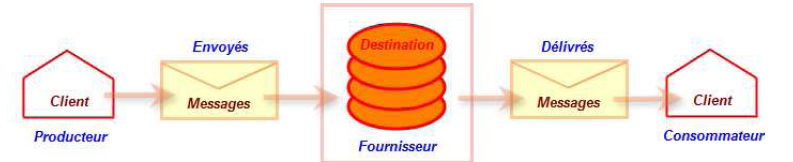
# Programmation EJB3 JMS sous JBOSS

## Introduction

Dans ce sujet, vous allez découvrir **une communication point à point**  entre un producteur de messages et un consommateur sous le serveur d’application WidFly 8.1



**JMS, (Java Message Service),** est la spécification de Sun pour la gestion des messages de la plate-forme Java EE (Nouvelle dénomination de J2EE). En peu de mots, JMS est aux Middleware Orientés Message ce que JDBC est aux bases de données relationnelles.

Un **fournisseur** : (**Provider**) : c'est l'élément qui a la charge de la livraison des messages entre les différents intervenants. Il s'occupe de traiter les envois et de faire en sorte qu'ils soient bien reçus. Il s'agit d'un service qui implémente l'API JMS pour échanger les messages entre deux clients.

2. Un **client** : c'est une application ou un composant d'application intervenant lors des échanges. Il envoie ou reçoit les messages. Il s'agit d'une classe Java qui utilise JMS pour émettre et/ou recevoir des messages. Un client envoie un message vers une file d'attente, et le client destinataire reste à l'écoute d'une file d'attente pour recevoir le message. Le transfert du message et sa persistance sont assurés par le fournisseur.

3. Un **message** : c'est, comme son nom l'indique, l'élément qui va transiter via une communication entre les clients. Un fournisseur sert toujours d'intermédiaire. Nous ne les envoyons donc pas directement d'un client à un autre. Un message est un ensemble de données échangées de manière asynchrone entre les composants. Il existe plusieurs types de messages (texte, objet, binaire, etc.).

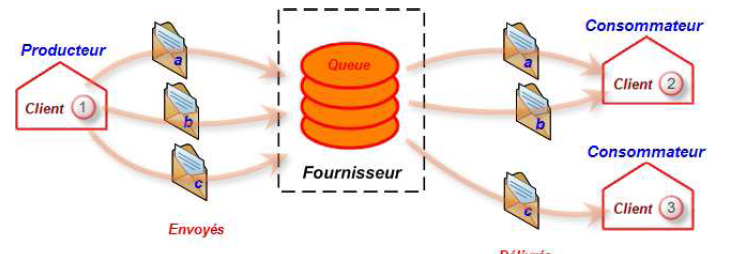
4. Les destinations : ce sont des objets configurés au niveau du fournisseur qui sont à disposition des clients et qui seront utilisés par ces derniers pour l'envoi et la réception des messages. Pour schématiser, nous pouvons dire qu'il s'agit de boîtes à lettres dans lesquelles sont placées les messages en attendant qu'un client vienne les réclamer. Ce sont des ressources à rechercher dans l'annuaire JNDI du fournisseur.

Ce TP illustre le mode **point à point** qui utilise les files d'attente (**javax.jms.Queue**) pour communiquer. Ce mode (un émetteur, un récepteur) s'apparente à l'envoi d'un e-mail.

## Mode point à point : Concept de QUEUE/File de message

Le mode point à point repose sur le concept de files d'attente (Queue). Cela signifie que chaque message est envoyé par un producteur dans une file d'attente, et est reçu par un seul consommateur. Une fois le message consommé, il disparaît de la file d'attente. Dans ce principe, les messages sont envoyés et empilés au fur et à mesure.

Lorsque l'application cliente consommatrice est libre, elle reçoit ainsi l'ensemble des messages empilés.



**Remarque** : Tant qu'un message n'est pas consommé, ou qu'il n'a pas expiré, il reste stocké au sein du fournisseur. Dès que le client devient actif, il peut alors consulter le message qui lui était destiné, et ceci sans aucun problème. Ceci peut se faire à tout moment. C'est vraiment le même principe que la messagerie.

