ISIMA
1 Rue de la Chebarde
63178 Aubière CEDEX

ORANGE
380 Rue Marcelin Berthelot
45400 Fleury-les-Aubrais

Rapport Ingénieur

Stage de 3ème année

Prototype pour la gestion de la file d'attente en boutique

Présenté par : Marie-Celine DESBROSSES

Tuteur Entreprise : Sylvain CARPENTIER

Tuteur ISIMA : Kun Mean Hou

Durée : 6 mois

Date de la soutenance : Vendredi 1er Septembre 2017

Remerciement

Je tiens à témoigner ma reconnaissance aux personnes qui m'ont aidée au cours de ce stage.

Je tiens à remercier tout particulièrement mon tuteur de stage Sylvain Carpentier, architecte logiciel, pour m'avoir accueillie et pour la confiance qu'il m'a accordée pour mener à bien mes missions. Je suis reconnaissante du temps qu'il m'a consacrée tout au long du stage.

Je remercie mon tuteur de l'école, Kun Mean Hou pour sa bienveillance au bon déroulement de ce stage.

J'adresse mes remerciements à Antoine Diaz, Romain Léopold, Michel Decima ainsi que l'ensemble de la communauté des architectes référents et des développeurs pour l'aide et l'expertise apportés pour les développements des prototypes. Je remercie aussi toute l'équipe de PNS.com, François Rivier, Clémenti Eliana, Brahami Saliha, Marcelet Xavier et Martellotto Laura pour leurs disponibilités à répondre à mes questions et du soutien technique qu'ils m'ont apportée.

Enfin, je remercie Florent Jeannot et toutes les personnes qui, de près ou de loin, se sont investies et m'ont portée de l'intérêt, elles ont fait de ce stage une expérience enrichissante.

Résumé

Dans le cadre d'Essentiel 2020, Orange et la DSI RCGP engage la rénovation de son SI pour s'adapter aux besoins de ses clients (digitalisation, multi-device et multi-canalité). C'est dans ce cadre que l'application de bienvenue en boutique/Gestion de la **File d'Attente** (GDFA) subit de grands changements d'architecture et intègre une API pour ses clients.

Pour améliorer les performances de la gestion de la file d'attente en boutique, le stage propose de tester la faisabilité et la validité de solutions permettant de gérer l'asynchronisme du service de l'envoi de SMS, ainsi que des solutions de persistance des données. Les prototypes développés utilisent des APIs proposées par PNS.com.

La première solution implémente le pattern « **Publish and Subscribe** » en utilisant une API de **bus de messages**. Cette solution permet de gérer les services de GDFA indépendamment des appels de services à ORMIS. La deuxième solution teste à travers plusieurs prototypes l'utilisation d'API pour faire appel à différentes **bases de données NoSQL**. Les types de bases NoSQL étudiés sont le graphe, le clé/colonnes et le document. Cette étude met en évidence les différences entre les solutions traditionnelles de SGBD Relationnelle et les solutions NoSQL, au niveau de la scalabilité, la flexibilité, des transactions et de la modélisation…

Les prototypes développés ont permis de voir les impacts de l'intégration des APIs, afin d'échanger sur ce point avec les équipes de PNS.Com dans le but de les faire évoluer.

Mots Clés : API, File d'Attente, « Publish and Subscribe », bus de messages, bases de données NoSQL

Abstract

As part of "Essentiel 2020", Orange and DSI RCGP are committed to renovating their IS to adapt to their customers' needs (digitization, multi-device and multi-channel). It is within this framework that the welcome application in store / **queue** management (GDFA) undergoes many architectural changes and integrates an **API** for its customers.

To improve the performance of the management of the waiting list in the shop, the traineeship proposes to test the feasibility and the validity of solutions allowing to manage the asynchronous service of the sending of SMS, as well as solutions of persistence of the One Solution for data persistence. The developed prototypes use APIs offered by PNS.com.

The first solution implements the "**Publish and Subscribe**" pattern using a **messages bus** API. This solution allows the management of the GDFA services independently of the calls of ORMIS' services. The second solution tests through several prototypes the use of API to implement different **NoSQL databases.** The SQL database types studied are the graph, the key / column and the document. This study highlights the differences between traditional Relational DBMS solutions and NoSQL solutions, in terms of scalability, flexibility, transactions and modeling...

The developed prototypes allowed to see the impacts of APIs integration and to discuss about it with the PNS.Com teams in order to improve them.

Key Words: API, queue, "Publish and Subscribe", messages bus, NoSQL databases

Table des matières

Table des Figures

Introduction

Pour réduire le temps d'attente des clients dans une boutique Orange, ceux-ci sont enregistrés en file d'attente dès leurs arrivées par l'application de Gestion De Files d'Attente (GDFA). GDFA gère l'accueil des clients soit physiquement dans les boutiques Orange, soit au travers des divers processus de « prise de rendez-vous » sur le site web, smartphone ou tablette. Les clients sont enregistrés dans la file d'attente selon leur profil et motif de visite, un calcul de priorité et de position est réalisé. Elle informe le client de l'évolution des différentes files d'attente en fonction de la typologie de la boutique et de l'avancement de leur demande et l'oriente ensuite vers un vendeur disposant de la bonne compétence.

Cette application est déployée dans environ 400 boutiques Orange en France. L'application GDFA est identifiée comme un composant cœur de la rénovation de la prise en charge des clients en boutique. Ce composant doit évoluer pour être en harmonie avec la future position de travail « full digital ».

L'objectif de mon stage est d'étudier l'intégration de nouveaux composants dans l'application et de mettre en perspective leurs impacts dans le cadre d'une éventuelle industrialisation. Je me suis intéressée sur la possibilité d'intégrer les API de PNS.com à travers différents prototypes. PNS (Profile and Syndication) est un service au sein de la DSI d'Orange qui publie des API pour les besoins des applications Orange (bus AAS). On se propose d'étudier les APIs de PNS.com qui permettent l'asynchronisme intelligent (pattern PUB/SUB), ainsi que la mise en œuvre de stockage de données NoSQL sur trois types de modélisations (Graphe, Clef/valeur-colonnes et document).

Comment gérer les envois de SMS de manière asynchrone ? Quels types de données doit-on gérer en NoSQL et pour quels besoins ?

Dans un premier temps, je présenterai le contexte de l'étude. Ensuite j'expliquerai le premier prototype développé dans le cadre de la gestion de l'envoi des SMS. Ensuite, j'exposerai les différents prototypes envisagés pour la persistance des données s'appuyant sur les modèles NoSQL. Pour finir, j'analyserai les résultats obtenus aux travers des prototypes.

I. Contexte de l'étude

1) Présentation de l'entreprise

a) Description générale d'Orange

Orange, société française de télécommunication, est l'un des principaux opérateurs européens et africains du mobile et de l'accès internet ADSL. Elle est également l'un des leaders mondiaux des services de télécommunication aux entreprises et par conséquent, possède une renommée internationale. Le groupe est présent auprès du grand public dans 28 pays et propose des services de connectivité dans plus de 220 pays et territoires.

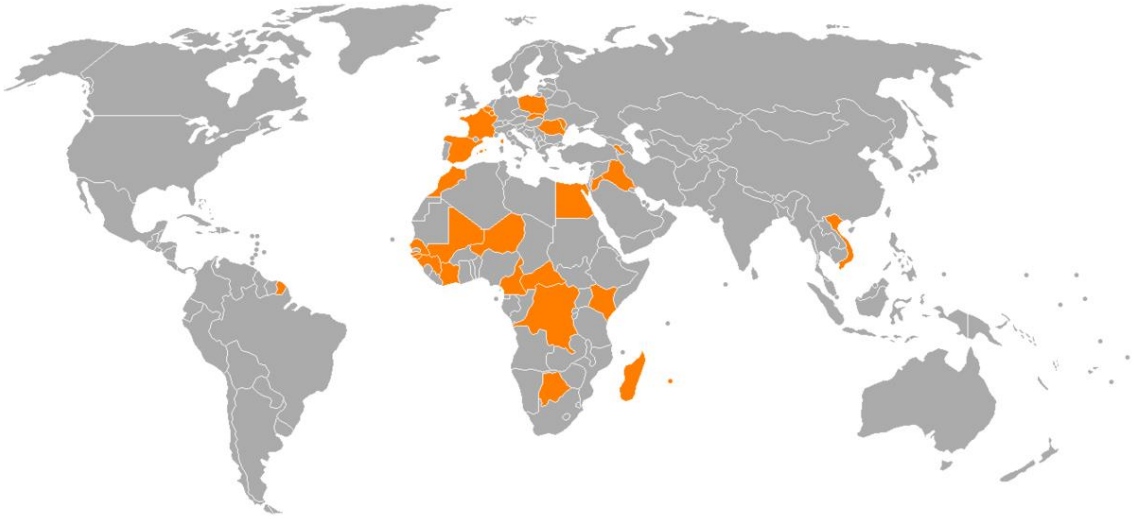

Figure 1 : Répartition géographique d'Orange dans le monde

b) Historique

Orange, héritier de France Télécom, porte les valeurs d'un groupe mondial d'origine française, fier de ses racines, mais aussi fier de ses conquêtes à l'échelle mondiale. Une épopée à découvrir à travers des moments forts qui constituent la mémoire de cette entreprise, d'hier à aujourd'hui. Voici les événements retraçant l'histoire de l'entreprise mais aussi les innovations qui constituent le socle commun de l'histoire des télécommunications.

- 1794 : premier message télégraphique transmis sur la ligne Lille-Paris : naissance de la télégraphie optique de Chappe
- 1923 : création du Ministère des Postes, Télégraphes et Téléphone (PTT)
- 1941 : création de la Direction des Télécommunications issue de la fusion de la direction de l'exploitation téléphonique et de la direction de l'exploitation télégraphique
- 1946 : la Direction des Télécommunications crée en 1941 devient la Direction Générale des Télécommunications (DGT)
- 1988 : la Direction Générale des Télécommunications devient France Télécom
- 1996 : transformation de France Télécom en société anonyme dont l'Etat est le seul actionnaire

- 2000 : acquisition de l'opérateur de téléphonie mobile Orange, marque créée en 1994
- 2004 : France Télécom devient une société privée
- 2006 : Orange devient la marque unique pour le mobile, la télévision, l'internet et les services numériques sur les principaux marchés
- 2013 : France Télécom change de nom et devient Orange

c) Structure et Organisation

Ci-dessous, voici l'organigramme représentant la hiérarchie entre le Président Directeur général, Stéphane Richard et le service dans lequel j'ai travaillé, le service UPLOAD.

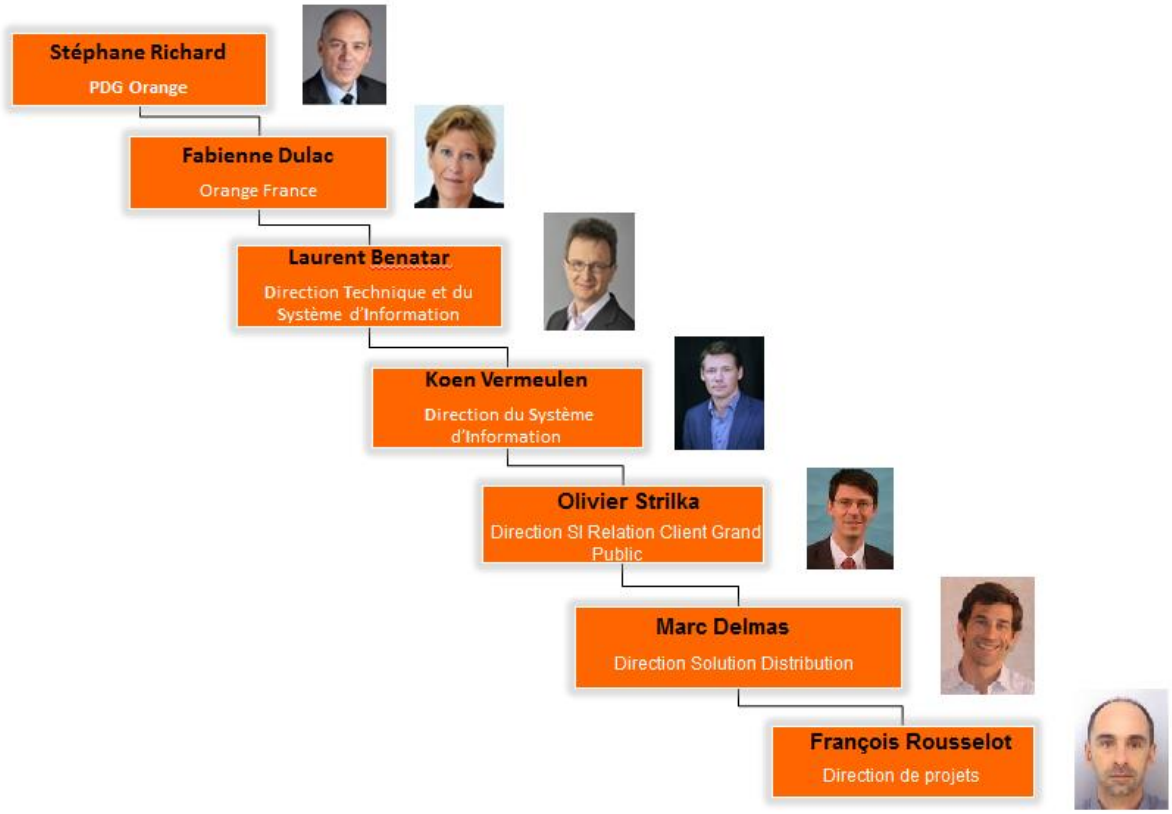

Figure 2 : Organigramme simplifié Orange

A l'échelle internationale, Orange s'organise autour de plusieurs directions dont Orange France, gérée par Fabienne Dulac, Directrice Exécutive. Au sein de la direction d'Orange France, il existe également d'autres directions dont la Direction du Système d'Information France (DSIF) présidée par Laurent Benatar. Elle est composée de 3400 personnes dont la mission principale est de concevoir, développer et intégrer les systèmes d'information (SI) d'Orange afin de mieux répondre aux attentes des besoins business et mieux satisfaire ses clients. Elle concerne donc l'ensemble des métiers qui interviennent pour maintenir et faire évoluer le SI. Pour atteindre ses objectifs, la DSI se décline en 13 directions parmi lesquelles se trouve la Direction des Relations Client Grands Publiques (RCGP) dont Olivier Strilka est responsable. Dans cette direction, on retrouve la direction solution assistance client,

distribution commerciale conduite par Marc Delmas. Enfin, dans cette dernière, l'entité UPLOADS (Urbanisme, Prototypage, Logiciel, Accompagnement, Devops, Sécurité), représentée par François Rousselot, est l'entité de rattachement de mon tuteur de stage, Sylvain Carpentier, architecte solution. Ce département transverse a pour mission de répondre aux objectifs techniques de la DSI RCGP et instruire les enjeux du programme « Essentiel 2020 » d'Orange sur les solutions « API » et « DEVOPS », sur les méthodes et les outils, et enfin d'accompagner et de transmettre les bonnes pratiques et les compétences dans les projets.

2) Diagramme de Gantt

Etapes	Num	début	durée en jour	fin
découverte	0	27/03/2017	20	16/04/2017
Développement Pattern Pub & Sub pour gestion SMS	1	17/04/2017	21	08/05/2017
Prototypage pour NoSQL Graphe	2	09/05/2017	27	05/06/2017
Prototypage pour NoSQL clé/Val	3	06/06/2017	20	26/06/2017
Prototypage pour NoSQL Doc	4	27/06/2017	20	17/07/2017
Synthèse sur utilisation de bdd NoSQL	5	18/07/2017	6	24/07/2017
Conclusion et Rapport	6	25/07/2017	13	07/08/2017
Gestion de la File d'Attente dans la boutique	Option	08/08/2017	45	22/09/2017

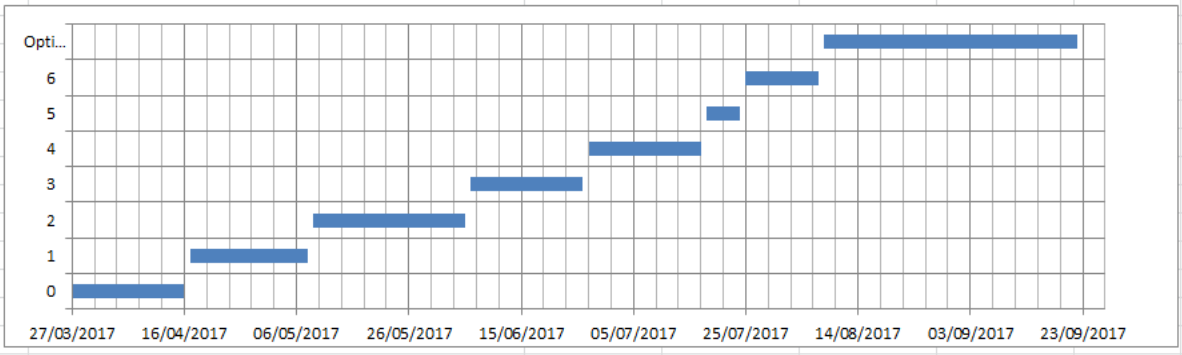

Figure 3 : Diagramme de Gantt prévisionnel

La première étape me permet de construire et de me familiariser avec l'environnement de travail. Cela m'a permis de découvrir la gestion de projets avec Maven, les composants de la génération automatique de code et de découvrir le socle logiciel de l'application existante.

La deuxième étape se concentre sur l'étude du pattern « Publish and Subscribe » proposé par l'API Bus As A Service de Pns.com utilisée pour implémenter ce pattern. Cette étape comporte la création d'un connecteur client de l'API qui utilise notamment la génération automatique de code ainsi que le service métier de gestion de SMS client. Un nouveau composant logiciel est développé pour permettre l'orchestration des services techniques (PUB/SUB) et métier.

La troisième étape sera l'utilisation des services de base de données NoSQL orientée graphe de PNS.com. On retrouvera la réalisation du modèle et l'implémentation du service pour les 3 cas d'utilisations suivant : nouveau client, client suivant et client supprimé. L'étape la plus chronophage étant la première solution NoSQL (Graphe).

Les étapes 4 et 5 auront réitéré ce qui aura été fait dans l'étape précédente mais pour les bases de données orientées document et clé/valeur. Ces étapes sont plus courtes car le code développé pour la base de données orientée graphe peut être réutilisé.

Enfin, une étude comparative est menée pour identifier et décrire les avantages et inconvénients de chaque modèle.

Le reste du temps est consacré à la rédaction du rapport et la synthèse.

A l'issue des étapes, une étude complémentaire va me permettre d'étudier la refonte de l'algorithme de la file d'attente et d'utiliser les fonctionnalités d'une base de données NoSQL de type document notamment MongoDB.

Etapes	Num	début	durée	fin
découverte	0	27/03/2017	21	17/04/2017
Développement Pattern Pub & Sub pour gestion SMS	1	18/04/2017	35	23/05/2017
Prototypage pour NoSQL Graphe	2	15/05/2017	30	14/06/2017
Prototypage pour NoSQL clé/Val	3	15/06/2017	20	05/07/2017
Prototypage pour NoSQL Doc	4	06/07/2017	15	21/07/2017
Synthèse sur utilisation de bdd NoSQL	5	22/07/2017	3	25/07/2017
Conclusion et Rapport	6	26/07/2017	26	21/08/2017
Gestion de la File d'Attente dans la boutique	Option	01/08/2017	52	22/09/2017

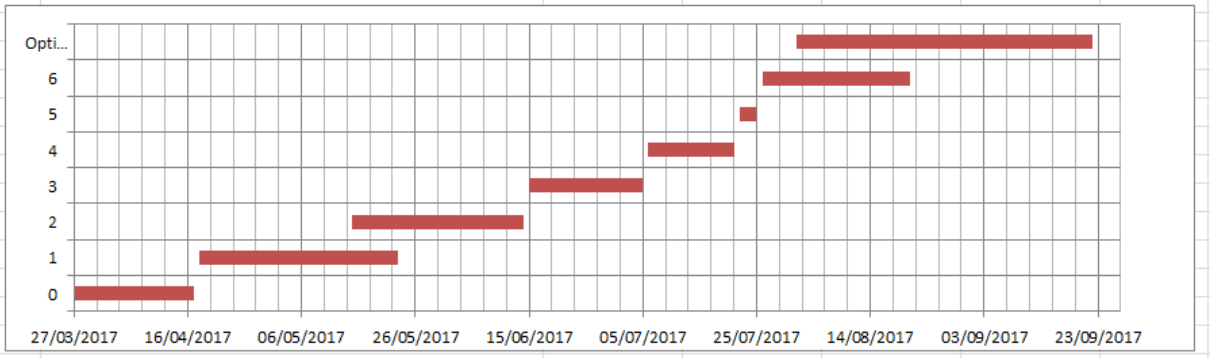

Figure 4 : Diagramme de Gantt final

Comme on peut le voir sur la figure, le développement du prototype de l'asynchronisme fût plus long que prévu. Cela est dû à la découverte de la génération de code et de l'utilisation de l'API. J'ai eu plusieurs difficultés pour accéder à l'API, notamment liés :

- aux paramétrages du Proxy,
- à la gestion des certificats,
- autorisations d'accès au bus de messages,
- à la conversion de message en objet JAVA.

Le premier prototype pour la persistance de données avec une base orientée graphe a demandé un peu plus de temps. En effet, il a fallu définir le périmètre fonctionnel pour le prototype pour répondre au cas d'utilisation et proposer une modélisation. A l'issue de ses premiers travaux préparatoires, la documentation de l'API a été revue pour prendre en compte le type de réponse lors de la récupération d'un nœud par une requête.

