

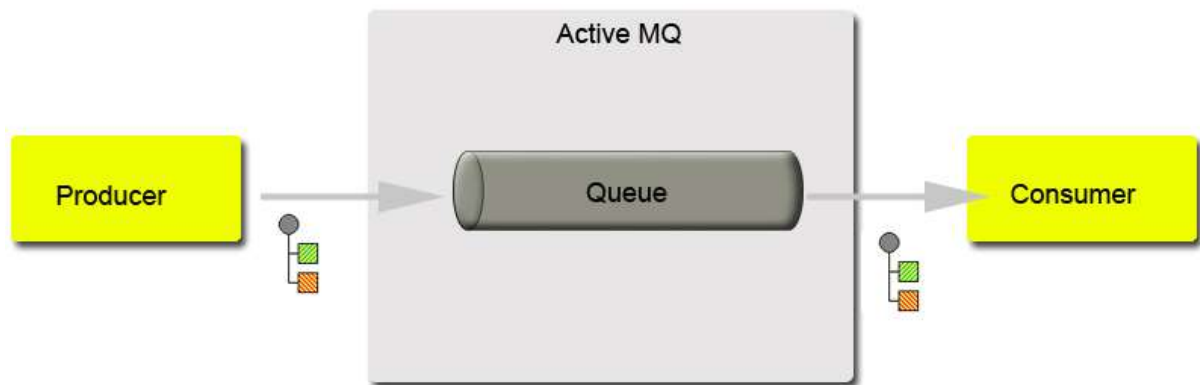
JMS et Apache Camel / Java

Exercice 1 : JMS et Active MQ – Intégration en mode one way.....	2
Exercice 2 : JMS et Active MQ – Intégration en mode Request / Response	7
Exercice 3 : Le routage des messages avec Camel – premiers pas	9
Exercice 4 : Intégration Camel et Active MQ.....	11
Exercice 5 : Configuration XML avec Camel	15

Exercice 1 : JMS et Active MQ – Intégration en mode one way

Active MQ est un serveur de messagerie Open Source de la fondation Apache, accessible, entre autre, via l'API java JMS.

Le but de cet exercice est d'installer, configurer et démarrer ActiveMQ puis de développer des programmes Java de type Producteur et Consommateur JMS.



Installation de Active MQ (version 5.X)

- Télécharger Active MQ : <http://activemq.apache.org/download.html>
- Décompresser l'archive sur votre poste dans un répertoire dont le chemin d'accès ne contient aucun espace.

Démarrer Active MQ

- Dans une console (shell/cmd), naviguer jusqu'au répertoire bin du répertoire où a été installé ActiveMQ et lancer la commande :

```
activemq start
```

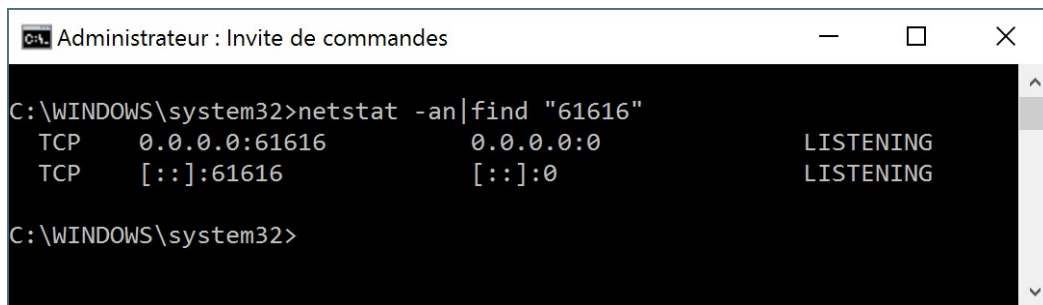
```
Invite de commandes - activemq start
ACTIVE_MQ_DATA: C:\Dev\apache-activemq-5.14.1\bin\..\data
Loading message broker from: xbean:activemq.xml
INFO | Refreshing org.apache.activemq.xbean.XBeanBrokerFactory$1@482f8f11: startup date [Wed Jan 11 17:37:17 CET 2017];
root of context hierarchy
INFO | Using Persistence Adapter: KahaDBPersistenceAdapter[C:\Dev\apache-activemq-5.14.1\bin\..\data\kahadb]
INFO | KahaDB is version 6
INFO | Recovering from the journal @1:1369606
INFO | Recovery replayed 3322 operations from the journal in 0.141 seconds.
INFO | PLISTore:[C:\Dev\apache-activemq-5.14.1\bin\..\data\localhost\tmp_storage] started
INFO | Apache ActiveMQ 5.14.1 (localhost, ID:sedna-50762-1484152639229-0:1) is starting
INFO | Listening for connections at: tcp://sedna:61616?maximumConnections=1000&wireFormat.maxFrameSize=104857600
INFO | Connector openwire started
INFO | Listening for connections at: amqp://sedna:5672?maximumConnections=1000&wireFormat.maxFrameSize=104857600
INFO | Connector amqp started
INFO | Listening for connections at: stomp://sedna:61613?maximumConnections=1000&wireFormat.maxFrameSize=104857600
INFO | Connector stomp started
INFO | Listening for connections at: mqtt://sedna:1883?maximumConnections=1000&wireFormat.maxFrameSize=104857600
INFO | Connector mqtt started
WARN | ServletContext@0.e.j.s.ServletContextHandler@7f0d96f2{/,null,STARTING} has uncovered http methods for path: /
INFO | Listening for connections at ws://sedna:61614?maximumConnections=1000&wireFormat.maxFrameSize=104857600
INFO | Connector ws started
INFO | Apache ActiveMQ 5.14.1 (localhost, ID:sedna-50762-1484152639229-0:1) started
INFO | For help or more information please see: http://activemq.apache.org
INFO | No Spring WebApplicationInitializer types detected on classpath
INFO | ActiveMQ WebConsole available at http://0.0.0.0:8161/
INFO | ActiveMQ Jolokia REST API available at http://0.0.0.0:8161/api/jolokia/
INFO | Initializing Spring FrameworkServlet 'dispatcher'
INFO | No Spring WebApplicationInitializer types detected on classpath
INFO | jolokia-agent: Using policy access restrictor classpath:/jolokia-access.xml
```