Les classes que nous allons utilisées sont :

-QueueConnectionFactory

- Queue

- QueueConnection

- QueueSession  transaction

- QueueSender / QueueReceiver / QueueBrowser

## Principe de fonctionnement d’un client JMS

Tout client JMS doit satisfaire les étapes suivantes :

* Trouver l’objet ConnectionFactory via JNDI (Java Naming and Directory Service)
* Trouver un ou plusieurs objets destination via JNDI
* Créer une connexion JMS
* Créer 1 ou plusieurs Sessions avec la connexion JMS
* Créer le(s) MessageProducer/MessageConsumer en précisant Session/Destination
* Demander à la connexion de démarrer la livraison des messages.
* Créer des messages et les envoyer et/ou recevoir des messages et les traiter.

### ConnexionFactory

Cet objet représente le point d’accès à 1 serveur **MOM** accessible via JNDI et enregistré par

l’administrateur. sous la forme d’une Queue. (fichier XML à déployer sous Jboss)

// on récupère le contexte du serveur Jboss

Context ctx=JBossContext.*getInitialContext*();

// on crée le point d’accès

*connectionFactory* = (ConnectionFactory)ctx.lookup("ConnectionFactory");

### Connection

Cet objet encapsule la liaison socket/RMI/http/… avec le protocole JMS. Il assure l’authentification du client (login/mot de passe). Il crée les sessions et fournit des métas données sur le provider et gère l’exceptionListener qui permet de récupérer les exceptions de façon asynchrone

// Crée la connexion au serveur via le protocole JMS

connection = *connectionFactory*.createConnection();

### Destination

Cet objet accède via le service JNDI à la queue définie par le serveur MON

// accès à la queue définie par le serveur MON

Destination d = (Destination)ctx.lookup("queue/uneQueue") ;

### Session

Cet objet assure le traitement des transactions :

* contexte mono-threadé
* fabrique les producteurs/consommateurs
* fabrique les destinations temporaires
* gère les numéros de série des messages (ordre par session et par destination)
* gère les transactions
* acquittement des messages (automatique ou manuel)

Session session = c.createSession(boolean transaction, int acquittement)

//acquittement : Session.AUTO\_ACKNOWLEDGE

session = connection.createSession(**false**, Session.*AUTO\_ACKNOWLEDGE*);

### Message Producer

Cet objet représente le producteur du message.

// production de message

messageProducer.send(unmessage);

### Message consumer

Cet objet représente le consommateur de messages. La réception peut être par rapport à la connexion  :

* Synchrone
* Asynchrone

#### Réception synchrone

messageConsummer = session.createConsumer(d);

connection.start();

// bloque la réception pendant 1000 ms

message = (TextMessage)messageConsummer.receive(1000);

**while** (message != **null**) {

System.*out*.println("Message reçu : " + message.getText());

message = (TextMessage)messageConsummer.receive(1000);

}

Message receive() ;//bloquante

Message receive(long timeout) ;//bloque pendant timeout ms

Message receiveNoWait() ;//Non Bloquant

#### Réception asynchrone

Pour cela, l'application doit appeler lala méthode onMessage() de l'interface javax.jms.MessageListener. Celle-ci est très spécialisée et permet ainsi la réception asynchrone des messages. Le développeur doit implémenter cette interface pour réaliser le traitement adéquat lors de la réception d'un message.