Le prototype sur la persistance de données avec une base orientée clé/valeur a évolué vers une base orientée colonne, car l'API, malgré son nom, utilise les colonnes pour stocker les données. Le modèle a beaucoup évolué car il y a des limitations au niveau de l'API.

Par exemple, on ne peut avoir plus de deux clés pour définir la donnée.

De plus, une fois le modèle choisi, il faut créer l'environnement du côté de la base (les tables et les colonnes définies).

Le prototype sur la persistance de données avec une base orientée document a été plus rapide à développer car les composants et l'expérience acquise lors des étapes précédentes ont pu être capitalisés. De plus, l'environnement pour ce prototype était déjà opérationnel.

3) Présentation des outils

L'application de la gestion de file d'attente utilise les outils suivant :

Spring Boot est un Framework permettant de simplifier le développement d'applications Spring. Il permet d'auto-configurer les composants détectés sur le classpath (par exemple, si Spring Boot détecte le driver Java de Mongo, ou un driver JDBC, ou Tomcat, etc., alors il configure automatiquement un ensemble de Bean (au sens Spring) pour utiliser ces composants) et de personnaliser les composants, pour passer outre l'auto-configuration. Il simplifie le déploiement (un jar unique, ou un war pour Tomcat).

Spring Tool Suite est un environnement de développement basé sur Eclipse qui est personnalisé pour développer des applications Spring. Il fournit un environnement prêt à utiliser pour implémenter, déboguer, exécuter et déployer les applications Spring, y compris les intégrations pour Git, Maven ...

Brackets est un éditeur de texte open source développé par Adobe qui facilite le développement sur des technologies Web. On remarque notamment l'édition rapide de fichiers ou encore l'affichage d'un nuancier pour faciliter la modification des codes de couleur dans les feuilles de styles éditées.…

Apache Maven est un outil pour la gestion et l'automatisation de production des projets logiciels Java. Il permet d'automatiser certaines tâches : compilation, tests unitaires et déploiement des applications qui composent le projet, de gérer des dépendances vis-à-vis des bibliothèques nécessaires au projet, de générer des documentations concernant le projet. Maven utilise un paradigme connu sous le nom de Project Object Model (POM) afin de décrire un projet logiciel, ses dépendances avec des modules externes et l'ordre à suivre pour sa production. Il est livré avec un grand nombre de tâches prédéfinies, comme la compilation de code Java ou encore sa modularisation.

MySQL (serveur/client) est un système de gestion de base de données relationnel.

Pour tester les différents prototypes, je me suis appuyée sur **:**

Python est un langage de programmation interprété, c'est-à-dire que les instructions qui lui sont envoyées, sont « transcrites » en langage machine au fur et à mesure de leur lecture. Python possède un nombre important de bibliothèques logicielles externes. Les bibliothèques ajoutées sont Requests et MySQLdb. **Requests** est un module python permettant d'utiliser le protocole HTTP pour envoyer des requêtes à un serveur. Il gère notamment les proxys, les cookies, les vérifications SSL, « upload » de fichiers multiparts … **MySQLdb** est un module pour se connecter à une base de données MySQL afin d'exécuter des requêtes MySQL.

Postman est un outil permettant d'exécuter des requêtes HTTP sur un serveur pour en interpréter la réponse en dehors de tout contexte métier. Les requêtes sont facilement créées et envoyées pour tester une API. Les requêtes peuvent être sauvegardées en local et permettent d'avoir des collections de tests.

4) Présentation de l'existant

Pour répondre aux nouveaux besoins et s'adapter aux supports tablettes/smartphones, GDFA devient une API.

a) Vue fonctionnelle

L'application comme son nom l'indique permet la gestion de la file d'attente en boutique.

Plusieurs acteurs interviennent dans les processus :

- **Responsable de boutique** (RB): personne qui gère la boutique
- **Administrateur national** : personne qui gère les référentiels de l'application (« users », compétences, organisation etc …)
- **Pilote** : personne accueillant le client en boutique, il peut aussi être un conseiller
- **Conseiller** : personne prenant en charge la demande du client
- **Chip PC en boutique** : un ordinateur pour la diffusion des messages sur les écrans de la boutique ainsi que l'affichage de file d'attente en temps réel
- **ORMIS** : une application pour la gestion des envois en masse de messages multicanaux (SMS, MMS, WAP Push, mails…) pour les applications Orange France

Le diagramme de use case montre les différents rôles attribués aux acteurs et les cas d'utilisations de GDFA.

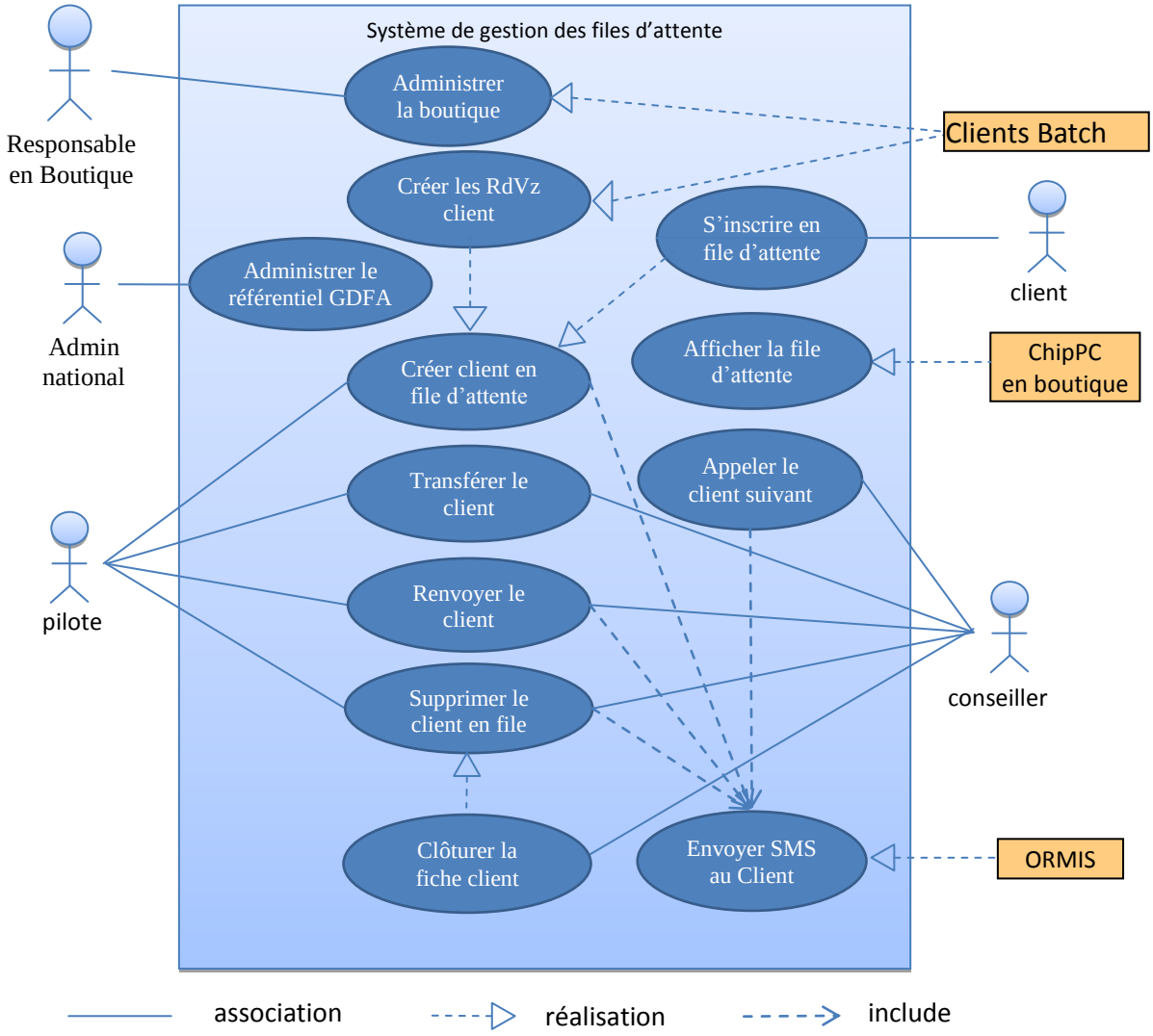

Figure 5 : Diagramme de use case fonctionnel

Administrer la boutique **:** consiste à mettre à jour le référentiel et les paramètres concernant la boutique dans la base de données par les responsables.

Clôturer l'entretien du client : permet au conseiller, de clôturer l'entretien du client. La clôture de l'entretien entraîne la suppression du client en file d'attente et la mise à jour du statut de la fiche client.

Fonctionnement de la GDFA, calcul du rang : choix files et priorités :

Le choix-file a son propre niveau de priorité, indépendant de sa file écran. Il correspond à un motif de visite et/ou de profil du client, mais également à une compétence vendeur. Le niveau de priorité des choix files comporte 5 niveaux différents (0 à 5). Plus le chiffre est

faible, plus la priorité est importante. Pour le calcul de la file d'attente, on doit respecter des règles de gestion : un client ne peut être dépassé que par deux clients ayant des priorités supérieurs. Les « solutions handicaps » sont réservées aux handicapés et femmes enceintes. Pour éviter de les faire attendre, elles sont des exceptions à cette règle et passent en premier dans tous les cas.

Fonctionnement de la GDFA, appel du client suivant :

Le vendeur appelle le client actif le plus haut de la file d'attente et correspondant aux compétences dont il dispose.

Clôture de l'entretien (de la fiche GDFA), si :

- le client est accueilli et servi (« client accueilli »)
- le client a quitté la boutique quand on l'appelle (« client parti »)

Un client reçu et redirigé vers l'espace services :

- retourne en file d'attente (pour un conseil technique)
- respect des règles de priorités
- conserve son heure d'arrivée

Un client venant à l'espace services et redirigé :

- retourne en file d'attente (selon le nouveau choix-file)
- respect des règles de priorités
- conserve son heure d'arrivée

Un client reçu et redirigé vers un autre choix-file

- dû à une mauvaise orientation du pilote
- ne peut être appelé par le même vendeur
- conserve son heure d'arrivée

Fonctionnement de la GDFA, gestion des fiches inactives :

Sur l'écran d'affichage, les rendez-vous apparaissent 30 minutes avant l'heure du rendez-vous. La fiche client est inactive, par contre le client est numéro 1. Le client ne peut être appelé dans ce cas car sa fiche est inactive. Lors de l'arrivée du client, le pilote active sa fiche. Le client peut arriver en retard, peu importe, il sera toujours numéro 1 malgré tout. Il sera appelé par un vendeur dès lors que sa fiche sera active et sera prioritaire.

Fonctionnement de la GDFA, gestion des SMS

Le client peut demander, lors de son inscription en file d'attente, la gestion de sa demande par SMS. Trois types de SMS lui sont proposés indépendant les uns des autres :

- « SMS - Inscription » : un SMS est envoyé au client dès lors que son inscription est validée par un ticket en file d'attente
- « SMS - C'est ton tour » : un SMS est envoyé au client quand le conseiller l'appelle pour l'entretien
- « SMS Shopping » : un SMS est envoyé aux clients dont le rang en file d'attente est « n-1 et n-2 », avec n représentant le rang du client en entretien pour la file d'attente concernée

b) Vue logicielle

Le modèle en couche choisi est sur trois niveaux majeurs que ce soit le Legacy (l'ancienne application) ou l'API.

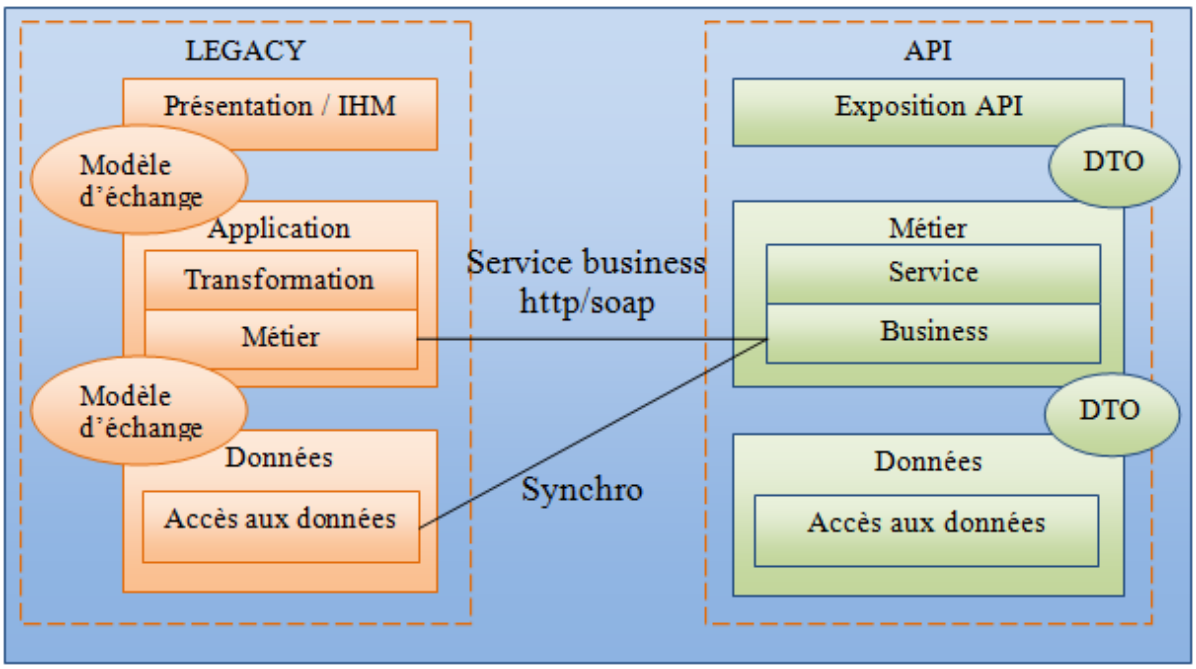

Figure 6 : le modèle en couches logicielles

i. Le modèle en couche du Legacy :

La couche présentation / IHM (interface Homme-Machine) est le développement des écrans de l'application GDFA.

La **couche Application** est une couche de traitement applicatif qui offre des services de haut niveau à la couche supérieure et qui invoque les services métiers de la couche métier.

La **couche Métier** est une couche de traitement métier pour manipuler les objets métiers comme les clients, files d'attentes…, gérer les interfaces avec les partenaires (services métiers back office), mais aussi de s'appuyer sur des services de haut niveau concernant le traitement de la persistance en base de données.

La **couche transformation** est une couche qui permet de réaliser le « mapping » entre la couche Application et la couche métier d'une part et d'autre part de la couche métier vers les ressources (Base de données, partenaire back office).

La **couche accès aux données** permet d'accéder à la base de données et comporte les « Entity » et les « DAO » (Data Object Transfert).

ii. Le modèle en couche de l'API :

La **couche d'interface** API est une couche logicielle qui permet de gérer les requêtes de message client de l'API (le protocole, l'architecture REST, les formats des données…).

La **couche logique service** permet de fournir des services de haut niveau à la couche supérieure, l'interface de l'API. Elle a pour principales missions d'orchestrer les services de la couche inférieure et d'agréger les données.

La **couche logique business** a pour mission de gérer les différentes ressources de l'application, ainsi que les « mapping » entre couches « Application » et « Métier » et couches « Métiers » et « Ressources » (SGBD, Interfaces externes).

La **couche accès aux données** a pour rôle d'héberger les « Entities » et les « DAOs ».

La **couche transformation** est transverse et elle permet d'héberger les « Mappings » techniques et fonctionnels. L'API comporte deux « Mappings » :

- un « Mapping » entre la représentation API (REST/JSON) et la représentation Business (Bean). Les objets métiers ou « Bean » sont appelés « DTO » (Data Transfert Object) dans l'API.
- un « Mapping » entre les Bean métiers (« DTO ») et les « Entities ».

II. Prototype de l'asynchronisme

Ce premier prototype a pour objectif de désynchroniser les services composants de l'API de GDFA et des services de l'application ORMIS, pour l'envoi des SMS. Cela garantit l'indépendance des processus Front des fonctions Back et ainsi proposer une architecture moins monolithique et plus réactive aux évolutions et à la gestion de la maintenance. Les clients de l'API deviennent indépendants des traitements de back office qui peuvent parfois être « lourd ». Par exemple, pour notre cas d'utilisation, l'indisponibilité d'ORMIS sera transparente pour le traitement des processus Front. Les latences réseaux, ainsi que la saturation d'ORMIS seront également transparentes pour le traitement des processus.

Pour mettre en œuvre cette solution, l'API implémentera le pattern « Publish and Subscribe » à travers le dialogue par un bus As A Service. Le bus utilisé sera une API proposée par PNS.com.

1) Présentation du « pattern »

Le pattern « publish and subscribe » s'appuie sur la notion de « Topic ». Les Topics sont des files d'attente particulières basées sur le modèle « publish and subscribe ». Les messages sont envoyés dans le Topic. Il y a autant de copie du message qu'il y a d'abonnés au Topic. Quand tous les abonnés ont consommé le message, le Topic est alors vide.

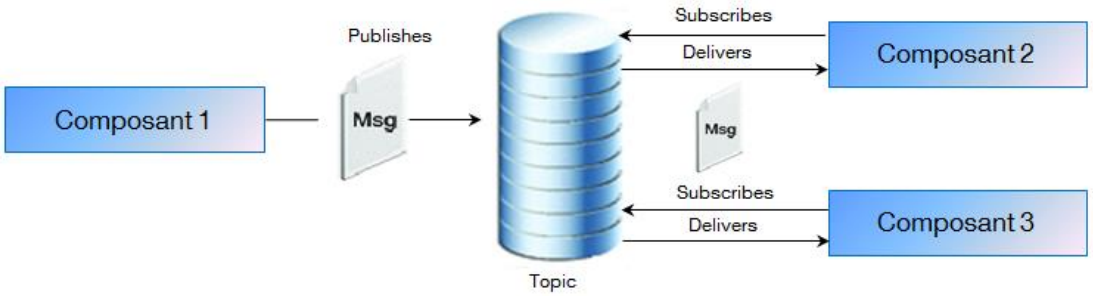

Figure 7 : Schéma du Pattern « Publish and Subscribe »

Pour le prototype, nous nous intéresseront au processus de création d'un client en file d'attente et le processus d'appel du client suivant et à l'envoi de SMS de confirmation de l'inscription et de SMS de l'appel du client.

2) Présentation de l'API de PNS.com

Le bus de messages « ZBus » est un service de publication et souscription qui permet d'envoyer et de recevoir des données en mode "As A Service".

a) Architecture et fonctionnement

Le système peut être représenté comme sur la figure suivante :

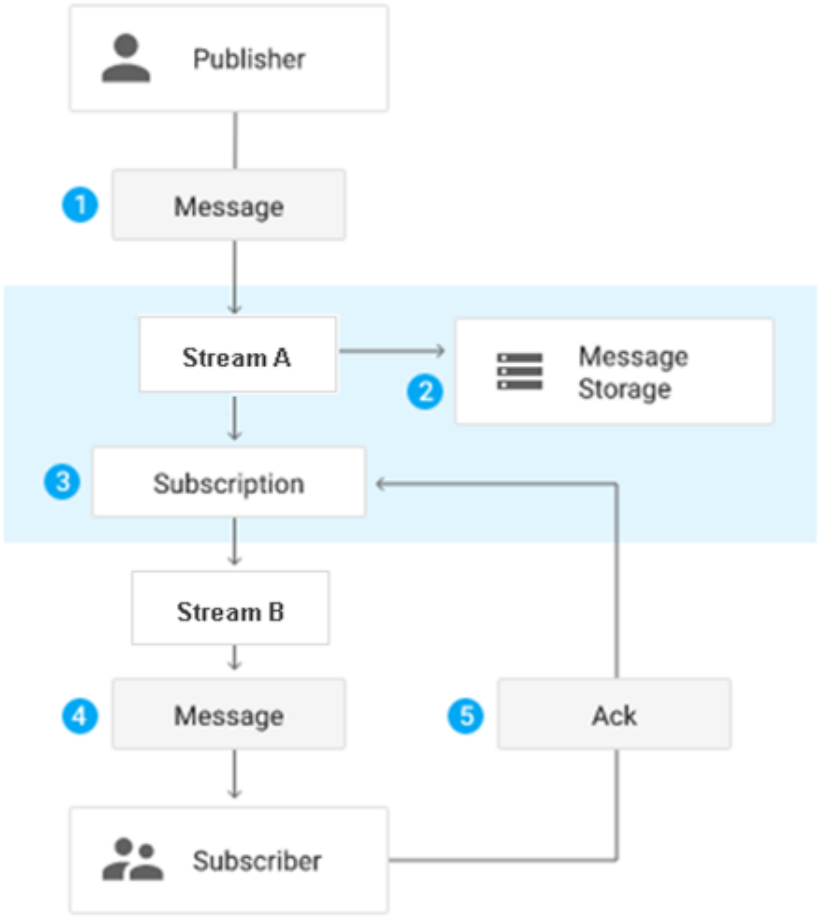

Figure 8 : Schéma d'utilisation de l'API

1. Une application d'éditeur crée un flux et envoie des messages à ce flux. Un message contient une charge utile et des attributs optionnels qui décrivent le contenu de la charge utile et le comportement du flux.
2. Les messages sont conservés dans une file de messages jusqu'à ce qu'ils soient livrés et reconnus par les abonnés.
3. Le service Pub / Sub expédie les messages à un flux pour tous ses abonnés, individuellement.
4. L'abonné reçoit les messages en attente de son abonnement et reconnaît chacun au service Pub / Sub.
5. Lorsqu'un message est reconnu par l'abonné, il est supprimé de la file d'attente de l'abonnement.

