Event Streaming for Microservices using Kafka

👩🏻 BARRACHIN Carlyne

Introduction

Event streaming is designed to handle continuous streams of data in real time. It is an effective method of ensuring that services in a microservices architecture communicate asynchronously, enabling them to react to system changes as they occur. Services can either publish or subscribe to events, enabling independent services to be developed, deployed and maintained separately, for greater scalability and flexibility.

Why Event Streaming?

In this tutorial, we will implement event streaming using **Apache Kafka**, a widely-used event streaming platform, to build a **Meeting Room Reservation System** with microservices that handle room availability. For example, when making an online meeting-room reservation, users should see immediate changes in availability.

Concepts of Producers and Consumers in Event Streaming

Before diving into implementation, it’s important to understand the basic building blocks of event streaming systems: **producers** and **consumers**.

Producers

A producer is responsible for **creating and publishing events** to an event broker (like Apache Kafka). In a microservice architecture, services that generate new information will act as producers.

Consumers

A consumer is responsible for **listening to and processing events**. Consumers subscribe to specific events and act accordingly when the events occur.

In this project

- The **Meeting Service** will be the **producer**, generating events such as room reservations or cancellations: ROOM_RESERVED & ROOM_CANCELED and publishes it in a Kafka topic (room-availability)
- The **Notification Service** will be the **consumer**, processing these events and notifying users. This service listens for ROOM_RESERVED & ROOM_CANCELED.

Here, we won't be sending notifications to users (another potential Service), but simply logging the fact that the Notification Service has received an event generated by the meeting service.

Step-by-Step Guide to Implementing Event Streaming with Kafka

Part 1: Installing and Configuring Kafka Using Docker

We will use **Docker** to run Kafka, as it simplifies the setup process and avoids the need for manual installation.

Step 1: Install Docker

First, ensure that Docker is installed and running on your system. Docker is needed to launch Kafka and Zookeeper, the latter of which coordinates Kafka brokers.

Step 2: Set Up a New Project

Open your preferred IDE (such as WebStorm or Visual Studio Code). Create a new project folder named EventStreamingTutorial.

Step 3: Create a Docker Compose File

In the project root, create a docker-compose.yml file. This file will allow us to set up and run Kafka and Zookeeper, along with additional services (such as a user interface for Kafka) in Docker containers.

Add the following code to the docker-compose.yml file:

```
version: '2'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - 22181:2181

  kafka:
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    ports:
      - 29092:29092
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
  kafka_ui:
    image: provectuslabs/kafka-ui:latest
    depends_on:
      - kafka
    ports:
      - 8080:8080
    environment:
      KAFKA_CLUSTERS_0_ZOOKEEPER: zookeeper:2181
      KAFKA_CLUSTERS_0_NAME: local
      KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: kafka:9092
```

This configuration will set up **Kafka**, **Zookeeper**, and a **Kafka UI** tool for monitoring Kafka events.

> Remarks:

> → **ZooKeeper** is used to manage metadata and coordinate Kafka services.
>
> → External applications can connect to Kafka via **localhost:29092.**
>
> → Access to the Kafka user interface via **localhost:8080**.

Step 4: Launch Kafka

Run the following command to start Kafka and Zookeeper:

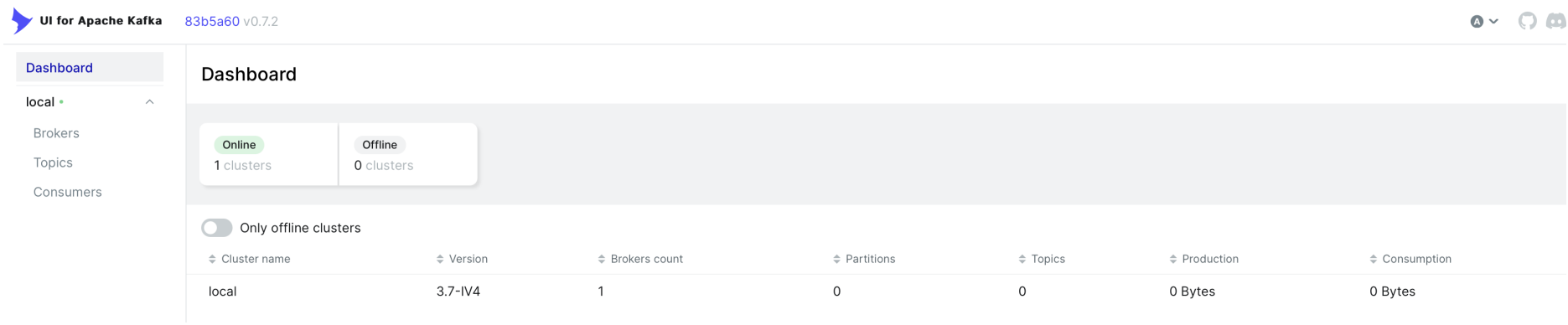
```
docker-compose up -d
```

Check if everything is working by :

