Implementar um chat em tempo real usando Java, Docker e Kafka envolve várias etapas. Vamos criar um sistema simples de chat com os seguintes componentes:

1. **Producer Service:** Um serviço que envia mensagens para um tópico Kafka.
2. **Consumer Service:** Um serviço que lê mensagens de um tópico Kafka e as transmite para os clientes.
3. **WebSocket Server:** Um servidor que permite a comunicação em tempo real entre o cliente e o servidor.
4. **Frontend:** Uma aplicação web simples para o chat.

**Passo a Passo**

**1. Configurar o Ambiente de Desenvolvimento**

* Instale o JDK, Maven, Docker e Kafka.
* Para facilitar, vamos usar o Docker Compose para configurar o Kafka.

**2. Configurar Kafka com Docker Compose**

Crie um arquivo docker-compose.yml para configurar o Kafka e o Zookeeper:

version: '2'

services:

zookeeper:

image: wurstmeister/zookeeper:latest

ports:

- "2181:2181"

kafka:

image: wurstmeister/kafka:latest

ports:

- "9092:9092"

expose:

- "9093"

environment:

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://localhost:9092

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

volumes:

- /var/run/docker.sock:/var/run/docker.sock

Inicie o Kafka:

docker-compose up -d

Verifique se as imagens estão no ar:

$ **sudo docker ps -a**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

a657b88c56a5 wurstmeister/kafka:latest "start-kafka.sh" 32 minutes ago Exited (1) 32 minutes ago chat\_kafka\_1

daee39c25a22 wurstmeister/zookeeper:latest "/bin/sh -c '/usr/sb…" 32 minutes ago Up 32 minutes 22/tcp, 2888/tcp, 3888/tcp, 0.0.0.0:2181->2181/tcp, :::2181->2181/tcp chat\_zookeeper\_1

**3. Criar o Producer Service**

**a. Dependências Maven**

Adicione as dependências do Kafka e WebSocket ao seu pom.xml:

<dependencies>

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-clients</artifactId>

<version>2.8.0</version>

</dependency>

<dependency>

<groupId>javax.websocket</groupId>

<artifactId>javax.websocket-api</artifactId>

<version>1.1</version>

</dependency>

<dependency>

<groupId>org.glassfish.tyrus</groupId>

<artifactId>tyrus-server</artifactId>

<version>1.13.1</version>

</dependency>

</dependencies>

**b. Implementar o Producer**

Crie a classe ChatProducer:

import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.ProducerRecord;

import org.apache.kafka.clients.producer.ProducerConfig;

import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;

public class ChatProducer {

private final KafkaProducer<String, String> producer;

private final String topic;

public ChatProducer(String topic) {

Properties props = new Properties();

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class.getName());

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class.getName());

producer = new KafkaProducer<>(props);

this.topic = topic;

}

public void sendMessage(String message) {

producer.send(new ProducerRecord<>(topic, message));

}

public void close() {

producer.close();

}

}

**c. Implementar o WebSocket Server**

Crie a classe WebSocketServer:

import javax.websocket.OnMessage;

import javax.websocket.OnOpen;

import javax.websocket.Session;

import javax.websocket.server.ServerEndpoint;

import javax.websocket.OnClose;

import javax.websocket.server.ServerEndpointConfig;

import java.io.IOException;

import java.util.Set;

import java.util.concurrent.CopyOnWriteArraySet;

@ServerEndpoint(value = "/chat")

public class WebSocketServer {

private Session session;

private static final Set<WebSocketServer> connections = new CopyOnWriteArraySet<>();

private static ChatProducer producer;

@OnOpen

public void onOpen(Session session) {

this.session = session;

connections.add(this);

if (producer == null) {

producer = new ChatProducer("chat\_topic");

}

}

@OnMessage

public void onMessage(String message) {

producer.sendMessage(message);

}

@OnClose

public void onClose(Session session) {

connections.remove(this);

}

public static void broadcast(String message) {

for (WebSocketServer client : connections) {

try {

synchronized (client) {

client.session.getBasicRemote().sendText(message);

}

} catch (IOException e) {

connections.remove(client);

try {

client.session.close();