Un Broker de messages est maintenant disponible sur la machine. Il écoute sur le port 61616.

Ne pas fermer la console, cela arrêterait le broker.

- Pour vérifier le bon fonctionnement de active MQ,
 - afficher sa console web d'administration à partir de l'url
<http://localhost:8161/admin>
 - ouvrir une autre console et taper

```
netstat -an|find « 61616 »
```



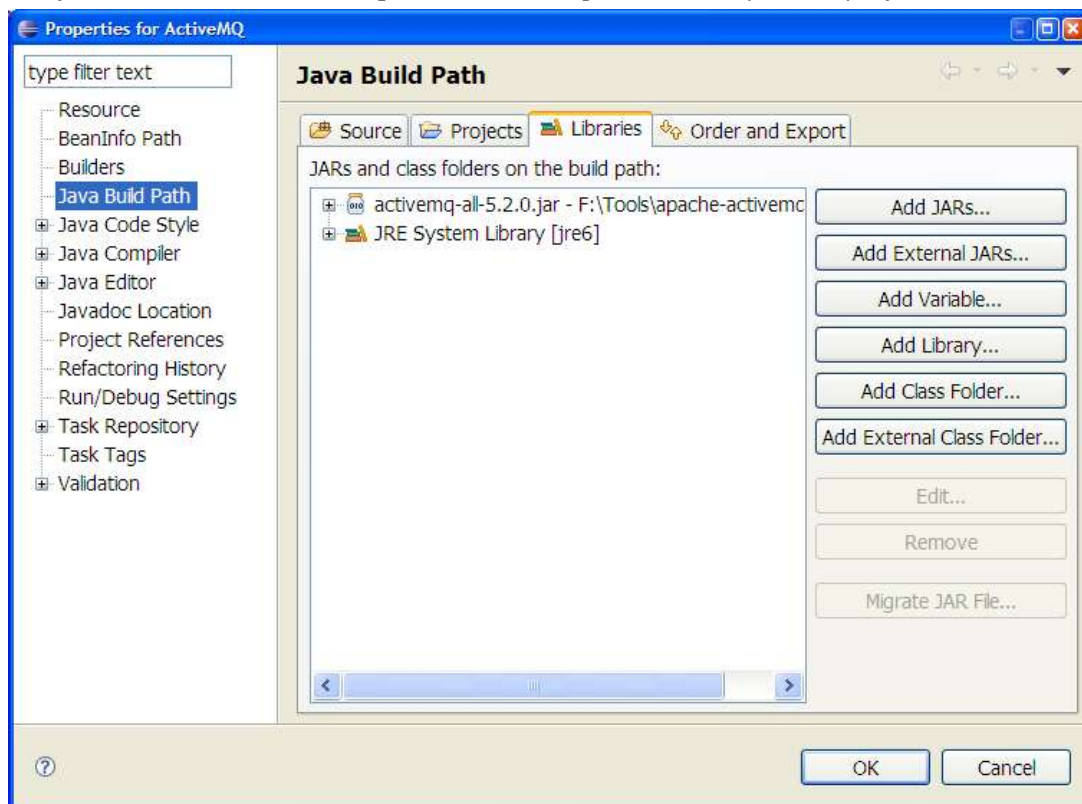
```
C:\WINDOWS\system32>netstat -an|find "61616"
TCP    0.0.0.0:61616      0.0.0.0:0          LISTENING
TCP    [::]:61616        [::]:0             LISTENING

C:\WINDOWS\system32>
```

La configuration des active MQ et des ports utilisés se trouve dans le fichier `conf/activemq.xml` du répertoire d'installation de active MQ.

