



L'api JMS

TP : Implémentation d'une communication asynchrone avec JMS et Apache ActiveMQ

Table des matières

I.	Objectif du TP.....	3
II.	Prérequis	3
III.	L'api JMS.....	3
a.	C'est quoi JMS ?	3
b.	Historique des versions JMS.....	4
c.	Le Protocole point-à-point (Queue)	5
d.	Le Protocole publish/subscriber (Topic)	6
e.	Les classes principales de JMS.....	7
1.	L'interface jakarta.jms.ConnectionFactory	7
2.	L'interface jakarta.jms.Connection.....	8
3.	L'interface jakarta.jms.Session	8
4.	L'interface jakarta.jms.MessageProducer	9
5.	L'interface jakarta.jms.MessageConsumer	9
6.	La structure du message JMS.....	9
7.	Le modèle de programmation avec JMS.....	10
f.	Les MOM (Message-Oriented Middleware) JMS.....	12
g.	Les protocoles de communication distribuée asynchrone.....	13
1.	Le protocole AMQP (Advanced Message Queuing Protocol)	14
2.	Le protocole MQTT : Message Queuing Telemetry Transport.....	16
3.	Le protocole STOMP : Simple/Streaming Text Oriented Messaging Protocol	17
IV.	Développement d'une application avec JMS, Apache ActiveMQ et Queue.....	18
a.	Installer Apache ActiveMQ.....	18
b.	Créer un projet Maven	19
c.	Le fichier pom.xml	19
d.	La classe JmsQueueProducer	20
e.	La classe JmsQueueConsumer.....	21

f. La classe Main	22
V. Développement d'une application avec JMS, Apache ActiveMQ et Topic.....	25
a. Développement du « Producer »	25
b. Développement du « Consumer »	28
c. Tester l'application	31
Conclusion	Erreur ! Signet non défini.

I. Objectif du TP

- ✓ Expliquer l'api JMS (Java Message Service).
- ✓ Expliquer les services fournis par les MOM (Message-Oriented Middleware).
- ✓ Expliquer le protocole AMQP (Advanced Message Queuing Protocol).
- ✓ Expliquer le protocole MQTT (Message Queuing Telemetry Transport).
- ✓ Expliquer le protocole STOMP (Streaming Text Oriented Messaging Protocol).
- ✓ Développer une 1^{ière} application avec JMS et Apache ActiveMQ en utilisant le modèle Queue.
- ✓ Développer une 2^{ème} application avec JMS et Apache ActiveMQ en utilisant le modèle Topic.

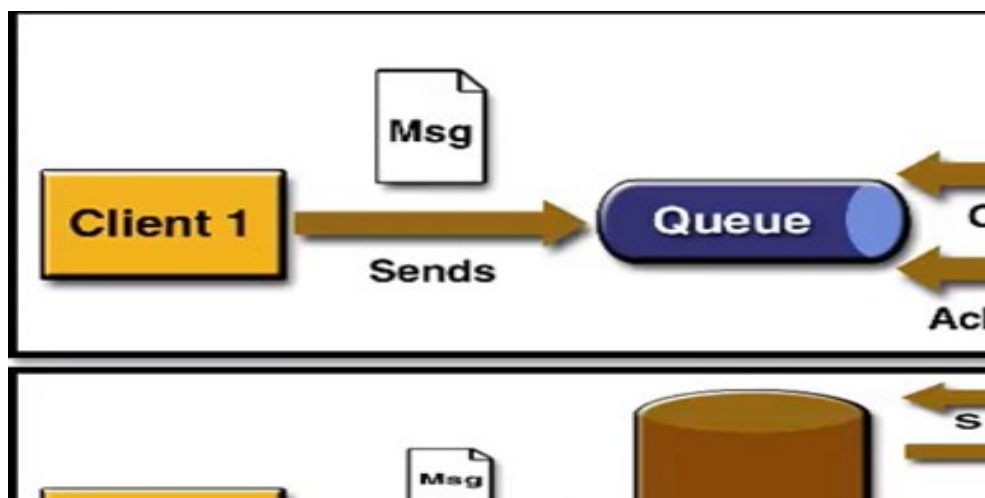
II. Prérequis

- IntelliJ IDEA ;
- JDK version 17 ;
- Une connexion Internet pour permettre à Maven de télécharger les librairies.

NB : Ce TP a été réalisé avec IntelliJ IDEA 2023.2.3 (Ultimate Edition).

III. L'api JMS

a. C'est quoi JMS ?



- **JMS**, acronyme de **Java Message Service**, est une API pour permettre un dialogue standard entre des applications ou des composants grâce à des brokers de messages ou MOM (Message-Oriented Middleware). Elle permet donc d'utiliser des services de messaging dans des applications Java comme le fait l'API JDBC pour les bases de données.
- La page officielle de JMS est à l'URL : <https://www.oracle.com/java/technologies/java-message-service.html>.
- Les brokers de messages ou MOM (Message-Oriented Middleware) permettent d'assurer l'échange de messages entre deux composants nommés clients. Ces échanges peuvent se faire dans un contexte interne (pour l'EAI) ou un contexte externe (pour le B2B).

- Les deux clients n'échangent pas directement des messages : un client envoie un message et le client destinataire doit demander la réception du message. Le transfert du message et sa persistance sont assurés par le broker.
- Les échanges de message sont :
 - ✓ Asynchrones :
 - ✓ Fiabiles : les messages ne sont délivrés qu'une et une seule fois
- Les MOM représentent le seul moyen d'effectuer un échange de messages asynchrones. Ils peuvent aussi être très pratiques pour l'échange synchrone de messages plutôt que d'utiliser d'autres mécanismes plus compliqués à mettre en œuvre (sockets, RMI, CORBA ...).
- Les MOM peuvent fonctionner selon deux modes :
 - ✓ Queue : Point to Point.
 - ✓ Topic : Publish/Subscribe
- Le mode Queue (point to point) repose sur le concept de files d'attente (queues). Le message est stocké dans une file d'attente puis il est lu dans cette file ou dans une autre. Le transfert du message d'une file à l'autre est réalisé par le MOM.
- Chaque message est envoyé dans une seule file d'attente. Il y reste jusqu'à ce qu'il soit consommé par un client et un seul. Le client peut le consommer ultérieurement : la persistance est assurée par le MOM.
- Le mode Publish/Subscribe repose sur le concept de sujets (Topics). Plusieurs clients peuvent envoyer des messages dans ce topic. Le MOM assure l'acheminement de ce message à chaque client qui doit être abonné à ce topic. Le message possède donc potentiellement plusieurs destinataires. L'émetteur du message ne connaît pas les destinataires qui se sont abonnés.

b. Historique des versions JMS

Version	Data de publication	Principales fonctionnalités
1.0	1998	*Prise en charge des modèles Queue et Topic. *Modes d'accusé de réception (AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, etc.). *Support pour les types de messages (TextMessage, ObjectMessage, BytesMessage, etc.).
1.1	2002	*Une seule API pour les deux modèles (simplifiant le développement). Sessions transitoires (Session.TRANSACTIONAL) pour gérer les transactions locales. *Meilleur support pour les brokers tiers et l'intégration avec Java EE. Devenue la norme pour les MOM.
2.0	2013	*Réduction de la verbosité (par exemple, les classes JMSContext,

