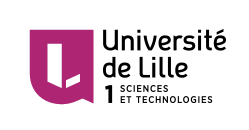
|  |
| --- |
| Datamining d’une architecture orientée service |
| Master 2 IAGL – 2014/2015 |



|  |
| --- |
| Thomas Buissart et Quentin Marrecau  02/12/2014 |

# Introduction

Les Service Oriented Architecture (SOA) sont devenues au fil des années la pierre angulaire des échanges inter-applicatifs. Elles répondent au « syndrome spaghetti », typique des architectures point à point, en proposant une nouvelle façon de développer des applications sous forme de services. Ce procédé consiste à décomposer chaque brique du Système Informatique (SI) en processus métiers, puis à développer des connecteurs (middleware) spécifiques permettant de les faire communiquer entre elles. Ces connecteurs permettent d’interfacer les applications utilisant potentiellement des protocoles ou des langages différents.

De cette façon il devient plus simple d’intégrer une nouvelle fonctionnalité au système existant, simplement en développant un nouveau connecteur. En proposant un couplage faible des applications entre elles, le SI devient plus modulaire, et indépendant des technologies utilisées.

Au-delà de ces points forts, les SOA ont également quelques inconvénients. En effet, le Middleware Oriented Messages (MOM), logiciel chargé de transporter les messages entre les applications, devient parfois similaire à une boîte au contenu inextricable. Des stacktraces parfois trop pauvres en informations et une mauvaise vision sur le déroulement d’un flux empêchent la résolution efficace des bugs.

L’API Java Messaging Service permet à une application Java d’invoquer les services d’un MOM. Une application source appelée « producer » émet un message qui sera consommé par une application cible appelée « consumer ». Les services proposés par le MOM se présentent sous deux formes, correspondant à deux modes de communication différents :

* Le mode point à point ou « queue » : le message émis par le producer n’est consommé qu’une seule fois par un unique consumer.



Queue

* Le mode « publish-subscribe » ou « topic » : le message émis par le producer est consommé par toutes les applications ayant souscris au topic

message

Topic

Les queues et topics sont aussi appelés brokers.

Dans les architectures orientées services, il est commun d’encoder les messages échangés au sein du MOM dans un langage « pivot », généralement XML ou JSON afin de faciliter la communication entre les différentes applications, et pour mieux représenter les données métiers.

Nous illustrerons en première partie les points faibles du MOM concernant la remontée des erreurs pouvant survenir dans le déroulement d’un flux. Dans un second temps, nous proposerons notre solution pour pallier à ces problèmes de manque de remontée d’informations dans les échanges inter-applicatifs. Enfin, nous présenterons nos résultats en s’appuyant sur notre projet sample, modélisant un échange de messages entre plusieurs applications, avec et sans notre solution.

# Présentation d’un cas standard

## Gestion des erreurs

Nous avons utilisé l’API JMS pour implémenter des échanges de messages. Le MOM que nous avons utilisé est la solution open source Apache ActiveMQ. Nous avons développé également des classes SampleProducer et SampleConsumer chargées respectivement de produire et de consommer un message dans les brokers.

### Production et consommation d’un message

Exemple d’envoi d’un message simple dans une queue :

