LO54

Rapport de projet

JMS – ActiveMQ

FAVRY – PROST

Table des matières

Index des illustrations

Introduction

Le présent rapport a pour objectif la restitution de l'expérience que nous avons vécue tout au long de ce projet de LO54, portant sur le développement d'une application en reprenant les éléments vus en cours, en travaux dirigés et en travaux pratiques.

L'application que nous devions développer était une application web permettant la gestion d'inscriptions à des cours. Elle devait donner la possibilité de rechercher un cours en particulier suivant différents critères (mots-clés, dates, lieux) et devait permettre à un utilisateur de s'inscrire aux différents cours. La persistance des données devait être gérée par Hibernate et la structure globale du projet devait respecter l'architecture SOA.

La particularité de notre groupe comparé aux autres était que nous devions intégrer une partie notification, qui enverrait un message lors de chaque inscription à une autre application, et ce en utilisant le Java Message Service (JMS).

Nous allons donc, dans ce rapport, commencer par décrire la technologie JMS en termes de fonctionnalité, d'utilité et de mise en place, puis nous décrirons l'application que nous avons développée.

Java Message Service (JMS)

Description

Le *Java Message Service* (abrévié en JMS) est une API créée en 1998 par *Sun Microsystem* (qui appartient désormais à *Oracle*), API permettant la transmission de messages entre applications ou parties d'applications, et fait partie de *Java Enterprise Edition* depuis la version 1.3, cette dernière étant la première à inclure une implémentation.

JMS fait partie des API de JEE à avoir le moins changé dans l'histoire de JEE, puisque seulement 3 versions différentes ont existé depuis 2001.

Système de transmission de messages

Comme nous venons de le dire, JMS permet la transmission de messages en proposant des services de création, d'envoi, de réception et de lecture de messages. La structure de transmission ainsi mise en place est une structure de type *Message-Oriented Middleware* (MOM). Cette architecture permet une transmission *asynchrone* ainsi qu'un faible couplage entre les émetteurs et les récepteurs.

Dans quel(s) cas l'utiliser ?

Le *Java Message Service* est particulièrement adapté aux communications entre applications où la notion de transmission asynchrone est utile, c'est-à-dire où le récepteur et l'émetteur peuvent être connectés à des moments différents.

De plus, grâce au faible couplage que nous offre la structure MOM, les applications dont l'exécution est extrêmement courte ne seront pas désavantagées, puisque la connexion/déconnexion de l'émetteur est rapide, et est surtout indépendante de celle du ou des récepteurs potentiels.

Types de communications

Avec l'architecture *Message-Oriented Middleware*, JMS permet de mettre en place deux types de communications, que nous allons maintenant présenter.

Chacune des interfaces d'entité de JMS (Client, Broker, etc.) possède deux sous-interfaces, une pour chaque type de communication.

File de messages

Le premier type de communication est le type *File de messages*, qui est une communication point-à-point (un seul émetteur et un seul récepteur) suivant un patron de P*roducteur-Consommateur*.

L'envoi de messages via ce type de communication-ci aura pour effet de placer le message en question dans une file d'attente PEPS (*Premier Entré, Premier Sorti)* où il restera pour une durée indéterminée. À tout moment, un consommateur va pouvoir venir lire un message dans cette file de messages, ce qui aura pour effet de le *consommer*, c'est-à-dire le retirer de la file. C'est pour cela qu'on parle de transmission *1-à-1* : un seul récepteur peut recevoir ce message, car la lecture retire le message de la file, rendant le message indisponible pour d'autres.

La file de messages oblige les consommateurs à venir lire périodiquement la file pour vérifier la présence de nouveaux messages. Cependant, il existe un mécanisme d'attente qui permet au consommateur de se mettre en attente passive jusqu'à l'arrivée d'un nouveau message à consommer dans la file sur laquelle il est en attente.

Publications

Le second type de communication proposé par JMS correspond à un patron *Publication* aussi appelé par son nom anglais *Publish-Subscribe*, et correspond à un mode de transmission de diffusion *1-à-plusieurs*, en se basant sur un système de sujets (*topics*) pour déterminer les destinataires.