La publication crée et envoie des messages dans un flux. La souscription crée une souscription à un flux pour recevoir les messages. La communication peut être de plusieurs types : one-to-many, many-to-one, one-to-one ou many-to-many. Cela permet de découpler les expéditeurs des récepteurs. La fonctionnalité d'acquittement garantit la livraison des messages. Les messages restent disponibles jusqu'à ce que le client reconnaisse leur

réception. Il existe deux modes d'accusé de réception permettant de déterminer si un message a été envoyé ou non : automatique ou manuel avec une requête.

b) Utilisation du bus de messages

i. Authentification

Pour avoir accès à l'API « ZBus », le client (producteur ou consommateur) doit s'abonner au service « ZBus » et obtenir un « keytab » associé à l'API en passant par OKAPI. Le client peut appeler OKAPI pour obtenir un « Token » appelé aussi un jeton d'authentification. Ce jeton est utilisé pour s'authentifier lors de l'appel de l'API et accéder à la ressource. Ce « Token » a un délai d'expiration, il doit donc être régénéré périodiquement. Toutes les API de PNS.com utilisent ce mode d'authentification.

ii. Publication de messages

Le client peut publier les messages un par un ou par lot par la méthode POST. Le corps de la requête est une liste en format JSON. Les crochets doivent toujours être présents lorsque le client envoie un ou plusieurs messages JSON.

iii. Consommation de messages

Les clients peuvent consommer des messages un par un ou par lot avec la méthode GET. Le paramètre « limit » de la requête spécifie le nombre de messages maximum à récupérer. Le corps de la requête retourné est une liste d'objets.

L'acquittement de messages permet de confirmer que les messages sont reçus et gérés par le consommateur. L'acquittement est réalisé par la méthode POST par l'utilisation d'un id unique qui est l'identifiant retourné en réponse à une requête GET pour identifier le paquet de messages. Cette action est dénommé « AckID »

Si un incident survient pendant le traitement des données, le consommateur peut demander à « ZBus » de signaler que les messages ne sont pas consommés (c'est-à-dire non reconnus). Le système conserve les données et les transmet ultérieurement. Cette API a la même utilisation que la demande d'acquittement : une demande POST avec les mêmes en-têtes et les données vides dans la réponse. Un code d'état est renvoyé pour indiquer si la requête est correcte ou non.

c) La gestion des erreurs

Le format d'erreur « ZBus » est composé d'un code d'état « http » et d'un corps de réponse qui décrit l'erreur. Le corps contient un code d'erreur d'application, une description courte et une description longue qui donne plus de détails.

Exemple :

```
Status code: 401
{
  "code": 1,
  "message": "Missing or invalid credentials",
```

```
  "description": "Missing, invalid or expired request token. Please provide or
renew OKAPI token"
}
```

Les erreurs courantes sont dans le tableau ci-dessous :

statut http	Description	Raisons
400	Bad request	- valeur du paramètre invalide - le body (corps du message) manquant - le body invalide - un champ du body manquant - le header (en-tête du message) manquant - une valeur d'un header invalide
404	Resource not found	- URL non valide - manque un paramètre du chemin - identifiant du Stream (flux) est invalide
401	Client not authorized	- Token invalide - Token expiré.
403	Forbidden Access	- client non autorisé. Exemple : consommateur ne peut pas publier
412	Precondition failed	- Les conditions de la demande ne sont pas respectées. Exemple: extraire des messages lorsque le client dépasse le maximum de messages.
405	Method not allowed	- Méthode non valide. Exemple: POST au lieu d'un GET.
500	Internal server error	- Incident avec le serveur

3) Implémentation

a) Génération d'un client JAX-RS à partir du swagger

Pour développer un client java de l'API rapidement, on peut utiliser le composant de la génération automatique du code.

Le générateur de code transforme une spécification d'API au format swagger en classe JAVA : des interfaces pour les ressources et les classes pour les représentations des données. Le code produit est automatiquement formaté et compilé. Comme les dépendances sont ajoutées au projet, cela permet de limiter les dépendances à « Jersey2 ».

Pour chaque ressource, deux interfaces sont générées. L'une est une interface avec les valeurs de retour « typées » (représentation des données retour spécifié) et l'autre est « non typée » (la réponse retour est une instanciation de la classe `Response` de java). La ressource du côté serveur implémente l'interface non typé. Cela permet d'avoir une réponse générique et de pouvoir construire les « Headers » de la réponse.

Avec quelques lignes dans le « pom.xml » du module, on obtient un client opérationnel grâce au « pattern Proxy » du « Framework Jersey » en utilisant l'interface « typée ». L'interface « typée » permet de mapper directement la réponse de l'API avec l'objet retour défini dans le swagger.

A tout moment, il est possible de générer à nouveau le code sans écraser l'implémentation réalisée par le développeur.

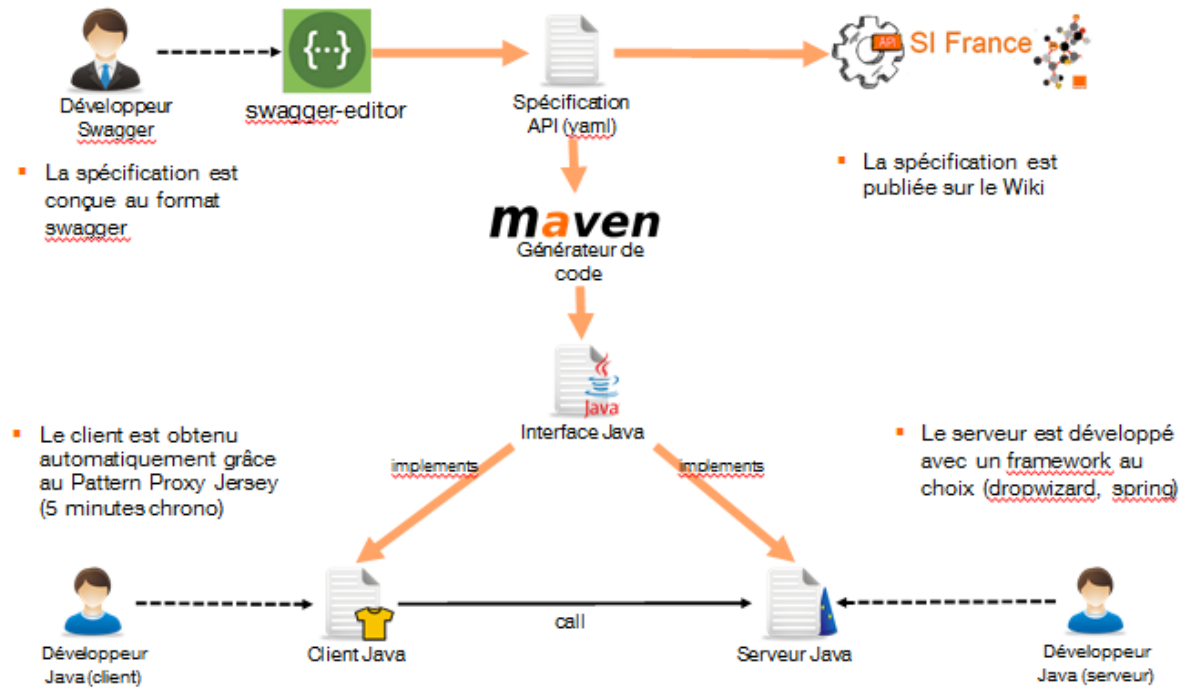

Figure 9 : Schéma de développement des spécifications de génération de code

b) Intégration à l'application

Pour implémenter le « ZBus », il faut développer deux clients « JAX-RS » : un pour l'authentification et un pour communiquer avec le bus. Le message envoyé et récupéré par le bus est un « POJO » de messages du type `MessageDTO`. Ce `MessageDTO` contient deux attributs : le processus en cours du client et son identifiant. À partir de ses deux informations, on peut récupérer dans la base les données nécessaires pour envoyer le SMS au client. De plus, il est nécessaire de développer des « managers » prenant en charge la publication et la souscription avec l'acquittement, ainsi qu'un un service pour souscrire les messages afin d'appliquer les traitements métier demandés. Le manager de la publication sera intégré directement au service existant de création d'un client.

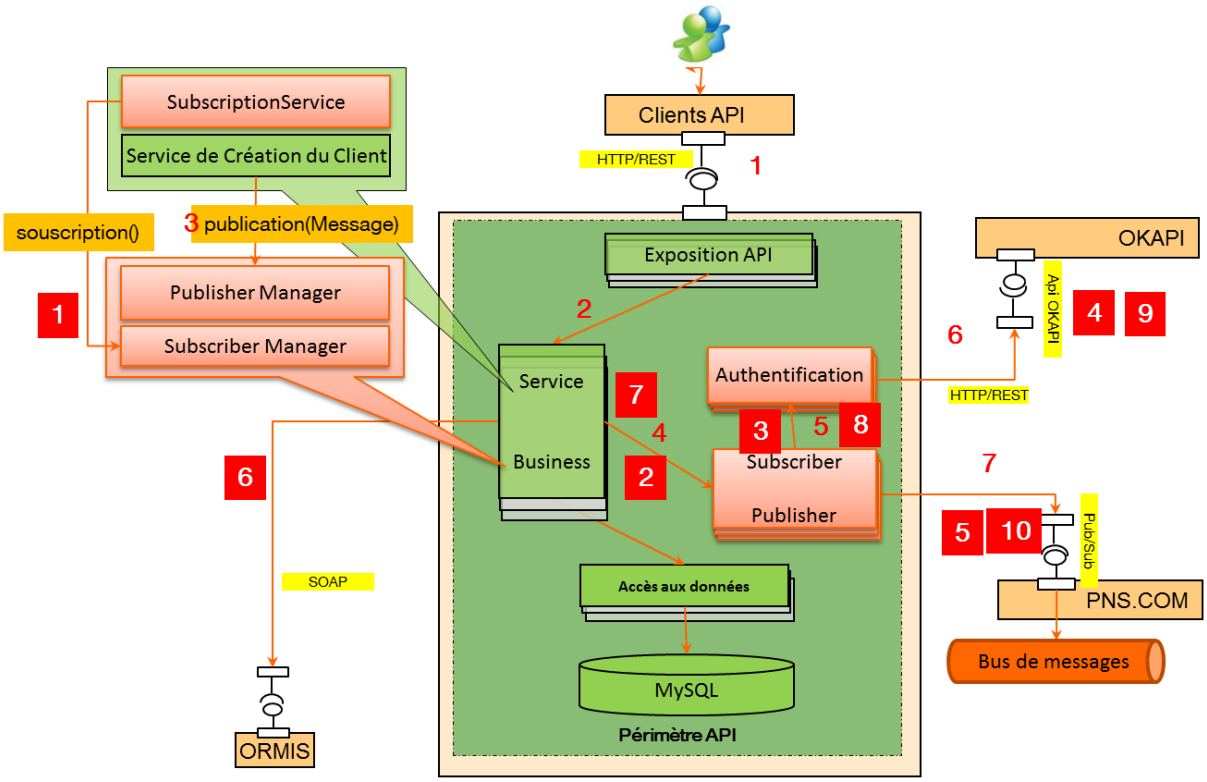

Figure 10 : Schéma du Prototype GDFA avec le bus de message

Les carrés orange représentent les nouvelles classes ou modules développés. Les chiffres en rouge suivent le parcours du client lors de l'appel par un utilisateur de GDFA pour l'inscription d'un client en boutique dans la file d'attente ou pour l'appel du client suivant. Par exemple, pour l'inscription du client dans la file d'attente :

1. L'utilisateur appelle le service de l'API pour la création par la méthode « POST ».
2. Des vérifications sont faites sur l'objet transmit par l'API. Un mapping est réalisé entre la représentation par l'API et le « DTO ». Puis le DTO est transmis au service `serviceNewInStoreCustomer`.
3. La création du client implémenté dans le `serviceNewInStoreCustomer` est lancée. A la fin de la création, le service appelle le « manager » pour une publication.
4. Le « manager » de publication crée le `MessageDTO` qui sera envoyé dans le bus et lance le `Publisher`.
5. Le `Publisher` va d'abord vérifier la validité du « Token ». Si le Token n'a pas expiré, on passe à l'étape 7.
6. Le client d'OKAPI est appelé et envoie un « POST » pour demander un nouveau « Token » à l'API d'OKAPI.
7. Une fois l'obtention du « Token » valide, on publie le `MessageDTO` dans le Bus de message par une requête « POST ». Une fois la réponse de PNS.com reçu,on renvoie la réponse à l'utilisateur de GDFA.

Les chiffres en blanc dans le carré rouge, sont pour le chemin de l'envoi des messages à ORMIS :

1. Le service de souscription se lance et fait appel au manager.
2. Le manager lance le `Suscriber`.
3. Comme pour le `Publisher`, le `Suscriber` vérifie que le « Token » est valide.
4. Le client d'OKAPI est appelé et envoie un « POST » pour demander un nouveau « Token » à l'API d'OKAPI.
5. On fait un « GET » pour récupérer les messages dans le bus de messages.
6. Le « manager » convertit la liste de messages en `MessageDTO` puis le service de souscription les envoie à ORMIS.
7. Si les `MessageDTO` sont tous envoyés, le « manager » rappelle le `Suscriber` pour faire l'acquittement.
8. Encore une fois, il y a une vérification de validité du « Token ».
9. Si besoin, on récupère un nouveau « Token ».
10. On envoie une requête « POST » pour acquitter les messages.

i. Gestion du Token

Pour s'authentifier aux Bus de « PNS.com », il est nécessaire de générer un « Token ».

Pour s'adresser à l'API du Token, il faut pouvoir gérer les certificats associés à la page.

En Java, la fonction `getDefaultTrustedCertificates` de la classe « CertificatManager » permet de générer un « KeyStore ». La fonction va ajouter les certificats qu'on récupère du dossier ressources par la méthode `setCertificateEntry` dans le « KeyStore ».

```
final KeyStore pnsKeyStore = CertificatManager.getDefaultTrustedCertificates();
```

Le « SslConfigurator » permet de configurer les instances « SSLContext » par la méthode `createSSLContext`. Le « KeyStore » est ajouté dans la configuration de SSL.

La classe « SSLContext » est initialisée avec un ensemble facultatif de gestionnaires de KeyStore et un protocole de sécurité. Le protocole ajouté est TLSv1.2. Il permet de sécuriser l'échange entre le client et le serveur en authentifiant le serveur, permettant la confidentialité et l'intégrité des données échangées et optionnellement en authentifiant le client.

```
SSLContext sslContext = SslConfigurator.newInstance()
                .keyStore(trustStore)
                .trustStore(trustStore)
                .securityProtocol(TLS12_SCURITY_PROTOCOL)
                .createSSLContext();
```

Le point d'entrée principal de l'API est un « javax.ws.rs.client.ClientBuilder » utilisé pour amorcer les instances « javax.ws.rs.client.Client » : objets configurables et lourds qui gèrent l'infrastructure de communication sous-jacente et servent d'objets racines pour accéder à

toute ressource Web. L'exemple suivant illustre l'amorçage et la configuration d'une instance Client avec l'ajout du contexte SSL défini :

```
Client client=ClientBuilder.newBuilder().sslContext(sslContext).build();
```

Comme l'instance « Client » est lourde, il est conseillé de ne construire qu'un petit nombre d'instances « Client » dans l'application. Les instances du client doivent être correctement fermées avant d'être disposées pour éviter des fuites de ressources.

À partir du « Client », une ressource décrite par l'interface peut être créée grâce à la fabrique de ressource client : « WebResourceFactory ». Cette fabrique va s'appuyer sur le « pattern Proxy » pour instancier une ressource de type « OauthApi » qui est l'interface générée par l'outil de génération de code et une cible de « web resource ».

```
OkapiOauthApi api = WebResourceFactory.
      newResource(OkapiOauthApi.class, client.target(URL_TOKEN))
```

On obtient un objet « OkapiOauthApi » directement utilisable. L'instance « api » permet d'utiliser toutes les méthodes décrites dans l'interface.

ii. Client de l'API ZBus

À partir de l'interface générée, la ressource client du « ZBus » est instanciée.

Pour l'utilisation de l'API, il faut prendre en compte les « Headers » de la requête. Il y a deux façons de les prendre en compte : soit en précisant la liste des headers dans la méthode `newResource()` de la classe `WebResourceFactory`, soit par l'utilisation de filtre.

Le filtre est représenté par une classe implémentant soit l'interface `ClientRequestFilter` avec l'annotation `@Priority(Priorities.HEADER_DECORATOR)`, soit l'interface `ClientResponseFilter` soit les deux. L'interface `ClientRequestFilter` implémente la méthode `filter()` qui est appelée avant l'envoi de la requête pour ajouter de l'information comme par exemple pour ajouter un header. L'annotation permet d'indiquer l'utilité du filtre. L'interface `ClientResponseFilter` implémente la méthode `filter()` qui est appelée au moment de la réponse. Par exemple, cette interface est utilisée pour récupérer l'identifiant d'acquittement des messages qui sont dans les headers de la réponse. Cela permet notamment d'utiliser directement l'interface typée de l'API ZBus pour avoir une réponse déjà structurée sous forme de liste.

L'annotation `@ComponentScan` qui est dans la classe de démarrage de l'application, permet de balayer la liste des composants avec l'annotation `@Configuration`. En fonction de la classe annotée par `@Configuration`, certaines actions sont effectuées. En plus, l'annotation `@Bean` ajoutée à la méthode, permet d'enregistrer l'objet dans le contexte de Spring et ainsi de limiter le nombre d'instanciations du client de l'API. La ressource de l'API pourra être réutilisée par la suite lors des appels à la ressource.

iii. Service de souscription

Le service de souscription doit pouvoir être lancé au démarrage de l'application du bus. Il lance un nouveau pool de threads qui lit les messages dans le bus et les acquitte après le traitement par le service d'envoi des SMS à ORMIS fourni par Orange.

La classe `ScheduledExecutorService` est un `ExecutorService` qui peut planifier l'exécution de tâches après un délai d'attente ou l'exécution répétée de tâches avec un intervalle de temps fixe entre chaque exécution. Les tâches sont exécutées de manière asynchrone par un des threads du pool.

Pour obtenir une instance de type `ScheduledExecutorService`, il est possible :

- d'invoquer un des constructeurs de la classe `ScheduledThreadPoolExecutor`
- d'utiliser des méthodes de la classe `Executors` qui sont des fabriques pour obtenir une instance de type `ScheduledThreadPoolExecutor : newSingleThreadExecutor` qui permet de créer un seul thread pour l'exécution ou `newFixedThreadPool` qui permet de créer un pool de threads dont la taille est fournie en paramètre
- Les tâches sont exécutées de manière asynchrone par un thread de travail, et non par le thread passant la tâche à `ScheduledExecutorService.`

Une fois la création du `ScheduledExecutorService` avec le nombre de thread voulu, on peut démarrer la tâche par l'une de ses méthodes:

- `schedule (Callable task, long delay, TimeUnit timeunit)` permet d'exécuter le `Callable` après le délai donné. `Callable` est une interface implémentant `call` qui renvoie un résultat et peut jeter une exception
- `schedule (Runnable task, long delay, TimeUnit timeunit)` est la même méthode que précédemment mais avec un `Runnable`. Le `Runnable` implémente une méthode `run` qui ne renvoie pas de valeur.
- `scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)` planifie une tâche à exécuter périodiquement. La tâche est exécutée la première fois après l' « initialDelay », puis chaque fois que la période expire. Si une exécution de la tâche donnée lance une exception, la tâche n'est plus exécutée. Si aucune exception n'est lancée, la tâche continuera d'être exécutée jusqu'à ce que le service `ScheduledExecutorService` soit arrêté.
- `scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)` fonctionne comme précédemment sauf que la période est interprétée différemment. Dans la méthode `scheduleAtFixedRate (),` la période est interprétée comme un délai entre le début de l'exécution précédente, jusqu'au début de la prochaine exécution. Dans cette méthode, cependant, la période est interprétée comme le délai entre la fin de l'exécution précédente, jusqu'au début de la prochaine. Le délai est donc entre les exécutions terminées, pas entre le début des exécutions.

Le `ScheduledExecutorService` de la souscription utilise la méthode `scheduleAtFixedRate` car la tache doit s'exécuter périodiquement. La tâche du « Runnable » est la méthode du manager de souscription qui va lire dans le bus, convertir la réponse en « MessageDTO », les envoyer aux services d'ORMIS puis acquitter le paquet de messages si tout se passe bien.

L'`ApplicationListener` est une interface qui va exécuter la méthode `onApplicationEvent` lors de l'événement choisi en paramètre. Ici, on implémente deux `ApplicationListener` : le premier sera pour lancer la tâche et le deuxième pour l'arrêter. Pour le lancement, `ApplicationListener` prend en paramètre `ApplicationReadyEvent` qui va permettre le démarrage automatique de la tâche dès que l'application est prête et a fini de s'initialiser. Pour l'arrêt de la tâche, il prend en paramètre `ContextClosedEvent` qui se déclenche dès le commencement de la fermeture de l'application.

