# Mise en oeuvre d'un système de publication/souscription basé sur les flux d'information de type RSS

Sylvain Bouquin

le cnam

# CONSERVATOIRE NATIONAL DES ARTS ET METIERS
# PARIS

_____

Mémoire présenté en vue d'obtenir

**le Diplôme d'Ingénieur du CNAM**

**Spécialité : INFORMATIQUE**
**Option : Ingénierie des Systèmes d'Information**

par
BOUQUIN Sylvain

_____

Mise en œuvre d'un Système de Publication/Souscription basé sur les Flux d'Information de type RSS

_____

Soutenu le 3 mars 2016

**JURY**

**Présidente :**

| | |
|---|---|
| METAIS Élisabeth | PR, CNAM |

**Membres :**

| | |
|---|---|
| DU MOUZA Cédric | MCF-HDR, CNAM |
| TRAVERS Nicolas | MCF, CNAM |
| FOURNIER-S'NIEHOTTA Raphaël | MCF, CNAM |
| COQUELET Pascal | Consultant senior IT, formateur |

cedric

# Table des matières

# Remerciements

La liste des personnes ayant contribué d'une manière ou d'une autre à l'aboutissement de ce long cursus passées au CNAM et se concluant par ce mémoire d'ingénieur est trop importante pour ne pas risquer d'en oublier. Je tiens donc à toutes les remercier et à m'excuser de ne pas les avoir citées.

Je remercie en premier lieu la FSGT, pour laquelle j'ai travaillé pendant sept ans comme responsable informatique. Elle m'a offert la possibilité d'accéder aux métiers de l'informatique et m'a facilité l'accès au cours du CNAM en parallèle de mon travail.

Le diplôme d'ingénieur du CNAM se concluant par un stage de fin d'études et un mémoire, j'en ai profité pour me forger une nouvelle expérience. Je remercie pour cela Cédric Du Mouza et Nicolas Travers, enseignants-chercheurs, de m'avoir ouvert la porte du laboratoire d'informatique du CNAM pendant près de 10 mois, en me proposant un sujet de stage passionnant. Vous en saurez plus en lisant la suite de ce mémoire ! Merci Nicolas, qui a suivi et orienté mon travail tout au long du projet, pour sa patience, sa disponibilité et tous ses précieux conseils. La combinaison harmonieuse de sa maîtrise du sujet de recherche et des technologies mises en œuvre ainsi que de son approche méthodologique ont été fondamentales à la réalisation de mon mémoire.

Toutes ces années d'études n'ayant pas été sans difficulté, il m'a fallu parfois aller chercher du soutien dans mon entourage. Je remercie particulièrement Thomas et Bruno, amis grimpeurs mais par ailleurs informaticiens, pour leur apport de connaissances en algorithmique et programmation Java.

La relecture d'un mémoire étant une étape importante, je remercie Claire et Delphine pour s'être proposées à cette tâche et avoir apporté leur regard de « non informaticiennes », fait part de leurs interrogations sur certains termes « barbares » ou structures étranges employées. J'espère avoir bien pris en considérations toutes leurs remarques.

Enfin, tous mes remerciements à Mariana, avec qui je partage ma vie depuis plusieurs années, qui a toujours su être présente et compréhensive. Au-delà de son soutien moral permanent, son aide « technique » a aussi fortement contribué à la réussite de mon parcours et particulièrement de certaines UE, grâce à ses précieuses explications et sa méthodologie.

# Liste des abréviations

| | |
|---|---|
| API | Application Programming Interface |
| BPMN | Business Process Model and Notation |
| CEDRIC | Centre d'Etude et De Recherche en Informatique et Communication |
| CVS | Concurent Version System |
| DAO | Data Access Object |
| EJB | Enterprise Java Bean |
| FiND | Filtering by Novelty and Diversity |
| HTML | HyperText Markup Language |
| JAR | Java Archive |
| JEE | Java Enterprise Edition |
| JPA | Java Persistance API |
| JRE | Java Runtime Environement |
| JSF | Java Server Faces |
| JSON | JavaScript Object Notation |
| JSP | Java Server Page |
| JVM | Java Virtual Machine |
| MVC | Model View Controller |
| NoSQL | Not Only SQL |
| POJO | Plain Old Java Object |
| RAM | Random Access Memory |
| RMI | Remote Methode Invocation. Composant JEE. |
| RSS | Really Simple Syndication |
| SGBD | Système de Gestion de Bases de Données |
| SPOF | Single Point Of Failure |
| SVN | Subversion |
| TDV | Term Discrimination Value |
| UML | Unified Modeling Language |
| WAR | Web Archive |
| XHTML | Extensible HyperText Markup Language |
| XML | Extensible Markup Language |

# Glossaire

| | |
|---|---|
| Ajax | L'architecture informatique Ajax (acronyme d'Asynchronous JavaScript and XML) permet de construire des applications Web et des sites Web dynamiques interactifs sur le poste client en se servant de différentes technologies ajoutées aux navigateurs Web. Ajax combine JavaScript, les CSS, JSON, XML, le DOM et le XMLHttpRequest afin d'améliorer maniabilité et confort d'utilisation des applications internet riches. (source : fr.wikipedia.org) |
| RSS / ATOM | RSS et Atom sont deux formats de syndication Web basé sur XML. Ils permettent de récupérer via des agrégateurs de flux de récupérer les informations des sites Web qui fournissent ce type de flux, auxquels ont peut s'abonner. |
| Jenkins | Outil open source d'intégration continue écrit en Java et fonctionnant dans un conteneur de servlets tel qu'Apache Tomcat, ou en mode autonome avec son propre serveur Web embarqué. Il s'interface avec des systèmes de gestion de versions tels que CVS, Git et Subversion, et exécute des projets basés sur Apache Ant et Apache Maven aussi bien que des scripts arbitraires en Shell Unix ou batch Windows. |
| Methodes Agiles | Méthodes de développement et de gestion de projet informatique se voulant plus pragmatique que les méthodes traditionnelles. Elles cherchent notamment à impliquer au maximum les différents acteurs du projet. Elles reposent sur le principe de cycles de développements incrémentaux, adaptatifs et itératifs. (Source : Wikipedia.fr) |
| MVC | Le motif de conception « Modèle-Vue-Controleur » est généralement mis en œuvre pour prendre en charge les interactions utilisateurs : le Modèle représente l'ensemble des composants qui sont chargés de réaliser des appels à la couche Services et de mettre les résultats de l'appel à la disposition de la Vue - la Vue représente l'interface utilisateur - le Contrôleur gère la synchronisation entre la Vue et le Modèle, il surveille les modifications du modèle et informe la vue des mises à jour nécessaires. |
| POJO | Un Plain Old Java Object (POJO) que l'on peut traduire en français par bon vieil objet Java. Cet acronyme est principalement utilisé pour faire référence à la simplicité d'utilisation d'un objet Java en comparaison avec la lourdeur d'utilisation d'un composant EJB. Ainsi, un POJO n'implémente pas d'interface spécifique à un framework comme c'est le cas par exemple pour un composant EJB. (source : fr.wikipedia.org) |
| RMI | Remote Method Invocation est une interface de programmation (API) pour le langage Java qui permet d'appeler des méthodes distantes. Cette bibliothèque qui se trouve en standard dans JSE, est une technologie qui permet la communication via le protocole HTTP entre des objets Java éloignés physiquement les uns des autres, autrement dit s'exécutant sur des machines virtuelles java distinctes. RMI facilite le développement des applications distribuées en masquant au développeur la communication |

client / serveur.

| | |
|---|---|
| Spring | framework pour construire et définir l'infrastructure d'une application java, dont il facilite le développement et les tests en s'appuyant sur des POJO qui n'ont pas besoins de s'exécuter dans un conteneur spécifique. |
| Struts | Le framework Struts met en œuvre le modèle MVC 2 basé sur une seule Servlet faisant office de contrôleur et des JSP pour l'IHM. L'application de ce modèle permet une séparation en trois parties distinctes de l'interface, des traitements et des données de l'application. (source : jmdoudoux.fr) |
| Swing | Swing est une bibliothèque graphique pour le langage Java. Il offre la possibilité de créer des interfaces graphiques identiques quel que soit le système d'exploitation sous-jacent. Il utilise le principe Modèle-Vue-Contrôleur (les composants Swing jouent en fait le rôle du contrôleur au sens du MVC) et dispose de plusieurs choix d'apparence (de vue) pour chacun des composants standards (source : fr.wikipedia.org). |
| Web Services | Un service Web est un programme informatique de la famille des technologies Web permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués. |

# Introduction

Nous assistons actuellement à un afflux massif de données échangées via Internet offrant un réel potentiel d'accès à l'information. Cependant la capacité à pouvoir retourner ces informations de manière pertinente est un défi majeur de traitement des données accumulées. Si ces données sont un réel atout pour l'accès à l'information, la problématique du traitement visant à les retourner de manière pertinente est un défi majeur. Cette masse d'information tellement considérable n'a en effet d'intérêt que si l'on est capable de la filtrer pour la rendre exploitable par un utilisateur. Cette problématique, dont les enjeux peuvent-être importants, attire l'attention d'un certain nombre d'acteurs du Web et de la recherche scientifique. Dans ce contexte, les équipes de recherche en « systèmes d'information » et en « bases de données » du laboratoire d'informatique du CNAM (Le CEDRIC) mènent des travaux autour du traitement et de la gestion des données massives depuis un certain nombre d'années. Elles travaillent entre autres sur un système nommé *FiND*[2] (*Filtering by Novelty and Diversity*), système de publications/souscriptions (Pub/Sub) basé sur un filtrage par pertinence, en temps réel, dont les données sont produites par des flux d'information de type RSS. Le mémoire présenté ici s'intéresse essentiellement à la mise en œuvre du système *FiND*.

**La première partie** est consacrée au contexte et aux problématiques soulevées par le système *FiND* ainsi qu'à la présentation de ses objectifs. Il semble tout d'abord utile de décrire l'environnement de recherche dans lequel le projet est porté. Sans rentrer dans les détails, nous nous concentrerons ensuite sur les différentes problématiques soulevées par le système *FiND* telles que :

- les flux d'information type RSS/ATOM ;
- le traitement de grand volumes de données en temps réel ;.
- la pertinence de la notification.

Nous présenterons ensuite la genèse du système Pub/Sub en question. Un retour « historique » sur ses différentes phases qui l'ont conduit à son état actuel ainsi qu'une présentation des différents acteurs permettront de positionner les différents niveaux d'intervention dans le projet global. Ainsi nous terminerons cette partie en détaillant les objectifs du projet liés

2 A Real-time Filtering by Novelty and Diversity for Publish/Subscribe Systems : http://cedric.cnam.fr/index.php/publis/article/view?id=3343

directement à ce mémoire, qualifiés de « mise en œuvre du système », mais qui peuvent se résumer de la manière suivante : « apporter les éléments et les modifications nécessaires au logiciel, lui permettant d'être exploitable et évolutif ». Cela se traduit notamment par la ré-architecture du logiciel, l'intégration d'une interface Web ou encore l'ajout de nouvelles fonctionnalités.

**La deuxième partie** s'intéresse aux concepts abordés dans le cadre de ce mémoire. Ils sont regroupés autour de trois domaines. Le premier domaine concerne la démarche de ré-ingénierie logicielle qui nous guidera dans les différentes étapes de construction d'un projet de logiciel. Le second domaine concerne l'architecture logicielle nous aidant à construire une structure évolutive et maintenable. La troisième domaine est liée à la gestion du très grand volume de données à traiter. Nous évoquerons succinctement les concepts de *Big-Data* pour nous diriger plus précisément sur la présentation des systèmes de stockage *NoSQL* ainsi que le modèle de programmation de requêtes distribuées *MapReduce* généralement mis en œuvre dans ces systèmes.

**La troisième partie** est consacrée à la « rétro-ingénierie » du système en place. Elle permet tout d'abord de faire un état des lieux de l'existant et de comprendre le fonctionnement du système *FiND*. Par ailleurs elle permet d'aider à déterminer ses points de restructuration et d'amélioration. Enfin cette étape apporte les bases d'une documentation, aspect fortement recommandé pour la maintenance et l'évolution d'un système. La démarche de formalisation employée passe essentiellement par la réalisation de diagrammes modélisant l'application et offrant ainsi une vision synthétique du système.

**La quatrième partie** est consacrée à la présentation et à la mise en œuvre des différentes technologies déployées dans ce système. Hormis celles liées au stockage, déjà employées lors de la « phase 2 » du système, ces technologies font partie des nouveautés de la « phase 3 ». On peut les classer en trois catégories :

- Celles liées au stockage et aux requêtes sur les données comme le Système de Gestion de Bases de Bonnées (SGBD) *NoSQL* MongoDB ou le *framework* de requêtes *MapReduce*.

- Celles liées à la programmation du logiciel en lui-même comme le langage Java, la plate-forme JEE et le *framework* Web JSF,
- Celles liées à « l'usine » logicielle comme le conteneur JEE et serveur Web Jetty, le gestionnaire de version Git, l'outil de *buid* Maven, les Ateliers de Génie Logiciel (AGL) Eclipse et Modelio.

**La cinquième partie** décrit plus précisément les nouveautés apportées au système. Cette phase implique d'avoir préalablement réalisé les deux précédentes. On y présentera la nouvelle architecture produite ; l'implémentation de l'interface Web reposant sur le *framework* JSF et ; la création d'un *workflow* permettant l'ajout, la modification « à chaud » et l'exécution de fonctions *MapReduce* utilisées pour interroger le SGBD.

Enfin, nous conclurons par le retour critique d'une expérience de 8 mois vécue au sein de ce projet permettant ainsi de dégager quelques perspectives d'évolution du système. On s'interrogera notamment sur les problèmes posés par la mise en production du nouveau système dans une situation réelle d'utilisation prenant notamment en compte l'arrivée d'items en flot continu.

# 1. Contexte, problématiques et objectifs du projet

Il semble tout d'abord important de décrire l'environnement dans lequel sont menés les travaux de recherche qui ont produit le système de publication/souscription, base du sujet de ce mémoire. Cela nous permet ainsi d'exposer les problématiques auxquelles s'intéressent les équipes de recherche. Elles nous permettront ainsi de mieux appréhender les différentes phases du système de filtrage et de notifications et de présenter les acteurs associés à celles-ci. Enfin nous terminerons par les objectifs de la phase de mise en œuvre du système traitée dans ce mémoire.

## 1.1 Présentation de la structure : Le CEDRIC

Le Centre d'Étude et de Recherche en Informatique et Communications (CEDRIC) regroupe l'ensemble des activités de recherche en informatique, mathématiques appliquées et électronique menées au CNAM. C'est au sein des équipes « Bases de Données Avancées » (Vertigo) et « Ingénierie des Systèmes d'Information et de Décision » (ISID), que sont menés, entre autres, les travaux de recherche relatifs au système *FiND* sur la base duquel repose ce projet de mémoire.

## 1.2 Problématique de recherche liée au Système de publication / souscription

L'explosion de la quantité d'informations publiée sur le Web a conduit à l'émergence d'un paradigme de syndication du contenu du Web, qui transforme le lecteur passif en un collecteur actif d'information. Les consommateurs d'information s'abonnent aux flux RSS/Atom et sont notifiés quand une nouvelle information (item) est publiée. La syndication Web est maintenant employée sur les sites Web, les blogs et les réseaux sociaux. Cependant, elle soulève des problèmes de passage à l'échelle, notamment concernant le filtrage en temps réel des flux, permettant aux utilisateurs de suivre effectivement et personnellement les informations qui les intéressent.

C'est dans le contexte de travaux de recherche menés notamment par C. du Mouza, M. Scholl, N. Travers et Z. Hmedeh, qu'est né le « Système de Filtrage par Nouveauté et Diversité pour la Publication/Souscription (note 2)». Les travaux de thèse de Z. Hmedeh (2013) intitulé « Indexation pour la recherche par le contenu textuel de flux RSS », sont les fondements de ce système. Pour nous permettre de comprendre le but du système Pub/Sub nous présentons l'évolution des travaux sous forme de « phases » du système.

## 1.3 Phase 1 : les fondements du système Pub/Sub

La mise à disposition de contenu (syndication), utilisé par les sites Web, repose essentiellement sur deux formats de données concurrents, mais très proches, RSS[3] ou *ATOM*, que l'on trouve sous forme de fichier XML[4]. Ces fichiers de publication d'information sont organisés sous forme « d'items ». Chaque item contient un ensemble de méta-données comme le titre, le lien vers l'information complète (généralement le site diffuseur), la courte description de l'information, la date de dernière publication. L'utilisateur intéressé par une information va s'abonner à ce flux et ainsi suivre ses éventuelles modifications.

Le problème posé par l'abonnement à un flux précis, c'est qu'il oblige l'utilisateur à choisir sa source d'informations. Il doit par ailleurs répéter cette opération d'abonnement sur chacune des sources dont il souhaite obtenir de l'information. Aussi chaque source lui fournira un type d'information prédéterminée. Autrement dit l'utilisateur ne peut pas décider lui-même de ne recevoir que les informations concernant, par exemple, le « football » si le site en question n'a pas prévu cette catégorie d'information sous forme de flux RSS. Il devra s'abonner d'un côté au flux d'informations du site, par exemple Libération, relatif aux dernières actualités. Puis d'un autre à celui du Monde relatif aussi aux dernières actualités. Des logiciels appelés « agrégateurs RSS », après avoir ajouté les sources souhaitées, permettent de collecter les nouveautés provenant des flux et de les notifier aux utilisateurs.

### 1.3.1 Problèmes posés par la notification

Généralement ce type de logiciel n'offre pas la possibilité de sélectionner un thème, en utilisant par exemple un mot clé comme « football » et ainsi le laisser notifier à l'utilisateur à chaque nouvelle information sur le thème en question parmi la liste des flux. Certains logiciels, parfois appelés « générateurs d'alertes », tel que Google Alerts, gèrent ces options. C'est cette manière de notifier qui a motivé les travaux de recherche. Ce type de système de notification offre, de fait, une autre façon de s'abonner et de recueillir l'information. L'utilisateur ne va plus s'abonner directement à des flux provenant d'une source particulière mais souscrire à un type d'information, indiqué par une liste de mots clés. Un des problèmes posés par ce type de système est la quantité d'informations potentiellement disponibles[5]. Quels choix va opérer le système pour notifier l'utilisateur sans le submerger d'informations qu'il ne saurait traiter ? En effet, le système est alors confronté à devoir notifier

3 RSS : Really Simple Syndication
4 XML : Extensible Markup Language
5 RSS feeds behavior analysis, structure and vocabulary : http://cedric.cnam.fr/index.php/publis/article/view?id=3071

potentiellement des millions d'utilisateurs face aux items, arrivant en flot continu, provenant de centaines voir de milliers de flux différents. En outre, si l'on souhaite être notifié de manière quasi instantanée, à chaque nouvelle information (item) correspondant aux critères de notifications, le problème se complexifie d'avantage.

### 1.3.2 Le système *FiND*

L'objectif du système *FiND* est de pouvoir notifier le plus pertinemment et rapidement possible l'utilisateur (note 2). Il doit être capable d'analyser le contenu de chaque nouvel item arrivant, à savoir un ensemble de mots, que l'on appelle « termes », contenus dans le titre et la description de l'item et les comparer avec l'ensemble des souscriptions (un autre « petit » ensemble de « termes »). De manière basique et sans aucun filtrage spécifique, une souscription est notifiée quand ses termes de recherche se retrouvent, partiellement ou intégralement, dans les termes de l'item. C'est ce que l'on appelle ici l'étape de *Matching*. Dans ce cadre, les travaux de recherche ont permis d'élaborer des algorithmes et des structures de données, permettant d'assurer un « passage à l'échelle » du Web, autrement dit, de prendre en compte la masse de données potentielles en croissance permanente et cela dans des délais très courts (inférieurs à la seconde), qu'on prendra le risque de qualifier de « temps réel[6] ».

Tout d'abord, les chercheurs ont fait le choix d'un stockage de toutes les structures de données uniquement en mémoire centrale, s'épargnant ainsi de l'utilisation de mémoire de masse, tels que les disques durs, ayant des temps d'accès lents comparés à la mémoire « vive ». Nous ne détaillerons pas ici les structures de données employées et la manière dont elles ont été implémentées, pour cela il est préférable de se référer à la thèse de Z. Hmedeh (2013). Cependant nous présentons une des structures d'indexation, parmi les trois testées, du dictionnaire et des souscriptions et qui a été reprise dans l'implémentation en base NoSQL (travaux de la phase 2).

L' illustration 1 montre une structure de liste inverse de type *Ranked-key Inverted List (RIL)* permettant de ne pas avoir à parcourir l'ensemble des souscriptions lors de l'arrivée d'un item. Ce type d'indexation permet de créer des listes de souscriptions uniquement pour les termes les plus fréquents. L'identifiant d'une souscription est ajouté à la liste ainsi que le reste de ses termes. Lorsqu'un item arrive, son terme le moins fréquent permettra d'identifier rapidement les souscriptions pouvant lui correspondre. Il suffira ensuite de vérifier les restes

6 Subscription Indexes for Web Syndication Systems : http://cedric.cnam.fr/index.php/publis/article/view?id=2279

des termes de ces souscriptions avec ceux de l'item pour réaliser le *Matching*

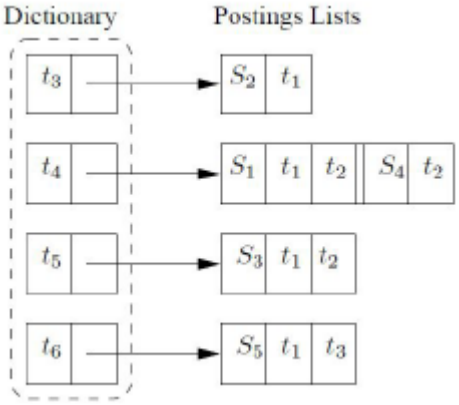


*Illustration 1: Structure d'indexation "Ranked-key inverted list" (source :thèse de Z. Hmedeh (2013))*

Afin d'éviter la submersion d'informations redondantes que l'on peut retrouver dans plusieurs flux et d'améliorer la satisfaction de l'utilisateur, les travaux se sont intéressés à la mise en place d'algorithmes de « filtrages intelligents capables de réduire l'ensemble des items notifiés pour chaque souscription[7] » . Le système va vérifier pour chaque nouvel item sa « nouveauté » et sa « diversité » afin de décider s'il doit le notifier à l'utilisateur. La vérification doit tenir compte des précédentes notifications afin de conserver sa pertinence. Pour déterminer la pertinence le système s'appuie sur l'importance des « termes ». Pour l'évaluer, chaque « terme » se voit attribuer une pondération, que l'on appelle ici « TDV » (*Term Discrimination Value*). Cette technique permet de déterminer un « score de pertinence » à l'item qui, s'il dépasse un seuil fixe, engendrera une notification. Ces « termes », qui peuvent atteindre plusieurs millions, sont stockés de manière unique et ordonnées précisément dans une structure de données que l'on appelle le « vocabulaire ». L'utilisation permanente et continue (pour chaque terme testé) de cette structure, impose de pouvoir y accéder le plus rapidement possible, d'où l'importance de la stocker en mémoire centrale.

Enfin, les notions de « nouveauté » et de « diversité » sont essentielles (note 7) pour éviter la redondance d'informations tout en couvrant un maximum d'actualités dans un « ensemble d'items minimaux ». La vérification de la nouveauté et de la diversité ne peut avoir lieu qu'à partir du moment où les termes d'une souscription se trouvent intégralement, ou partiellement

7 TDV-based Filter for Novelty and Diversity in a Real-time Pub/Sub System : http://cedric.cnam.fr/index.php/publis/article/view?id=3344

selon la configuration choisie, dans l'item testé et donneraient lieu à une notification si l'on se contentait de ce test : c'est l'étape de *Matching*.

**La nouveauté** permet surtout d'éviter la redondance d'une même information. En l'occurrence un item ayant déjà fait l'objet d'une notification ne sera plus notifié même s'il apparaît dans un autre flux. Il est par ailleurs possible qu'un nouvel item, tel que l'item 3 dans l'exemple ci-dessous, soit déjà couvert par un autre, soit dans l'exemple, l'item 2. Pour vérifier cette couverture, on mesure la similarité entre deux items en observant la différence et l'importance des termes entre les deux items.

**La diversité** permet d'apporter de la variété à un sujet. L'item retourné doit se différentier de ceux ayant déjà fait l'objet d'une notification du point de vue du fond et non de la forme. Plus précisément, comme il est défini dans la thèse de Z.Hmedeh (2013) « La diversité de l'information peut être considérée comme l'inverse de la similarité : moins les résultats d'un ensemble sont similaires plus son information est diverse (ce qui revient à augmenter la distance moyenne entre les éléments de cet ensemble) ».

Afin d'illustrer simplement le concept de nouveauté et de diversité prenons l'exemple suivant tel qu'il est proposé dans la thèse de Z.Hmedeh (2013). La souscription s = « Football 2010 » a déjà été notifiée par les items :

- I1 = Football 2010 Italie
- I2 = Football 2010 France Allemagne

Le tableau ci-dessous montre un ensemble de nouveaux items publiés. Nous indiquons le filtrage qui sera appliqué à la souscription « s » tenant compte des précédents Items « I1 » et « I2 » ayant fait l'objet d'une notification.

Légende : Oui (apporte nouveauté ou diversité) ; Non (n'apporte rien)

| Item | Nouveauté | Diversité |
|---|---|---|
| I3 = Football 2010 France | Non | Non |
| I4 = Football 2010 France Italie | Oui | Non |
| I5 = Football 2010 Italie Brésil | Oui | Oui |

- I3 est inclus dans I2, par conséquent il n'apporte ni nouveauté ni diversité car son information est couverte par I2 .

- I4 n'est couvert par aucun des items de l'historique. Il apporte donc de la nouveauté. Il n'apporte pour autant pas de diversité puisque son information est couverte par I1 et I2.
- I5 n'est couvert par aucun des items de l'historique. Il apporte donc de la nouveauté. Son information n'est pas non plus couverte par l'ensemble des items de l'historique. Ni I1 et ni I2 ne couvre l'information Brésil. Il apporte donc de la diversité.

Si l'on peut concevoir l'intérêt de stocker l'ensemble des structures de données en mémoire centrale, pour des raisons de réduction des temps d'accès aux données, on peut aussi identifier certaines limites. La mémoire centrale n'est pas persistante, cela signifie qu'en cas de redémarrage de la machine toutes les données stockées seront perdues. Il faut donc prévoir un système de persistance pour la restauration en cas de problème. On peut aussi s'interroger sur le problème d'extensibilité en cas de manque d'espace de stockage. Les machines on effectivement des limites en termes de capacités à accueillir de la mémoire volatile (centrale). Si la limite est atteinte, il faudrait donc envisager de pouvoir multiplier le nombre de machines pour distribuer le stockage et les traitements sur celles-ci. C'est cette problématique qui a motivé les travaux de la « seconde phase » du système.

## 1.4 Phase 2 : implémentation du système sur un SGBD *NoSQL*

Les travaux de cette seconde phase cherchent à stocker la majorité des structures de données du système de « publication/souscription », dans une base de données. Le but est d'obtenir des temps de traitement équivalents à l'implémentation en mémoire centrale, alors que les structures de données sont, dans ce cas, gérées en mémoire de masse (disques durs). Ces nouveaux travaux menés dans le cadre du mémoire de Master de M. Han (2014) intitulé « An Intelligent Publish / Subscribe at Web Scale », proposent une implémentation sur un système de stockage et de calcul distribué reposant sur la base de données *NoSQL* MongoDB et le *framework MapReduce*. Les bases de données *NoSQL* qui sont présentées plus en détail dans la seconde partie relative aux technologies mises en œuvres, sont réputées pour leur capacité à supporter de très grands volumes de données en les répartissant sur plusieurs serveurs. Elles sont aussi très performantes en lecture. *MapReduce,* présenté en seconde partie, est un *framework* souvent utilisé pour réaliser des requêtes sur ce type de bases en distribuant les traitements sur une « grappe » (*cluster)* de bases de données.

Les travaux relatifs à cette implémentation ont conduit à la ré-écriture des algorithmes de *matching* et de filtrage (nouveauté et diversité) en s'appuyant sur le langage Java pour la

partie applicative du système et JavaScript (langage imposé par le SGBD) pour la partie requêtes *MapReduce*. Les structures de données ont elles aussi été repensées pour pouvoir s'intégrer au nouvel environnement. Globalement il y a trois structures de données : celle des souscriptions , celle des items et celle du vocabulaire. Cette dernière est toujours stockée en mémoire centrale. Nous présenterons plus en détail ces algorithmes et structures de données dans la partie sur la rétro-conception, puisqu'ils sont en lien direct avec les travaux de ce mémoire.

Les objectifs du travail de ce mémoire de Master consistaient surtout à valider la faisabilité du système de publication/souscription dans un environnement distribué. Plusieurs implémentations des algorithmes et structures de données ont été testées (cf : mémoire de M.Han (2013)). L'environnement de test s'est réalisé sur la base d'un million de souscriptions pour mille items et 1,5 millions de termes. Les tests ont été effectués sur un *cluster* de 1 à 8 serveurs de bases de données MongoDB.

L'architecture du système mise en place dans les travaux de Master, que nous appellerons parfois « logiciel de publication / souscription » ou Pub/Sub, ne permet pas à ce stade une véritable exploitation par un utilisateur final et n'est pas conçue de manière à pouvoir évoluer « facilement ». C'est justement cette dimension du travail que doit prendre en charge ce que nous appelons la « phase 3 » du système et que nous traiterons dans ce mémoire d'ingénieur. Nous aurions pu l'appeler « phase 2.1 » puisqu'elle est dans la continuité directe de la « phase 2 » et ne repose pas sur les travaux de la phase 1, au-delà de ses concepts.

## 1.5 Phase 3 : mise en œuvre du système de publications / souscriptions

Les objectifs de cette phase sont de l'ordre de la mise en œuvre du système précédent (phase 2). Il s'agit particulièrement d'apporter de l'évolutivité au système et de le rendre plus exploitable. Nous n'intervenons donc pas sur les algorithmes et structures de données en place qui sont le cœur de l'application. Toutes les adaptations ou évolutions de code existant n'ont pas pour but de modifier les résultats et les performances. Au mieux on les améliore, mais ce n'est pas l'axe principal. Nous ne remettons pas en question les technologies déjà mises en œuvre. Autrement dit nous conservons le langage Java pour tout ce qui concerne la partie applicative, le SGBD MongoDB et le *framework MapReduce*. Voici donc les axes de travail qui nous intéressent et qui sont l'objet de ce mémoire :

- Ré-architecturer le logiciel pour le rendre plus évolutif et maintenable : le principe d'architecture en couche et en *tiers* est mis en œuvre.

- Apporter des fonctionnalités liées à l'exploitation des statistiques : ajout de nouvelles fonctions *MapReduce*, *framework* de requêtes distribuées sur le SGBD.
- Création d'un *workflow* pour la gestion « à chaud » des fonctions *MapReduce* : offrir une interface Web permettant à l'utilisateur d'ajouter ou de modifier les fonctions *MapReduce* et de les exécuter, comme celle liées aux statistiques.
- Apporter une documentation au système : fournir les documents et diagrammes nécessaires aidant la reprise du système par de nouveaux développeurs. Ceci intègre la génération d'une documentation de code, la Java-Doc.

Les objectifs ainsi définis impliquent malgré tout d'autres tâches sous-jacentes comme :

- Étape de rétro-conception : il s'agit de comprendre le travail effectué en phase 2 (les algorithmes, les structures de données, l'architecture logicielle et l'infrastructure) et de le formaliser sous forme de documents rédigés ou de diagrammes. Cette étape permet aussi de réaliser un état des lieux du système et d'en dégager plus précisément les points d'amélioration ou de restructuration.
- Intégration d'une « usine logicielle » : il s'agit d'intégrer les outils nécessaires ou utiles à l'évolution, au suivi, la maintenance et l'exploitation du système. Sont donc mis en place un outil de gestion de projet, Maven, un outil de suivi de version, Git et la mise en œuvre d'un conteneur de Servlet / serveur Web pour le fonctionnement de l'interface Web.

# 2. Concepts abordés dans le projet

Dans cette partie nous décrivons les concepts abordés directement dans cette phase du projet, sachant que ceux, plus généraux, relatifs notamment à la publication/souscription (nouveauté et diversité par exemple) ont déjà été évoqués précédemment. Les concepts sont répartis en trois domaines et nous permettent d'y faire référence selon les besoins :

- Le premier domaine concerne la démarche de ré-ingénierie logicielle. Nous décrivons globalement l'ensemble des étapes nécessaires à la construction d'un projet de logiciel de l'analyse de l'existant jusqu'à son évolution.
- Le deuxième domaine concerne l'architecture logicielle. Cette étape est fondamentale pour nous permettre de bâtir des « fondations saines » au logiciel sans pour autant remettre en question ses fonctionnalités. Une bonne architecture facilitera grandement l'évolution et la maintenance du logiciel.
- Un troisième domaine concerne le BigData et les systèmes de stockage *NoSQL*. Tel que cela a été évoqué dans la partie « contexte et problématique », la prise en compte et la gestion de très grands volumes de données sont les aspects essentiels du projet *FiND*. Nous tentons d'expliquer le plus succinctement possible la vaste notion de BigData pour ensuite nous intéresser plus particulièrement aux différents types de systèmes de stockage de données utilisés dans un contexte BigData. Nous présentons enfin le modèle de programmation *MapReduce* généralement mis en œuvre dans ces systèmes. Il devient particulièrement pertinent pour réaliser des requêtes complexes dans un environnement de stockage de données distribué tel que celui mis en œuvre dans le projet.

