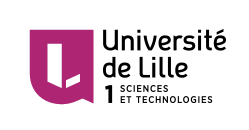
|  |
| --- |
| Datamining d’une architecture orientée service |
| Master 2 IAGL – 2014/2015 |



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

Table des matières

[I. Introduction 4](#_Toc405240114)

[II. Présentation d’un cas standard 6](#_Toc405240115)

[1. Gestion des erreurs 6](#_Toc405240116)

[a. Envoi d’un message 6](#_Toc405240117)

[b. Réception d’un message 7](#_Toc405240118)

[c. Cas d’une erreur dans la production du message 7](#_Toc405240119)

[2. Simulation de conditions réelles 8](#_Toc405240120)

[a. Scénario 8](#_Toc405240121)

[b. Gestion des exceptions 9](#_Toc405240122)

[III. Amélioration de la visibilité des erreurs 10](#_Toc405240123)

[1. Des sondes 10](#_Toc405240124)

[2. Pourquoi des sondes ? 10](#_Toc405240125)

[3. Résultats 11](#_Toc405240126)

[IV. Supervision des erreurs 11](#_Toc405240127)

[1. Sondes et persistances de données 12](#_Toc405240128)

[2. Exploitation des résultats 13](#_Toc405240129)

[V. Conclusion 15](#_Toc405240130)

# 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

Afin de mettre en évidence que les solutions logicielles d’aujourd’hui ne permettent pas un suivi précis des messages et des erreurs, nous avons donc décidé de mettre en place un premier projet, portant le nom de BusinessSample, qui reproduit les conditions réelles d’exécution d’un projet basé sur les technologies SOA suivantes : JMS et ActiveMQ. L’API JMS a été utilisée pour implémenter les échanges de messages, et ActiveMQ en tant que MOM.

## Gestion des erreurs

Dans le BusinessSample, deux classes SampleProducer et SampleConsumer sont chargées respectivement de produire et de consommer un message dans les brokers. Nous allons ici décrire leur fonctionnement et montrer la gestion standard des erreurs.

### Envoi d’un message

Le code suivant gère l’envoi de message :

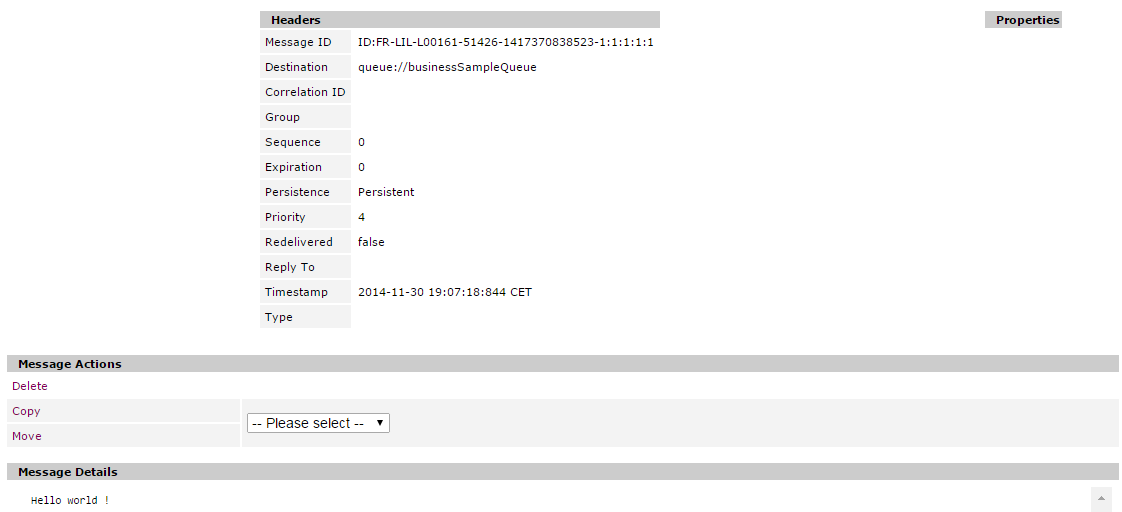
String message = "Hello world !";

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

producer.sendToJMSQueue(message);

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

Détails du message :



* Affichage dans la console Eclipse : **AUCUN AFFICHAGE**

### Réception d’un message

Le code suivant gère la réception d’un 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 :
* Affichage dans la console Eclipse: **INFO SimpleExchange:52 - Received: Hello world !**

Nous voyons ici que tracer les messages est assez simple. Cependant nous allons voir dans la partie suivante que leur suivi n’est pas forcément assuré.

