Architecture SI - TP1

Étude de cas

PhilRaphShop, une plateforme de commerce en ligne, envisage de refondre l'architecture de son système d'informations.

Tout d'abord, elle souhaite optimiser le traitement des commandes passées par ses utilisateurs. Actuellement, les commandes reçues sont accessibles simultanément par toutes les équipes en charge de la préparation des commandes. Cela conduit fréquemment à des confusions et à des problèmes de coordination. Plusieurs équipes peuvent ainsi intervenir sur une même commande. Pour remédier à cette situation, l'entreprise projette de mettre en place un système plus efficace permettant d'attribuer chaque commande effectuée à une seule équipe. Elle vise à éliminer le besoin de coordination et les chevauchements inter-équipes pour traiter les commandes de manière efficace et rapide.

En plus d'améliorer l'expérience utilisateur, la plateforme désire améliorer la communication avec ses utilisateurs. Ainsi, elle étudie la mise en place d'un système d'envoi d'informations automatisé concernant l'arrivée de nouveaux produits, les futures opérations de maintenance et les promotions.

Quels sont les besoins et les solutions proposées ?

› Traitement des commandes (Queue)
Pour le traitement des commandes effectuées, le système doit permettre d'envoyer une commande à une seule équipe de préparation.

L'implémentation d'un système de Queue assure que le message est réceptionné exclusivement par un unique consommateur, correspondant ici à l'équipe. Dans ce cas, chaque commande sera assignée à une équipe parmi celles disponibles.

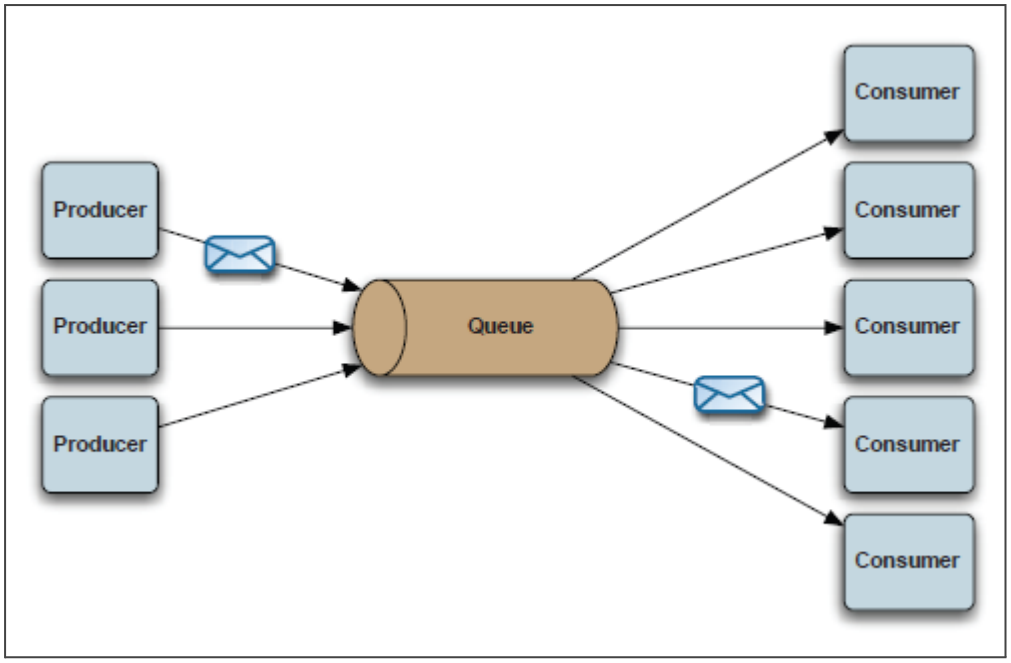

Schéma du fonctionnement d'une Queue

La Queue fonctionne sur le principe FIFO (First-In, First-Out). Ceci garantit que les commandes sont traitées dans l'ordre exact de leur réception. Ce comportement est essentiel pour maintenir la fiabilité et la prévisibilité du traitement des commandes.