} catch (IOException ex) {

// Ignore

}

}

}

}

}

**4. Criar o Consumer Service**

**a. Implementar o Consumer**

Crie a classe ChatConsumer:

import org.apache.kafka.clients.consumer.ConsumerConfig;

import org.apache.kafka.clients.consumer.ConsumerRecords;

import org.apache.kafka.clients.consumer.KafkaConsumer;

import org.apache.kafka.clients.consumer.ConsumerRecord;

import org.apache.kafka.common.serialization.StringDeserializer;

import java.time.Duration;

import java.util.Collections;

import java.util.Properties;

public class ChatConsumer {

private final KafkaConsumer<String, String> consumer;

private final String topic;

public ChatConsumer(String topic) {

Properties props = new Properties();

props.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

props.put(ConsumerConfig.GROUP\_ID\_CONFIG, "chat\_group");

props.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class.getName());

props.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class.getName());

consumer = new KafkaConsumer<>(props);

this.topic = topic;

consumer.subscribe(Collections.singletonList(topic));

}

public void consumeMessages() {

while (true) {

ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));

for (ConsumerRecord<String, String> record : records) {

WebSocketServer.broadcast(record.value());

}

}

}

public void close() {

consumer.close();

}

}

**b. Inicializar o Consumer**

Crie a classe ChatConsumerApp:

public class ChatConsumerApp {

public static void main(String[] args) {

ChatConsumer consumer = new ChatConsumer("chat\_topic");

consumer.consumeMessages();

}

}

**5. Criar Dockerfiles para os Serviços**

**Dockerfile para o Producer Service**

Crie um arquivo Dockerfile no diretório do Producer Service:

FROM openjdk:11-jre-slim

WORKDIR /app

COPY target/producer-service-jar-with-dependencies.jar producer-service.jar

EXPOSE 8080

CMD ["java", "-jar", "producer-service.jar"]

**Dockerfile para o Consumer Service**

Crie um arquivo Dockerfile no diretório do Consumer Service:

FROM openjdk:11-jre-slim

WORKDIR /app

COPY target/consumer-service-jar-with-dependencies.jar consumer-service.jar

CMD ["java", "-jar", "consumer-service.jar"]

**6. Construir as Imagens Docker**

No diretório de cada serviço, execute:

mvn package

sudo docker build -f producerService-Dockerfile -t producer-service .

sudo docker build -f consumerService-Dockerfile -t consumer-service .

**7. Criar Manifests do Kubernetes**

**Deployment e Service para o Producer Service**

Crie um arquivo producer-service.yaml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: producer-service

spec:

replicas: 1

selector:

matchLabels:

app: producer-service

template:

metadata:

labels:

app: producer-service

spec:

containers:

- name: producer-service

image: producer-service

ports:

- containerPort: 8080

---

apiVersion: v1

kind: Service

metadata:

name: producer-service

spec:

selector:

app: producer-service

ports:

- protocol: TCP

port: 8080

targetPort: 8080

**Deployment e Service para o Consumer Service**

Crie um arquivo consumer-service.yaml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: consumer-service

spec:

replicas: 1

selector:

matchLabels:

app: consumer-service

template:

metadata:

labels:

app: consumer-service

spec:

containers:

- name: consumer-service

image: consumer-service

**8. Deploy no Kubernetes**

Inicie o Minikube (se estiver usando):

minikube start

Implante os serviços:

kubectl apply -f producer-service.yaml

kubectl apply -f consumer-service.yaml

**9. Criar o Frontend**

Crie um arquivo HTML simples para o frontend:

<!DOCTYPE html>

<html>

<head>

<title>Chat</title>

</head>

<body>

<h1>Chat</h1>

<div id="chatbox"></div>

<input type="text" id="message" placeholder="Type a message..."/>

<button onclick="sendMessage()">Send</button>

<script>

let ws = new WebSocket("ws://localhost:8080/chat");

ws.onmessage = function(event) {

let chatbox = document.getElementById("chatbox");

chatbox.innerHTML += event.data + "<br/>";

};

function sendMessage() {

let message = document.getElementById("message").value;

ws.send(message);

}

</script>

</body>

</html>

Coloque este arquivo em um servidor web (por exemplo, usando Python para simplicidade):

python3 -m http.server 8000

**Conclusão**

Este exemplo mostra como criar um chat em tempo real usando Java, Docker e Kafka. Ele inclui a configuração do Kafka usando Docker Compose, a implementação de serviços Producer e Consumer em Java, e a configuração de WebSocket para comunicação em tempo real. Este exemplo pode ser expandido e melhorado para incluir mais funcionalidades e otimizações.