Préparer le projet Java

- Sous Eclipse, créer un nouveau projet Java.
- Ajouter l'archive `activemq-all-5.X.Y.jar` au buildpath du projet.



Cette archive contient, entre autre, l'ensemble des classes implémentant les SPI (Service Provider Interface) de JMS. Elle intègre également l'ensemble des dépendances utiles, elle est donc la seule et unique archive dont on a besoin.

- Pour pouvoir utiliser active MQ avec l'API JMS et JNDI, il faut configurer l'environnement JNDI. Pour cela, ajouter, à la racine du répertoire source du projet, le fichier « `jndi.properties` » décrit ci-dessous.

```
java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory

# use the following property to configure the default connector
java.naming.provider.url = tcp://localhost:61616

# use the following property to specify the JNDI name the connection factory
# should appear as.
connectionFactoryNames = connectionFactory

# register some queues in JNDI using the form
# queue.[jndiName] = [physicalName]
queue.MyQueue = example.MyQueue

# register some topics in JNDI using the form
# topic.[jndiName] = [physicalName]
topic.MyTopic = example.MyTopic
```

Le fichier `jndi.properties` est un standard du monde Java. Il permet de configurer un environnement JNDI utilisable par une application JavaSE. Ce fichier doit se trouver dans le classpath de l'application.

Les propriétés `java.naming.factory.initial` et `java.naming.provider.url` sont des propriétés standards de ce fichier. Les autres propriétés présentes ici sont spécifiques à active MQ et permettent d'associer des noms JNDI à des ressources active MQ.

Avec activeMQ, le seul fait de demander à accéder à une destination entraîne sa création au sein du broker. La queue `example.MyQueue` n'apparaît donc nulle part ailleurs que dans ce fichier ; elle sera créée quand le code demandera à la récupérer.

En environnement JavaEE, l'utilisation de JMS ne nécessite pas d'intégrer ce fichier à l'application. L'environnement JNDI est mis à disposition par le serveur d'application lui-même.

Développement du consommateur

- Développer une classe nommée `Consumer` permettant de consommer des messages de type texte, présent dans la queue active MQ nommée `example.MyQueue`.
 - Cette classe contiendra une méthode `main`.
 - Une fois lancé, le système attend indéfiniment des messages de type texte.
 - Dès qu'un message arrive, il affiche son contenu dans la sortie standard.
 - L'envoi d'un message autre que de type texte, ou possédant un corps vide, entraîne l'arrêt de la consommation.

```
import javax.jms.Connection ;
import javax.jms.ConnectionFactory ;
import javax.jms.Destination ;
import javax.jms.Message ;
import javax.jms.MessageConsumer ;
import javax.jms.Session ;
import javax.jms.TextMessage ;
import javax.naming.InitialContext;

public class Consumer {

    private ConnectionFactory connectionFactory;
    private Destination destination;

    public static void main(String[] args) {
        try {
            Consumer consumer = new Consumer();
            consumer.connect();
            consumer.waitForMessage();
        } catch (Throwable t) {
            t.printStackTrace() ;
        }
    }

    private void waitForMessage() throws Exception {
        // Créer une connexion au système de messagerie
        // Et affiche les messages au fur et à mesure de leur arrivée dans la queue
    }

    private void connect() throws Exception {
        // Initialise les attributs connectionFactory et destination.
    }
}
```

Aucun autre import de classe n'est nécessaire que ceux déjà indiqués dans le squelette de la classe. Seules des classes de l'API de base JMS sont utilisées. Le code est totalement indépendant de activeMQ et fonctionnerait tout aussi bien avec n'importe quel provider JMS.

Développement du producteur

- Développer une classe nommée `Producer` permettant de poster des messages de type text et dans la queue active MQ nommée `example.MyQueue`.
 - Cette classe contiendra une méthode `main`.
 - Une fois lancé, le système attend indéfiniment qu'on saisisse un texte qui sera envoyé en tant que message.
 - Si le message « QUIT » est saisi, l'application s'arrête.

```
import java.util.Scanner ;

import javax.jms.Connection ;
import javax.jms.ConnectionFactory;
import javax.jms.DeliveryMode;
import javax.jms.Destination ;
import javax.jms.MessageProducer ;
import javax.jms.Session ;
import javax.jms.TextMessage ;
import javax.naming.InitialContext;

public class Producer {

    private ConnectionFactory connectionFactory;
    private Destination destination;

    public static void main(String[] args) {
        try {
            Producer producer = new Producer();
            producer.connect();
            producer.sendMessage();
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }

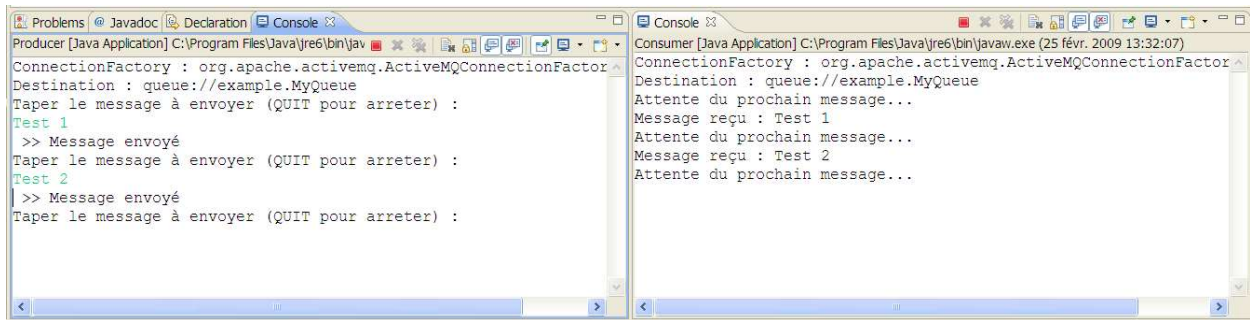
    private void sendMessage() throws Exception {
        // Créer une connexion au système de messagerie
        // Emet des messages au fur et à mesure que l'utilisateur les saisit
    }



    private void connect() throws Exception {
        // Initialise les attributs connectionFactory et destination.
    }
}
```

