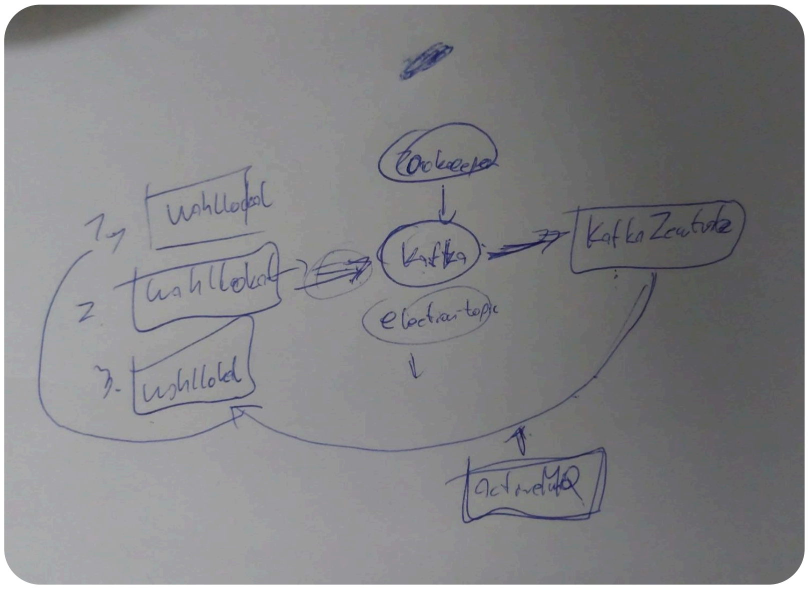
#CS #school

Author: Oliwier Przewlocki

preparation & structure



The entire project is instantiated with a docker-compose file, to leverage the easy expanding of the number of voting centers. Here is the docker-compose file used:

```yaml
version: '3'
services:
  zookeeper:
    image: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  kafka:
    image: wurstmeister/kafka:latest
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: "true"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock

  wahllokal1:
    image: wahllokal
    ports:
      - "8081:4200"
    container_name: wahllokal1
    environment:
      SERVER_PORT: 4200
      SPRING_PROFILES_ACTIVE: wahllokal1
      KAFKA_BROKER: kafka:9092
      ACTIVEMQ_BROKER: activemq:61616

  wahllokal2:
    image: wahllokal
    ports:
      - "8082:4200"
    container_name: wahllokal2
    environment:
      SERVER_PORT: 4200
      SPRING_PROFILES_ACTIVE: wahllokal2
      KAFKA_BROKER: kafka:9092
      ACTIVEMQ_BROKER: activemq:61616

  wahllokal3:
    image: wahllokal
    ports:
      - "8083:4200"
    container_name: wahllokal3
    environment:
      SERVER_PORT: 4200
      SPRING_PROFILES_ACTIVE: wahllokal3
      KAFKA_BROKER: kafka:9092
      ACTIVEMQ_BROKER: activemq:61616

  kafkazentrale:
    image: kafkazentrale
    ports:
      - "8084:8080"
    container_name: kafkazentrale
    environment:
      SERVER_PORT: 8080
      SPRING_PROFILES_ACTIVE: kafkazentrale
      KAFKA_BROKER: kafka:9092
      ACTIVEMQ_BROKER: activemq:61616

  activemq:
    image: rmohr/activemq:latest
    container_name: activemq
    ports:
      - "61616:61616"
      - "8161:8161"
    environment:
      ACTIVEMQ_ADMIN_LOGIN: admin
      ACTIVEMQ_ADMIN_PASSWORD: admin
```

- **Zookeeper**: keeps track of which brokers are part of the Kafka cluster
- **Kafka**: A messaging system (Event Streaming Platform) comprised of queues, brokers, clusters, etc.
- **wahllokal**: The voting center, i.e. the Producer.
- **kafkazentrale**: The Consumer, it listens to the producers and saves their incoming messages
- **ActiveMQ**: The message broker for communicating with the wahllokal that the message was successfully received.

The **wahllokals** are a modified version of the first assignment. Here are the necessary changes

Gkv

wahllokal

```
spring.kafka.bootstrap-servers=kafka:9092
spring.activemq.broker-url=tcp://activemq:61616
```

> In `application.properties`

```yaml
spring:
  kafka:
    bootstrap-servers: localhost:9092
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer
```

> In `application.yml` created on the same level as application.properties

```xml
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

> In `pom.xml`

Now we have to create a service that will communicate with a kafka topic and send the data.

```java
@Service
@AllArgsConstructor
public class ElectionProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public void sendElectionData(String regionId, WarehouseData data) {
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            String message = objectMapper.writeValueAsString(data);
            kafkaTemplate.send("election-topic", regionId, message);

            System.out.println("Wahllokal " + regionId
                + " hat die Wahldaten gesendet: " + message);
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
    }
}
```

Here the data is serialized and sent to the "election-topic" topic. There's also a log message. This method will be used in the controller that regulates the `/warehouse/{inID}/data` endpoint.

```
private final ElectionProducer producer;

//...

//@GetMapping...

producer.sendElectionData(inID, data);
```

Creating a Dockerimage

On the same level as the `pom.xml`, the Dockerfile needs to be created.

```
FROM openjdk:17-jdk-alpine

WORKDIR /app

COPY libs/proto-library-1.0-SNAPSHOT.jar /app/libs/proto-library-1.0-SNAPSHOT.jar

COPY .mvn/ .mvn
COPY mvnw .
COPY pom.xml .