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

Le code suivant génère une exception de type StringIndexOutOfBoundsException :

String message = "Hello world !";

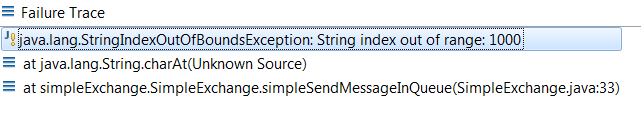
SampleProducer producer = **new** SampleProducer();

// génération d'une exception

message.charAt(1000);

producer.sendToJMSQueue(message);

* Affichage dans la console d’Eclipse :

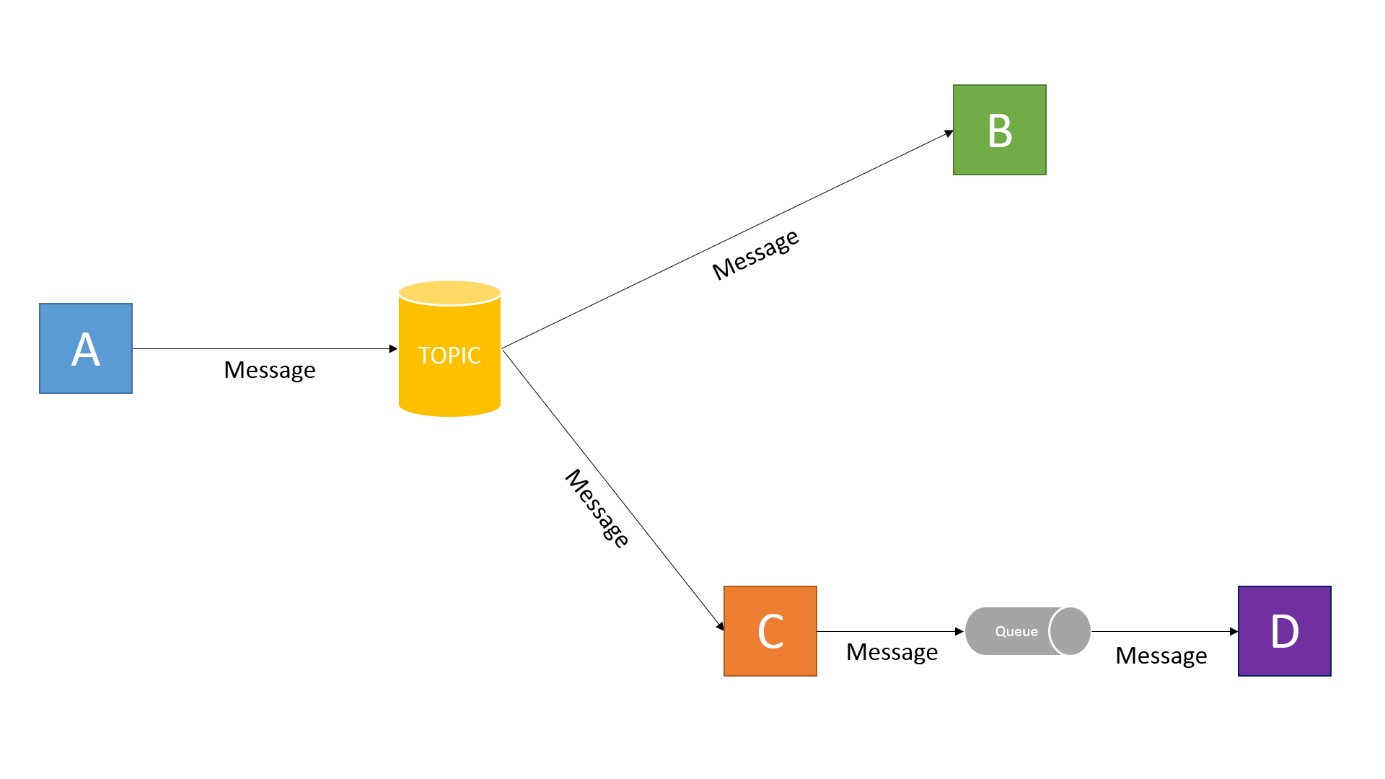


Nous voyons ici que le fait d’ajouter une erreur à l’envoi du message n’est détectable que dans les logs d’éclipse. Aucun suivi n’est assurée autant du coté JMS que du côté d’ActiveMQ, par conséquent le consumer n’est pas notifié de la « perte » du message. Pour aller un peu plus loin nous avons décidé donc de créer un scénario avec plusieurs acteurs. C’est ce que nous verrons dans cette prochaine partie.

