INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Image : http://img.decision-achats.fr/

Leriche Florian

Rapport de stage ingénieur

Étudiant : LERICHE Florian
Maîtres de stage : BERTHAUD Laurent, SACENDA Cyril
Entreprise : Sopra Steria
Clients : BP1818, Neuflize OBC

Date : 13/02/2017 - 12/07/2017
Lieu : Paris, France

Le Digital Banking

SOPRA STERIA GROUP

Table des matières

Remerciements

TODO

Introduction

Le présent rapport détail le travail qui s'inscrit dans le cadre de la réalisation d'un stage ingénieur effectué par les élèves ingénieurs de l'Institut National des Sciences Appliquées de Rouen. Celui-ci s'est déroulé via la participation à deux missions distinctes pour le compte de la société de services *Sopra Steria* du 13 février au 12 juillet 2017 ; suite à une convention tripartite signée entre le département Architecture des Systèmes d'Information de L'INSA, l'entreprise d'accueil et moi-même. Pendant ce stage, j'ai eu l'occasion de participer à plusieurs projets qui m'ont permis d'appréhender le métier d'ingénieur en informatique, d'acquérir de l'expérience ainsi que de m'épanouir aussi bien dans le plan personnel que professionnel.

La première mission à laquelle j'ai pris part s'est déroulée du 13 février 2017 jusqu'à la fin de mon stage, pour le compte du client *Neuflize OBC*. La seconde mission a débuté le 29 mars 2017 pour la banque privée *BP1818* et s'est déroulée en parallèle de la première jusqu'à la fin du stage. La répartition du temps de travail était la suivante :

— Lundi, mardi et vendredi chez le client *BP1818*
— Mercredi et jeudi chez le client *Neuflize OBC*

Sopra Steria est une entreprise de services du numérique et l'un des leader européens dans la transformation numérique. Ainsi, l'objectif premier de mon stage a été d'accompagner certains des clients banques privées de Sopra Steria dans leur transformation digital en travaillant à la fois sur la réalisation et l'industrialisation d'une application mobile à destination des clients de la banque Neuflize ainsi que sur la mise en place d'une application web à destination des banquiers travaillant chez BP1818.

Dans un premier temps, je présenterai l'entreprise d'accueil et les client pour lesquels j'ai travaillé, leurs domaines d'activités, leurs origines, leurs personnel ainsi que leurs locaux. Dans un second temps, je détaillerai de manière précise les sujets des missions qui m'ont été confiées au sein des équipes. Enfin, je décrirai en profondeur le déroulement de mon stage ainsi que les différents travaux que j'ai accompli et les conditions dans lesquelles je les ai réalisé.

Pour des raisons de clarté et de cohérence, ce rapport présentera deux parties distinctes concernant le déroulement du stage, chacune ayant pour objectif de décrire le travail réalisé chez les clients cités plus haut et suivant le même schéma.

1. Présentation de l'entreprise

1.1. Historique

Sopra Steria est une entreprise de services du numérique (ESN) résultant de la fusion, en janvier 2015, de deux des plus anciennes sociétés de services en ingénierie informatique françaises, *Sopra* et *Steria*.

1.1.1. 1968 - 1985 : Les débuts

Création des sociétés Sopra et Steria respectivement en 1968 et 1969, période marquée par la naissance de l'industrie des services informatiques.

La **SO**ciété de **PR**ogrammation et d'**A**nalyses (Sopra), fondée par Pierre Pasquier et François Odin, est avant tout une entreprise de services de conseils technologiques spécialisée dans l'édition de logiciel. Elle signera, quelques années plus tard, son premier grand contrat d'infogérance bancaire qui marquera son initiation au savoir-faire relatif à la banque. Cela aboutira à la création de la première plateforme bancaire de Sopra qui proposera par la suite des logiciels à destination des banques. Par la suite, l'édition de solutions bancaires deviendra son activité phare avec la mise en production de sa première application concernant les crédits.

La **S**ociété d'**E**tude et de **R**éalisation en **I**nformatique et **A**utomatisme (Steria) contrôlée par Jean Carteron est aussi une société de services informatiques. L'informatisation de l'Agence France Presse est désignée comme étant l'une des première prouesse de la société qui participera par la suite au développement du minitel en travaillant sur la conception de son architecture.

