Spring Boot Microservices with Spring Cloud Stream Example

MICROSERVICES SPRING BOOT SPRING CLOUD

In this tutorial, you will learn how to build two Spring Boot microservices that communicate asynchronously using **Spring Cloud Stream** with **Apache Kafka**. This guide is designed for beginners and includes a working example with detailed explanations for each step.

What You'll Learn:

- How to build microservices using Spring Boot.
- How to set up **Spring Cloud Stream** to send and receive messages using Kafka.
- How to run microservices with **Kafka** as a messaging broker.

Introduction to Spring Cloud Stream and Kafka

Spring Cloud Stream is a framework for building event-driven microservices connected to messaging systems like Kafka or RabbitMQ. It abstracts the messaging infrastructure, allowing developers to focus on writing business logic.

Why Use Spring Cloud Stream with Kafka?

- **Asynchronous Communication**: Microservices can communicate without direct calls, improving system resilience.
- **Decoupling**: Services are loosely coupled, as they only communicate through messages.
- **Scalability**: Kafka handles large volumes of data efficiently.

Prerequisites

Before starting, ensure that you have the following tools installed:

- **JDK 17** or later
- **Maven** (to build the project)
- **Kafka** and **Zookeeper** installed (or use Docker to run Kafka)
- **IDE** (IntelliJ IDEA, Eclipse, etc.)

Step 1: Create the Projects

We will create two microservices:

1. **employee-service**: Sends employee data to Kafka.
2. **department-service**: Listens to Kafka and receives employee data.

Step 2: Set Up `employee-service`

2.1 Create the Project

Go to **Spring Initializr** and generate a Spring Boot project with the following dependencies:

- **Spring Web**
- **Spring Cloud Stream**
- **Spring for Apache Kafka**

2.2 Configure `application.yml`

Create a configuration file `src/main/resources/application.yml` for the **employee-service** to define Kafka bindings.

```yaml
spring:
  cloud:
    stream:
      bindings:
        output:
          destination: employee-topic
          content-type: application/json
      kafka:
        binder:
          brokers: localhost:9092
```

Explanation:

- **output.destination=employee-topic**: Specifies the Kafka topic for sending messages.
- **brokers=localhost:9092**: Defines the Kafka broker address.

2.3 Create the Employee Model

Define an `Employee` class to represent employee data.

```java
package com.example.employeeservice;

public class Employee {
    private String id;
    private String name;
    private String department;

    // Constructors, getters, and setters
    public Employee(String id, String name, String department) {
        this.id = id;
        this.name = name;
        this.department = department;
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
```

```
    }

    public String getDepartment() {
        return department;
    }
}
```

2.4 Create a Message Producer

Create a service that will send employee data to Kafka.

```
package com.example.employeeservice;

import org.springframework.cloud.stream.function.StreamBridge;
import org.springframework.stereotype.Service;

@Service
public class EmployeeProducer {

    private final StreamBridge streamBridge;

    public EmployeeProducer(StreamBridge streamBridge) {
        this.streamBridge = streamBridge;
    }

    public void sendEmployee(Employee employee) {
        streamBridge.send("output", employee);
    }
}
```

Explanation:

- **StreamBridge**: Allows sending messages to a Kafka topic dynamically.
- **sendEmployee()**: Sends employee data to the `employee-topic`.

2.5 Create a REST Controller

Create a REST controller to trigger message sending.

```
package com.example.employeeservice;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class EmployeeController {
```

```
    private final EmployeeProducer employeeProducer;

    public EmployeeController(EmployeeProducer employeeProducer) {
        this.employeeProducer = employeeProducer;
    }

    @PostMapping("/employees")
    public String createEmployee(@RequestBody Employee employee) {
        employeeProducer.sendEmployee(employee);
        return "Employee sent: " + employee.getName();
    }
}
```

2.6 Create a Dockerfile

Create a `Dockerfile` for `employee-service`:

```
FROM openjdk:17-jdk-alpine
WORKDIR /app
COPY target/employee-service-0.0.1-SNAPSHOT.jar employee-service.jar
EXPOSE 8081
ENTRYPOINT ["java", "-jar", "employee-service.jar"]
```