Aucun autre import de classe n'est nécessaire que ceux déjà indiqués dans le squelette de la classe. Seules des classes de l'API de base JMS sont utilisées. Le code est totalement indépendant de activeMQ et fonctionnerait tout aussi bien avec n'importe quel provider JMS.

Lancement du système

- S'assurer que Active MQ est bien lancé.
- Démarrer le Consumer.
- Démarrer le Producer.
- Saisir des messages dans le Producer et regarder le résultat dans le Consumer.



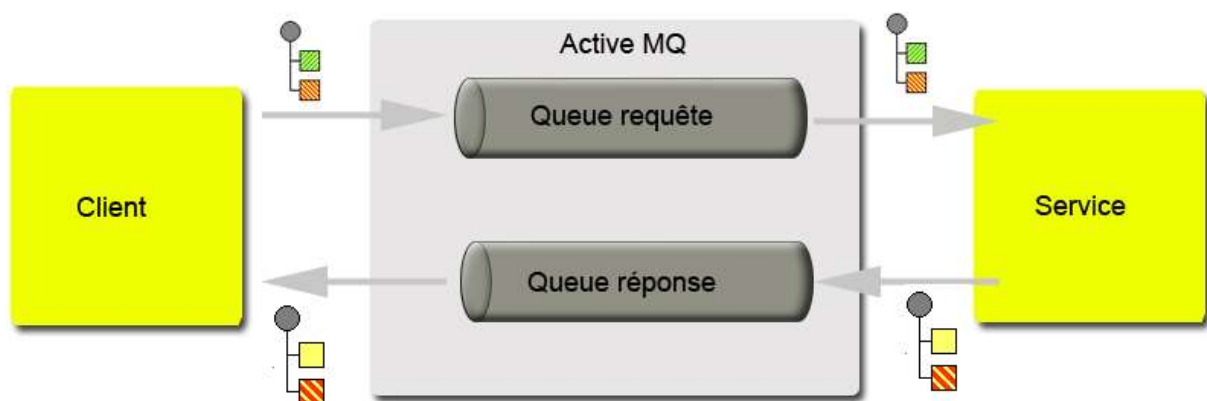
Sous Eclipse, utiliser le bouton  de la vue Console pour créer une nouvelle console, puis utiliser le bouton  pour figer chaque console sur un des deux processus.

- Vérifier à l'aide de la console web d'administration que les messages envoyés sont bien présents dans une destination tant qu'ils n'ont pas été consommés, et qu'ils n'y sont plus une fois consommés
- Essayer d'arrêter Active MQ après production sans consommation et regarder ce que sont devenus les messages
- Peut-on modifier ce comportement ?

Exercice 2 : JMS et Active MQ – Intégration en mode Request / Response

Le but de ce deuxième exercice est encore une fois de développer deux classes Java qui devront communiquer via une Queue disponible sous Active MQ.

Cette fois ci, une première classe (Client) enverra un message dans une queue requête. Le message sera récupéré et traité par la seconde classe (Service) qui créera et publiera un message réponse dans une queue réponse. La classe Client pourra alors récupérer la réponse.



Préparer le projet Java

- Sous Eclipse, créer un nouveau projet Java.
- Ajouter l'archive `activemq-all-5.X.Y.jar` au buildpath du projet.
- Ajouter le fichier `jndi.properties` en configurant les deux queues nécessaires.

Développer le service

Le service est une classe Java proposant une seule méthode métier nommée « `reverse` ». Elle prend en entrée une chaîne de caractère et renvoie la chaîne inversée (« ABCD » renvoie « DCBA »).

Pour invoquer ce service, on envoie des messages de type texte dans la queue requête. La méthode `reverse` est invoquée et le résultat est placé dans un autre message de type texte, publié dans la queue réponse.