Avant la diffusion d'un message, les différents récepteurs potentiels vont contacter le *broker* pour s'enregistrer sur un ou plusieurs sujets (*subscribe*). Lorsqu'un client souhaite envoyer un message, il va alors choisir un sujet sur lequel le publier (*publish*), et le *broker* va alors se charger de le transmettre à tous les clients qui sont enregistrés sur ce sujet. Ils vont alors tous recevoir le message à l'identique, d'où la communication *1-à-plusieurs*.

Contrairement à la communication par file de messages, le modèle P*ublish-Subscribe* se base sur des événements pour la réception des messages, et les différents récepteurs peuvent définir des callbacks pour la réception des messages.

Messages

Le *Java Message Service* propose également, en plus des différents types de communication, de créer vos propres messages. Nous allons voir maintenant quelles sont les différentes caractéristiques de ces messages, puis comment les générer, les envoyer et les recevoir.

Structure

Les messages de JMS sont composés de 3 parties distinctes : les en-têtes, les propriétés et le corps du message.

En-tête

Les en-têtes sont des champs généraux communs à tous les messages, et donnant les informations de base à propos de ceux-ci (destination, mode de communication, identifiant, horodateur, durée de vie, etc.). La grande majorité des champs sont remplis automatiquement par JMS lorsque le message est envoyé, aussi les modifier à la main avant l'envoi peut ne pas avoir d'effet.

Les en-têtes sont prédéfinis dans JMS, il n'est pas possible d'en rajouter un.

Propriété

Ensuite, les propriétés sont également des champs, mais contrairement aux en-têtes, les propriétés ne sont pas obligatoires et uniques à chaque message. Elles servent à spécifier des métadonnées supplémentaires ne faisant pas partie du contenu, ce qui peut par exemple permettre de filtrer les messages d'une certaine manière.

Les propriétés sont libres dans JMS, chacun pouvant définir ses propres propriétés comme bon lui semble. Les messages peuvent contenir plusieurs propriétés.

Corps

Enfin, le corps de message est le contenu *réel* du message. Il va définir le type de message qui sera envoyé, et correspond dans l'API à des interfaces différentes, chacune permettant l'envoi d'un certain type de données. JMS propose 5 types de corps de message différents :

- TextMessage
- MapMessage
- BytesMessage
- StreamMessage
- ObjectMessage

Chaque type est destiné à contenir certaines données, par exemple une chaîne de caractères pour le TextMessage, un set d'associations *clé-valeur* pour le MapMessage ou encore un objet sérialisable pour l'ObjectMessage.

Un message donné ne peut contenir au maximum qu'un seul corps, mais peut cependant être créé sans celui-ci : JMS propose en effet un 6ème type, le Message, qui est un message ne contenant pas de corps, et étant seulement composé d'en-têtes et de propriétés.

Mettre en place une communication simplement

Créer le client

Pour mettre en place un client capable d'envoyer et de recevoir des messages, il faut instancier 3 éléments, mais les interfaces diffèrent suivant le type de communication (*pub-Sub* ou *file de messages*) que vous utilisez. La table ci-contre donne le nom des 3 interfaces à instancier en fonction du type de communication :

Communication *Pub-Sub*	Communication *Queue*
TopicConnectionFactory	QueueConnectionFactory
TopicConnection	QueueConnection
TopicSession	QueueSession

Les interfaces *ConnectionFactory* sont les premiers éléments à instancier, et vont nous permettre de créer des instances *Connection*. La *ConnectionFactory* recevra, à son instanciation, les éléments permettant de se connecter au *broker* (l'URL de connexion, éventuellement des identifiants), et chacune des instances *Connection* que nous produirons grâce à cette *ConnectionFactory* seront configurés pour contacter ce *broker*-ci en particulier.

Ensuite, la *Connection* est une instance qui va représenter la connexion entre le client et le *broker* pour lequel elle est configurée. Instancier et démarrer une instance *Connection* a pour effet de se connecter physiquement au *broker* via la création d'un socket TCP/IP. En outre, l'instance *Connection* permet de créer des instances *Session*.