Step 3: Set Up `department-service`

3.1 Create the Project

Go to **Spring Initializr** and generate another Spring Boot project with the following dependencies:

- **Spring Web**
- **Spring Cloud Stream**
- **Spring for Apache Kafka**

3.2 Configure `application.yml`

Create a configuration file `src/main/resources/application.yml` for `department-service`:

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: employee-topic
          content-type: application/json
      kafka:
```

```
        binder:
          brokers: localhost:9092
```

Explanation:

- **input.destination=employee-topic**: Listens to the same Kafka topic (`employee-topic`) to receive messages.

3.3 Create the Employee Model

Create the same `Employee` model as in `employee-service` to deserialize the received message:

```
package com.example.departmentservice;

public class Employee {
    private String id;
    private String name;
    private String department;

    // Constructors, getters, and setters
    public Employee() {}

    public Employee(String id, String name, String department) {
        this.id = id;
        this.name = name;
        this.department = department;
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getDepartment() {
        return department;
    }
}
```

3.4 Create a Message Consumer

Create a service to consume employee data from Kafka.

```
package com.example.departmentservice;

import org.springframework.context.annotation.Bean;
```

```
import org.springframework.stereotype.Service;
import java.util.function.Consumer;

@Service
public class EmployeeConsumer {

    @Bean
    public Consumer<Employee> input() {
        return employee -> {
            System.out.println("Received employee: " + employee.getName() + " from department " + employee
        };
    }
}
```

Explanation:

- **@Bean Consumer<Employee> input()**: Registers a function that consumes messages from Kafka.

3.5 Create a Dockerfile

Create a `Dockerfile` for `department-service`:

```
FROM openjdk:17-jdk-alpine
WORKDIR /app
COPY target/department-service-0.0.1-SNAPSHOT.jar department-service.jar
EXPOSE 8082
ENTRYPOINT ["java", "-jar", "department-service.jar"]
```

Step 4: Set Up Kafka with Docker Compose

Create a `docker-compose.yml` file to run **Kafka** and **Zookeeper**:

```
version: '3.8'

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:latest
    environment:
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
```

```
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    ports:
      - "9092:9092"
    depends_on:
      - zookeeper
```

Run Kafka and Zookeeper:

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

Step 5: Build Docker Images

Navigate to the root directories of each service and run:

For `employee-service`:

```
mvn clean package
docker build -t employee-service .
```

For `department-service`:

```
mvn clean package
docker build -t department-service .
```

Step 6: Create a Docker Compose File for Services

Create a `docker-compose.yml` file to run both microservices with Kafka:

```
version: '3.8'

services:
  employee-service:
    image: employee-service
    build:
      context: ./employee-service
    ports:
      - "8081:8081"
    networks:
      - microservices-net

  department-service:
    image: department-service
    build:
      context: ./department-service
    ports:
      - "8082:8082"
    networks:
```

```
      - microservices-net

  kafka:
    image: confluentinc/cp-kafka:latest
    ports:
      - "9092:9092"
    environment:
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
    networks:
      - microservices-net

  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOK

EEPER_CLIENT_PORT: 2181
    ports:
      - "2181:2181"
    networks:
      - microservices-net

networks:
  microservices-net:
    driver: bridge
```

Step 7: Run Docker Compose

Navigate to the directory containing the `docker-compose.yml` file and run:

```
docker-compose up --build
```

Docker Compose will build and start the containers.

Step 8: Test the Microservices Communication

Use **Postman** or **curl** to send employee data to the `employee-service`:

```
curl -X POST http://localhost:8081/employees \
-H "Content-Type: application/json" \
-d '{"id":"1", "name":"John Doe", "department":"Engineering"}'
```

The `department-service` should log the received employee data in the console.

Conclusion

You have successfully built two Spring Boot microservices that communicate asynchronously using **Spring Cloud Stream** and **Kafka**. This setup demonstrates how to build scalable, event-driven microservices architecture.

Next Steps:

- Add more microservices to the system.
- Implement error handling and retries for message delivery.