# Exemple d’application synchrone : Modepoint à point avec une Queue

Cette application est développée sous

|  |  |
| --- | --- |
| * **Intellij 2016.2.3** | * **Widfly 10.0** |

## Configuration du service JMS de WidFly 10.0

Contrairement à Jboss 6, Widfly n’installe pas le service JMS par défaut. Il faut le configurer dans le fichier standalone.xml en ajoutant les balises suivantes :

### Création de la queue sous le serveur WidFly 12.0

### Extension

<extension module="org.wildfly.extension.messaging-activemq"/>

### subsystem xmlns

<subsystem xmlns="urn:jboss:domain:messaging-activemq:3.0">

<server name="default">

<security-setting name="#">

<role name="guest" send="true" consume="true" create-non-durable-queue="true" delete-non-durable-queue="true"/>

</security-setting>

<address-setting name="#" dead-letter-address="jms.queue.DLQ" expiry-address="jms.queue.ExpiryQueue" max-size-bytes="10485760" page-size-bytes="2097152" message-counter-history-day-limit="10"/>

<http-connector name="http-connector" socket-binding="http" endpoint="http-acceptor"/>

<http-connector name="http-connector-throughput" socket-binding="http" endpoint="http-acceptor-throughput">

<param name="batch-delay" value="50"/>

</http-connector>

<in-vm-connector name="in-vm" server-id="0">

<param name="buffer-pooling" value="false"/>

</in-vm-connector>

<http-acceptor name="http-acceptor" http-listener="default"/>

<http-acceptor name="http-acceptor-throughput" http-listener="default">

<param name="batch-delay" value="50"/>

<param name="direct-deliver" value="false"/>

</http-acceptor>

<in-vm-acceptor name="in-vm" server-id="0">

<param name="buffer-pooling" value="false"/>

</in-vm-acceptor>

<jms-queue name="ExpiryQueue" entries="java:/jms/queue/ExpiryQueue"/>

<jms-queue name="DLQ" entries="java:/jms/queue/DLQ"/>

<jms-queue name="FileMessages" entries="java:jboss/exported/jms/queue/FileMessages"/>

<connection-factory name="InVmConnectionFactory" entries="java:/ConnectionFactory" connectors="in-vm"/>

<connection-factory name="RemoteConnectionFactory" entries="java:jboss/exported/jms/RemoteConnectionFactory" connectors="http-connector"/>

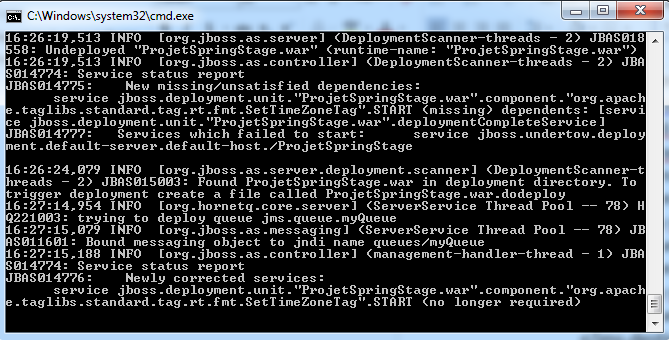
<pooled-connection-factory name="activemq-ra" entries="java:/JmsXA java:jboss/DefaultJMSConnectionFactory" connectors="in-vm" transaction="xa"/>

</server>

</subsystem>

Nous pouvons à présent créer une queue sur le serveur  en suivant les étapes suivantes :

### On lance le serveur avec standlone



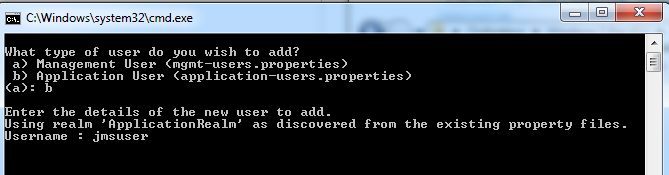
|  |
| --- |
| On lance un client jboss-cli On se connecte au serveur    On exécute la commande suivante : Ajout d’une file via la console ( à ne pas faire pour ce tp ) La file est déjà insérée dans le fichier standalone.xml, code que vous venez d’insérer.  [standalone@localhost:9990 /]  **jms-queue add --queue-address=FileMessages --entries=** **java:jboss/exported/jms/queue/FileMessages**    On peut contrôler l’ajout dans le fichier Standalone.xml |
|  |

Notre file est à présent opérationnelle pour envoyer ou consommer des messages.

