

---

**Author:** Manuel Fellner

**Version:** 14.11.2023

# 1. Beantwortung der Fragestellung

## 1. Eigenschaften der Message Orientated Middleware (MOM)

- **Nachrichtenvermittlung:** MOM ermöglicht die Kommunikation zwischen verteilten Anwendungen/Systemen durch den Austausch von Nachrichten
- **Asynchrone Kommunikation:** MOM unterstützt asynchrone Kommunikation, bei der Sender und Empfänger jedoch nicht gleichzeitig aktiv sein müssen.
- **Zuverlässigkeit:** MOM sorgt für die zuverlässige Zustellung von Nachrichten, selbst wenn Systemkomponenten ausfallen oder eine vorübergehende Downtime haben.
- **Nachrichtenverwaltung:** Neben der Vermittlung bietet MOM auch die Verwaltung dieser Nachrichten in z.B. Form von Message Queues an.

## 2. Was versteht man unter einer transienten und synchronen Kommunikation?

- **Transiente Kommunikation:** Hierbei werden Nachrichten nicht permanent in der Queue gespeichert; der Empfänger muss also zur selben Zeit erreichbar sein. Ansonsten geht die Nachricht verloren.
- **Synchrone Kommunikation:** Hierbei wartet der Sender auf eine Antwort vom Empfänger. Kommunikation erfolgt in Echtzeit, das Programm (oder Ähnliches) wird erst weiter ausgeführt, wenn die Nachricht erhalten wurde.

## 3. Funktionsweise einer JMS Queue

- Eine JMS Queue ist eine "Warteschlange", in der Nachrichten in der Reihenfolge ihres Eingangs abgelegt werden (z.B. ähnlich wie ein E-Mail Postfach)
- Sender senden Nachrichten an die Queue, und Empfänger lesen Nachrichten aus der Queue.
- Die Nachrichten bleiben in der Queue, bis sie erfolgreich von einem Empfänger verarbeitet wurden (wenn PERSISTENT)
- Jeder Empfänger erhält eine Kopie der Nachricht, und nur der Empfänger, der zuerst auf die Nachricht zugreift, verarbeitet sie.

## 4. JMS Overview - Wichtige JMS-Klassen im Zusammenhang

- **ConnectionFactory:** Erzeugt Connections für die Anwendung
- **Connection:** Bietet eine Verbindung zur JMS-Implementierung her
- **Session:** Sitzung für die Erstellung von Producern und Consumern.
- **Destination:** Stellt das Ziel für Nachricht dar (Queue oder Topic)
- **MessageProducer:** Sendet Nachrichten an eine Destination

- **MessageConsumer**: Empfängt Nachrichten von einer Destination
- **Message (TextMessage)**: Repräsentiert die eigentliche Nachricht

## 5. Funktionsweise eines JMS Topic

- Eine JMS Topic ist ein Mechanismus für die Veröffentlichung/Abonnierung von Nachrichten
- Sender senden Nachricht an ein Topic und alle Abonnenten dieses Topics erhalten eine Kopie der Nachricht.
- Die Nachricht wird an alle aktiven Abonnenten verteilt, wodurch ein Broadcast entsteht
- Es ermöglicht die Implementierung von Publish-Subscribe-Messaging

## 6. Lose gekoppeltes verteiltes System

- Ein lose gekoppeltes verteiltes System ist ein System, dessen Komponenten unabhängig voneinander funktionieren und miteinander kommunizieren können, ohne stark voneinander abhängig zu sein
- **Beispiel**: Web Services, bei denen Dienste über standardisierte Schnittstellen kommunizieren. Jede Komponente kann Änderungen vornehmen, solange sie die vereinbarte Schnittstelle einhält
- "Lose gekoppelt" bedeutet, dass Änderungen in einer Komponente minimale Auswirkungen auf andere Komponenten haben und die Interaktion über gut definierte Schnittstellen erfolgt (wie es hier z.B. mit ApacheMQ passiert)