		<p>JMSProducer, JMSConsumer).</p> <p>*API orientée annotations : Simplification de l'intégration avec les applications CDI (Context and Dependency Injection).</p> <p>*Delivery delay : Ajout d'une fonctionnalité permettant de différer la livraison des messages.</p> <p>*Gestion des exceptions : Plus d'options pour la gestion des erreurs.</p>
--	--	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

c. Le Protocole point-à-point (Queue)



- Le modèle **point à point** est basé sur une **file d'attente** (Queue) où les producteurs (envoyeurs de messages) envoient des messages à une file d'attente et où un consommateur (récepteur de messages) reçoit les messages de cette même file. Voici les caractéristiques du modèle **point à point** (P2P) utilisant une **Queue** :
 - **Un seul consommateur pour chaque message** : Un message envoyé à une queue ne sera consommé que par un seul consommateur. Une fois qu'un message est consommé, il n'est plus disponible pour d'autres consommateurs.
 - **Séquencement des messages** : Les messages sont consommés dans l'ordre dans lequel ils ont été envoyés à la queue, bien qu'il n'y ait pas de garantie stricte de livraison ordonnée dans tous les cas, en particulier en cas de pannes ou de problèmes de réseau.
 - **Durabilité** : Une queue peut être configurée pour assurer la durabilité des messages, ce qui signifie que les messages sont persistés en cas de panne du serveur.
- Dans JMS, une **Queue** représente une file d'attente où les messages sont stockés jusqu'à ce qu'ils soient consommés. Elle est définie par l'interface **jakarta.jms.Queue**.
- Ci-après, les composants principaux de JMS :
 - **Producteur (Producer)** : Il envoie des messages vers la Queue.
 - **Consommateur (Consumer)** : Il reçoit des messages de la Queue.
 - **ConnectionFactory** : Fournit une connexion à un MOM (par exemple, ActiveMQ, RabbitMQ).
 - **Destination** : C'est une abstraction qui représente la source ou la cible des messages (une **Queue** ou un **Topic**).
 - **Message** : Contenu de la communication, qui peut être textuel, binaire ou sous forme d'objet.
- Voici un aperçu du cycle de vie typique des messages dans une **Queue** JMS :

1. Un **producteur** envoie un message à une Queue.
2. Le message est stocké dans la Queue jusqu'à ce qu'un consommateur le récupère.
3. Le **consommateur** reçoit le message et le traite.
4. Une fois que le consommateur reçoit le message, celui-ci est supprimé de la Queue.

d. Le Protocole publish/subscribe (Topic)



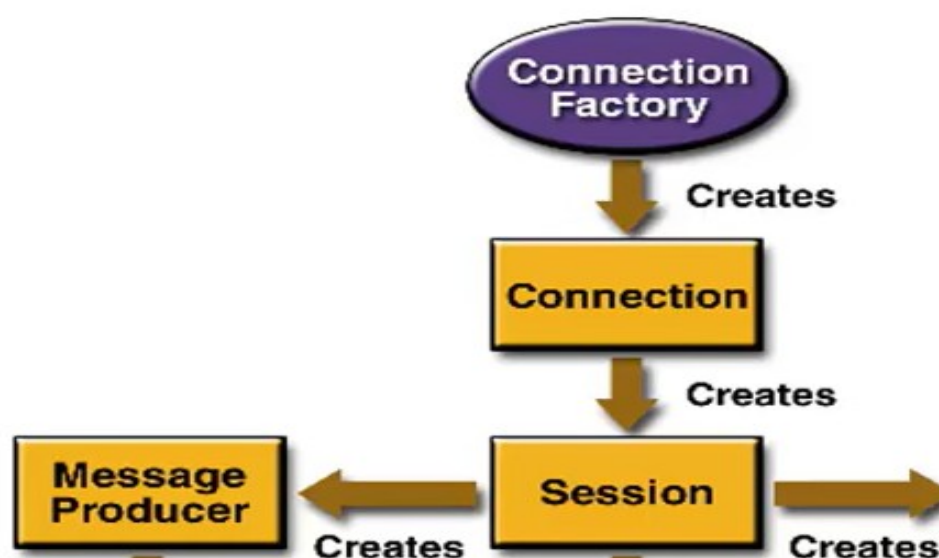
- Le modèle **publish/subscribe** (ou **pub/sub**) dans JMS fonctionne différemment du modèle **point à point** utilisé avec les **Queues**. Dans le modèle **pub/sub**, un **producteur** (ou **publisher**) envoie un message à un **Topic**, et tous les **consommateurs** (ou **subscribers**) abonnés à ce Topic recevront ce message. Plusieurs abonnés peuvent recevoir le même message simultanément, contrairement à une **Queue** où un seul consommateur reçoit chaque message.
- Ci-après, les caractéristiques d'un Topic :
 - **Multiple abonnés** : Les messages envoyés à un **Topic** sont reçus par tous les abonnés inscrits à ce **Topic**. Cela permet à plusieurs consommateurs de recevoir le même message.
 - **Durabilité** : Les messages dans un **Topic** peuvent être durables, ce qui signifie qu'un abonné qui est momentanément hors ligne peut toujours recevoir les messages envoyés pendant son absence une fois qu'il se reconnecte (si le **Topic** est configuré en mode durable).
 - **Un seul producteur, plusieurs consommateurs** : Un seul producteur peut envoyer un message à un Topic, mais tous les consommateurs inscrits à ce Topic recevront ce message.
- Dans JMS, un **Topic** est une destination de message qui est utilisée pour envoyer des messages dans le modèle **publish/subscribe**. Un **Topic** est défini par l'interface **jakarta.jms.Topic**. Ci-après, les composants principaux :
 - **Producteur (Publisher)** : L'application qui envoie des messages à un Topic.

- **Consommateur (Subscriber)** : L'application qui s'abonne à un Topic et reçoit les messages envoyés à ce Topic.
- **ConnectionFactory** : Fournit une connexion au serveur de messagerie (par exemple, ActiveMQ, RabbitMQ).
- **Destination** : Représente soit une Queue (modèle **point à point**) soit un Topic (modèle **publish/subscribe**).
- **Message** : Le contenu envoyé entre le producteur et le consommateur. Cela peut être un texte, un objet ou un message binaire.

