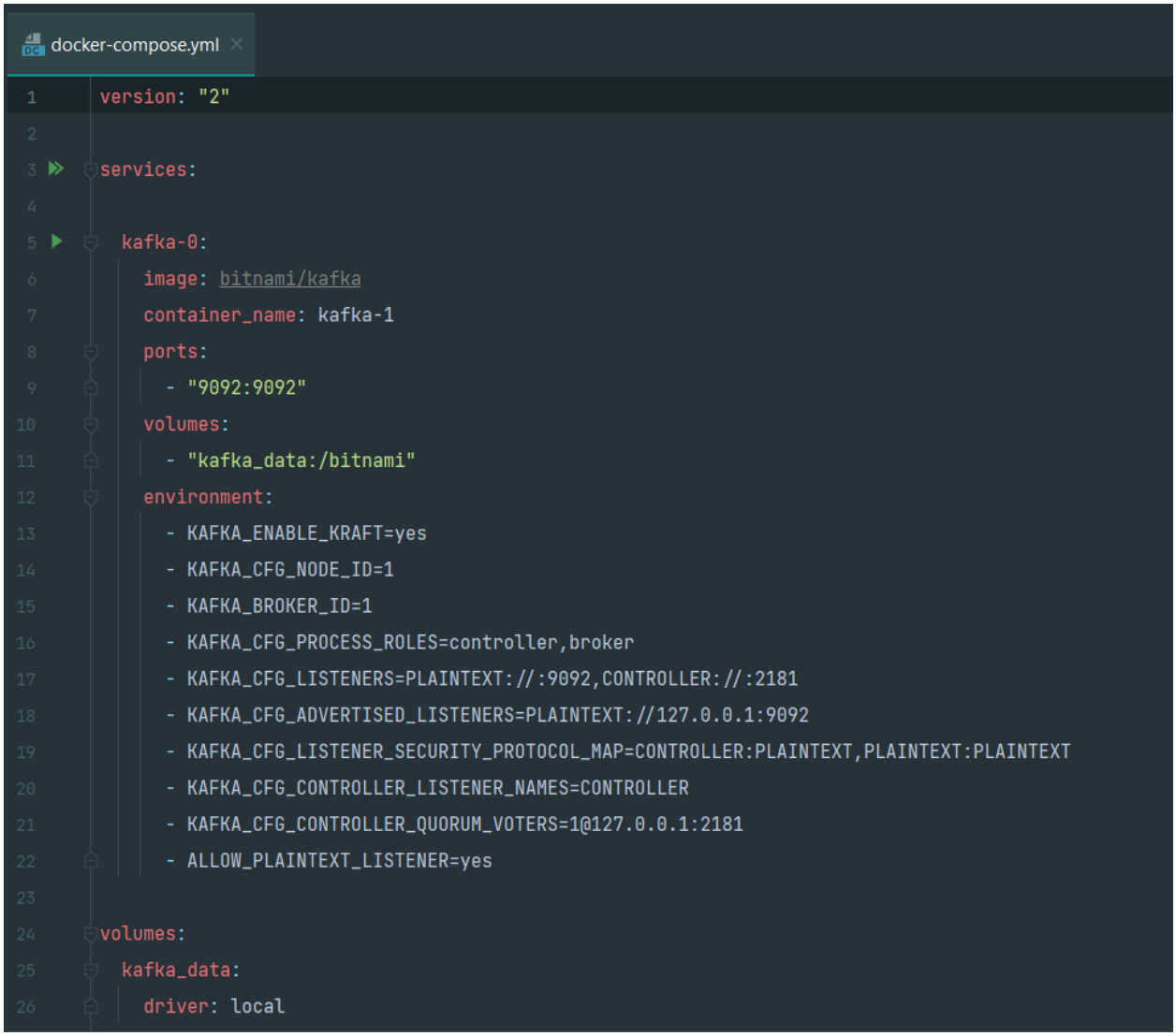
1. Create a docker compose file to create kafka broker



- `KAFKA_ENABLE_KRAFT=yes`: enables Kafka's kRaft mode

- `KAFKA_CFG_NODE_ID=1`: sets the unique identifier for the Kafka node (broker). In a multi-broker setup, each broker should have a unique node ID

- `KAFKA_BROKER_ID=1`: specifies the unique identifier for the Kafka broker. The broker ID should match the node ID you've set earlier. It ensures that the broker is uniquely identified in the Kafka cluster.

***Notice**:*

`KAFKA_CFG_NODE_ID` is used when you are running Kafka in KRaft mode, and it's essential for the broker's participation in the Raft-based consensus protocol.

`KAFKA_BROKER_ID` is used in traditional Kafka configurations and is essential for broker identification within the Kafka cluster.

If you are not running Kafka in KRaft mode, you can technically omit `KAFKA_CFG_NODE_ID` and only set `KAFKA_BROKER_ID` to define the broker's identity. However, it's still a good practice to include both for consistency and to make it easier to switch between different Kafka modes if needed in the future.

- `KAFKA_CFG_PROCESS_ROLES=controller,broker`: Specifies the roles that the Kafka node will assume. In this case, it's acting both as a controller (handling metadata and cluster management) and as a broker (handling data storage and serving client requests).

- `KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,CONTROLLER://:2181`: define the listener configurations for the Kafka broker

- `KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://127.0.0.1:9092`: This specifies the advertised listener configuration. It tells clients how to reach this broker. In this case, it's configured to advertise the broker's hostname as `127.0.0.1` and port `9092` for plaintext communication.

- `KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT`: defines a map of listener names to security protocols. In this case, it maps the listener `CONTROLLER` to the `PLAINTEXT` security protocol and the listener `PLAINTEXT` to `PLAINTEXT`. It specifies that both listeners use plaintext communication.