- Créer la classe `Service`.
- Développer la méthode `reverse`.
- Ajouter les méthodes nécessaires à la connexion au système de messagerie, à la récupération des requêtes et à l'émission des réponses.

Le code du service est semblable à ce qui a été fait dans l'exercice 1. La particularité ici est que la classe est à la fois producteur et consommateur de messages.

Développer le client

Le client permettra d'invoquer le service. Il doit donc émettre des messages (à partir d'un texte saisi à la console par un utilisateur) et afficher les réponses quand elles sont disponibles.

On souhaite cependant pour continuer à émettre des requêtes sans avoir besoin d'attendre la réponse à chaque fois. On doit donc créer un consommateur de message asynchrone (ie implémenter l'interface `MessageListener`)

- Créer la classe `Client`
- Ajouter le code nécessaire pour que le `Client` soit un `MessageListener` associé à la queue réponse.
- Ajouter le code nécessaire pour que le `Client` émette les requêtes dans la queue requête.

Un exemple de `MessageListener` peut être trouvé au chapitre 33 du tutorial J2EE 1.4 :

<http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html>

Tester le système

- S'assurer que Active MQ est démarré.
- Lancer le service.
- Lancer le client.
- Emettre des requêtes et analyser les réponses.

```
Service [Java Application]
Attente de la prochaine requête...
Message reçu : ABCD
Réponse envoyée
Attente de la prochaine requête...

Client [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (25 févr. 2009 14:36:28)
Taper le texte à inverser (QUIT pour arreter) :
ABCD
>> Requête envoyée, ID = ID:europa-4067-1235568988343-0:1:1:1
Taper le texte à inverser (QUIT pour arreter) :
>> Réponse obtenue, requête ID:europa-4067-1235568988343-0:1:1:1 --> DCBA
```


Questions ouvertes

- Quelles sont les difficultés posées par la consommation asynchrone de messages ? Comment y remédier ?
- Que penser de la façon dont le système a été développé ? Y aurait-il une façon plus élégante de concevoir ce système ?

Exercice 3 : Le routage des messages avec Camel – premiers pas



Apache Camel est un moteur de routage et de médiation de messages prenant en compte un grand nombre de protocoles et de transports standards, et s'appuyant sur des règles pour déterminer l'acheminement de ces messages.

L'implémentation d'Apache Camel est basée sur les POJOs (Plain Old Java Objects) ; il utilise également un langage de type DSL (Domain Specific Language) reposant sur Java, pour exprimer les règles de routage de manière claire, ainsi que les modèles d'intégration mis en œuvre.

Apache Camel permet d'appliquer les principaux modèles d'intégration d'entreprise décrits dans l'excellent ouvrage Enterprise Integration Patterns.

D'un certain point de vue, Apache Camel peut être considéré à lui seul comme un Enterprise Service Bus (ESB) léger !

A noter également que celui-ci peut être utilisé au travers du composant JBI servicemix-camel du projet Apache ServiceMix, afin d'assurer le routage des messages entre services JBI.

Préparation de l'environnement

- Télécharger une version 2.X du framework Camel (<http://camel.apache.org/download.html>)
- Décompresser l'ensemble des ressources dans un répertoire dont le chemin d'accès ne contient aucun espace.

Parmi les ressources se trouve la documentation au format HTML ou PDF (répertoire doc/manual). Malheureusement, Camel souffre d'une documentation chaotique ; mais ce document reste la seule référence disponible à ce jour.

Un article intéressant est cependant disponible à cette adresse :

<http://soa.sys-con.com/node/504392?page=0,1>.