## 2.1 Démarche de ré-ingénierie logicielle

En référence à la définition traduite et donnée par Marin Flower[8] dans *« Refactoring – Improving the Design of Existing Code »* , La ré-ingénierie logicielle est un processus ayant pour objectif de transformer et améliorer un logiciel à partir du modèle existant sans toutefois modifier son comportement externe ». Dans notre cas, les algorithmes étant déjà écrits et ayant été testés et validés, l'évolution du système ne doit dégrader ni ses fonctions ni ses performances. L'objectif de cette démarche est donc, au mieux, d'en conserver les

8 Martin Flower est auteur de plusieurs ouvrages dans le domaine de la conception logicielle et de l'approche Agile.

performances actuelles, mais surtout de lui apporter de l'évolutivité. Cette démarche opère en plusieurs étapes, qui peuvent varier plus ou moins selon le contexte et les objectifs souhaitables. Cependant le diagramme ci-dessous donne une vision théorique des différentes phases de ré-ingénierie inspirant l'approche menée dans ce projet.

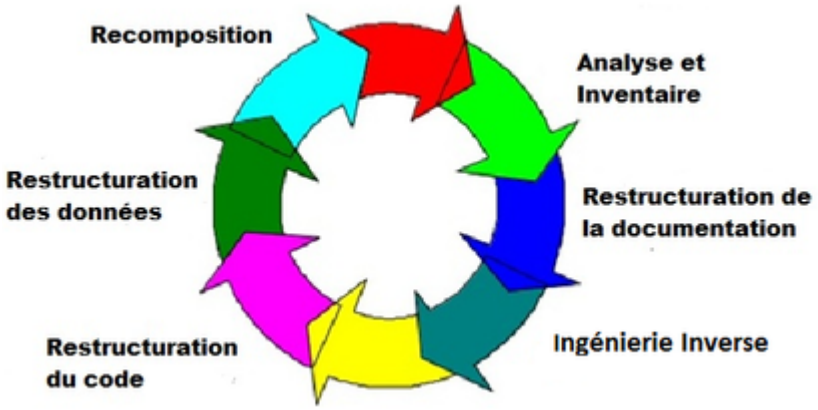


*Illustration 2: Les différentes phases de la réingnierie logicielle (ref: fr.wikipedia.org/wiki/Réingénierie_logicielle)*

Comme on peut le voir sur le diagramme de l'illustration 2, le processus est une roue. La démarche de ré-ingénierie peut-être incrémentale. Il est tout à fait possible d'effectuer le processus complet sur une partie du système – cas de notre étude -, puis de réitérer la démarche sur une autre partie. C'est notamment le type d'approche préconisée par les méthodes Agiles[9].

Comme évoqué précédemment, ce diagramme propose une vision théorique que l'on adapte en fonction de nos besoins.

**Phase d'analyse et d'inventaire**

Cette phase est la première dans notre cas aussi. Il s'agit là de comprendre le fonctionnement du logiciel. Une prise en mains des technologies utilisées suivie de l'analyse du code et des structures de données.

**Phase de construction de la documentation**

Au fil de l'analyse décrite ci-dessus la documentation est apportée et enrichie. D'une part l'intégration de commentaires dans le code, permettant notamment de générer une

9 Methodes Agiles : methodes de développement et de gestion de projets informatiques se voulant plus pragmatiques que les méthodes traditionnelles. Elles cherchent notamment à impliquer au maximum les différents acteurs du projet. Elles reposent sur le principe de cycles de développements incrémentaux, adaptatifs et itératifs. (Source : Wikipedia.fr)

documentation Java. D'autre part une documentation rédigée sous forme de descriptions des principes de fonctionnement du système ou de descriptions de certains éléments importants.

**Phase de restructuration du code**

Cette phase permet de réorganiser le code, sans impacter le fonctionnement du système. Des tests sont donc effectués en parallèle afin de vérifier que l'on ne régresse pas fonctionnellement. Il s'agit notamment de scinder les méthodes et les classes trop longues, de les renommer si besoin, d'éliminer les redondances ou encore de rendre le code plus générique pour une meilleure évolutivité dans le temps.

**Phase de reconstruction des données**

Il s'agit ici d'intervenir sur la structure des données. En fonction de la restructuration du code réalisé précédemment, la structure de données pourra être modifiée. Dans le cadre de cette étude, peu de modifications sont envisagées à ce niveau. Cependant, certaines évolutions du système peuvent mener à l'intégration de nouveaux éléments dans la structure de données.

**Phase d'intégration de nouveaux éléments**

Il s'agit là d'apporter des nouvelles fonctionnalités. Celles-ci peuvent avoir un effet sur la structure de données, plus en tant qu'ajout de nouveaux éléments qu'en tant que modification d'éléments existants. Même s'il n'est pas nécessaire d'avoir effectué toutes les phases précédentes sur l'ensemble du système, il est évidement préférable que les évolutions s'appuient sur des « bases saines », soit un code évolutif. Cela ne signifie pas non plus que ce même code ne doive pas un jour ou l'autre repasser par une phase de restructuration de code.

## 2.2 Conception d'une architecture applicative et logicielle

Notons tout d'abord que la littérature à tendance à diverger dans la manière d'exposer les concepts d'architecture applicative et logicielle. Nous proposons ici de prendre pour référence le cours « NFE 107 » sur l'urbanisation et l'architecture des systèmes d'informations (2013), dispensé au CNAM (cf : Chapitre 5). L'architecture d'un système peut-être abordé sous deux points de vue. Le point de vue logique et le point de vue physique :

- Le point de vue logique permet de représenter le système au niveau de ses fonctions, indépendamment de toutes considérations physiques. Dans notre cas, cette vue logique sera représentée par un modèle en couches, chacune ayant un rôle particulier.

- Le point de vue physique détermine les composants matériels comme les serveurs et les technologies adoptées. Cette vue sera représentée par des niveaux communément appelés les *tier*. En l'occurrence un *tier* peut correspondre à une ou plusieurs couches.

Le schéma ci-dessous (illustration 3) représente une architecture classique basée sur cinq couches et sur lequel sont représentés les « *tiers* ». Étant donné qu'un *tier* est une unité physique particulière, il est intéressant d'indiquer à ce stade la nature du *tier*. Le schéma ci-dessous s'appuie donc sur une architecture applicative de type « Java Enterprise Edition » (JEE), technologie mise en œuvre dans le projet.

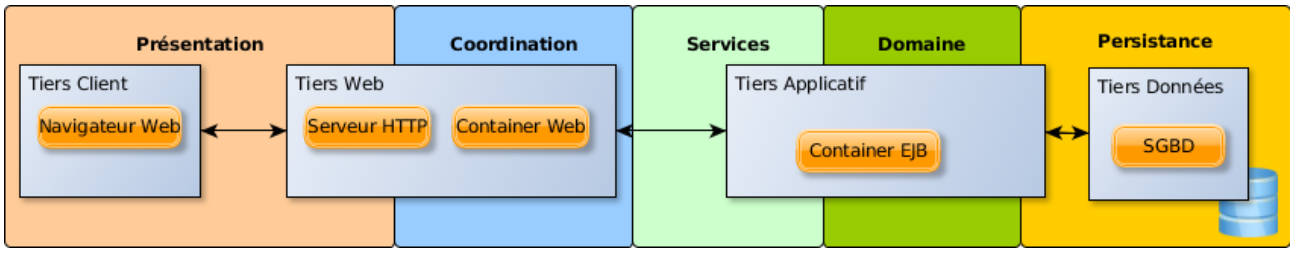

*Illustration 3: Architecture en couches + tiers liés à la technologie "JEE" (adapté du cours NFE107)*

### 2.2.1 Point de vue logique - Principes d'un modèle en couches

Le principe du modèle en couches consiste essentiellement à séparer les responsabilités pour mieux gérer la complexité d'un système. Lorsque que nous parlons d'architecture applicative en couches, nous distinguons généralement 5 couches. Même si globalement le rôle des couches ne fait pas débat au sein de la littérature, il y a malgré tout diverses manières de les décrire et de les nommer. Dans ce cadre, nous nous satisferons de l'approche décrite dans le schéma tout en gardant à l'esprit qu'il en existe d'autres.

Une couche peut être considérée comme un sous-système. L'empilement des couches permet d'isoler les responsabilités au sein de packages d'une même catégorie.

Comme le souligne Bailet (2012), afin de limiter les dépendances entre les couches on veille à ce qu'elles se fassent du bas vers le haut. Dans ce cas une couche a besoin de la couche inférieure pour être compilée et non d'une couche supérieure. On évite ainsi l'effet « spaghetti », en interdisant pratiquement les dépendances cycliques entre packages, minimisant ainsi les impacts d'un changement dans une couche.

Par ailleurs le passage d'une couche vers une autre doit se faire au travers des interfaces.

L'objectif est donc d'appliquer ces principes dans la mise en œuvre du projet Pub/Sub.

Les cinq couches présentées sont les couches de bases, fondamentales au système. Cependant il est possible d'ajouter des couches supplémentaires, appelées « couches transverses ».

**Couche « présentation »**

Cette couche est en charge de la manière dont l'utilisateur va interagir avec le système. C'est la couche de plus haut niveau. C'est dans cette couche que l'on va retrouver les interfaces graphiques. Dans le cadre d'une application Web, cette couche sera notamment en charge de la production des pages Web.

**Couche de « coordination » (ou couche de contrôleur)**

Cette couche est en charge de la gestion des liens entre l'utilisateur et la couche « services » (ou « métiers » si la couche « services » n'est pas présente). Elle permet de gérer les accès, les éventuelles erreurs et exceptions levées, la mise à jour des interfaces graphiques. C'est souvent au travers de cette couche ainsi que de celle de présentation que le « motif de conception » (*design pattern)* « Modèle Vue Contrôleur[10]» (MVC) est mis en œuvre.

**Couche « services »**

Cette couche est en charge des liens entre le contrôleur et la logique métier de l'application. Elle représente l'implémentation des cas d'utilisation. En ce sens elle doit implémenter la logique métier pour exposer des services à diverses applications clientes (une console, un navigateur Web, une application mobile). C'est notamment dans cette couche que l'on retrouvera les Web Services.

**Couche « domaine » (ou métier)**

Cette couche est en charge de toute la gestion des règles métiers de l'application. Elle implémente toutes les règles de gestion, les processus, les traitements de l'application. Elle manipule aussi toutes les données.

**Couche « persistance »**

Cette couche est responsable de la communication avec les supports de stockage des données.

10 Le motif de conception « Modèle-Vue-Controleur » est généralement mis en œuvre pour prendre en charge les interactions utilisateurs :
- le Modèle représente l'ensemble des composants qui sont chargés de réaliser des appels à la couche Services et de mettre les résultats de l'appel à la disposition de la Vue
- la Vue représente l'interface utilisateur
- le Contrôleur gère la synchronisation entre la Vue et le Modèle, il surveille les modifications du modèle et informe la vue des mises à jour nécessaires.

Elle prend en charge toutes les opérations de base sur les données (création, modification, suppression, lecture). Elle offre aux couches supérieures un niveau d'abstraction des données, appelé Data Access Object (DAO).

**Couches « transverses »**

Ces couches vont prendre en charge des tâches n'impactant pas directement le « cœur » de l'application. Elles sont transverses, car elles peuvent être utilisées par toutes les autres couches selon le besoin. Elles gèrent notamment la sécurité, les logs, la configuration, le monitoring de l'application.

### 2.2.2 Point de vue physique - les « tiers »

La séparation en couches permet d'organiser logiquement l'architecture du système en garantissant les principes de modularité, maintenabilité, découplage, ré-utilisabilité. Afin de garantir une véritable indépendance entre les couches il est important que celle-ci s'opère aussi au niveau physique. On parle alors de niveaux physiques (*tier view*). Les différents « tiers », correspondent à une ou plusieurs couches selon le nombre de « tiers » employés.

- Un modèle « 1-tier » est une unité physique dans laquelle s'exécute toutes les couches (ex : une application monoposte).
- Un modèle « 2-tiers » est représenté par 2 unités physiques dans lesquelles s'exécutent toutes les couches. C'est le cas notamment du modèle « client / serveur ».
- Un modèle « N-tiers » est représenté par un nombre d'unités physiques indéterminés dans lesquelles s'exécutent toutes les couches. C'est généralement ce type de modèle qui est adopté dans les projets Web de dernières générations. Le schéma précédent (illustration 3) est l'illustration d'une architecture JEE où l'on peut voir les différents « tiers » impliqués.

## 2.3 *Big-Data*, généralités

La problématique du projet, face à la quantité de données qu'il doit manipuler, peut-être considéré comme un projet orienté *Big-Data*. La notion de *Big-Data* est malgré tout assez vaste et renferme plusieurs aspects. Il ne s'agit pas ici de fournir une description « anthropologique » du terme, qui risquerait de nous mener dans bien des embarras, mais de fournir quelques éléments de compréhension nous permettant de situer notre projet dans ce

concept un peu large.

La notion de *Big-Data* fait actuellement référence à « l'explosion » de données massives qu'il devient difficile de traiter avec des outils traditionnels. Chaque jour la quantité de données produites ne cesse de croître. Elles proviennent actuellement de sources très diverses (ex : opérations commerciales, transactions financières, réseaux de capteurs, réseaux sociaux). Comme l'indique BERANGER (2014), les systèmes de bases de données relationnelles et les outils d'aide à la décision n'ont pas initialement été conçus pour manipuler de telles quantités et variétés de données. C'est dans ce cadre que sont apparues de nouvelles approches de traitement et de stockage permettant ainsi de faire face à de tels volumes. Les géants du Web tels que Yahoo, Google, Facebook, Twitter, Amazon ont fortement contribué à la mise en place de ces nouveaux dispositifs. On verra à ce titre que nombre des solutions techniques disponibles sur le marché - et que nous utilisons dans ce projet -, sont souvent liées à ces « géants ».

Si l'on souhaite avoir une définition du concept de *Big-Data* on peut s'appuyer sur celle du Gartner (2012), basée sur les « 3V » :

- Volume de données important à traiter : il est souvent considéré en tant que poids (le « téraoctet » est devenu une norme), mais peut aussi être considéré en tant que nombre d'éléments à traiter (10 millions d'éléments est devenu classique).
- Vitesse de production des données : il s'agit de la fréquence à laquelle les données sont générées, capturées et partagées. L'évolution des technologies permet actuellement de produire des données très rapidement (toutes les secondes, voire moins).
- Variété des données : les sources d'informations sont tellement diverses qu'il devient difficile de les structurer pour les traiter plus facilement et rapidement.

Pour répondre à ces contraintes sont apparus de nouveaux concepts, approches et technologies. Toutes pouvant se « cacher » derrière le vocable de *Big-Data* . Comme cela a déjà été évoqué, le *Big-Data* couvre un grand nombre d'applications différentes. Les problématiques, les approches et les outils peuvent varier selon que l'on est, par exemple, dans le domaine de la finance, du commerce, de la recherche médicale ou encore des réseaux sociaux. Pour cela nous laisserons à présent de côté le concept vaste de *Big-Data* pour nous recentrer sur une de ses dimensions qui concerne directement notre projet : celle relative aux

systèmes de gestion de bases de données *NoSQL*.

## 2.4 Stockage de données à grande échelle : les bases *NoSQL*

Les bases de données *NoSQL* (*Not Only SQL*) sont conçues pour gérer des volumes de données à très grande échelle. Elles ont généralement la propriété de pouvoir les répartir sur un grand nombre de serveurs qui les rendent extensibles (*scalable)*. Ces bases ne sont pas relationnelles comme celles reposant sur le langage standardisé « SQL » permettant de manipuler des données dites « structurées ». Les bases *NoSQL* permettent de manipuler des données non structurées.

### 2.4.1 Caractéristiques des bases *NoSQL*

Les bases *NoSQL* ne s'opposent pas aux bases traditionnelles relationnelles (SGDBR). En effet la plupart des SGBDR sont transactionnels ce qui leur impose de respecter les contraintes de : *Atomicity Consistency Isolation Durability (ACID*[11]*)*. Ces propriétés permettent de garantir l'intégrité des données d'un système. Si ces propriétés sont plutôt aisées à garantir dans un système centralisé (un seul serveur), cela est beaucoup plus difficile dans le cas d'un système distribué tout en conservant un niveau de performance acceptable. A l'inverse, les systèmes *NoSQL* étant plutôt destinés à un environnement distribué, ils ne sont pas en capacité de garantir les propriétés « ACID », tout du moins sans compromettre les performances. Pour comprendre les propriétés considérées par les bases *NoSQL*, on peut partir du théorème « CAP » défini par E. Brewer[12] (2000) :

- ***C****onsistency* (C)**:** tous les nœuds du système voient exactement les mêmes données au même moment. C'est la cohérence des données.
- ***A****vailability* (A): la perte de nœuds n'empêche pas les survivants de continuer à fonctionner correctement.
- ***P****artition tolerance* (P): aucune panne moins importante qu'une coupure totale du réseau ne doit empêcher le système de répondre correctement (ou encore : en cas de morcellement en sous-réseaux, chacun doit pouvoir fonctionner de manière

11 **Atomicity :** chaque mise à jour de la base doit être atomique, soit réalisée en totalité ou pas du tout.
**Consistency** : une modification de la base ne peut-être effectuée sans respecter les contraintes d'intégrité référentielle
**Isolation :** les transactions ne peuvent être en concurrence les unes avec les autres
**Durability :** chaque transaction effectuée doit être définitive et ne peut revenir dans son état initial
12 Eric Brewer est chercheur en informatique et professeur à l'université de Berkeley (California)

autonome).

Le théorème « CAP » indique que dans le contexte d'une architecture distribuée il n'est pas possible d'obtenir plus de deux de ces propriétés en même temps. Il est donc nécessaire de faire des choix. Soit « AC » ou « AP » ou « CP ». Si les bases relationnelles implémentent plutôt les propriétés de cohérence et de disponibilité (AC) dans le cadre d'une architecture distribuée, en revanche les bases *NoSQL* implémentent plutôt celles de cohérence et résistance au partitionnement (CP) ou de disponibilité et résistance au partitionnement (AP). A titre indicatif, MongoDB est de type CP.

Partant du principe du théorème CAP, les bases *NoSQL* doivent renoncer à une propriété. C'est ainsi que la propriété de cohérence est généralement reléguée au dernier rang des priorités, alors que celle de disponibilité passe en tête. Pour cela et par opposition aux propriétés « ACID » qu'elles ne peuvent respecter, les bases *NoSQL* sont plutôt qualifiées de « BASE » :

- ***B**asically **A**vailable* (BA)*:* le système garanti une grande disponibilité.
- ***S**oft state* (S*):* l'état du système peut changer au fil du temps et cela sans l'intervention de l'utilisateur grâce à la propriété suivante.
- ***E**nvetually consistent* (E): le système devrait rester cohérent au fil du temps à moins qu'il ne *reçoive* une requête d'un utilisateur qui ne permettrait pas d'avoir les mêmes données en même temps sur chaque nœud du système.

### 2.4.2 Les différents types de bases *NoSQL*

Partant des propriétés précédemment définies pour les bases *NoSQL* chacun des types va apporter des caractéristiques différentes selon les besoins. Les différents types de bases peuvent être classées en quatre catégories.

#### *Les bases de type « clé / valeur » ou associatives*

Il s'agit de la catégorie de base de données la plus basique. Dans ce modèle chaque objet est identifié par une clé unique qui constitue la seule manière d'y accéder. Dans ce modèle on ne dispose généralement que des quatre opérations de base : *Create, Retrieve, Update, Delete.* Ces bases ont l'avantage d'être très performantes en lecture et en écriture et permettent une extensibilité (*scalability )* élevée. On les retrouve très souvent comme système de stockage de cache ou de sessions distribuées, notamment là où l'intégrité relationnelle des données est non significative.

Exemple de base « clé/valeur » : SimpleDB, DynamoDB (Amazon), Voldemort (LinkedIn), Redis.

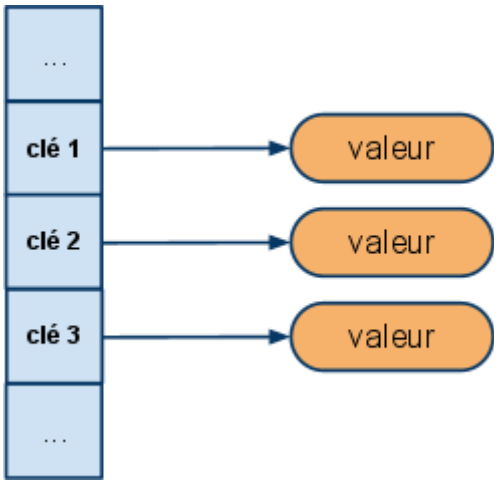


*Illustration 4: Modèle de bases NoSQL type "clé / valeur" (ref: A. Foucret - SMILE)*

***Les bases documentaires :***

Elles sont constituées de collections de documents. Elles offrent des fonctionnalités plus avancées que les bases de type « clé/valeur ». Un document est composé de champs et de valeurs associées. On peut accéder aux données soit par la clé soit par la valeur, ce qui les rapproche des bases relationnelles. Les valeurs peuvent-être d'un type simple (entier, chaîne de caractère, date ,…) ou d'un type composé de plusieurs couples clé/valeur. Le stockage des documents est considéré comme semi-structuré et des fonctionnalités plus avancées que les bases de type clé/valeur. Les documents peuvent être très hétérogènes au sein de la base. Ces bases sont très adaptées et restent performantes pour le stockage de très grandes collections de documents. Les formats de données JSON et XML sont le plus souvent utilisés afin d'assurer le transfert des données sérialisées entre l'application et la base. C'est ce type de bases que nous utilisons dans ce projet. Dans la partie concernant les technologies mises en œuvre nous voyons plus en détail le fonctionnement de MongoDB.

Exemple de bases documentaires : CoucheDB (JSON), MongoDB (BSON)

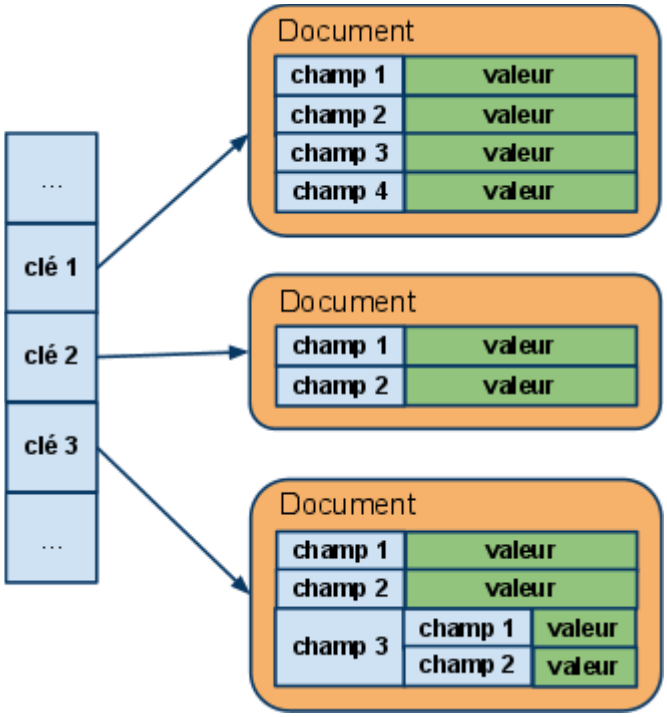


*Illustration 5: Modèle de bases NoSQL type documentaires (ref: A. Foucret - SMILE)*

### *Les bases orientées colonnes*

Ces bases organisent les données en colonne, définie par un couple clé /valeur ; en super-colonne contenant des colonnes et en familles de colonnes regroupant plusieurs colonnes ou super-colonnes. Les familles de colonnes peuvent-être assimilées aux tables dans les modèles relationnels.

Les super-colonnes situées dans les familles de colonnes sont souvent utilisées comme les lignes d'une table de jointure dans le modèle relationnel. Comparativement au modèle relationnel, les bases orientées colonnes offrent plus de flexibilité. Il est en effet possible d'ajouter une colonne ou une super colonne à n'importe quelle ligne d'une famille de colonnes à tout instant sans avoir eu besoin de l'avoir prévu au moment de la conception, comme c'est généralement le cas dans un modèle relationnel. Ces bases sont particulièrement adaptées à l'analyse de données, au stockage distribué et ont une forte capacité à la montée en charge, s'appuyant généralement sur le *framework MapReduce* présenté dans la suite du document.

Exemple de base orientée colonnes : Cassandra (Facebook/Apache), Hbase, BigTable (Google), SimpleDB (Amazon)

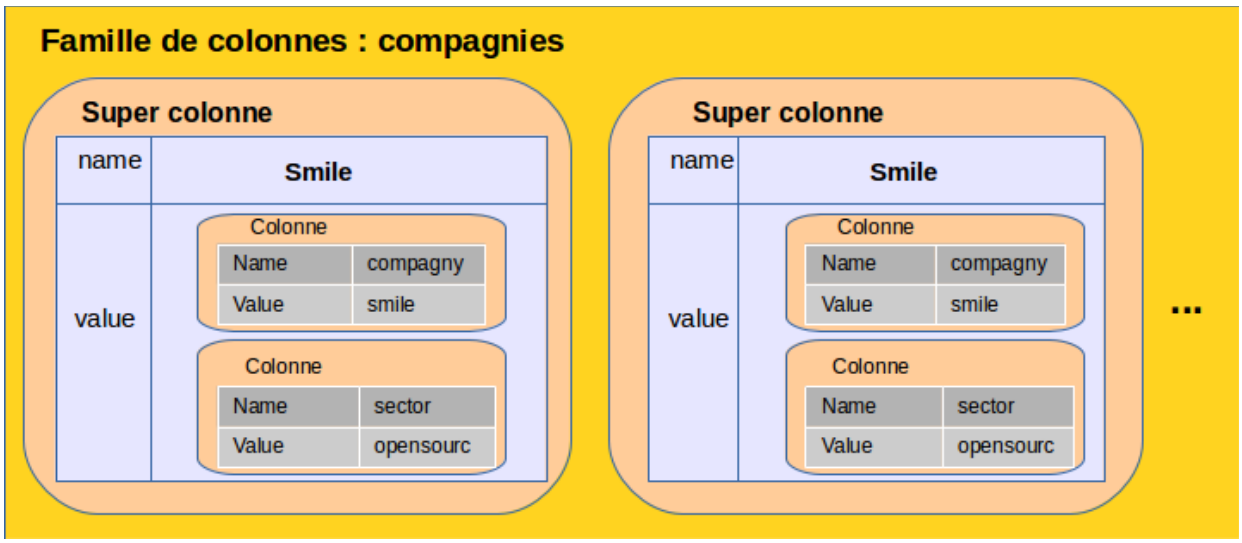


*Illustration 6: Modèle de bases NoSQL orientées colonnes (ref: A. Foucret - SMILE)*

***Les bases orientées graphe :***

Le modèle de ces bases reposent sur deux concepts : les nœuds sont des objets qui se présentent sous la forme d'une base documentaire. Les relations entre les objets (les nœuds) décrits par des arcs orientés et disposant de propriétés (nom, date,...). Ces bases sont très adaptées aux problématiques liées aux réseaux ayant des relations complexes (cartographie, relations entre personnes). Certaines de ces bases respectent les propriétés « ACID ».

Exemples de bases orientées graphe : Neo4j, InfoGrid, Sones GraphDB, HyperGraphDB.

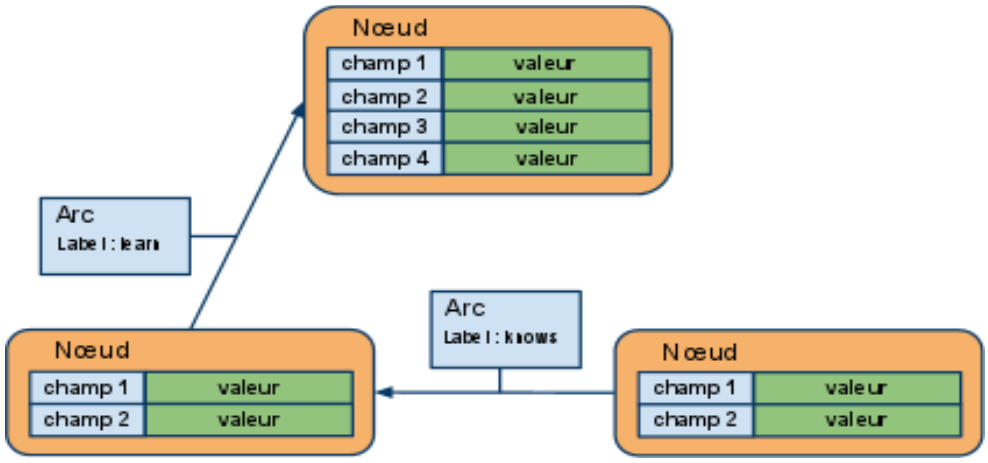


*Illustration 7: Modèle de bases NoSQL orientées graphe (ref: A. Foucret - SMILE)*

## 2.5 MapReduce

*MapReduce* est un modèle de programmation distribuée (qualifié de *framework)* très utilisé dans les systèmes *NoSQL* et sur de très grands volumes de données. Il a été inventé par « Google » et révélé au public à travers la publication *MapReduce : Simplified Data Processing on Large Clusters* (2004), pour répondre à sa problématique d'indexation du Web (des milliards de pages). Le *framework* permet de répartir les traitements sur des *clusters (*grappes*)* de serveurs. Il est ainsi tolérant aux pannes et automatise la parallélisation des calculs sur plusieurs machines, l'ordonnancement de l'exécution des programmes, la répartition de charge et la gestion des communications entre les machines du *cluster.*

### 2.5.1 Principe de fonctionnement

Une opération *MapReduce* fournit le résultat d'une requête en deux étapes. Il s'agit de diviser les données à traiter en partitions indépendantes puis de les traiter en parallèle. C'est l'étape de *Map*. Ensuite, les résultats des traitements sont combinés. C'est l'étape de *Reduce*. Voyons plus précisément comment cela fonctionne.

Etape de *Map* : chaque item d'une liste d'éléments de type « clé/valeur » est passé à la fonction *Map* qui retournera un nouvel élément de type « clé/valeur ». Par exemple : à un couple (utilisateurID, nomUtilisateur), on affecte le couple (rôle, nomUtilisateur). A l'issue du *Map* on obtient une liste contenant les utilisateurs groupés par rôle.

Etape de *Reduce* : la fonction *Reduce* est appliquée sur le résultat du *Map* (une liste d'éléments « clé/valeur ») et permet d'effectuer une opération sur cette liste. Par exemple : compter le nombre d'éléments de cette liste.

L'exemple ci-dessous (illustration 8) permet d'illustrer le fonctionnement de *MapReduce*. Il s'agit de partir d'une collection de commandes effectuées par des clients (*orders*) puis de déterminer le montant total des achats de chaque client. L'opération de *Map* va retourner pour chaque client son identifiant associé à une liste de montant, exemple : {A123 : [500,250]}. L'opération de *Reduce* va regrouper pour chaque client la somme totale des montants, exemple : {id : A123, value : 750}. On remarque bien que le résultat est de la forme « clé-valeur ».

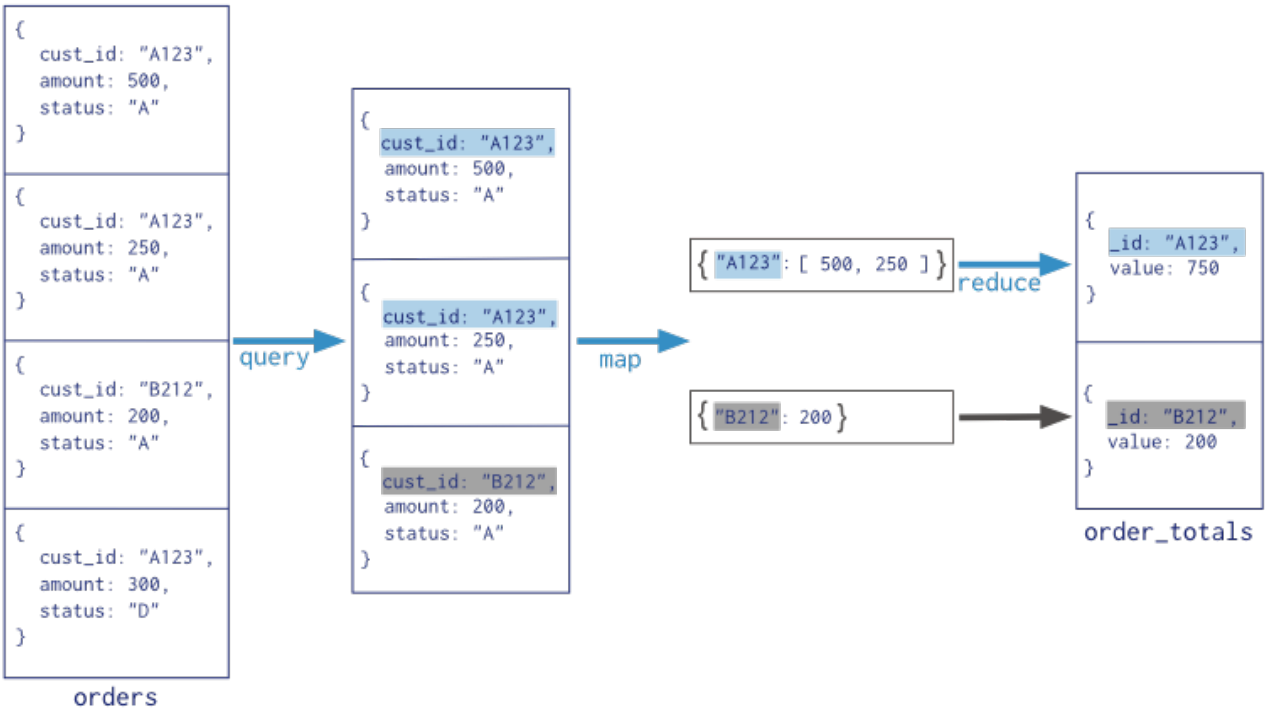


*Illustration 8: Etapes de "Map" et de "Reduce" pour une liste de commandes (ref: docs.mongodb.org)*

# 3. Phase de rétro-conception et état des lieux

Cette phase de rétro-conception a un double objectif. D'une part, permettre la compréhension du code et des technologies utilisées au cours de la « phase 2 » du système *FiND*. D'autre part de fournir des supports (diagrammes et documentation) permettant d'envisager les points d'amélioration.