# 2. Durchführung der Übung

## 2.1 Installation von ApacheMQ

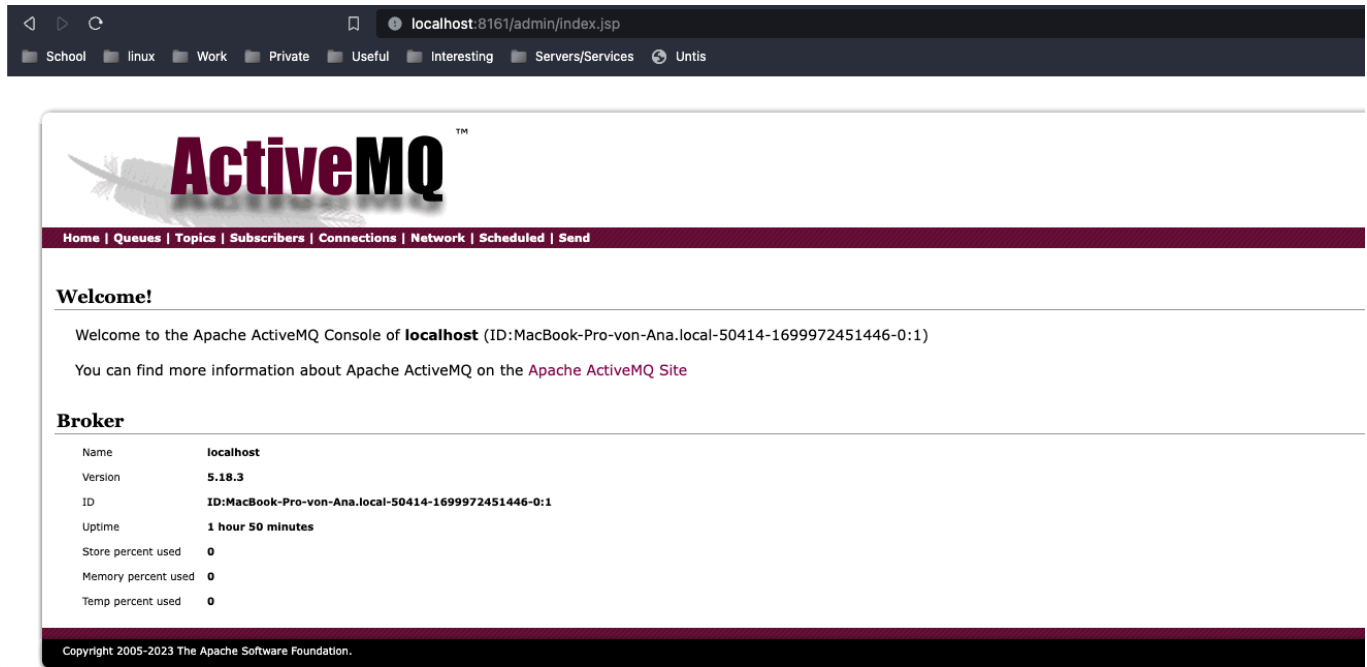
Da ich diese Übung auf einem MacOS Gerät, welches über `Brew` verfügt, durchgeführt habe, konnte ich ApacheMQ ganz einfach mit dem folgenden Befehl installieren:

```
brew install apache-activemq
```

Nach der Installation konnte in den Service direkt mit folgendem Befehl starten:

```
brew services start activemq
```

Damit läuft der Service!



## 2.2 MOMApplication.java

In dieser Klasse wird die eigentliche MOM Applikation gestartet. Dies findet auf einem gesonderten Port statt, da der Port 8080 von der API des Warehouses (Aufgabe 1) belegt ist.