1.1.2. 1985 - 2000 : Croissance et entrée en bourse

Sopra repense sa structure industrielle en décidant de se recentrer sur des activités précises telles que l'intégration de systèmes et l'édition de logiciels et décroche son premier grand projet national avec le ministère de l'intérieur. Elle est introduite en Bourse en 1990 et multiplie les contrats ainsi que les acquisitions avec, par exemple, le rachat de *SG2 ingénierie* marquant une forte croissance. Elle profitera de ses performances pour étendre son expertise à l'échelle internationale en s'implentant dans différents pays tels que le Royaume-Uni ou encore l'Allemagne.

De même, Steria étend son influence en dehors de la France, jusqu'en Arabie Saoudite où elle réalisera le système informatique de la banque centrale saoudienne. Elle intégrera le marché des transports à son domaine d'expertise, notamment grâce au projet d'automatisation du RER A en France, à Paris. Ses futures acquisitions lui permettront une entrée en Bourse en 1999.

1.1.3. 2000 - 2014 : Transformation numérique

L'essor des technologies du numériques, à savoir l'informatique et internet, provoque une mutation des marchés qui a pour conséquence d'apporter de nombreux clients potentiels à la recherche de partenaires pouvant les accompagner dans leur transformation numérique. Les deux sociétés répondront à cette problématique et développeront leurs activités de conseil. Sopra créera sa filiale *Axway* regroupant ses activités dans le domaine du progiciel et qui entrera aussi en Bourse de manière autonome en 2011. Toujours fortement impliquée dans le domaine bancaire, elle décidera de créer sa filiale *Sopra Banking Software* en 2012 et réalisera de nombreux projets avec les grands noms du secteur bancaire

français (Crédit agricole, Société général, BNP, etc...)

Steria, de son côté, se retrouvera dans un contexte économique difficile et vera le prix de son action chuter. Elle continuera malgré tout de multiplier les acquisitions (*Bull* en Europe, *Mummert Consulting* en Allemagne ou encore *Xansa* au Royaume-Uni). Elle remportera plusieurs des plus gros contrats (notamment SSCL pour la gestion du back office de plusieurs ministères de l'administration britannique) de son histoire avec le gouvernement britannique.

1.1.4. 2014 - 2016 : Fusion absorption

En 2014, Sopra, forte d'une grande croissance, prend la décision d'absorber Steria en devenant actionnaire majoritaire à plus de 90%. Il s'agit là d'une excellente opportunité misant sur la complémentarité des deux géants de l'informatique aussi bien sur le plan métier que sur le plan géographique. En effet, comme nous l'avons dit plus haut, Steria est très présente au Royaume-Uni contrairement à Sopra. De plus, Sopra se vera ainsi faire l'acquisition de nombreux centres de compétences qui viendront renforcer son écrasante influence à travers l'europe. Les deux entreprises partagent beaucoup de points communs tels que la taille, les domaines d'activités ou encore les objectifs, ce qui constitue un atout majeur concernant leur fusion et leur projets d'avenir. La fusion des deux groupes donne donc naissance à Sopra Steria, une entreprise possédant un poids écrasant, un capital titanesque permettant de gagner la confiance de nombreux clients ainsi qu'une très grande expertise lui permettant de mener à bien beaucoup de projets emblématiques tels que :

Figure 1.1. – Projets emblématiques

1.2. Identité et quelques chiffres

Sopra Steria est une entreprise de services du numérique devenue aujourd'hui le leader européen de la transformation numérique. Cette dernière accompagne les métiers dans leur transformation digitale en leur fournissant des services de conseils et en leur permettant d'établir les spécifications qui répondront au mieux à leurs attentes concernant la mise en place d'un système informatique. Elle assure ensuite toute la chaîne de production en réalisant la conception, le développement et la mise en production des solutions créées ainsi que leur maintien une fois ces dernières déployées.

Elle compte plus de 38 000 collaborateurs présents dans plus de 20 pays différents et dispose d'un très grand nombre de savoir-faire informatiques qui lui permettent d'offrir un panel de services extrêmement

Figure 1.2. – Quelques chiffres

complet. Celle-ci est côtée en Bourse et réalise en 2016 un chiffre d'affaire de 3.7 milliards d'euros dont la majeure partie provient de ses actions réalisées en France et au Royaume-Uni.