L'état des lieux du système est réalisé selon quatre points de vue :

- Description des jeux de données utilisés pour les tests : il s'agit des fichiers contenant les souscriptions, les items et le vocabulaire.
- Le code Java réalisé pour la partie applicative du système Pub/Sub.
- Les fonctions *MapReduce* réalisées pour les requêtes à la base de données : elles sont mises en œuvre au niveau du *matching* et du filtrage des items et permettent de déterminer si un item apporte de la nouveauté ou de la diversité à une souscription donnée.
- Les structures des données mises en œuvre en base de données et mémoire centrale : il s'agit de comprendre la manière dont sont structurées les données dans les collections de la base de données MongoDB.

## 3.1 Description des jeux de données

Avant d'étudier les technologies et les algorithmes, rappelons qu'au moment de cette phase d'analyse, les travaux réalisés (code écrit) étaient destinés à des fins de tests sur la base de jeux de données préalablement acquis (ou fabriqués) et pré-formatés.

Nous avons donc en possession 3 types de données :

- Un fichier de souscriptions
- Un fichier d'items
- Un fichier contenant le vocabulaire

**Le fichier de souscriptions :** nous disposons de jeux de données pouvant aller jusqu'à 100 millions de souscriptions. C'est un fichier de type texte. Chaque ligne du fichier correspond à une souscription. Chaque portion délimitée par « ; » correspond à la représentation d'un terme, exemple :

```
24  used 472607
```
=> soit `24 = id_terme , used = le terme,  472607 = le poid du terme`

Soit la 1ere souscription (ligne) comprend 2 termes : used et kill

Extrait du fichier :

```
24   used 472607; 101   kill 225544;
843 iran 57281; 93   run 232356;
```

**Le fichier d'items :** nous disposons de jeux de données pouvant aller jusqu'à 10 millions de d'items. C'est un fichier de type texte. Il a été fabriqué sur la base de flux RSS. Chaque ligne correspond à un item. Une ligne est composée de son « timestamp », permettant d'évaluer son ancienneté, suivi de la liste des termes de l'item séparés par un « ; ».

Extrait du fichier :

```
0;regulator;focus;exchange;traded;funds;etf;according;recent;article
;investmentnew;leverage;...
15000;iz;lsc;andrea;z;terrain;profile;nord;italy;times;hear;pierre;m
arinon;island;weak;signal;....
```

**Le fichier des termes :** nous disposons de jeux de données d'environ 1,5 millions de termes correspondant à l'ensemble des termes contenus dans les items. C'est un fichier de type texte. Chaque ligne correspond à un terme. Une ligne est composée de l'identifiant du terme, du terme, et de son « TDV »

Extrait du fichier :

```
1;new;0.004897788193684461
2;news;0.004897788193684461
3;source;0.004897788193684461
```

## 3.2 Le code Java

C'est la partie applicative du système, qui offre notamment le point d'entrée. On y gère la logique métier de l'application, les interactions avec la base de données et celles avec le client (utilisateur). On effectue aussi une grande partie des traitements, et particulièrement l'intégration du processus complet de gestion du système, soit la prise en charge d'un item dès son arrivée pour aller jusqu'à la notification des souscriptions. Le code Java nous permet aussi

de prendre en charge la gestion des souscriptions de sa réception, qui peut provenir soit d'un fichier pré-formaté destiné aux tests, soit de la saisie dans un formulaire par un utilisateur, jusqu'à son ajout dans la base de données.

Le code Java tel qu'il est réalisé permet d'effectuer globalement trois processus dépendant les uns des autres :

1. **Chargement du fichier des termes en mémoire.** L'accès aux termes étant tellement fréquent qu'il est plus efficace de charger l'ensemble des termes en mémoire centrale. Ceci est surtout rendu possible par le fait que la liste des termes dispose d'une très faible croissance comparée à la liste des souscriptions et des items. Cette liste restera aussi d'une taille relativement limitée, au point de pouvoir être contenue en mémoire.

2. **Ajout du fichier des souscriptions dans la collection Subscription.** Ce processus nécessite que la liste des termes soit préalablement chargée en mémoire pour pouvoir s'exécuter.

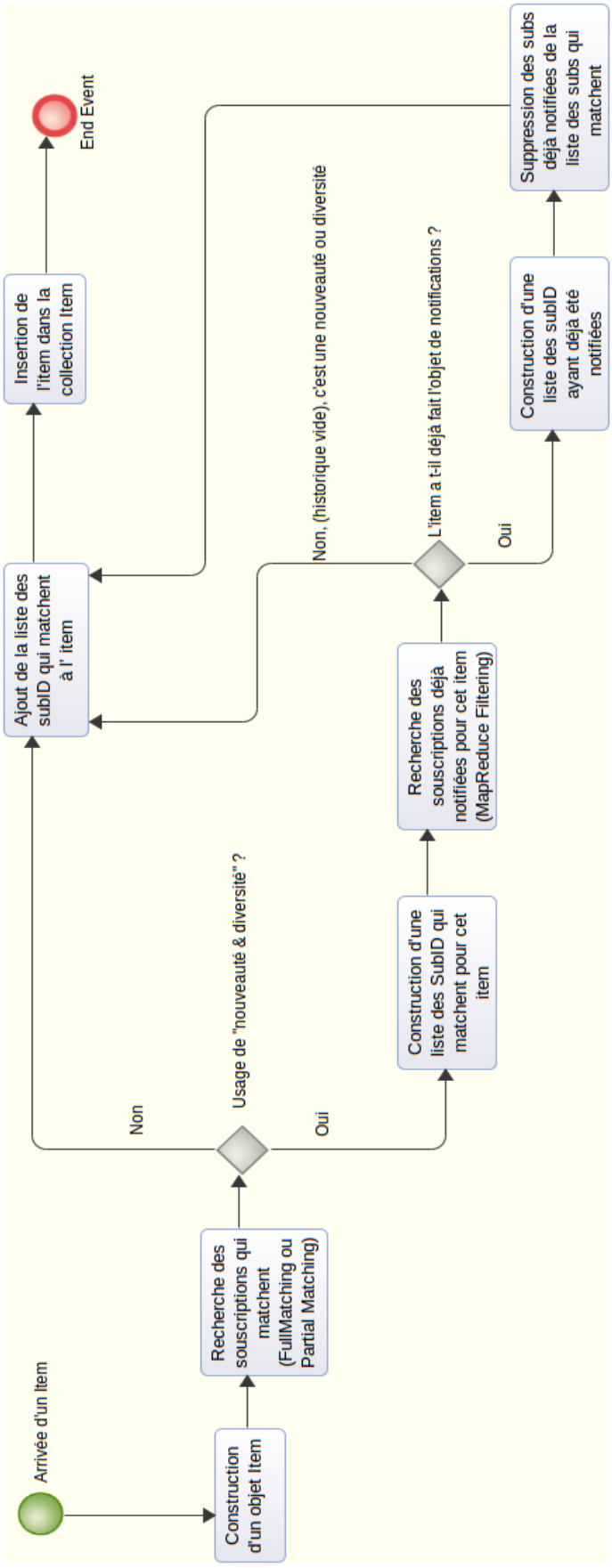


*Illustration 9: Processus de traitement d'un item ("matching" + "filtering")*

**3. Phase de traitement d'un item (*matching* et filtrage).** C'est le processus principal du système représenté par l'illustration 9. Il s'agit de charger chaque item du fichier de manière séquentielle et de vérifier pour chaque souscription, que cet item répond aux critères d'une souscription. Si oui, alors on vérifie s'il apporte de la nouveauté ou de la diversité à cette souscription. Ce processus a naturellement besoin que les deux processus précédents soient préalablement exécutés pour pouvoir fonctionner. Nous n'allons pas ici décrire l'ensemble du code méthode par méthode. Cependant observer l'architecture globale et analyser quelques classes permet de mieux percevoir les améliorations possibles à envisager.

### 3.2.1 Modèle de paquetage

On observe 5 *packages sur l'illustration 10*:

- Le *package com.mongodb* ne fait pas partie des développements du système à proprement parler. Cette librairie Java d'interfaçage avec le serveur MongoDB est fournie par l'éditeur de *MongoDB*. En l'occurrence nous ne faisons qu'utiliser ses classes et ses interfaces, d'où le fait qu'aucune flèche ne partent de ce package.
- Le *package util* a essentiellement deux rôles et est relativement transversal à l'ensemble de l'application. Il permet de définir la configuration globale du système en définissant notamment des variables d'environnement évitant ainsi d'inscrire en dur les paramètres du système dans les classes. Il gère aussi la connexion au SGBD et l'accès aux collections.
- Le *package matching* est en quelques sortes le cœur du système puisqu'il est en charge de tout le processus de *matching* et de filtrage des items avec les souscriptions.
- Le *package entity* se charge de la représentation métier d'un Item. C'est finalement le seul objet métier clairement représenté dans le système.
- le *package converter* gère essentiellement le formatage des données et son injection dans les médias de stockage, soit en mémoire centrale pour les termes soit dans le SGBD pour les souscriptions.

**Premières observations**

Ce modèle semble être assez simple en ne comportant finalement que 4 paquetages. Deux paquetages comportent même qu'une seule classe. Pour autant comme on le voit dans la suite de l'analyse, certaines classes méritent d'être découpées pour apporter de la lisibilité et de la flexibilité. Le fait que seul l'item soit représenté en tant que classe métier peut poser quelques

problèmes pour l'évolution future du système. Comme cela a déjà été évoqué auparavant, la logique même du système repose sur trois types de données fondamentales : les items, les termes et les souscriptions. Ces deux dernières ne sont pas explicitement représentées. On peut aussi remarquer qu'il n'existe pas une hiérarchie très explicite dans ce modèle comme le préconise, par exemple, un modèle en couches. On remarque notamment une forte dépendance entre tous les packages au regard de leurs liens. En outre aucune interface n'est mise en place permettant ainsi de limiter le couplage entre les paquetages et d'offrir plus de souplesse pour l'intégration d'implémentations supplémentaires.

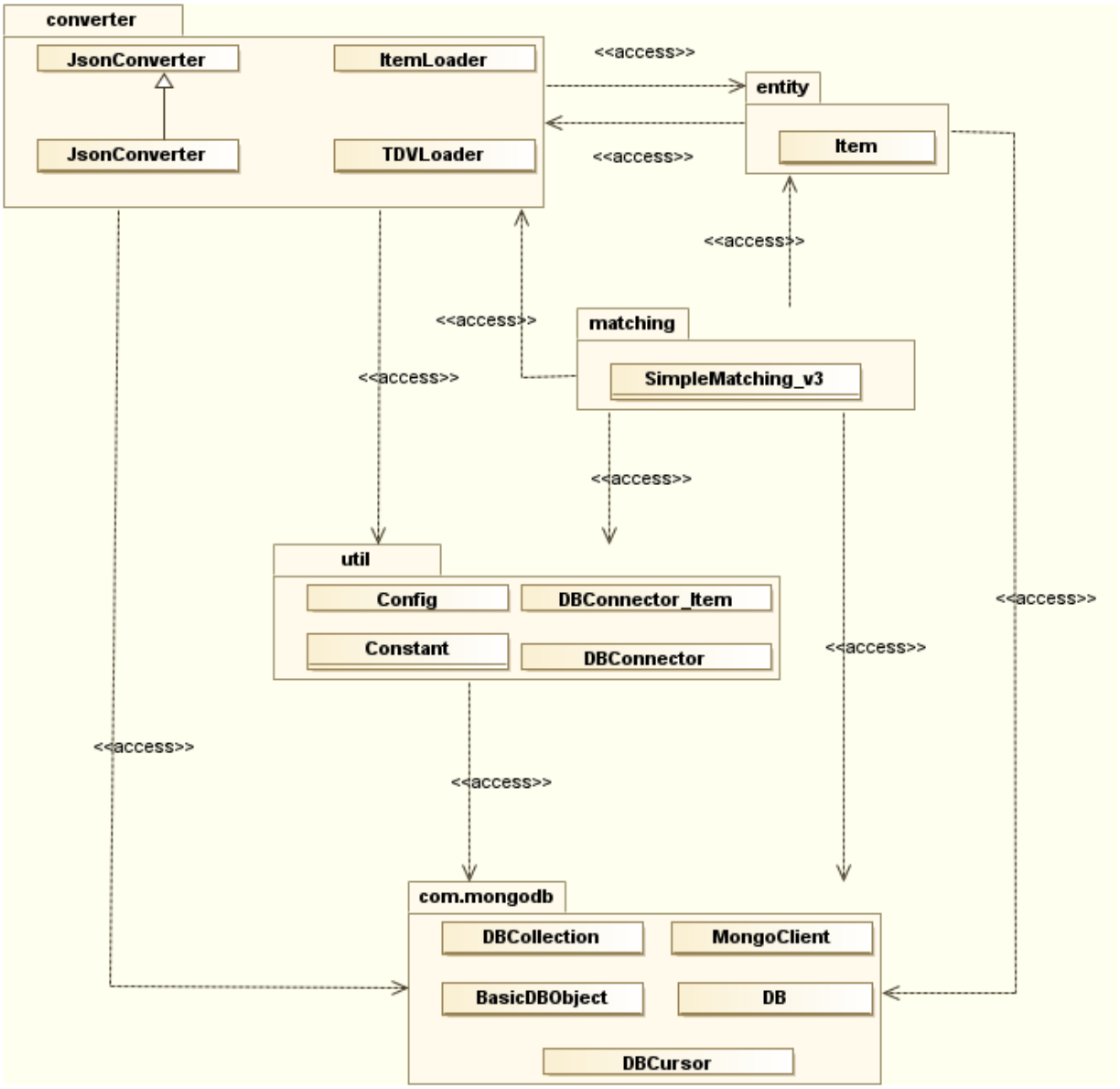


*Illustration 10: Diagramme de paquetage de la « phase 2 »*

### 3.2.2 Le modèle de classes

Le modèle de classes suivant (illustration 11) nous permet d'avoir une vue plus fine sur l'architecture et sa complexité. Pour des raisons de lisibilité, le modèle de classes proposé est allégé. Seules les classes déterminantes dans le fonctionnement du système ont été conservées. Autrement dit les liens des classes de configurations et de gestion des logs n'apparaissent pas. Ces dernières peuvent être considérées comme des classes transverses au système uniquement là pour en simplifier la gestion. Elles sont aussi en lien avec tellement de classes que la lisibilité du diagramme aurait été compliquée.

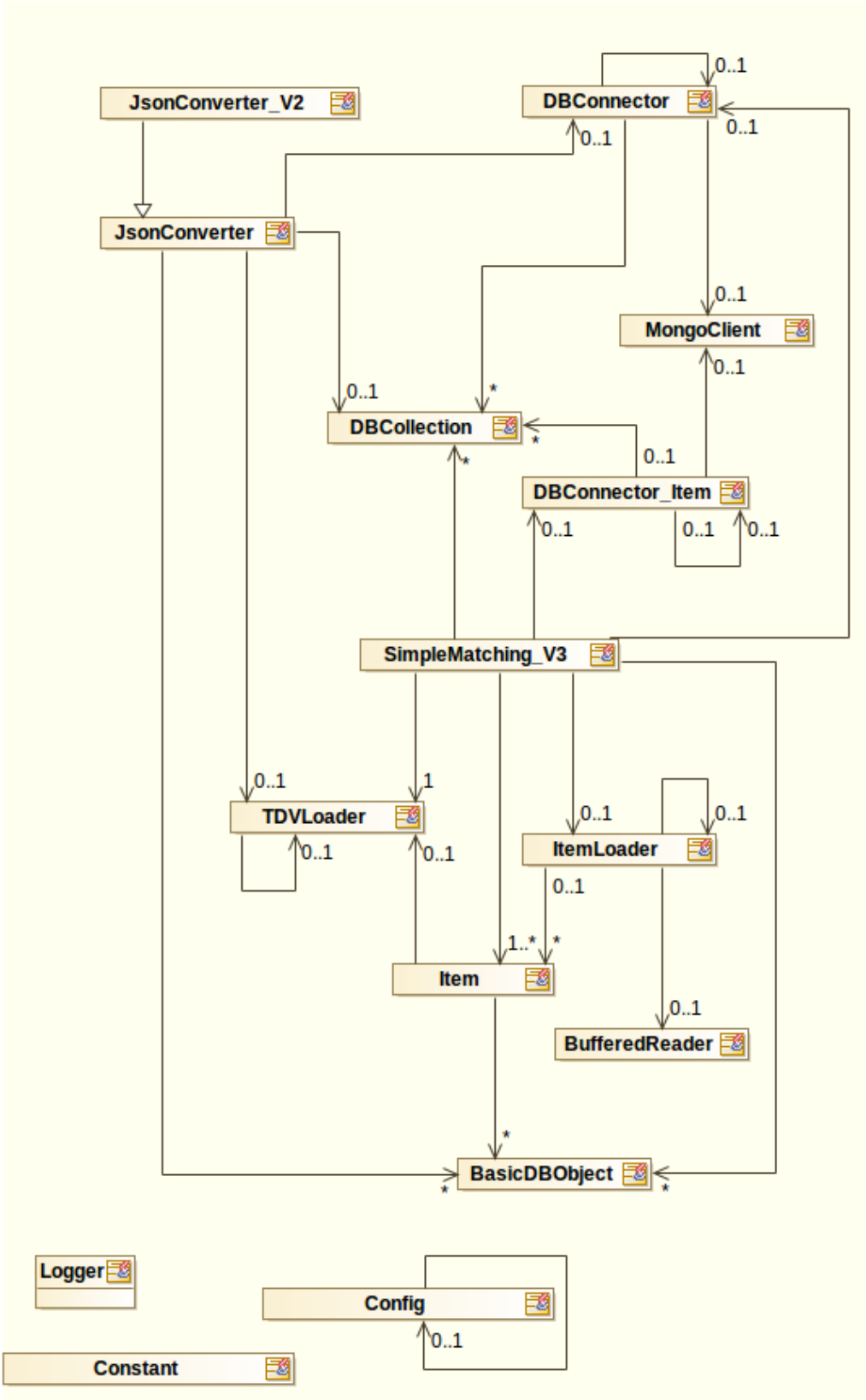


*Illustration 11: Diagramme de classe de la « phase 2 »*

**Observations**

Comme pour le modèle de package, il apparaît un fort couplage entre les classes. D'une part parce qu'aucune interface n'est mise en place mais aussi parce que le nombre de liens est important entre certaines classes. Cela augmente fortement la complexité du modèle qui devient difficile à maintenir ainsi que sa lisibilité. La classe SimpleMatching est dépendante de sept autres classes, sans compter les classes utilitaires dont la modification engendre, qui si elles sont modifiées engendrera une modification de la classe. Dans la suite des travaux nous nous interesserons particulièrement sur l'analyse de cette classe et de son évolution. En matière de dépendance, on remarque aussi que les classes sont fortement liées à la librairie (*driver*) MongoJava. Cela signifie que dans l'état actuel du système il est difficilement envisageable de s'appuyer sur un autre SGBD que MongoDB sans gros effort d'adaptation. On verra qu'il est malgré tout difficile de s'abstraire d'un SGBD *NoSQL* tant la normalisation est faible dans ce domaine. Cependant nous tentons d'aller dans cette direction en limitant l'utilisation de cette librairie à quelques classes et à une seule couche (DAO).

Aussi les projets d'évolution du système comme l'intégration d'une couche graphique pour gérer et exploiter le système sont difficilement envisageables dans le contexte actuel, notamment si l'on fait le choix d'utiliser un *framework* qui impose certain principe de fonctionnement.

Enfin, pour faciliter la reprise par une autre équipe de développement et la maintenance dans le temps on veille à respecter quelques principes de développement notamment :

1. Nommer explicitement les classes.
1. Éviter la duplication de code. S'appuyer éventuellement sur l'héritage ou des patrons de conceptions l'exploitant.
2. Découper les « grosses » classes en plusieurs petites, quand cela est possible, afin de distinguer les responsabilités et permettre ainsi de nommer les classes explicitement.

Le modèle de classes actuel ne respecte pas tout à fait les trois principes évoqués. Pour le comprendre nous décrivons succinctement l'utilité des classes.

**Description succincte des classes et observation**

Package MongoDB

Les classes et interfaces utilisées dans ce paquetage servent à la connexion au SGBD, à

manipuler et créer les objets d'une collection (les documents dans le langage JSON). Ces classes ne sont pas modifiables puisqu'elles sont fournies par l'éditeur et que nous les utilisons telle qu'elles.

Package « util »

**DBConnector :** classe permettant de se connecter au SGBD et de retourner la collection Subscription. Le *patrtern* « Singleton » est utilisé afin de ne pouvoir instancier qu'une seule instance de cette collection dans le système.

**DBConnector_item :** cette classe joue le même rôle que DBConnector mais pour la collection Item.

remarque : on observe clairement une duplication de code, à priori inutile, dont la logique consiste à créer une classe de connexion pour chaque collection.

**Config et Constant :** ces classes offrent les outils nécessaires au paramétrage du système. La classe *Constant* va permettre de définir la valeur des constantes du système comme le chemin des fichiers de données ou encore le nombre de souscriptions à tester. *Config* va surtout permettre de charger « en mémoire » le fichier des propriétés du système (`config.properties).`

Package « converter »

**JsonConverter :** cette classe permet de charger à partir d'un fichier de type texte les souscriptions dans la collection Subscription. Elle crée donc les objets (ou document JSON) de type DBObject pour les insérer dans la collection.

**ItemLoader :** classe permettant de charger les items à partir d'un fichier de type texte et passer chaque item à la classe Item.

**TDVLoader :** classe permettant de charger en mémoire centrale la liste des termes à partir d'un fichier de type texte.

Package Item

***Item*** : classe permettant de créer un objet Item (ou document *JSON*) de type *DBObject* pour être inséré dans la collection *item* . Construit ou modifie l'historique de l'item.

Package Matching

***SimpleMatching_V3* :** cette classe est probablement la plus complexe et la plus dense. Elle gère à elle seule l'ensemble du processus de *matching* et *filtering* tel que décrit dans la figure 1.

**Analyse de la classe SimpleMatching_V3**

Cette classe étant tellement déterminante et centrale pour le fonctionnement du système, qu'il semble important d'en décrire son principe. Les diagrammes de séquence correspondants à cette classe (Annexe 1 et Annexe 2 ) permettent d'en comprendre les détails.

Rappel des paramètres impliqués dans la notification :

- *Full-Matching :* ce paramètre détermine si une souscription peut-être notifiée en vérifiant si l'ensemble des termes d'une souscription est un sous-ensemble des termes d'un item.
- *Novelty :* ce paramètre, appliqué après le *Full-Matching*, permet d'éviter la redondance d'une même information lors des notifications. En l'occurrence un item ayant déjà fait l'objet d'une notification ne sera plus notifié même s'il apparaît dans un autre flux.
- *Diversity :* ce paramètre permet d'apporter de la variété à un sujet. L'item retourné doit se différentier de ceux ayant déjà fait l'objet d'une notification. Autrement dit l'item devra apporter de l'information complémentaire mais non redondante.

Sachant que le paramètre de *Full-Matching* est toujours un préalable à l'application des paramètres de *Novelty* et de *Diversity*, la classe SimpleMatching_V3 va gérer le processus complet de filtrage d'un item, comme décrit par l'illustration 9. Elle combine les paramètres pour générer les listes des souscriptions notifiées pour l'item traité. Cette classe déclenche les fonctions *MapReduce* (requêtes sur les collections de base de données MongoDB) de *Full-Matching, Novelty et Diversity.*

Nous décrivons, ici, uniquement le fonctionnement et l'organisation de manière succincte de cette classe afin d'en tirer un premier bilan :

- *main()* : c'est le point d'entrée de l'application. Lance l'exécution de la methode startTest() pour chacun des types de Matching : *fullMatching, fullMatching + novelty, fullmatching + novelty + diversity.*

- *startTest()* : charge l'ensemble des items en mémoire centrale, crée les connexions aux collections Subscription et Item et lance *testUnit()*.
- *TestUnit()* : c'est la méthode qui va permettre d'itérer le processus sur chaque item. Ainsi, pour chaque item elle exécutera en premier la méthode de recherche des souscriptions répondants aux critères, soit le *fullMatching*. En second elle lancera la méthode de filtrage pour déterminer si l'item apporte de la nouveauté ou de la diversité à la souscription, au cas où la souscription a déjà fait l'objet d'une notification (si la souscription n'a jamais été notifiée on s'arrête au *FullMatching,* car le filtrage n'est pas nécessaire).

Les autres méthodes de la classes vont se charger :

- d'écrire les fonctions de *Map* et de *Reduce*, soit techniquement parlant, elles retourneront des chaînes de caractères
- d'exécuter ces fonctions de *Map* et de *Reduce* par le biais des méthodes du paquetage MongoJava
- d'écrire des résultats dans les collections
- quelques autres méthodes, que l'on peut considérer comme « annexes » vont se charger de la création d'index sur les collections ou encore du nettoyage des collections, préalable à un nouveau test.

Exemple du fonctionnement simplifié de l'exécution d'une fonction *MapReduce* utilisant la librairie MongoJava :

```
 // création de la fonction Map
String Map = "function(){var new_item_ids = ["+item.getListTermIDs()+"] ;
                          fullMatching_v3(this, new_item_ids) ;}" ;
// création de la fonction Reduce
String Reduce = "function(key, values) { return key; }";


// creation de la commande MapReduce
MapReduceCommand cmd = new MapReduceCommand(collection_Subscription, Map,
Reduce, une requête);
```

```
// On applique la commande MapReduce sur la collection Subscription
retournant un élément de sortie.

MapReduceOutput out = collection_sub.MapReduce(cmd);
```

Remarques :

Cette classe très dense regroupe plusieurs groupes de fonctions que l'on pourra découper pour une meilleure lisibilité :

- Le point d'entrée et l'application des paramètres d'entrée du système : le *main()*.
- l'écriture des fonctions *MapReduce* qui retournent des chaînes de caractères.
- l'exécution des fonctions (requêtes) *MapReduce*.
- Les actions (requêtes) sur les collections, comme la création d'index ou la suppression de données.

## 3.3 Les fonctions *MapReduce* du système *FiND*

Les fonctions *MapReduce* sont écrites en JavaScript et vont être déployées sur les différents nœuds du cluster de bases de données (MongoDB dans notre cas). Il existe deux manières d'ajouter des fonctions *MapReduce* au système MongoDB. La manière la plus simple d'écrire et d'exécuter une fonction *MapReduce* est de l'insérer directement dans la console via l'*API* fournie par MongoDB. Cette manière nécessite une intervention humaine, ne permet pas de faire varier les paramètres d'entrée dynamiquement ou encore d'exécuter les fonctions en boucle de manière automatique. Nous devons toujours passer par le programme principal, via le langage permettant de s'interfacer avec la base MongoDB, en l'occurrence Java, pour exécuter les fonctions *MapReduce*, en utilisant, dans notre cas, le *driver* « JavaMongo *»*. Ceci étant, découvrons les deux manières de stocker les fonctions et de les exécuter.

**1. Écriture des fonctions dans leur intégralité à l'intérieur des classes du langage programmation :** dans le cas présent, il s'agit d'écrire les fonctions MapReduce dans les classes Java en utilisant des chaînes de caractères (*String*) qu'il est préférable de placer dans des *buffers* (*StringBuffer*) quand il s'agit de grandes fonctions pour des raisons de performance. C'est la méthode la plus classique que l'on retrouve dans la littérature.

Exemple d'un Map et d'un Reduce dans le fichier Java:

```
String Map = "function(){for(sub in this.Subscriptions){
                  var key = this.Subscriptions[sub].subID;
                  var value = {count: 1, items:[this.itemID]};
                  emit(key,value);}}" ;

String Reduce = "function(subID, values){
                  var val = {count:0, items:[]}
                  for(var i=0; i<values.length; i++){
                  val.count += values[i].count;
                  val.items[i] = values[i].items[0];}
                  return val;}";
```

Ci-dessous un exemple d'exécution des précédentes fonctions *MapReduce* en utilisant la classe *MapReduceCommand* fourni par le drivers « JavaMongo » :

```
MapReduceCommand cmd = new MapReduceCommand(collection,Map,Reduce,
                        "itemsBySub",MapReduceCommand.OutputType.REPLACE,nu
ll);
```

**2. La fonction est stockée à l'intérieur d'une collection spéciale de MongoDB, appelée System.js.** Cette fonction, identifiée par un nom, peut-être ensuite être appelée par le programme.

Exemple : Les fonctions ci-dessous identifiées par "Map_itemBySub" et "Reduce_itemBySub" sont stockées dans la collection System.js.

```
//Fonction Map nombre d'item notifiés par souscription
db.System.js.save({
      _id : "Map_itemBySub",
      value:
            function(item){
                  for(sub in item.Subscriptions){
                        var key = item.Subscriptions[sub].subID;
                        var value = {count: 1, items:[item.itemID]};
```

```
                        emit(key,value);
                }
        }
});

//Fonction Reduce nombre d'item notifiés par souscription
db.System.js.save({
      _id : "Reduce_itemBySub",
      value:
            function(subID, values){
                  var val = {count:0, itemIDs:[]}
                  for(var i=0; i<values.length; i++){
                        val.count += values[i].count;
                        val.itemIDs[i] = values[i].items[0];
                  }
                  return val;
      }
});
```

Dans les classes Java l'écriture des fonctions *MapReduce* se résume à une toute petite chaîne de caractères faisant référence, par son identifiant, à la fonction stockée dans la collection *System.js*.

```
String Map = "function(){Map_itemBySub(this);}";
String Reduce = "function(key,values){return Reduce_itemBySub(key,values);}";
```

Enfin nous utilisons de la même manière la commande Java pour l'exécution du *MapReduce :*

```
MapReduceCommand cmd = new MapReduceCommand(collection,Map,Reduce,
                        "itemsBySub",MapReduceCommand.OutputType.REPLACE,null);
```

Avantages de cette méthode de stockage des fonctions sur le serveur :

- le principal avantage est la possibilité de modifier à « chaud » une fonction ou d'en ajouter une nouvelle. En effet si les fonctions étaient écrites dans les classes Java, une modification ou un ajout de fonction nécessiterait une re-compilation des classes et donc un redémarrage de l'application.
- les fonctions enregistrées dans la collection System.js sont préalablement compilées et réparties sur l'ensemble des machines du cluster et à leur appel elles peuvent être exécutées. Les fonctions ne sont plus qu'en attente de leurs paramètres.
- lors de l'exécution d'une fonction *MapReduce* par le code de l'application (ici Java), les fonctions *MapReduce* incluent dans les *String* sont envoyées au *cluster* du SGBD. Plus la *String* à envoyer est importante plus il subsiste de communication réseau qui peuvent augmenter la latence dans les traitements. Il est donc préférable de n'envoyer qu'une très petite chaîne de caractères sur le réseau.

Limitation de la fonction *Map* :

Il n'est pas possible d'imbriquer une fonction *Map* à l'intérieur d'une autre fonction *Map*, dont le résultat n'est pourtant rien d'autre qu'un tableau de « clé-valeur ». Il est donc nécessaire d'exécuter préalablement une commande *MapReduce*, qui retournera un tableau de clé-valeur que l'on peut passer au *Map*.

### *Fonctions liées à l'étape de matching*

*fullMatching* et *partialMatching* : fonctions de « *Mapping* » utilisées dans la première phase, permettant de rechercher si l'item testé correspond (*match*) avec une souscription. Pour le *fullMatching* la recherche se fait sur l'ensemble strict de tous les mots clés de la souscription. Pour le *partial matching* un sous-ensemble de mots clés (termes) de la souscription présent dans l'item est suffisant pour valider la correspondance. Dans les tests effectués, le *partial matching* n'est pas utilisé, car le résultat renverrait trop de souscriptions « notifiables» et ferait perdre en précision et en performance.