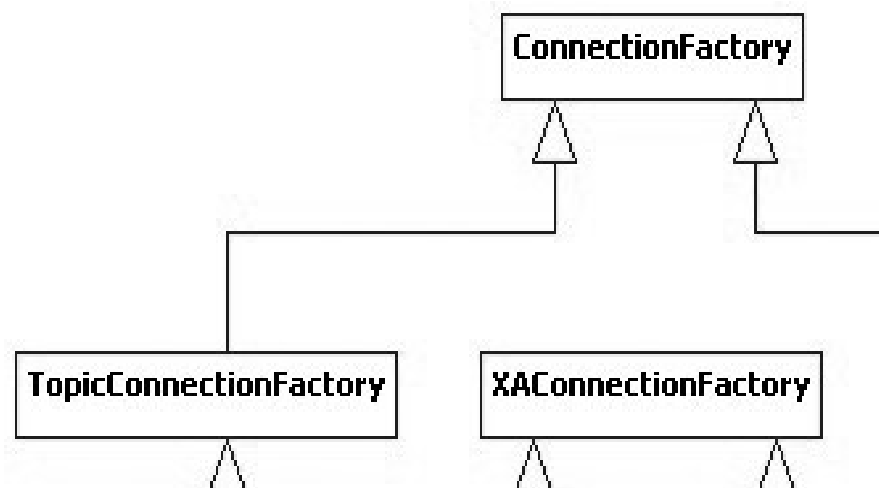
▪ Voici le cycle de vie des messages dans un **Topic** JMS :

1. Un **producteur** (publisher) envoie un message à un **Topic**.
2. Tous les **consommateurs** (subscribers) abonnés à ce **Topic** reçoivent le message.
3. Les abonnés peuvent traiter le message de manière indépendante.
4. Les messages peuvent être persistés dans un **Topic** durable, afin qu'ils soient reçus par les abonnés même si ces derniers étaient hors ligne au moment où le message a été envoyé.

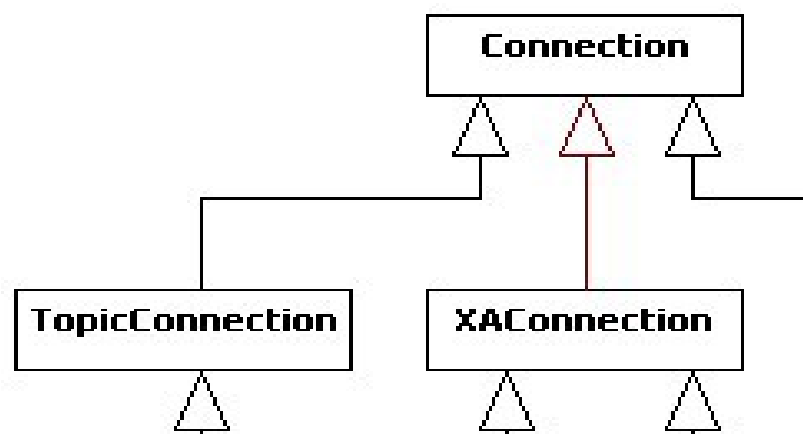
e. Les classes principales de JMS



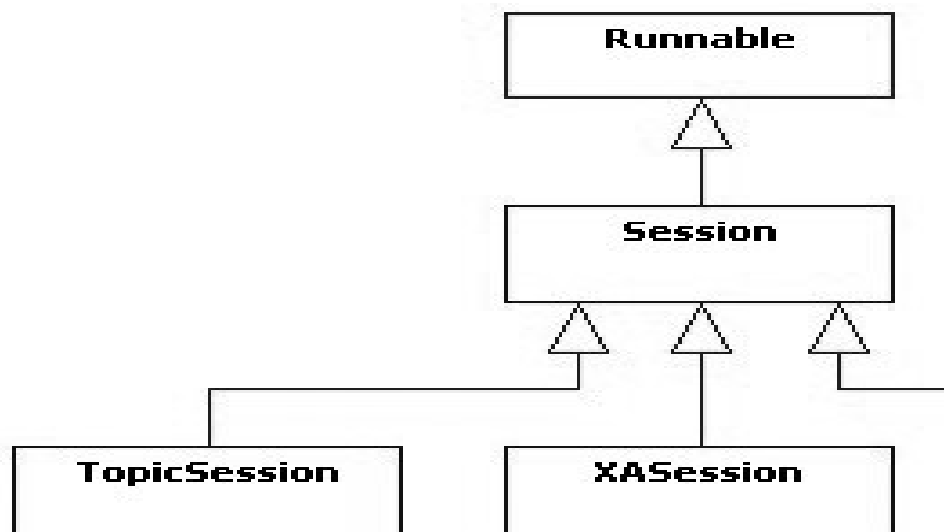
1. L'interface `jakarta.jms.ConnectionFactory`



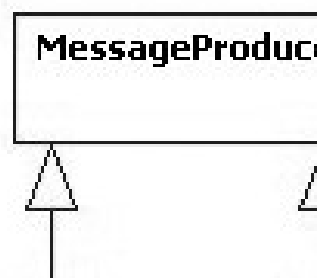
2. L'interface jakarta.jms.Connection



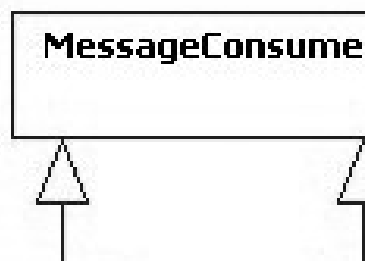
3. L'interface jakarta.jms.Session



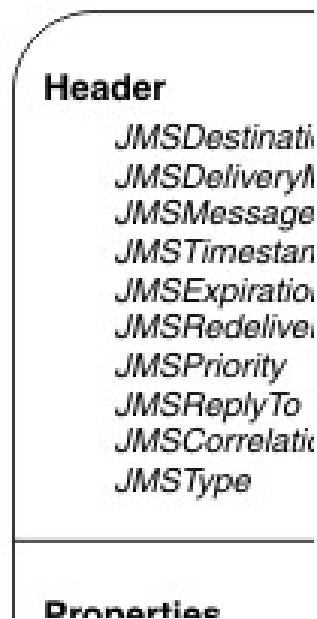
4. L'interface jakarta.jms.MessageProducer