Der Code ist der folgende:

```
@SpringBootApplication
public class MOMApplication {

    public static void main(String[] args) {
        // We need to run this app on another port because the API from the
        // warehouse is taking up port 8080
        SpringApplication app = new SpringApplication(MOMApplication.class);
        app.setDefaultProperties(Collections
            .singletonMap("server.port", "8081"));
        app.run(args);
    }
}
```

## 2.3 MOMController

In dieser Applikation wird das Schreiben sowie das Lesen in/aus der Message Queue von einem Zugriff auf `http://localhost:8081/warehouse/all` getriggert.

Dafür haben wir einen extra Controller aufgesetzt:

```

@RestController
public class MOMController {

    @CrossOrigin
    @RequestMapping(value = "/warehouse/all", produces =
MediaType.APPLICATION_JSON_VALUE)
    public String allWarehouseData()    {
        // send, read and return all messages from the queue
        return new MOMReceiver().getAllWarehouseData();
    }
}

```

## 2.4 MOMSender

Damit wir Daten in die Message Queue schicken können, benötigen wir einen Sender. In diesem werden die Daten der Wharehouse API (-> 1. Aufgabe) konsumiert, gespeichert und in die Queue gesendet.

Der Code ist der folgende:

```

private static String warehouseUUID = "469d7240-b974-441d-9562-
2c56a7b28767";
private static String warehouseAPIUrl = "http://localhost:8080/warehouse/" +
warehouseUUID + "/data";

private static String user = ActiveMQConnection.DEFAULT_USER;
private static String password = ActiveMQConnection.DEFAULT_PASSWORD;
private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;
private static String queueName = "warehouse-LINZ";

public MOMSender() {
    System.out.println("Sender started...");

    // create a connection to the apacheMQ broker
    Session session = null;
    Connection connection = null;
    MessageProducer producer = null;
    Destination destination = null;
    try {
        // init new connection
        ConnectionFactory connectionFactory = new
ActiveMQConnectionFactory(user, password, url);
        connection = connectionFactory.createConnection();
        connection.start();
    }
}

```

```

// Create the session
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
destination = session.createTopic(queueName);

// Create the producer
producer = session.createProducer(destination);
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

// Create the message
String currentWarehouseData = consumeWarehouseAPI();
TextMessage message =
session.createTextMessage(currentWarehouseData);
producer.send(message);
System.out.println(message.getText());

connection.stop();

} catch (Exception e) {
    System.out.println("[MessageProducer] Caught: " + e);
    e.printStackTrace();
} finally {
    try { producer.close(); } catch (Exception e) {}
    try { session.close(); } catch (Exception e) {}
    try { connection.close(); } catch (Exception e) {}
}
System.out.println("Sender finished.");
}

public static String consumeWarehouseAPI() {
    System.out.println("Consuming the warehouse API with the url " +
warehouseAPIUrl + "...");
    RestTemplate restTemplate = new RestTemplate();
    return restTemplate.getForObject(warehouseAPIUrl, String.class);
}

```

- Wie man hier sehen kann, verwenden wir hier keine klassische `Queue`, sondern ein `Topic`. Das heißt, dass der Receiver immer aktiv sein muss, sobald Daten in die MOM geschickt werden.

## 2.5 MOMReceiver

Diese Klasse ist die wichtigste - Sie beinhaltet das Senden sowie das Empfangen von Nachrichten aus der Queue.