- checking that everything is in order in the terminal

```
[+] Running 4/4
 ✔ Network eventstreamingtutorial_default         Created
 ✔ Container eventstreamingtutorial-zookeeper-1   Created
 ✔ Container eventstreamingtutorial-kafka-1       Created
 ✔ Container eventstreamingtutorial-kafka_ui-1    Created
```

- visiting the Kafka UI tool at http://localhost:8080.



You should have 1 cluster Online.

Part 2: Implementing the Microservices

We will now create two microservices: the **Meeting Service** (which will act as the event producer) and the **Notification Service** (which will act as the consumer).

Step 1: Create the Meeting Service (Producer)

1.1 Install Nest.js CLI and Create a New Service

In the EventStreamingTutorial folder, run the following commands to create the **Meeting Service.**

Install nestjs if you haven't already:

```
npm install -g @nestjs/cli
```

Create a nestjs project named meeting-service:

```
nest new meeting-service
cd meeting-service
```

1.2 Install Kafka Dependencies

```
npm install @nestjs/microservices kafkajs
```

1.3 Configure Kafka in *app.module.ts*

In this file, we set up the Kafka connection by configuring the **ClientsModule** to register a Kafka client. This module configuration enables our application to communicate with Kafka as both a producer and consumer of events.

We need to register the Kafka Client :

```
ClientsModule.register([
  {
    name: 'MEETING-SERVICE',
    transport: Transport.KAFKA,
    options: {
      client: {
        brokers: ['localhost:29092'],
      },
      consumer: {
        groupId: 'meeting-group',
      },
    },
  },
])
```

name: 'MEETING-SERVICE': The client is named **MEETING-SERVICE**.

transport: Transport.KAFKA: This sets Kafka as the communication protocol for the microservice client.

brokers: ['localhost:29092']: Specifies the Kafka broker's address. A **broker** is a server that manages and distributes messages between producers and consumers.

consumer: { groupId: 'meeting-group' }: Sets up a consumer group ID. This ID ensures that only one instance in the group processes each message, preventing duplicates when scaling horizontally.

You can find the complete code below :

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ClientsModule, Transport } from '@nestjs/microservices';

@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'MEETING-SERVICE',
        transport: Transport.KAFKA,
        options: {
          client: {
            clientId: 'meeting',
            brokers: ['localhost:29092'],
          },
          consumer: {
            groupId: 'meeting-group',
          },
        },
      },
    ]),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

1.4 Create the Meeting Controller (*app.controller.ts*)

Now we will define routes for reserving and canceling a room. When a room is reserved or canceled, an event will be emitted to Kafka.

First, we need to import the ClientKafka class:

`import { ClientKafka } from '@nestjs/microservices';`

This one is a Kafka client provided by NestJS. It is used to send messages to a Kafka broker as a producer. ClientKafka simplifies interaction with Kafka by providing ready-to-use methods for sending events or subscribing to specific topics.

Then, we need to inject the ClientKafka into the constructor:

`constructor(@Inject('MEETING-SERVICE') private readonly client: ClientKafka,) {}`

The name 'MEETING-SERVICE' refers to a specific Kafka configuration defined in *app.module.ts*.
This allows the controller to publish messages to Kafka via this client instance.

After that, we can start creating methods for publishing events. For example, if we want to create a simple method for reserving a room, we can write :

```
@Post('reserve')
async reserveRoom(@Body() body: { roomId: string }) {
  const message = `ROOM_RESERVED: Room ${body.roomId} reserved`;
  this.client.emit('room-availability', message);
  return { message: 'Room reserved successfully' };
}
```

When this method is triggered, it creates a message in the format ROOM_RESERVED, containing the ID of the reserved room (roomId).

The **emit** method provided by ClientKafka publishes the event to the Kafka topic '**room-availability**'. By sending the message to this topic, Kafka distributes it to any consumers subscribed to this topic.

We can use the same principle to **cancel** a room.