5. L'interface jakarta.jms.MessageConsumer



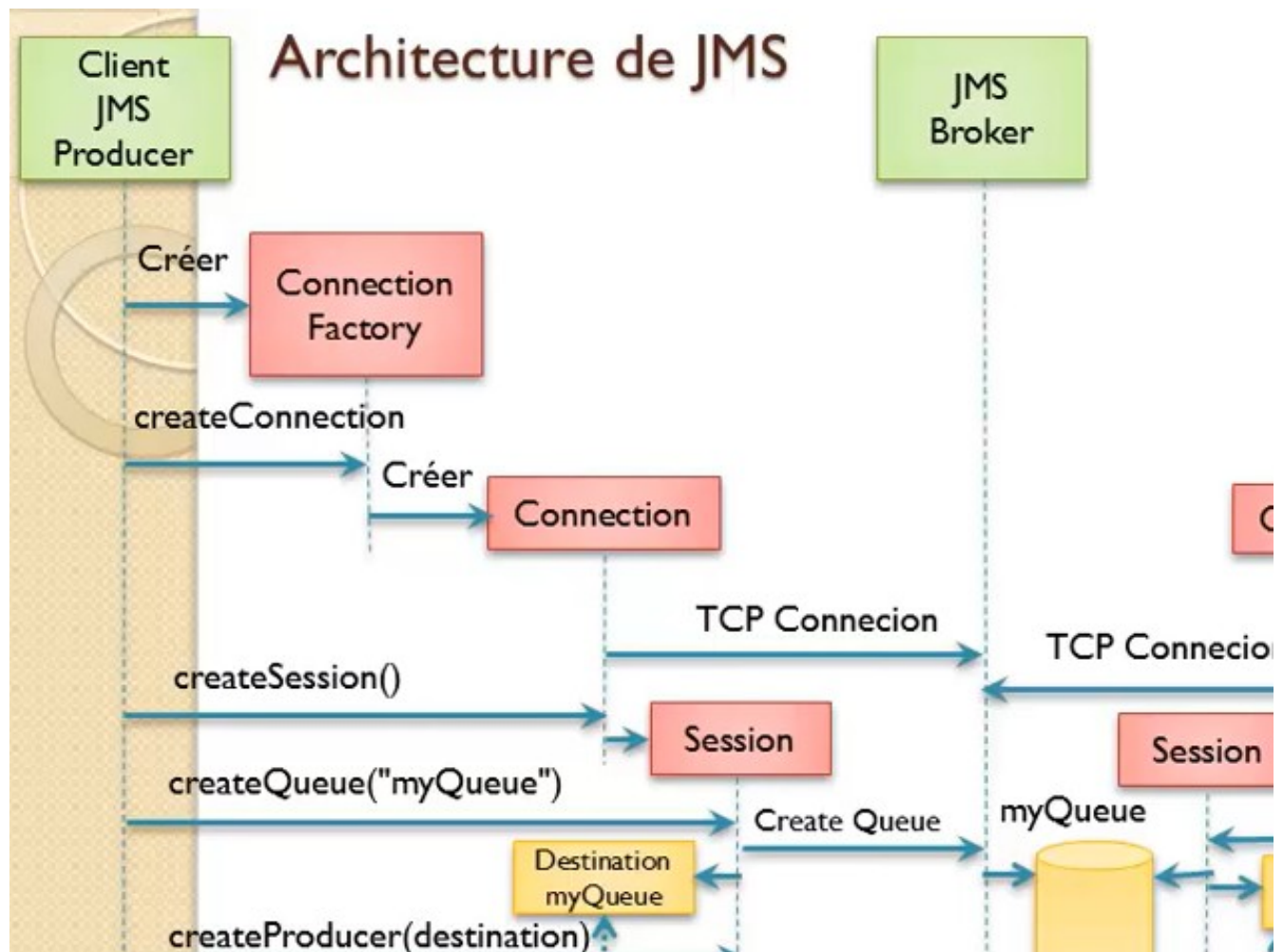
6. La structure du message JMS



- Les messages sont encapsulés dans un objet de type ***jakarta.jms.Message*** : ils doivent obligatoirement implémenter l'interface **Message** ou l'une de ses sous-classes.
- Un message est constitué de trois parties :
 - L'en-tête (header) : contient des données techniques
 - Les propriétés (properties) : contient des données fonctionnelles
 - Le corps du message (body) : contient les données du message
- L'interface **Session** propose plusieurs méthodes ***createXXXMessage()*** pour créer des messages contenant des données au format **XXX**.
- Il existe aussi pour chaque format des interfaces filles de l'interface **Message** :

Interface	Type de contenu	Cas d'usage
<code>TextMessage</code>	Texte (plain text, JSON)	Messages légers et
<code>BytesMessage</code>	Données binaires	Fichiers, flux audio
<code>MapMessage</code>	Clés-valeurs	Données structurées

7. Le modèle de programmation avec JMS



Exemple de code pour un Producer :

```

// Créer une connexion à ActiveMQ
ActiveMQConnectionFactory connectionFactory = new ActiveMQ
Connection connection = connectionFactory.createConnection()
connection.start();

// Créer une session et une queue
Session session = connection.createSession(false, Session
Destination destination = session.createQueue(destinationName);

// Créer un producteur pour envoyer un message
MessageProducer producer = session.createProducer(destinationName);

// Créer un message et l'envoyer

```

Exemple de code pour un Consumer :

```

ActiveMQConnectionFactory connectionFactory = new ActiveMQ
Connection connection = connectionFactory.createConnection()
connection.start();

// Créer une session et une queue
Session session = connection.createSession(false, Session
Destination destination = session.createQueue(destinationName);

// Créer un consommateur pour recevoir le message
MessageConsumer consumer = session.createConsumer(destinationName);

// Attendre un message et le traiter
Message message = consumer.receive(1); // Attente de 1 seconde

if (message instanceof TextMessage textMessage) {
    System.out.println("Message reçu : " + textMessage.getText());
}

```

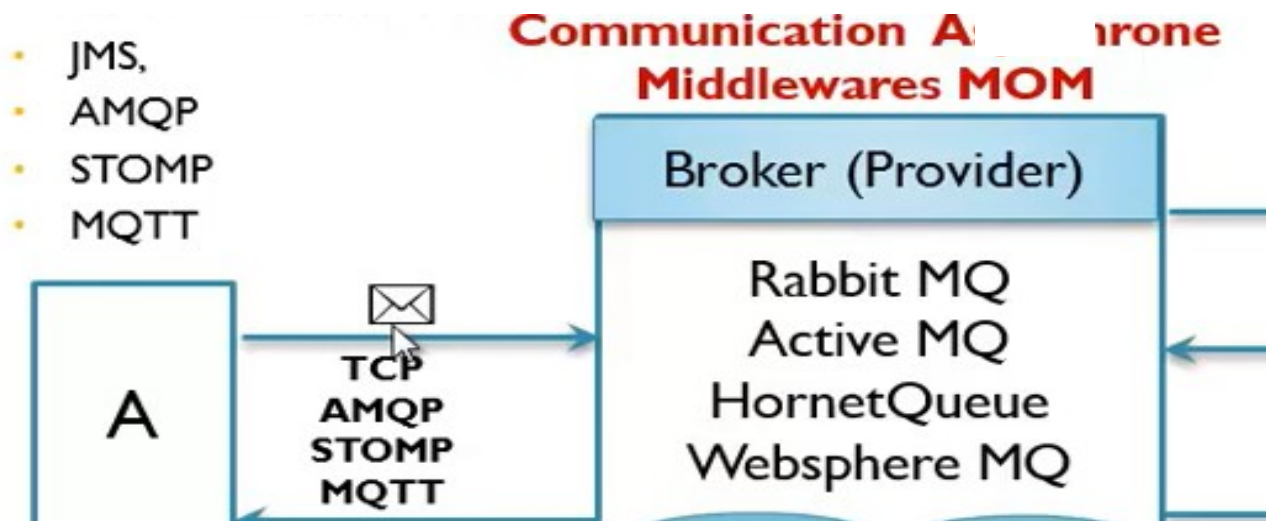
f. Les MOM (Message-Oriented Middleware) JMS

- Un **Message-Oriented Middleware** (MOM) est une plateforme logicielle qui facilite l'échange de messages entre différentes applications ou systèmes, souvent dans des environnements distribués. Les MOM sont utilisés pour gérer la communication asynchrone, garantir la fiabilité et permettre la mise à l'échelle.