## Simulation de conditions réelles

### Scénario

Nous avons choisi d’implémenter le scénario suivant :



*Explications : L’application A produit un message dans un Topic pour lequel les applications B et C ont souscris afin de consommer le message. L’application C renvoie alors le message dans une Queue sur laquelle une application D va consommer.*

### Gestion des exceptions

Afin de produire des exceptions nous avons décidé de créer un générateur d’exception. Ce générateur d’exception consiste en fait à produire une liste contenant des objets Exception, ou null.

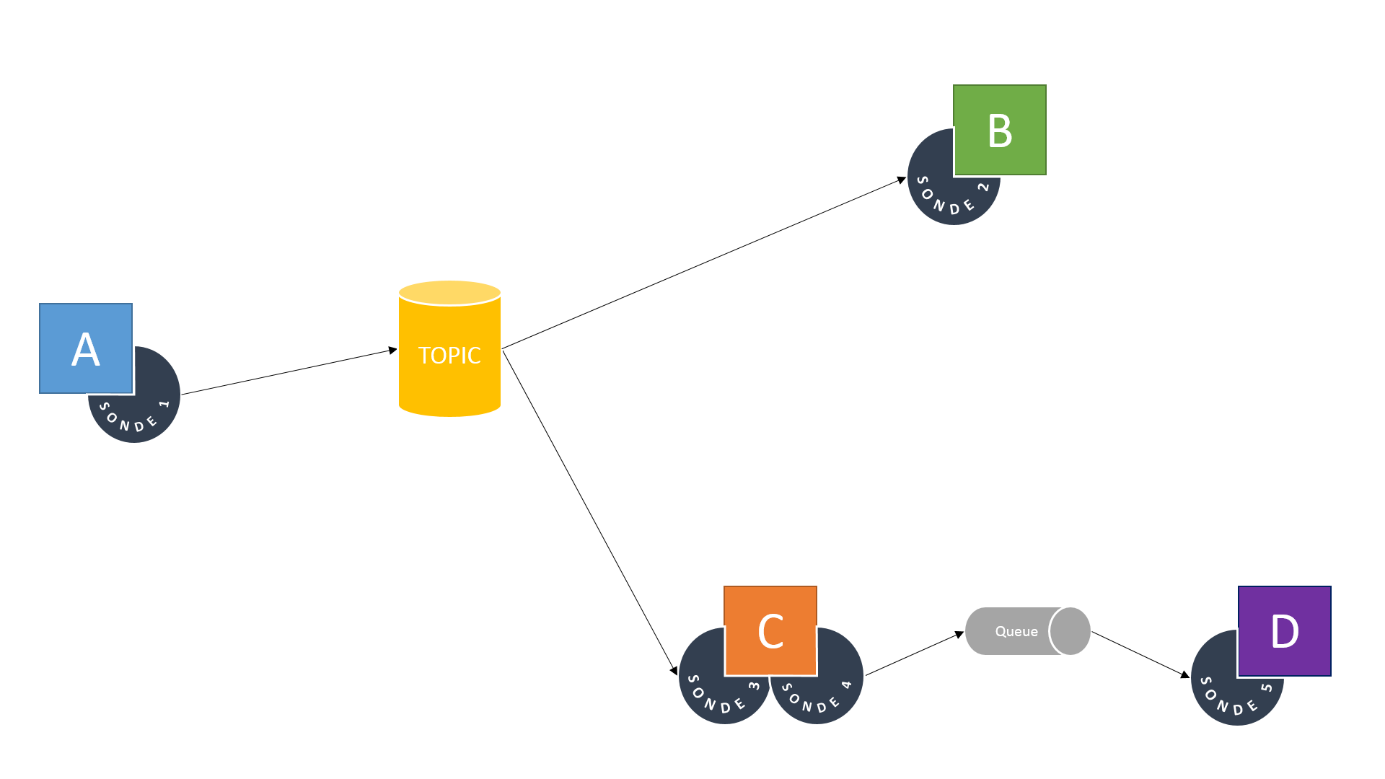
Note : La gestion des exceptions a été implémentée de façon relativement générique. Pour implémenter votre propre générateur il suffit d’implémenter la classe AbstractExceptionList et de changer la référence utilisée dans la classe ExceptionListManager.