#### Suppression d’une file

**jms-queue remove --queue-address=FileMessages**

### Création d’un utilisateur : jmsuser/ jmsepul98! add-user

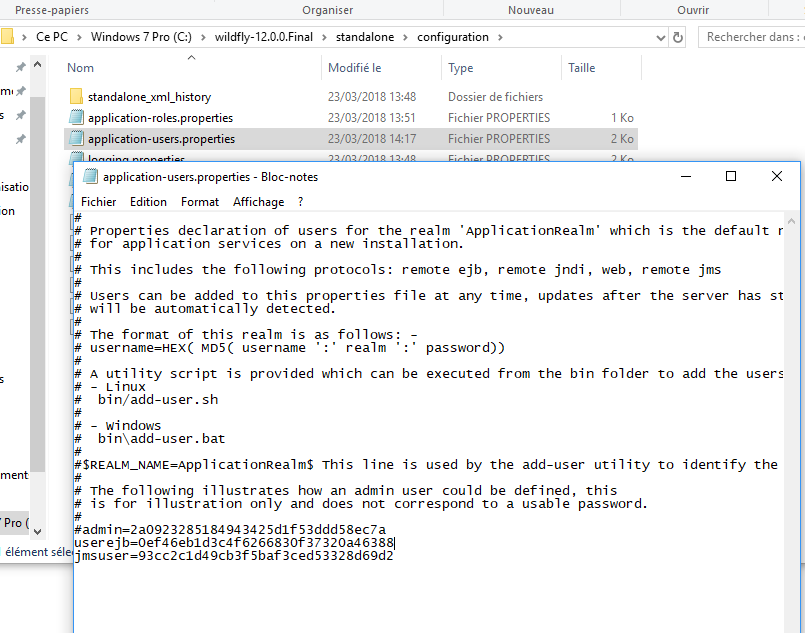


Ajoutez-le dans le groupe guest

Le fichier application-roles sera mis à jour.

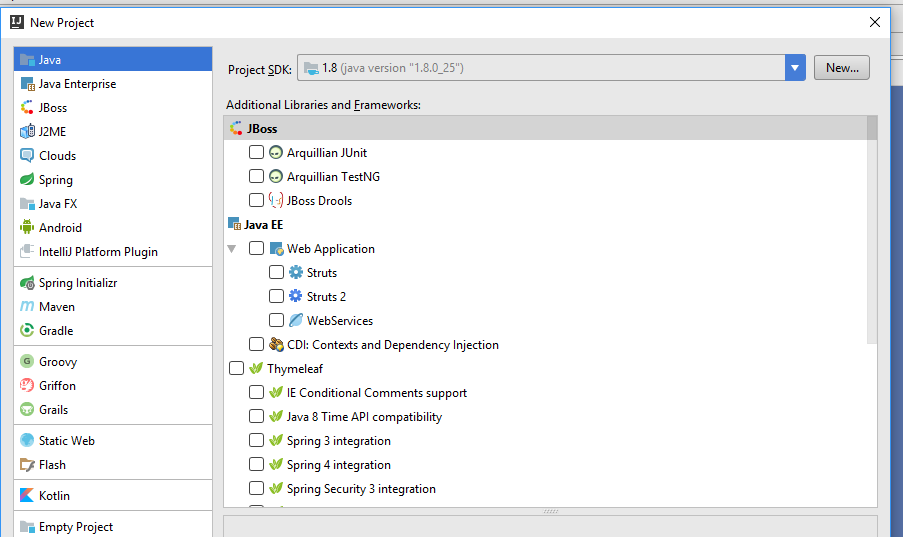
Information

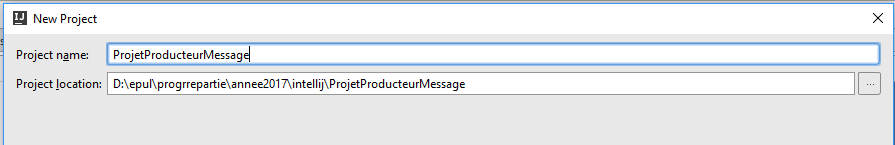
Pour supprimer un utilisateur, il faut aller dans le fichier application-users.properties