- Les principaux rôles des MOM dans le cadre de JMS sont les suivants :
 - ✓ **Gestion des messages :**
 - Les MOM assurent le routage, la livraison et le stockage des messages.
 - Ils garantissent que les messages sont correctement envoyés et reçus, même en cas de défaillance.
 - ✓ **Modèles de communication :** les MOM implémentent les deux principaux modèles de JMS :
 - Point-to-Point (P2P) : Basé sur des files d'attente (Queues).
 - Publish/Subscribe (Pub/Sub) : Basé sur des sujets (Topics).
 - ✓ **Fiabilité et persistance :** les MOM offrent des mécanismes pour assurer la persistance des messages afin qu'ils ne soient pas perdus en cas de panne.
 - ✓ **Découplage :** Grâce aux MOM, les applications productrices (producers) et consommatrices (consumers) sont découplées. Elles n'ont pas besoin de connaître leur existence mutuelle ni d'être en ligne en même temps.
 - ✓ **Interopérabilité :** Les MOM permettent à des applications écrites dans différents langages ou utilisant des technologies variées de communiquer entre elles.
- Exemples de MOM compatibles avec JMS :

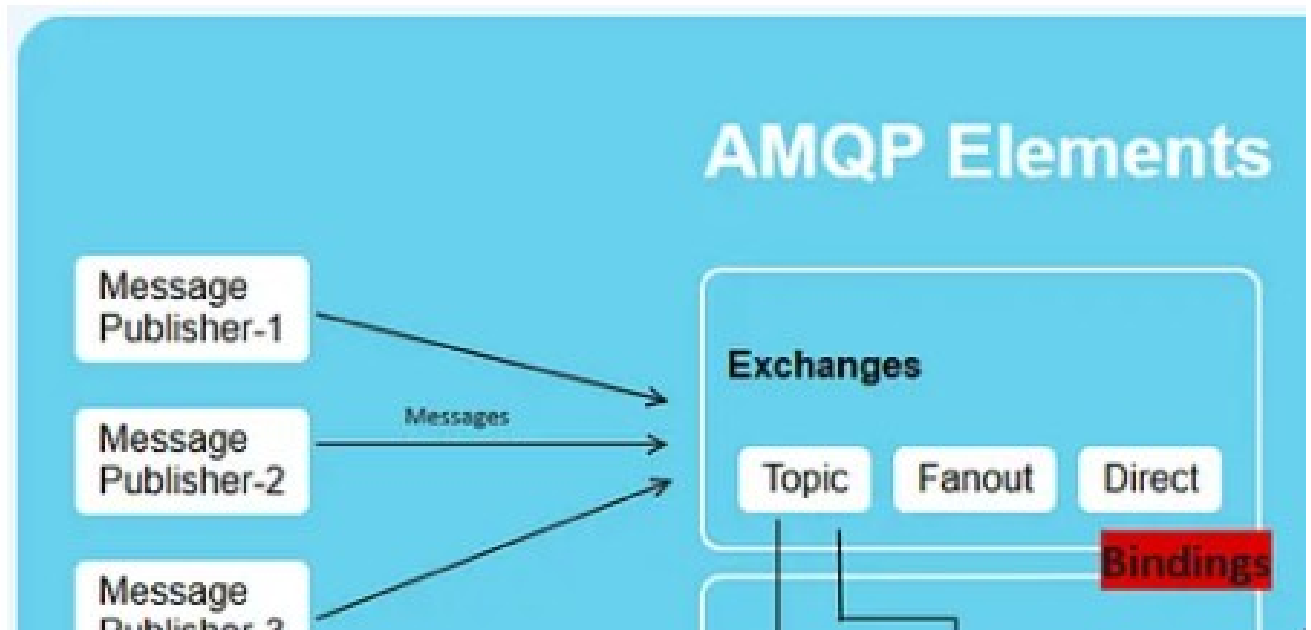
S.N.	JMS Provider Software	
1	WebSphere MQ	IBM
2	Weblogic Messaging	Oracle
3	Active MQ	Apache Foundation
4	Rabbit MQ	Rabbit Technologies (o
5	HornetQ	JBoss
6	Sonic MQ	Progress Software

g. Les protocoles de communication distribuée asynchrone



- Les protocoles de communication ne font pas directement partie de JMS, mais JMS utilise des protocoles de communication sous-jacents pour acheminer les messages entre les clients et les MOM.
- Les protocoles de communication sont gérés par les MOM. A titre d'exemple, Apache ActiveMQ supporte les protocoles suivants :
 - ✓ AMQP : Advanced Message Queuing Protocol.
 - ✓ MQTT : Message Queuing Telemetry Transport
 - ✓ STOMP : Streaming Text Oriented Messaging Protocol
 - ✓ OpenWire
 - ✓ REST
 - ✓ RSS and Atom
 - ✓ WSIF
 - ✓ WS Notification
 - ✓ XMPP

1. Le protocole AMQP (Advanced Message Queuing Protocol)



- L'architecture d'AMQP (Advanced Message Queuing Protocol) est conçue pour permettre une communication fiable, évolutive et interopérable entre différentes applications dans des environnements distribués. Les composants principaux de l'architecture AMQP :
 1. Producers (Producteurs)
 2. Brokers (Intermédiaires de message)
 3. Queues (Files d'attente)
 4. Consumers (Consommateurs)
 5. Exchanges (Échanges)
 6. Bindings (Liens)
- Les échanges sont responsables de router les messages provenant des producteurs vers une ou plusieurs queues selon des règles spécifiques définies dans des **bindings**. Les types d'échanges :
 1. **Direct Exchange** : Route les messages vers une queue ayant une clé de routage exacte.
 2. **Fanout Exchange** : Diffuse les messages à toutes les queues liées.
 3. **Topic Exchange** : Route les messages en fonction de modèles basés sur des mots-clés.
 4. **Headers Exchange** : Route les messages en fonction des en-têtes, plutôt que des clés de routage.
- Les bindings définissent les relations entre les Exchanges et les Queues. Ils permettent de configurer les règles de routage des messages.
- La structure d'un message AMQP :

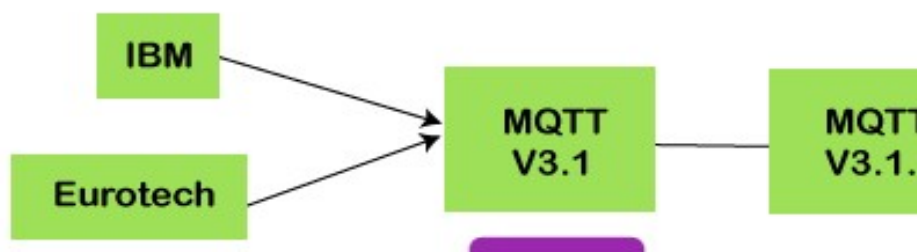


Ci-après un exemple d'un message AMQP :