Chacun des acteurs (Application A, B, C, ou D) est amené à piocher dans cette liste. Si l’élément pioché n’est pas null alors l’exception est déclenchée.

Comme vu dans la première partie la remontée des exceptions n’est pas fameuse si ce n’est lorsqu’on a les yeux sur la console Eclipse en développement, ou sur les logs du serveur en conditions réelles d’exécution. Nous allons maintenant détailler comment nous avons amélioré la gestion des erreurs.

# Amélioration de la visibilité des erreurs

## Des sondes

Le principe repose sur un système de sonde. Ainsi dès lors qu’un message est émis/reçu par un acteur, on place une sonde afin d’analyse le message. Les sondes (implémentées sous le nom de Sentinel) sont visibles sur le schéma suivant et peuvent se greffer sur les applications :

*Commentaire : dans le cas de l’application C on place deux sondes, une sonde chargée de récupérer le message, et une autre pour analyser le message envoyé à D.*

## Pourquoi des sondes ?

L’idée de sonde est venue naturellement, le fait d’avoir des petits modules pouvant se greffer n’importe où permet d’affiner la gestion des erreurs. En plus de pouvoir s’immiscer n’importe où cela minimise les modifications de l’application existante.

Le rôle d’un objet Sentinel est d’enregistrer l’état d’exécution de l’application au départ d’un envoi ou d’une réception de message via une méthode d’initialisation personnalisable. Lorsqu’une exception est lancée par l’application en question, le Sentinel sera chargé d’appliquer le traitement choisi.

## Résultats

Avec l’utilisation des sondes nous avons pu obtenir un meilleur suivi des messages qui transitaient entre les applications. Par exemple dans le cas d’un envoi de message depuis l’application A nous avons pu avoir sur la console Eclipse le résultat suivant alors qu’auparavant nous n’avions rien :

1. messageOrigineId=0,
2. errorMessage=null
3. errorDate=null
4. id=106
5. flowType=PRODUCED
6. status=SUCCESS
7. name=Application A produces
8. stackTrace=null
9. messageOrigine=

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<ns2:facture xmlns:ns2="http:///xml/message">

<id>16d8d214-9dc4-475b-94a9-9686be2b1f39</id>

<montant>0.771646</montant>

<client>

<nom>Bibi</nom>

<prenom>Bobo</prenom>

<adresse>10 rue du moulin 20293 Pépé</adresse>

</client>

</ns2:facture>

*Commentaire : Ici l’opération d’envoi s’est bien passée : A la ligne 1 on a associé un ID au message envoyé. A la ligne 4 l’ID de la transaction réalisée. A la ligne 6 l’état de la transaction. Et enfin à la ligne 9 le message qui a été envoyé.*