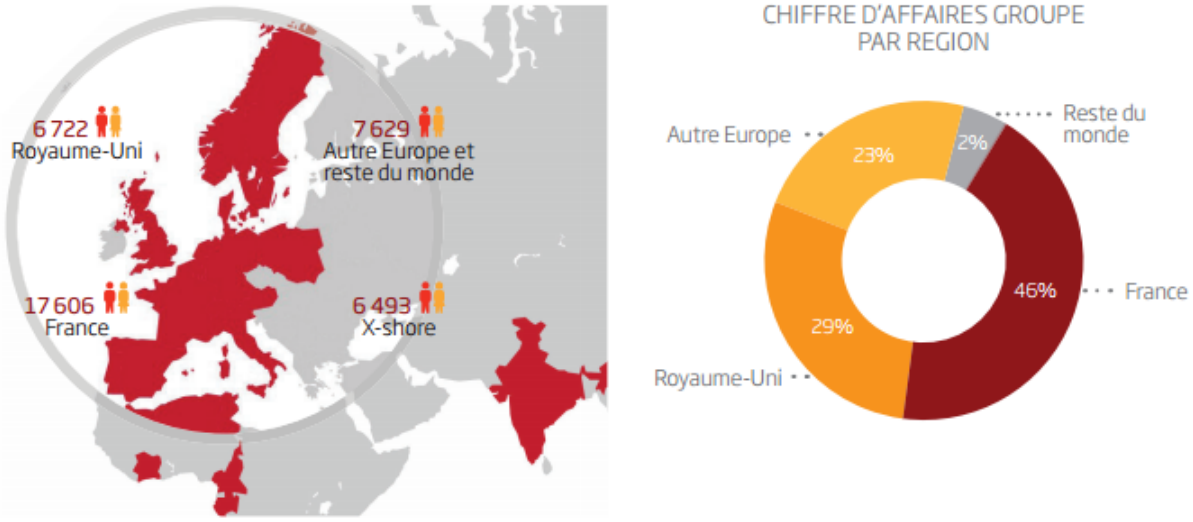

Figure 1.3. – Répartition internationale

1.3. Domaine d'activités

Sopra Steria agit dans de nombreux secteurs d'activités stratégiques lui permettant de suivre l'évolution de ses clients et de leur garantir des services en adéquation avec leur besoins.

1.3.1. Conseil et intégration de systèmes

Sopra Steria Consulting est la filiale orientée conseil de Sopra Steria dont l'objectif est d'assister les clients dans leur transformation numérique. Les consultants sont chargés de d'élaborer les stratégies et programmes de transformation avant de concevoir puis mettre en oeuvre les solutions qui répondront aux besoins des grandes entreprises. Une fois les solutions déployées chez le client, elles seront maintenues et auront la possibilité d'évoluer afin de d'offrir une certaine flexibilité permettant de répondre aux problématiques de la transformation continue. Enfin, Sopra Steria Consulting assure l'urbanisation des données permettant aux entreprises d'avoir accès à de nombreuses données leur permettant de suivre la satisfaction de leur clients et d'optimiser leur services.

1.3.2. Edition de solutions

Les solutions développées sont regroupées dans trois grands domaines qui sont les suivants :
— Bancaire avec la filiale Sopra Banking Software fournissant des progiciels à destination du secteur Banque et Finance
— Immobilier pour la gestion des patrimoines immobiliers
— Ressource Humaine avec Sopra HR Software fournissant des logiciels RH et s'occupant de l'externalisation des processus RH (voir BPS ci-dessous)

1.3.3. Infrastructure management

Sopra Steria adapte les infrastructures et repense la DSI des grandes entreprises afin qu'elles entrent en adéquation avec les nouvelles technologies et les mutations qu'elles impliquent concernant le numérique (cloud, big data etc...). Elle propose des offres de mise en place d'infrastructure as a service (iaas) consistant à offrir une infrastructure informatique (load balancers, bande passante etc...) reposant sur des ressources matérielles virtualisées située dans le Cloud. Sopra Steria propose aussi d'intégrer et de personnaliser les services Cloud (Infrastructure As A Service, Platform As A Service et Software As A Service) au sein des entreprises.

