Messaging

AMT

11 - Messaging

Résumé du document

Cette note explique Jakarta Messaging (JMS), une API Java pour l'envoi, la réception et la gestion de messages entre applications. Elle présente les principaux modèles de communication, comme les queues pour le point-à-point et les topics pour le publish/subscribe, ainsi que des concepts comme la persistance, les garanties de livraison et l'accusé de réception. Enfin, elle détaille les types de messages et leurs propriétés, tout en mettant l'accent sur la fiabilité et la scalabilité des systèmes de messagerie.

Table des matières

1. Messaging en Java (JMS)

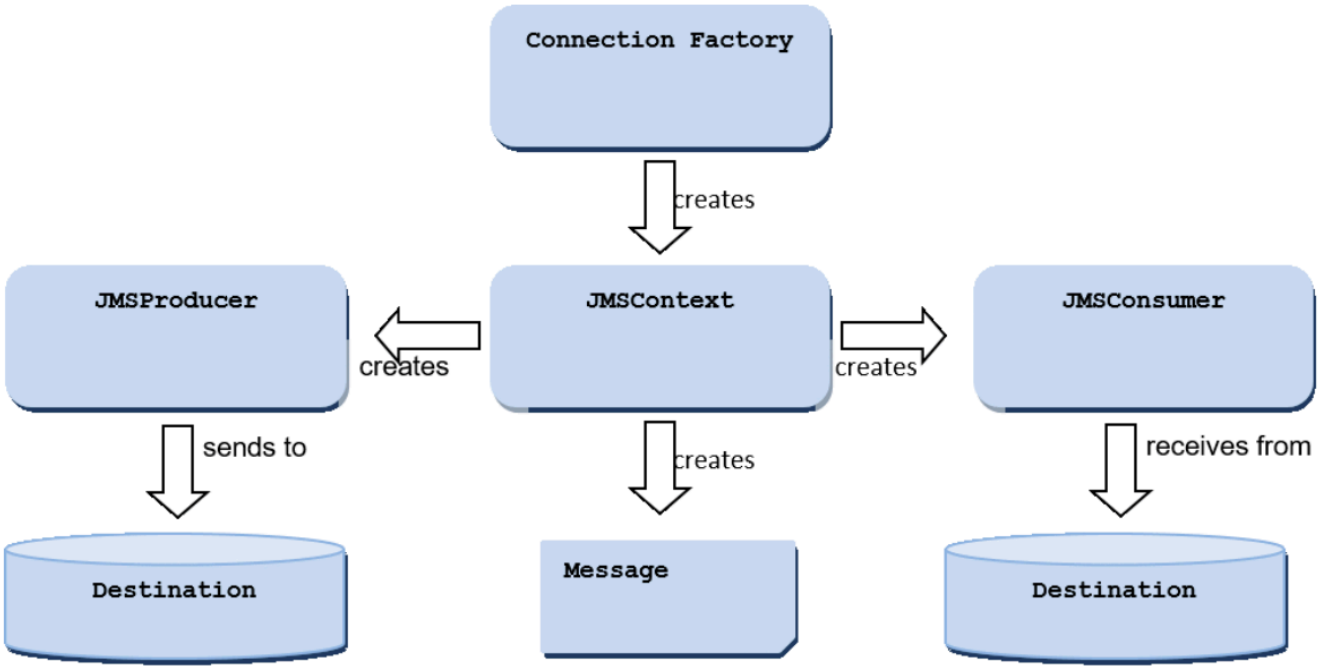
Jakarata Messaging (JMS) est une API Java qui permet de créer, envoyer, recevoir et lire des messages. Elle permet de créer des applications Java qui communiquent entre elles en envoyant des messages.

1.1. Besoin de messaging

Le messaging est utilisé pour communiquer entre des applications qui ne sont pas nécessairement en ligne en même temps.