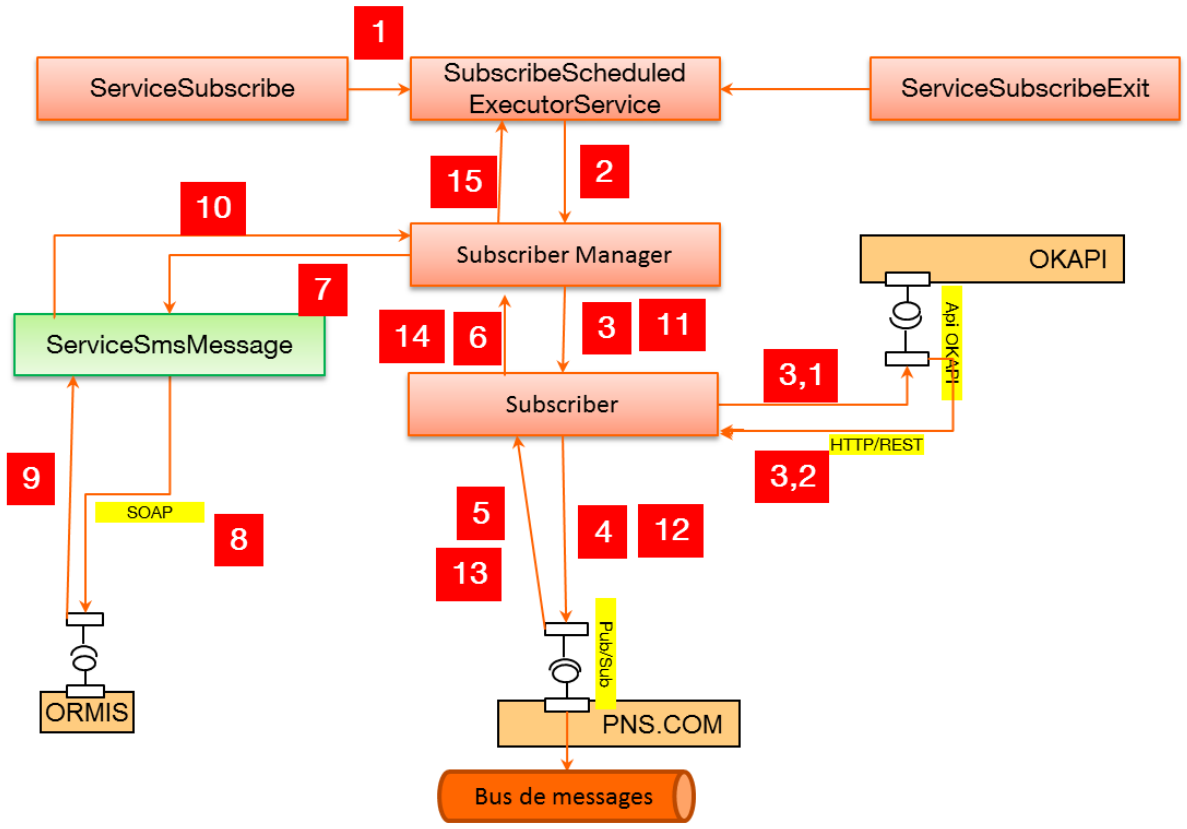

Figure 11 : Schéma simplifié du fonctionnement de la souscription

1. Le « ServiceSubscribe » lance le « ScheduledExecutorService » de la classe « SubscribeScheduledExecutorService », dès que GDFA est prête à démarrer.
2. Le « ScheduledExecutor » exécute la souscription du « SubscriberManager ».
3. Le « manager » appelle le « Subscriber » qui vérifie s'il y a besoin d'un nouveau « Token ».
   3.1. Si besoin, on envoie une requête à OKAPI.
   3.2. On reçoit le nouveau « Token ».
4. On interroge le bus de messages avec une requête « GET ».

5. PNS.com renvoie une réponse.
6. La réponse reçue est renvoyé au « manager ». Si le statut de la réponse est 204, cela veut dire que le bus est vide et plus aucune action n'est faite. Si le statut est 200, la réponse n'est pas vide et les MessageDTO sont convertis.
7. Le « Manager » envoie un par un les MessageDTO au « ServiceSmsMessage » .
8. Le « ServiceSmsMessage » envoie les SMS à envoyer à ORMIS.
9. ORMIS indique si l'opération s'est bien déroulée.
10. Le « ServiceSmsMessage » indique au « manager » s'il n'y a pas eu d'erreur
11. En cas d'erreur, le manager demande un « nack » (non acquittement des messages), sinon un « ack » (acquittement des messages).
12. Le « Subscriber » envoie la requête à PNS.com.
13. L'API indique si l'opération s'est terminée avec succès.
14. Le « Subscriber » renvoie la réponse au « manager ».
15. Le « manager » indique la fin de la souscription au « ScheduledExecutorService » qui pourra lancer une nouvelle souscription.

III. Prototypes de la persistance des données

Ces prototypes s'intéressent à trois APIs pour persister les données dans une base de données NoSQL :

- l'API « GyGraph » de PNS.com pour une base de données orientée graphe
- l'API « ValKey » de PNS.com pour une base de données orientée clé colonne
- l'API « ElastikSearch » pour une base de données orientée document

Les prototypes couvrent trois cas d'utilisation :

- la création du client dans la file d'attente
- l'appel du client suivant
- la suppression du client dans la file d'attente

Pour couvrir ces cas d'utilisation, un périmètre restreint est défini. Il va prendre en compte les entités correspondantes : aux clients en boutique, aux trois types de SMS disponibles, aux SMS demandés par les clients, à un utilisateur qui correspond à l'employé de la boutique et à une boutique.

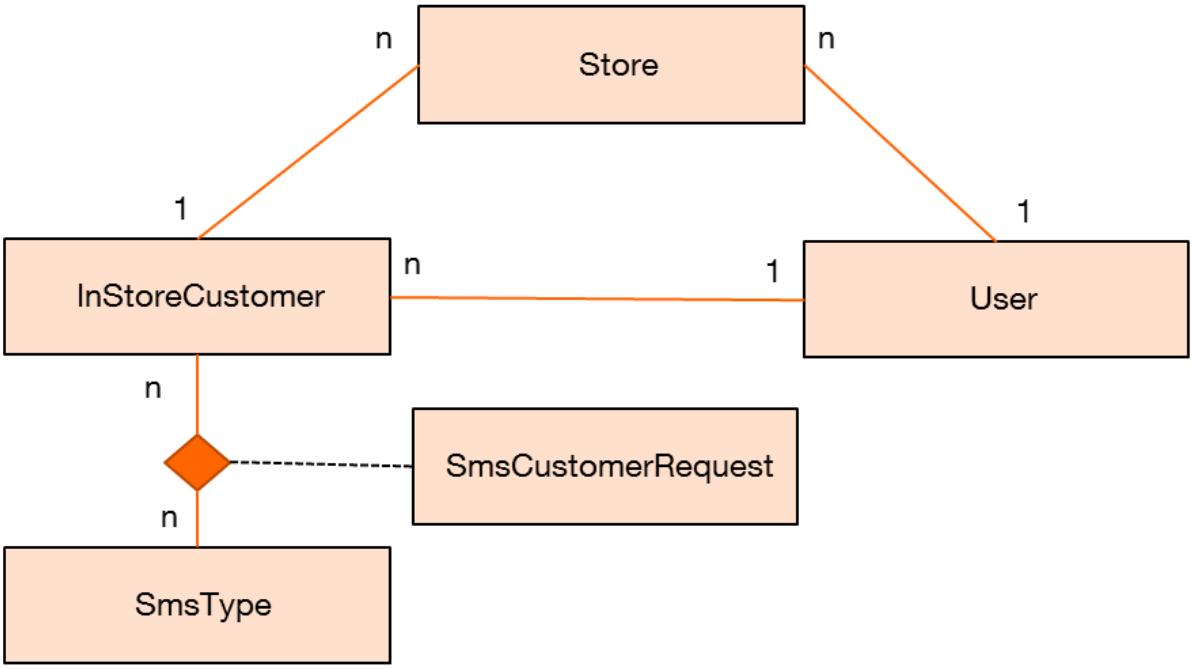

Figure 12 : Périmètre du prototype

Les rectangles beiges représentent les entités et les traits oranges les relations entre les entités. La cardinalité sur la figure ci-dessus se lit comme : « un « store » à plusieurs « users » ». L'entité « SmsCustomerRequest » est une table de lien.

1) Base de données orientée graphe

a) Présentation d'une base de données orientée graphe

La base orientée graphe est un type de base de données NoSQL qui utilise la théorie des graphes pour stocker, mapper et interroger des relations.

i. Théorie des graphes

La théorie des graphes a eu une grande utilité et une grande pertinence dans de nombreux problèmes de domaines variés. Les algorithmes de théorie des graphes les plus utilisés incluent différents types de calculs du plus court chemin, les chemins géodésiques…

Le problème est modélisé sous forme d'un graphe. La base se compose d'un ensemble de nœuds et d'arêtes. Chaque nœud représente une entité par exemple une boutique, et chaque arête, une connexion ou une relation entre deux nœuds.

Les nœuds sont définis par un identifiant unique, un ensemble d'arêtes sortantes et/ou entrantes, ainsi qu'un ensemble de propriétés exprimées sous la forme de paires clé/valeur. Les arêtes se définissent par un identifiant unique, un nœud de départ et/ou un nœud d'arrivée, ainsi qu'un ensemble de propriétés.

ii. Modèle pour la base orientée graphe.

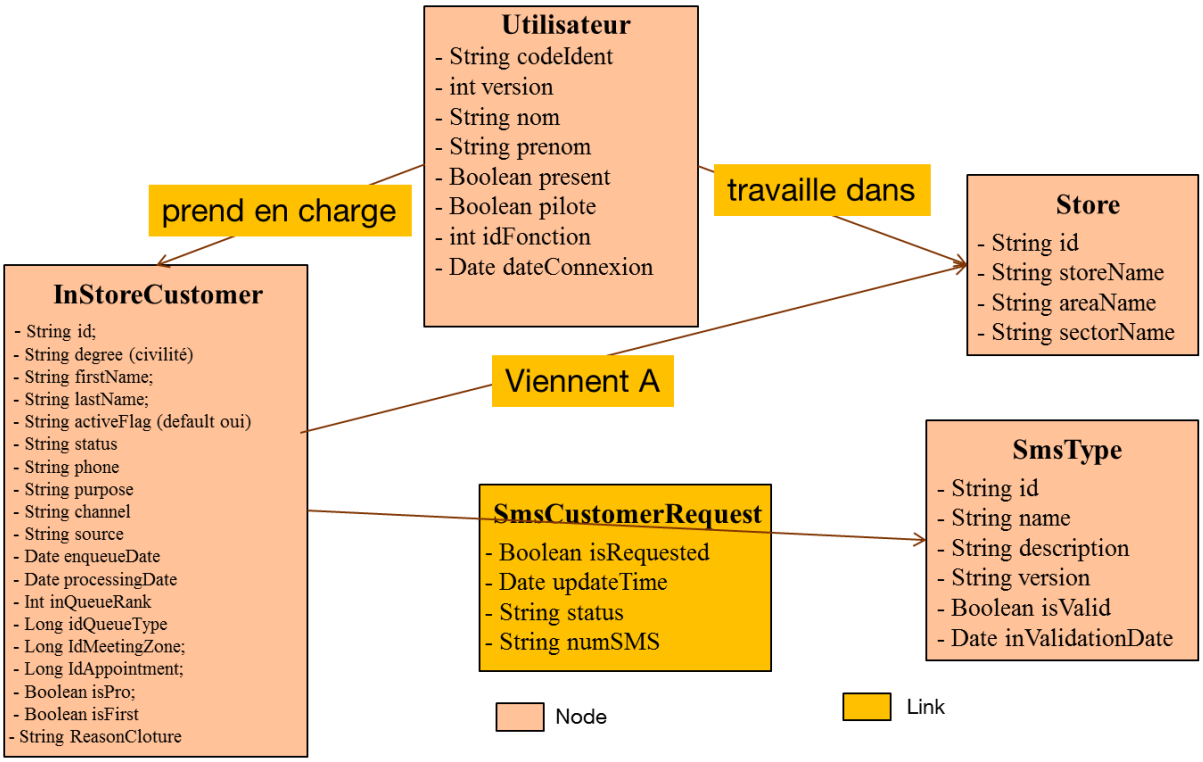

Figure 13 : Modèle de la Base de données orientée graphe

Le modèle pour le prototype définit les différents types de nœuds et arêtes comme sur la figure ci-dessus. Les nœuds sont représentés par des rectangles roses qui contiennent un

type et plusieurs propriétés. Les arêtes sont représentées par des flèches avec un rectangle orange qui contient le type et zéro ou plusieurs propriétés.

b) Présentation de l'API

L'objectif de « GyGraph » est de fournir un service de base de données orientée graphe interrogé via une API « RESTful ».

i. Méthodes CRUD sur les nœuds

La requête « PUT …/nodes » permet de créer un nœud avec des propriétés. L'appel vérifie l'existence de la donnée avant de l'ajouter. Si l'élément existe déjà, le code de statut 412 est renvoyé à l'utilisateur.

La requête « GET …/nodes/{id} » est utilisée pour obtenir un nœud aves ses propriétés et ses voisins à partir de son identifiant qui est composé : {type du nœud}:{id du nœud}. Sans paramètres sur la requête, la réponse se compose seulement du nœud simple avec son type et son nom. Il vérifie l'existence sur le nœud. Certains filtres optionnels sont autorisés dans les paramètres de requête pour étendre les données du nœud récupéré : il pourrait s'agir de ses propriétés, de ceux de ses arêtes et / ou des nœuds voisins. Tous les paramètres de requête peuvent être combinés. La réponse est renvoyée au format JSON, par exemple :

```
{
    "name": "{node_name}",
    "type": "{node_type}",
    "properties": {
        "{key1}": "{value1}",
        ...
    }
    "links": [
     {
        "type": "{link_type}",
        "direction": "{link_direction}",
        "properties": {
            "{key1}": "{value1}",
            ...
        }
        "otherNode": {
            "name": "{otherNode_name}",
            "type": "{otherNode_type}",
            "properties": {
                "{key1}": "{value1}",
                ...
            }
      },
      ...
    ]
}
```

La requête « PATCH …/nodes/{id} » permet d'ajouter, remplacer ou de supprimer une ou des propriétés d'un nœud à partir de son identifiant.

La requête « DELETE …/nodes/{id} » supprime le nœud avec toutes ses propriétés. Lorsqu'un nœud est supprimé, toutes ses arêtes sont également supprimées afin de conserver la cohérence dans la base de données.

ii. Méthodes CRUD sur les arcs

La requête « PUT …/links » permet de créer l'arête entre deux nœuds. De même que pour le nœud, il y a une vérification de l'existence de la donnée. Le paramètre "withNodes" est booléen pour définir si les nœuds doivent être créés dans le processus.

La requête « GET …/links/{id} » récupère l'arête avec les nœuds à partir de son identifiant. L'identifiant est composé : {type du nœud source}:{id du nœud}:{type de l'arc}:{type du nœud destination}:{id du nœud}. La réponse est renvoyée au format JSON :

```
    {
        "type": "type1",
        "source": {
            "name" : "{src_name}",
            "type" : "{src_type}"
        },
        "destination": {
            "name" : "{dst_name}",
            "type" : "{dst_type}"
        },
        "properties": {
            "p1": "val1",
            "p2": "val2",
            "p3": "val3"
        }
    }
```

La requête « PATCH …/links/{id} » permet d'ajouter, de remplacer ou de supprimer une ou des propriétés d'une arête à partir de son identifiant.

La requête « DELETE …/links/{id} » permet de supprimer l'arête avec toutes ses propriétés. Lorsqu'une arête est effacée, les nœuds source et destination ne sont pas supprimés : ils peuvent continuer à exister en tant que nœuds, même s'ils ne sont liés à rien d'autre.

iii. Récupérer tous les voisins d'un nœud ou d'une arête

Avec la requête POST …/nodes, on obtient plusieurs nœuds avec leurs propriétés et leurs voisins. La requête fonctionne exactement comme celle du GET. Il suffit d'ajouter la possibilité de demander plusieurs nœuds source dans une même requête, grâce à un corps JSON au lieu d'un identifiant unique dans l'URL.

La gestion des erreurs est la même que pour l'API du ZBus.

c) Intégration dans l'application

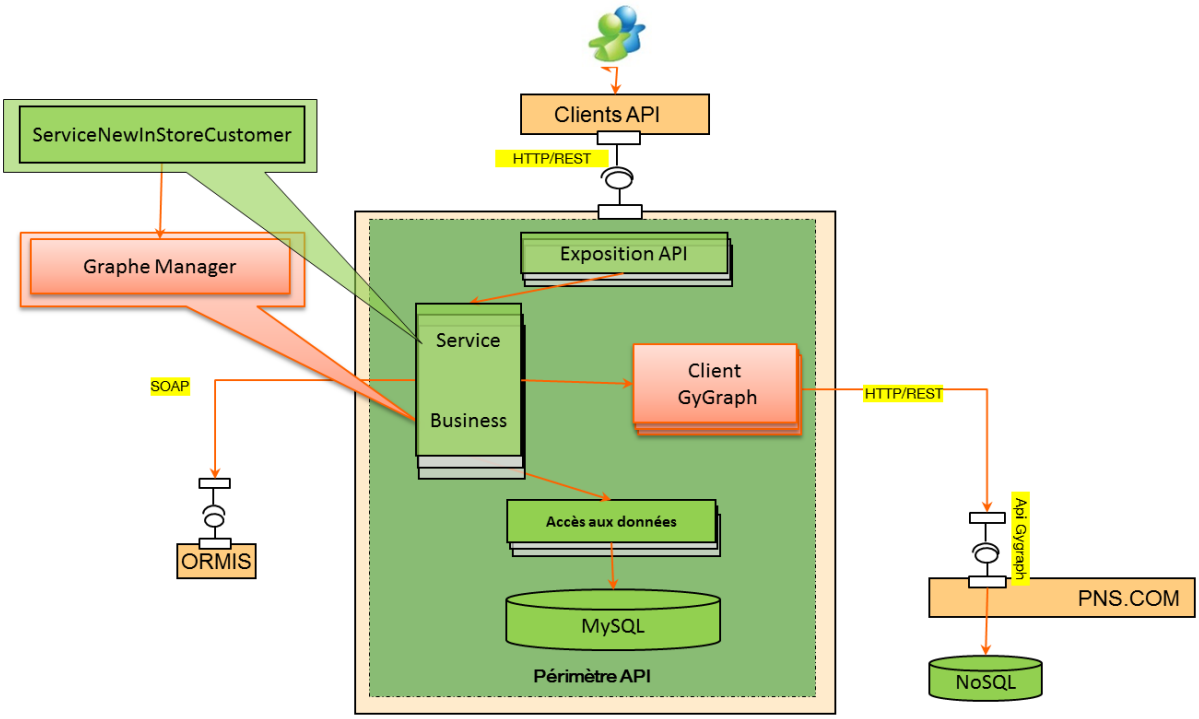

Figure 14 : Schéma du Prototype GDFA avec la base orientée graphe

En rouge, on retrouve les nouveaux objets et en vert, les objets existants. Lors d'une demande du client de l'API, le « ServiceNewStoreCustomer » va faire appel au « GraphManager » pour modifier la base de données NoSQL. Le « GraphManager » utilise le « ClientGyGraph » pour exécuter les requêtes HTTP sur l'API.

i. Nouveau Client pour l'API GyGraph

Pour intégrer l'API, la génération automatique de code a été utilisée pour fournir une interface qui est utilisée pour instancier un client « JAX-RS » comme pour le client du « ZBus ». Le client de l'API est un Bean qui est instancié au démarrage de l'application de GDFA.

En s'appuyant sur le modèle défini, les « POJO » (Plain Old Java Objects) sont développés pour représenter les données stockées dans la base NoSQL. Les classes définies sont les objets représentants les types de nœuds et les types d'arêtes qui possèdent des propriétés.

Pour communiquer entre les « POJO » du graphe et les « DTO » qui sont dans l'application, une classe de « mapping » est implémentée. Le « mapping » se fait entre le « `InStoreCustomerDTO` » et « `InStoreCustomerGraph` » en utilisant les « getters » et les « setters » des classes.

Un deuxième « mapping » est nécessaire pour communiquer entre les « POJO » du graphe et le format des données envoyées par l'API. En effet, il faut pouvoir passer de l'entité du graphe au nœud ou arête. Le nœud et l'arête sont définis dans le contrat de l'interface et

sont des classes générées en même temps que l'interface. Les attributs des classes de l'entité du graphe doivent être transformés en une `Map<String,String>`. Le premier « String » représente le nom de l'attribut et le deuxième sa valeur. Pour faire un « mapping » rapide, ObjectMapper est utilisé.

```
ObjectMapper mapper = new ObjectMapper();
Map<String, String> properties = mapper.convertValue(objet, new
TypeReference<Map<String, String>>() {});
```

`ObjectMapper` convertie l'objet en paramètre en `Map` en s'appuyant sur les « getters ». Il va aussi convertir la `Map` reçue dans la réponse en Objet, par exemple en `InStoreCustomerGraph` :

```
objet = mapper.convertValue(map, InStoreCustomerGraph.class);
```

Pour faire la conversion en « POJO », le `mapper` s'appuie sur le constructeur par défaut et les « setters » de la classe.

ii. Intégration aux services logiques de GDFA

Dans le module logique, un « manager » nommé `GraphManager` est développé pour gérer les appels aux clients de l'API. Le `GraphManager` est ajouté en attribut dans le `serviceNewInStoreCustomer` qui est le service contenant les fonctions représentant le parcours du client, le service d'envoi des SMS et dans le calcul du rang.

Le « manager » va contenir les fonctions utiles pour répondre aux besoins des cas d'utilisation.