Cette instance *Session* est un objet mono-thread représentant un contexte de communication entre une application et le *broker* JMS, permettant la création et la consommation de messages. La session contient également d'autres éléments de configuration, comme la politique d'accusés réception (automatique lors de la réception, manuelle, etc.).

```
//Instantiation de nos objets ConnectionFactory et Connection
TopicConnectionFactory connectionFactory = new ActiveMQConnectionFactory( brokerURL: "tcp://localhost:10000");
TopicConnection connection = connectionFactory.createTopicConnection();

//Ouverture de la connexion
connection.start();

//Création de la Session
TopicSession session = connection.createTopicSession( b: false, Session.AUTO_ACKNOWLEDGE);

//Création d'un publicateur
Topic topic = session.createTopic( s: "monTopicPubSub");
TopicPublisher publisher = session.createPublisher(topic);
```

Illustration 1: Exemple de code pour la mise en place d'un publicateur JMS

La figure ci-contre présente le code nécessaire pour instancier tous ces objets pour une communication de type *Pub-Sub*, démarrer la connexion avec le *broker* JMS et créer un *publisher* sur le sujet *monTopicPubSub*. Comme précisé plus tôt, nous utilisons l'implémentation *ActiveMQ* créée par Apache.

Pour mettre en place un *subscriber*, il suffit de remplacer d'appeler la méthode *session.createSubscriber(topic)*. Vous devez également spécifier un MessageListener qui sera appelé lors que la réception de messages.

Mettre en place le *middleware*

La mise en place d'un *middleware* est très simple avec ActiveMQ. Il suffit de spécifier un nom au broker, de lui rajouter des connecteurs, c'est-à-dire des sockets sur lesquels les clients peuvent se connecter, et de le démarrer. Un exemple de cette séquence est disponible ci-dessous :

```
BrokerService broker = new BrokerService();

broker.setBrokerName("lo54_jms_broker");
broker.addConnector( bindAddress: "tcp://localhost:10000");
broker.start();
```

Illustration 2: Exemple de code pour la mise en place d'un broker ActiveMQ

Envoyer et recevoir des messages

Dernière étape, l'envoi de message, qui peut se faire une fois que le *publisher* est instancié. Pour se faire, le code suivant peut être utilisé :

```
/* ... instanciation de obj ...  */

//Création d'un ObjectMessage
ObjectMessage msg = session.createObjectMessage();
msg.setObject(obj);

//Publication du message
publisher.publish(msg);
```

Illustration 3: Exemple de code pour l'envoi d'un message

Rapport d'expérience

Pour rappel, nous devions pour ce projet développer une application web permettant l'inscription à des cours ainsi que la recherche de cours via différents critères (mots-clés, date et lieu), tout en incluant un système de notifications d'inscription en utilisant *Java Message Service*.

Nous allons donc décrire l'application que nous avons développée sous 3 angles : tout d'abord l'architecture que nous avons créée, puis les différentes interfaces, pour finir par expliquer plus particulièrement les notifications via JMS.

La base de données devant être utilisée par l'application web était donnée, nous n'avions donc pas à la concevoir.

Architecture globale

L'architecture que nous avons mise en place correspond à une architecture N-tiers *Service Oriented* (SOA), avec différentes couches logicielles :

Couche persistance

La couche de persistance est la couche où nous assurons la persistance des différentes données. Cette persistance est gérée par l'ORM *Hibernate*, et nous utilisons le principe des *java beans* pour matérialiser logiciellement les éléments des tables de la base de données qui nous ont été fournis. Des annotations ont été utilisées pour effectuer le lien entre une table et son *bean*. Les clés étrangères dans les tables ont été configurées comme des relations *many-to-one* dans les *beans*, avec un *lazy-loading* et une propagation des modifications.

```
@ManyToOne(cascade = CascadeType.ALL, fetch=FetchType.LAZY)
@JoinColumn(name="COURSE_CODE")
private Course course;

@ManyToOne(cascade = CascadeType.ALL, fetch=FetchType.LAZY)
@JoinColumn(name="LOCATION_ID")
private Location location;
```

Illustration 4: Extrait de la classe CourseSession montrant les liens many-to-one mis en place pour les clés étrangères

Couche traitements

La couche traitement contient les classes de *dao*, qui sont au nombre de cinq dans notre application, et qui implémentent toutes l'une des deux interfaces suivantes :

- IRepoDao : cette interface correspond aux dao chargés de gérer l'accès aux données de la base de données, et possède une implémentation par *javabeans* existant dans le projet. Ces implémentations utilisent toutes *Hibernate* directement pour récupérer les données en créant des transactions.
- IPublisherDao : cette interface définit les méthodes pour gérer un *publisher* pouvant envoyer des chaînes de caractères et des objets sérialisables sur un sujet (*topic*) prédéfini.