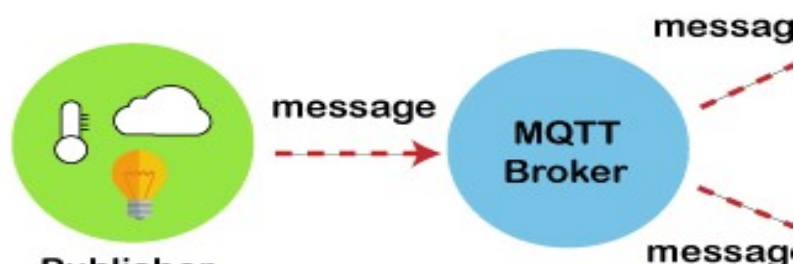


2. Le protocole MQTT : Message Queuing Telemetry Transport

- Le protocole **MQTT** (Message Queuing Telemetry Transport) est un protocole de messagerie léger et ouvert, conçu pour permettre une communication efficace et fiable dans des environnements où la bande passante est limitée, où la latence doit être minimale, ou où les dispositifs ont des ressources limitées (comme les capteurs ou les appareils IoT).
- Histoire de MQTT :



- L'architecture de MQTT :



- La structure du message MQTT :

MQTT Packet Structure

Fixed Header, Present in all MQTT Packet

Variable Header, Present in Some MQTT Packets

3. Le protocole STOMP : Simple/Streaming Text Oriented Messaging Protocol

- STOMP (Simple/Streaming Text Oriented Messaging Protocol) est un protocole simple basé sur du texte qui permet la communication entre clients et serveurs dans un système de messagerie. Il est largement utilisé pour échanger des messages dans des architectures distribuées ou orientées événements, et il est souvent utilisé avec les Brokers comme ActiveMQ, RabbitMQ, ou Apache Kafka.
- Les messages STOMP sont basés sur un format texte simple, ce qui facilite leur compréhension et leur débogage. Les interactions se font par l'envoi de frames (trames) textuelles structurées.
- Chaque trame contient trois parties principales :
 - ✓ Commande (CONNECT, SEND, SUBSCRIBE, etc.)
 - ✓ En-têtes (clé-valeur pour transmettre des métadonnées)
 - ✓ Corps du message (contenu principal)

- Exemple de trame STOMP :

```
SEND
destination:/queue
Hello, World!
```

Le caractère ^@ représente la fin de la trame.

- Les commandes courantes utilisées dans STOMP :
 - ✓ **CONNECT** : Pour initier une connexion entre un client et un serveur.
 - ✓ **SEND** : Pour envoyer un message à une destination.
 - ✓ **SUBSCRIBE** : Pour s'abonner à une destination et recevoir des messages.
 - ✓ **UNSUBSCRIBE** : Pour se désabonner d'une destination.
 - ✓ **ACK/NACK** : Pour accuser réception ou rejeter un message.
 - ✓ **DISCONNECT** : Pour terminer une connexion.
- STOMP est indépendant du langage de programmation. Il existe des bibliothèques STOMP pour de nombreux langages, comme Java, Python, JavaScript, et C++.

- Sinon, STOMP a quelques limites :
 - ✓ Manque de robustesse : Comparé à AMQP ou MQTT, STOMP offre moins de fonctionnalités avancées (comme les propriétés complexes des messages ou les garanties de livraison).
 - ✓ Consommation de bande passante : Son format textuel peut être moins efficace que des protocoles binaires comme AMQP ou MQTT.

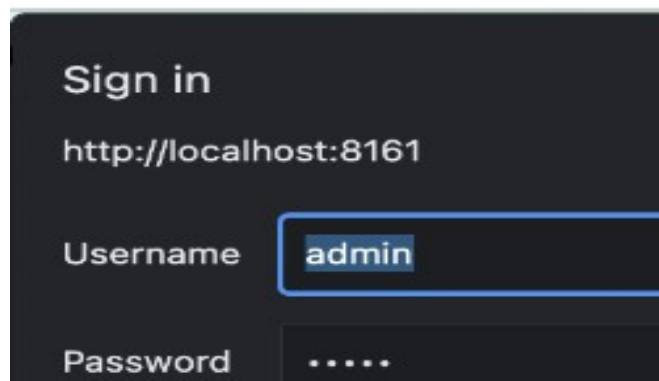
IV. Développement d'une application avec JMS, Apache ActiveMQ et Queue

a. Installer Apache ActiveMQ

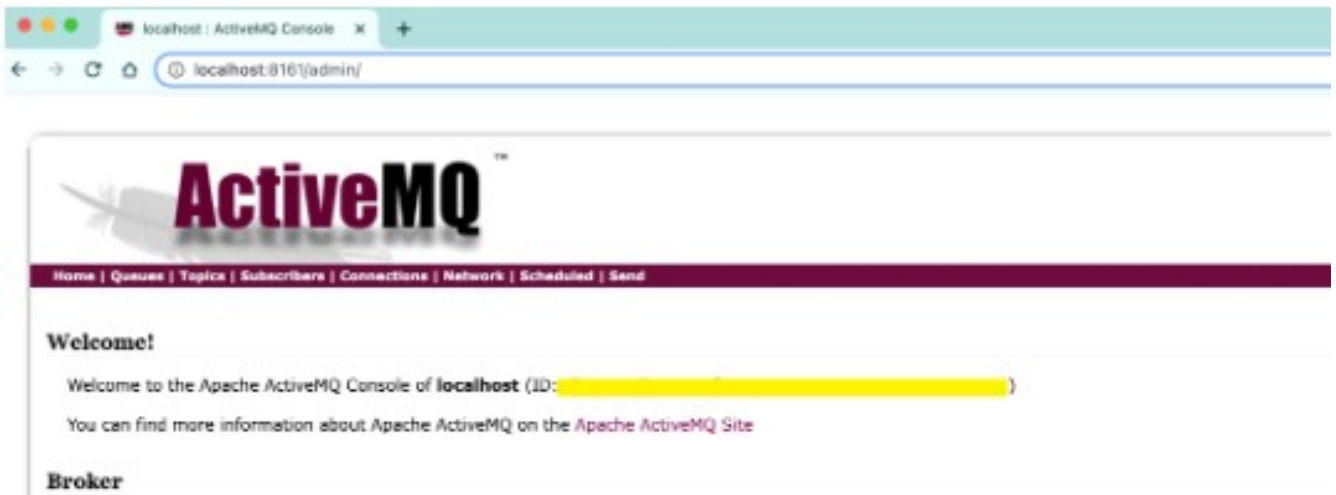
- Apache ActiveMQ peut être téléchargé à partir du site web d'Apache et est disponible sous forme de fichier zip. Le lien pour la version (6.1.4) sous Windows est :
 - <https://www.apache.org/dyn/closer.cgi?filename=/activemq/6.1.4/apache-activemq-6.1.4-bin.zip&action=download>
- Une fois que vous avez téléchargé et extrait le fichier, vous pouvez démarrer le serveur en exécutant la commande à partir du répertoire bin :

```
activemq start
```

- Dans le navigateur, ouvrez l'URL <http://localhost:8161/admin>.
- Entrer le compte (admin/admin) pour se connecter.