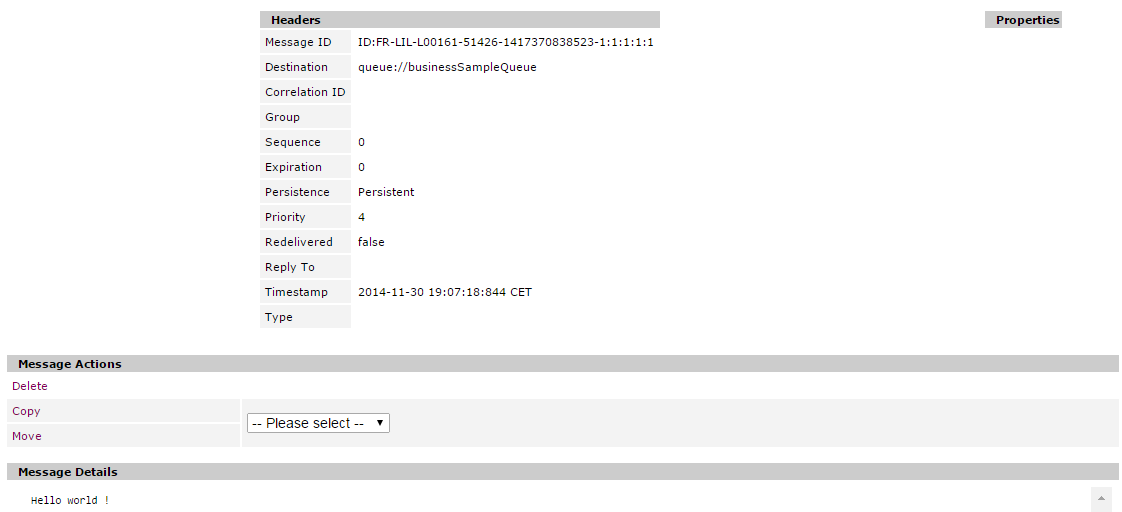
String message = "Hello world !";

SampleProducer producer = **new** SampleProducer();

producer.sendToJMSQueue(message);

simple_send_queue.pngAffichage dans l’interface d’administration d’ActiveMQ :

Détails du message :



Consommation du message :

// récupération de la destination depuis les propriétés

Properties jndiProperties = **new** Properties();

jndiProperties.load(SampleProducer.**class**.getClassLoader()

.getResourceAsStream("jms/jms.properties"));

InitialContext context = **new** InitialContext(jndiProperties);

Destination queue = (Destination) context.lookup("businessSampleQueue");

// création du consumer

SampleConsumer consumer = **new** SampleConsumer((Destination) queue);

// consommation du message

Message received = consumer.consume();

TextMessage receivedTextMessage = (TextMessage) received;

String text = receivedTextMessage.getText();

// affichage du message reçu dans les logs

*logger*.*info*("Received: " + text);

Affichage dans l’interface d’administration d’ActiveMQ après consommation du message : 

Dans la console Eclipse :

## INFO SimpleExchange:52 - Received: Hello world !

### Cas d’une erreur dans la production du message

Le code suivant va générer une exception de type StringIndexOutOfBoundsException :

String message = "Hello world !";

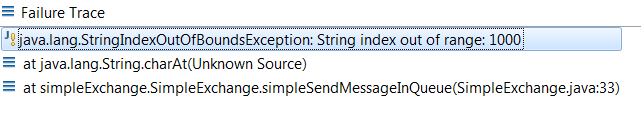
SampleProducer producer = **new** SampleProducer();

// génération d'une exception

message.charAt(1000);

producer.sendToJMSQueue(message);

Nous pouvons observer cette stacktrace dans Eclipse :



Aucun message n’a été envoyé car le programme ……….

Aucun message dans le broker, et consommateur qui n’est pas notifié

## Implémentation de scénarios concrets

Implémentation de scénarios en conditions réelles

Projet businessSample (kézako ? / schémas explicatifs )

Générateur d’erreurs + exemples d’erreurs remontées (ou pas)

Comparatif des erreurs générées et affichées sur la console ActiveMQ

# Améliorer la gestion des erreurs

## Constat

Apporter une solution en conservant l’existant

Faciliter au maximum l’intégration de l’outil. Se concentrer la problématique de la visibilité

* API

## Présentation de la librairie Sentinel

Schémas explicatifs (reprise du schéma du businessSample avec les sondes Sentinel)

# Exploitation des résultats

## Améliorer la visibilité

IHM

Tableaux récapitulatifs des échanges de données ( résultats mis en BDD par l’API )