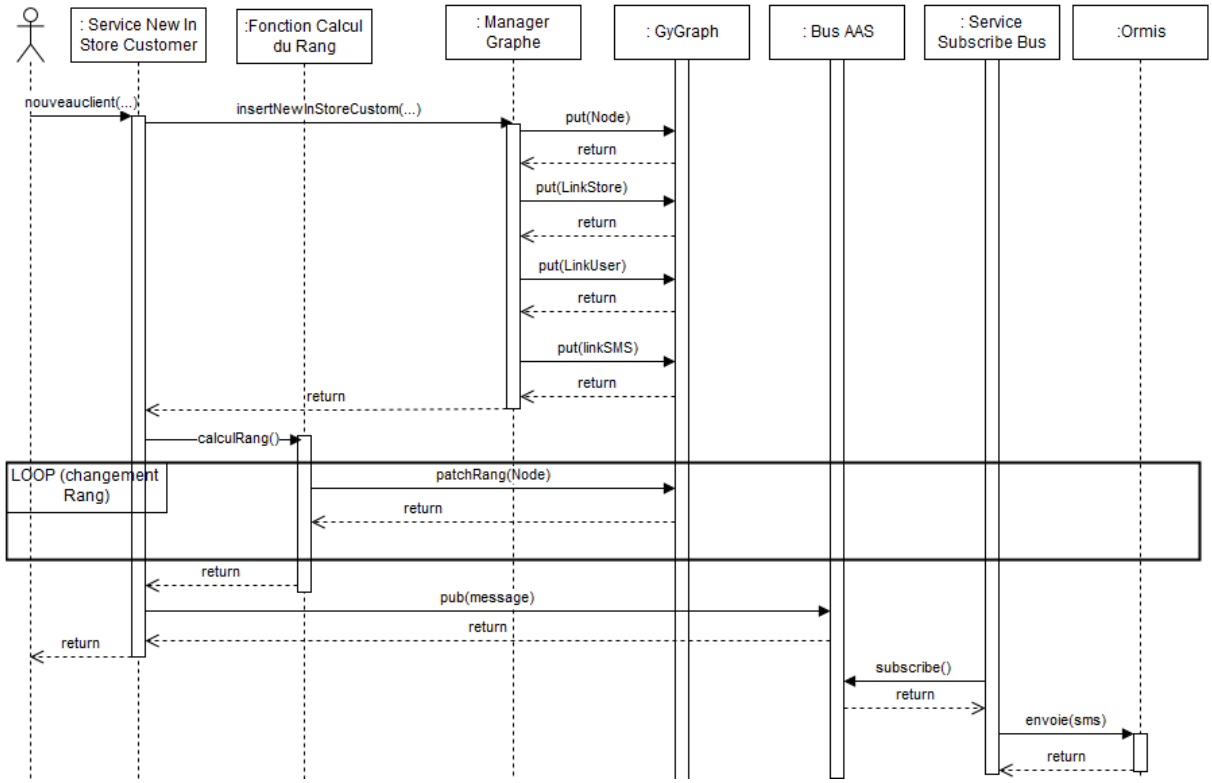

Figure 15 : Diagramme de séquence de création d'un client

Pour commencer dans le cas de la création d'un client en boutique, il y a beaucoup de création à faire dans la base, il faut créer le nouveau nœud du client et faire les liens avec les autres nœuds : pour les SMS, pour l'utilisateur et la boutique. Quand le service de création passe dans la fonction du calcul du rang, on appelle un traitement qui met à jour dans la base le rang et le nombre de dépassements des clients.

Le service s'occupant de l'envoi des SMS doit mettre à jour la base de données pour chaque envoi.

Le « manager » est utilisé dans le service de création, d'envoi des SMS, et du calcul du rang.

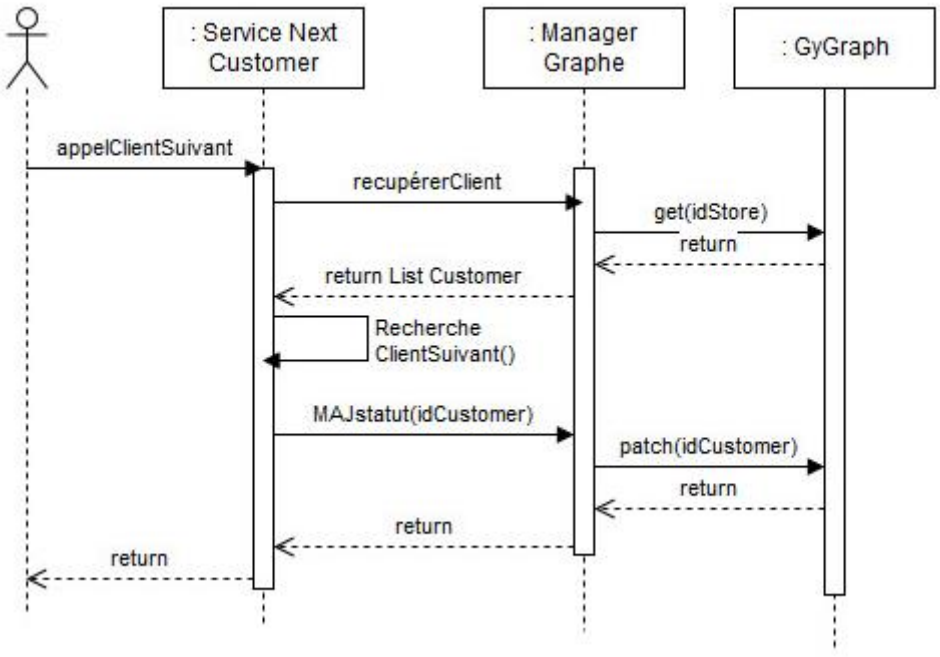

Figure 16 : Diagramme de séquences d'Appel du client suivant en boutique

Ensuite, pour la recherche du client suivant, le manager contient une fonction permettant de récupérer le client avec le rang le plus petit. Comme l'API ne permet pas la recherche sur un attribut, la fonction récupère tous les clients d'une boutique. On interroge le nœud de la boutique en demandant tous les nœuds voisins ayant une relation de type « ViennentA ». Ensuite, on récupère une liste de nœuds où on peut rechercher le client suivant.

```
Optional<LinkResponse> lienClient = nodeVendeur.getLinks().stream().
        filter(x ->GdfaConstants.STATUS_INITIALISEE.
                equals(x.getOtherNode().getProperties().get("status"))).
        min(comparator);
```

La recherche utilise un « stream ». Celui-ci renvoie un flux séquentiel avec cette collection comme source. Il permet ensuite d'utiliser d'autres fonctions. Ici par exemple, on filtre avec les clients qui n'ont toujours pas été servis puis on récupère le client avec le plus petit rang.

```
Comparator<LinkResponse> comparator =  new Comparator<LinkResponse>(){
```

```
        @Override
        public int compare(LinkResponse f1, LinkResponse f2){
          return
           f1.getOtherNode().getProperties().get("inQueueRank").compareTo(
            f2.getOtherNode().getProperties().get("inQueueRank")
          );
          }
};
```

Les « `Comparator` » peuvent passer à un ordre de tri (comme « `Collections.sort` » ou « `Arrays.sort` ») pour permettre un contrôle précis sur l'ordre de tri. Les comparateurs peuvent également être utilisés pour contrôler l'ordre de certaines structures de données ou pour commander des collections d'objets qui n'ont pas de commande naturelle. Ici, le « `Comparator` » défini permet de comparer les deux objets et de les trier selon leur rang.

Après avoir récupéré le client, on modifie les données dans la base.

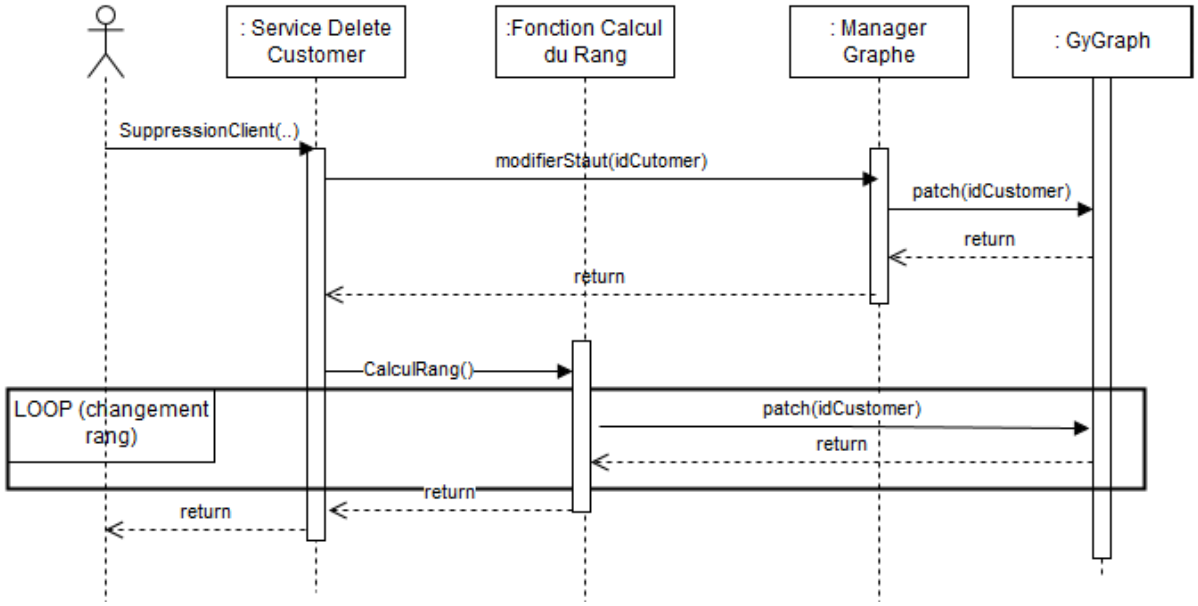

Figure 17 : Diagramme de séquence de suppression d'un client

Une fois que la prestation est réalisée pour le client, les données du client ne sont pas supprimées mais elles sont mises à jour. Le statut passe à clôturé, on spécifie la raison de clôture, le rang du client devient nul et sa fiche devient inactive.

```
properties.put("status", GdfaConstants.STATUS_CLOTUREE.toString());
properties.put("idMotifCloture", String.valueOf(idClosureReason));
properties.put("activeFlag",null);
properties.put("inQueueRank",null);
```

Puis, la fonction de calcul du rang démarre pour mettre à jour les rangs des autres clients dans la file d'attente. Dans ce cas, les clients restants dans la file d'attente voient leurs rangs baisser : un client au rang 2 passe au rang 1.

On peut voir sur la figure ci-dessous, le résultat dans le graphe d'un client ayant fini sa prestation en boutique.

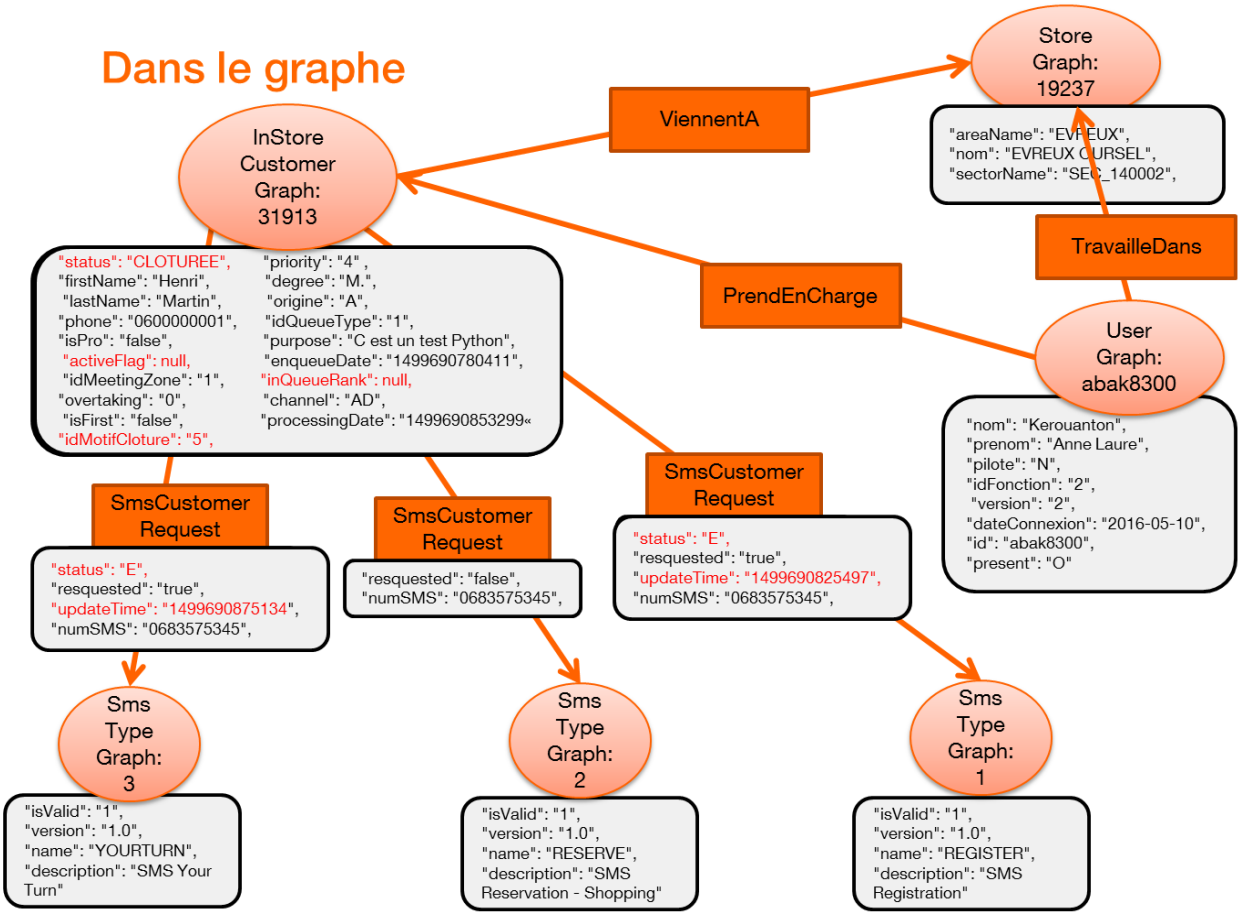

Figure 18 : Exemple d'un client dans une base orientée graphe

2) Base de données orientée clé/colonnes

a) Présentation d'une base orientée clé/colonnes

Une base de données orientée colonnes est un système de gestion de base de données qui enregistre les données dans des colonnes, plutôt que dans des lignes. L'objectif d'une base de données orientée colonnes est d'optimiser l'écriture et la lecture de données vers et depuis le disque, afin d'accélérer le renvoi des résultats d'une requête.

Dans une base de données orientée colonnes, toutes les valeurs de la colonne 1 sont regroupées physiquement, suivies par toutes les valeurs de la colonne 2, etc. Les données sont consignées par ordre d'enregistrement, de sorte que la 100e entrée de la colonne 1 et la 100e entrée de la colonne 2 appartiennent au même enregistrement d'entrée.

Ainsi, les éléments de données – par exemple, les noms des clients – sont accessibles ensemble dans une seule et même colonne, plutôt que chacun dans une ligne.

1ère clé	2ème clé	Table	Colonne	Valeur
idStore (int)		Store	« Information »	storeName (String), areaName (String), sectorName (String)
	idUser (String)	User	« Information »	version (String), firstName (String), lastName (String), isPresent (boolean), isPilote (boolean), idFonction (int), dateConnexion (Date)
	idInStore Customer (String)	InStoreCustomer	« Identification »	degree (String), firstName (String), lastName (String), activeFlag (boolean), phone (String), purpose (String), channel (String), source (String), enqueueDate (Date), idQueueType (Long), IdMeetingZone (Long), IdAppointment (Long), isPro (boolean), isFirst (boolean), priority (int)
			« Rank »	inQueueRank (int), overtaking (int)
			« Status »	status (String), processingDate (Date), ReasonCloture (Long)
			« User »	idUser (String)
			« SmsCustomerRequest Inscription»	idSmsType (int), isRequested (boolean), updateTime (Date), status (String), numSMS (String)
			« SmsCustomerRequest YouTurn»	idSmsType (int), isRequested (boolean), updateTime (Date), status (String), numSMS (String)
			« SmsCustomerRequest Shopping»	idSmsType (int), isRequested (boolean), updateTime (Date), status (String), numSMS (String)

Figure 19 : Modèle de la Base de données orientée clé/colonnes

Le modèle pour le prototype se construit en fonction des requêtes qui sont à exécuter. Le découpage en colonne des tables s'est fait en fonction des valeurs à modifier. En effet, chaque modification d'une valeur de colonne entraine l'écrasement de l'ancienne, par exemple, lors du calcul de rang, le rang et le nombre de dépassement du client sont écrasés mais on n'a pas forcement toutes les informations relatives au client donc une colonne lui est dédiée.

Les relations de type (1-n), par exemple entre un client et une boutique ou entre une boutique et l'employé en boutique (user), sont représentées par une clé composée. L'ordre de la clé est aussi important, par exemple la clé de la boutique est en premier car cela permet de récupérer plus facilement l'ensemble des clients d'une boutique.

La relation de type (n-n) entre un client et les types de SMS est représentée par une colonne dédiée pour chaque type de sms.

b) Présentation de l'API

L'API de PNS.com ValKey fournit un accès en lecture et en écriture aux données stockées. Cette donnée est envoyée avec le format ci-dessous :

```
{
  "users": [
    {
      "credentials": {
        "{cred_type}": {cred_val}
      },
      "fields": [
        "{field_name}"
      ],
      "options": {
        "{option_name}": {option_val}
      }
    }
  ]
}
```

Le « users » est l'élément qui contient toutes les données relatives à un utilisateur de l'API.

Le « fields » est identifié par son nom et contient une valeur. Les noms sont uniques pour chaque utilisateur et la valeur peut ou non exister. Il y a un contrat d'interface pour désigner les « fields » auxquelles l'utilisateur a accès. Par exemple, le « fields » de corps pour écrire toutes les colonnes pour deux clients en boutique :

```
    "fields": [
      "StoreCustomer" : {
        "row1": {
          "storecustomer-identification" : "identification1",
          "storecustomer-rank"           : "rank1",
          "storecustomer-smsinscription" : "smsinscription1",
          "storecustomer-smsshopping"    : "smsshopping1",
          "storecustomer-smsyouturn"     : "smsyouturn1",
          "storecustomer-status"         : "status1",
          "storecustomer-user"           : "user1"
        },
        "row2": {
          "storecustomer-identification" : "identification2",
          "storecustomer-rank"           : "rank2",
          "storecustomer-smsinscription" : "smsinscription2",
          "storecustomer-smsshopping"    : "smsshopping2",
          "storecustomer-smsyouturn"     : "smsyouturn2",
          "storecustomer-status"         : "status2",
          "storecustomer-user"           : "user2"
        }
```

```
        }
    }
```

Le « credentials » est l'identifiant unique d'un utilisateur. Le « user » peut être identifié par un ou plusieurs « credentials ». Par exemple, pour le prototype, le « credentials » sera l'identifiant de la boutique (« idStore »).

Les « options » indiquent les options relatives aux données par exemple : la date d'écriture de la donnée, la durée de vie.

Les requêtes possibles sont :

- PSOT pour récupérer des informations avec le body décrivant le type de données que l'on veut récupérer.
- GET pour récupérer une colonne à partir de la clé.
- GET pour récupérer l'ensemble des colonnes à partir d'une clé unique.
- PATCH pour ajouter ou modifier une ou plusieurs colonnes.

c) Intégration dans l'application

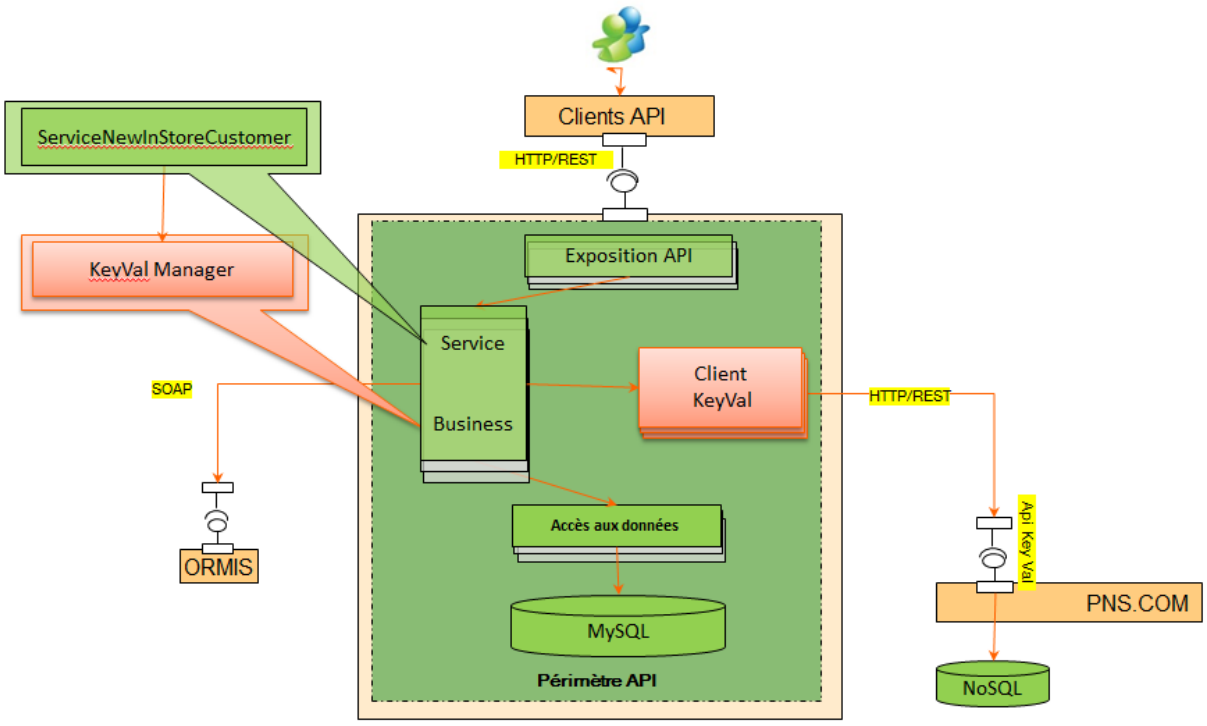

Figure 20 : Schéma du Prototype GDFA avec la base orientée clé/colonnes

En rouge, on retrouve les nouveaux éléments : « KeyValManager » et le connecteur avec l'API « ValKey ». Le connecteur permet l'accès aux données stockées dans une base NoSQL. Le « manager » contient les fonctions faisant le lien entre les services et le connecteur.