Cela permet notamment:

- interopérabilité entre applications
- découpler les composants et les applications
- communication asynchrone
- haute scalabilité et disponibilité
- gestion de la charge et basculement

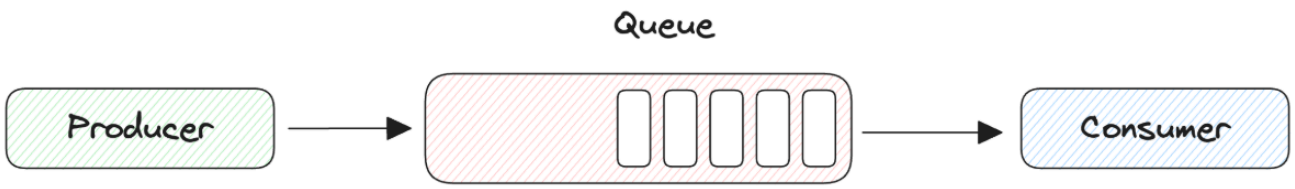


- La `Connection Factory` est utilisée pour créer une connexion à un serveur de messagerie.
- Le `Context` est utilisé pour créer des destinations, producers, consumers, messages, etc.
- La `Destination` est l'endroit où les messages sont envoyés et reçus.
- Le `Producer` est utilisé pour envoyer des messages à une destination.
- Le `Consumer` est utilisé pour recevoir des messages d'une destination.
- Le `Message` est l'objet qui contient les données à envoyer.

2. Concepte de messaging

2.1. Queue modèle

Dans le modèle en queue, les messages sont envoyés à une file d'attente et les consommateurs lisent les messages de la file d'attente. Le modèle en queue est utilisé pour gérer des tâches asynchrones.



Le modèle en queue est un des plus simple, il est basé sur le principe de PTP (point-to-point) communication. Une queue est une destination ou sont envoyés les messsages. Plusieurs producteurs peuvent y écrire et plusieurs consommateurs peuvent accéder à la queue mais que un seul consommateur peut lire un message.

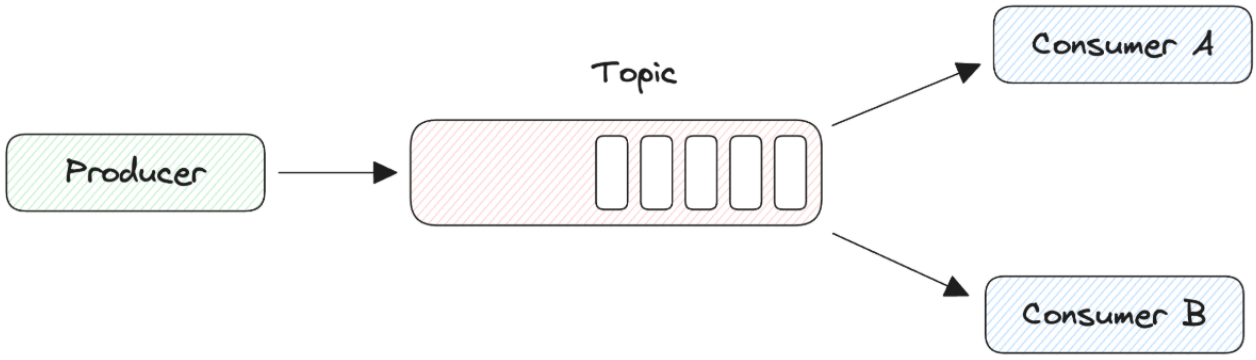
Il s'agit d'un modèle **fire-and-forget**, le producteur envoie le message et ne s'occupe pas de savoir si le message a été reçu ou non.

2.1.1. Exemple

```
// PRODUCER
var context = connectionFactory.createContext()) {

// Create the queue and the producer
var queue = context.createQueue("queue");
var producer = context.createProducer();

// Create the message
var message = context.createTextMessage();
message.setText(text);

// Send the message
producer.send(queue, message);

// CONSUMER
var context = connectionFactory.createContext();
var queue = context.createQueue("queue");
var consumer = context.createConsumer(queue);

// Handle the message asynchronously
consumer.setMessageListener(message -> {
    // Handle the message
});
```

2.2. Topic modèle

Dans le modèle de topic, les messages sont envoyés à un topic et les consommateurs s'abonnent à un topic pour recevoir les messages. Le modèle de topic est utilisé pour la communication de type publish/subscribe.



Le modèle de topic est basé sur le principe de **publish/subscribe**. Un producteur envoie un message à un topic et tous les consommateurs qui sont abonnés à ce topic reçoivent le message. Le message peut donc être consommé par **plusieurs** consommateurs.

Il s'agit d'un modèle **fire-and-forget**, le producteur envoie le message et ne s'occupe pas de savoir si le message a été reçu ou non.