La fonction *Reduce* pour la phase de *matching* est très simple :

```
"function(key, values) { return key; }";
```

La fonction n'effectue aucun traitement mis à part renvoyer un tableau des souscriptions qui

correspond avec l'item testé. Ce résultat sera ensuite utilisé dans la phase de filtrage afin de déterminer si l'item apporte de la diversité  et/ou de la nouveauté à la souscription.

### *Fonctions liées à l'étape de « filtrage »*

Il ne s'agit pas ici de décrire précisément les algorithmes de filtrage qui ont fait l'objet du mémoire de M.Han (2014). Cela dit il semble intéressant d'en comprendre globalement le principe afin de pouvoir les réorganiser si besoin et les manipuler.

Dans la version développée par Miyoung Han, nous retrouvons essentiellement :

- une grande fonction de « *Mapping* » que l'on appellera « *Map_filtering* ». Cette fonction s'appuie sur d'autres fonctions. Le schéma de l'illustration 12 montre les liens entre-elles.
- deux fonctions *Reduce* dont les noms parlent d'eux même : *« Reduce_novelty »* et *« reduce_Diversity »*

**Analyse du *Map* déclenchée par la fonction *filtering***

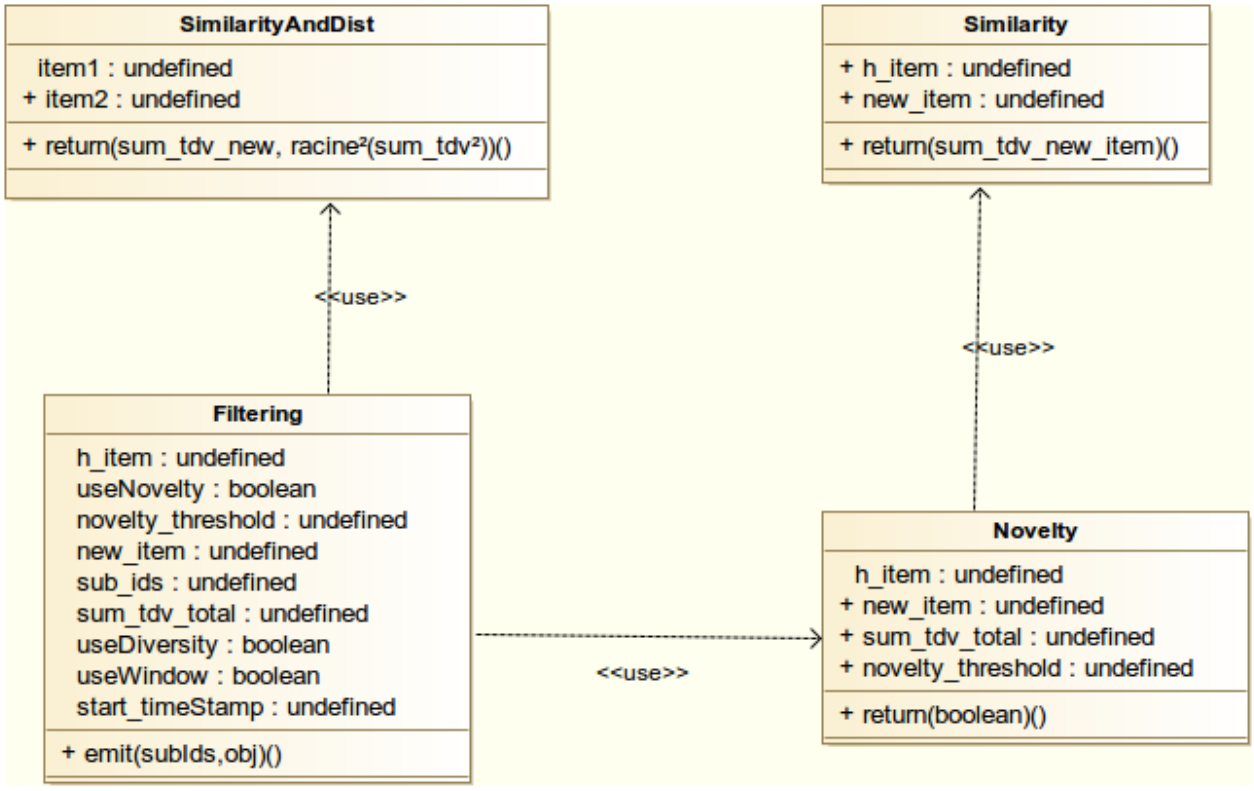


*Illustration 12: Représentation des fonctions utilisées pour le  "Mapping" du filtrage (Map_filtering)*

Le schéma précédent (illustration 12), montre les quatre fonctions impliquées dans le *Mapping* de l'étape de filtrage. La fonction qui réalise le *Mapping*, que l'on reconnaît grâce à la fonction *emit()*, est *Filtering*. Attention tout de même à ne pas s'attacher à la formalisation du diagramme, qui ressemble à un modèle de classe que l'on utilise pour modéliser des fonctions JavaScript. La zone des attributs correspond aux paramètres d'entrées des fonctions. La zone des méthodes correspond aux paramètres de sortie.

Le diagramme d'activité ci-dessous (illustration 13) est une représentation simplifiée du principe de fonctionnement de la fonction *Map_filtering*. La fonction est appliquée sur la collection *item*. Cela signifie que le processus décrit ci-dessous s'appliquera en boucle sur chaque item de la collection.

**Remarques** :

Cette fonction est assez lourde puisque qu'elle est utilisée pour deux cas d'utilisations :

- Vérifier uniquement la nouveauté de l'item. Dans ce cas la fonction *emit()* retournera :
  - clé => un tableau de souscription
  - valeur => novelty=true
- Vérifier la nouveauté et la diversité de l'item. Dans ce cas la fonction *emit()* retournera : soit (ce n'est pas une nouveauté) **:**
  - clé => un tableau de souscription
  - valeur => novelty=false,

  soit (c'est une nouveauté) **:**
  - clé => un tableau de souscription
  - valeur => (novelty=true, distance)

Ces deux cas d'utilisation pourront probablement être scindés en deux fonctions afin d'en améliorer la lisibilité et la maintenance.

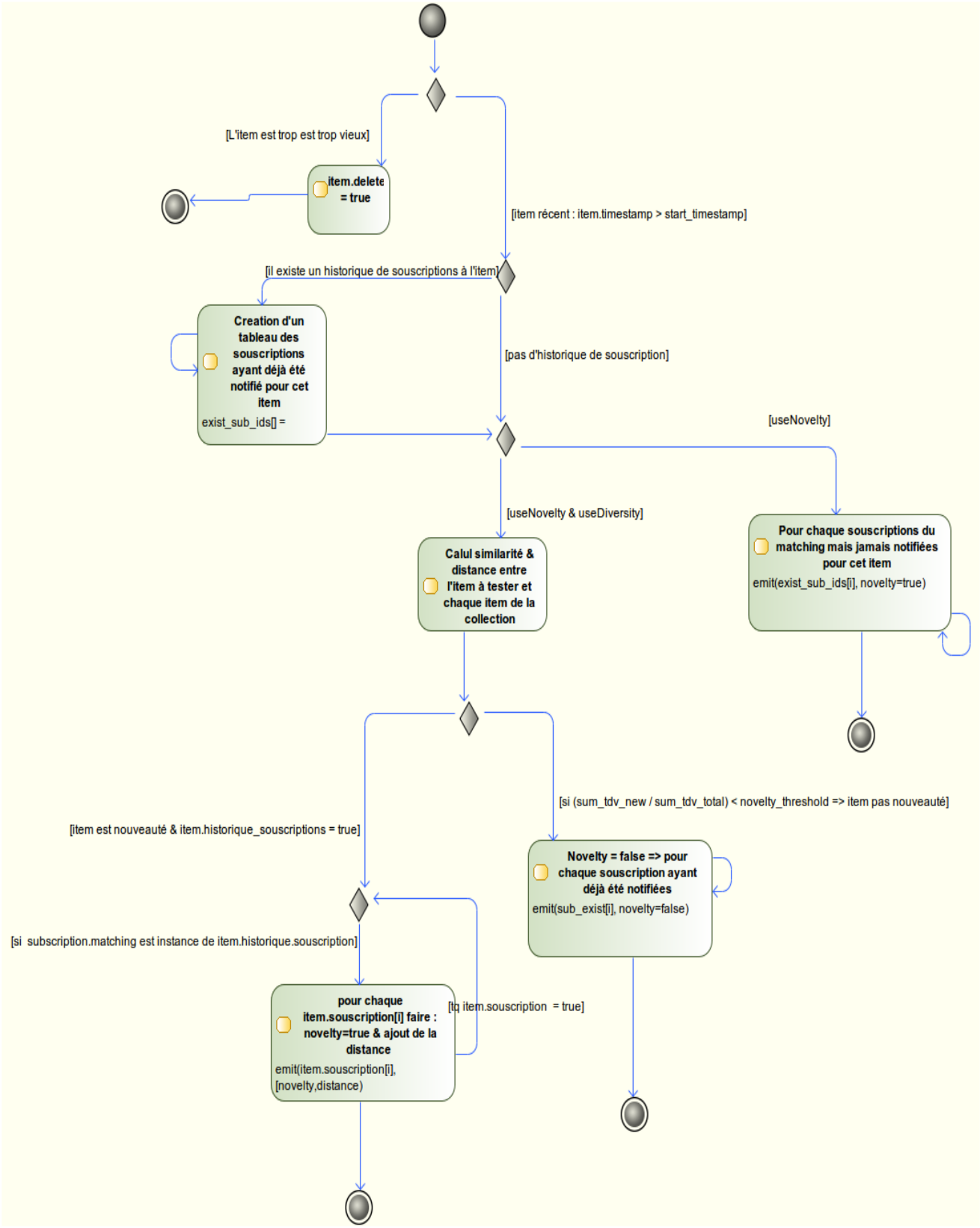


*Illustration 13: Principe de fonctionnement de la fonction de "filtering" (Map)*

**Analyse des fonctions de *Reduce* de l'étape de filtrage**

La fonction `Reduce_novelty` :

Cette fonction qui prend en entrée le résultat du `Map_filtering`, en l'occurrence, une liste de souscriptions chacune associée à une valeur, va permettre de retourner des souscriptions ayant pour valeur :

- soit `novelty=true`. Ces valeurs permettront de notifier ces souscriptions pour l'item en question en tant que nouveautés pour elles.

- soit `null`. Alors la souscription ne sera pas notifiée, car l'item n'est pas une nouveauté.

la fonction *Reduce*`_novelty&diversity`

Cette fonction qui prend aussi en entrée le résultat du `Map_filtering` va permettre de retourner des souscriptions ayant pour valeurs :

- soit `null`. Alors la souscription ne sera pas notifiée, car l'item n'est pas une nouveauté. Il ne peut non plus être une diversité.

- soit un objet contenant la liste des distances entre item. Alors la souscription sera notifiée, car l'item est une diversité.

## 3.4 Structure des données du système

Les données sont réparties sur deux médias de stockage. Le premier, persistant, est la base de données *NoSQL* MongoDB dans laquelle on range les souscriptions et les items. Le deuxième, volatile, est la mémoire centrale dans laquelle on range le vocabulaire. L'illustration 14 montre une représentation conceptuelle du stockage dans la base de données.

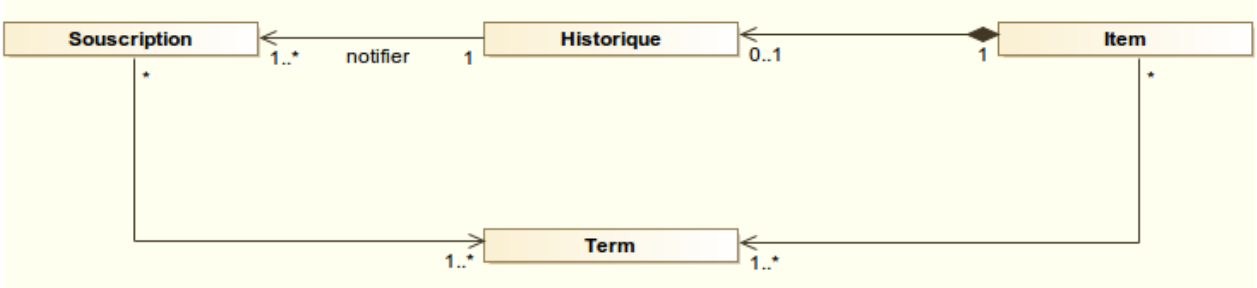


*Illustration 14: Modèle conceptuel des données*

### 3.4.1 La base de données MongoDB

Voyons maintenant comment sont organisées les données dans les collections. Elles sont essentiellement regroupées dans deux collections : Subscription et Item.

**Collection Subscription**

Tel que présenté dans l'exemple ci-dessous, chaque document représente une souscription. Les souscriptions telles qu'on les voit ci-dessous, une fois insérées dans la collection, ne vont plus évoluer. Effectivement, une souscription ne peut être modifiée. On ne peut, par exemple pas ajouter ou retirer de termes. Cela dit, la valeur du TDV ou le rang d'un terme pourrait eux évoluer dans le temps, mais la gestion des termes ne fait pas partie du champ d'étude de ce mémoire.

```
{
	"_id" : ObjectId("54cb646ce4b0c3f426a7a741"),
	"subID" : 19,
	"thresholds" : {
		"matching" : 0.5,
		"novelty" : 0.5
	},
	"max_termID" : 59,
	"max_tdv" : 0.07345766325423794,
	"pair_termID" : [
				[
					59,
					14
				]
	],
	"terms" : [
			{
				"termID" : 59,
				"tdv" : 0.07345766325423794,
				"weight" : 0.9374926938307581
			},
			{
				"termID" : 14,
				"tdv" : 0.004897788193684461,
				"weight" : 0.062507306169242
			}
	 ]
}
```

L'objectif de ce mémoire n'est pas de décrire le rôle précis de chaque champ ou sous-document. Il est préférable pour cela, de se référer aux travaux de recherches de M.Han (2014) « *An Intelligent Publish/Subscribe System at Web Scale, p. 12-13 ».*

Ce que l'on peut retenir de la structure d'un document souscription :

- chaque souscription possède un identifiant.
- on stocke sa liste des termes de recherche.
- on stocke d'autres informations moins déterminantes, comme le *max_termID* qui pourrait très bien être calculé à chaque requête. Ce type de champs permet donc un accès plus rapide à la valeur recherchée.

**Collection Item**

Ci-dessous, la représentation d'un item dans la collection. Le document (au sens JSON) *itemID : 12* présenté ci-dessous, a volontairement été tronqué pour en améliorer la lecture, l'objectif étant la compréhension de la structure de données.

```
{"itemID" : 12,
      "timestamp" : NumberLong(63000),
      "sumDist" : 0,
      "sumDist_temp" : 0,
      "delete" : false,
      "temp" : "",
      "terms" : [
            {
                  "termID" : 2,
                  "tdv" : 0.004897788193684461
            },
            {
                  "termID" : 5,
                  "tdv" : 0.004897788193684461
            },//... suite de la liste des termes
      ],
      "Subscriptions" : [
            {
                  "subID" : 326,
                  "distItems" : [
                        {
                              "timestamp" : NumberLong(63000),
                              "dist" : 0.43195264360894275
                        }
                  ]
            },// suite de la liste des souscriptions ayant été notifiées
plusieurs fois (diversité)
            {
                  "subID" : 176
            },
            {
                  "subID" : 365
            }// suite de la liste des souscriptions n'ayant été notifiées
qu'une seule fois (nouveauté)
      ]
}
```

Comme pour la collection *Subscription*, le détail du rôle des champs et sous-documents est précisé dans le mémoire de M.Han (2014) en pages 13-14.

On peut malgré tout s'arrêter sur les choix faits par les concepteurs de la structure de données. On remarque que l'on stocke pour chaque item la liste des souscriptions notifiées pour cette item. C'est ce que nous appelons ici « l'historique de l'item ». En effet le choix aurait pu être inverse, en stockant à l'intérieur la collection *Subscription* et pour chaque souscription la liste des items ayant fait l'objet d'une notification.

Ce que l'on peut retenir de la structure d'un document item :

- son identifiant
- sa liste de termes
- la liste des souscriptions distinguée par :
    - les souscriptions ayant déjà été notifiées pour lesquelles ont retrouve une valeur de distance permettant, ainsi, de déterminer s'il s'agit d'une diversité.
    - Les souscriptions n'ayant été notifiées qu'une seule fois et s'agissant d'une nouveauté
- le `timestamp` pour déterminer l'âge de l'item
- `delete` pour savoir si l'item fait partie ou non de l'évaluation (déterminé en fonction de la fenêtre d'évaluation et de l'âge de l'item).

**Remarques sur la structure de la base :** nous avions vu qu'il était possible d'organiser les documents d'une collection de plusieurs manières. Ici le choix fait a été d'embarquer les données dans la donnée parente. L'avantage est une lecture plus rapide. L'inconvénient est la mise à jour qui est plus lente. Le problème posé par le système de notification est surtout l'accès rapide aux données stockées dans la base puisqu'il s'agit de savoir très rapidement si une souscription est candidate ou non à une notification en fonction de son historique de notifications. Par ailleurs il n'est pas vraiment question ici de mises à jours des données. En effet le système ne réalise que des insertions de nouvelles souscriptions dans l'historique de notifications des items. Aussi l'opération de dé-normalisation de la base qui consiste à regrouper le maximum de données dans une seule collection quitte à ce qu'elles soient dupliquées, augmente considérablement la vitesse de lecture. C'est pour cela que le choix à été fait de regrouper le maximum d'informations concernant les souscriptions dans la

collection d'items sans se limiter à leurs seuls identifiants.

### 3.4.2 Gestion du vocabulaire

Tel que déjà évoqué dans l'analyse des classes Java, le vocabulaire est stocké en mémoire centrale sous forme d'objet afin de pouvoir être accédé rapidement. Les termes sont les éléments du système qui nécessitent le plus d'accès. Par ailleurs c'est le type de données qui grossit le moins du système. C'est pour cette raison, qu'il est possible et judicieux de charger l'ensemble en mémoire.

***Principe de fonctionnement***

Un fichier de type texte comportant l'ensemble des termes associés à leur TDV est maintenu à jours. À ce moment de l'analyse, il n'y avait pas de gestion automatique du vocabulaire. Une des étapes préalables à l'utilisation du système est le chargement de ce fichier en mémoire centrale.

Un objet *TDVLoader* comportant l'ensemble des termes est composé de la manière suivante :

**1. Un tableau des TDV , `array_tdv`,** de la taille du nombre de termes. Les termes étant déjà identifiés dans le fichier et de manière chronologique à partir du numéro 1, on range les *TDV* de chaque terme dans le tableau à partir de l'indice 1 qui constitue donc l'identifiant du terme.

Exemple :

| Indice 1 : Tdv du terme 1 | Indice 2 : Tdv du terme 2 | Indice 3 : Tdv du terme 3 |
|---|---|---|
| 0.07345766325423794 | 0.00489778819368446 1 | 0.00489778819368446 1 |

**2. Un autre tableau similaire au premier, `array_tdv_square`**, mais comportant le carré des TDV.

**3. Une collection de type « clé-valeur » (*Has*Map), `Map_voca`,** composé du terme en lui-même et de son identifiant. Exemple :

| terme | id |
|---|---|
| new | 1 |
| news | 2 |
| source | 3 |

Une méthode pour charger les termes à partir du fichier dans les tableaux et collection *Map*.
Trois méthodes pour accéder aux différents éléments des termes :

- `getTDV(id)` : retourne le tdv d'un terme à partir de son identifiant
- `getTDVSquare(id)` : retourne le carré du TDV d'un terme à partir de son identifiant
- `getTermID(term)` : retourne l'identifiant d'un terme à partir du terme

Remarque : cette implémentation ne permet pas de récupérer un terme, par exemple à partir de son identifiant.

# 4. Choix technologiques et mise en œuvre

Cette partie est consacrée à la présentation et à la mise en œuvre des différentes technologies déployées dans ce projet. Hormis celles liées au stockage, déjà employées lors de la « phase 2 » du projet, ces technologies font partie des nouveautés de la « phase 3 ». On peut les classer en trois catégories :

- celles liées au stockage et aux requêtes sur les données comme le système de gestion de bases de données (SGBD) *NoSQL* MongoDB ou le *framework* de requêtes *MapReduce*.
- celles liées à la programmation du logiciel en lui-même comme le langage Java, la plate-forme JEE et le *framework* Web JSF,
- celles liées à « l'usine » logicielle comme le conteneur JEE et serveur Web Jetty, le gestionnaire de version Git, l'outil de *buid* Maven, les ateliers de génie logiciel Eclipse et Modelio.

L'illustration 15 montre de manière non exhaustive comment sont imbriquées et intégrées les différentes technologies liées au projet. L'outil Modelio n'est pas représenté, car s'il aide à la compréhension du système en fournissant des modèles, il n'est pas déterminant dans le fonctionnement du logiciel et ne participe pas non plus à la production directe de code. L'illustration 15 offre une représentation dans un environnement à un seul développeur. Dans le cas de plusieurs développeurs, il faudrait envisager un serveur Maven décentralisé du poste du développeur. Nous pourrions ensuite disposer de n-postes développeurs, chacun exploitant les ressources du serveur Maven.

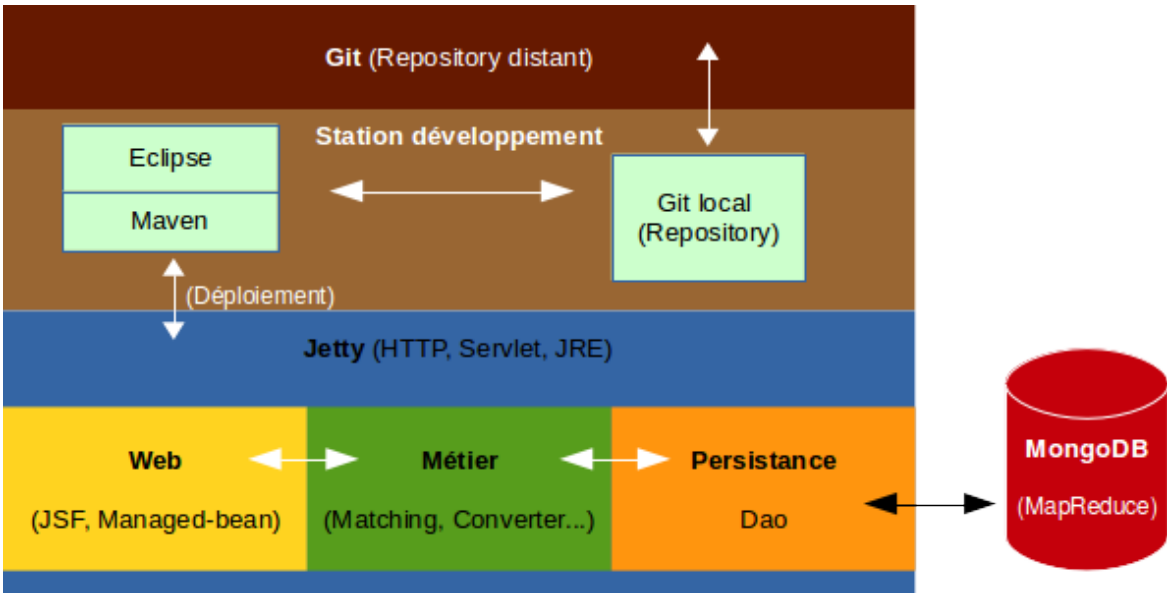


*Illustration 15: Intégration des différentes technologies*

## 4.1 La plate-forme JEE et le *framework* JSF

La technologie mise en œuvre et chargée du *tier* Web dans le projet est JavaServer Face (JSF), un *framework* MVC. Avant de présenter ce *framework* il semble intéressant de présenter succinctement la plate-forme et les principes de JEE. JSF faisant partie intégrante de cette plate-forme, cela nous aidera à comprendre nos choix.

### 4.1.1 Java Enterprise Edition en bref :

JEE est une plate-forme, basée sur le langage Java, orientée serveur pour le développement d'applications distribuées, définissant des spécifications et proposant des API[13] visant à faciliter le développement d'applications d'entreprise. Les développements JEE reposent sur le découpage en « couches » et « tiers » tel que nous venons de le présenter, à savoir la couche « présentation », la couche « métier » et la couche » persistance ».

### 4.1.2 L'environnement d'exécution des applications JEE

Les composants développés à l'aide des API étant de nature différente (un EJB, une Servlet), JEE définit des conteneurs pour chacun d'eux leur permettant de s'exécuter voire de communiquer entre eux à l'aide d'interfaces. On peut citer par exemple les conteneurs orientés Web permettant d'exécuter notamment les Servlet et JSP, qui nous intéressent

13 Une Application Programming Interface (API) est un ensemble normalisé de classes, de méthode ou de fonctions servant de façade par laquelle un logiciel offre des services à d'autres logiciels . (Source : fr.wikipedia.org)

particulièrement dans ce projet. On trouvera aussi les conteneurs d' EJB. Les serveurs d'applications Java peuvent aussi implémenter un ou plusieurs types de conteneur.

### 4.1.3 Les composants Web JEE : Servlets et JSP

Les aspects qui nous intéressent particulièrement sont ceux des couches « présentation » et « coordination » ainsi que tous les éléments liés au *tier* Web. Il existe un certains nombres de technologies basées sur JEE pour réaliser des applications Web. Cependant, la plupart de ces technologies, et c'est le cas notamment de JSF, reposent sur le composant Servlet lui-même souvent associé à celui de Java Server Pages (JSP). Ces composants sont à l'origine de la production de pages Web en Java. Pour comprendre le fonctionnement du *framework* JSF il semble important de comprendre les grands principes des Servlet et des JSP. Rappelons que notre *tier Web* (couches « présentation » et « coordination ») doit implémenter le motif de conception MVC et c'est d'ailleurs généralement comme cela que sont employées les Servlets.

L'emploi des Servlet et des Java Server Pages (JSP) est probablement l'approche la plus basique pour réaliser des applications Web en Java. Si elle permet de bien comprendre le mécanisme de production de pages Web, cette approche n'en est pas pour autant la plus simple et la plus productive. En effet elle nécessite la production de beaucoup de code, car il faut développer, en plus des fonctionnalités propre au projet, un ensemble de fonctionnalités classiquement rencontrées dans la plupart des applications Web. Il faut aussi implémenter le motif de conception MVC.

Rôle et fonctionnement des Servlets et des JSP :

Les pages JSP : s'apparentent à des pages HTML qui prennent leurs valeurs au moment de l'exécution. Les valeurs sont calculées par le programme générant ainsi la page HTML dynamiquement en fonction d'un contexte. C'est cette page HTML qui sera retournée à l'utilisateur via son navigateur Web. La page JSP prend le rôle de la « vue » dans le modèle MVC. Il y a autant de vues qu'il y a de squelettes de pages Web à fournir. (cf : illustration 16)

Les Servlets : elles prennent le rôle du contrôleur. Une Servlet est une classe Java implémentant l'interface « Javax.Servlet ». La Servlet va donc recevoir la requête client (l'utilisateur) qui va personnaliser la réponse. Pour cela, elle va appeler des traitements effectués par le modèle (elle le met à jour) puis elle indique la vue à retourner au client (la page JSP). La Servlet peut éventuellement avoir besoin d'informations provenant de la couche

« métier » pour mettre à jour le modèle (flèche « 2 » dans le schéma de l'illustration 16.

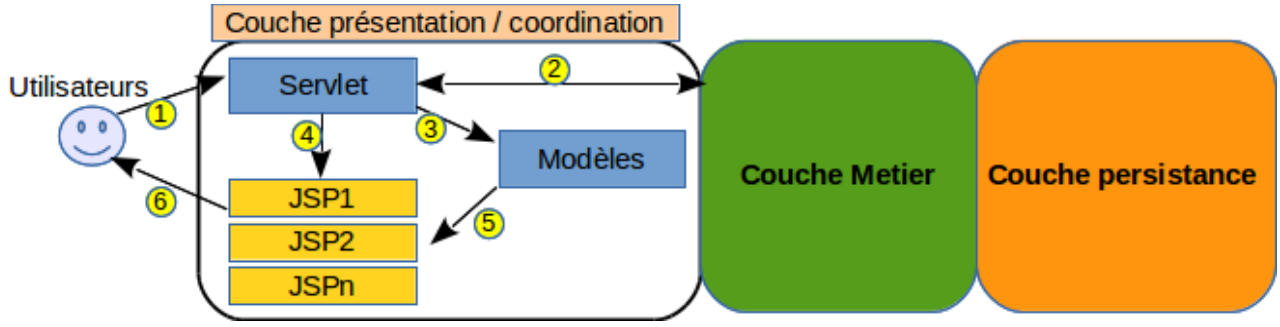


*Illustration 16: Rôle des Servlets et des JSP dans MVC et l'architecture en couches (schéma d'après le cours de Serge Tahé, « Les bases du Web MVC en Java »*

On s'aperçoit qu'il n'y a qu'une seule Servlet. En réalité il devrait en exister plusieurs, sauf que ces Servlet ont toutes le même mécanisme. Seul le contenu de leurs méthodes diffèrent. On est donc tenté de factoriser ce code redondant pour n'avoir plus qu'une seule Servlet et déléguer les traitements réalisés à l'intérieur des méthodes à des classes externes. C'est dans ce contexte que sont apparus des *framework* implémentant le motif de conception MVC et facilitant notamment le travail des développeurs quant à la gestion des Servlet. Nous verrons, après avoir fait un tour d'horizon des différents types de *framework* Java, que JSF s'appuie fortement sur l'approche décrite dans le schéma de l'illustration 16.

### 4.1.4 Les différents types de *Framework* Java

Les premiers *framework* apparus, comme Struts[14] ou plus récemment Spring[15] reposent pleinement sur les Servlet et les pages JSP. Ces *frameworks* même s'ils nécessitent un temps d'apprentissage assez long vont nous permettre :

- de respecter les bonnes pratiques de développement en s'appuyant notamment sur le motif de conception MVC.
- de nous décharger de taches répétitives prises en charges par le *framework.*
- de s'affranchir en grande partie du mécanisme « lourd » des Servlet masqué par le *framework.*

14 Struts met en œuvre le modèle MVC 2 basé sur une seule Servlet faisant office de contrôleur et des JSP pour l'IHM. L'application de ce modèle permet une séparation en trois parties distinctes de l'interface, des traitements et des données de l'application. (source : jmdoudoux.fr)

15 Spring est un *framework* capable de prendre en charge tous les niveaux d'une architecture JEE. En l'occurrence, ce qui nous intéresse ici, c'est la partie Web de Sping qui intègre lui aussi le patron de conception MVC en s'appuyant sur une seule Servlet.

- d'exploiter des librairies fournies en standard par le *framework* comme l'exploitation « d'AJAX [16]» sans avoir à en écrire.

Même si tous les *frameworks* MVC exploitent le mécanisme des Servlet (pas forcément celui des JSP comme nous le verrons avec JSF2), il en existe un grand nombre qui peuvent être classés « globalement » en deux catégories.

### *Les frameworks basés sur les requêtes (ou actions)*

Ces *frameworks* prennent en entrée la requête du client et détermine ce que le serveur doit faire, renvoyant ainsi une réponse en conséquence. On va écrire des classes représentant des actions effectuées par l'utilisateur, comme « ajouter une souscription» ou « voir le détail d'une souscription ». Ces classes sont chargées de récupérer les données transmises via les requêtes HTTP. Globalement on travaille en permanence sur la paire « requête/réponse ». C'est un type de *framework* qui nécessite quand même beaucoup d'écriture de code Java mais aussi l'écriture de « HTML, CSS, JavaScript » destiné à l'aspect des vues. Les *frameworks* tels que Struts ou Spring évoqués précédemment entrent dans cette catégorie.

### *Les framework basés sur les Composants*

Ces *framework* découpent logiquement le code en « composants », masquant ainsi le chemin d'une requête dans une application. À l'inverse des *frameworks* basés sur les requêtes, on va tenter ici de s'abstraire au maximum du concept des requêtes et des réponses. L'application Web pourra ainsi être vue comme une collection de composants présentant leurs propres méthodes de rendu et des actions permettant d'effectuer des tâches. Ces solutions ont été développées pour faciliter les développeurs Java peu familiers avec le développement Web. Les développements d'applications Web avec ce type de *framework* se rapprochent des développements traditionnels d'une application de bureau. Il n'est plus nécessaire de maîtriser le fonctionnement des Servlet comme celui des technologies de présentation Web tel que « HTML, CSS, JavaScript ».

L'avantage d'un *framework* basé sur les composants ne nécessite pas d'écrire autant de code que dans le cadre d'un *framework* basé sur les requêtes. De plus il ne nécessite pas forcément l'apprentissage d'autres langages du Web (HTML, CSS, JavaScript ). Cependant ils offrent

16 L'architecture informatique Ajax (acronyme d'Asynchronous JavaScript and XML) permet de construire des applications Web et des sites Web dynamiques interactifs sur le poste client en se servant de différentes technologies ajoutées aux navigateurs Web. Ajax combine JavaScript, les CSS, JSON, XML, le DOM et le XMLHttpRequest afin d'améliorer maniabilité et confort d'utilisation des applications internet riches. (source : fr.wikipedia.org)

beaucoup moins de possibilités en termes de rendu et de maîtrise du processus de création des pages Web qu'avec les *framework* basés sur les requêtes.