On utilise la même méthode pour générer un client qui permet la connexion et faire l'appel à l'API.

Des nouveaux « POJO » sont développés pour suivre le modèle défini. La représentation des colonnes est vue au niveau objets comme une sous classe de la classe `InStoreCustomerKV.`

Il y a toujours un « mapping » entre les « POJO » du clé/colonnes et des « DTO » mais le « mapping » entre les « POJO » et le corps des messages envoyés et reçus est plus important que pour le graphe. En effet, la modélisation étant différente, le « mapping » sera différent. De plus, on peut envoyer plus d'une donnée à la fois. Exemple de body pour l'ajout d'un utilisateur dans la base :

```
{
    "users":[
        {
           "credentials":{
                  "generic":"19237"
        },
        "fields":{
            "StoreUser":{
                "abak8300":{
                    "storeuser-information":
                    {
                        "idStore":19237,
                        "idUser":"abak8300",
                        "version":2,
                        "firstName":"Anne Laure",
                        "lastName":"Kerouanton",
                        "idFonction":2,
                        "dateConnexion":1498833219486,
                        "pilote":false,
                        "present":true
                    }
                }
            }
        },
        "options":{
            "ttl":604800000,
            "timestamp":1498833219490
        }
    }]
}
```

Alors que pour le graphe, les données sont accessibles directement dans le champ propriétés ce qui permet de les récupérer plus facilement sans enchainer plusieurs méthodes getters :

```
{
    "name":"abak8300",
    "type":"UserGraph",
    "properties":{
        "idStore":19237,
        "idUser":"abak8300",
        "version":2,
        "firstName":"Anne Laure",
        "lastName":"Kerouanton",
```

```
        "idFonction":2,
        "dateConnexion":1498833219486,
        "pilote":false,
        "present":true
    }
}
```

Le manager nommé `ValKeyManager` possède les mêmes méthodes que celui du graphe mais avec des implémentations différentes. La plus grande différence au niveau du « manager » est pour la recherche du client suivant. Pour cette API, il est nécessaire d'envoyer une requête « POST » pour demander tous les clients de la boutique avec le corps du message :

```
{
  "users": [{
      "credentials": {
          "generic": "1923"
      },
      "options": { },
      "fields": [ "StoreCustomer" ]
  }]
}
```

On voit l'importance d'avoir mis en première clé le numéro de la boutique pour récupérer tous les clients. Ensuite, sur la liste de clients que l'on récupère, on utilise un « stream » avec un filtre et la fonction min avec un nouveau comparateur pour comparer les rangs de ses deux objets.

Les autres fonctions du manager restent semblables à celle du manager graphe et sont utilisés au même endroit dans le `ServiceNewInStoreCustomer`, service de SMS et calcul de rang.

A la fin de la prestation, on retrouve dans la base le client comme sur la figure ci-dessous. Le cercle rose représente la clé des colonnes et les rectangles gris représentent chacune une colonne avec sa valeur.

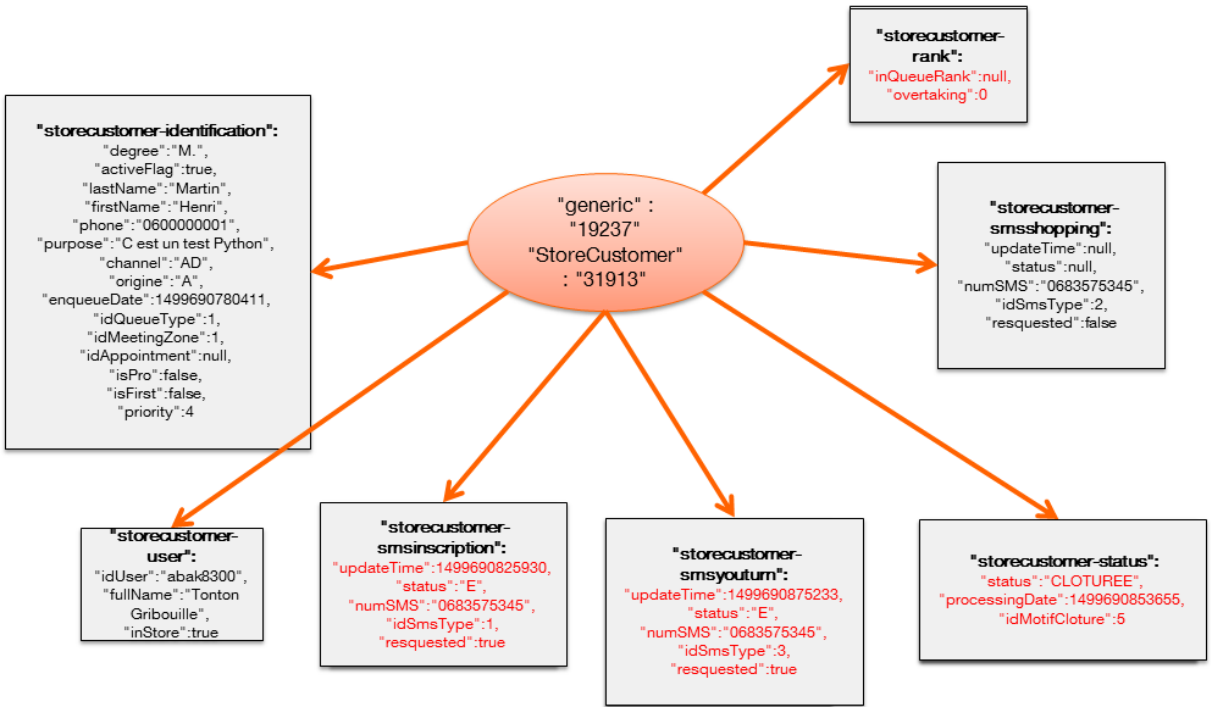

Figure 21 : Exemple d'un client dans une base orientée clé/colonnes

3) Base de données orientée document

a) Présentation d'une base orientée document

Une base de données orientée documents gère et stocke des données au niveau du document. Le document est la représentation d'une donnée. Elle ressemble à une base de données orientée clé/valeur, mais elle affiche une structure imposée aux données : l'ensemble des données est décrit dans une structure de document au format « JSON » ou « XML ». Cette structure permet d'interroger tous les composants du schéma défini.

Ce type de NoSQL ne demande aucun schéma fixe, un document peut contenir n'importe quel type d'information.

Type Document	Id (type)	Nom de la colonne	Type de la colonne
Store	**idStore** (int)	storeName	String
		areaName	String
		sectorName	String
User	**idUser** (String)	firstName	String
		lastName	String
		isPresent	boolean
		isPilote	boolean
		idFonction	int
		dateConnexion	Date
		idStore	int

Figure 22 : Modèle de la Base de données orientée document pour la boutique et l'utilisateur

Type Document	Id (type)	Nom de la colonne	Type de la colonne
InStore Customer	**idInStore Customer** (Long)	degree	String
		firstName	String
		lastName	String
		activeFlag	boolean
		phone	String
		purpose	String
		channel	String
		source	String
		enqueueDate	Date
		idQueueType	Long
		IdMeetingZone	Long
		IdAppointment	Long
		isPro	boolean
		isFirst	boolean
		priority	int
		inQueueRank	int
		overtaking	int
		status	String

Type Document	Id (type)	Nom de la colonne	Type de la colonne
InStore Customer	**idInStore Customer** (Long)	processingDate	Date
		ReasonCloture	Long
		idUser	String
		idStore	int
Sms Customer Request Inscription	**idInStore Customer** (Long)	isRequested	boolean
		updateTime	Date
		status	String
		numSMS	String
Sms Customer Request YouTurn	**idInStore Customer** (Long)	isRequested	boolean
		updateTime	Date
		status	String
		numSMS	String
Sms Customer Request Shopping	**idInStore Customer** (Long)	isRequested	boolean
		updateTime	Date
		status	String
		numSMS	String

Figure 23 : Modèle de la Base de données orientée document pour le client et les SMS

On retrouve ci-dessus la modélisation pour la base NoSQL document. Les relations de type (1-n) sont :

- la boutique et l'utilisateur,
- la boutique et le client
- et l'utilisateur et le client

Ce type de relations est modélisé par une colonne.

b) Présentation de l'API

L'API de PNS.com n'étant pas encore disponible, « ElasticSearch » a été choisi pour la remplacer.

« ElasticSearch » est un moteur de recherche libre (open source) d'architecture « REST » basé sur Apache Lucene. Il permet l'indexation de données sous forme de documents, leur recherche et une analyse de ces données en temps réel.

Aucun format de document n'est imposé. Lors de l'ajout d'un document « JSON », « ElasticSearch » va détecter la structure de données, indexer ces données et les rendre consultables. Ensuite, suivant les spécificités du domaine d'utilisation, il est intéressant de structurer fonctionnellement ces documents pour personnaliser la façon dont ces données seront indexées.

Les requêtes « GET » permettent de récupérer un ou plusieurs documents. Il est possible de faire des recherches dans la base avec l'option de « query » pour spécifier une demande par exemple : `inStoreCustomerDoc/_search?q={store :19237},` la requête permet de retourner tous les clients qui sont dans la boutique 19237.

c) Intégration dans l'application

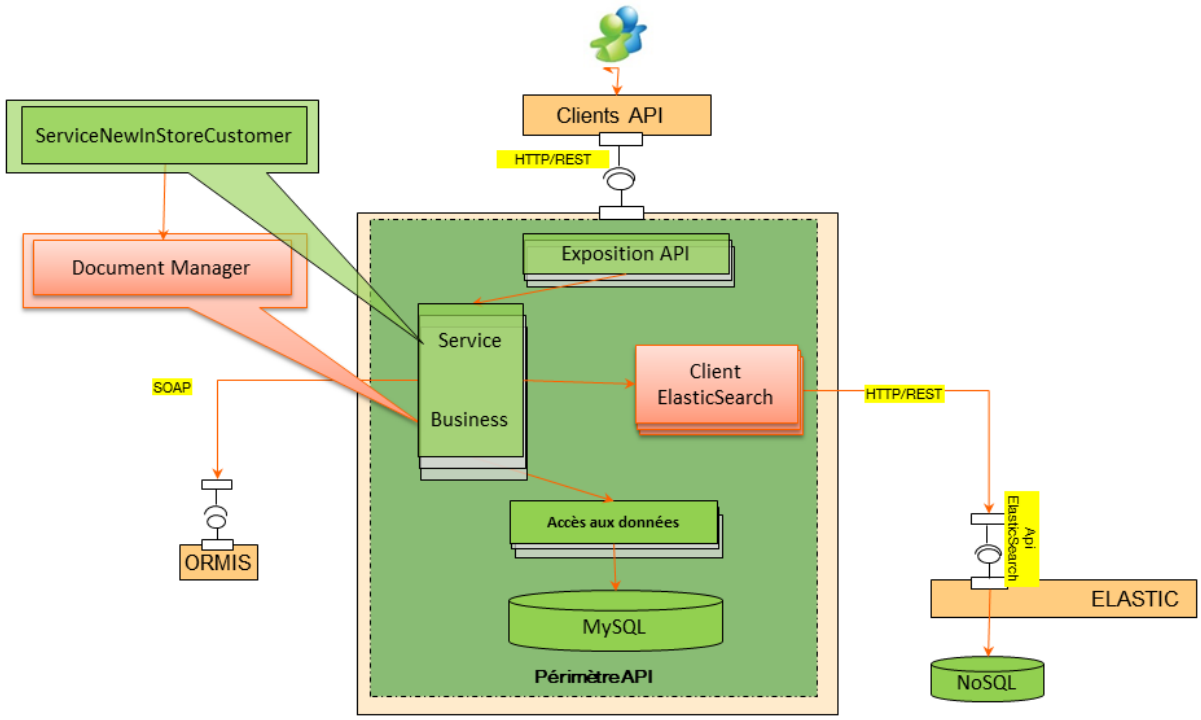

Figure 24 : Schéma du Prototype GDFA avec la base orientée document

En rouge, on retrouve les nouveaux éléments : « DocumentManager » et le connecteur avec l'API d'ElasticSearch. Lors du traitement d'une requête du client de l'API GDFA, le « ServiceNewInStoreCustomer » fait appel au « DocumentManager » pour faire le lien avec la base NoSQL.

Pour utiliser la génération automatique de code et avoir un client « JAX-RS » simplement, un « swagger » a été rédigé avec les fonctions utiles pour le prototype. Les méthodes mises dans le « swagger » sont :

- « GET » pour récupérer un document par son indexe
- « PUT » pour ajouter un document dans la base
- « POST » pour modifier un document
- « GET » avec la ressource « _search » pour faire une recherche parmi les documents

On utilise la même méthode pour générer un client qui permet la connexion et faire appel à l'API.

Des nouveaux « POJO » sont développés pour suivre le modèle défini.

Il y a toujours un « mapping » entre les « POJO » du document et des « DTO » et le mapping entre les « POJO » et une « `Map` » pour envoyer le corps des requêtes.

Le manager nommé « `DocumentManager` » ressemble à celui du graphe. La plus grande différence au niveau des « managers » est pour la recherche du client suivant. Ici, on envoie une requête de recherche dans la base avec des critères. Ensuite, sur la liste de clients que l'on récupère, on utilise un « stream » avec la fonction « min » avec un nouveau comparateur pour comparer les rangs entre les deux objets. Il n'y a pas besoin de filtrer la liste puisque cela a été fait du côté de la base.

Les autres fonctions du « manager » restent semblables à celle du « manager » graphe et sont utilisées au même endroit dans le `ServiceNewInStoreCustomer`, service de SMS et calcul de rang.

A la fin de la prestation dans le modèle document, on retrouve dans la figure ci-dessous le contenu de la base.

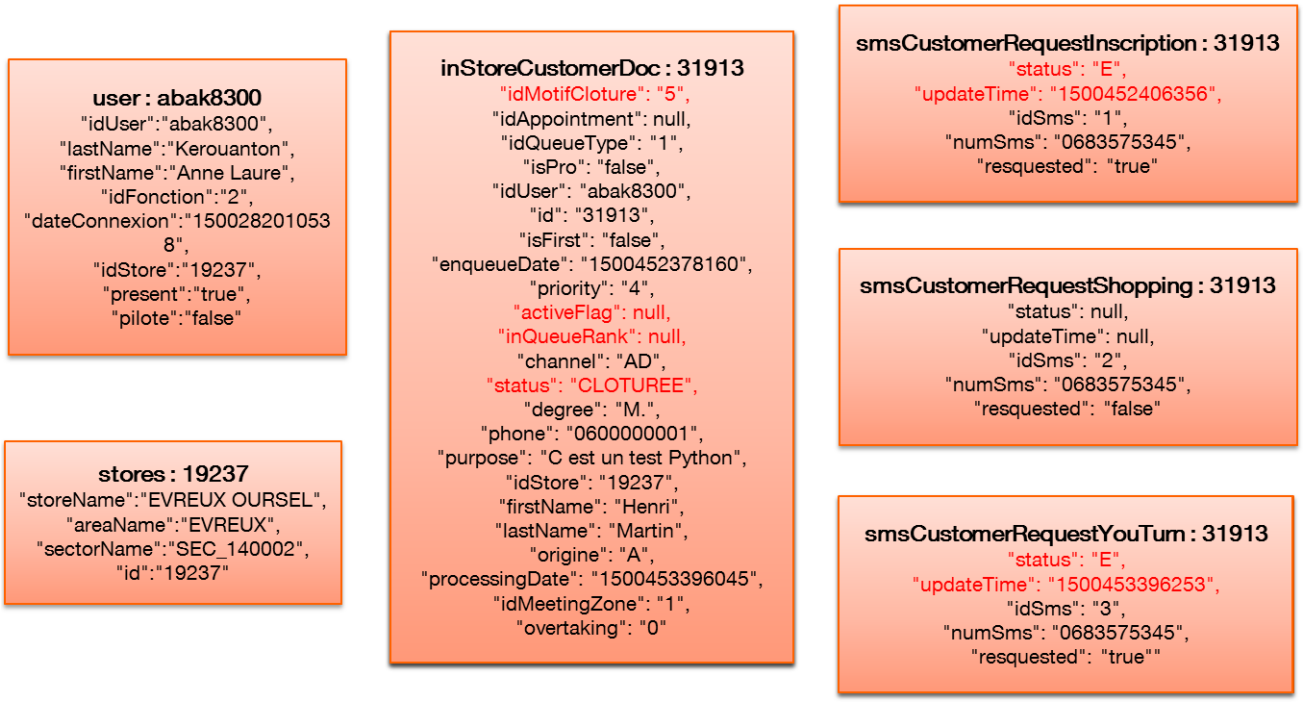

Figure 25 : Exemple d'un client dans une base orientée document

IV. Résultat et Discussion

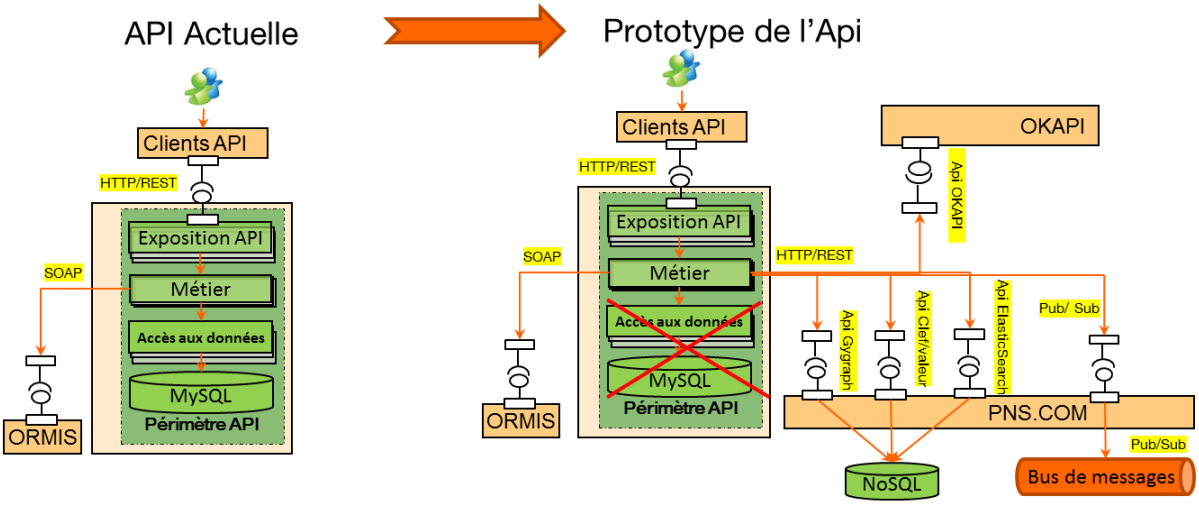

Figure 26 : Logiciel actuelle et Prototype

Cette figure explicite les modifications. Dans l'API actuelle, le service de gestion des SMS est synchrone et la persistance des données est gérée seulement avec une base relationnelle MySQL. Un prototype a été développé pour prendre en compte la gestion des SMS et la persistance des données avec du NoSQL. Ce prototype implémente tous les clients de PNS.com avec le choix de la base de données NoSQL dans le fichier de configuration au lancement de l'application GDFA.

1) Pour la gestion de l'asynchronisme

Il est assez difficile d'illustrer le résultat mais pour suivre l'exécution, des traces sont ajoutées.

a) Démonstration du processus de création

Du côté serveur, l'application démarre comme on peut le voir sur la figure ci-dessous (en rouge). Au lancement de l'application, l'application « listener » lance le nouveau « Pool » de « Thread » qui permet de souscrire dans le bus. La souscription requête l'API pour récupérer les messages et l'envoyer à ORMIS. On remarque que c'est bien le pool principal qui lance le service de souscription mais que la souscription se fait bien dans un nouveau pool de Thread.