1.3.4. Business Process Services

Sopra Steria propose d'externaliser certaines des fonctions de l'entreprise telles que la finance, les ressources humaines pour la gestion du personnel ou encore des processus métiers spécialisés afin d'améliorer l'efficacité et la rentabilité de chacun de ces processus. Ces fonctions sont alors confiées à des partenaires ayant l'expertise nécessaire pour les exécuter. L'objectif premier du client faisant appel à ce genre de service est de se recentrer uniquement sur son coeur de métier.

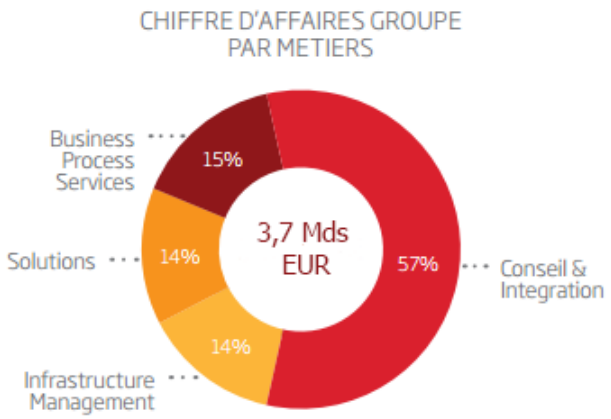

Figure 1.4. – Secteurs d'activités

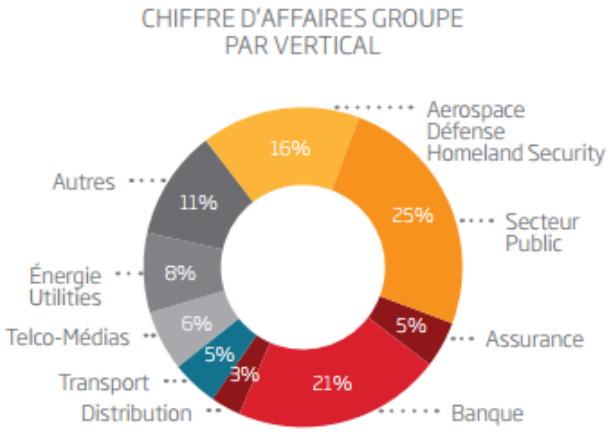

Figure 1.5. – Répartition des activités

1.4. Organisation

L'organisation du groupe Sopra Steria est articulée autour de différentes structures opérationnelles et fonctionnelles donc une structure permanente globale décrite sur la figure 1.6.

Comité exécutif

Le comité exécutif est composé du directeur général ainsi que de l'ensemble des directeurs adjoints et est en charge de piloter les projets et affaires les plus importantes du groupe. Il gère aussi l'organisation de la société dans son ensemble.

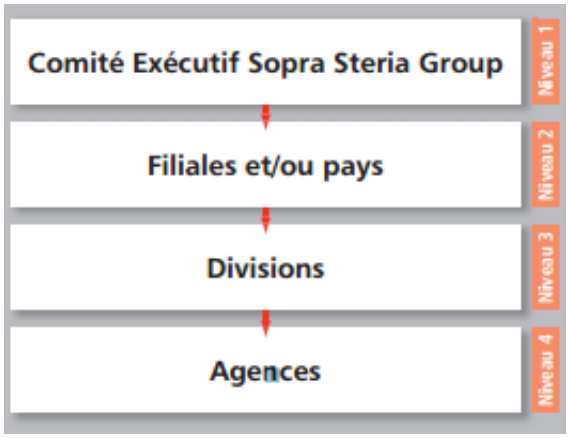

Figure 1.6. – Organisation du groupe

Filiales et/ou pays

Cette section désigne l'ensemble des grandes entités qui représente soit une partie métier dont nous avons parlé plus haut (conseil, BPS, etc...) soit une zone géographique pouvant faire référence à un pays complet. Ces parties sont alors ensuite découpées en un ensemble de divisions.

Divisions

Les divisions sont créées en fonction de la géographie ainsi que du secteur économique concerné (bancaire, transport, tertiaire etc...).