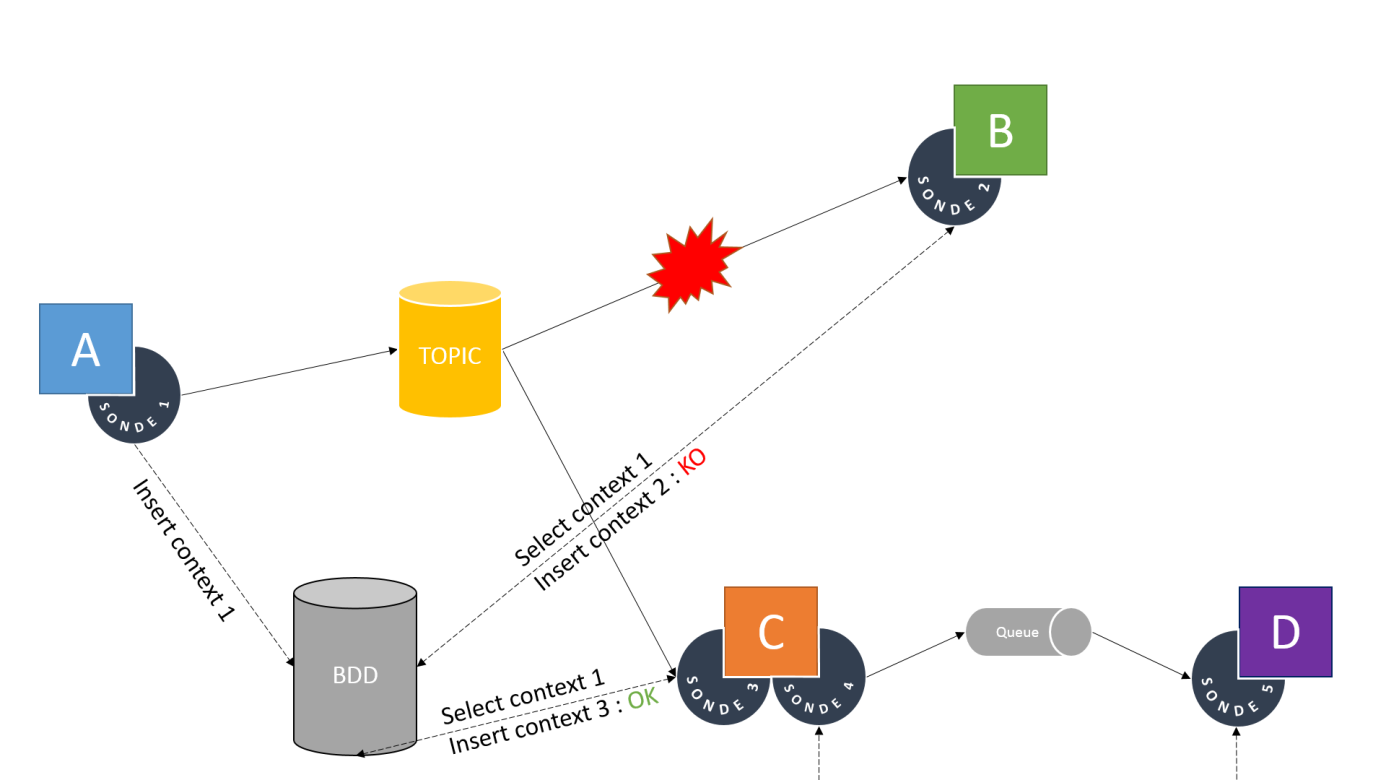
*Remarque : Ici dans notre implémentation des sondes nous avons choisi d’afficher le contexte d’exécution de l’application au moment de l’envoi du message. Nous aurions très bien pu imaginer autre chose, comme l’écriture dans un fichier de log etc.*

L’affichage d’un tel message sur la console d’éclipse ou dans les logs est utile mais n’est pas vraiment ergonomique et pratique. Pour ajouter une véritable couche de supervision des messages et des erreurs qu’ils peuvent engendrer, nous avons à ces sondes afin d’exploiter au maximum les données affichées.

# Supervision des erreurs

Afin de pouvoir superviser les erreurs remontées par les sondes, nous avons mis en place une persistance des données au cœur même de leurs méthodes afin de faciliter l’exploitation des données par une application tiers mais également de pouvoir garder une trace des erreurs sur le long terme à des fins d’analyse de problèmes éventuels.