#### 4.1.4.1 Tentative avec « Wicket », un *framework orienté composants*

« Wicket » fait partie de ces *framework* orientés composants correspondant « en théorie » à nos besoins. Ce *framework* aurait pu convenir parfaitement à nos besoins mais s'est avéré présenter deux inconvénients. C'est d'une part un *framework* en perte de vitesse, qui souffre d'une documentation très éparpillée, et d'autre part il n'assure pas la compatibilité d'une version à une autre. On se trouve donc facilement confronté à des documentations d'une version qui n'est plus maintenue, et cela même sur le site officiel. En l'occurrence malgré les efforts pour mettre en œuvre ce *framework*, il semble qu'il ne soit plus vraiment adapté pour un projet nouveau et qui plus est pour des novices de cette technologie.

#### 4.1.4.2 Adoption du framework Java Server Face (JSF)

Avant de rentrer dans les détails de son fonctionnement exposons les principaux avantages et inconvénients qui ont attiré notre attention :

- cette technologie est non seulement toujours maintenue, mais elle fait aussi partie des spécifications JEE à partir de la version « 6 » qui assure une certaine pérennité.
- c'est un des *framework* MVC en Java les plus répandus. Il existe une grande masse critique et beaucoup de retours d'expériences sur cette technologie.
- la documentation est très abondante, surtout en anglais mais aussi en français, et très cohérente par rapport aux différentes versions (1.x ou 2.x).
- cette technologie relevant d'une spécification (JEE), plusieurs implémentations existent et il est relativement facile de passer de l'une à l'autre.
- malgré tout JSF souffre d'une réputation de lenteur (lourdeur) par rapport à d'autres *framework*, notamment ceux orientés « requêtes ». Cependant dans sa version « JSF 2 », celle utilisée pour ce projet, il semble qu'un grand nombre de défauts aient été corrigés.
- Sa mise en œuvre et son mécanisme sont plus complexes que Wicket initialement choisi.

#### 4.1.4.3 Présentation de JSF

Il est important de préciser avant tout la version utilisée. En effet, la littérature peut faire

référence soit à la version « 1.x » du *framework* soit à sa version « 2.x ».

**La version 1,** au-delà du fait qu'elle exploitait le mécanisme des Servlet, elle exploitait aussi celui des JSP. Son fonctionnement, que nous ne décrirons pas ici, engendrait une grande lourdeur et des performances manifestement très critiquées.

**La version 2**, mise en œuvre dans ce projet et apparue en juin 2009, utilise toujours le mécanisme des Servlet, au travers d'une Servlet unique. En revanche elle abandonne l'exploitation des JSP au profit des Facelets en tant que technologie par défaut pour gérer les vues. Des fonctionnalités permettant la création de composants purement XML (appelés composants composites) ont été ajoutées aux Facelets. Ajax a été introduit comme composant intégré à la technologie. Les annotations ont été introduites pour éliminer, autant que possible, les volumineux et verbeux fichiers de configuration autant que possible. Associées à cette version, des bibliothèques de composants graphiques sont apparues s'intégrant parfaitement et sans configuration particulière au *framework*. Celle utilisée dans ce projet est PrimeFaces que nous aurons l'occasion d'aborder dans l'implémentation du *framework.*

JSF s'appuie sur une architecture JEE. Le diagramme ci-dessous (illustration 17), reprenant le schéma de l'illustration 16 et l'implémentation du motif de conception MVC, montre plus en détail les différents *tiers* impliqués dans l'architecture et leur positionnement par rapport aux couches.

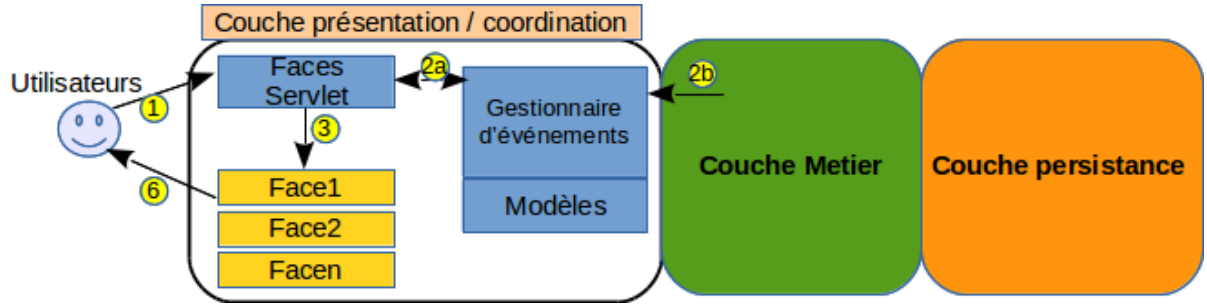


*Illustration 17: Architecture JSF au niveau de la couche présentation (source : http://tahe.developpez.com)*

Dans un projet JSF :

- le contrôleur **C** est la Servlet FacesServlet
- les vues **V** (Face) sont implémentées par des pages *X*HTML utilisant la technologie des Facelets,
- les modèles **M** et les **gestionnaires d'événements** sont implémentés par des classes Java souvent appelées *backing beans* ou plus simplement *beans*.

### *Mise en œuvre de JSF*

Voyons maintenant comment concrètement mettre en œuvre JSF. L'exemple servant de démonstration reprend, de manière simplifiée, un cas du projet.

Avec JSF il y a finalement peu d'éléments à manipuler. Il y en a essentiellement deux, exceptés quelques fichiers de configuration que nous présenterons par la suite. Cette mise en œuvre nécessite d'ajouter préalablement les librairies nécessaires au fonctionnement de JSF. La FacesServlet est un contrôleur unique intégré au *framework*, que nous n'avons même pas à manipuler, chargé d'aiguiller l'intégralité des requêtes entrantes vers les bons composants. Ce principe d'architecture est appelé *Front Controler*.

Le modèle : les classes de type *bean* et plus particulièrement des *beans* managés (*ManagedBean)* par JSF. Ce sont des objets portant des propriétés que l'on peut directement exploiter dans les « Facelets » (les pages *X*HTML dans notre cas). Ces *ManagedBean* doivent respecter quelques règles pour pouvoir fonctionner.

Les vues : JSF peut toujours utiliser les pages JSP pour construire ses vues. Dans sa version 2, le *framework* intègre les Facelets, une sorte de page XHTML reposant pleinement sur le langage XML et plus facile à manipuler que les pages JSF. Ce sont ces dernières que nous utilisons dans le projet. En réalité ces Facelets ne peuvent contenir que des balises prédéterminées. Il est impossible d'ajouter ses propres scripts Java ou JavaScript directement à l'intérieur. La contrepartie de cette contrainte apporte néanmoins une plus grande productivité et de la clarté dans ces fichiers.

### *Production d'une page Web avec JSF*

Déclaration d'un *managed-bean* dans le fichier faces-config.xml

```
<managed-bean>
     <managed-bean-name>configBean</managed-bean-name>
     <managed-bean-class>sys.beans.ConfigBean</managed-bean-class>
     <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

Ce mécanisme est obligatoire soit dans la version 1.x de JSF soit dans le cas de l'utilisation d'un serveur d'application n'intégrant pas de conteneur d'EJB comme Jetty utilisé dans ce projet. Sinon il est possible de se contenter d'une annotation du type `@managedBean` placée avant la classe comme on peut le remarquer dans la classe présentée ci-dessous. Ce Bean se trouve donc dans le paquetage `sys.beans` identifié comme `ConfigBean.Java.` Il est

appelé dans les pages *X*HTML « *configBean* »

**Extrait de la classe « ConfigBean.**Java **»**

```
package sys.beans;

import Java.util.List;
import Javax.faces.bean.ManagedBean;
...

//déclaration du bean pour JSF
@ManagedBean(name="configBean")

public class ConfigBean{

	// champs du formulaire
	private int nbSubscription = Config.NB_Subscription;

	// Getter et Setter

	public int getNbSubscription(){
		return nbSubscription;
	}
	public void setNbSubscription(int nbsub) {
		Config.NB_Subscription = nbsub;
		this.nbSubscription = Config.NB_Subscription;
	}

// autres traitements
...
}
```

Voici un *managed bean* très minimaliste complètement exploitable dans une page *X*HTML.

Dans le projet, nous déclarons systématiquement l'annotation *@ManagedBean* afin d'assurer sa prise en compte en prévision de l'utilisation d'un serveur intégrant un conteneur d'EJB. Dans ce cas, il s'agit d'une part de pouvoir récupérer la valeur de *nbSubscription*, pour l'afficher par exemple, via la méthode *get* standardisée par le *framework*, *getNbSubscription*. D'autre part de pouvoir modifier sa valeur via l'autre méthode standardisée, set, via la méthode *setNbSubscription*. Le champ *nbSubscription* doit toujours avoir une visibilité *private* et ne peut être accessible que par ses méthodes (les Getter et les Setter) portant le même nom que le champ mais précédées soit par *get*, soit par *set*.

**Extrait de la page *X*HTML exploitant le « bean » « configBean »**

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xHTML1/DTD/xHTML1-transitional.dtd">
<HTML xmlns="http://www.w3.org/1999/xHTML"
      xmlns:h="http://Java.sun.com/jsf/HTML"
      xmlns:f="http://Java.sun.com/jsf/core"
      xmlns:ui="http://Java.sun.com/jsf/facelets"
      xmlns:p="http://primefaces.org/ui">

      <!-- ligne 1 -->
      <h:outputText value=" Type de Parametre " styleClass="entete"/>
      <h:outputText value=" Modifier " styleClass="entete"/>
      <h:outputText value=" Valeur en cours " styleClass="entete"/>
      <!-- ligne 2 -->
      <h:outputText value="nb Subscription"  styleClass="info"/>
      <h:inputText id="inputText" value="#{configBean.nbSubscription}"/>
      <h:outputText value="#{configBean.nbSubscriptionDB}"/>

      <p:commandButton type="submit" id="submit" value="Valider"/>
</HTML>
```

L'illustration 18 ci-dessous permet d'avoir un aperçu du rendu de cette page Web . Pour afficher la valeur d'un champ on utilise un composant de type *Output*, ici `outputText` et lorsque l'on souhaite modifier un champ on utilise un composant de type *Input*, ici `inputText`. Le nom défini pour le Bean, déclaré par l'annotation ou dans le fichier faces-config, est utilisé pour y faire référence et ainsi en le concaténant avec le nom du champ on obtient sa valeur *(ex : configBean, nbSubscription)*. Il n'est donc pas nécessaire ici d'appeler la méthode (`get` ou `set`) liée à ce champ, elles sont appelées automatiquement par le *framework* selon le contexte (*input* ou *output*).

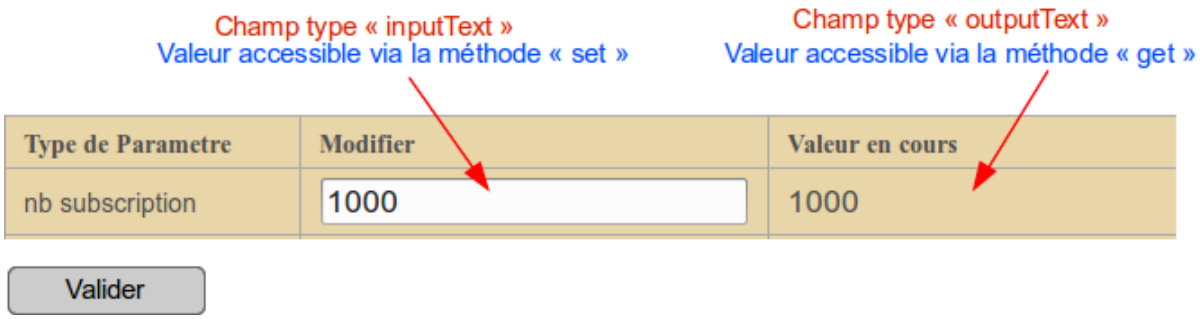


*Illustration 18: Exemple de formulaire et de nature des champs accessibles dans une page JSF*

**Le fichier « Web.xml »**

Comme dans toutes applications Web Java, le fichier Web.xml est utilisé pour la configuration du projet. Dans le cadre de JSF ce fichier ne comporte pas trop d'instructions pour un fonctionnement basique. Le fichier ci-dessous en est l'exemple parfait puisqu'il représente l'intégralité du fichier Web.xml tel qu'il est utilisé dans le projet. Il ne semble pas nécessaire de s'étendre trop sur la nature des différents champs du fichier de configuration, les commentaires servant à les présenter.

```
<?xml version="1.0" encoding="UTF-8"?>
<Web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://Java.sun.com/xml/ns/Javaee"
xsi:schemaLocation="http://Java.sun.com/xml/ns/Javaee
http://Java.sun.com/xml/ns/Javaee/Web-app_3_0.xsd" version="3.0">
<!-- manière de conserver l'état des beans -->
  <context-param>
    <param-name>Javax.faces.STATE_SAVING_METHOD</param-name>
    <!-- valeur possible client / server -->
    <param-value>server</param-value>
  </context-param>
  <!-- mode de développement du projet -->
  <context-param>
    <param-name>Javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <!-- affichage des commentaires -->
  <context-param>
    <param-name>Javax.faces.FACELETS_SKIP_COMMENTS</param-name>
    <param-value>true</param-value>
  </context-param>
  <!-- déclaration de la Servlet unique de JSF -->
  <Servlet>
      <Servlet-name>Faces Servlet</Servlet-name>
      <Servlet-class>org.apache.myfaces.Webapp.MyFacesServlet</Servlet-
class>
      <load-on-startup>1</load-on-startup>
  </Servlet>
  <!-- format des pages JSF -->
  <Servlet-Mapping>
    <Servlet-name>Faces Servlet</Servlet-name>
    <url-pattern>*.xHTML</url-pattern>
  </Servlet-Mapping>
  <!-- durée de validité d'une session -->
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
  <!-- page d'accueil de l'application -->
  <welcome-file-list>
    <welcome-file>admin.xHTML</welcome-file>
  </welcome-file-list>
</Web-app>
```

**Commentaires sur JSF :** on s'aperçoit que JSF une fois compris le principe n'est pas un *framework* très difficile à mettre en œuvre et qu'il peut très bien être utilisé pour de petits projets. Il faut malgré tout garder à l'esprit que ce *framework* masque des concepts bien plus complexes et que l'on peut parfois être surpris par ces comportements. Même s'il n'est pas spécialement utile de maîtriser tous les rouages qui se cachent derrière le *framework* on se rend compte qu'il peut-être parfois intéressant d'en comprendre un minimum les concepts.

### 4.1.5 Intégration d'un outil de *build* : Maven

#### *Présentation générale*

Maven est un outil de *build*, c'est-à-dire un outil permettant de construire des livrables tels qu'un exécutable, une archive de type JAR ou de type WAR destinés à être déployés sur un serveur d'applications. Une des phases les plus importantes de ce processus de construction est celle de compilation des fichiers sources. Cependant Maven, comme d'autres produit similaires, notamment le très populaire mais plus ancien ANT, ne se limite pas à la compilation. Dans ce cadre, certes nous nous appuyons sur Maven puisque c'est l'outil retenu pour ce projet, mais cette présentation reste relativement commune par rapport à l'ensemble des outils de *build* sachant que tous n'intègrent pas toutes les possibilités exposées ici. Maven permet donc d'assurer le cycle de vie complet d'un projet de développement allant de la compilation jusqu'au déploiement sur un serveur d'application.

Dans le cadre d'une compilation voici globalement le processus effectué :

1. préparation et validation des sources en adaptant les propriétés du projet selon son contexte (ex : environnement de développement ou de production).
2. compilation des sources.
3. les phases 1 et 2 sont aussi exécutées sur les sources des tests unitaires (si elles existent).
4. exécution des tests unitaires sur le projet.
5. création des livrables (JAR, WAR, etc).

Tel qu'évoqué précédemment, Maven peut effectuer d'autres tâches comme :

- déployer les livrables créés sur un serveur d'application (ex : Tomcat, Jetty).

- gérer les librairies dépendantes au projet (téléchargement, version).
- création de rapport sur la qualité du code.

Maven va donc nous aider à automatiser un ensemble de tâches facilitant la gestion du projet. La raison pour laquelle Maven a été choisi est certes un peu arbitraire puisque nos besoins sont plutôt basiques et probablement tous les gestionnaires de projets auraient convenus. Cependant Maven semble être l'outil de *build* le plus utilisé dans les nouveaux projets Java. Par ailleurs, son paramétrage est relativement simple, utilisant un principe différent d'autres outils. Ces deux arguments ont suffi à adopter ce produit, nous épargnant une longue phase de comparaisons.

### *Principes de fonctionnement de Maven*

Maven opte pour une description de projet à l'intérieur d'un fichier principal, *pom.xml*. Il repose sur un principe de convention de projet. Cela signifie que si l'on respecte les conventions définies par Maven alors il n'y a rien à configurer. Par exemple, il définit une arborescence du projet. Il ira notamment rechercher les codes sources par défaut dans le répertoire *src/main/Java.* Afin de faciliter le démarrage d'un nouveau projet, une fois le logiciel Maven et les *plugins* nécessaires installés, une ligne de commande permet de construire le « squelette » du projet (ex : l'arborescence, le nom du projet). Un autre avantage très intéressant est la gestion des dépendances du projet. Si l'on observe le cas de notre projet on s'aperçoit qu'il s'appuie sur un certain nombre de librairies dont les versions peuvent évoluer dans le temps (voir même en cours de développement pour des raisons de test).

Extrait du fichier « pom.xml » :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>cnam.fr</groupId>
    <artifactId>pubsub-jsf2</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>
    <name>pubsub-jsf2</name>

    <properties>
        <myfaces.version>2.2.8</myfaces.version>
        <Javax.version>7.0</Javax.version>
    </properties>

   <dependencies>
```

```
        <dependency>
            <groupId>org.mongodb</groupId>
            <artifactId>mongo-Java-driver</artifactId>
            <version>2.12.4</version>
        </dependency>
        <dependency>
            <groupId>Javax</groupId>
            <artifactId>Javaee-api</artifactId>
            <version>${Javax.version}</version>
        </dependency>
    </dependencies>
</project>
```

Les premiers éléments du fichier définissent le projet puis le format du *build* (ici WAR).

Ensuite une clause de propriété optionnelle. Elle est utilisée ici pour définir des variables de versions des dépendances qui peuvent du coup être réutilisées pour plusieurs dépendances. Puis viennent les dépendances en elles-mêmes. On remarque dans l'extrait, l'emploi du *plugin* MongoDB et de l'API JEE. Cela permet de très facilement ajouter une nouvelle librairie ou d'en mettre une à jour. Il suffit d'ajouter les quelques lignes correspondantes à la dépendance (trouvables entre autre très facilement sur le site *http://mvnrepository.com*) au fichier *pom.xml*. Le téléchargement et l'installation des librairies se fait automatiquement.

### 4.1.6 Intégration du gestionnaire de versions Git

Un logiciel de gestion de versions permet d'enregistrer toutes les modifications sur un fichier (ou un ensemble de fichiers) tels que des programmes informatiques. Dans le cas du projet, ce logiciel est utilisé pour suivre l'évolution des scripts du logiciel, mais aussi pour suivre l'évolution de tous les documents annexes au projet. En l'occurrence le projet Modelio, le logiciel utilisé pour réaliser la modélisation comportant les différents diagrammes, sont des projets avec suivi de versions. Voyons les intérêts majeurs d'utiliser ce type de logiciels, presque incontournable dans le génie logiciel :

- possibilité de revenir sur une version antérieure du projet.
- possibilité de stocker le projet et toutes ses évolutions sur un (ou des) serveur(s).
- possibilité de partager un projet à plusieurs personnes.
- possibilité d'être mobile et de récupérer les sources du logiciel n'importe où.

#### *Pourquoi Git ?*

Il existe en effet un certain nombre de logiciels de suivi de versions. On peut citer notamment le très populaire Subversion (SVN) qui lui-même a remplacé CVS (*Concurrent Versions*

*System*), tous les deux reposant sur un principe de dépôt centralisé et unique. Git est plus récent que les deux précédents et s'avère être l'option privilégiée dans le lancement d'un nouveau projet. Même si cet outil renferme une grande complexité de fonctionnement et un grand nombre de fonctionnalités, il peut être mis en œuvre sans trop d'investissement pour une utilisation basique.

**Fonctionnement de Git**

Ce qui a manifestement fait la popularité de cet outil est leur fonctionnement en mode décentralisé (ou distribué), à la différence de SVN. L'avantage est donc de pouvoir disposer de dépôts sur plusieurs machines assurant ainsi une continuité de service. En effet, dans le cadre d'un système distribué tous les collaborateurs au projet (ex : les développeurs) disposent de l'intégralité de l'évolution du projet sur leur machine locale faisant d'elle un nœud du système. Il devient donc possible de travailler sur le projet sans pour autant avoir besoin d'être connecté à un serveur. Il est aussi possible de restaurer le dépôt d'un client sur un serveur, s'il venait à disparaître de ce dernier.

Un autre avantage de Git est la gestion des branches du projet. Il est relativement aisé et quasiment instantané de créer et fusionner des variantes d'un projet. Git gère les branches sous forme de pointeurs, en ne dupliquant que les fichiers variants. À la différence de SVN, Git ne recopie pas l'ensemble des fichiers à chaque nouvelle branche, ce qui accélère d'une part la création de branches et d'autre part allège considérablement le poids des dépôts et les transferts réseaux. C'est probablement la fonctionnalité la plus caractéristique de Git et on est fortement encouragé à l'exploiter.

### 4.1.7 Intégration d'un serveur Java (applications et Web) : Jetty

Lorsque l'on s'oriente vers le développement d'une application Web, le recours à un serveur d'applications est incontournable. Avant tout, il est important de distinguer les rôles entre serveur d'applications et serveur Web. Le serveur Web se charge uniquement du traitement des requêtes HTTP et transmet au serveur d'applications le code qu'il ne sait pas traiter et qui lui est destiné, soit le code Java. Le serveur d'applications, va traiter ce code et retourner une réponse via un « connecteur ». Ces deux serveurs sont souvent embarqués dans le même logiciel, ce qui est le cas de Tomcat et de Jetty, mais pas seulement. Il y a parfois un abus de langage en parlant uniquement de serveur d'applications pour retourner des pages Web. Il existe un certain nombre de logiciels de ce type , pour des utilisations diverses, mais qui se recoupent souvent. Afin de ne pas se perdre dans l'offre disponible - , et nous ne parlons ici

que du monde Java -, il est indispensable d'identifier nos besoins et nos contraintes. En effet, parmi l'offre disponible, on peut facilement se retrouver avec un serveur d'applications, qui fera certes ce dont on a besoin, mais qui surtout en fera beaucoup trop. Le risque est que la solution devienne complexe à mettre en place et qu'elle nécessite beaucoup de ressources matérielles pour un besoin faible. Autrement dit le choix de notre serveur d'application est en relation étroite avec les choix opérés en termes de technologies pour le projet.

Il nous faut donc faire un petit rappel sur les choix retenus. Globalement il y en a deux :

- le cœur de l'application est réalisé en « pur Java ». Autrement dit, si notre application n'avait pas besoin d'interface Web, un simple JRE (Java Runtime Environnement) suffirait à l'exécuter.
- l'interface Web, est développée à l'aide du *framework* JSF, lui-même reposant sur la technologie des Servlets. En l'occurrence un simple conteneur de Servlets suffit à faire fonctionner l'application.

Partant de ces constats, nous pouvons nous permettre de disposer d'un serveur d'applications Web rapide à mettre en œuvre, relativement léger et nécessitant peu de ressources.

**Deux options : Tomcat et Jetty**

Étant donné nos contraintes et nos besoins, deux options « raisonnables » s'offrent à nous, même s'il en existe d'autres. Les serveurs Tomcat et Jetty. Les deux intègrent un conteneur de Servlet et un serveur HTTP (un serveur Web). Ces deux serveurs ne sont d'ailleurs pas vraiment considérés comme des serveurs d'applications étant limités dans leurs fonctionnalités. On les appelle souvent des conteneurs de Servlet. Tomcat, même si c'est probablement le serveur d'applications Web Java le plus utilisé, est plus lourd à mettre en place que Jetty. Il nécessite plus de configuration. C'est aussi le conteneur de Servlet de référence défini par Sun le fondateur de Java. Les deux serveurs ne prennent pas en charge les EJB, même s'il est possible d'intégrer un conteneur d'EJB à Tomcat. C'est effectivement une fonctionnalité qui serait utile au projet, mais pas indispensable. Elle permettrait, notamment, de pouvoir exploiter les annotations déclarant les fameux *managed-bean* indispensable au fonctionnement de JSF, et cela sans avoir à les déclarer dans le fichier de configuration Faces-config.

***Choix de Jetty***

Jetty est donc assez proche de Tomcat, le choix est donc un peu arbitraire, même si les retours

d'expériences semblent donner Jetty comme plus simple et plus rapide à redémarrer.

Ce qui est particulièrement intéressant pour la phase de développement et les tests avec Jetty c'est la possibilité de l'exploiter en tant que *plugin* Maven. Il suffit de l'ajouter en tant que dépendance et le serveur est intégré au paquetage de déploiement, le WAR. Ceci permet réellement de gagner du temps, car il n'est pas nécessaire de reconfigurer quoi que soit lorsque l'on récupère le dépôt sur une station de développement (ex : pour un autre développeur). La version du serveur correspondra toujours à ce qui est indiqué dans le fichier *pom.xml* de Maven.

Un autre point intéressant de Jetty, du fait de sa légèreté, est sa rapidité de redémarrage (~ 3 secondes) et ceci de manière automatique à chaque modification du code.

### 4.1.8 Utilisation d'un outil de modélisation : Modelio

Pour réaliser les divers diagrammes permettant la modélisation de l'application, il est utile de s'appuyer sur un outil conçu à cet effet, intégrants les normes et notations de modélisation. Pour les besoins du projet, la notation UML ainsi que quelques extensions comme l'intégration de la notation BPMN[17] servant à modéliser les processus, sont suffisants.

L'outil Modelio[18] répond notamment à nos besoins, d'autant plus qu'il est disponible sur les plate-formes Windows, Linux et Mac. Le projet étant développé dans son intégralité sur une plate-forme Linux, il n'est pas évident de trouver un outil « abouti » de modélisation fonctionnant sur celle-ci. La conception de ce logiciel reposant sur Eclipse, le fait d'être étant familiarisé avec ce dernier à facilité la prise en main. Aussi, ce produit est distribué sous licence *open-source* et gratuit dans sa version standard.

Modelio a été utilisé pour :

- Les diagrammes de classes, de séquences et d'activités concernant la rétro-conception et la nouvelle architecture.
- la modélisation à partir des sources
- la réalisation de processus utilisant la notation soit la notation BPMN ou les diagrammes d'activités.

17 BPMN : Business Process Model and Notation.
18 Modelio : disponible sur https://www.modeliosoft.com

## 4.2 MongoDB : base *NoSQL* mise en œuvre dans le projet

MongoDB est une base documentaire développée en C++ et fait partie des bases *NoSQL* les plus populaires. Comme la plupart des solutions *NoSQL* c'est un logiciel *open-source.*

### 4.2.1 Structure des données dans MongoDB

Les données sont regroupées sous forme de collections de documents, qui peuvent s'apparenter aux tables dans les modèles SQL. Chaque document est représenté au format JSON et dispose d'une clé unique pour l'identifier dans la collection. Par exemple le document ci-dessous représente l'extrait d'un item dans la collection Item de notre projet, au format JSON. On remarque au passage l'imbrication de sous-document.

```
{"_id" : ObjectId("555773eee4b084e96963ccf9"),
 "itemID" : 12,
 "timestamp" : NumberLong(63000),
 "terms" : [
             {
              "termID" : 2,
              "tdv" : 0.004897788193684461
             },
            ]
"Subscriptions" : [
                    {
                     "subID" : 326,
                     "distItems" : [
                        {
                          "timestamp" : NumberLong(63000),
                          "dist" : 0.43195264360894275
                        }
                       ]
                     },
                    …
                    ]
}
```

### 4.2.2 Architecture de MongoDB

MongoDB peut fonctionner selon plusieurs modes :

- centralisé, sur un seul serveur basé sur le principe d'un client et d'un serveur. Un seul processus nommé Mongod est utilisé et traite directement les données issues des requêtes du client.
- réplication « maître / esclave » : on lance deux serveurs Mongod, un maître et un esclave. Les données sont écrites uniquement sur le maître qui les répliques sur l'esclave. Les accès en lecture sont possibles sur les deux serveurs à condition

d'accepter une certaine latence sur l'esclave due à la réplication du maître vers l'esclave. Étant donné que l'on ne peut écrire que sur le maître, cette configuration pose un problème de tolérance à la panne en cas d'indisponibilité de celui-ci.

- réplication via un ensemble de serveurs traitant les mêmes données (*Replica Set) :* ce mode est plus avancé que celui de maître esclave, car en utilisant un système de réplication sur un minimum de trois machines, il permet d'éviter la présence d'un point de défaillance unique (*Single Point Of Failure – SPOF).* Ce système est donc tolérant à la panne, puisque chaque nœud du système peut prendre le rôle du maître en cas défaillance de celui-ci. Cette configuration s'apparente toutefois au mode maître-esclave où le maître réplique les requêtes et données sur les esclaves. Autrement dit la répartition de charge se fait uniquement en lecture.

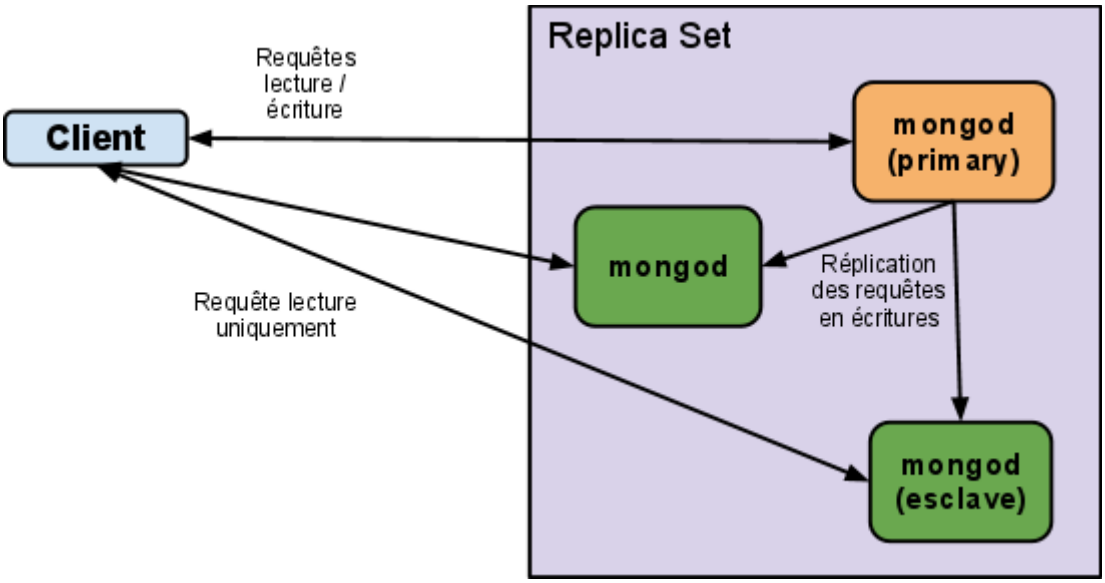


*Illustration 19: Mode Replicat Set dans MongoDB (ref: A. Foucret - SMILE)*

- partitionnement et distribution des données sur plusieurs serveurs *(Sharding) :* l'intérêt de ce mode est justement la distribution des traitements sur les différents serveurs du *cluster*. Chaque nœud s'appelle un *Shard*. Un *Shard* peut-être composé d'un seul serveur, d'un couple client/serveur, ou d'un *replicat Set.* Une collection de documents sera découpée en morceaux (*chunks*) qui seront répartis automatiquement sur les différents *Shards*. Un autre type de serveur, appelé Mongos, est chargé de diriger les requêtes du client vers le *Shard* approprié. Il est conseillé d'en placer au moins deux pour éviter la présence d'un *SPOF. À* partir des versions 2 de MongoDB, ces serveurs sont en charge de l'agrégation des résultats, incluant l'étape de *Reduce*, ce qui signifie

qu'ils sont susceptibles d'être fortement sollicités.

Dans le cadre du projet c'est l'architecture en mode *Sharding* qui est retenue pour la mise en production. C'est en effet la seule parmi les quatre qui permet une distribution des traitements sur toutes les machines.

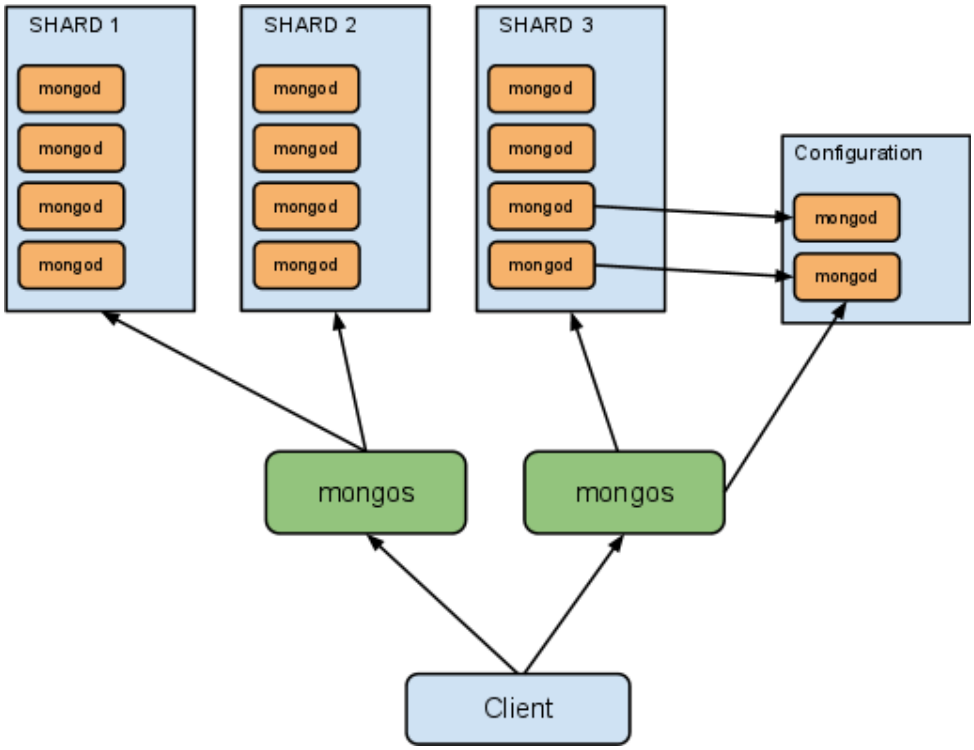


*Illustration 20: Mode Sharding dans MongoDB (ref: A. Foucret - SMILE)*

### 4.2.3 Outils de développements supportés par MongoDB

Un grand nombre de langages sont supportés par MongoDB pour communiquer avec le serveur, dont les plus courants Java, Python, PHP, Ruby, C++, JavaScript. Dans le cadre du projet nous utilisons le *driver* de communication Java. Certains *framework* commencent à implémenter MongoDB sous forme de *plugin*, comme JPA[19] et Spring[20] dans un environnement JEE, permettant de conserver plus d'indépendance entre la partie applicative du logiciel et le serveur de bases de données.

*MapReduce* est aussi implémenté par MongoDB. Les fonctions s'écrivent en JavaScript. Si

19 JPA : Java Persistence API, est une interface de programmation Java permettant aux développeurs d'organiser des données relationnelles dans des applications utilisant la plateforme Java.

20 Spring : framework pour construire et définir l'infrastructure d'une application java, dont il facilite le développement et les tests en s'apuyant sur des POJO qui n'ont pas besoins de s'exécuter dans un conteneur spécifique.

l'on reprend l'illustration 8 sur les étapes relatives au *Map* et au *Reduce*, voici comment seraient écrites les fonctions dans MongoDB :

Les fonctions *Map* et *Reduce*, la requête optionnelle et l'option de sortie :