Agences

Enfin, les divisions sont constituées d'agences qui agissent de manière autonome concernant la gestion de leur budget, des ressources humaines ou encore du pilotage des projets. Dans mon cas, j'ai été assigné à l'agence 512 de la division Banque et Finance de Sopra Steria France. Ainsi, les missions sur lesquelles j'ai été assigné étaient cohérentes avec la division dont je dépendais et concernaient toutes deux des clients banque privée dont je vais parler plus en détails dans la partie suivante.

1.5. Clients

1.5.1. Neuflize OBC

La première mission sur laquelle j'ai été assigné a commencé le 13 février, d'abord à temps plein puis à mi-temps à partir du 29 mars (mercredi, jeudi) pour le compte du client Neuflize OBC. Ce dernier est une banque française privée, fruit de la fusion entre la banque *Neuflize Schlumberger Mallet Demachy* (NSMD) et la banque *Odier Bungener et Courvoisier* (OBC) en 2006. Neuflize OBC est devenue par la suite une filiale de la Banque Générale des Pays-Bas *ABN AMRO* dont le capital est détenu à 100% par l'état néerlandais. Mon stage s'est déroulé dans les locaux du siège social de Neuflize situé dans le 8ème arrondissement de Paris.

1.5.2. BP1818

La seconde mission sur laquelle j'ai été assigné a commencé le 29 mars à mi-temps (lundi, mardi et vendredi) pour le compte du client BP1818. Celui-ci est aussi une banque française privée, filiale de *Natixis*, une banque créée en 2006 et fait partie du groupe *Banque Populaire et Caisse d'Epargne* (BPCE) connu comme étant le deuxième acteur bancaire en France. Elle compte environ 500 collaborateurs et gère plus de 29 milliards d'euros à ce jour.

2. Présentation des sujets

Le secteur bancaire est actuellement en effervescence, impacté par la venue de nombreux nouveaux acteurs qui viennent boulverser les règles établies. En effet, le progrès dans le domaine de l'informatique a permis de fournir des services inédits pouvant répondre aux attentes de la nouvelle génération de client marquée par l'essor des technologies du numérique. Ainsi, de nouveaux acteurs sont apparus, les banques en ligne, c'est-à-dire des banques uniquement disponibles sur internet. Ces dernières ne possèdent pas de locaux physiques et n'ont que très peu de personnel. Néanmoins, elles sont capables de permettre à leurs clients de suivre l'état de leur compte bancaire ou patrimoine financier en temps réel. Il est possible de commander une carte bleue, un chéquier ou encore d'obtenir un rib sans avoir à se déplacer et bien d'autres services sont disponibles selon les banques et cela gratuitement. Les coûts de gestion des ressources humaines, d'organisation ou encore de matériels sont grandement réduits ce qui permet à ces nouvelles banques de pouvoir défier les grands groupes présents à l'échelle nationale. Ces derniers ont donc la nécessité de réagir afin de rester compétitif sur ce marché en pleine évolution.

On parle ainsi de transformation digitale des banques pour désigner la transition marquée par le processus de dématérialisation de l'économie en faisant appel aux technologies modernes. Neuflize OBC, comme nous l'avons vu dans la partie précédente, est une banque privée. Cette dernière est actuellement en cours de transformation afin de pouvoir répondre aux besoins de ses clients. Un site web a été créé permettant de proposer les services dits "banque au quotidien" qui regroupe les fonctionnalités classiques à savoir consultation de comptes, impression de rib ou encore réalisation d'une transaction bancaire. Cependant, les exigences sont toujours plus élevées, la génération actuelle étant toujours connectée via l'utilisation d'un smartphone, Neuflize s'est vu attribuer le besoin de produire une application mobile dans le but de permettre à ses clients de pouvoir accéder à leurs informations n'importe où et n'importe quand.

L'équipe de développement constituée par Sopra Steria était en charge de la réalisation de la partie *backend* de l'application, la partie *frontend* ayant été déléguée à l'équipe qui a conçu le site web. Neuflize a décidé d'exposer des API dans le but de permettre la réalisation d'échanges au sein de son SI et vers l'extérieur. Ainsi, une architecture multicouches a été mise en place avec notamment :

- Une couche d'API Management
- Une couche de microservices
- Une couche API Backend

La couche backend a pour objectif d'exposer des services unitaires développés par *Elcimaï Financial Software* (EFS), un éditeur spécialisé dans la dématérialisation des flux financiers, créateur de la solution **WeBank**. Cette solution logicielle propose des services grandement sécurisés permettant de mettre en place de l'authentification via l'utilisation de tokens, de la gestion de portefeuilles titres ou encore la signature numérique de transactions bancaires. Ces web services sont actuellement utilisés par le site internet de la banque et sont donc réemployés pour l'application mobile afin d'assurer une certaine cohérence.