Techniquement, voici l'implémentation de l'OrderProducer.

```java
package org.efrei.order;

import javax.jms.*;

public class OrderProducer {

    public void produceOrder() throws JMSException {
        // Connexion à ActiveMQ via la classe ActiveMQUtil
        Connection connection = ActiveMQUtil.getConnection();

        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Queue queue = session.createQueue("OrderQueue");
        MessageProducer producer = session.createProducer(queue);

        String text = "Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2";
        TextMessage message = session.createTextMessage(text);

        producer.send(message);
        System.out.println("Message envoyé : " + text);

        session.close();
    }
}
```

Le OrderProducer place les commandes dans la Queue. Il crée une session JMS, une queue spécifique “OrderQueue”, et y envoie des messages qui représentent les commandes. Ces messages sont ensuite disponibles pour être consommés par les équipes de préparation.

Ici, la commande est représentée par un texte. Une implémentation avancée impliquerait de sérialiser une instance de Order et de l’envoyer à la Queue.

Ce message est consommé par une instance de PreparationTeamConsumer.

```
package org.efrei.order;

import javax.jms.*;

public class PreparationTeamConsumer {

    public void consumeOrder() throws JMSException {
        // Connexion à ActiveMQ via la classe ActiveMQUtil
        Connection connection = ActiveMQUtil.getConnection();

        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Queue queue = session.createQueue("OrderQueue");
        MessageConsumer consumer = session.createConsumer(queue);

        consumer.setMessageListener(message -> {
            if (message instanceof TextMessage textMessage) {
                try {
                    System.out.println("Commande reçue  : " + textMessage.getText());
                } catch (JMSException e) {
                    throw new RuntimeException(e);
                }
            }
        });

    }
}
```

L’OrderProducer transmet correctement la commande à la PreparationTeamConsumer ! Cela permet donc de pouvoir envoyer une commande à une équipe. Chaque instance de PreparationTeamConsumer attend les messages dans la Queue, prête à traiter la commande reçue exclusivement par elle.

```
 INFO | For help or more information please see: http://activemq.apache.org
Message envoyé : Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2
Commande reçue  : Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2
 INFO | Apache ActiveMQ 5.12.0 (localhost, ID:Teroaz-54840-1710099446952-0:1) is shutting down
```

Cependant, nous souhaitons nous assurer que les commandes sont bien dispatchées.

Pour ce faire, il faut utiliser du multithreading pour définir plusieurs consumers qui traitent les commandes simultanément.

Voici l'implémentation multithreadée.

```java
package org.efrei.order;

public class OrderQueueImplementation {

    public static void main(String[] args) {

        ActiveMQUtil.getConnection();

        final OrderProducer orderProducer = new OrderProducer();
        final PreparationTeamConsumer preparationTeamConsumer1 = new PreparationTeamConsumer();
        final PreparationTeamConsumer preparationTeamConsumer2 = new PreparationTeamConsumer();

        // Exécution de la production des commandes dans son propre thread
        new Thread(() -> {
            try {
                for (int i = 0; i < 5; i++) {
                    orderProducer.produceOrder();
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }).start();

        // Exécution des consommateurs dans leurs propres threads
        new Thread(() -> {
            try {
                preparationTeamConsumer1.consumeOrder(1);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }).start();

        new Thread(() -> {
            try {
                preparationTeamConsumer2.consumeOrder(2);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

L'exécution avec la production de 5 commandes et 2 équipes de préparation nous confirme que les commandes sont bien dispatchées.

```
Message envoyé : Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2
[Team 2] Commande reçue  : Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2
Message envoyé : Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2
Message envoyé : Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2
Message envoyé : Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2
Message envoyé : Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2
[Team 1] Commande reçue  : Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2
[Team 2] Commande reçue  : Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2
[Team 1] Commande reçue  : Commande de l'utilisateur XYZ, contenant : 1x produit1, 2x produit2
```

> Envoi d'informations aux utilisateurs
L'amélioration de la communication avec les utilisateurs de PhilRaphShop passe par la mise en place d'un système capable de diffuser des informations de manière globale. Pour rappel, les informations sont : l'arrivée de nouveaux produits, les futures opérations de maintenance et les promotions.

La mise en place d'un système de Topic s'avère être la meilleure solution car cela permettra d'envoyer des messages à plusieurs consommateurs qui seront abonnés à des topics spécifiques (cette communication doit être personnalisable selon les intérêts des utilisateurs).

En effet, à l'opposé des Queues, où chaque message est destiné à un unique consommateur, un Topic permet à un message d'être reçu par plusieurs consommateurs qui s'y ont souscrit. Ce modèle est appelé Publish-Subcribe ou Pub/Sub.

Pour simplifier l'implémentation, parmi les Topic envisagés, nous sélectionnons : "NewProductsTopic".

Voici le ProductNotificationPublisher.