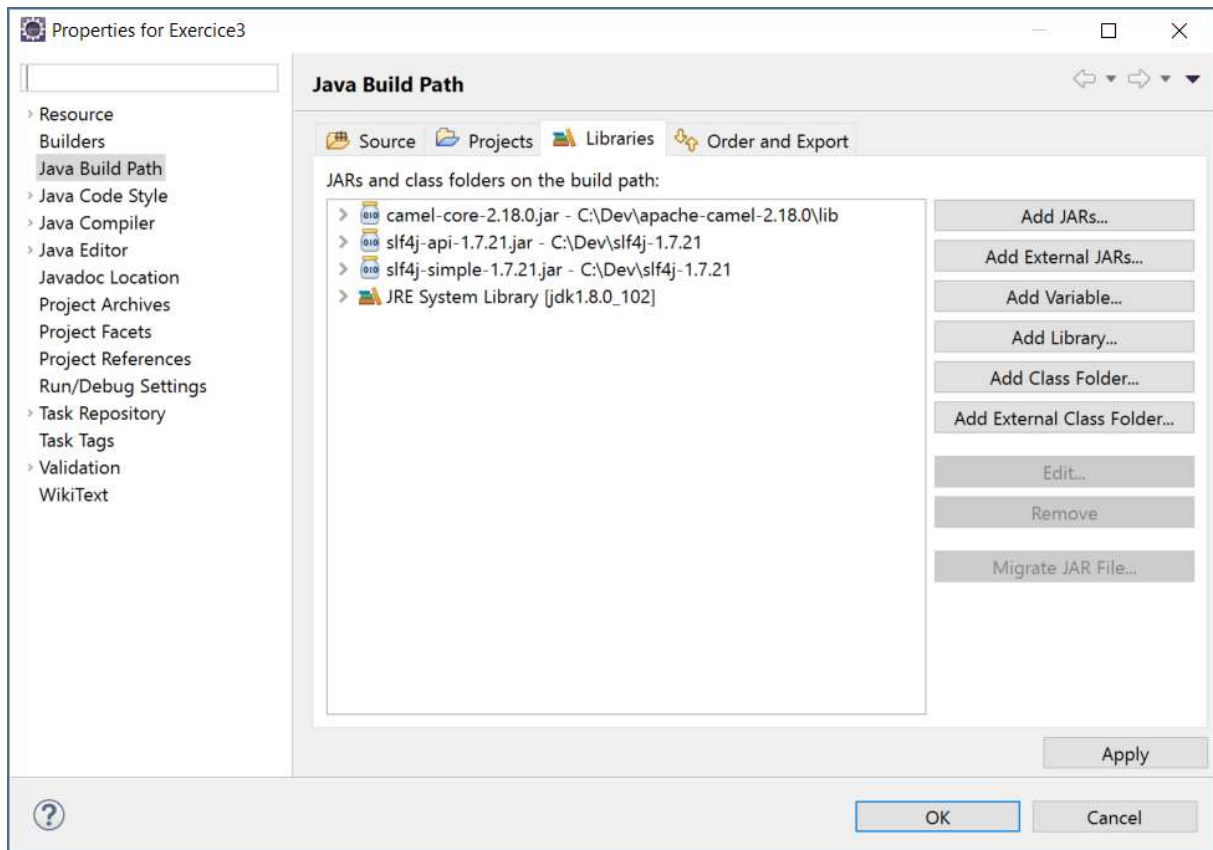
La javadoc de l'API est également disponible à l'adresse

<http://camel.apache.org/maven/current/camel-core/apidocs/index.html>

Développement d'un pipe simple

- Sous Eclipse, créer un nouveau projet Java.

- Ajouter à son buildpath :
 - camel-core-2.X.Y.jar se trouvant dans le répertoire `lib` de Apache Camel,
 - l'API et l'implémentation du système de log SLF4J.



- Créer les répertoires `C:\temp\in` et `C:\temp\out`.

Le but de cet exercice est de créer un pipe de traitement de messages arrivant sous forme de fichiers textes dans le répertoire `in`. Dès qu'un fichier arrive, il est récupéré, son contenu est affiché et il est déplacé vers le répertoire `out`.



- Créer une nouvelle classe Java, écrire la méthode `main` donnée ci-dessous, et :
 - Remplacer `URI-IN` et `URI-OUT` par les valeurs adéquates
 - Compléter le code pour permettre l'affichage du contenu du fichier sur la sortie standard

```
public static void main(String[] args) {
    try {
        CamelContext context = new DefaultCamelContext();
        context.addRoutes(new RouteBuilder() {

            public void configure() throws Exception {
                from("URI-IN")
                    .process(new Processor() {

                        public void process(Exchange e) throws Exception {
                            GenericFileMessage<File> fileIn =
                                (GenericFileMessage<File>) e.getIn();
                            System.out.println("Echange reçu : " + fileIn);
                            //
                            // COMPLETER AVEC L'AIDE DE LA JAVADOC CAMEL
                            //
                        }

                    })
                    .to("URI-OUT");
            }

        });
        context.start();
        // ajouter ici un code permettant de pauser le thread courant
        context.stop();
    } catch (Throwable t) {
        t.printStackTrace();
    }
}
```

- Lancer le programme.
- Créer un fichier texte et le déposer dans le répertoire in.
- Analyser les logs et regarder le contenu du répertoire out.

Comment ça marche ?

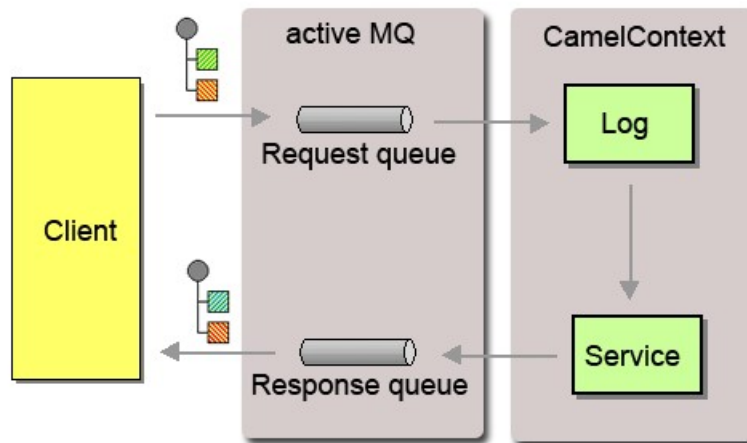
L'interface `CamelContext` est la classe de base du framework Camel. C'est dans une instance de cette interface que l'on stocke les routes reliant les EndPoints. La configuration du `CamelContext` est ici réalisée directement en code Java, mais il est également possible de l'externaliser sous forme de document XML.