RUN ./mvnw install:install-file \
    -Dfile=/app/libs/proto-library-1.0-SNAPSHOT.jar \
    -DgroupId=com.oliwier \
    -DartifactId=proto-library \
    -Dversion=1.0-SNAPSHOT \
    -Dpackaging=jar

RUN ./mvnw dependency:go-offline

COPY src ./src

RUN ./mvnw clean package -DskipTests
```

```
EXPOSE 4200

CMD ["java", "-jar", "target/nationalwahlen-0.0.1-SNAPSHOT.jar"]
```

> The proto-library needs to be built extra, because it is a local library and can’t be directly accessed from the isolated container.

KafkaZentrale

Because we need to deserialize the data, we need the model structure from the Wahllokal project. We need to create the appropriate classes (`PartyData`, `PreferredCandidate`, `WarehouseData`). An alternative to the manual creation would be to create a shared library, but that would be too much work so I went with the manual creation.

First, we need to set up the kafka service and add the appropriate dependencies:

```
spring.kafka.bootstrap-servers=kafka:9092
spring.activemq.broker-url=tcp://activemq:61616
```

> In `application.properties`

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: election-group
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
```

> In `application.yml` created on the same level as application.properties

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

> In `pom.xml`

Now, we need a Consumer service to receive the messages sent by the Wahllokals

```

@Getter
@Service
@AllArgsConstructor
```

```java
public class ElectionConsumer {

    private final SuccessMessageService successMessageService;
    private List<WarehouseData> collectedData;

    @KafkaListener(topics = "election-topic", groupId = "election-group")
    public void consumeElectionData(String message) {

        System.out.println("Received election data: " + message);
        ObjectMapper objectMapper = new ObjectMapper();
        try {

            WarehouseData data = objectMapper
                .readValue(message, WarehouseData.class);
            collectedData.add(data);

            successMessageService.sendSuccessMessage(data.getRegionID());

        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
    }
}
```

Whenever a Wahllokal sends a message to the election-topic, the `consumeElectionData` method gets triggered and the data is received, deserialized and added to an ArrayList.

The `sendSuccessMessage` method will be mentioned in the Ek part of this assignment.

Now we just need to create a Controller that opens an endpoint:

```java


@RestController
@AllArgsConstructor
public class CentralController {

    private final ElectionConsumer consumer;

    @GetMapping(value = "/election2024/results", produces = {
        MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE })
    public ResponseEntity<List<WarehouseData>> getElectionResults(@RequestParam
    (value = "format", required = false, defaultValue = "json") String format) {

        List<WarehouseData> results = consumer.getCollectedData();

        HttpHeaders headers = new HttpHeaders();
        if ("xml".equalsIgnoreCase(format)) {
            headers.setContentType(MediaType.APPLICATION_XML);
        } else {
            headers.setContentType(MediaType.APPLICATION_JSON);
        }
        return new ResponseEntity<>(results, headers, HttpStatus.OK);
    }}
```

That either returns an xml or a json depending on the format chosen.

The last step is to generate a Docker image using a Dockerfile

```
FROM openjdk:17-jdk-alpine

WORKDIR /app

COPY .mvn/ .mvn
COPY mvnw .
COPY pom.xml .

RUN ./mvnw dependency:go-offline

COPY src ./src

RUN ./mvnw clean package -DskipTests

EXPOSE 8080

CMD ["java", "-jar", "target/KafkaZentrale-0.0.1-SNAPSHOT.jar"]
```

> Maven is copied and buit extra because it needs to build the package.

Now you can execute `docker-compose up -d` in teh folder with the docker-compose file and everything should work.

Ekv

For the success message to be sent, you need to have an ActiveMQ dependency added to both KafkaZentral and Wahllokal:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

KafkaZentrale

You need to add a service that sends the message via ActiveMQ:

```
@Service
@AllArgsConstructor
public class SuccessMessageService {
    private final JmsTemplate jmsTemplate;

    public void sendSuccessMessage(String regionId) {
        String message = "SUCCESS for Wahllokal " + regionId;
        jmsTemplate.convertAndSend("successTopic", message);
        System.out.println("Zentralrechner hat eine Rückmeldung gesendet: "
            + message);
    }
}
```

Wahllokal

You need to add a service that receives the message sent from KafkaZentrale:

```
@Service
public class ZentraleListener {
    @JmsListener(destination = "successTopic")
    public void receiveSuccessMessage(String message) {
        System.out.println("Wahllokal hat die Rückmeldung erhalten: " + message);
    }
}
```

Btw, you can also send the electionData via:

```
@Service
@AllArgsConstructor
public class ElectionProducer {

    private final JmsTemplate jmsTemplate;

    public void sendElectionData(String regionId, WarehouseData data) {
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            String message = objectMapper.writeValueAsString(data);
            jmsTemplate.convertAndSend("electionQueue", message);
            System.out.println("Wahllokal " + regionId
                + " hat die Wahldaten gesendet: " + message);
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
    }
}
```

And

```


@Getter
@Service
@AllArgsConstructor
```

```java
public class ElectionConsumer {

    private final SuccessMessageService successMessageService;
    private List<WarehouseData> collectedData;

    @JmsListener(destination = "electionQueue")
    public void consumeElectionData(String message) {
        System.out.println("Received election data: " + message);

        ObjectMapper objectMapper = new ObjectMapper();
        try {
            WarehouseData data = objectMapper.readValue(message,
                WarehouseData.class);
            collectedData.add(data);
            successMessageService.sendSuccessMessage(data.getRegionID());
            System.out.println("Daten wurden erfolgreich deserialisiert und gespeichert.");
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
    }
}
```