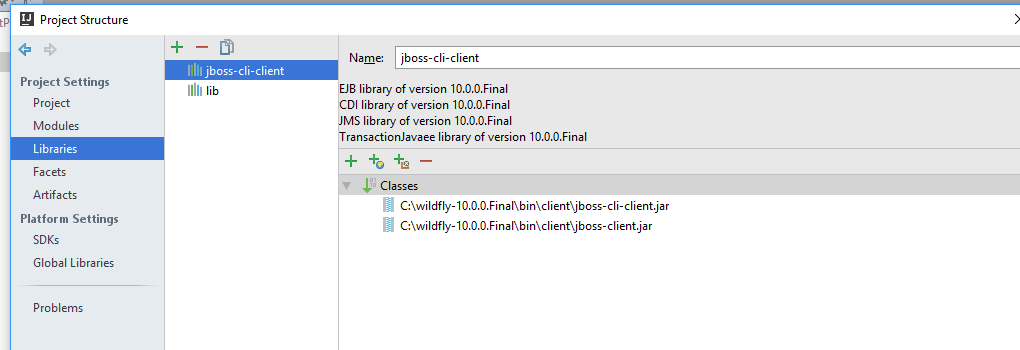
## Création du client Producteur

Sous Intellij, créez un projet ProjetProducteurMessage de type java



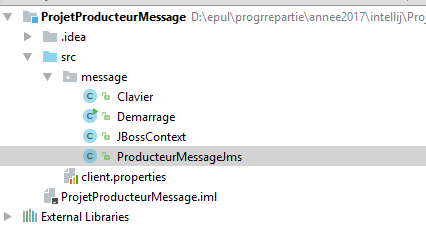


Ajoutez les librairies suivantes à votre projet



Ajoutez un package nommé message

**Code de la classe Producteur**



Vous devez rajouter à votre projet ces fichiers qui sont présents sous Spiral

#### Code du client.properties ( non copiable sous Spiral)

remote.connections=default

remote.connection.default.host=127.0.0.1

remote.connection.default.port=8080

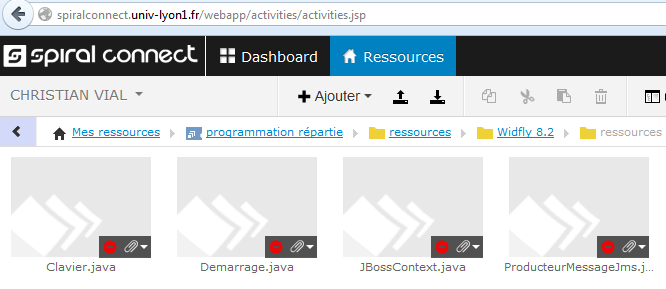
remote.connection.default.connect.options.org.xnio.Options.SASL\_POLICY\_NOANONYMOUS=false

remote.connection.default.connect.options.org.xnio.Options.SASL\_POLICY\_NOPLAINTEXT=false

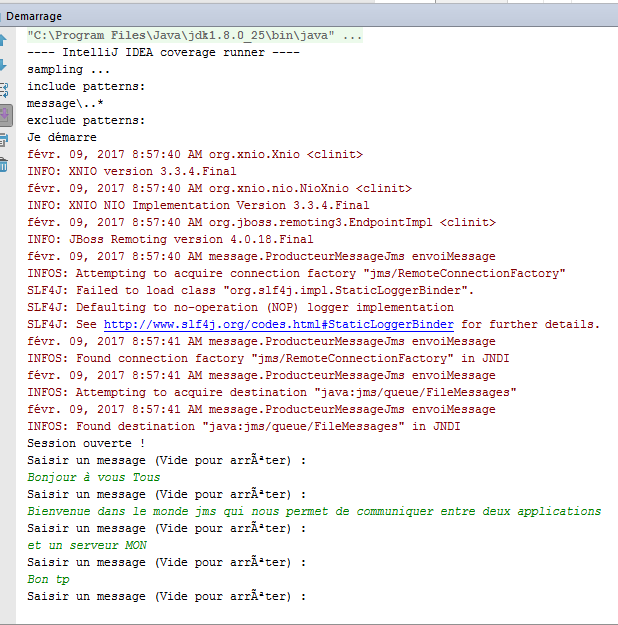
remote.connection.default.connect.options.org.xnio.Options.SASL\_DISALLOWED\_MECHANISMS=${host.auth:JBOSS-LOCAL-USER}