You can find the complete code below :

```
import {Body, Controller, Get, Inject, Post} from '@nestjs/common';
import { ClientKafka } from '@nestjs/microservices';
import {AppService} from "./app.service";

@Controller('meetings')
export class AppController {
  constructor(
      @Inject('MEETING-SERVICE') private readonly client: ClientKafka,
  ) {}

  @Post('reserve')
  async reserveRoom(@Body() body: { roomId: string }) {
    const message = `ROOM_RESERVED: Room ${body.roomId} reserved`;
    this.client.emit('room-availability', message);
    return { message: 'Room reserved successfully' };
  }

  @Post('cancel')
  async cancelReservation(@Body() body: { roomId: string }) {
    const message = `ROOM_CANCELED: Reservation for Room ${body.roomId} canceled`;
    this.client.emit('room-availability', message);
    return { message: 'Reservation canceled successfully' };
  }
}
```

> Remarks:

> - **POST /meetings/reserve**: This route handles room reservation requests. When called, it publishes an event to Kafka, notifying other services that the specified room has been reserved.
> - **POST /meetings/cancel**: This route handles room cancellation requests. It publishes an event to Kafka to inform other services that the room reservation has been canceled and is now available again.

1.5 Remove the `getHello()` method from the project

You might get errors if you don't remove this method throughout the project.

In the app.controller.ts file delete:

```
@Get()
getHello(): string {
  return this.appService.getHello();
}
```

In the app.service.ts file delete :

```
getHello(): string {
  return 'Hello World!';
}
```

In the app.controller.spec.ts file delete:

```
describe('root', () => {  it('should return "Hello World!"', () => {    expect(appController.
getHello()).toBe('Hello World!');  });});
```

Step 2: Create the Notification Service (Consumer)

2.1 Create the Notification Service and Install Kafka Dependencies

Similar to the previous steps :

```
nest new notification-service
cd meeting-service
```

```
npm install @nestjs/microservices kafkajs
```

2.2 Modify main.ts for Kafka Consumer

```
import { NestFactory } from '@nestjs/core';
import { Transport, MicroserviceOptions } from '@nestjs/microservices';

import { AppModule } from './app.module';

async function bootstrap() {
    const app = await NestFactory.createMicroservice<MicroserviceOptions>(AppModule, {
        transport: Transport.KAFKA,
        options: {
            client: {
                brokers: ['localhost:29092'],
            },
            consumer: {
                groupId: 'notification-consumer',
            },
```

```
        }
    });

    await app.listen();
}
bootstrap().then(r => console.log('Notification Service is listening'));
```

The **NestFactory.createMicroservice** method initializes the application as a Kafka microservice, allowing it to communicate directly with Kafka brokers. The brokers array specifies the address of the Kafka brokers the service connects to. Here, it's localhost:29092, meaning the Kafka broker is running locally on port 29092.

2.3 Configure Kafka in app.module.ts

In this module, we will set up the necessary configuration for the notification service, which will consume messages from Kafka.

We register the **NOTIFICATION-SERVICE** client in the imports array of the module:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ClientsModule, Transport } from '@nestjs/microservices';

@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'NOTIFICATION-SERVICE',
        transport: Transport.KAFKA,
        options: {
          client: {
            clientId: 'notification',
            brokers: ['localhost:29092'],
            connectionTimeout:3000,
            authenticationTimeout: 1000,
          },
        },
      },
    ]),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

2.4 Create the Notification Controller (app.controller.ts**)**

In this section, we define a controller to handle incoming Kafka messages related to room availability.

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';
import {Ctx, KafkaContext, MessagePattern, Payload} from "@nestjs/microservices";

@Controller()
export class AppController{
  @MessagePattern('room-availability')
  handleEvent(@Payload() message: any, @Ctx() context: KafkaContext) {
    const originalMessage = context.getMessage();
    const response =
        `Receiving a new message from topic: ` + context.getTopic() + ": " +
```

```
        JSON.stringify(originalMessage.value);
    console.log(response);
    return response;
  }
}
```

The **handleEvent** method is decorated with **@MessagePattern('room-availability')**, indicating that it will process messages from the Kafka topic room-availability.

Within the method:

- **@Payload() message: any** retrieves the content of the incoming message, allowing us to access its data.
- **@Ctx() context: KafkaContext** provides access to the Kafka context, enabling us to extract additional information about the message.

In the method body, we call **context.getMessage()** to retrieve the original Kafka message and construct a response string that includes the topic name and the message value. This response is then logged to the console.

> Remarks:
>
> The **handleEvent** method currently serves as a basic verification step to ensure that the consumer successfully receives messages from the room-availability topic. At this stage, it simply logs the incoming message to the console.
>
> However, this functionality can be expanded in the future. For instance, we could notify users when a new room becomes available. This could involve sending notifications through various channels, such as email or in-app alerts. The logging serves as a foundation for developing more advanced features in response to Kafka messages.

2.5 Remove the `getHello()` method from the project as in part **1.6**.

Step 3: Launch the 2 microservices

3.1 Start the Meeting Service

Open a terminal in the root of the notification-service folder and write:

```
npm run start
```

3.3 Start the Notification Service

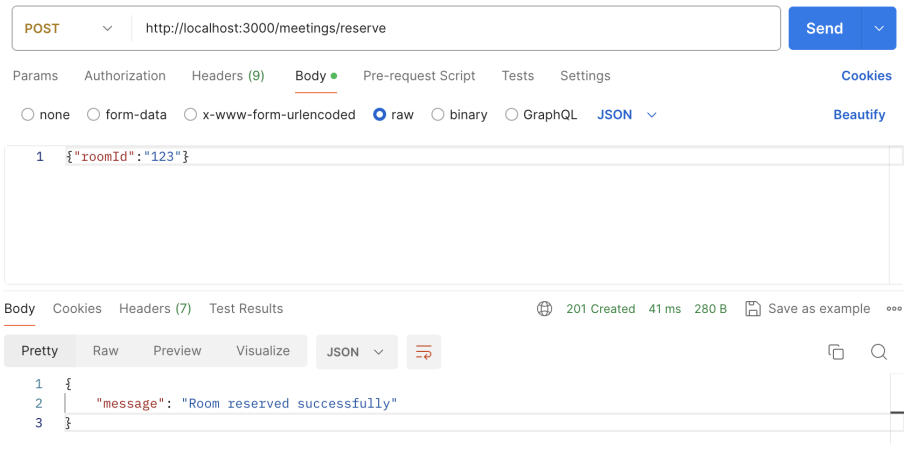
Open a terminal in the root of the meeting-service folder and write:

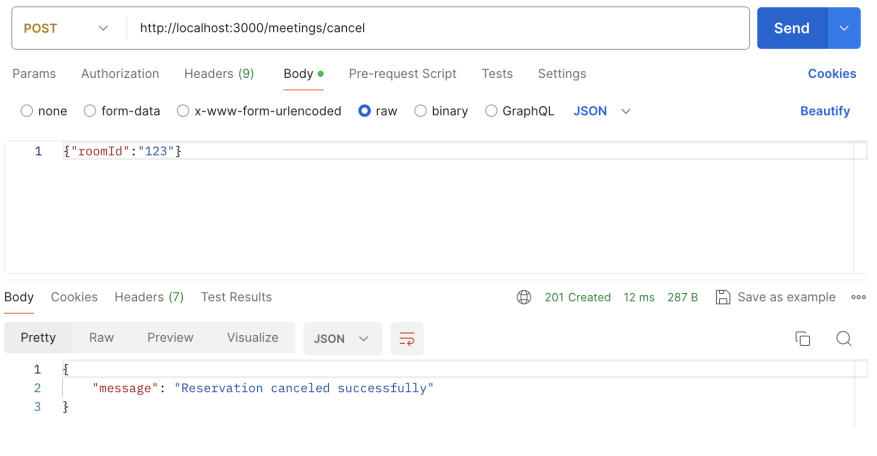
```
npm run start
```

Part 3: Testing the Event Flow

1. Use **Postman** or **cURL** to send a POST request to reserve or cancel a room using the **Meeting Service**.

- **Postman**





- **CURL**