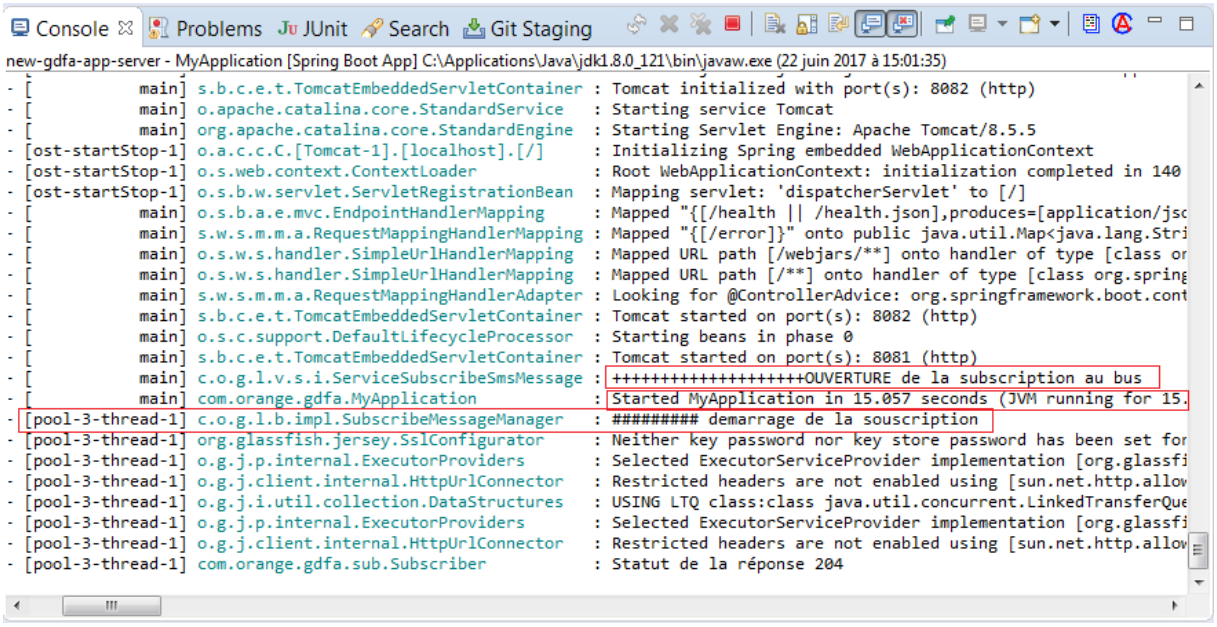

Figure 27 : Capture des traces au démarrage de l'Application GDFA

Du côté client, il y a « Postman » qui va permettre de simuler l'arrivée d'un client en boutique. Le client créé demande le SMS d'inscription. L'envoi de la requête lance le processus de création d'un client en file d'attente côté serveur.

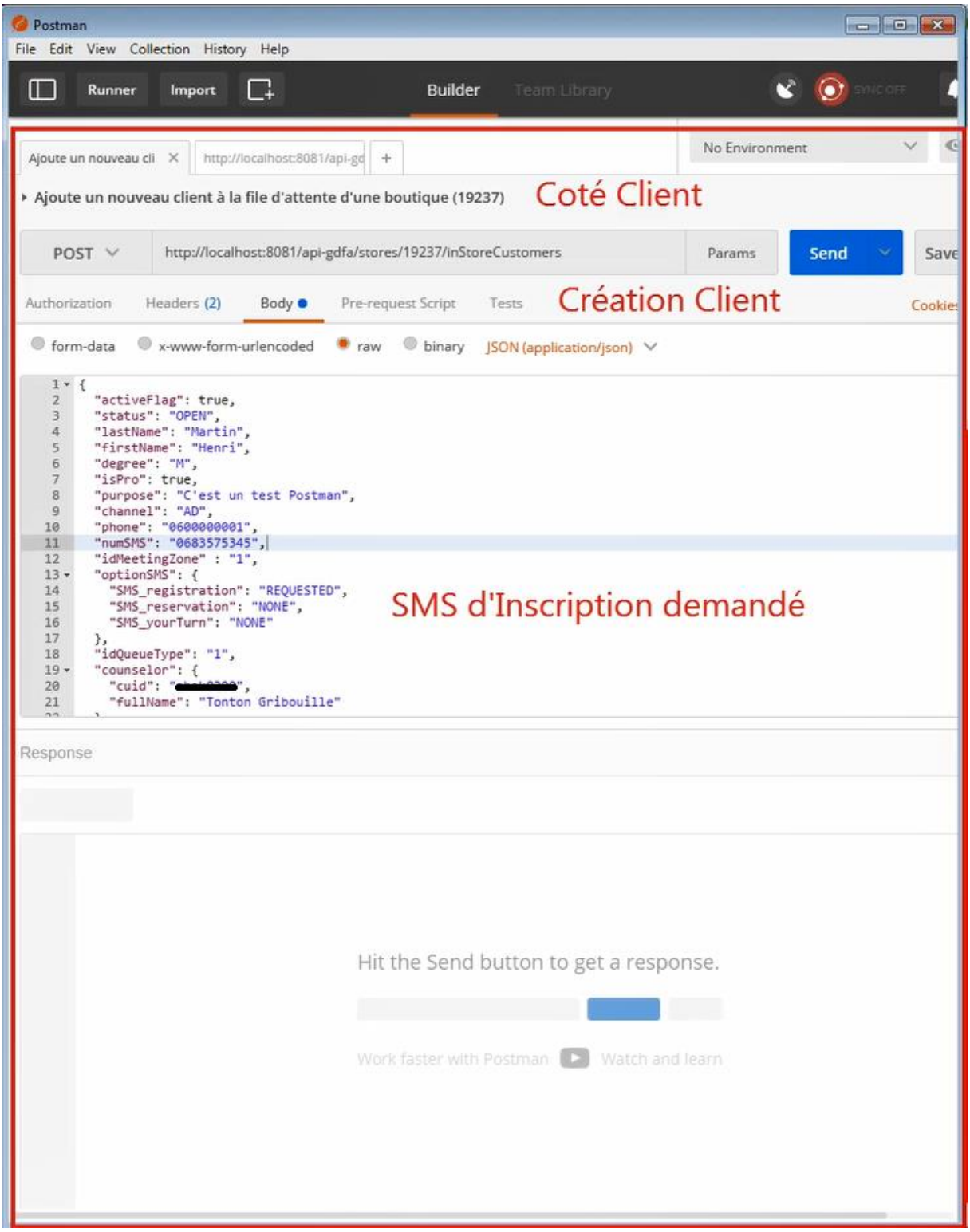

Figure 28 : Capture de Postman pour la création d'un client en boutique

Le serveur reçoit la requête de création et traite la requête. Un message est publié dans le bus comme on peut le voir sur la figure ci-dessous.

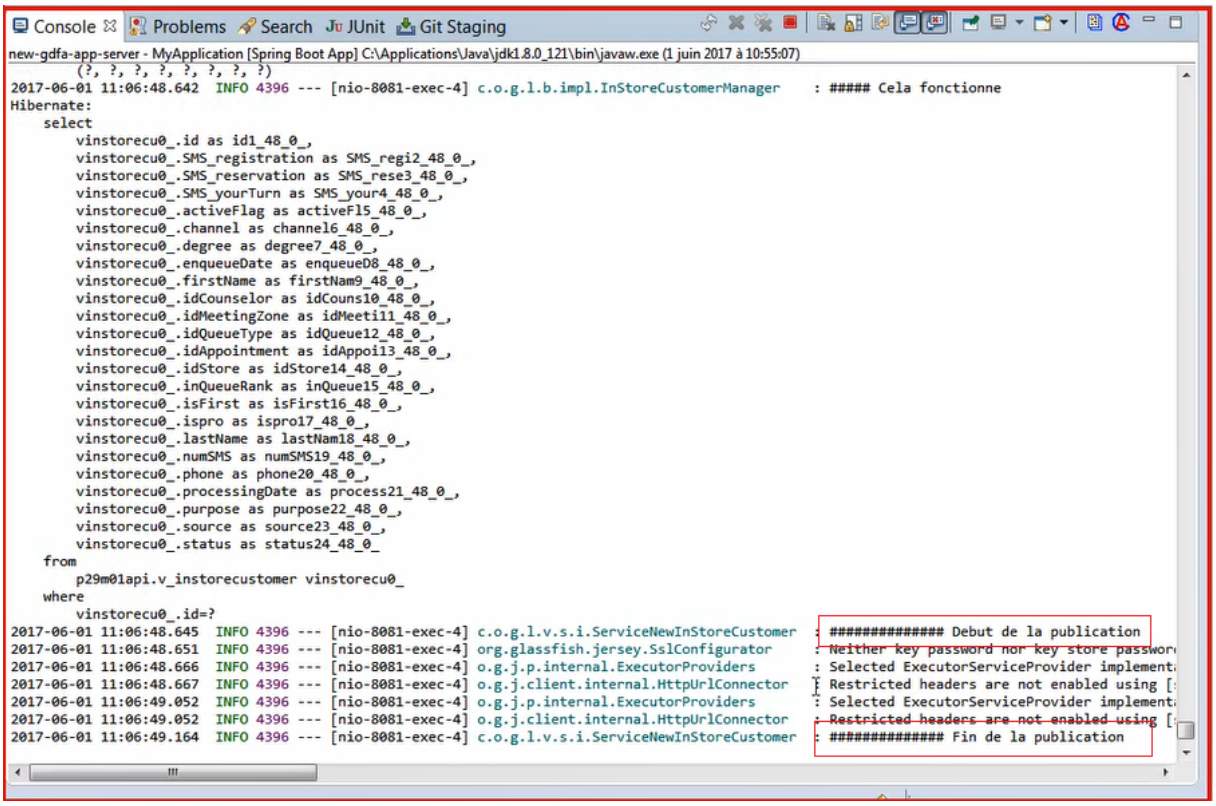

Figure 29 : Capture d'écran de la trace lors de la publication dans le processus de création

A la fin du traitement la réponse est envoyée au client sans que le SMS d'inscription ne soit envoyé.

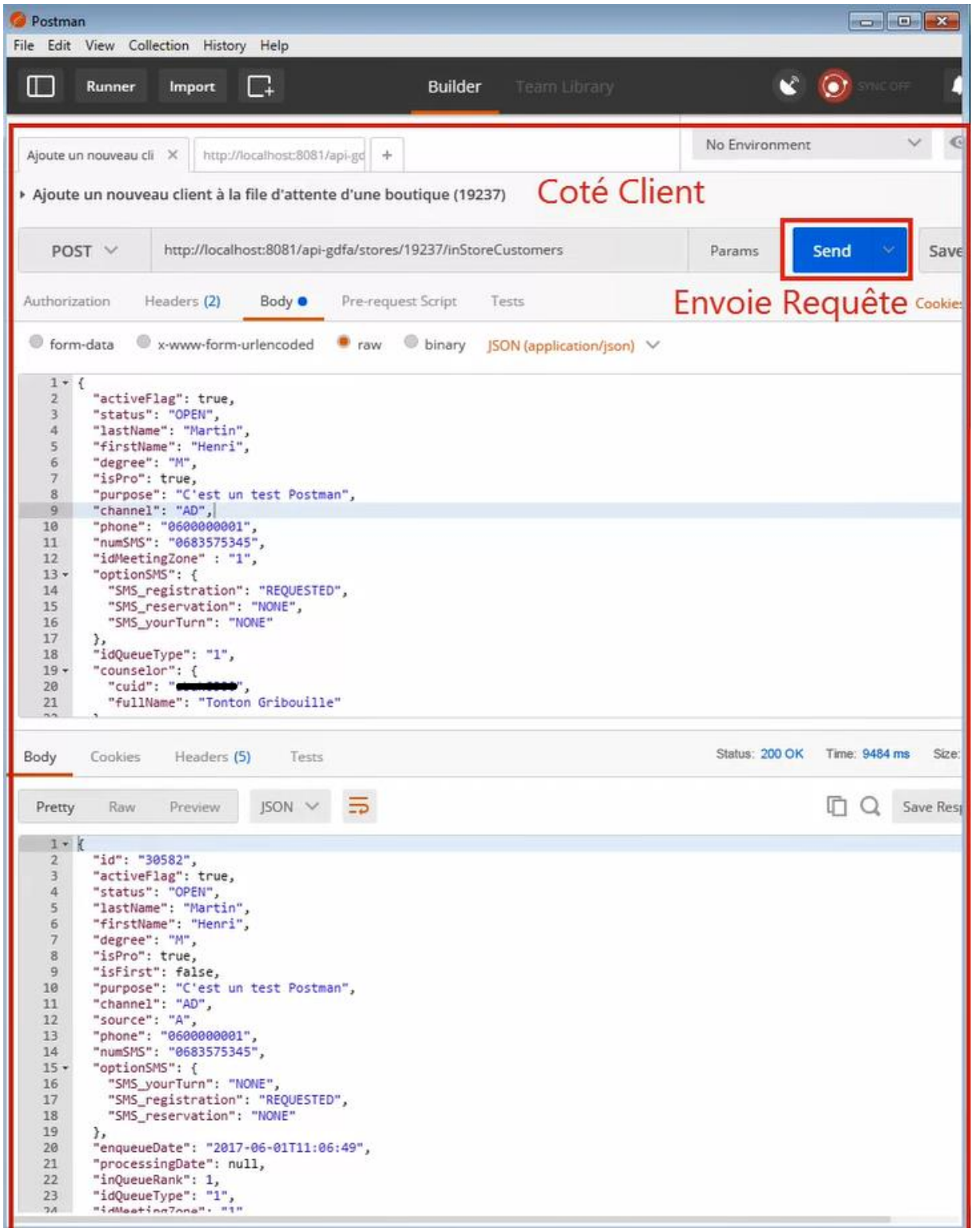

Figure 30 : Capture d'écran de la réponse envoyé à Postman

Périodiquement, le service de souscription interroge le bus pour récupérer les messages et les envoyer à ORMIS.

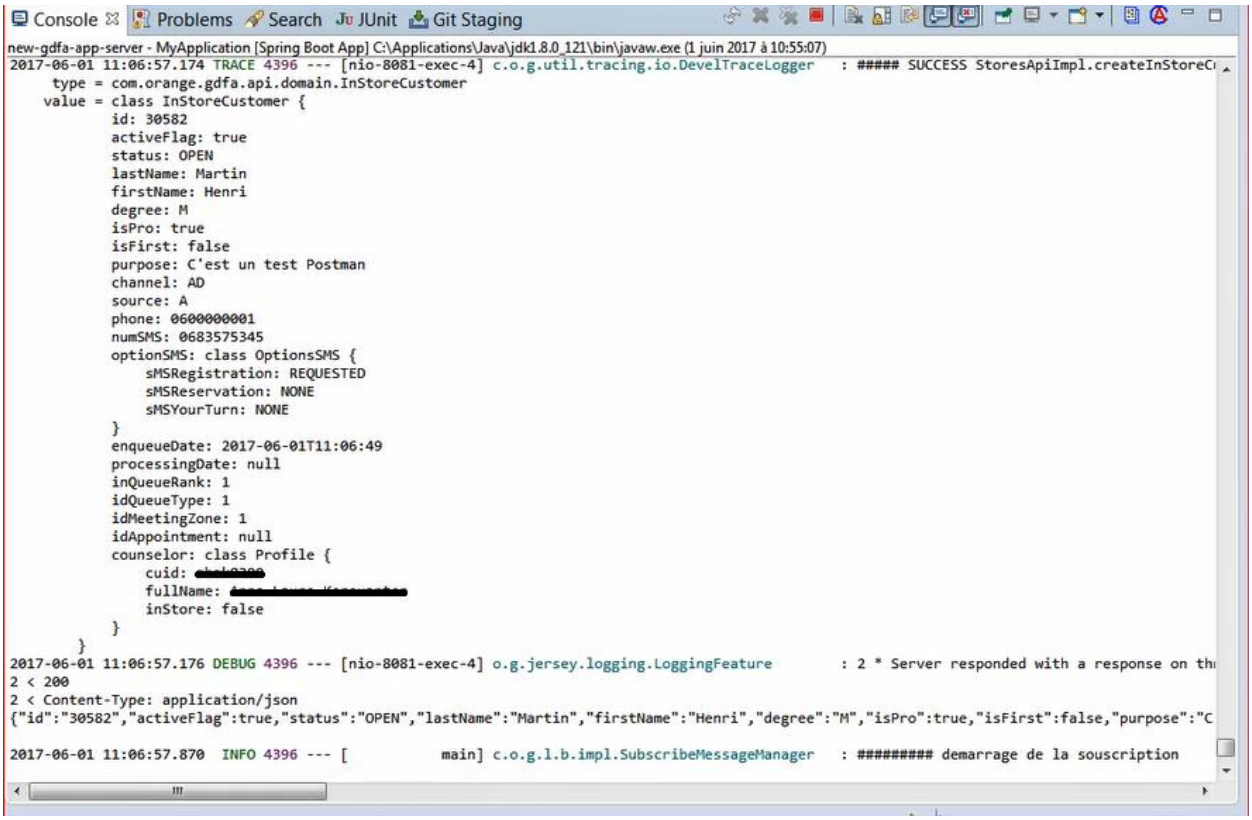

Figure 31 : Capture des traces lors de la fin de la création et début de la souscription

Le bus de messages permet bien de désynchroniser l'envoi des SMS. Dans notre exemple, un client est ajouté à la file d'attente de la boutique. En fin de processus de création, un message est publié dans le ZBus pour la gestion des SMS. Ensuite, le service de souscription interroge le bus régulièrement et souscrit aux messages publiés. Le contenu du message récupéré est envoyé au service d'envoi de SMS via ORMIS.

b) Remarques sur l'API

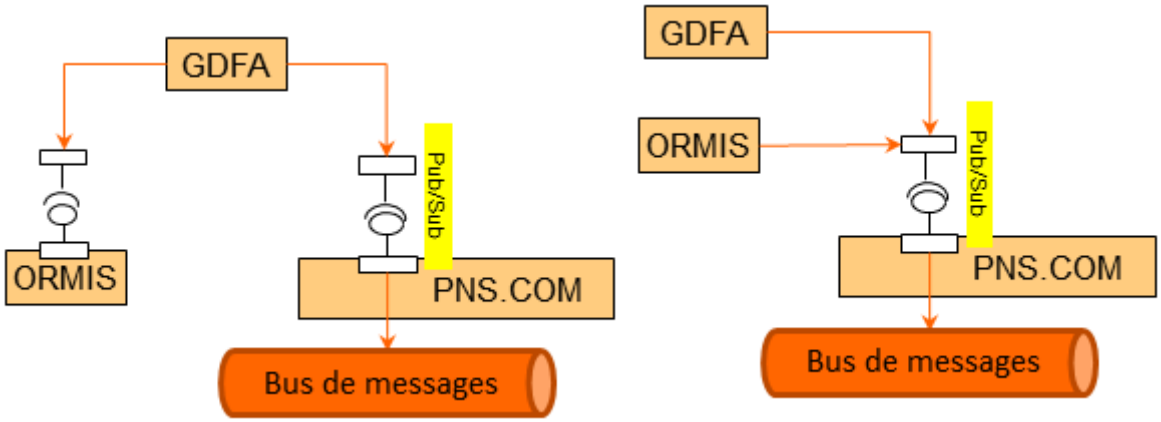

Figure 32 : Fonctionnement de GDFA avec ORMIS et PNS.com

Dans le prototype développé, GDFA fait la publication et la souscription des messages dans le bus de messages, puis envoi le message comme sur la figure 32 de gauche. Il aurait été intéressant de laisser la partie écoute du bus de message à ORMIS comme sur la figure 32 à

droite. GDFA publie dans le bus de messages et ORMIS souscrit et se charge d'envoyer le SMS. Mais ORMIS est une vieille technologique qui est très difficile à faire évoluer. D'où le fait que la souscription du bus se fait toujours dans GDFA.

On peut envisager que le GET renvoie une réponse « MetaData » avec une liste des messages, l'« ackid » et la date de réception pour faciliter l'intégration de l'application car il est assez gênant d'avoir l'« ackid » dans les headers.

Le client « OKAPI » qui permet de gérer l'authentification ne peut pas être utilisé avec un client JAX-RS. D'où le fait de développer un YAML qui permet d'avoir une interface pour récupérer un « Token ». Comme la durée du « Token » est assez courte (5 min), il faut qu'il soit généré périodiquement. Une classe permet de vérifier la validité du « Token » et d'en régénérer un en cas de besoin.

c) Remarques sur le fonctionnement du bus de messages

Le bus fonctionne comme une pile, dès que le message est consommé, il est retiré de la pile. Mais si aucun acquittement n'est fait, il ne sera pas supprimé et donc au bout d'un cycle complet, il réapparaitra.

L'acquittement se fait sur un paquet de messages et ne peut pas se faire sur un message spécifique. L'identifiant qui est généré par « RabbitMQ » n'est valable que pendant un certain temps et est de la forme 'lettres aux hasards + chiffres correspondant aux messages', par exemple : ackdedjejdej&1,2,3. Le fait d'accéder au numéro des messages peut inciter l'utilisateur à le modifier pour essayer de n'acquitter qu'une partie des messages. Hors, cela ne peut pas être réalisable car l'acquittement se fera toujours sur le paquet de messages reçus. Donc on peut proposer d'encoder l'ackid pour éviter toute modification.

2) Gestion de la persistance

Les différents prototypes réalisés permettent de faire une comparaison entre les bases de données relationnelles SQL classiques et les bases de données NoSQL orientées graphes, clé/colonnes, documents.

Scalabilité Horizontale

La scalabilité horizontale d'une base de données est sa capacité à s'étendre sur plusieurs serveurs au lieu d'être limitée à un seul.

L'architecture distribuée du modèle NoSQL permet l'utilisation d'un grand nombre de serveurs. Tous les serveurs sont en mode actif. Le partage de la charge sur plusieurs serveurs permet de facilement augmenter la capacité de stockage et de traitement de la base. Tous les serveurs sont en mode actifs pour atténuer les limitations de taille et de haute disponibilité ou de redondance.

Une architecture conventionnelle des bases de données relationnelles SQL est centralisée et ne permet pas la scalabilité horizontale. Le modèle SQL fonctionne avec deux serveurs : un

en mode actif, l'autre passif. Un basculement se fait du serveur actif, en défaut vers le passif qui devient actif, en cas de problème. On appelle cela un PRA, plan de reprise d'activité.