```
Map = function(){emit(this.cust_id, this.amount);}
Reduce = function(key, values){ return Array.sum(values)}
query = {status : 'A'}
output = 'order_totals'
```

Execution des fonctions, soit le *MapReduce* (*orders* est la collection):

```
db .orders.MapReduce(Map, Reduce, {query, out})
```

# 5. Nouveautés apportées au système Pub/Sub

Cette partie est consacrée plus particulièrement à la présentation des nouveautés. Elle repose évidement sur les phases précédentes, celle de rétro-conception et celle de l'intégration des nouvelles technologies. Les différentes réalisations produites s'articulent autour de trois niveaux d'interventions. L'ordre dans lesquelles elles sont présentées a une relative importance puisque que globalement l'intervention sur un niveau dépend généralement de l'intervention sur un niveau supérieur. Cependant la démarche adoptée est plutôt incrémentale que séquentielle. Par exemple la phase de restructuration du code n'a pas attendu d'être terminée pour commencer à intégrer l'interface graphique même si elle en dépend. Nous avons donc procédé à un système d'aller et retour selon les besoins. Voici les trois niveaux sur lesquels nous intervenons :

- mise en place d'une nouvelle architecture logicielle : nous présentons ici la réorganisation du code existant et la création du nouveau code dans la logique d'une architecture en couches.
- intégration du *framework* Web JSF : nous présentons ici les différentes interfaces Web produites, leur utilité et la manière dont elles ont été implémentées.
- création d'un *workflow* permettant l'ajout, la modification « à chaud » et l'exécution de fonctions *MapReduce* utilisées pour interroger le SGBD. Nous présentons là aussi les interfaces Web produites permettant d'exploiter ce *workflow.*

## 5.1 Phase de restructuration du code

Les remarques émanant de la phase d'analyse nous permettent d'envisager la restructuration du code ainsi que l'évolution de l'application.

Cette restructuration du code n'apporte pas de nouvelles fonctionnalités et ne modifie pas le fonctionnement du système en place. Elle va cependant largement faciliter l'intégration des nouveautés comme l'utilisation d'un *Framework* Web pour la partie interface graphique qui impose une certaine architecture.

### 5.1.1 Points de restructuration envisagés

- Vers un modèle en couches : afin d'assurer une meilleure évolutivité et modularité du système et de limiter le couplage entre les packages.
- Représentation des objets métiers des éléments de données fondamentales au système :

les souscriptions, les items et les termes.

- Découpage et renommage en plusieurs classes de la classe Matching.
- Éliminer les redondances de code : si nécessaire intégration de classes abstraites.
- Intégration d'interfaces offrant notamment la possibilité d'avoir plusieurs implémentations pour certaines parties du code.

### 5.1.2 Nouvelle architecture logicielle

L'architecture d'origine ayant subi beaucoup d'évolutions, nous nous limiterons à présenter des modèles simplifiés afin d'en améliorer la compréhension et la lisibilité.

Nous présentons l'architecture globale dans un premier temps pour ensuite zoomer sur certaines parties qui semblent assez pertinentes pour la compréhension.

Même si nous tentons de respecter les communications entre les couches comme cela est préconisé par les démarches d'architecture logicielle telles qu'elles ont été décrites précédemment, on reconnaîtra que la réalité de l'architecture proposée diverge en certains points avec la théorie. En effet vouloir se conformer strictement à la théorie nécessite, dans le cadre du projet, un alourdissement du code avec potentiellement des risques de baisser le niveau de performance de l'application. Cependant nous nous efforçons d'utiliser une interface pour toutes les communications inter-couches.

Comme nous l'avons évoqué dans la partie précédente, chercher à tous prix à s'abstraire du SGBD (MongoDB en l'occurrence), nécessite un gros effort de réécriture de code. Le système de notifications développé repose, dans sa forme actuelle, sur le *framework MapReduce*. Si le concept *MapReduce* est le même quel que soit le SGBD utilisé, la manière d'écrire les requêtes et le langage utilisé peuvent varier complètement d'un système à un autre. En outre, changer de SGBD reviendrait à réécrire complètement l'ensemble des requêtes *MapReduce*, faisant ainsi perdre la logique de ré utilisabilité du code. Chercher le plus haut niveau d'abstraction possible ne fait donc pas partie des objectifs du projet.

#### *Le modèle d'architecture en couches du projet*

Comparé au modèle en cinq couches, préconisé par la démarche d'architecture présentée précédemment, nous n'en n'avons retenues que quatre comme le montre le schéma ci-dessous. Dans notre contexte, la couche Service ne présentait pas vraiment d'intérêt. Cela dit, il sera toujours possible, et le modèle retenu devrait le permettre assez facilement, d'intégrer

la couche Service si besoin. Pour ce faire, les modifications interviendront alors au niveau de la couche de coordination.

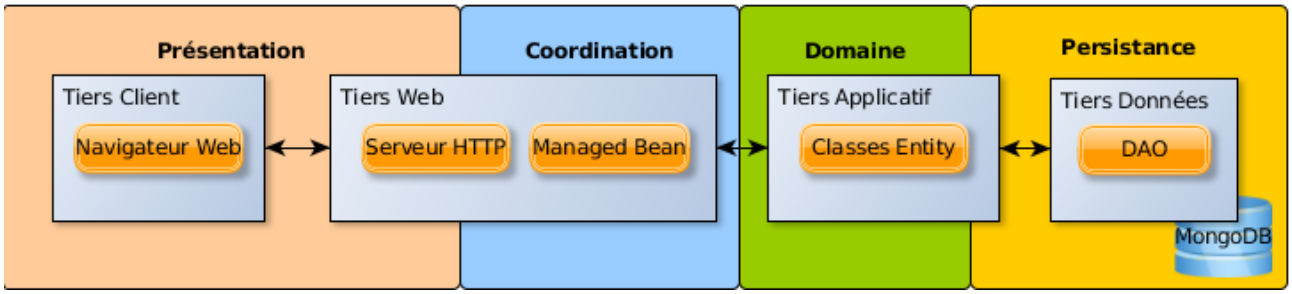


*Illustration 21: Modèle d'architecture en 4 couches + tiers JEE*

Le diagramme de paquetages ci-dessous (illustration 22) va nous permettre d'illustrer nos différentes couches. Pour des raisons de lisibilité, ce diagramme est allégé. La priorité est mise sur la représentation des interfaces quand elles sont présentes. Dans ce cas il faut considérer qu'il existe aussi les classes « concrètes » les implémentant.

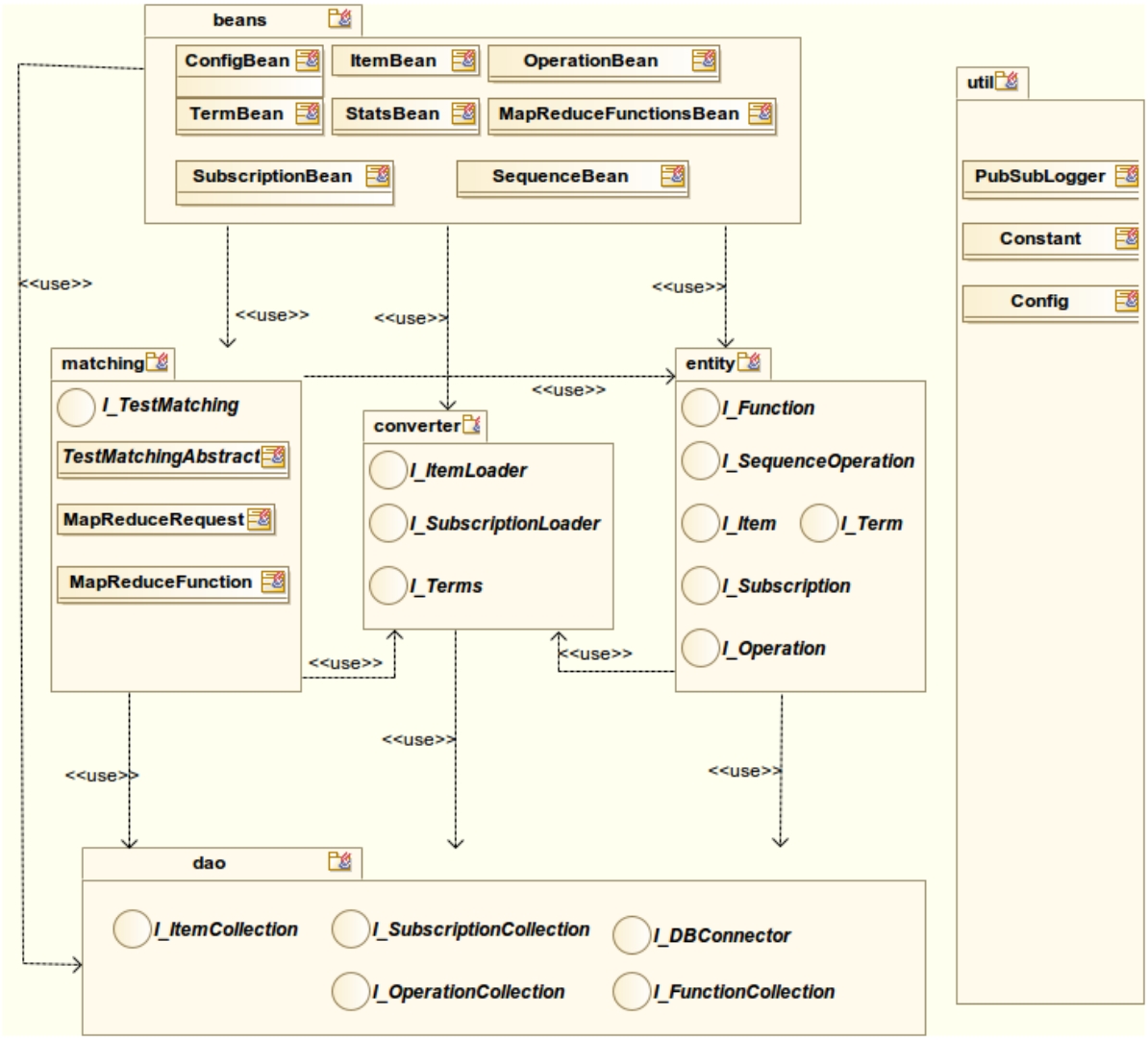


*Illustration 22: Diagramme de paquetages de l'architecture du système PubSub*

Remarque : l'organisation des paquetages illustre bien le sens unique d'utilisation des éléments du dessous. Par exemple le paquetage Beans, impliqué dans les couches Présentation et Coordination, s'il venait à être modifié ou supprimé, ne remettrait pas en question le fonctionnement des autres paquetages.

### *Composition des couches et des niveaux (tiers) mis en œuvre*

- **Présentation et coordination :** ces couches vont gérer toute la partie Web. D'une part, le navigateur Web de l'utilisateur sur lequel nous n'intervenons pas. D'autre part, la partie contrôleur et modèle au sens Modele-Vue-Controleur tel que nous l'avons déjà présenté. C'est au niveau de ces deux couches que le *framework* JSF est mis en œuvre. Nous reviendrons plus en détail ultérieurement sur l'implémentation de JSF. C'est en l'occurrence le paquetage *beans* qui représente le Modèle. La partie contrôleur du MVC est masquée dans le *framework*.
- **Domaine :** c'est au travers de cette couche que l'on retrouve toute la logique métier de l'application. Elle est représentée par les paquetages *matching*, *entity* et *converter*.
- **Persistance :** cette couche prend en charge toutes les interactions avec les supports de stockage. Le package *dao* est le seul à accéder à la base de données pour réaliser toutes les opérations de bases (lecture, création, modification, suppression), ainsi que la création des index dans la base.
- **Couches transverses :** nous plaçons le paquetage *util* à ce niveau étant donné son rôle transverse.

### *Remarques concernant le diagramme de paquetages*

Il ne semble pas nécessaire de revenir sur le rôle des paquetages qui existaient déjà dans la première architecture. Concentrons-nous plutôt sur les évolutions. Nous laissons de côté pour l'instant le paquetage *beans*, impliqué dans l'interface Web, puisque nous y reviendrons plus en détails quand on abordera le *framework* JSF.

**Paquetage *matching* :** la grande classe SimpleMatching, présente dans la première architecture, a été scindée en trois classes :

- TestMatching : le point d'entrée du test d'un item avec les souscriptions. Deux implémentations de cette classe étant proposées, une classe abstraite est privilégiée

pour éviter la redondance de code.

- I_TestMatching : c'est l'interface de TestMatching exposée pour être utilisée par les couches supérieures.
- MapReduceRequest : toutes les commandes (requêtes) *MapReduce* présentes dans la précédente classe SimpleMatching sont regroupées dans cette classe. Cette classe est utilisée par TestMatching.
- MapReduceFunction : on retrouve dans cette classe les fonctions *MapReduce* nécessaires au *Matching*. Cette classe ne produit que des chaînes de caractères qui sont utilisées par les commandes *MapReduce*.

**Paquetage *converter* :** l'évolution notable de ce paquetage est l'intégration d'interfaces. Cela a permis diverses implémentations. Ce paquetage est chargé du formatage des souscriptions, des items et des termes. Les diverses implémentations permettent la prise en charge des éléments provenant de plusieurs sources (par exemple d'un flux RSS) et non plus uniquement d'un fichier stocké dans un répertoire.

**Paquetage *entity* :** au-delà du fait que des interfaces représentant un item, une souscription ou un terme soient exposées, de nouvelles classes ont été créées. Nous ne les détaillerons pas maintenant car, elles font partie des nouvelles fonctionnalités présentées plus tard. Il s'agit des classes utilisées pour gérer les statistiques et les fonctions *MapReduce*. Elles permettent entre autres de déclencher des opérations *MapReduce* et de configurer un *workflow* de fonctions *MapReduce* imbriquées.

**Paquetage *dao* :** ce paquetage est nouveau. Comme son rôle est essentiellement de s'interfacer directement avec la base de données, la classe de connexion à la base a été placée à ce niveau. Cette classe aà d'ailleurs été rendue générique permettant ainsi de se connecter à n'importe quelle base *MongoDB* et d'accéder à n'importe quelle collection.

**Paquetage *util* :** ce paquetage était déjà présent. Nous le conservons dans son rôle transverse. La classe de connexion à la base de données n'étant pas transverse, elle a été déplacée vers le paquetage *dao*.

## 5.2 Intégration de l'interface Web : le *framework* JSF

Lorsque l'on envisage d'intégrer une interface graphique un certains nombre d'options s'offrent à nous et les choix ne sont pas évidents à faire. Il est donc important de déterminer à l'avance l'objectif de cette interface. Sans rentrer trop dans les détails, il semble important de préciser les besoins et les contraintes. Dans cette partie nous nous intéressons donc à la couche « présentation » mettant en œuvre le modèle de conception MVC et plus particulièrement le paquetage Bean.

Le cahier des charges :

- L'application doit pouvoir être accessible de l'extérieur, c'est-à-dire sur un poste différent de celui où l'application est hébergée.
- Utilisation possible par plusieurs personnes en même temps.
- Préférence pour un client léger (type navigateur Web) afin de ne pas avoir à installer l'application pour chaque utilisateur
- L'interface graphique doit servir :
    - à la configuration de l'application
    - à visualiser des statistiques
    - à ajouter de nouvelles souscriptions
    - à lancer différents services : le *matching*, recharger le vocabulaire, recharger les propriétés
- Les aspects esthétiques de l'application ne sont pas une priorité. Ils ne nécessitent pas d'y consacrer trop de temps.

La nature des besoins nous orientent rapidement vers le choix d'une interface graphique type « client/serveur » et plus particulièrement orientée *Web*. Le langage utilisé jusqu'à présent pour le développement de l'application étant du Java, il nous a semblé cohérent de rester sur celle-ci notamment en termes d'intégration. L'apprentissage d'un nouveau langage ou de nouvelles technologies étant un frein, il devenait judicieux de songer à l'emploi d'outils nécessitant le moins d'apprentissage tant qu'ils répondent aux besoins.

Partant du principe de l'utilisation d'une technologie *Web* en Java, restait à déterminer le

meilleur compromis entre celle qui nécessiterait le moins d'effort, celle qui serait la plus productive et celle qui serait la plus pérenne. En matière de *Web* et de Java et souhaitant s'appuyer sur notre modèle d'architecture, la plate-forme Java Enterprise Edition (JEE) devient incontournable. Dans notre cas nous ne voulions pas trop remettre en question l'architecture de base mise en place. Nous souhaitions pouvoir nous appuyer sur celle-ci de manière complètement indépendante. C'est pour cela que nous avons fait le choix d'un *framework* Java et plus précisément JSF. Les raisons qui ont motivées les choix ont déjà été présentées dans la partie sur les concepts et les technologies mises en œuvre.

### *Mise en œuvre du framework JSF, les pages Web*

Sans revenir sur le fonctionnement du *framework,* qui a été décrit dans la partie « concepts et technologies », nous allons expliquer comment sont développées et intégrées les différentes interfaces graphiques dans le système. Pour rappel, nous souhaitons le maximum d'indépendance avec les couches inférieures de l'architecture, et c'est entre autre pour cette raison qu'elles ont été mises en place. Le diagramme de l'illustration 22 montre les liens entre le paquetage *bean*, correspondant à l'interface Web et les couches inférieures.

#### Le paquetage *bean* et les vues

On retrouve dans ce paquetage l'ensemble des *managed-bean* correspondant au Modèle du *pattern* MCV de JSF (voir illustration 17). Ces *managed-bean* s'appuient sur les classes métiers des couches inférieures. Pour rappel, la page affichée au client (navigateur) est une page *X*HTML composée de balises dont certaines font référence aux composants fournis par les *managed-bean*. Présentons succinctement le rôle de chaque *managed-bean*. Une copie d'écran des pages Web associées aux *manged-bean* est présentée en même temps.

***ConfigBean :*** cette classe met à disposition les composants permettant la configuration du système, le lancement des tâches comme le *matching*, l'ajout des souscriptions dans la collection à partir d'un fichier, le chargement du vocabulaire en mémoire. On peut aussi choisir l'opération *MapReduce* utilisée pour le filtrage des items permettant de réaliser des tests de nature différentes. Ceux-ci sont définis dans la section de gestions des opérations *MapReduce*.

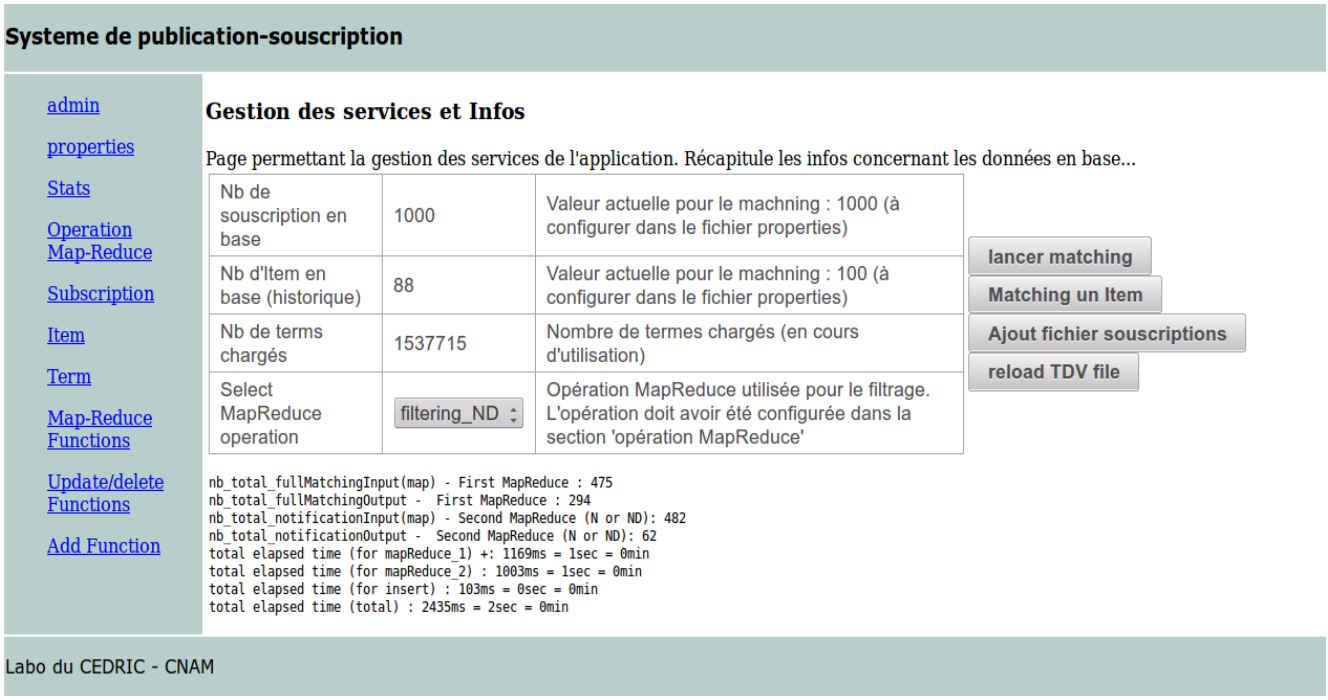


*Illustration 23: Page Web de gestion des services*

***PropertiesBean :*** cette classe permet de gérer les paramètres de l'application. En d'autres termes, elle agit directement sur le fichier de configuration Config.properties, type de fichier très souvent employé dans une application Java. Sans décrire les différents paramètres on trouvera globalement :

- une partie relative à la configuration du serveur de base de donnée.
- une partie relative aux collections utilisées permettant de modifier leur nom (par exemple pour tester des données ou des quantités différentes).
- une partie relative au *Sharding*. Celle-ci permet de définir si l'on travaille sur une base de donnée dans un environnement « mono base » ou si l'on travaille dans environnement de base distribuées. En mode *Sharding*, les collections seront créées pour être distribuées. Dans ce mode on peut aussi définir manuellement une clé de *Shard* agissant sur la performance d'accès aux données réparties sur les différentes *Shard*s.
- une partie concerne la quantité de données utilisée pour les tests (valable uniquement à partie d'un fichier de données). On définira notamment le nombre de souscriptions et items.
- Une partie relative aux paramètres de filtrage : *novelty, diversity*, suppression de

l'historique avant chaque test, mise à jour de l'historique…

admin
properties
Stats
Operation Map-Reduce
Subscription
Item
Term
Map-Reduce Functions
Update/delete Functions
Add Function

**Valeurs du fichier Properties**

Recharger fichier properties

| Propertie | Value | Description |
|---|---|---|
| SERVER_NAME | localhost | Serveur MongoDB : adresse IP, ou nom |
| SERVER_PORT | 27017 | Port du serveur MongoDB |
| DBNAME | pubsub_sys | Nom de la base MongoDB utilisée |
| COLLECTION_SUBSCRIPTION | subscription | Nom de la collection des souscriptions |
| COLLECTION_ITEM | item | Nom de la collection des items |
| COLLECTION_OPERATION | sequence | Nom de la collection contenant la liste et dépendences des opérations MapReduce |
| COLLECTION_FUNCTION | system.js | Nom de la collection contenant les fonctions Map et Reduce (filtrage). |
| OPERATION_ITEMBYSUB | itemBySub_simple | Nom de l'opération permettant d'obtenir le nombre d'items par souscription (historique) |
| useSharding | false | L'utilisation du système est en mode sharding ou pas. Permet de créer les collections en mode sharding ou non) |
| SHARDKEY_DEFAULT | _id | Clé de shard par défaut |
| SHARDKEY_SUBSCRIPTION | subID | Clé de shard pour la collection subscription |
| SHARDKEY_ITEM | itemID | Clé de shard pour la collection item |
| SHARDKEY_OPERATION | seqID | Clé de shard pour la collection operation |
| NB_SUBSCRIPTION | 1000 | Nombre de souscription à utiliser pour le filtrage (matching) |
| NB_ITEM | 100 | Nombre d'item à utiliser pour le filtrage (matching) |
| SKIP_ITEM | 0 | |
| NB_TEST | 1 | Nombre de tests à effectuer pour le maching |

*Illustration 24: Page Web de configuration des paramètres de l'application*

***itemBean* :** classe permettant d'obtenir les informations concernant un item en particulier à partir de son identifiant. On aura accès notamment :

- à la taille de son historique, c'est-à-dire au nombre de souscriptions qui ont été notifiées pour cet item.
- la liste des termes associées à cet item
- la liste des souscriptions correspondant à l'historique. Cette information pouvant être très importante et donc inexploitable nous la donnons sous forme tronquées.

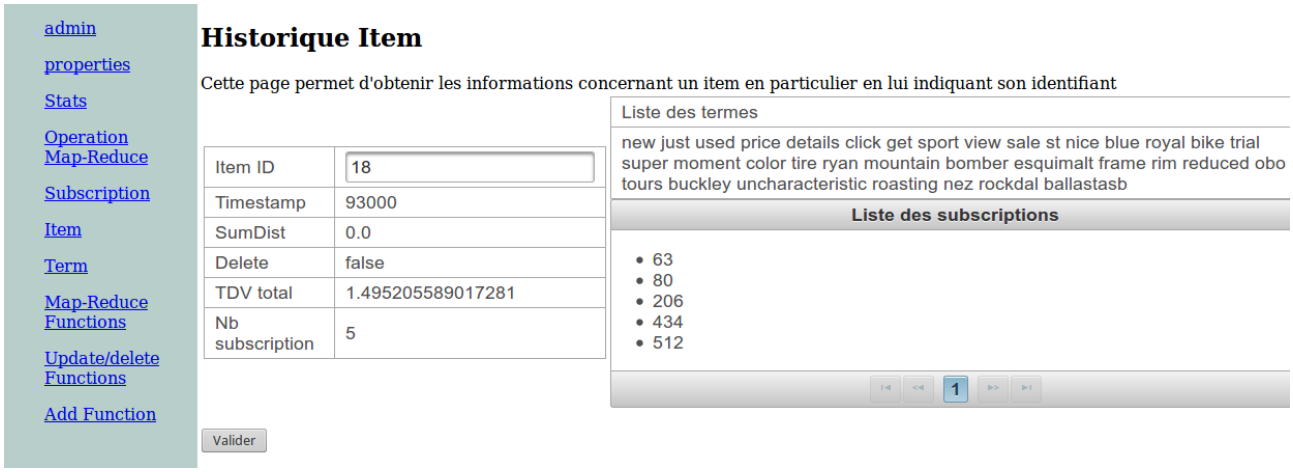


*Illustration 25: Page Web relative aux informations d'un item*

***SubscriptionBean* :** classe permettant d'obtenir les informations concernant un item en particulier à partir de son identifiant. On aura accès notamment :

- à la taille de son historique, c'est-à-dire au nombre de souscriptions qui ont été notifiées pour cet item. Afin d'apporter de la souplesse au fonctionnement, cette valeur s'obtient à partir du résultat d'une opération *MapReduce*, que l'on définira dans la section « *properties* » en indiquant la valeur de OPERATION_ITEMBYSUB.
- la liste des termes associées à cette souscription.
- la liste des souscriptions correspondant à l'historique. Cette information pouvant être très importante et donc inexploitable nous la donnons sous forme tronquées.

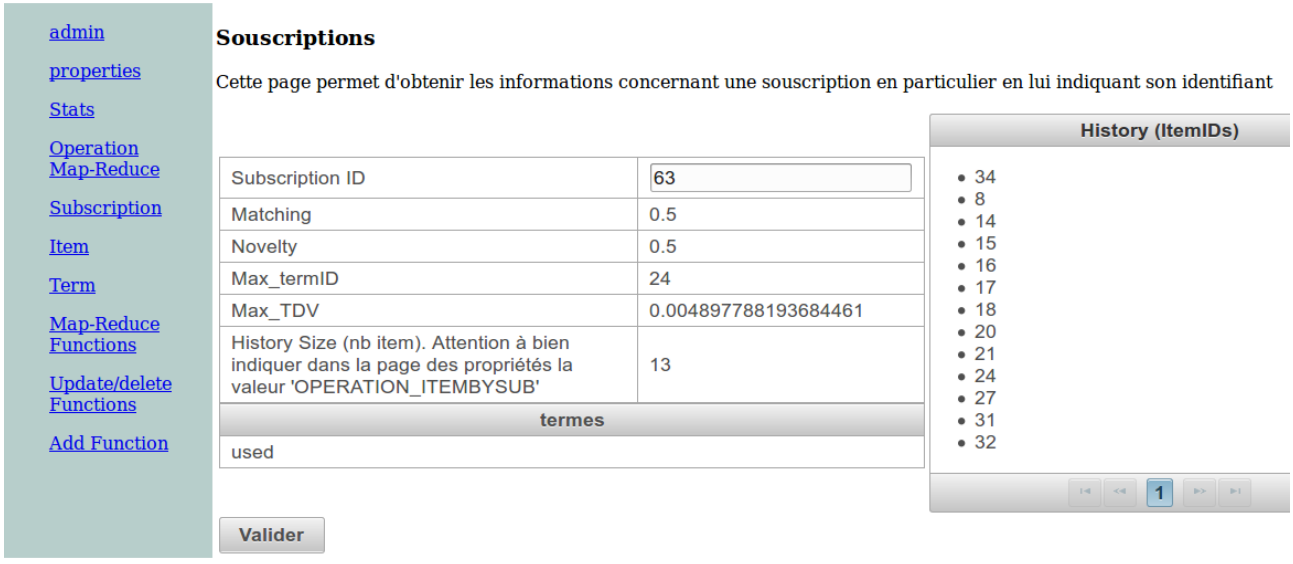


*Illustration 26: Page Web relative aux informations d'une souscription*

***termBean* :** classe permettant d'obtenir les informations concernant un terme en particulier à partir de son nom ou de son identifiant. On aura accès notamment :

- au nombre de souscriptions qui référencent ce terme.
- la liste des souscriptions référençant ce terme. Cette information pouvant être très importante et donc inexploitable nous la donnons sous forme tronquées.

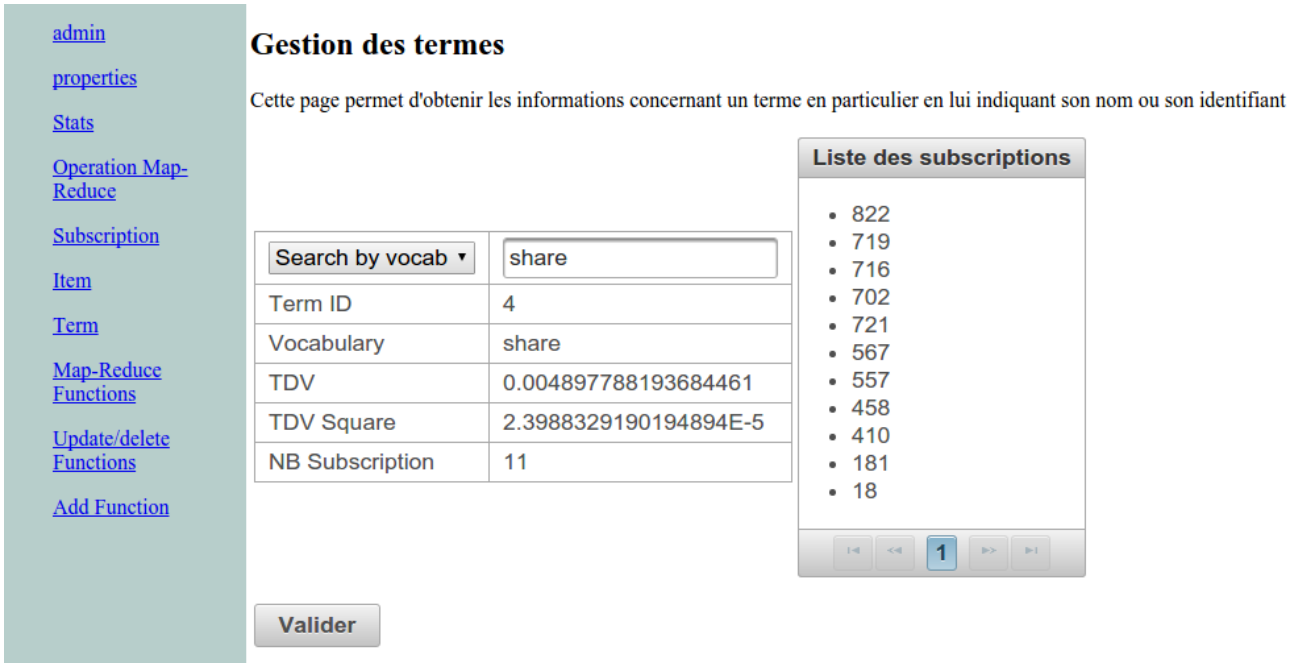


*Illustration 27: Page Web relative aux informations d'un terme*

**Les autres interfaces Web seront présentées plus en détails dans la partie relative au Workflow. Cependant elles sont exposées ci-dessous de manière succincte.**

***statsBean* :** Visualiser le résultat d'une opération (une fonction *Map* et *Reduce*) préalablement ajoutée. Permet aussi de visualiser le détail de l'opération (dépendance avec le résultat d'une autre opération) (illustration 31)

***operationBean* :** Ajout ou suppression d'une nouvelle opération composée d'une fonction *Map* et d'une fonction *Reduce*. Peut s'appliquer sur le résultat (une collection) d'une autre opération, ou peut s'appliquer directement sur une collection racine (Item ou Subscription). Les fonctions *Map* et *Reduce* doivent préalablement avoir été ajoutées à la collection des fonctions (System.js) (illustration 30)

***updateFunction* :** Mise à jour ou suppression d'une fonction *Map* ou *Reduce* présente dans la collection System.js (illustration 29).

***addFunction* :** Permet l'ajout d'une nouvelle fonction dans la collection System.js. Si la fonction existe déjà (même _id), la fonction existante sera remplacée par la nouvelle (illustration 28).

### *Problèmes rencontrés*

#### Choix de l'implémentation de JSF : Apache ou Oracle !

Il existe deux implémentations JSF. Celle d'Oracle nommée Mojara et celle de la fondation Apache nommée MyFaces. Les différences entre les deux implémentations sont à notre stade d'utilisation difficiles à percevoir. Elles interviendraient, selon quelques *benchmark* trouvés sur la « toile », essentiellement au niveau des performances selon les cas d'utilisations. Et pourtant, l'implémentation Mojara ne nous a pas permis de faire fonctionner le *framework* correctement. Les composants JSF déclarés dans le Face-Config n'étaient pas interprétés. L'utilisation de l'implémentation Apache a réglé le problème.

#### Le *framework* impose son fonctionnement

Imposer un mode de fonctionnement peut sembler être une bonne démarche. Il évite notamment de créer du code « exotique » souvent compréhensible uniquement par le développeur lui-même. Cela permet par ailleurs à un autre développeur qui connaît le *framework* de comprendre rapidement le code réalisé. Cependant, cette démarche impose un certain apprentissage et peut poser des soucis lorsque que l'on veut contourner un problème difficile à résoudre. Avec JSF les composants apportent beaucoup de productivités. C'est le cas par exemple lorsque l'on souhaite employer du JavaScript ou de l'Ajax. Le *framework* prévoit des composants pour importer son propre code JavaScript dans les pages Xhtml. A ce titre, il est même préférable d'utiliser une des librairies de composants graphiques fournissant des composants Ajax, comme *PrimeFaces* que nous avons utilisé dans le projet mais n'a pas toujours fonctionné correctement. Là aussi, on rencontre encore le problème de savoir qu'elle est la meilleure option tant il existe de librairies graphiques disponibles. Cependant lorsque le *framework* ne dispose pas du composant qui convient, il faut alors le créer soit même. Pour cela il est nécessaire de bien maîtriser les principes de fonctionnement et d'architecture du *framework.*

## 5.3 Un workflow pour la gestion des fonctions *MapReduce*

### 5.3.1 Présentation

Cette partie est consacrée à la création d'un *workflow* pour la gestion « à chaud » des fonctions *MapReduce*. L'objectif est d'offrir une interface Web permettant à l'utilisateur d'ajouter, de modifier ou de supprimer les fonctions *MapReduce* et de les exécuter. Cette fonctionnalité s'avère très utile dans un environnement de recherche où l'on cherche à effectuer des tests et les analyser. L'intérêt est de pouvoir intervenir sur les fonctions sans avoir à intervenir directement au niveau du code et d'être obligé de le recompiler à chaque changement. Néanmoins, si l'utilisateur qui exécute les fonctions n'a nul besoin de s'y connaître en JavaScript et en *MapReduce* ce n'est pas le cas du concepteur des fonctions.

### 5.3.2 Terminologie

La fonction : il s'agit d'un bloc de code « atomique » écrit en JavaScript. On distinguera essentiellement d'un côté la fonction *Map* et de l'autre la fonction *Reduce*. Il est possible d'avoir des fonctions « intermédiaires » qui peuvent-être appelées par les fonctions *Map* et *Reduce*. Elles ne sont généralement là que pour alléger le code d'un *Map* ou d'un *Reduce*. Ces fonctions comportent un identifiant pour être appelé et sont enregistrées dans la collection Systeme.js

L'opération : il s'agit de l'exécution des fonctions *Map* et *Reduce* prenant en entrée une collection et fournissant en sortie un résultat sous la forme d'une liste de couple « clé-valeur ». Dans notre cas nous stockons ce résultat dans une collection.

La séquence d'opérations : il s'agit d'imbriquer avec le résultat d'une opération dans une autre. Dans notre cas la séquence est conceptuelle, puisque dans les faits une opération prend toujours en entrée une collection, qu'elle provienne de l'exécution d'une opération ou non (ex : la collection Item). Cependant une opération peut déclencher l'exécution d'une autre fonction, et cela de manière récursive, afin d'être sûr d'obtenir une collection actualisée.

Deux point de vues d'utilisation:

- **le concepteur** des fonctions *Map* et *Reduce*, des opérations et de leur imbrication.
- **l'utilisateur** des opérations *MapReduce* qui ne fait qu'exécuter les opérations et analyser les résultats.

### 5.3.3 Fonctionnement et utilisation du *workflow*

Nous présentons dans cette section uniquement le fonctionnement du w*orkflow* du point de vue du « concepteur » et de « l'utilisateur », soit les interfaces Web mises à disposition. Leur « mécanique » interne est présentée dans la section suivante.

**Etapes préalables pour parvenir à exécuter une opération *MapReduce* :**

1. Ajouter au minimum une fonction *Map* et une fonction *Reduce* (illustration 28). Il est possible d'ajouter directement les fonctions dans la collection System.js via une console Linux (Terminal) se connectant au SGBD MongoDB.
2. Création de l'opération *MapReduce* (ex : Répartition_historique) comportant une fonction *Map* et une fonction *Reduce*, puis une collection sur laquelle sera appliquée l'opération (illustration 30)
3. Enfin, on peut exécuter l'opération que l'on vient de créer, par exemple Répartition_historique (illustration 31).

**Interfaces Web du point de vue « concepteur »**

Ces interfaces s'adressent aux personnes maîtrisant d'une part la programmation de fonctions *MapReduce* pour MongoDB et d'autres part l'architecture de la base de données et des collections du système Pub/Sub. Une erreur de conception ne renverra « rien du tout » au mieux mais retournera des informations erronées au pire.

Ajout d'une nouvelle fonction : l'ajout se présente tel qu'illustré ci-dessous (illustration 28). Le nom de la fonction doit être unique. L'ajout d'un nom déjà existant remplacera la fonction portant ce nom. La fonction est rédigée selon le formalisme imposé par MongoDB. Cette fonctionnalité bien qu'elle puisse être intéressante, nécessite une grande attention de la part du concepteur. Le problème vient surtout du formatage de la fonction et des erreurs possibles. En effet aucune erreur n'est remontée actuellement. C'est une difficulté rencontrée qui n'a pu être réglée. MongoDB est très peu bavard quant à l'ajout d'objets dans une collection surtout s'il correspond au modèle attendu. Il faut donc considérer cette fonctionnalité comme un dépannage. Le plus sûr reste l'ajout des fonctions directement dans la collection System.js via

la console MongoDB.


*Illustration 28: page Web associée au "managed-bean" "addBean"*

Mise à jour et suppression de fonctions : sur l'image ci-dessous (illustration 29) une liste déroulante permet de sélectionner la fonction soit à modifier soit à supprimer. Le bouton Show permet d'afficher le contenu de la fonction. Encore une fois la fonction Update doit être utilisée avec prudence en faisant très attention au format de la fonction pour les mêmes raisons que pour la fonctionnalité d'ajout de fonctions ci-dessus.

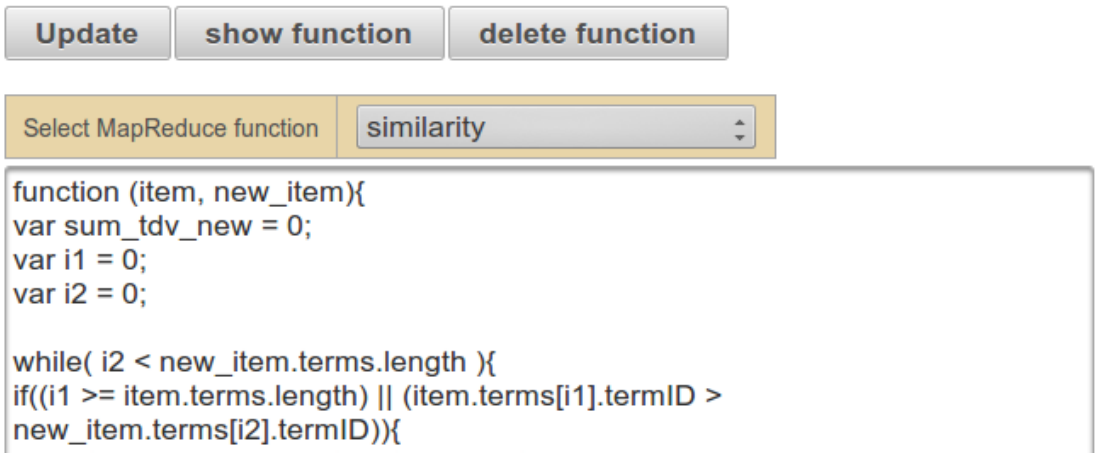


*Illustration 29: page Web associée au "managed-bean" "updateBean"*

Création des opérations ou séquences : une fois les fonctions écrites et enregistrées il est possible de créer les opérations. À savoir qu'une fonction peut-être réutilisée de différentes manières selon le résultat attendu. Le concepteur doit connaître le rôle de chaque fonction et ce qu'elle retourne pour pouvoir l'utiliser dans une opération. Outre le nom de l'opération qui doit être unique, il faut indiquer la fonction *Map* et *Reduce* pouvant fonctionner ensemble, puis l'opération dont elle dépend. La dépendance est en réalité une collection résultant de l'exécution de l'opération du même nom. Dans l'illustration 30 la dépendance « itemBySub_simple », qui est une opération, a générée une collection

« *Map*_itemBySub_simple ». Dans le cas où une opération n'a pas de dépendance (champ « null »), la collection sur laquelle est appliquée l'opération est Item, que l'on considère comme étant la racine.

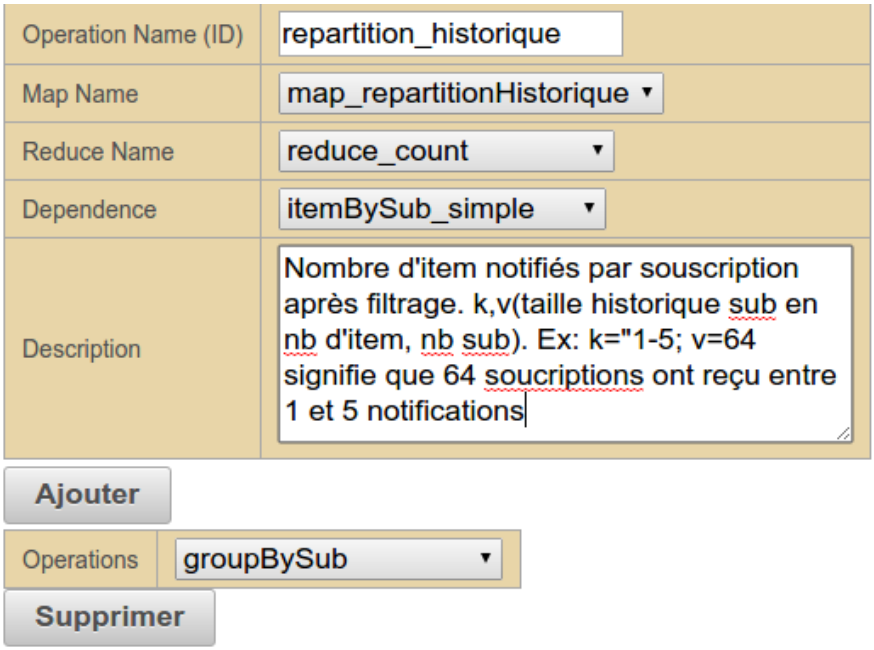


*Illustration 30: Ajout et suppression d'une opération MapReduce*

**Interface Web du point de vue de l'utilisateur**

L'utilisateur du *workflow* dispose d'une seule interface lui permettant d'exécuter une opération (qui peut-être une séquence d'opérations) sans avoir à se soucier de leur dépendance ou la manière dont les fonctions sont programmées. À la différence du concepteur, l'utilisateur n'a pas vraiment besoin de maîtriser l'architecture de la base de données du système Pub/Sub. L'écran est divisé en deux parties. Une partie est générique pour l'affichage des résultats montrant une liste de clé-valeur, tel est le type de retour d'une opération *MapReduce*. Une autre partie donne le détail de l'opération en indiquant notamment la collection d'entrée, le nom des fonctions *Map* et *Reduce* qu'elle utilise ou encore sa dépendance.


*Illustration 31: page Web associée au manged-bean "statsBean"*

### 5.3.4 Conception interne du *workflow*

La conception intervient à cinq « niveaux » :

- la description des opérations et leur dépendance sont enregistrées dans une collection Opération (illustration 32)
- l'organisation et le stockage des fonctions *MapReduce* dans la collection System.js (illustration 32)
- les algorithmes de gestion et d'exécution des fonctions et des opérations (les classes Java).
- l'interface utilisateur , reposant sur le *framework* JSF exploitant les *manged-bean* et les pages *X*HTML

**Représentation des opérations dans la collection Operation**

L'illustration 32 montre à droite un extrait simplifié de la collection « opération » et son lien avec celle stockant les fonctions *MapReduce,* décrite au paragraphe suivant. Elle montre la représentation de deux opérations dépendantes. On remarque que la deuxième opération `"repartitionHistorique",` a besoin de la première, `"itemBySub_simple"` , pour fonctionner. La collection en entrée d'une opération est soit la collection « racine » Item, soit le résultat d'une opération. Lorsqu'une opération n'a aucune dépendance la collection d'entrée est Item. Enfin, la cardinalité de « 2 » indique qu'une opération ne peut prendre que deux fonctions en entrée, celle de *Map* et celle de *Reduce*. En revanche une fonction *Map* ou *Reduce* peut-être utilisée dans une multitude d'opérations.

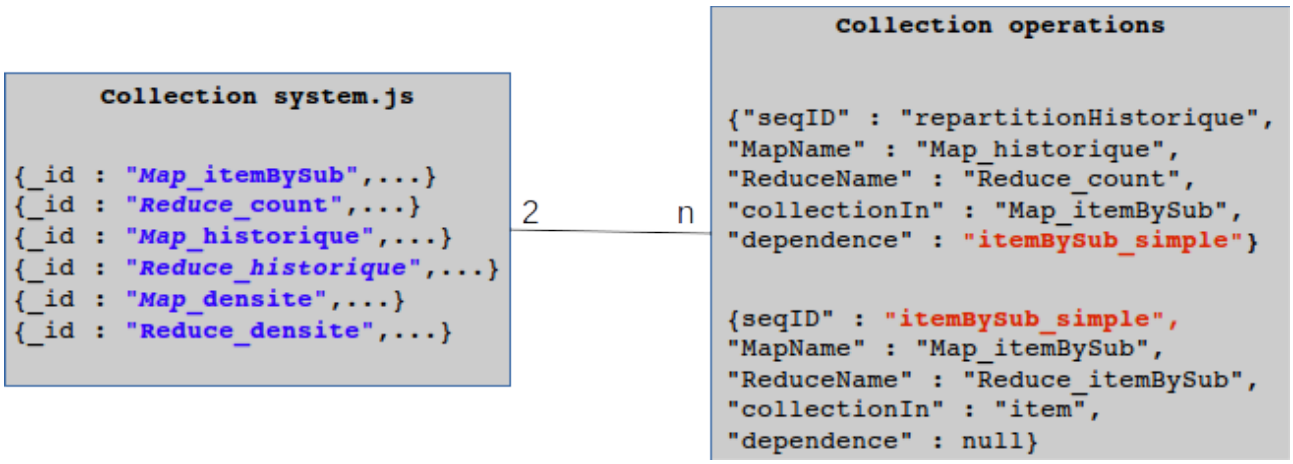


*Illustration 32: Représentation des collections de stockage des fonctions MapReduce et des opérations*

### Organisation des fonctions *MapReduce* et stockage dans la collection System.js

Ce qui rend possible l'ajout de fonction *MapReduce* pendant le fonctionnement du programme (« à chaud ») c'est le fait de n'avoir à l'écrire que dans une collection (illustration 32) et donc ne pas avoir à recompiler le code pour la faire fonctionner. Une fonction est stockée et gérée dans la collection System.js propre à MongoDB prend deux valeurs :

- un identifiant de fonction auquel on fait référence lorsque l'on veut l'exécuter
- une valeur qui contient le corps (l'algorithme) de la fonction et prenant en entrée généralement des paramètres.

Prenons l'exemple des fonctions *Map* et *Reduce itemBySub* qui permet d'obtenir le nombre de notifications (d'items) pour chaque souscription. On remarque bien l'identifiant et la valeur de chaque fonction. La première ligne, utile au SGBD, indique seulement que la fonction est sauvegardée dans la collection System.js.