remote.connection.default.username=jmsuser

remote.connection.default.password=jmsepul98!



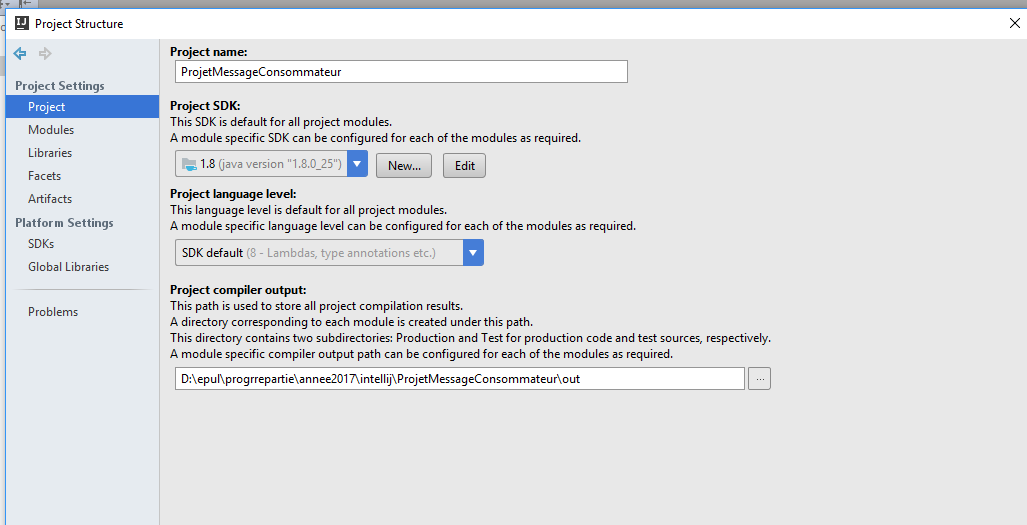
**cution de la production de messages**



Les messages sont envoyés à la file nommée FileAttenteJMS. Ils peuvent être alors consommés par un client.

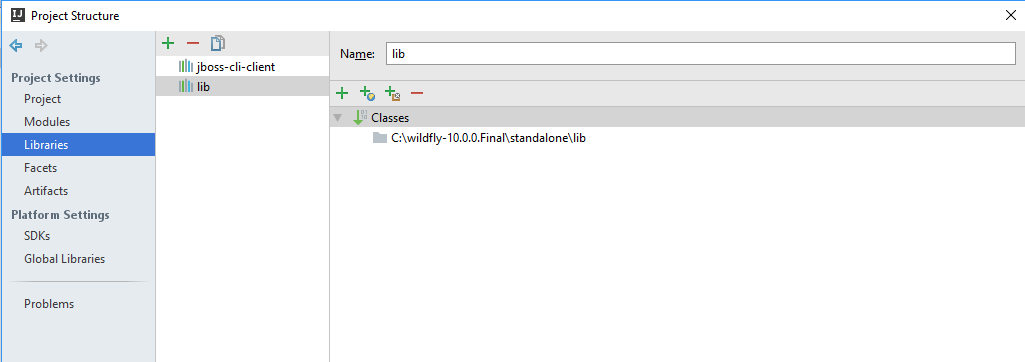
# Création du client Consommateur

Projet de type Java Application

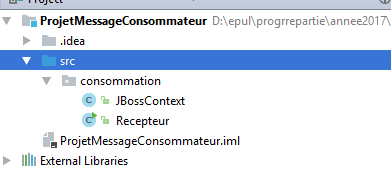


Pour réaliser cette partie Client, il suffit de créer un projet de type java Application avec un package nommé consommation