2.2.1. Subscription durable

Jusqu'à présent, nous devons avoir un consommateur connecté au sujet pour recevoir le message. Cependant, dans certains cas, nous voulons recevoir le message même si le consommateur n'est pas connecté. Pour résoudre ce problème, nous pouvons créer un abonnement durable. Un abonnement durable est un abonnement qui survit à la déconnexion du consommateur.

Pour créer un abonnement durable, le consommateur doit fournir un identifiant client et un nom d'abonnement.

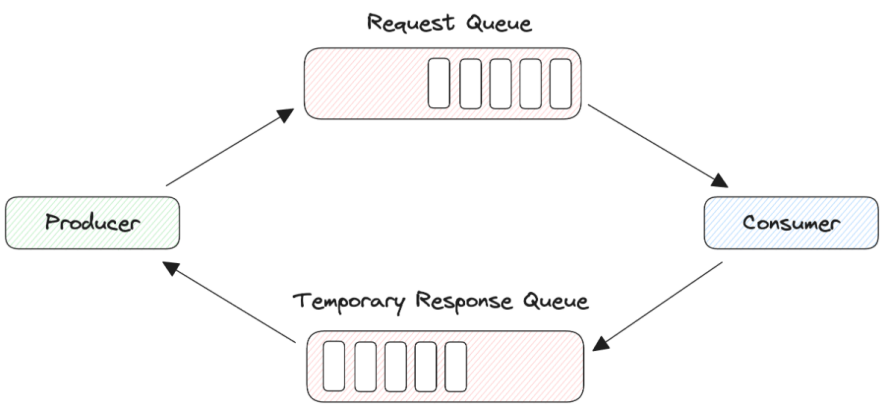
```
var context = connectionFactory.createContext();
var topic = context.createTopic("topic");
var consumer = context.createDurableConsumer(topic, "subscription", "clientID");
```

2.2.2. Exemple

```
// PRODUCER
var context = connectionFactory.createContext()) {

// Create the topic and the producer
var topic = context.createTopic("topic");
var producer = context.createProducer();

// Create the message
var message = context.createTextMessage();
message.setText(text);

// Send the message
producer.send(topic, message);

// CONSUMER
// Create a JMS context
var context = connectionFactory.createContext();
var topic = context.createTopic("topic");
var consumer = context.createConsumer(topic);

// Handle the message asynchronously
consumer.setMessageListener(message -> {
    // Handle the message
});
```

2.3. Request/Response

Le modèle de request/response est utilisé pour envoyer une requête à un service et attendre une réponse. Le modèle de request/response est utilisé pour la communication synchrone.



De ce fait, le modèle fire-and-forget n'est plus adapté. Pour cela, le système suivant est mis en place:

- Le **producteur** envoie un message à une queue, initialise une queue temporaire, inclut la file d'attente temporaire dans le message et attend la réponse sur la queue temporaire.
- Le **consommateur** reçoit le message, le traite et envoie la réponse à la queue temporaire.
- Le **producteur** reçoit la réponse et la traite.

Ce modèle est utilisé pour les tâches asynchrones durant un certain temps.

2.3.1. Exemple

```
// PRODUCER
// Create the queue and the producer
var requestQueue =
context.createQueue("request");
var requestProducer =
context.createProducer();

// Create the request
var requestMessage =
context.createTextMessage();
requestMessage.setText(name);

// Create the response queue and set it as the
reply-to destination
var responseQueue =
context.createTemporaryQueue();
requestMessage.setJMSReplyTo(responseQueue);

// Send the request
requestProducer.send(requestQueue,
requestMessage);

// Block and wait for the response
var responseConsumer =
context.createConsumer(responseQueue);
var responseMessage =
responseConsumer.receive(10000L);
if (responseMessage instanceof TextMessage
textMessage) {
    response = textMessage.getText();
}
```

```
// CONSUMER
var context =
connectionFactory.createContext();
var requestQueue =
context.createQueue("request");
var messageConsumer =
context.createConsumer(requestQueue);

// Handle the request asynchronously
messageConsumer.setMessageListener(message ->
{
  try {
    if (message instanceof TextMessage
textMessage) {
        // Create the response
        var responseText = "Hello " +
textMessage.getText() + "!";
        var responseMessage =
context.createTextMessage(responseText);

        // Send the response to the
destination specified in the request
        var responseDestination =
message.getJMSReplyTo();
        var responseProducer =
context.createProducer();

responseProducer.send(responseDestination,
responseMessage);
    }
  } catch (Exception e) {
      e.printStackTrace();
  }
});
```