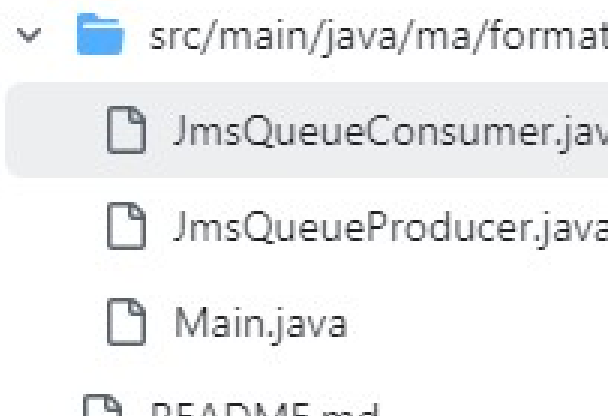


L'écran ci-dessous s'affichera :



b. Créer un projet Maven

- Créer un projet Maven, par exemple : jms-activemq-queue-example.
- L'arborescence du projet est la suivante :



c. Le fichier pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>ma.formations.jms</groupId>
  <artifactId>jms-activemq-queue-example</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
```

```

        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-all</artifactId>
        <version>6.1.4</version>
    </dependency>

    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>2.24.2</version>
    </dependency>

    <!-- Log4j Core -->
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.24.2</version>
    </dependency>

</dependencies>

</project>

```

d. La classe JmsQueueProducer

```

package ma.formationen.jms;

import jakarta.jms.*;
import org.apache.activemq.ActiveMQConnectionFactory;

public class JmsQueueProducer {
    private final String destinationName;
    private final String brokerUrl;

    public JmsQueueProducer(String destinationName, String brokerUrl) {
        this.destinationName = destinationName;
        this.brokerUrl = brokerUrl;
    }

    public void send(String message) throws JMSException {
        // Créer une connexion à ActiveMQ
        ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerUrl);
        Connection connection = connectionFactory.createConnection();
        connection.start();

        // Créer une session et une queue
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Destination destination = session.createQueue(destinationName);

        // Créer un producteur pour envoyer un message
        MessageProducer producer = session.createProducer(destination);

        // Créer un message et l'envoyer
        TextMessage messageObject = session.createTextMessage(message);
        producer.send(messageObject);
    }
}

```

```

        System.out.println("Message envoyé : " + messageObject.getText());
        // Fermer la connexion
        producer.close();
        session.close();
        connection.close();
    }
}

```



- ✓ **Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE)**

L'option **AUTO_ACKNOWLEDGE** est un mode d'accusé de réception des messages. Elle est utilisée pour indiquer que le client JMS accepte automatiquement les messages une fois qu'ils sont reçus et traités avec succès par le Consumer.

- ✓ Cette configuration présente des limitations. En effet, si l'application ou le consommateur plante après la réception d'un message mais avant de terminer son traitement, le message sera considéré comme consommé et ne sera pas redélivré par le MOM.
- ✓ Ci-après, les autres alternatives fournies par JMS :
 1. **CLIENT_ACKNOWLEDGE** : Le client appelle explicitement la méthode **message.acknowledge()** après avoir traité un message.
 2. **DUPS_OK_ACKNOWLEDGE** : Permet au MOM de redélivrer des messages en cas de doute, avec une éventuelle duplication.
 3. **TRANSACTIONAL** : Utilisé dans des sessions transactionnelles, où les messages sont accusés ou annulés en lot.

e. La classe JmsQueueConsumer

```

package ma.formation.jms;

import jakarta.jms.*;
import org.apache.activemq.ActiveMQConnectionFactory;

public class JmsQueueConsumer {
    private final String destinationName;
    private final String brokerUrl;

    public JmsQueueConsumer(String destinationName, String brokerUrl) {
        this.destinationName = destinationName;
        this.brokerUrl = brokerUrl;
    }

    public void receive() throws JMSException {
        ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerUrl);
        Connection connection = connectionFactory.createConnection();
        connection.start();
    }
}

```

```

// Créer une session et une queue
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Destination destination = session.createQueue(destinationName);

// Créer un consommateur pour recevoir le message
MessageConsumer consumer = session.createConsumer(destination);

// Attendre un message et le traiter
Message message = consumer.receive(1); // Attente de 1 seconde

if (message instanceof TextMessage textMessage) {
    System.out.println("Message reçu : " + textMessage.getText());
} else {
    System.out.println("Aucun message reçu.");
}
// Fermer la connexion
consumer.close();
session.close();
connection.close();
}
}

```

f. La classe Main

```

package ma.formationen.jms;

import jakarta.jms.JMSEException;

public class Main {
    public static void main(String[] args) {
        String brokerUrl = "tcp://localhost:61616";
        String destinationName = "queue1";
        JmsQueueProducer producer = new JmsQueueProducer(destinationName, brokerUrl);
        JmsQueueConsumer consumer = new JmsQueueConsumer(destinationName, brokerUrl);
        try {
            producer.send("premier msg");
            consumer.receive();
            producer.send("deuxième msg");
            consumer.receive();
            producer.send("....");
            consumer.receive();
            producer.send("other message");

        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }
}

```

- Exécuter la méthode main et observer les résultats suivants :

Message envoyé : première

Message reçu : premier

Message envoyé : deuxième

Message reçu : deuxième

- Vue que le dernier message « *other message* » n'a été consommé par aucun consommateur, ce dernier reste au niveau de la queue « queue1 » comme illustré ci-après :






ActiveMQTM

[Home](#) | [Queues](#) | [Topics](#) | [Subscribers](#) | [Connections](#) | [Network](#) | [Scheduled](#) | [Send](#)

Browse queue1



ActiveMQ

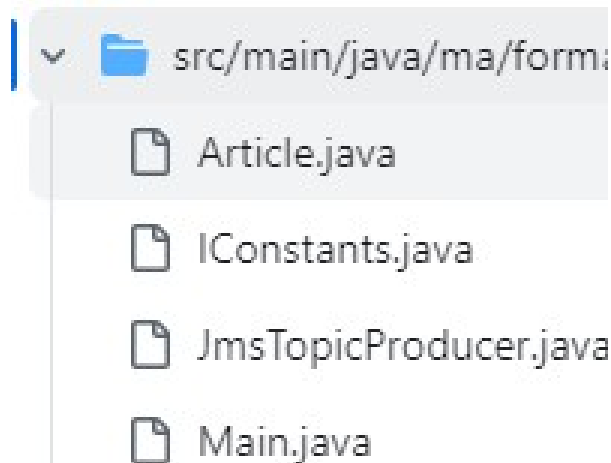
[Home](#) | [Queues](#) | [Topics](#) | [Subscribers](#) | [Connections](#) | [Network](#) | [Scheduled](#) | [Send](#)

Headers	
Message ID	ID:DESKTOP-MDPPCLG-53677-1733941838803-13:1:1:1
Destination	queue://queue1
Correlation ID	
Group	
Sequence	0
Expiration	0
Persistence	Persistent
Priority	4
Redelivered	false
Reply To	
Timestamp	2024-12-11 19:30:39:149 WEST

V. Développement d'une application avec JMS, Apache ActiveMQ et Topic

a. Développement du « Producer »

- Créer un projet Maven, par exemple : jms-activemq-topic-producer-example
- L'arborescence du projet est la suivante :



- Le fichier pom.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>ma.formations.jms</groupId>
  <artifactId>jms-activemq-topic-producer-example</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.activemq</groupId>
      <artifactId>activemq-all</artifactId>
      <version>6.1.4</version>
    </dependency>

    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-api</artifactId>
      <version>2.24.2</version>
    </dependency>

    <!-- Log4j Core -->
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-core</artifactId>
```

```

        <version>2.24.2</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.36</version>
        <scope>provided</scope>
    </dependency>

</dependencies>

</project>