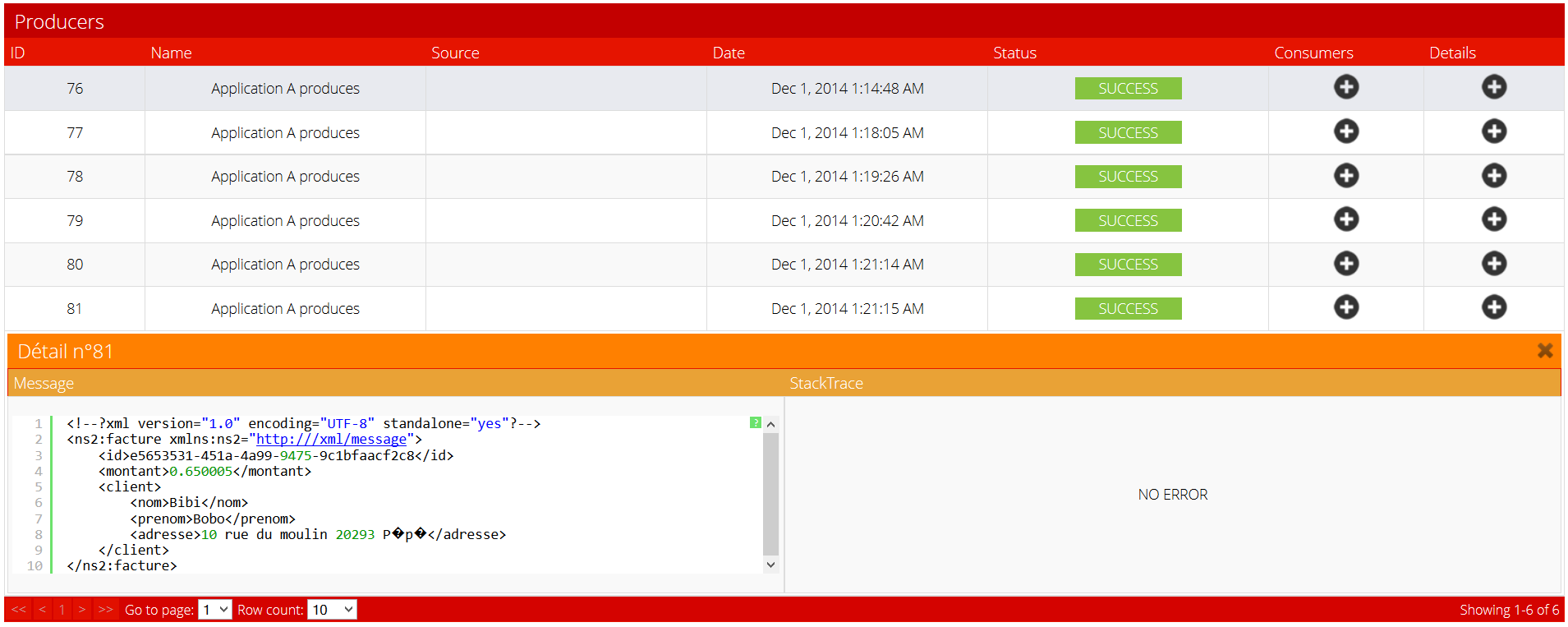
## Sondes et persistances de données



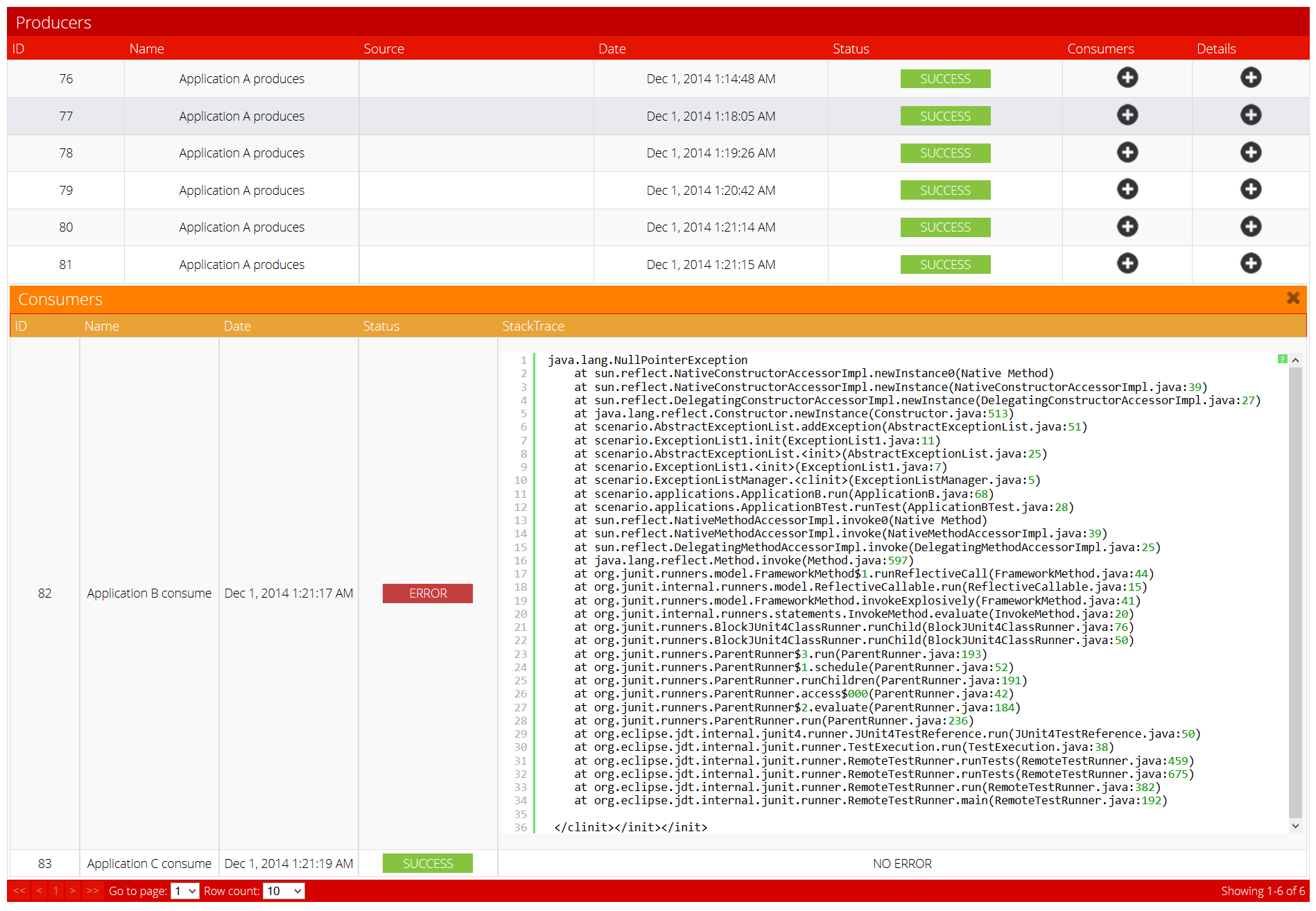
*Explications : Ici nous pouvons constater que la sonde de l’application A va enregistrer en base de données son contexte d’exécution au moment de l’envoi du message que l’on appellera 1. Nous avons ajouté au header du message qui transite, l’ID de ce contexte d’exécution. La sonde de l’application B pourra alors insérer en base de données que le message issu de l’application A n’a pas été traité correctement (suite au déclenchement de notre Exception). L’application C pourra insérer en base de données que le message issu de l’application A bien été reçu etc.*

## Exploitation des résultats

Nous avons décidé d’exploiter les données afin d’offrir à l’utilisateur une supervision des flux entre ses applications avec une IHM forte. Nous allons ici en décrire les fonctionnalités.



Quand on clique sur le + associé à la colonne *détails* : on peut voir les détails du message envoyé par A ainsi que le statut de l’envoi qui est à « SUCCESS ».



Quand on clique sur le + associé à la colonne *consumers*: on peut voir les consommateurs du message envoyé par A :

* B qui est au statut « ERROR » et dont on peut voir la stacktrace associée
* C qui est au statut « SUCCESS »

# Conclusion

Nous avons réussi à élaborer un concept permettant d’intégrer au sein d’une application existante des petits modules. Ces modules appelés Sentinel permettent une meilleure remontée des erreurs sur des technologies SOA qui à la base sont très perfectibles à ce niveau. Nos Sentinels permettent de collecter les informations du SI pour les afficher sur une IHM de façon clair.

Pour ce faire nous avons reproduit une situation réelle, en simulant des échanges de message entre plusieurs acteurs. Nous avons ensuite mis en évidence que la remontée d’erreur y était quasi inexistante. Par le biais de nos sondes Sentinel, nous avons démontré qu’il était possible d’obtenir des données sur les contextes d’exécution de chacun des acteurs. Et enfin que les données collectées par les sondes pouvaient être exploitées afin de mieux comprendre et suivre les erreurs via une IHM forte.

Nous sommes donc passés du statut de messages quasi-inexistant à une IHM forte collectant les informations relatives aux flux d’informations.

On peut imaginer par la suite une amélioration de l’IHM en ajoutant des filtres basés sur l’analyse des traces d’exécution afin de regrouper tous les messages qui ont pu causer une même erreur.