```
db.System.js.save({
      _id : "Map_itemBySub",
      value:
            function(item){
                  for(sub in item.Subscriptions){
                        emit(item.Subscriptions[sub].subID, 1);
                  }
      }
});
```

```
db.System.js.save({
      _id : "Reduce_itemBySub",
      value:
            function(subID, items){
                  var count = 0;
                  for(i in items){
                        count += items[i];
                  }
                  return count;
            }
});
```

Dans les écrans de création (illustration 28) et de mise à jour des fonctions (illustration 29), que le champ « nom de la fonction » correspond à l'ID et le champ permettant l'écriture correspond à la valeur de la fonction.

La plus grande complexité des algorithmes se situe plutôt au niveau de l'exécution des opérations *MapReduce* qu'au niveau de leur création.

### Algorithmes de gestion et d'exécution des fonctions et des opérations

Le diagramme de classe ci-dessous montre l'ensemble des classes intervenants dans la gestion du *workflow*. Cette vue d'ensemble est détaillée dans la suite du document en fonction des taches réalisées.

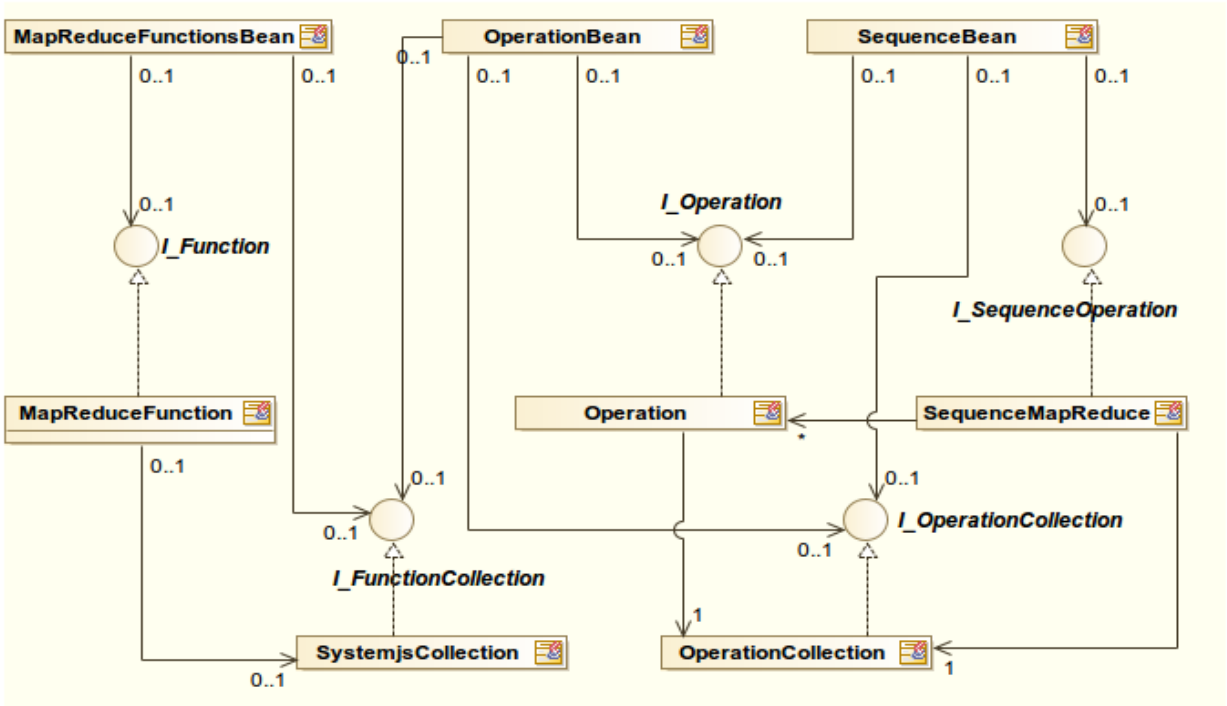


*Illustration 33: Diagramme de classes impliquées dans la gestion du workflow*

Le diagramme de classes ci-dessous (illustration 34) correspond à la gestion des fonctions *Map* et *Reduce* (illustration 28 et 29). Il montre notamment les méthodes permettant l'ajout, la mise et jour et la suppression des fonctions. C'est uniquement la classe *MapReduce*Fonction qui gère cela tout en s'appuyant sur la classe SystemJSCollection représentant de la collection System.js.

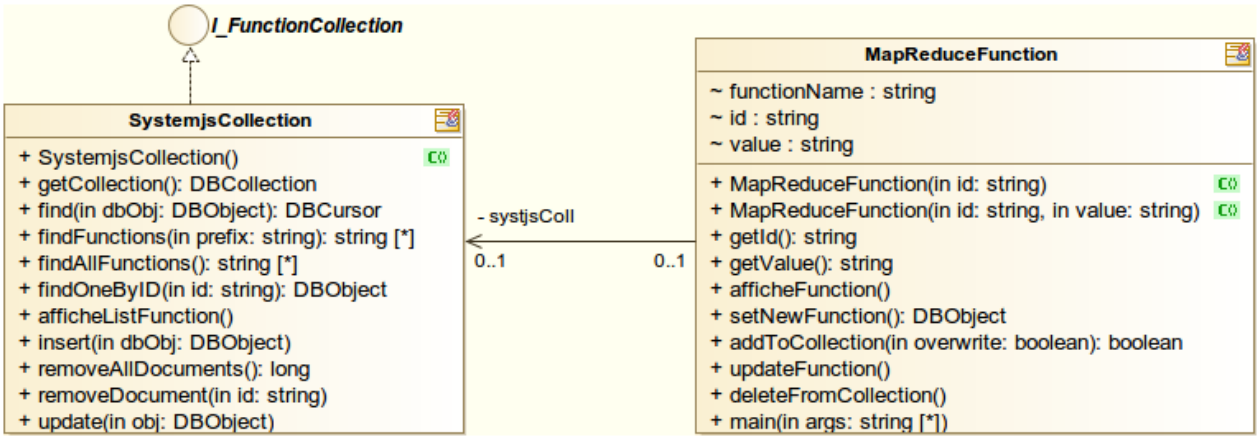


*Illustration 34: Diagramme de classes impliquées dans la gestion des fonctions MapReduce*

Le diagramme de classes ci-dessous (illustration 35) correspond à la gestion des opérations *MapReduce*. Cette gestion implique d'avoir préalablement créées les fonctions *Map* et *Reduce*. La classe Operation gère cela tout en s'appuyant sur la classe OperationCollection, via son interface, qui est la classe Dao représentant de la collection Operation.

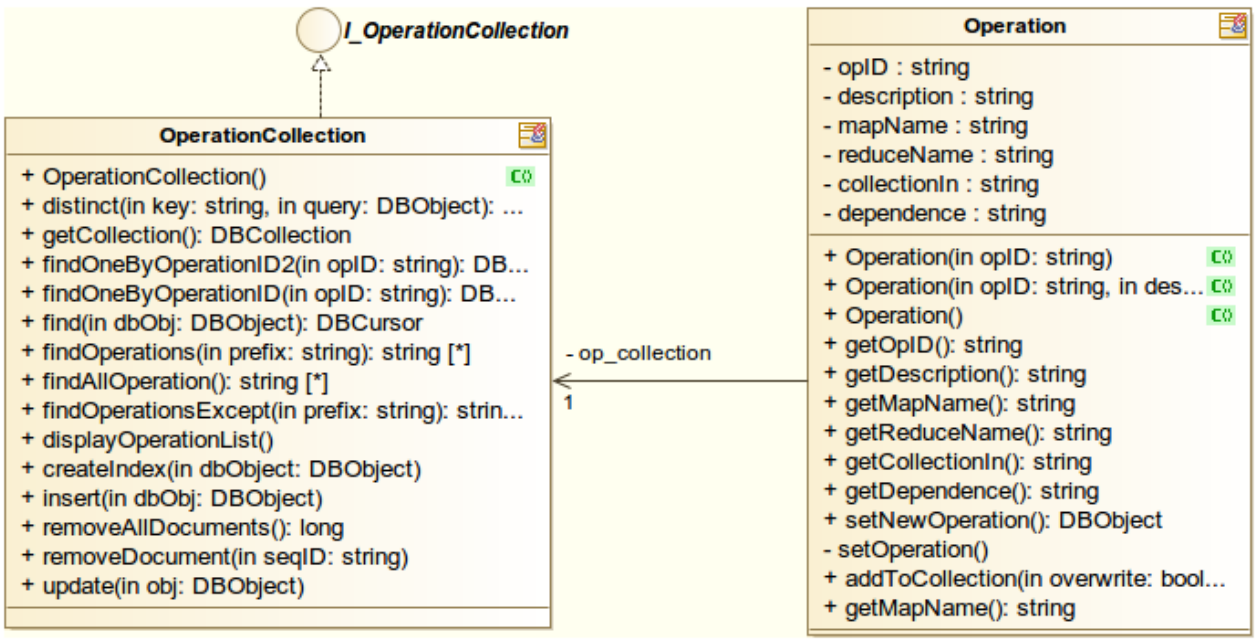


*Illustration 35: Diagramme de classes impliquées dans la gestion des opérations MapReduce*

Une fois la création des fonctions et des opérations effectuées il est possible d'exécuter ces dernières. Le diagramme de classe ci-dessous (illustration 36) correspond à l'exécution des opérations *MapReduce* (ou d'une séquence d'opérations). La classe SequenceOperation qui gère cela tout en s'appuyant sur les classes vues dans les diagrammes précédents, soit les classes Operation et OperationCollection.

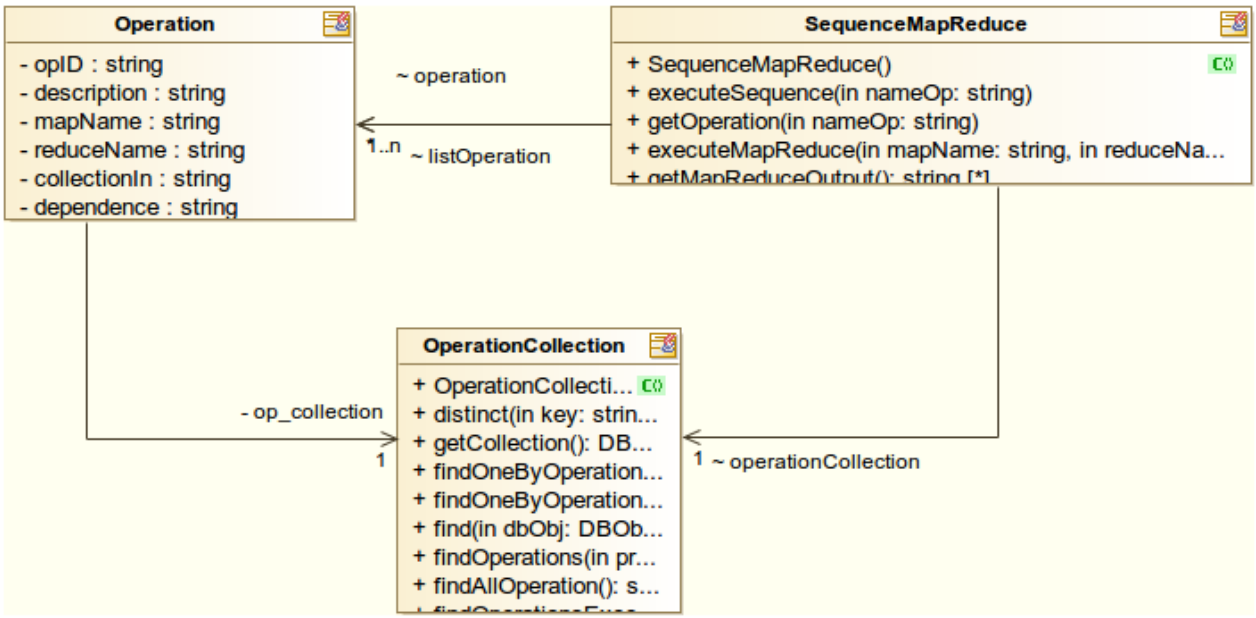


*Illustration 36: Diagramme de classes impliquées dans l'exécution d'une opération (séquence)*

Code associé à la classe SequenceMapReduce

Le code ci-dessous permettra de comprendre le fonctionnement de l'exécution des opérations. Les trois méthodes ci-dessous sont ensuite formalisées dans un diagramme d'activités pour une vision différente.