On crée une route reliant les EndPoints correspondant aux deux répertoires in et out. Entre ces deux EndPoints, on ajoute un `Processor` qui se charge de logger le contenu du fichier. A noter qu'en l'absence de ce `Processor`, le transfert entre les deux EndPoints est exécuté par Camel.

L'objet de type `Exchange` passé en paramètre de la méthode `process` contient le message d'entrée et permet de définir le message de sortie. Dans l'exemple ci-dessus, le message de sortie n'étant pas défini, il est valorisé par défaut avec le message d'entrée.

Exercice 4 : Intégration Camel et Active MQ

Allons plus loin dans l'utilisation de Camel. On dispose d'une classe java proposant une méthode qui renvoie le prix d'un produit à partir de son code. On souhaite exposer cette méthode à travers Active MQ, le traitement des messages devant être effectué dans une route Camel.



Préparation de l'environnement

On implémente un système en mode requête-réponse, les messages transitant par des queues Active MQ. Le CamelContext doit donc être configuré de manière à récupérer les messages de la queue requête et poster les réponses dans la queue réponse. On utilisera donc les bibliothèques Camel permettant d'utiliser JMS.

Créer un nouveau projet Java et ajouter à son buildpath les JARs nécessaires :

- La bibliothèque `activemq-all-5.X.Y.jar`
- La bibliothèque `camel-core-2.X.Y.jar`
- La bibliothèque `camel-jms-2.X.Y.jar`

Développement du système

La documentation Javadoc de Camel est disponible sur le site www.javadoc.io.

- Développer la classe implémentant le service de récupération du prix d'un produit (ne pas hésiter à modifier la logique de récupération du prix, mais veiller à respecter la signature fournie dans l'exemple ci-dessous)

```
public class FournisseurService {

    public float getPrix(String idProduit) {
        return 5.0f;
    }

}
```

- Développer le programme initialisant et démarrant le CamelContext. En respectant les traitements présentés dans le schéma du système (log puis récupération du prix).

Pour que le CamelContext connaisse l'environnement JMS que l'on utilise, il faut lui ajouter un composant (Component) de type JmsComponent. Le code est le suivant :

```
context.addComponent("jms-test",
    JmsComponent.jmsComponentAutoAcknowledge(connectionFactory));
```

Le nom « `jms-test` » est le nom logique du composant. Lorsque des EndPoints utilisant cet environnement JMS sont référencés dans une route, cela se fait de cette manière :

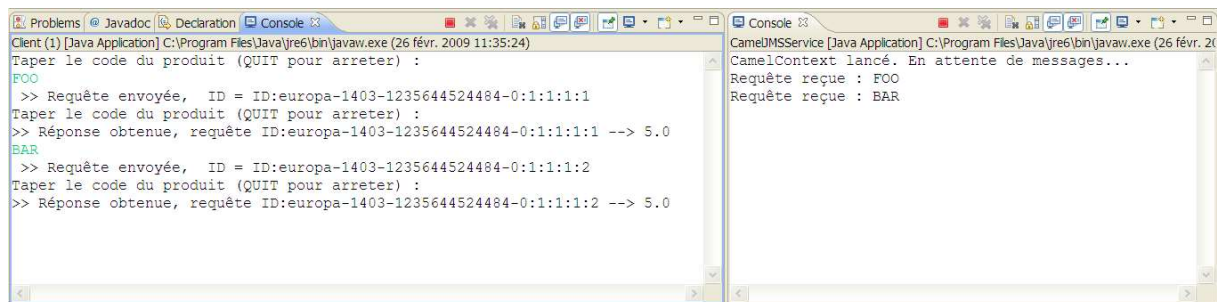
```
from("jms-test:fournisseur.requestQueue")
```

Le nom du endpoint "`jms-test:fournisseur.requestQueue`" signifie : la destination JMS nommée « `fournisseur.requestQueue` » (attention, il s'agit du nom physique dans Active MQ, pas d'un nom JNDI) dans le système JMS géré par le composant « `jms-test` ».

- Développer un client pour tester le système (client JMS standard).

Test du système

- Vérifier que active MQ est démarré.
- Lancer le CamelContext.
- Lancer le client.
- Analyser les logs.



Corrélation des messages

- Ajouter les mécanismes permettant d'associer messages de requêtes et messages de réponses (principe de corrélation des EIP).

Développement d'un client avec l'API Camel

Emission de la requête : `ProducerTemplate`

Camel propose une classe `ProducerTemplate` permettant de créer des échanges entre des EndPoints.