```

- **La classe Article :**

```

package ma. formations.jms;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.io.Serializable;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Article implements Serializable {
    private Long id;
    private String description;
    private Double price;
}

```

- **L'interface IConstants :**

```

package ma. formations.jms;

public interface IConstants {
    String TOPIC_NAME = "my_topic";
    String BROCKER_URL = "tcp://localhost:61616";
}

```

- **La classe JmsTopicProducer :**

```
package ma.formationen.jms;

import jakarta.jms.*;
import org.apache.activemq.ActiveMQConnectionFactory;

public class JmsTopicProducer {
    private final String destinationName;
    private final String brokerUrl;

    public JmsTopicProducer(String destinationName, String brokerUrl) {
        this.destinationName = destinationName;
        this.brokerUrl = brokerUrl;
    }

    public void send(Article article) throws JMSException {
        ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerUrl);
        Connection connection = connectionFactory.createConnection();
        connection.start();
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Destination destination = session.createTopic(destinationName);
        MessageProducer producer = session.createProducer(destination);
        ObjectMessage messageObject = session.createObjectMessage(article);
        producer.send(messageObject);
        System.out.println("Message envoyé : " + messageObject.getObject());
        producer.close();
        session.close();
        connection.close();
    }
}
```

- **La classe Main :**

```
package ma.formationen.jms;

import jakarta.jms.JMSException;

public class Main {
    public static void main(String[] args) {

        JmsTopicProducer producer = new JmsTopicProducer(IConstants.TOPIC_NAME, IConstants.BROCKER_URL);
        try {
            producer.send(Article.builder().id(1L).description("ARTICLE_1").price(1522.0).build());
            producer.send(Article.builder().id(2L).description("ARTICLE_2").price(2100.0).build());
            producer.send(Article.builder().id(3L).description("ARTICLE_3").price(15000.0).build());
            producer.send(Article.builder().id(4L).description("ARTICLE_4").price(900.0).build());

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

b. Développement du « Consumer »

- Créer un projet Maven, par exemple : jms-activemq-topic-consumer-example
- L'arborescence du projet est la suivante :



- Le fichier pom.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ma.formation.jms</groupId>
  <artifactId>activemq-example-consumer</artifactId>
  <version>1.0-SNAPSHOT</version>
  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.activemq</groupId>
      <artifactId>activemq-all</artifactId>
      <version>6.1.4</version>
    </dependency>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-api</artifactId>
      <version>2.24.2</version>
    </dependency>
    <!-- Log4j Core -->
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-core</artifactId>
      <version>2.24.2</version>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.36</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

- **L'interface IConstants :**

```
package ma.formationen.jms;

public interface IConstants {
    String TOPIC_NAME="my_topic";
    String BROCKER_URL="tcp://localhost:61616";
}
```

- **La classe Article :**

```
package ma.formationen.jms;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.io.Serializable;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Article implements Serializable {
    private Long id;
    private String description;
    private Double price;
}
```

- **La classe JmsTopicConsumer**

```
package ma.formationen.jms;
import jakarta.jms.*;
import org.apache.activemq.ActiveMQConnectionFactory;
import java.io.IOException;

public class JmsTopicConsumer {
    private final String destinationName;
    private final String brokerUrl;

    public JmsTopicConsumer(String destinationName, String brokerUrl) {
        this.destinationName = destinationName;
        this.brokerUrl = brokerUrl;
    }

    public void receive() throws JMSException, IOException {
        System.setProperty("org.apache.activemq.SERIALIZABLE_PACKAGES", "*");
        ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerUrl);
        Connection connection = connectionFactory.createConnection();
        connection.setClientID("142");
        connection.start();
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Topic destination = session.createTopic(destinationName);
        TopicSubscriber subscriber = session.createDurableSubscriber(destination, "subscriber1");
        TopicSubscriber subscriber2 = session.createDurableSubscriber(destination, "subscriber2");
    }
}
```

```

subscriber.setMessageListener(msg -> {
    if (msg instanceof ObjectMessage objectMessage) {
        try {
            Article article = (Article) objectMessage.getObject();
            System.out.println("Message reçu par subscriber 1 : " + article);

        } catch (JMSEException e) {
            throw new RuntimeException(e);
        }
    }
});

subscriber2.setMessageListener(msg -> {
    if (msg instanceof ObjectMessage objectMessage) {
        try {
            Article article = (Article) objectMessage.getObject();
            System.out.println("Message reçu par subscriber 2 : " + article);
        } catch (JMSEException e) {
            throw new RuntimeException(e);
        }
    }
});
// Le programme restera en écoute, tant que la connexion est ouverte.
System.in.read();

subscriber.close();
subscriber2.close();
session.close();
connection.close();
}
}

```



- La propriété `org.apache.activemq.SERIALIZABLE_PACKAGES` définit la liste des packages autorisés pour la désérialisation d'objets. Par défaut, aucun package n'est autorisé.
- Cette configuration permet la désérialisation d'objets de n'importe quel package, ce qui n'est pas **recommandé en production** car cela expose le système aux attaques de désérialisation.
- Dans les bonnes pratiques, au lieu d'utiliser la sérialisation java, optez pour les formats plus sûrs comme : **JSON** (avec Jackson ou Gson), **Apache Avro**, **Protobuf** (Protocol Buffers) ou **XML**. Nous verrons d'un l'atelier suivant comment utiliser ce genre de convertisseur avec Spring.

- **La classe Main :**

```
package ma.formationen.jms;
import jakarta.jms.JMSEException;
import java.io.IOException;
public class Main {
    public static void main(String[] args) {
        JmsTopicConsumer consumer = new JmsTopicConsumer(Iconstants.TOPIC_NAME, Iconstants.BROCKER_URL);
        try {
            consumer.receive();
        } catch (JMSEException | IOException e) {
            e.printStackTrace();
        }
    }
}
```

c. Tester l'application

- Lancer ActiveMQ : ([ACTIVEMQ_PATH]/bin/activemq start).
- Exécuter la méthode Main de l'application jms-activemq-topic-consumer-example.
- Exécuter la méthode Main de l'application jms-activemq-topic-producer-example.
- Vérifier les résultats suivants :
 - o Au niveau de la console d'Intellig du projet jms-activemq-topic-producer-example :

```
Message envoyé : Article(id=1, description=ARTI
Message envoyé : Article(id=2, description=ARTI
Message envoyé : Article(id=3, description=ARTI
```

- o Au niveau de la console d'Intellig du projet jms-activemq-topic-consumer-example :

```
Message reçu par subscriber 1 : Article(id=1, descripti
Message reçu par subscriber 2 : Article(id=1, descripti
Message reçu par subscriber 1 : Article(id=2, descripti
Message reçu par subscriber 2 : Article(id=2, descripti
Message reçu par subscriber 1 : Article(id=3, descripti
Message reçu par subscriber 2 : Article(id=3, descripti
```