```
/** Excecuter une sequence d'operations MapReduce - doit récuperer au
préalable la liste des operations */
	public void executeSequence(String nameOp){
		getOperation(nameOp);
		Operation o;
		while(!listOperation.empty()){
			//on dépile la liste des séquences parents
			o = listOperation.pop();
			// on execute chaque MapReduce de la pile (parent)
			executeMapReduce(o.getMapName(), o.getReduceName(),
				o.getCollectionIn());
		}}
	/** Récupérer la liste des opérations (dans la collection operation)
*/
	public void getOperation(String nameOp){
```

```
            operation = new Operation(nameOp);

            if(operation.getDependence() != null){
                  listOperation.push(operation);//on alimente la pile
                        d'opération (sequences)
                  getOperation(operation.getDependence());//on va récupérer
la                séquence suivante
            }else{//si il n'y a plus de séquence parent
                  listOperation.push(operation);//on empile la dernière
séquence
            }
      }

      /** Excecuter un MapReduce particulier */
      public void  executeMapReduce(String MapName, String ReduceName,
String                                          collectionIn){
            String Map = "function(){"+MapName+"(this);}";
            String Reduce = "function(key,values){return
      "+ReduceName+"(key,values);}";
            MapCollection =
      DBConnector.getInstance().getCollection(collectionIn);
            MapReduceCommand cmd = new
      MapReduceCommand(MapCollection,Map,Reduce,
                        MapName,MapReduceCommand.OutputType.REPLACE,null);
            out = MapCollection.MapReduce(cmd);
      }
```

L'illustration 37 détail sous forme de diagramme d'activité le fonctionnement de l'exécution d'une opération (ou séquence). Sachant qu'une opération peut dépendre du résultat d'une autre, résumons le processus d'exécution d'une opération « op »:

1. On vérifie si l'opération « op» dépend d'une autre
2. Si oui, on empile l'opération parent « op*i* » dans une pile.
3. On retourne en « 1 » jusqu'à ce que l'opération « op*i* » n'ai plus de parent.
4. (nous disposons d'une pile d'opérations que l'on va dépiler). La première opération de la pile n'a donc pas de parents. On peut l'exécuter et ajouter le résultat à une collection portant le nom de l'opération qui sera utilisée, en autre, pour l'opération suivante.
5. Tant que la pile d'opérations n'est pas vide on retourne en « 4 » pour exécuter l'opération suivante. La dernière opération de la pile exécutée est l'opération que l'on souhaitait exécuter initialement.

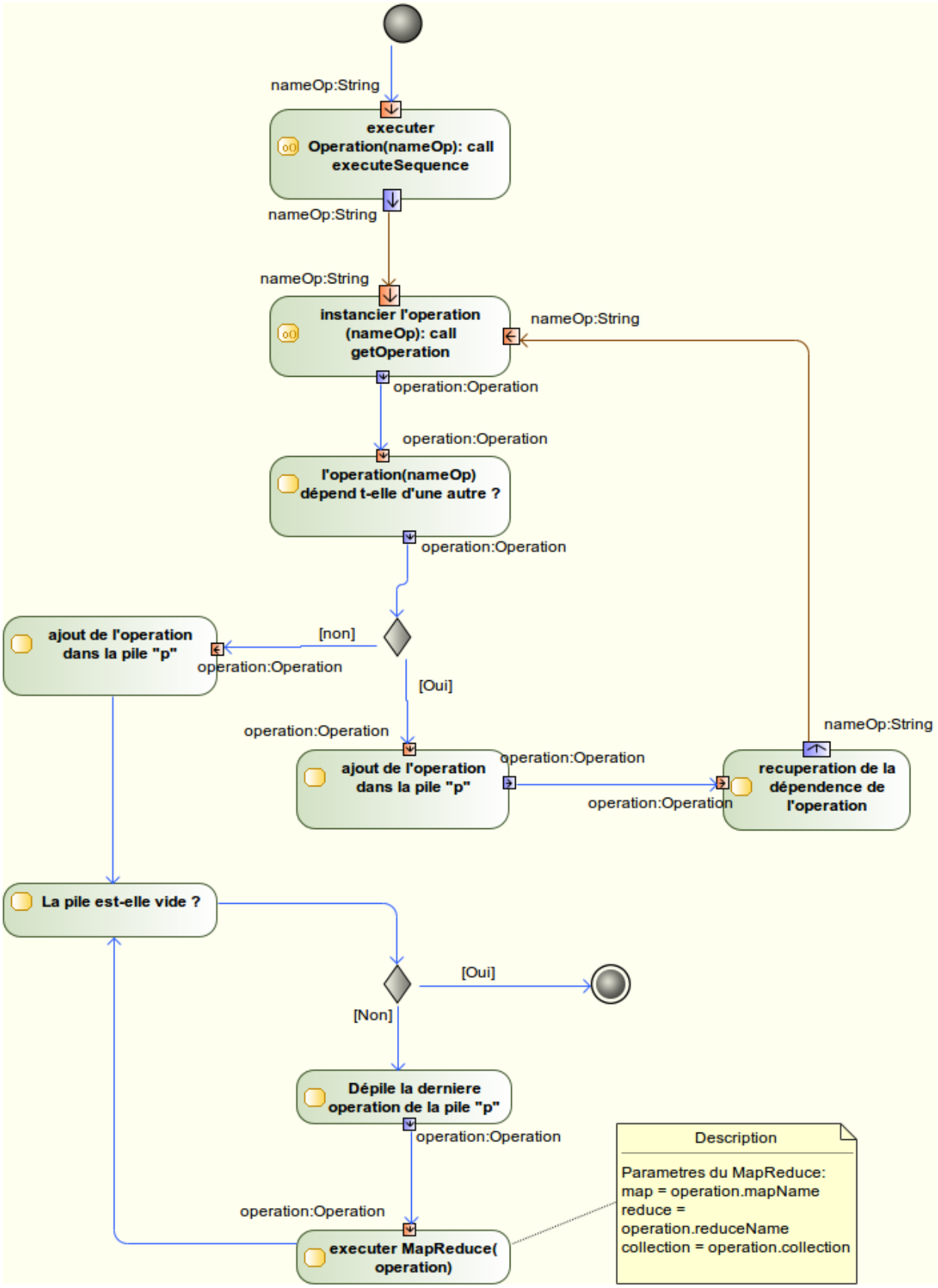


*Illustration 37: Diagramme d'activité de l'exécution des opérations MapReduce*

# 6. Conclusions et Perspectives

Pour mener à bien ce projet de mise en œuvre du logiciel de publication/souscription il a fallu passer par plusieurs phases. Avant de tirer le bilan des différentes actions effectuées et de donner un aperçu de la répartition du temps de travail effectué, il semble important de rappeler brièvement les objectifs fixés. Ceci nous permettra alors de formuler des perspectives d'évolution pour le système Pub/Sub.

Nous pouvons résumer l'ensemble des objectifs que nous nous étions fixés autour de deux dimensions :

- Restructuration du logiciel pour le rendre plus évolutif et maintenable.
- Offrir de nouvelles possibilités au logiciel comme l'intégration d'une interface web ou encore la gestion ergonomique des fonctions *MapReduce*.

Pour atteindre ces objectifs assez généraux il a tout d'abord été nécessaire de bien cerner le contexte et la problématique du système *FiND*. Même si cette étape a été indispensable au démarrage de la nouvelle phase du logiciel, elle s'est poursuivie tout au long du processus de développement du projet. C'est notamment au cours de cette étape que nous avons pu clarifier les besoins.

L'étape suivante a été celle de la rétro-conception. Dans le cas, qui était le nôtre, d'une version déjà existante d'un logiciel, celle-ci consiste essentiellement à comprendre les développements qui ont été réalisés auparavant. Cette étape qui demande généralement du temps, ne produit pour ainsi dire aucune évolution visible et exploitable au système existant, ce qui peut sembler frustrant. Elle a permis cependant de fournir un certain nombre de documents utiles pour la suite du projet, comme la production des diagrammes ou encore la documentation du code. Au cours de cette étape nous avons pu formuler des hypothèses d'évolutions de manière plus précises.

Cette étape a aussi permis de tirer quelques leçons concernant la gestion d'un projet de développement. En l'occurrence, il me parait vraiment judicieux, voir indispensable, de mettre en place, dès le démarrage d'un projet, un système de documentation autour de trois niveaux afin de limiter le temps de reprise d'un projet existant par de nouveaux intervenants.

1. Le premier niveau concerne la documentation du code lui-même. Il permet d'expliciter les différents choix de programmation.
2. Le deuxième niveau concerne la description de la logique d'architecture du logiciel. Celui-ci permet, sans avoir à rentrer dans le détail du code de comprendre les choix de conceptions. On s'appuie dans ce cas sur les différents diagrammes disponibles (classes, séquences, activités, paquetages...) et les explications textuelles des choix de conceptions. On y décrira par exemple le rôle d'un paquetage ou les classes mises en œuvre pour un processus donné (et pouvant être difficile à comprendre).
3. Enfin le troisième niveau concerne plus particulièrement le fonctionnement et l'utilisation du système. On peut notamment y retrouver une procédure d'installation ou de reprise après panne du logiciel ainsi que les différents paramètres à utiliser. Cette dernière est particulièrement importante pour pouvoir rapidement entrer dans le cœur du logiciel.

Enfin l'étape de mise en œuvre du logiciel Pub/Sub à proprement parler a consisté d'une part à l'implémentation de nouvelle technologies et d'autre part au développement de nouvelles fonctionnalités. Comme on peut le remarquer au travers de ce mémoire un nombre non négligeable de technologies intervient dans ce projet. Certaines concernent plus particulièrement le stockage et l'accès aux données. Certaines concernent directement le fonctionnement du logiciel en lui-même. D'autres concernent l'usine logicielle, c'est-à-dire les technologies utilitaires n'intervenant pas directement dans le fonctionnement du logiciel. Assimiler le principe de fonctionnement de toutes ces technologies a nécessité beaucoup de temps, difficile à gérer dans le délai de huit mois imparti pour la réalisation de ce projet. Il a donc fallu aller à l'essentiel et approfondir uniquement en cas de besoins. C'est probablement une des questions les plus difficiles que j'ai eu à résoudre au cours de cette expérience. Lorsque que l'on rencontre un problème, doit-on le résoudre à tout prix, au risque de dépenser une grande partie de son temps, peut-on le reporter ultérieurement ou tout simplement le contourner. Il y a effectivement des problèmes que j'ai dû laisser en suspend, ne remettant pas spécialement en question l'avancé du projet.

**Répartition du temps de travail et métriques de qualité du logiciel**

Afin d'avoir une visibilité sur la production du projet, nous présentons d'une part une vue synthétique de la répartition du temps passé sur le projet et d'autre part quelques métriques

utilisés dans le contrôle de qualité des logiciels.

Le diagramme de Gantt ci-dessous montre une vue approximative de la durée de chaque grande tâche. On remarque que la plupart des taches se chevauchent avec d'autres. En effet certaine tache ont pu être démarrée parallèlement à d'autre, sachant, qu'étant seul à travailler sur le projet, quand je m'occupais de l'une j'interrompais l'autre. Par exemple l'outil Git a été mis en place quelque temps après le démarrage de l'architecture, ce qui explique pourquoi la mise en place de l'usine logicielle s'étale globalement sur un mois. Par ailleurs la tache de production des interfaces Web est elle aussi très étalée. En réalité, la réalisation de ces interfaces était liée à la mise en place de l'architecture dans un premier temps et dans un second temps elle était liée à la création du *workflow*.

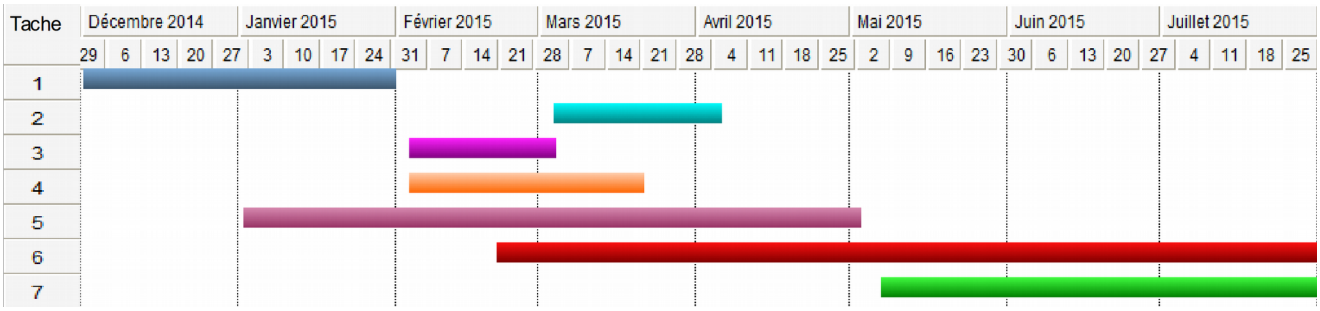

*Illustration 38: Diagramme de Gantt - répartition du temps de travail*

Taches :

1. Phase de rétro-conception
2. Mise en œuvre de MongoDB et *MapReduce*
3. Mise en œuvre du *framework* JSF (+compréhension)
4. Mise en œuvre de l'usine logicielle
5. Production de la nouvelle architecture
6. Production des interfaces Web (utilisation de JSF)
7. Création du *workflow* de gestion des fonctions *MapReduce*

Dans le domaine de la qualité logicielle plusieurs indicateurs permettant de la mesurer sont à notre disposition. Pour obtenir ces mesures le logiciel SonarQube[21] à été employé. Dans notre cas nous n'avons pas beaucoup exploité ces mesures. Cependant il semble intéressant d'en

21 SonarQube est un logiciel *OpenSource*, écrit en Java, de gestion de la qualité du logiciel. Il couvre sept domaines de qualité : l'architecture et la conception, le niveau de commentaire du code, la duplication de code, les tests unitaires, le niveau de complexité, détection du risque de *bug* et respect des règles de codage. (sonarqube.org)

livrer quelques-unes permettant notamment d'avoir une visibilité sur l'évolution et l'ampleur du logiciel.

- Le logiciel comprend actuellement 4949 lignes de code dont la répartition est la suivante :
  - 4084 lignes de code Java (973 lignes dans sa phase 2, la précédente version)
  - 865 ligne de code HTML (aucune dans la phase 2)
  - 59 classes (15 classes dans la phase 2)
- La mesure de la complexité[22] du code semble être un point intéressant à livrer puisque notre objectif était de simplifier le code et d'en améliorer l'évolutivité, sachant que plus le code est complexe plus il est difficile à maintenir et à faire évoluer. Selon les mesures du logiciel, nous pouvons noter l'évolution de la complexité pour les méthodes et les classes du programme Pub/Sub :

| | **Phase 2** | **Phase 3** |
|---|---|---|
| **par méthode** | 2,8 | 1,3 |
| **par classe** | 14,3 | 11,9 |

**Perspectives**

Au cours des travaux réalisés nous nous sommes essentiellement intéressé à apporter une meilleure structure au projet de manière à pouvoir le faire évoluer plus facilement. Les perspectives qui vont maintenant être présentées s'intéressent plus particulièrement à la mise en production du logiciel Pub/Sub. Nous pourrions proposer pour cela trois point d'amélioration :

**1. Evolution de l'infrastructure pour un passage à l'échelle**

Le logiciel tel qu'il est réalisé au moment de la rédaction de ce mémoire, n'a été utilisé que dans le cadre de tests par quelques personnes avec des jeux de données pré-formatés. Cela

22 La complexité « cyclomatique » mesure la complexité du code en comptabilisant le nombre de chemins possibles au sein du code. C'est l'utilisation du nombre de boucles ou d'instructions conditionnels qui augmente cette complexité. Plus le nombre est élevé plus l'algorithme est complexe.

signifie que l'on peut déterminer à l'avance le taux d'utilisation de l'infrastructure et par conséquent la charge des différents serveurs. L'infrastructure à deux niveaux était donc suffisante pour supporter les différents tests, soit:

1. un serveur comportant une « JVM » + un conteneur de Servlet + un serveur Web
2. un serveur (ou cluster) de bases de données MongoDB

En revanche si l'on souhaite que le logiciel Pub/Sub puisse fonctionner en production à l'échelle du Web l'infrastructure actuelle risque d'être confronté à des limitations, particulièrement au niveau de l'environnement Java. L'infrastructure distribuée au niveau des bases de données MongoDB est déjà intégrée et prévue nativement pour supporter l'extensibilité (*scalability)* via son mode *Sharding*. Cependant l'environnement Java (le serveur) doit gérer plusieurs taches qui pourraient être séparées afin de limiter les éventuelles saturations des capacités des machines ou réseaux. On pourrait imaginer pouvoir séparer notamment la partie web de la partie traitement des items et des souscriptions, comprenant le filtrage, qui sont véritablement deux taches indépendantes et pouvant fonctionner en parallèle. À l'intérieur de cet ensemble de taches (processus de gestion des items, des souscriptions et du vocabulaire ainsi que celui du filtrage), nous pourrions encore envisager une autre séparation. D'un côté, la gestion des processus de gestion des items, des souscriptions et du vocabulaire qui consiste à récupérer ces données à partir d'une source (par exemple à partir des flux RSS pour les items) puis les formater afin qu'elles puissent être prises en charge par le processus de filtrage. De l'autre, celui du filtrage s'appuyant sur les données formatées. Cependant pour envisager pouvoir déplacer des processus sur des environnements séparés il faudrait modifier l'architecture afin que les processus puissent communiquer entre eux. On pourrait notamment envisager transformer chaque processus en composants indépendants exécutables et uniquement accessible via des interfaces. Ils seraient alors déployables sur des environnements différents. L'utilisation de méthodes de communications entre composants telles que les composants JEE RMI[23] (*Remote Method Invocation*) ou l'emploi de Web Services[24], plus standards seraient à priori des solutions crédibles.

23 RMI : Remote Method Invocation est une interface de programmation (API) pour le langage Java qui permet d'appeler des méthodes distantes. Cette bibliothèque qui se trouve en standard dans JSE, est une technologie qui permet la communication via le protocole HTTP entre des objets Java éloignés physiquement les uns des autres, autrement dit s'exécutant sur des machines virtuelles java distinctes. RMI facilite le développement des applications distribuées en masquant au développeur la communication client / serveur.

24 Web Services : Un service Web est un programme informatique de la famille des technologies Web permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués.

### 2. Améliorer l'ergonomie de l'interface Web

Cette évolution n'est pas vraiment une nouveauté mais plutôt la continuité du travail engagé sur l'interface Web. Si l'on souhaite mettre en production cette application Web, l'ergonomie devient alors importante. Il est souhaitable de pouvoir offrir à l'utilisateur un confort d'emploi au risque de le perdre et peut-être même finalement perdre l'intérêt du logiciel. La bonne prise en charge de la gestion des erreurs (de saisies et de traitement), est notamment un point important pour envisager une exploitation sans (trop) craindre le « plantage » de l'application. Intégrer par exemple de l'AJAX, à l'heure où cette technologie est mature et entrée dans nos habitudes d'utilisation (même si on ne le sait pas), apporterait un grand confort d'utilisation.

### 3. Amélioration de la productivité du logiciel

L'intégration continue est un concept qui consiste essentiellement à vérifier la qualité du code, le valider et le déployer sur un serveur. Ce concept repose généralement sur une brique logicielle permettant l'automatisation de tâches comme la compilation du code, l'exécution des tests unitaires pour éviter la régression du code, des tests de performances, etc. Concernant le projet Pub/Sub il faudrait envisager un environnement dédié à l'intégration et accessible à l'ensemble des développeurs. Celui-ci s'appuierait sur les dépôts de code, en l'occurrence dans notre cas les dépôts GIT, qui doivent être partagés par l'ensemble des développeurs. Systématiser la création de tests unitaires et fonctionnels sans quoi l'intégration continue perd tout son intérêt. Une solution comme Jenkins[25] seraient donc une solution crédible pour améliorer la productivité du logiciel.

25 Jenkins : outil open source d'intégration continue écrit en Java et fonctionnant dans un conteneur de servlets tel qu'Apache Tomcat, ou en mode autonome avec son propre serveur Web embarqué. Il s'interface avec des systèmes de gestion de versions tels que CVS, Git et Subversion, et exécute des projets basés sur Apache Ant et Apache Maven aussi bien que des scripts arbitraires en shell Unix ou batch Windows.

# Bibliographie


### Ouvrages

BAILET Thomas. Architecture logicielle – Pour une approche organisationnelle, fonctionnelle et technique. France : ENI, 2012, 365 p.

BANKER Kyle. MongoDB in Action. USA : Manning publication, 2012, 311 p.

CHODOROW Kristina, DIROLF Michael. MongoDB, The Definitive Guide. USA : O'Reilly, 2010, 193 p.

DASHORST Martijn, HILLENIUS Eelco. Wicket in Action. USA : Manning publication, 2009, 392 p.

LAFOSSE Jérôme. Développement n-tiers avec Java EE (Architectures, GlassFish, JSF, JPA, JWS, EJB, JMS, SOAP, REST). ENI, 2011, 901 p.

### Ouvrages en ligne

BERANGER Florent. Big Data – Analyse et valorisation de masses de données. Smile. 2014, 52 p. [en ligne]. Disponible sur : http://www.smile.fr/Livres-blancs/Erp-et-decisionnel/*Big-Data* (consulté le 24/06/2015)

CHACON Scott, STRAUB Ben. Pro Git. 2e ed. Apress, 2014, 574 p. [en ligne]. Disponible sur : http://git-scm.com/book/en/v2 (consulté le 24/06/2015)

FOUCRET Aurelien. *NoSQL* – Une nouvelle approche du stockage et de la manipulation des données. Smile. 55 p. [en ligne]. Disponible sur : http://www.smile.fr/Livres-blancs/Culture-du-Web/*NoSQL* (consulté le 24/06/2015).

Sonatype, Inc. Maven : The definitive Guide FR – Traduction française. ed. Révisée. 2009, 430 p. [en ligne]. Disponible sur : http://maven-guide-fr.erwan-alliaume.com/ (consulté le 24/06/2015)

TAHÉ Serge. Introduction aux frameworks JSF2, Primefaces et Primefaces mobile. ed. Révisée. Developpez.com, 2012, 424 p. [en ligne]. Disponible sur : http://tahe.developpez.com/Java/primefaces (consulté le 18/05/2015)

TAHÉ Serge. Les bases du développement Web MVC en Java, par l'exemple. ed. Révisée. Developpez.com, 2006, 264 p. [en ligne]. Disponible sur : http://tahe.developpez.com/Java/basesWebmvc (consulté le 18/05/2015)

TAHÉ Serge. Introduction à la programmation Web en Java – Servlets et pages JSP. ed. Révisée. Developpez.com, 2002, 215 p. [en ligne]. Disponible sur : http://tahe.developpez.com/Java/web (consulté le 18/05/2015)

**Travaux universitaires et publications**

DEAN Jeffrey, GHEMAWAT Sanjay. *MapReduce* : Simplified Data Processing on Large Cluster. Publication parue dans « OSDI: Operating System Design and Implementation », San Francisco USA, 2004, pages 137-150.

HAN Miyoung. An Intelligent Publish/Subscribe System at Web Scale. Master Informatique, Paris : UPMC, 2014, 36 p.

HMEDEH Zeinab. Indexation pour la recherche par le contenu textuel de flux RSS. Thèse en informatique, Paris : Lab. Cedric – CNAM, 2013, 172 p.

HMEDEH Z, DU MOUZA C, TRAVERS N. A Real-time Filtering by Novelty and Diversity for Publish/Subscribe Systems. Publication parue dans « SSDBM'15, International Conference on Scientific and Statistical Database Management », San Diego – USA, juin 2015, pages 1-4.

HMEDEH Z, DU MOUZA C, TRAVERS N. TDV-based Filter for Novelty and Diversity in a Real-time Pub/Sub System. Conférence Internationale avec comité de lecture : IDEAS'15, International Database Engineering & Applications Symposium, Yokohama - Japon, 2015, Vol. 19, pages 136-145,

HMEDEH Z , KOURDOUNAKIS H, CHRISTOPHIDES V, DU MOUZA C, , SCHOLL M, TRAVERS N. Subscription Indexes for Web Syndication Systems. International Conference on Extending Database Technology (EDBT'12), Berlin - Germany, 2012, pages.311-322.

TRAVERS N, HMEDEH Z , DU MOUZA C, VOUZOUKIDOU N, CHRISTOPHIDES V, SCHOLL M. RSS feeds behavior analysis, structure and vocabulary. Conférence Internationale avec comité de lecture « IJWIS », vol 10, 2014, pages 291-320.

**Ressources internet**

Eclipse.org/jetty. Site officiel de Jetty [en ligne]. Disponible sur : <http://www.eclipse.org/jetty> (consulté le 24/06/2015)

Mongodb.org. Documentation officielle [en ligne]. Disponible sur : http://docs.mongodb.org/manual/

b3d.bdpedia.fr. Cours dédié aux bases de données documentaires et distribuées, proposé par le département d'informatique du CNAM. [en ligne]. Disponible sur : http://b3d.bdpedia.fr

Gartner. Article définissant le concept de *Big-Data*. 2001. [en ligne]. Disponible sur : http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf

# Annexes

## Table des annexes

## Annexe 1  diagramme de séquence de la classe SimpleMatching_V3

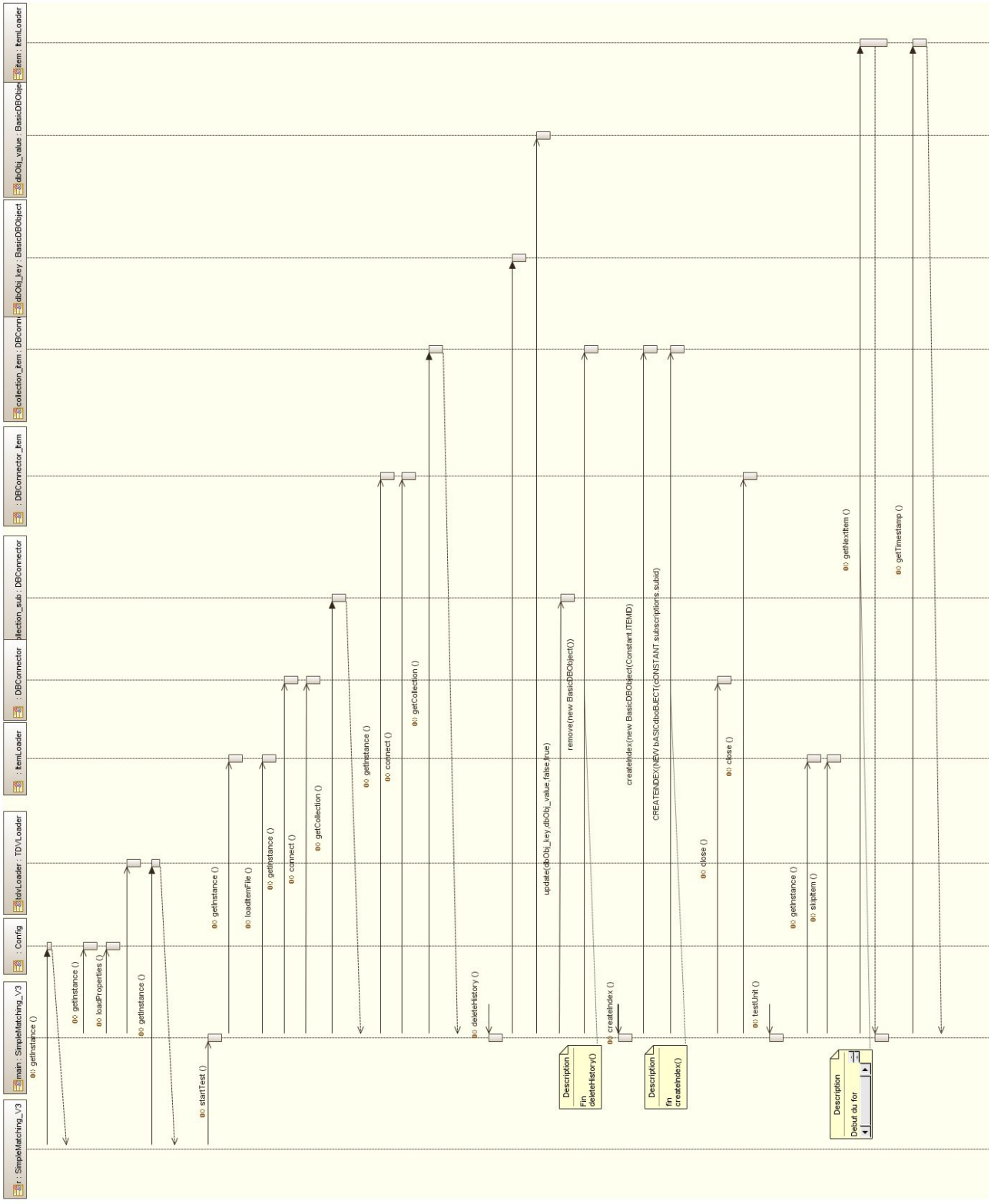


*Illustration 39: Diagramme de séquence de la classe SimpleMatching_V3 (phase 2)*

Note : la méthode *testUnit()* n'est pas détaillée dans ce diagramme, étant conséquente elle alourdirait trop le diagramme. Pour le détail de cette méthode, voir l'annexe suivante.

## Annexe 2 : Diagramme de séquence de la méthode *testUnit()*

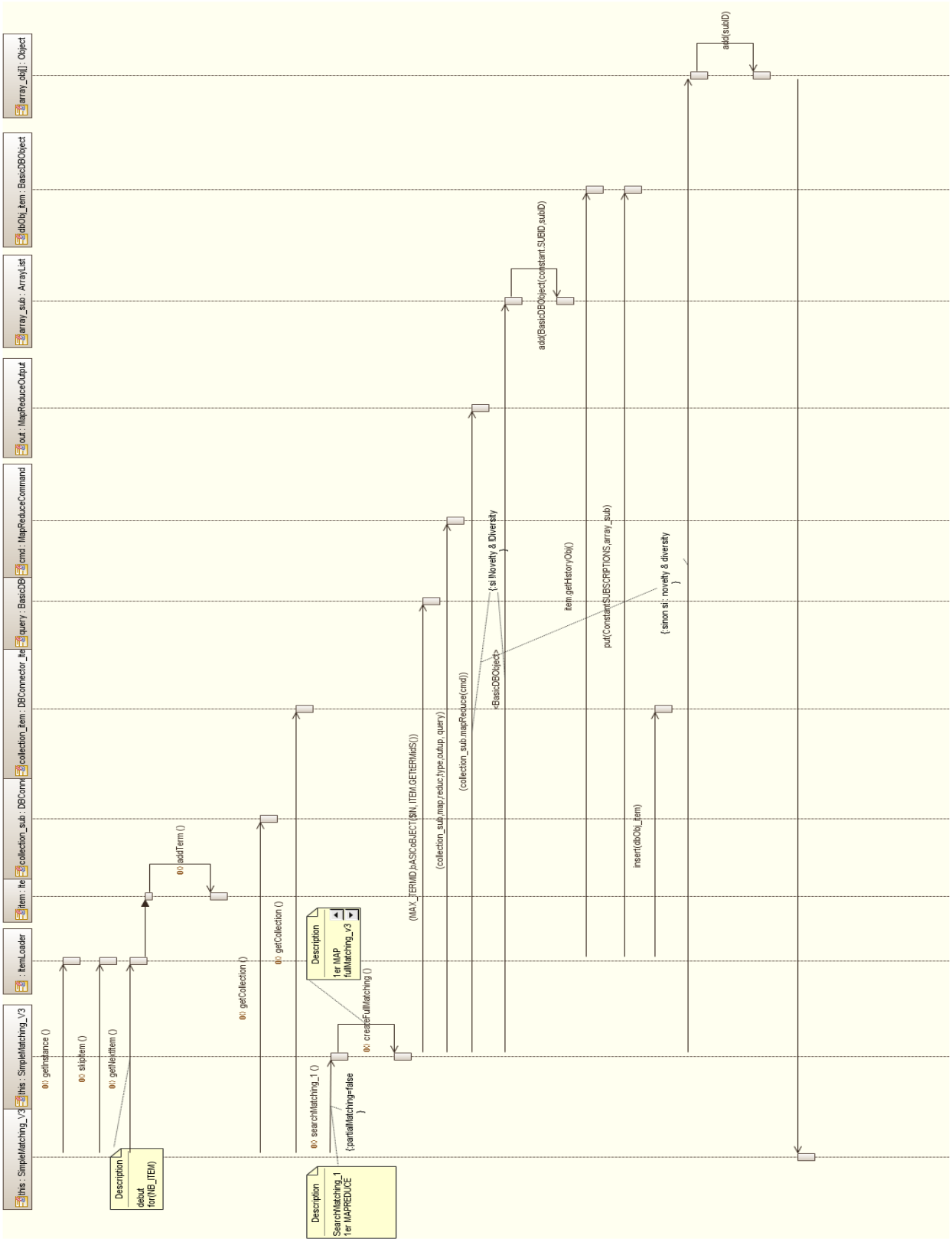


*Illustration 40: Diagramme de séquence de la méthode testUnit() (phase 2)*

# Table des illustrations

Mise en œuvre d'un Système de Publication / Souscription basé sur les Flux d'Information de type RSS

---


## Résumé

Afin d'accéder à la quantité d'informations disponible sur le Web via les flux de type RSS, le système de publication / souscription - traité dans le cadre de ce projet - propose à l'utilisateur une notification intelligente. Pour répondre à la double problématique, de gestion des données en masse et de notification en temps réel, le système a connu plusieurs évolutions. Le système, dans une phase précédente, a été implémenté autour de l'environnement de stockage distribué *NoSQL*, MongoDB avec l'objectif de valider le fonctionnement des algorithmes de filtrage. Ce mémoire, vise maintenant l'exploitation et la mise en production du système par un utilisateur final. Commençant par un travail de rétro-conception afin de comprendre l'existant, nous nous intéressons ensuite à la ré-architecture du système le rendant ainsi plus évolutif et maintenable. Puis une interface Web exploitant le *framework* JSF est intégrée, offrant une exploitation ergonomique. Enfin, sont implémentées de nouvelles fonctionnalités permettant à l'utilisateur d'exploiter les données du système et d'ajouter ses propres requêtes sans avoir à modifier le code.



**Mots clés :** syndication Web, système de publication/souscription, notification, architecture logicielle, *NoSQL*, MongoDB, *MapReduce*, Java Server Face.


---


## Abstract

To access the enormous amount of information available on the Web via RSS feeds, the publish / subscribe system concerned by this project offers the user intelligent notifications. To meet the dual problem of mass data management and real-time notification, the system has undergone several changes. The system has been previously implemented around the distributed storage environment *NoSQL*, MongoDB. The objective was to validate the operation of the filtering algorithms on this solution. This report, aims to the exploiting and final utilization of the system by an end user. Firstly, a retro-design work is undertaken to understand the existing works. Afterwards, the re-architecture of the system is pursued to make it more scalable and maintainable. Then a web interface, exploiting the JSF framework, is set providing an ergonomic use. Finally, new functionalities are implemented enabling the user to operate the system's data and add its own database requests without having to modify the code.



**Keys words :** Web syndication, publish / subscribe system, notification, software architecture, *NoSQL*, MongoDB, *MapReduce*, Java Server Face.