```
curl -X POST http://localhost:3000/meetings/reserve -d '{"roomId": "123"}' -H "Content-Type: application/json"
```

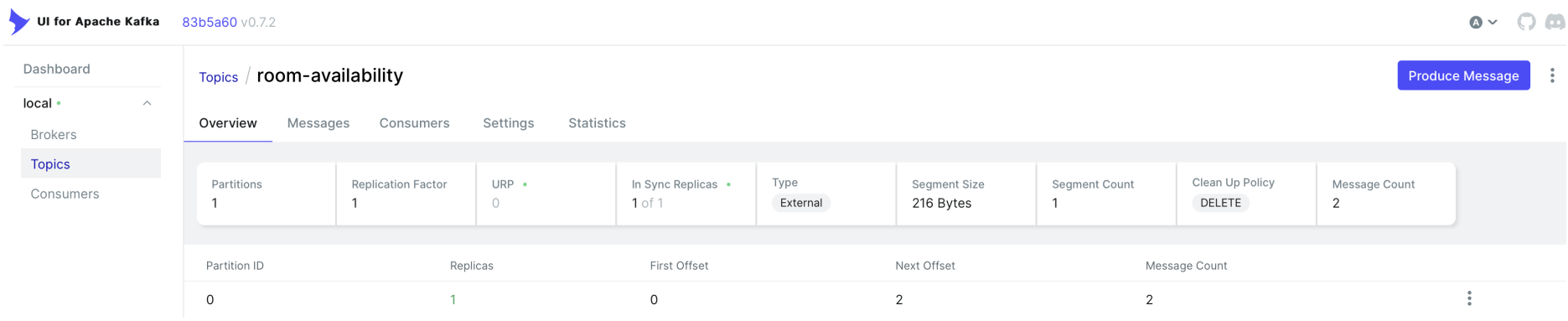
```
curl -X POST http://localhost:3000/meetings/cancel -d '{"roomId": "123"}' -H "Content-Type: application/json"
```

2. Observe the **Notification Service** logs. You should see the event:

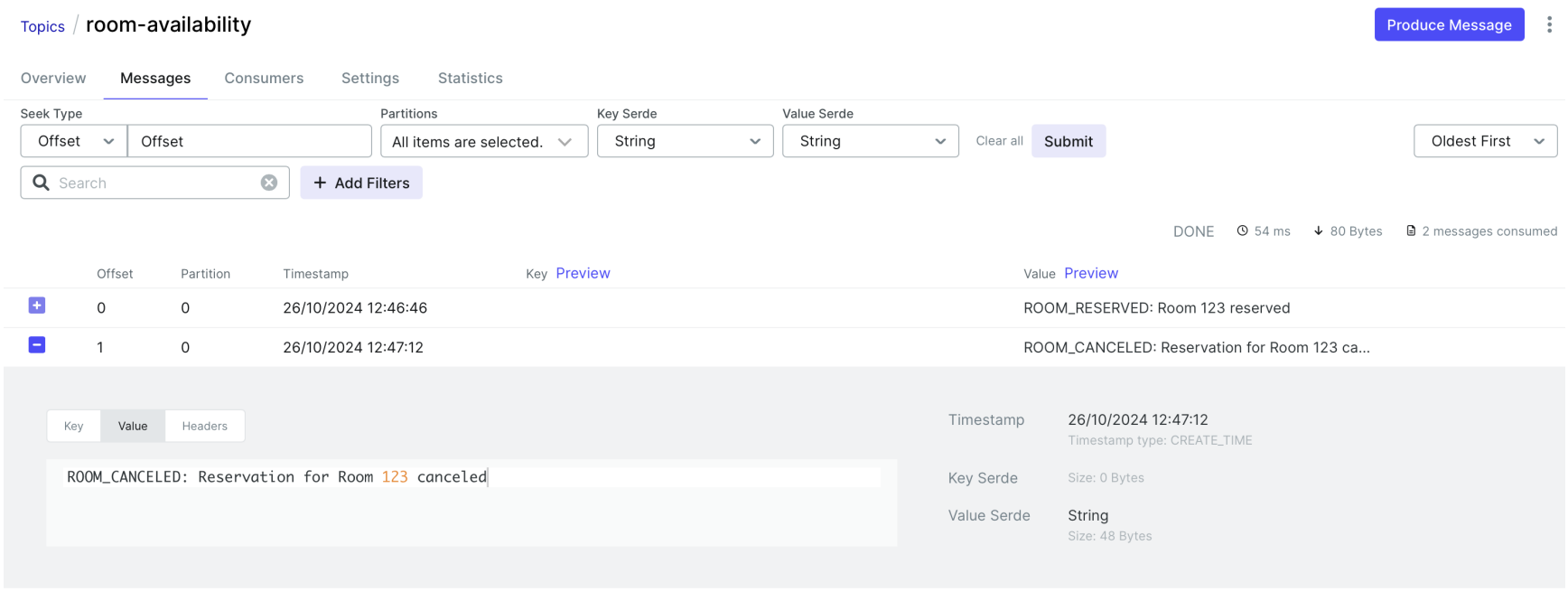
```
Receiving a new message from topic: room-availability: "ROOM_RESERVED: Room 123 reserved"
Receiving a new message from topic: room-availability: "ROOM_CANCELED: Reservation for Room 123 canceled"
```

This confirms that the event has successfully been produced by the Meeting Service and consumed by the Notification Service.

3. Observe the **Kafka UI**



We can see the topics corresponding to the communication channels where events are sent and consumed by services. In our case, we see the “room-availability” topic, used to manage room reservation events.



When you click on “Messages” (next to “Overview”), you can see all those sent on this channel. These messages represent the events generated, such as room reservations and cancellations.