Der Code ist hierbei der folgende:

```
private static String user = ActiveMQConnection.DEFAULT_USER;
private static String password = ActiveMQConnection.DEFAULT_PASSWORD;
private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;
private static String queueName = "warehouse-LINZ";

public String getAllWarehouseData() {
    System.out.println( "Receiver started." );

    // Create the connection.
    Session session = null;
    Connection connection = null;
    MessageConsumer consumer = null;
    Destination destination = null;
    StringBuilder receivedMessages = null;

    try {
        ConnectionFactory connectionFactory = new
ActiveMQConnectionFactory(user, password, url);
        connection = connectionFactory.createConnection();
        connection.start();

        // Create the session
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        destination = session.createTopic(queueName);

        // Create the consumer
        consumer = session.createConsumer(destination);

        // Start receiving
        receivedMessages = new StringBuilder();

        // Let's send the warehouse data here
        new MOMSender();

        TextMessage message = (TextMessage) consumer.receive(1000);
        while ( message != null ) {
            receivedMessages.append(message.getText());
            message.acknowledge();
            message = (TextMessage) consumer.receive(1000);
        }
        connection.stop();

    } catch (Exception e) {
        System.out.println("[MessageConsumer] Caught: " + e);
    }
}
```

```

        e.printStackTrace();

    } finally {
        try { consumer.close(); } catch ( Exception e ) {}
        try { session.close(); } catch ( Exception e ) {}
        try { connection.close(); } catch ( Exception e ) {}
    }

    System.out.println( "Receiver finished." );
    System.out.println(receivedMessages.toString());
    return receivedMessages.toString();
}

```

Hierbei ist besonders die folgende Codezeile wichtig:

```

// Let's send the warehouse data here
new MOMSender();

```

Wenn wir uns erinnern: Wir verwenden hier keine `Queues`, sondern `Topics`. `Topics` sind quasi Abos, welche Clients machen können.

Ein Client kann also das `warehouse-LINZ` Topic abonnieren und bekommt dann direkt, nachdem eine Nachricht eingegangen ist, diese zugeschickt.

Das bedeutet aber auch, dass die Nachricht an das Topic gesendet werden muss, wenn auch der Receiver aktiv und bereit ist.

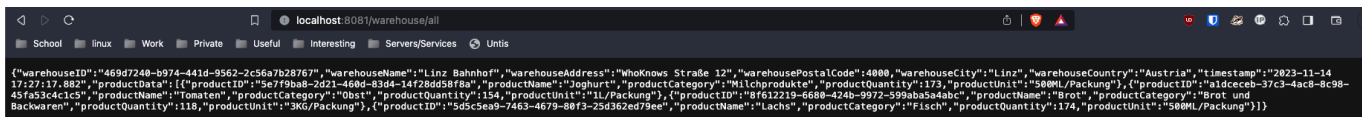
Wenn jetzt nur der Sender die Nachricht in das Topic sendet, der Receiver aber nicht aktiv ist, wird das Topic leer sein (diese Erkenntnis habe ich voller Schmerz gemacht).

### 3. Testen des Programmes

Wir starten das Programm einmal und navigieren zur URL, welche den Sender sowie Receiver triggert:

```
http://localhost:8081/warehouse/all
```

Wir werden mit folgendem Ergebnis begrüßt; den Daten des Linzer Warenhauses:



```

{"warehouseID":"469d7240-b974-441d-9562-2c56a7b28767","warehouseName":"Linz Bahnhof","warehouseAddress":"WhoKnows Straße 12","warehousePostalCode":4000,"warehouseCity":"Linz","warehouseCountry":"Austria","timestamp":"2023-11-14 17:27:17.882","productData":[{"productID":"5e7f9ba8-2d21-460d-83d4-14f28dd58f8a","productName":"Joghurt","productCategory":"Milchprodukte","productQuantity":173,"productUnit":"500ML/Packung"}, {"productID":"a1dceeb-37c3-4ac8-8c98-45fa34c4c1c5","productName":"Tomaten","productCategory":"Obst","productQuantity":154,"productUnit":"11/Packung"}, {"productID":"8f632215-6688-424b-9972-599ab55d4abc","productName":"Brot","productCategory":"Brot und Backwaren","productQuantity":118,"productUnit":"3KG/Packung"}, {"productID":"5d5c5ea9-7463-4679-88f3-25d362ed79ee","productName":"Lachs","productCategory":"Fisch","productQuantity":174,"productUnit":"500ML/Packung"}]}

```