  Nous avons également créé une classe JmsDao, qui est une implémentation de *IpublisherDao* pour JMS, et utilise ActiveMQ comme implémentation de JMS.

Couche service

La couche service contient les différentes classes de services, qui seront utilisées par les contrôleurs des interfaces. Elles contiennent le code métier et utilisent les classes de *dao* pour réaliser les actions sur les *javabeans* ou pour envoyer des notifications.

Nous avons créé deux classes de services, nommé *CourseService* et *PublisherService*, qui vont utiliser respectivement des instances *CourseDao* et *IPublisherDao* pour le traitement.

Le premier service permet la récupération des sessions de cours enregistrées, des lieux, ainsi que la récupération d'une liste contenant, pour chaque cours, la prochaine session enregistrée.

Le second service contient la méthode pour envoyer la notification d'inscription d'un client donné pour un cours.

Couche contrôleur

Pour la couche contrôleur, nous avons créé 3 pages JSP et 3 *servlets*. Les pages JSP correspondent aux 3 pages différentes de l'interface, que nous détaillerons plus tard. 2 des 3 *servlets* correspondent quant à eux à des messages de succès et d'échec de l'inscription avec un lien vers la page principale, le troisième *servlet* permet d'effectuer l'inscription et d'aiguiller le client vers l'un des deux autres *servlets* suivant le résultat. C'est également ce *servlet*-ci qui va déclencher l'émission de la notification d'inscription dans le cas d'un succès.

Application web

Comme nous l'avons précisé plus tôt dans ce rapport, l'application web est composée de 3 pages JSP et de 3 servlets.

La première des pages, *index.jsp*, contient une liste affichant, pour chaque cours présent dans la base de données, la date et le lieu de sa prochaine session. Elle contient également un bouton pour rejoindre la page suivante, *search.jsp*.

Illustration 5: Aperçu de la page d'accueil

Cette dernière, *search.jsp*, contient un formulaire permettant la spécification des critères de recherche : ce formulaire possède un champ texte pour les mots-clés, un sélecteur de dates pour restreindre la recherche en fonction de la date des sessions et une liste déroulante contenant tous les lieux enregistrés. L'utilisateur peut utiliser tout ou partie de ces éléments pour restreindre la liste des sessions, qu'il peut actualiser en appuyant sur un bouton intitulé « Rechercher » présent dans le formulaire.

En dessous du formulaire se trouve la liste des sessions correspondantes, initialement vide, qui correspondent aux critères spécifiés.