La couche microservices est au coeur du sujet de ce stage. En effet, j'ai intégré l'équipe de projet en charge du développement de cette couche. Cependant, la phase de développement touchant à sa fin, j'ai principalement participé à la phase d'industrialisation via la réalisation de certains travaux indépendants (automatisation de tests fonctionnels, tests de charges ou encore dashboards pour le client). C'est en majeure partie pour cette raison que j'ai été assigné sur un second projet dont le développement venait tout récemment de commencer.

3. Neuflize OBC

3.1. Présentation du projet

Avant le début de ce projet, la banque Neuflize OBC (nous abrégerons maintenant NOBC) possédait déjà un site web mis à disposition de ses clients comme il est possible de l'observer sur le schéma figure 3.1. Celui-ci a été réalisé par NOBC qui a fait le choix de faire appel à EFS pour construire son backend. Comme nous l'avons déjà dit, EFS est un prestataire et éditeur de la solution *Webank* proposant de nombreux services grandement sécurisés permettant de faire de l'authentification, de la signature de transactions numériques etc... NOBC a donc développé un backend pour le site web pouvant communiquer avec ces services via le transfert de *fichiers* contenant les données à persister. EFS dispose de ses propres bases afin de stocker lesdits fichiers impliquant donc une communication bidirectionnelle entre le backend (au niveau *core banking*) et EFS afin que NOBC puisse maintenir ses bases de données internes à jour. Ainsi, une synchronisation des fichiers a lieu une fois par jour afin d'actualiser les bases et d'assurer la cohérence.

Cependant, dans le cadre d'une application mobile, il n'était pas possible d'interroger directement EFS de la même manière que pour le web. En effet, pour des raisons de performances, les conventions préconisent d'essayer de faire correspondre un écran avec un seul service. Cela permet d'améliorer la fluidité lors de la navigation sur l'application et de réduire considérablement les temps de chargement de chaque page. De plus, NOBC souhaitait se séparer à terme d'EFS c'est pourquoi une couche d'*API microservices* a été développée afin de réaliser la composition des services EFS afin de répondre à ce besoin. Pour cela, EFS s'est vu attribuer la tâche de mettre en place une *surcouche* afin d'exposer ses services et transmettre les réponses via le format de données JSON et non plus directement via du SQL. La communication entre EFS et l'API microservices étant établie, il fallait maintenant sécuriser l'accès à la couche microservices. Afin de ne pas réinventer la roue et d'obtenir un résultat optimal en fonction du coût du projet, il a été décidé de mettre en place une couche *API Gateway* basée sur l'API d'Axway, une filiale de Sopra Steria. Cette couche permet entre autre d'assurer la sécurité en agissant comme pare-feu et proxy mais aussi comme routeur. Elle sera décrite plus en détails dans la partie 3.2.1.

Enfin, une fois la gateway mise en place, cette dernière est devenue le point d'entrée du projet. Elle reçoit les requêtes depuis la *Rest Layer* mise en place par l'équipe chargée du développement du front end de l'application, à savoir PBI, qui est notre client et qui consomme nos microservices.

L'API microservices est une API REST utilisant la stack Netflix OSS qui permet d'intégrer les patterns classiques aux application distribuées et dont les composants sont intégrés via Spring Cloud. Les microservices proposés par cette couche sont les suivants :

— Account-service : gérant les informations liées aux comptes des utilisateurs
— Profile-service : gérant les informations liées aux profils des utilisateurs
— Transaction-service : gérant les informations liées aux transactions bancaires

Ces microservices permettent de fournir de nombreuses fonctionnalités de consultation et de transaction en renvoyant des réponses au format JSON. La liste des web services associés au microservice correspondant est disponible en annexe A.1. A partir de maintenant, il sera possible de se référer à cette annexe lorsque qu'un service sera mentionné afin de connaître l'utilité de celui-ci.