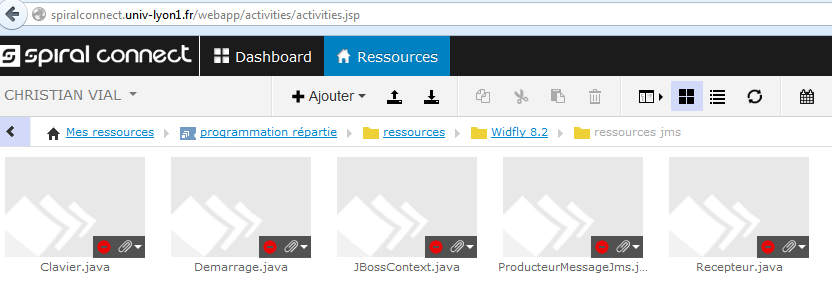
Vous devez ajouter les bibliothèques suivantes



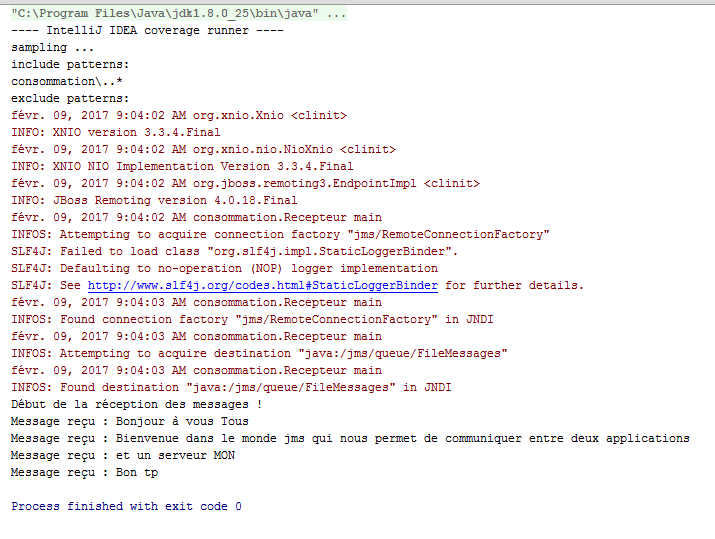
## Structure de l’application



Les fichiers sont sous Spiral dans :



**Exécution de la partie Client Consommateur**



Notre client consomme bien les messages envoyés par le producteur.

# Application 2 : Envoyer une image avec une Queue

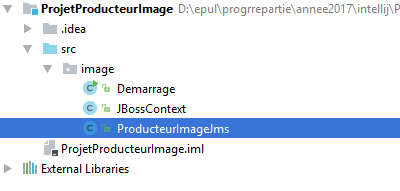
## Ajout d’une queue dans StandAlone.xml

Les images seront stockées dans une file nommée demoQueue

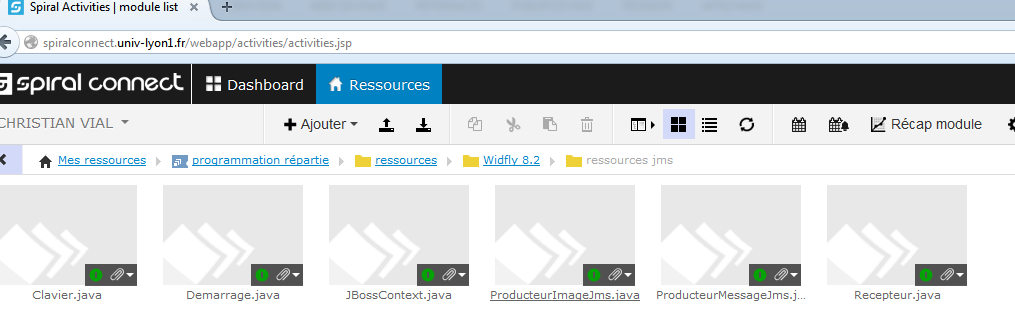


## Création du client Producteur

Sous Intellij créez un projet de type Java Projet dont l’architecture est :