Un `ProducerTemplate` peut être obtenu en invoquant la méthode `createProducerTemplate()` du `CamelContext`.

```
ProducerTemplate pt = context.createProducerTemplate();
```

De là, on dispose de diverses méthodes pour créer des échanges (In-Only ou In-Out).



```
pt.sendBody("jms-test:fournisseur.requestQueue", "Article_1");
```

Réception de la réponse : l'API Endpoint

Il est possible, à partir du `CamelContext`, d'obtenir une référence sur n'importe quel `EndPoint` connu du contexte. Il suffit d'utiliser la méthode `getEndPoint()`.

```
JmsEndpoint responseEndPoint = (JmsEndpoint)
    context.getEndpoint("jms-test:fournisseur.responseQueue");
```

A partir de cet objet `EndPoint`, on peut émettre ou recevoir des messages. Une façon simple de recevoir des messages est de créer un `Consumer` :

```
JmsConsumer consumer = responseEndPoint.createConsumer(new Processor() {
    public void process(Exchange e) throws Exception {
        System.out.println("Réponse reçue : " + e.getIn().getBody());
    }
});
```

La javadoc de l'API Camel-JMS est disponible à l'adresse :
<http://camel.apache.org/maven/current/camel-jms/apidocs/>

Vous trouverez également des informations utiles ici : <http://camel.apache.org/jms.html>

Développement d'un nouveau client

- Développer un client du service en utilisant les API camel.
- Tester ce nouveau client.
- Ce client assure-t-il la corrélation entre réponses et requêtes ?

Exercice 5 : Configuration XML avec Camel

Jusqu'à présent, les routes Camel étaient déclarées directement dans le code Java. On utilisait pour cela le Java DSL.

Pour plus d'information sur la syntaxe Java DSL :

http://fusesource.com/docs/router/1.5/defining_routes/FMRS.BJDS.html

Nous allons maintenant externaliser cette déclaration des routes dans un fichier XML. Ce fichier XML sera au format Spring (<http://www.springframework.org>).

Préparation de l'environnement

- Créer un nouveau projet Java.
- Ajouter les librairies nécessaires.

Développement des étapes de la route

- Créer une classe pour chaque processor déclaré dans la route de notre système
 - Une pour le log
 - Une pour l'appel au service

Il n'est pas nécessaire d'implémenter l'interface Processor pour Ces classes. La seule contrainte est de placer le traitement des messages dans une méthode publique, sans valeur de retour, et prenant une instance de Exchange en paramètre.

Configuration XML

- Créer un fichier nommé camel-server.xml dans le répertoire src. Le contenu de ce fichier est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context.xsd
         http://camel.apache.org/schema/spring
         http://camel.apache.org/schema/spring/camel-spring.xsd">

  <camel:camelContext id="camel">
    <camel:route>
      <camel:from uri="activemq:fournisseur.requestQueue"/>
```



```

        <camel:bean ref="logger" method="log" />
        <camel:bean ref="serviceProxy" method="call" />
        <camel:bean ref="logger" method="log" />
        <camel:to uri="activemq:fournisseur.responseQueue"/>
    </camel:route>

</camel:camelContext>

<bean id="activemq" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
        <bean class="org.apache.activemq.ActiveMQConnectionFactory">
            <property name="brokerURL" value="tcp://localhost:61616" />
        </bean>
    </property>
</bean>

<bean id="logger" class="com.leuville.camel.Logger" />

<bean id="serviceProxy" class="com.leuville.camel.ServiceProxy" />
</beans>

```

Il s'agit d'un fichier XML au format Spring, incluant également des balises issues des schémas Camel. Le `CamelContext` y est déclaré, ainsi que la route de notre système :

- Entrée par la queue `fournisseur.requestQueue`.
- Appel au processor de log (méthode `log`).
- Appel au processor invoquant le service (méthode `call`)
- Post de la réponse dans la queue `fournisseur.responseQueue`.

Afin que le système JMS soit reconnu, le composant Active MQ doit être déclaré. C'est l'utilité du bean « `activeMQ` ».

Les deux beans « `logger` » et « `serviceProxy` » décrivent des instances des deux classes processor.

Développement du lanceur de serveur

- Créer la classe `Server`.
- Y ajouter une méthode `main` :
 - Récupérant le `ApplicationContext` Spring à partir du fichier XML.
 - Récupérant le `CamelContext`.
 - Démarrant le `CamelContext`.

```

ApplicationContext appContext = new ClassPathXmlApplicationContext("camel-server.xml");
CamelContext context = (CamelContext) appContext.getBean("camel");
context.start();

```

Test du système

- Lancer la classe `Server`.
- Utiliser un des clients du TP précédent pour constater que le système fonctionne de manière identique.

Il existe de nombreuses balises XML que l'on peut utiliser pour définir une route. Chaque balise participe à la définition d'un Enterprise Integration Pattern.

La liste complète des EIP implémentables avec Camel et la configuration XML associée est disponible à l'adresse <http://camel.apache.org/enterprise-integration-patterns.html>.