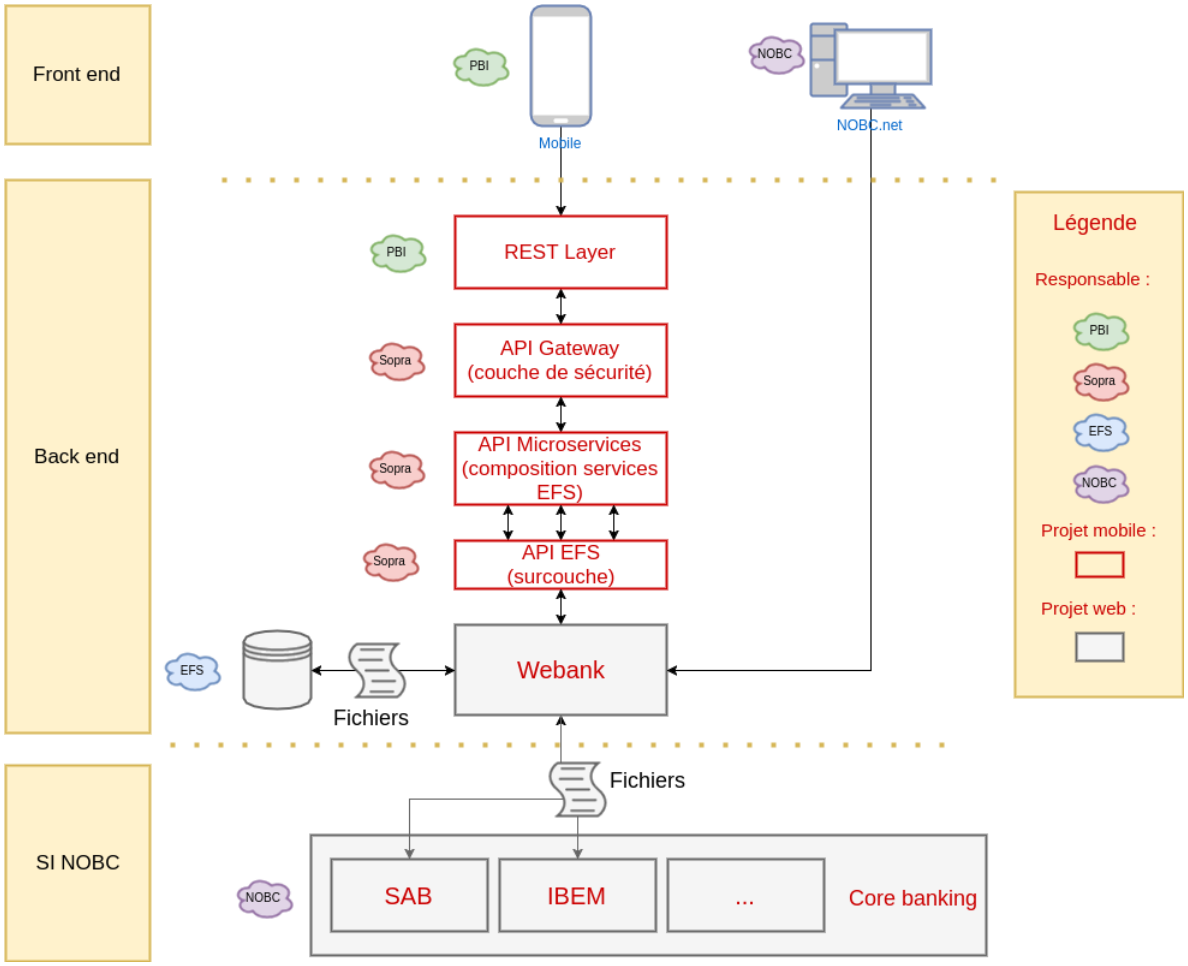

Figure 3.1. – Présentation du projet et des acteurs

3.2. Architecture du projet mobile

Dans cette partie nous allons présenter plus en détails l'architecture globale du projet d'application mobile de Neuflize OBC. Comme nous l'avons vu précédemment, ce projet est basé sur une architecture multicouches dont la structure est représentée dans sa globalité en annexe A.2. Nous allons maintenant décrire chacune des couches afin de comprendre le fonctionnement du projet. Cependant, les technologies employées étant nombreuses, il serait peu pertinent de toutes les expliciter, ainsi seul l'essentiel en lien avec le stage sera ici décrit. Toutefois, il est possible de retrouver toutes les briques en annexe.