2.4. Types de messages

Il existe plusieurs types de messages que vous pouvez envoyer et recevoir en utilisant JMS. Les types de messages les plus courants sont:

- **TextMessage** contient une chaîne de caractères
- **BytesMessage** contient un tableau d'octets
- **ObjectMessage** contient un objet Java sérialisable
- **MapMessage** contient un ensemble de paires nom-valeur
- **StreamMessage** contient un flux de valeurs primitives Java

2.5. Propriétés des messages

Les messages JMS peuvent contenir des propriétés qui peuvent être utilisées pour ajouter des métadonnées aux messages sous forme de clé-valeur.

2.5.1. Exemple

```
var producer = context.createProducer();
var message = context.createTextMessage("Hello, World!");
message.setStringProperty("type", "order");
message.setIntProperty("amount", 100);
producer.send(queue, message);

var consumer = context.createConsumer(queue, "type = 'order' AND amount > 100");
```

3. Fiabilité et livraison

3.1. Message Acknowledgement

Lorsqu'un message est consommé, le consommateur doit envoyer un accusé de réception pour indiquer que le message a été traité avec succès. JSM propose plusieurs façons de gérer l'accusé de réception:

- **AUTO_ACKNOWLEDGE**: L'accusé de réception est envoyé automatiquement lorsque le message est reçu.
- **CLIENT_ACKNOWLEDGE**: L'accusé de réception est envoyé manuellement par le consommateur.
- **SESSION_TRANSACTED**: L'accusé de réception est envoyé lorsque la session est confirmée. Soit toutes les opérations sont confirmées, soit aucune.
- **DUPS_OK_ACKNOWLEDGE**: L'accusé de réception est envoyé automatiquement, mais il peut y avoir des doublons.

3.1.1. Client Acknowledge Example

```
// Create a JMS context
var context = connectionFactory.createContext(Session.CLIENT_ACKNOWLEDGE);
var queue = context.createQueue("queue");
var consumer = context.createConsumer(queue);

// Handle the message asynchronously
consumer.setMessageListener(message -> {
    try {
        // Process the message
    } catch (Exception e) {
        // Handle the error
    } finally {
        // Acknowledge the message
        message.acknowledge();
    }
});
```

3.1.2. Session Acknowledge Example

```
// Create a JMS context with transacted session
var context = connectionFactory.createContext(Session.SESSION_TRANSACTED);
var queue = context.createQueue("queue");
var producer = context.createProducer();
var consumer = context.createConsumer(queue);

try {
    // Send a message
    var message = context.createTextMessage("Hello, World!");
    producer.send(queue, message);

    // Receive and process the message
    var receivedMessage = consumer.receive();

    // Commit the transaction
    context.commit();
} catch (Exception e) {
    // Rollback the transaction in case of an error
    context.rollback();
}
```

3.2. Garantie d'ordre

JMS garantit l'ordre des messsages dans une queue.

- **Out-of-Order**: Les messages peuvent être livrés dans un ordre différent de l'ordre dans lequel ils ont été envoyés.
- **FIFO**: Les messages sont livrés dans l'ordre dans lequel ils ont été envoyés.
- **LIFO**: Les messages sont livrés dans l'ordre inverse de l'ordre dans lequel ils ont été envoyés.
- **Priority**: Les messages sont livrés en fonction de leur priorité.

Il est primordial de consulter la documentation du broker car les garanties d'ordre peuvent varier.

3.3. Garantie de livraison

En fonction du broker utilisé, la garantie de livraison des messages peut varier. Voici quelques garanties de livraison courantes:

- **At-most-once**: Le message est livré au plus une fois.
- **At-least-once**: Le message est livré au moins une fois.
- **Exactly-once**: Le message est livré exactement une fois.

3.4. Autre garanties

- **Persistant**: Le message est stocké sur le disque et n'est pas perdu si le broker tombe en panne.
- **Transactionnel**: Le message n'est délivré que si la transaction est validée.
- **Failover**: Les messages sont répliqués sur plusieurs machines.
- **Scalability**: Les messages sont distribués sur plusieurs machines.

3.5. Persistance

Les messages peuvent être persistants ou non persistants. Les messages persistants sont stockés sur le disque et ne sont pas perdus si le broker tombe en panne.