```
○ ○ ○

package org.efrei.notifications;

import org.efrei.ActiveMQUtil;

import javax.jms.*;

public class ProductNotificationPublisher {
    public void publishNewProduct(String productInfo) throws JMSException {

        Connection connection = ActiveMQUtil.getConnection();

        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Topic topic = session.createTopic("NewProductsTopic");

        MessageProducer producer = session.createProducer(topic);
        TextMessage message = session.createTextMessage(productInfo);
        producer.send(message);

        System.out.println("Publication des infos d'un nouveau produit : " + productInfo);
    }
}
```

Cette classe est responsable de publier les informations sur les nouveaux produits dans le NewProductsTopic. Dans une implémentation complète, lorsqu'un nouveau produit est disponible, le publisher sérialise les détails du produit en un message et le publie sur le Topic.

Une fois le message publié, l'ensemble des subscribers le reçoivent.

Dans l'implémentation, un subscriber est défini par la classe UserNotificationSubscriber.

```java
package org.efrei.notifications;

import org.efrei.ActiveMQUtil;

import javax.jms.*;

public class UserNotificationSubscriber {
    public void subscribeToNewProducts(int id) throws JMSException {

        Connection connection = ActiveMQUtil.getConnection();

        connection.start();
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Topic topic = session.createTopic("NewProductsTopic");

        MessageConsumer consumer = session.createConsumer(topic);
        consumer.setMessageListener(message -> {
            TextMessage textMessage = (TextMessage) message;
            try {
                System.out.println("[Sub " + id + "] Notification d'un nouveau produit : " + textMessage.getText());
            } catch (JMSException e) {
                e.printStackTrace();
            }
        });

    }
}
```

La classe UserNotificationSubscriber définit un consommateur qui s'abonne au NewProductsTopic pour recevoir les notifications sur les nouveaux produits. Lorsqu'un message est publié sur ce Topic, tous les subscribers actifs reçoivent le message de manière simultanée.

Le résultat observé est bien celui attendu : l'ensemble des subscribers reçoivent l'information d'un nouveau produit de manière simultanée.

```
Publication des infos d'un nouveau produit : Nouveau produit 0
Publication des infos d'un nouveau produit : Nouveau produit 1
Publication des infos d'un nouveau produit : Nouveau produit 2
Publication des infos d'un nouveau produit : Nouveau produit 3
Publication des infos d'un nouveau produit : Nouveau produit 4
[Sub 1] Notification d'un nouveau produit : Nouveau produit 0
[Sub 2] Notification d'un nouveau produit : Nouveau produit 1
[Sub 3] Notification d'un nouveau produit : Nouveau produit 1
[Sub 4] Notification d'un nouveau produit : Nouveau produit 2
[Sub 5] Notification d'un nouveau produit : Nouveau produit 2
[Sub 1] Notification d'un nouveau produit : Nouveau produit 1
[Sub 2] Notification d'un nouveau produit : Nouveau produit 2
[Sub 3] Notification d'un nouveau produit : Nouveau produit 2
[Sub 4] Notification d'un nouveau produit : Nouveau produit 3
[Sub 5] Notification d'un nouveau produit : Nouveau produit 3
[Sub 1] Notification d'un nouveau produit : Nouveau produit 2
[Sub 2] Notification d'un nouveau produit : Nouveau produit 3
[Sub 3] Notification d'un nouveau produit : Nouveau produit 3
[Sub 4] Notification d'un nouveau produit : Nouveau produit 4
[Sub 5] Notification d'un nouveau produit : Nouveau produit 4
[Sub 1] Notification d'un nouveau produit : Nouveau produit 3
[Sub 2] Notification d'un nouveau produit : Nouveau produit 4
[Sub 3] Notification d'un nouveau produit : Nouveau produit 4
[Sub 1] Notification d'un nouveau produit : Nouveau produit 4
```