modo dev. En otro terminal, ejecute alguna petición HTTP

```
curl -X POST http://localhost:8080/employee
-H 'cache-control: no-cache'
-H 'content-type: application/json'
-H 'postman-token: 00fb55a5-7c90-af25-b1f4-9aa74a445405'
-d '{"firstName":"Vimal","lastName":"Kumar", "empCode":"1823"}'
```

Una vez recibido el mensaje podrá ver el siguiente mensaje

## Resumen

Has visto cómo podemos utilizar la mensajería reactiva con Quarkus y Kafka. Has implementado una aplicación sencilla que produce y consume mensajes.

## Referencias

- [Quarkus](#) - Marco Java nativo de Kubernetes adaptado a GraalVM y HotSpot.
- [Quarkus Reactive Messaging](#) - Utilizar SmallRye Reactive Messaging para interactuar con Apache Kafka.
- [Kafka](#) - Apache Kafka es un proyecto de intermediación de mensajes de código abierto.
- [Docker](#) - Proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software