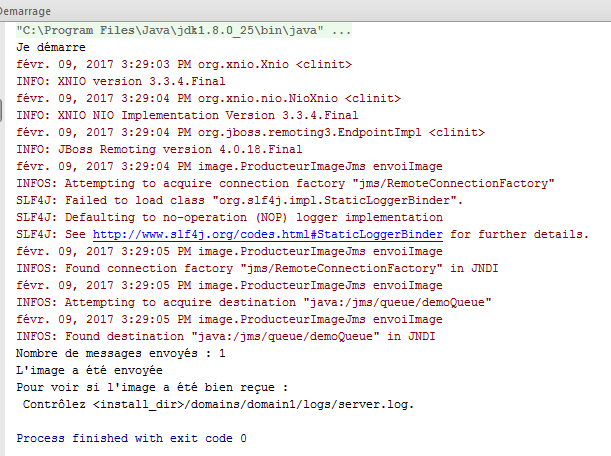


Les fichiers sont présents sous Spiral



### Démarrage de l’application

**Exécution de la production de l’envoi d’une image**

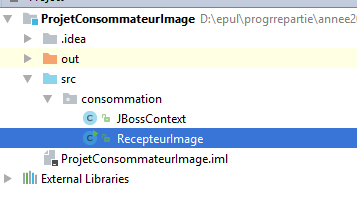


Les messages sont envoyés à la file nommée demoQueue. Ils peuvent être alors consommés par un client.

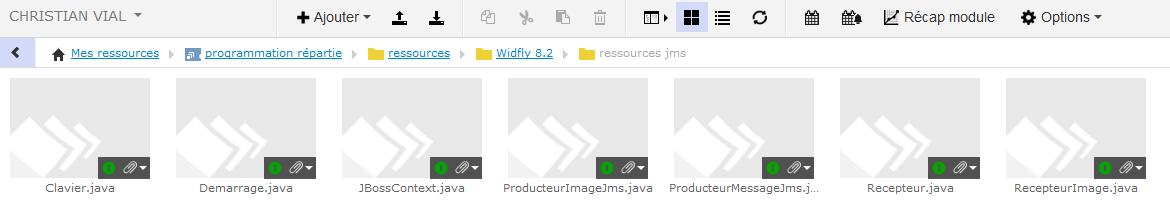
## Création du client Consommateur

Pour réaliser cette partie Client, il suffit de créer un projet de type java Application avec un package nommé consommateur. Cette classe va exécuter un processus pour visualiser l’image reçue. Dans notre exemple, nous prendrons **PictureViewer** de QuickTime.

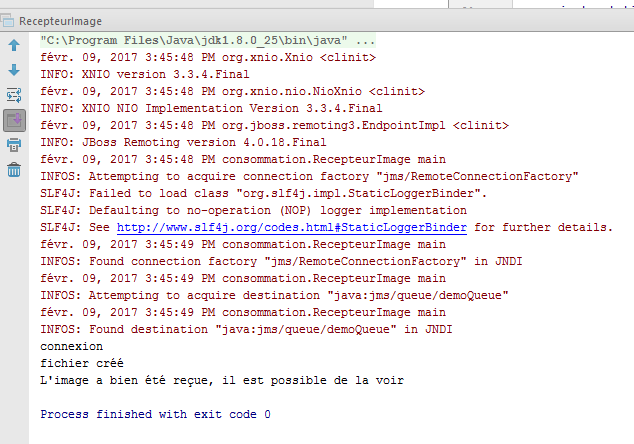
### Architecture de l’ application



Les fichiers sont sous Spiral



#### Exécution de la partie Client Consommateur : Lecture et affichage de l’image



Notre client consomme bien les images envoyées par le producteur.

|  |  |
| --- | --- |
|  |  |