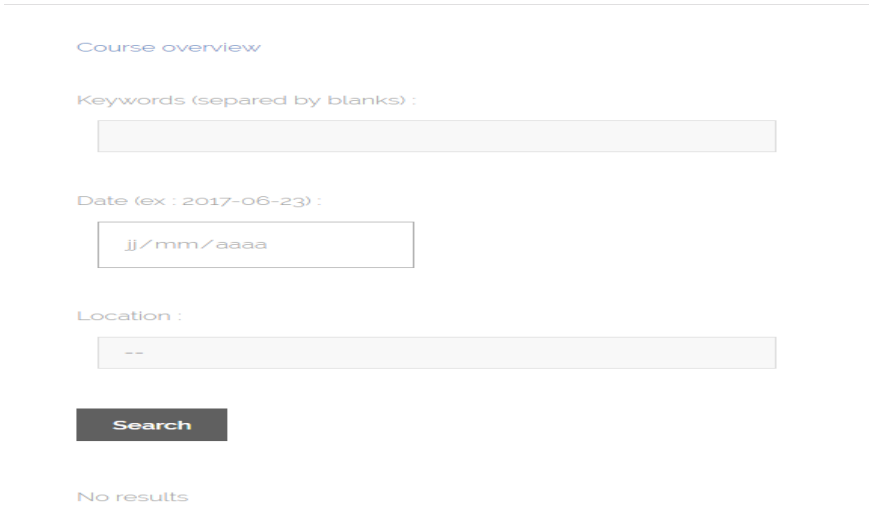

Illustration 6: Aperçu de la page de recherche de sessions de cours

Les éléments de la liste sont des liens vers la troisième page, *register.jsp*, qui contient le formulaire d'inscription à une session, la session ayant été sélectionnée par l'utilisateur. Un rappel des informations de la session est également présent sur la page.

Illustration 7: Aperçu de la page d'inscription à une session de cours

Lorsque l'utilisateur valide l'inscription, le premier servlet va être appelé : *RegistrationServlet*. Comme nous l'avons dit plus tôt, ce *servlet* va tenter d'inscrire le client à la session d'un point de vue logiciel. S'il y parvient, il va alors envoyer la notification d'inscription JMS et va rediriger la requête vers le *servlet* affichant le succès de l'opération via la classe *RequestDispatcher*. Dans le cas d'un échec, aucune notification JMS n'est publiée et la redirection est faite vers le servlet affichant l'échec de l'opération.

Les *servlets* de succès et d'échec affichent un texte similaire, contenant un message décrivant le statut de l'opération d'inscription (succès/échec), accompagné de liens permettant de retourner vers la page *index.jsp*.

Le *servlet* d'erreur affiche en plus la raison de l'erreur, qui peut se trouver en tant que paramètre dans la requête http. La présence de ce paramètre est due au *servlet RegistrationServlet* qui peut, dans certains cas

d'erreur, l'ajouter à la requête avant de la rediriger. Cependant, tous les cas d'erreurs ne sont pas pris en compte, aussi l'affichage du message d'erreur change suivant si la raison de l'erreur est connue ou non.

Intégration JMS

Pour intégrer l'utilisation du *Java Message Service* dans les notifications d'inscription, nous avons mis en place le service de publication décrit précédemment. Ce service, lorsqu'une inscription est faite, se connectera à un *broker* JMS et transmettra un message texte contenant les informations sur l'inscription.

Nous avons choisi le mode de transmission *Publisher-Subscriber* car même si dans le cadre de ce projet nous n'avons qu'un seul destinataire au maximum, il nous semblait plus logique de nous baser sur un système de notifications par événements avec plus d'un *consommateur*. De plus, le nombre de *publisher* n'est lui pas fixé à 1, et on peut très bien avoir plusieurs serveurs web proposant l'application que nous avons développée. Dans un cas de figure comme celui-ci, le client doit pouvoir recevoir des messages de la part de plusieurs *producteurs.*

Côté réception de notifications, nous avons créé une petite application *client lourd* mettant initialisant le *broker* ainsi que le *consommateur* des notifications. Pour le moment, *localhost* est utilisé pour l'adresse du *broker* de manière a éviter de passer par le réseau de l'UTBM lors des démonstrations, mais l'adresse peut être modifiée facilement. Nous avons utilisé JavaFX pour créer son interface, qui est composée uniquement d'une zone de texte non-modifiable affichant les informations et les messages lorsqu'ils arrivent. Attention cependant à bien démarrer en premier le serveur Tomcat au prix d'avoir des conflits de port pour le protocole RMI lancé par Tomcat et ActiveMQ (si on démarre le module et l'application sur la même machine).

Conclusion

Ce projet nous a permis de nous faire la main sur différents aspects de J2EE, certains vu en cours et d'autres que nous avons explorés personnellement. De plus, cela nous a donné l'occasion de nous plonger dans l'API JMS, API que l'un de nous voulais expérimenter et utiliser dans un projet personnel concret.

Cela nous a également permis d'approfondir les connaissances vues en cours, comme l'architecture *SOA* ou encore l'utilisation d'*Hibernate*, ainsi que de maintenir nos compétences dans la manipulation de certains outils que nous avions déjà utilisé par le passé pour divers projets, professionnels ou non, comme par exemple *Maven*.