- `KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER`: specifies the listener name used for controller communication. In this case, it's set to `CONTROLLER`

- `KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@127.0.0.1:2181:` define the controller quorum voters. This is essential for leader election and coordination.

- `ALLOW_PLAINTEXT_LISTENER=yes:` allows plaintext listeners for this Kafka broker

2. Launch the container using : "docker compose up" in cmd

3. Using "docker exec -it <container-name> bash" command to interact with the container

4. Create a topic using the following command:

kafka-topics.sh --create --topic <topic-name> --bootstrap-server <broker-list> --partitions <num-partitions> --replication-factor <replication-factor>

***Notice**: bootstrap-server <broker-list> is defined in the environment variable: `KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT`*

Ex: "`kafka-topics.sh --create --topic my-topic --bootstrap-server 127.0.0.1:9092 --partitions 3 --replication-factor 1`"

"

5. Create a Producer

-Kafka Consumer Configuration:

```
String bootstrapServers = "127.0.0.1:9092";

Properties properties = new Properties();
properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
properties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
properties.setProperty(ProducerConfig.ACKS_CONFIG, "1");
```

+ Define the Kafka broker's address (`bootstrapServers`) that the producer will connect to.
+ Configure the Kafka producer with various properties:

- `ProducerConfig.BOOTSTRAP_SERVERS_CONFIG`: Set the list of Kafka brokers to connect to. In this case, it's `127.0.0.1:9092`.
- `ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG`: Specify the serializer for message keys. Here, it's set to use the StringSerializer.
- `ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG`: Specify the serializer for message values. Also, it uses the StringSerializer.
- `ProducerConfig.ACKS_CONFIG`: Set the acknowledgment mode to "1," which means the producer receives acknowledgment once the leader broker has received the message.

- **Create an instance of the KafkaProducer class by passing in the configured properties.**

```
KafkaProducer<String, String> producer = new KafkaProducer<>(properties);
```

- **Produce message to Kafka:**

```java
for(int i = 0; i < 10; i++) {
    ProducerRecord<String, String> producerRecord =
            new ProducerRecord<>( topic: "demojava", value: "my message ");

    Phuong Tran La Nhat
    producer.send(producerRecord, new Callback() {
        Phuong Tran La Nhat
        @Override
        public void onCompletion(RecordMetadata recordMetadata, Exception e) {
            if(e == null) {
                log.info("Receive new metadata \n" +
                        "Topic: " + recordMetadata.topic() + "\n" +
                        "Partition: " + recordMetadata.partition() + "\n" +
                        "Offset: " + recordMetadata.offset() + "\n" +
                        "Timestamp: " + recordMetadata.timestamp());
            } else {
                log.error("Error while producing", e);
            }
        }
    });
    try {
        Thread.sleep( millis: 500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

- This loop sends ten messages to the "demojava" Kafka topic. You can adjust the loop's iteration count to produce more or fewer messages.
- `ProducerRecord`: Create a ProducerRecord object with the topic name ("demojava") and the message content ("my message").
- `producer.send`: Send the producer record to the Kafka cluster. It includes a callback function to handle the acknowledgment and logging of metadata.
- `Thread.sleep`: Introduce a delay (500 milliseconds) between sending messages to simulate a slower production rate. You can adjust the sleep duration as needed.

- **Clean up and close:**

```
producer.flush();

producer.close();
```

+ `producer.flush()`: Ensure that any pending messages are sent before closing the producer.
+ `producer.close()`: Close the Kafka producer to release its resources.

6. Create a Consumer:

- **Kafka Consumer Configuration:**

```
String bootstrapServers = "127.0.0.1:9092";
String groupId = "my-group";
String topic = "demojava";

Properties properties = new Properties();
properties.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
properties.setProperty(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
properties.setProperty(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
properties.setProperty(ConsumerConfig.GROUP_ID_CONFIG, groupId);
properties.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
```

+ Define the Kafka broker's address (`bootstrapServers`) that the consumer will connect to.
+ Configure the Kafka consumer with various properties:
    - `ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG`: Set the list of Kafka brokers to connect to. In this case, it's `127.0.0.1:9092`.
    - `ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG`: Specify the deserializer for message keys. Here, it's set to use the StringDeserializer.
    - `ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG`: Specify the deserializer for message values. Also, it uses the StringDeserializer.

- `ConsumerConfig.GROUP_ID_CONFIG`: Set the consumer group ID. Messages with the same group ID are consumed by one consumer within the group.
- `ConsumerConfig.AUTO_OFFSET_RESET_CONFIG`: Set the consumer's start offset to "earliest," which means it will consume messages from the beginning of the topic.

- **Kafka Consumer Initialization**

  Create an instance of the KafkaConsumer class by passing in the configured properties.

```
Consumer<String, String> consumer = new KafkaConsumer<>(properties);
```

- **Subscribe to Kafka Topic**

  Subscribe the consumer to the Kafka topic named "demojava." You can adjust this to subscribe to multiple topics if needed.

```
consumer.subscribe(Collections.singleton(topic));
```

- **Consume Messages**
  + Enter an infinite loop to continuously poll for new messages from the subscribed topic.
  + `consumer.poll`: Poll for records with a timeout of 100 milliseconds. Adjust the timeout as needed based on your application's requirements.
  + For each received record, the code logs the topic, key, value, partition, and offset.

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    records.forEach(record -> {
        log.info("Received message:\n" +
                        "Topic: {}\n" +
                        "Key: {}\n" +
                        "Value: {}\n" +
                        "Partition: {}\n" +
                        "Offset: {}",
                record.topic(), record.key(), record.value(),
                record.partition(), record.offset());
    });
}
```