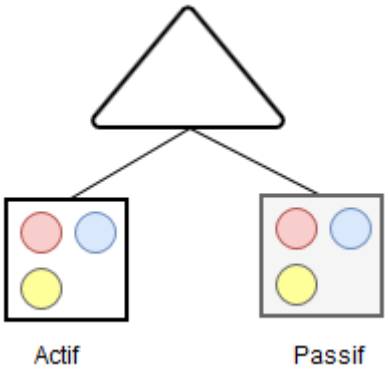

Figure 33: Représentation des serveurs SQL

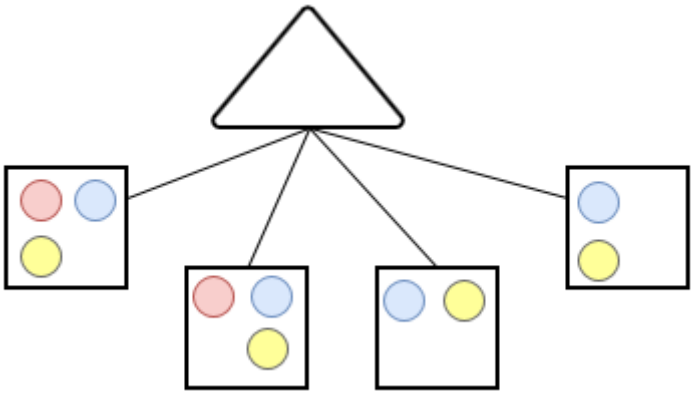

Figure 34 : Représentation des serveurs NoSQL

Scalabilité Verticale

La scalabilité verticale est sa capacité à améliorer les performances d'un serveur. Pour les améliorer, il faut faire varier l'espace disque, la RAM ou la puissance du CPU. Dans le modèle SQL, cette scalabilité est la seule solution pour améliorer les aptitudes générales de la base de données. Les changements se font en parallèle sur les deux serveurs. Alors qu'avec le NoSQL, la scalabilité verticale peut se faire indépendamment pour chacun des serveurs. En effet, comme on peut le voir sur le schéma, les serveurs peuvent jouer des rôles différents. Dans le cas NoSQL, la scalabilité horizontale est préférée à celle verticale.

Atomicité

La propriété d'atomicité assure qu'une transaction se fait au complet ou pas du tout. Avec les produits de bases de données relationnelles, l'atomicité est respectée ; la requête doit s'exécuter entièrement et correctement. Ce concept de tout ou rien montre une certaine rigidité du modèle SQL pour l'exécution d'une transaction. Avec l'API graphe et l'API Val Key, il est possible de forcer avec le header X-HTTP-Force-Status-200 pour faire une transaction

qui s'exécute partiellement ; la réponse renvoie les informations de ce qui n'a pas été réalisé en terme de traitement. Ceci donne de la souplesse dans le traitement des transactions en cas d'erreur. En effet, on peut ainsi choisir de faire un Roll back complet ou une correction intelligente mais un développement est nécessaire pour l'implémenter dans la couche applicative métier. « ElastikSearch » peut aussi faire une transaction qui s'exécute partiellement. (L'atomicité peut permettre de garder la cohérence de la base.)

La complexité des transactions

Dans MySQL, le langage SQL est un langage riche qui permet de réaliser des transactions complexes (requêtes avec plusieurs jointures, mise à jour de plusieurs entités en même temps, imbrication de requêtes). Hibernate intégré à Java permet de faire des requêtes complexes sans connaitre le langage SQL. Avec les APIs NoSQL, les transactions restent assez simples : une action avec une requête (CRUD). Pour récupérer les données avec les APIs de PNS.com, il faut connaitre l'id, on ne peut pas faire une recherche à partir d'un attribut. Pour accéder à des requêtes de lecture sur la base avec des paramètres, une API Select est disponible mais non utilisée dans le cadre du prototype. « ElastikSearch » permet de faire des transactions un peu plus complexes. Par exemple, il est possible de faire une mise à jour, suppression à partir d'une requête et non avec l'id ou la clé de l'entité. Mais on ne peut pas avoir d'imbrication de requêtes.

Accès aux données

L'extraction de données se fait par l'intermédiaire d'Hibernate et de SQL pour MySQL alors que le NoSQL est accessible par les API spécifiques. Hibernate permet de faire le « mapping » entre la base de données et les entités de l'application. Alors que le « mapping » pour le NoSQL reste à faire du côté applicatif. Le langage SQL possède de nombreuses fonctions d'agrégations et permet de faire des jointures. Il faut noter qu'une jointure peut faire baisser les performances de la base relationnelle et une vue peut être préférée. Les jointures ne sont disponibles qu'avec le graphe car dans ce modèle, la relation est représentée par un arc. Et dans le modèle clé/colonne une relation entre deux entités peut être représentée par la composition d'une clé mais la jointure devra être faite du côté applicatif. Les fonctions d'agrégation dans le modèle NoSQL restent à faire du côté applicatif.

Modèle

L'architecture des données est réalisée en fonction de la demande finale du client (en fonction des requêtes qui seront émises donc en fonction de l'usage). On gagne en performance pour répondre aux besoins spécifiques en agrégeant. Mais les performances peuvent se détériorer si le besoin change ou un nouveau besoin survient.

Le modèle graphe reste proche d'un modèle relationnel tout en restant simple, la complexité sera plutôt dans la couche applicative. Le modèle document et clé valeur sont proches. Dans le modèle document, une clé est associée à une grande quantité

d'informations structurées de manière hiérarchique. Pour compenser l'absence de relation, il y a plus de redondances (duplications) de données dans les modèles clé/Valeur et document. L'agrégation des différentes ressources est réalisée dans la couche applicative.

Dans le graphe, les relations entre les nœuds sont représentées par des arcs qui peuvent aussi contenir des propriétés. (Par exemple, en relationnel, une relation n-n est représentée par une table intermédiaire dite table de lien qui fait la relation entre les deux autres tables ; en graphe, toutes les relations (1-1, 1-n, n-n) sont présentées sous formes d'arc.)

Dans le clé/colonnes, pour réaliser le modèle, il est nécessaire de penser aux requêtes qui vont être émises et aux valeurs qui vont être modifiées. La représentation de relation 1-n et 1-1 peut se modéliser :

- par la composition d'une clé, c'est-à-dire que la clé contient deux éléments. Par exemple, le numéro de la boutique et le numéro du client.
- une colonne. Par exemple, le vendeur est une colonne (avec la redondance de données pour éviter de faire deux requêtes).

Les relations n-n peuvent être aussi représentées par une composition de clés (id de la boutique + id du client + id du type de SMS) mais ici cela n'était pas possible du fait de la limitation à deux clés, donc on a décidé de faire une colonne pour chaque type de SMS (cela n'est pas la meilleure solution => si on n'a pas le nombre de relations possibles (possibilité d'avoir beaucoup de valeurs de colonne à nulle)).

Flexibilité

Dans MySQL, le tableau compte toujours la même quantité de valeurs et comportent toujours le même type de données (espace alloué pour les valeurs nulles = espace perdu). Dans le NoSQL, chaque instance a sa propre structure unique. Des champs peuvent être ajoutés comportant une valeur définie au choix. (Pour deux entités de même type, le même attribut peut avoir un type différent.) Dans le graphe et le document, on peut ajouter dynamiquement une nouvelle base, un nouveau type d'entité et un nouvel attribut à une entité. Dans le clé/colonnes de l'API, on peut seulement ajouter un nouvel attribut à une colonne mais pas de nouvelle colonne. Donc le choix du modèle doit être pensé en amont selon les besoins.

Cohérence des données

En cas d'erreurs, « Hibernate » intègre le « Roll back » automatique et le « commit ». Dans le NoSQL, on doit gérer nous-même la cohérence des données en cas d'erreur. Cette gestion doit prévoir un traitement du côté applicatif, par exemple : la suppression des données ou continuer en prenant en compte les informations. Dans l'API graphe, la suppression d'un nœud entraine la suppression des arcs qui lui sont rattachés ce qui assure une certaine cohérence.

Intégrité

Dans MySQL, l'intégrité des données est assurée par la notion de clé étrangère et de clé primaire, il y a aussi la possibilité de garantir qu'une valeur ne soit pas nulle. Dans le NoSQL, l'intégrité fonctionnelle de la donnée n'est pas garantie du côté serveur mais du côté applicatif. Le modèle graphe permet d'assurer l'intégrité avec la notion d'arc (les nœuds peuvent être créés par la création du lien).

Les APIs

L'API Val Key est parfois lente lors des actions PATCH. Pour l'instant, on est limité sur le nombre de clés ce qui peut modifier un modèle et donc le rendre moins performant. Le modèle doit prendre en compte que la modification d'une colonne entraine l'écrasement de l'ancienne valeur de la colonne et donc le découpage en colonne doit prendre en compte les modifications qui peuvent être réalisées ensuite (par exemple le rang qui est souvent modifié). Les verbes sur les ressources sont inadaptés : POST pour une lecture et PATCH pour une création et modification.

Pour l'ElastikSearch, je n'ai pas trouvé le swagger officiel. L'API offre beaucoup de possibilité et elle est rapide. L'ElastikSearch se charge lui-même du numéro de version du document.

Le swagger de l'API GyGraph a été modifié pour spécifier la réponse des requêtes GET.

Choix entre MySQL et le NoSQL

NoSQL est un modèle de BDD plutôt adapté au web, où l'utilisateur et l'accès à l'information sont mis en avant (Accès direct à l'information).

	Adapté	**Pas Adapté**
Base de données orientée clé valeur	besoin de haute disponibilité et faible latence	gestion de relations complexes
	données codées de multiples façons sans schéma rigoureux	recherche par autre élément que la clé
	Exemple : gestion des sessions, à la desserte de contenus publicitaires et à la gestion des profils utilisateur ou produit	
Base de données orientée documents	stockage de différents types de données pour chaque document	transactions complexes
	schéma non rigide	si besoin agrégation de données
	si requête sur élément autre que la clé	
	Exemple : journalisation d'événements, commerce en ligne, à la gestion de contenu et au traitement analytique approfondi et des prototypes rapides	
Base de données orientée colonnes	système avec peu d'opérations en écriture	système affichant des requêtes très variées
	colonnes lues fréquemment et	transaction ACID

	simultanément	
	données non permanente (possibilité de définir l'expiration automatique)	
	Exemple : journalisation d'événements, gestion de contenu et au comptage et/ou la catégorisation d'analytiques	
Base de données orientée graphe	données avec un nombre indéterminé de relations	données changeant souvent
	Exemple : la prise en charge de moteurs de recommandations, l'acheminement et l'expédition des livraisons, les systèmes sensibles à la localisation, les réseaux des transports publics, les cartes routières, les conditions préalables des programmes d'étude et les topologies réseau	mises à jour en temps réel de grands volumes de données
		performance peut diminuer si partitionnement de base de données sur un réseau

Pour des modèles complexes où la cohérence des données est importante, il est préférable de rester sur une base relationnelle. Le type NoSQL est réservé à des modèles prédéfinis ou assez simple. On peut suggérer que GDFA soit moins monolithique en utilisant à la fois du relationnel et du NoSQL. En effet, les données stockées pour GDFA doivent être cohérente et les transactions doivent respecter le principe ACID. Mais, il serait intéressant de coupler le relationnel avec du NoSQL pour par exemple, l'affichage de la file d'attente sur les écrans de la boutique ou des publicités en utilisant une base de données orientée clé/valeur.

Conclusion

Pour améliorer les performances de la gestion de la file d'attente en boutique, le stage propose de tester la faisabilité et la validité de solutions permettant de gérer l'asynchronisme du service de l'envoi de SMS et une solution pour la persistance des données. Les prototypes développés utilisent des APIs proposées par PNS.com.

La première solution implémente un pattern « Publish and Subscribe » en utilisant une API de bus de messages. La publication des messages est réalisée dans les processus du cycle de vie du client et la souscription des messages fait appel à un pool de threads qui se lance au démarrage et interroge le bus périodiquement. Cela permet de gérer les services de GDFA des appels à l'application ORMIS de façon asynchrone.

La deuxième solution teste à travers plusieurs prototypes l'utilisation d'APIs pour faire appel à différentes bases de données NoSQL. Les types de bases NoSQL étudiés sont le graphe, le clé/colonnes et le document. Cette étude met en évidence les différences entre le NoSQL et le relationnel au niveau de la scalabilité, la flexibilité, des transactions et de la modélisation… Le choix du NoSQL sur le relationnel provoque une rupture dans les pratiques, les préconisations d'architecture, de conception et de développement. On passe d'un modèle tout ou rien à un modèle ouvert plus complexe qui demande plus de développement du côté applicatif. De plus, il me semble intéressant que GDFA parte sur du NoSQL pour certains cas d'utilisation mais en gardant un socle relationnel. Le NoSQL ne serait qu'un simple complément à la base relationnelle.

Ce stage m'a permis de découvrir ce qu'est le travail d'ingénieur informatique dans une grande entreprise. Travailler sur un projet aussi diversifié pour l'entreprise Orange a été une réelle opportunité. J'ai pu améliorer mes compétences dans le domaine de la communication, approfondir mes connaissances informatiques avec l'utilisation de technologie comme Maven, Spring Boot, et la connaissance de l'écosystème API (HTTP, REST/JSON)

Pour la suite du stage, j'étudie la possibilité d'un nouvel algorithme pour calculer le rang des clients dans la file d'attente. Le résultat du calcul sera stocké dans une base de données en local. Il serait également intéressant d'améliorer les prototypes en y intégrant l'industrialisation des clients des APIs de PNS.com réalisée par un développeur expert.

Lexique

AckId est un identifiant renvoyé par l'API « ZBus » lors d'un « GET » pour acquitter le paquet de message.

API est un acronyme pour Applications Programming Interface. Une API est une interface de programmation qui permet de se « brancher » sur une application pour échanger des données. Une API est ouverte et proposée par le propriétaire du programme.

API « GyGraph » est une API proposée par PNS.com pour accéder à une base de données orientée graphe.

API « KeyVal » est une API proposée par PNS.com pour accéder à une base de données orientée clé/colonnes.

API « ElasticSearch » est un moteur de recherche open source qui permet d'utiliser une base de données orientée document.

Un **Bean** est un objet instancié, assemblé et géré par le conteneur Spring. Ces Bean sont créés avec les métadonnées de configuration que vous fournissez au conteneur.

Bus de message est une file stockant des messages utilisée à la demande.

Certificat est un fichier de données qui lie une clé cryptographique aux informations d'une organisation ou d'un individu.

Un **client JAX-RS** est basé sur Java pour accéder aux ressources Web. Il ne se limite pas aux ressources mises en œuvre en utilisant JAX-RS. Il fournit une abstraction de niveau supérieur par rapport à une API de communication HTTP simple et une intégration avec les fournisseurs d'extension JAX-RS, afin de permettre une mise en œuvre concise et efficace de solutions réutilisables côté client qui utilisent les implémentations client existantes et bien établies de la communication HTTP.

Commit désigne l'enregistrement effectif d'une transaction dans la base de données.

CRUD est l'acronyme pour « Create Read Update Delete ». Ceci désigne les quatre opérations de base pour la persistance des données, en particulier le stockage d'informations en base de données.

DAO est l'acronyme pour Data Access Object. C'est un modèle pour concevoir une solution. Il permet de s'abstraire de la façon dont les données sont stockées au niveau des objets métier.

DEVOPS correspond à la fusion des tâches qu'effectuent les équipes d'une entreprise chargées du développement des applications (Dev) et celles de l'exploitation des systèmes (Ops, pour opérations).

DSI est l'acronyme pour Direction du Système d'Information.

DTO est l'acronyme pour Data Transfert Object. C'est un modèle pour simplifier le transfert de données entre deux systèmes ou sous-système d'une application.

Entity objets représentatifs de la structure du schéma de la base de données.

Essentiel 2020 est la stratégie d'Orange pour améliorer ses services et ses prestations pour 2020.

Framework est un ensemble d'outils et de composants logiciels organisés conformément à un plan d'architecture et des patterns, l'ensemble formant ou promouvant un « squelette » de programme.

GDFA est l'acronyme de Gestion de la File d'Attente.

« **GET** » est une requête HTTP pour récupérer des informations à partir du serveur sans avoir d'effet sur les données.

Hibernate est un Framework open source gérant la persistance des objets en base de données relationnelle.

HTTP signifie Hypertext Transfer Protocol (traduction: protocole de transfert hypertexte). Ce protocole définie la communication entre un client (exemple: navigateur) et un serveur sur le World Wide Web (WWW).

Framework Jersey est un Framework open-source pour développer des services Web en langage JAVA.

Kerberos est un protocole d'authentification réseau qui repose sur un mécanisme de clés secrètes et l'utilisation de tickets.

Keystore représente un moyen de stockage pour les certificats.

Keytab est un fichier utilisé pour authentifier plusieurs systèmes distants à l'aide de Kerberos sans entrer de mot de passe.

Manager est une classe permettant de gérer d'autres classes.

Mapping est une méthode permettant de définir la correspondance entre deux modèles de données.

Maven est un outil permettant de faciliter la gestion de projet.

MessageDTO est la classe définissant le message qui va être envoyé dans le bus de messages.

MetaData est une donnée servant à définir ou décrire d'autres données.

NoSQL signifie « Not Only SQL ». C'est une base de données qui n'est pas relationnelle.

ObjectMapper fournit des fonctionnalités pour la lecture et l'écriture de « JSON », à destination et en provenance des « POJO » basiques. Il est également hautement personnalisable pour fonctionner à la fois avec différents styles de contenu « JSON » et pour prendre en charge des concepts d'objets plus avancés tels que le polymorphisme et l'objet.

ORMIS une application pour la gestion des envois en masse de messages multicanaux (SMS, MMS, WAP Push, mails…) pour les applications Orange France.

OKAPI est un service au sein d'Orange qui gère les « Tokens ».

Pattern est un modèle spécifique représentant la solution à un problème récurrent dans la conception d'application.

Pattern Proxy est un patron de conception. Un proxy est une classe se substituant à une autre classe. Cela permet une simplification d'utilisation d'objet complexe et le contrôle d'accès aux méthodes de la classe substituée.

PNS est l'acronyme de « Profile and Syndication ». C'est un service au sein de la DSI d'Orange qui met à disposition des clients différentes API.

Polymorphisme est un concept de la programmation orientée objet. Il consiste à livrer une interface unique à des entités ayant différents types.

Pool de threads est un ensemble de threads utilisables pour exécuter des tâches en fonction des besoins.

« **POST** » est une requête HTTP qui accepte un corps de message pour la plupart du temps le stocker.

PRA est l'acronyme de plan de reprise d'activité. Ce plan permet de remettre en route les besoins informatiques nécessaires.

Project Object Model (POM) est l'unité de travail fondamentale de Maven. C'est un fichier XML qui contient des informations sur le projet et les détails de configuration utilisés par Maven pour créer le projet. Il contient des valeurs par défaut pour la plupart des projets.

Prototype est un premier exemplaire construit d'un ensemble mécanique, d'un appareil, d'une machine et qui est destiné à en expérimenter en service les qualités en vue de la construction en série.

PUB/SUB signifie « publish and subscribe ». C'est un pattern.

« PUT » est une requête HTTP qui permet de mettre à jour une valeur ou de la créer si elle n'existe pas encore.

RabbitMQ est un système permettant de gérer des files de messages afin de permettre à différents clients de communiquer très simplement.

RCGP est l'acronyme pour Relation des Clients Grand Publique.

Ressource est l'ensemble des référents accessibles depuis un site web.

Rollback est une commande utilisée pour annuler une transaction dans une base de données.

Socket désigne une interface de connexion.

Spring est une solution open source complète pour le développement d'applications reposant sur un conteneur qui implémente le motif de conception inversion de contrôle et sur l'utilisation de l'AOP (programmation orientée aspect). L'inversion de contrôle est assurée de deux façons différentes : la recherche de dépendances (un objet interroge le conteneur pour trouver ses dépendances avec les autres objets) et l'injection de dépendances (création dynamique des dépendances entre les classes en s'appuyant sur une description ou de manière programmée). Le but de Spring est de faciliter et de rendre productif le développement d'applications, particulièrement les applications d'entreprises.

Les instances de « **SSLContext** » représentent une implémentation de protocole de « socket » sécurisé.

Stream représente un flux de données entrant ou sortant.

Swagger est un format de définition puissant pour décrire les API RESTful. La spécification crée une interface RESTful pour développer et consommer facilement une API en effectuant un mappage efficace de toutes les ressources et opérations associées.

Théorie des graphes est une discipline informatique qui étudie les graphes.

Token ou jeton d'authentification est une solution d'authentification forte.

Bibliographie et Webographie

Orange, « Dossier d'architecture de référence », février 2017

Orange, « Application GDFA – Etape 2 Architecture solution», février 2017

Orange, « Application GDFA : Architecture logicielle », février 2016

Orange, « Cahier des Charges POC GDFA », mars 2017

Principe de mise en œuvre d'une génération automatique (Mis à jour en juillet 2016)
http://doc-api.si.francetelecom.fr/dokuwiki/doku.php?id=apigp:codegen

ZBus - Publish / Suscribe service (Mis à jour en juillet 2017)
https://developer-inside.sso.infra.ftgroup/afi/apis/zbus/overview

PnS ValKey API – Documentation (Mis à jour en novembre 2016)
https://developer-inside.sso.infra.ftgroup/afi/apis/keyvalue/

PnS GyGraph API - Documentation (Mis à jour en juillet 2017)
https://developer-inside.sso.infra.ftgroup/afi/apis/gygraph/overview

Quel SGBD NoSQL pour vos besoins IT ? Critères de choix (Mis à jour en juin 2015)
http://www.lemagit.fr/conseil/Quel-SGBD-NoSQL-pour-vos-besoins-IT-Criteres-de-choix

Designing Orange API (créer en janvier 2017)
http://recommendations.si.fr.intraorange/designing-orange-api/

Documentation ElasticSearch (consulté en juillet 2017)
https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html