3.2.1. Axway API Gateway

Deux instances de l'API Gateway d'Axway ont été installées en mode « actif/actif » afin d'assurer la mise en place de la couche *security* et celle de la couche *management*. La répartition de charge est gérée par l'instance positionnée en amont.

L'API Gateway de la couche security est le serveur traitant les appels API. Cette dernière est en charge de la sécurité applicative des appels vers la couche API Management. Elle est positionnée dans le tier 1, reçoit les appels « HTTPS » et a principalement pour objectif d'effectuer les actions suivantes :

— Vérifier la validité des certificats partenaires
— Filtrer les requêtes entrantes
— Agir comme un pare-feu applicatif afin de vérifier le contenu des messages REST

— Répartir la charge vers les composants en aval
— Protéger le SI en limitant le nombre d'appels API via un mécanisme de régulation du traffic et en limitant le nombre d'appels au SI via un mécanisme de cache

La couche API management, dans le tier 2, permet de configurer et d'exposer les API. Elle assure les fonctionnalités suivantes :

— Publier et sécuriser les API
— Gérer le cycle de vie des API
— Gérer l'authentification et les habilitations (développeurs et administrateurs API)
— Embarquer les développeurs d'applications consommatrices d'API
— Auditer, suivre la consommation des API, gérer les quotas
— Assurer la haute disponibilité

Une interface web avait aussi été mise à notre disposition par Axway, nous permettant de traquer toutes les requêtes effectuées.

Afin d'assurer la sécurité l'API management utilise *OAuth2* ainsi que *JWT*. OAuth2 est un framework d'authentification permettant d'authentifier plusieurs applications différentes. JWT est un protocole d'authentification permettant de créer et valider des *tokens* (ou jetons) de sécurité. Ces tokens sont ensuite utilisés pour limiter l'accès des utilisateurs aux API. Pour chaque utilisateur qui se connecte, la Rest Layer envoie une assertion SAML. Si cette assertion est vérifiée par l'API management un token est généré contenant les informations permettant à un utilisateur d'accéder aux ressources protégées. Ainsi, toute les requêtes émises vers l'api microservices doivent contenir un token permettant l'authentification sans quoi elles seront rejetées (erreur 401 : Unauthorized). J'ai simplifié ici la procédure et limité les informations à ce qui est nécessaire pour la compréhension du rapport. En effet, celle-ci est extrêmement complexe et dépasse ce qui a été effectué dans le cadre du stage.

3.2.2. Microservices

Avant d'aller plus loin, nous allons expliciter ce qu'est une architecture microservices [1] afin de pouvoir comprendre la structure des API. Il s'agit d'un paradigme d'architecture qui jouit actuellement d'une grande popularité aux dépends de celles plus classiques (N-tiers, SOA...), inventée afin de répondre aux problématiques soulevées par les projets de grande ampleur.

Cette approche consiste à développer une application sous forme d'un ensemble de services dont la granularité correspond à une fonctionnalité élémentaire en terme métier. Chacun de ces services doit posséder son propre contexte d'exécution et ainsi être testable et déployable indépendemment en favorisant un couplage le plus faible possible. Ils peuvent être écrit dans des langages différents et communiquer entre eux via, par exemple, le protocole HTTP et la mise en place d'une API REST, ce qui est le cas pour ce projet. On parle alors de microservices, terme qui s'oppose aux applications plus classique que l'on dit monolithiques.

Les applications d'entreprises classiques sont très souvent construites sur une architecture trois tiers constituées de trois parties majeures :
— Une interface client permettant la présentation des données
— Un serveur contenant la logique métier et effectuant le traitements des données
— Une couche d'accès aux données permettant de gérer les données persistantes

L'ensemble des services est contenu dans la même application et ces derniers sont donc exécutés dans un même processus. Ainsi, le moindre changement nécessite de rebuild et redéployer l'application entièrement. La scalabilité horizontale (par exemple un ajout de serveur) en est impactée. En effet, l'application entière doit être migrée si l'on souhaite changer de matériels afin d'améliorer les performances.

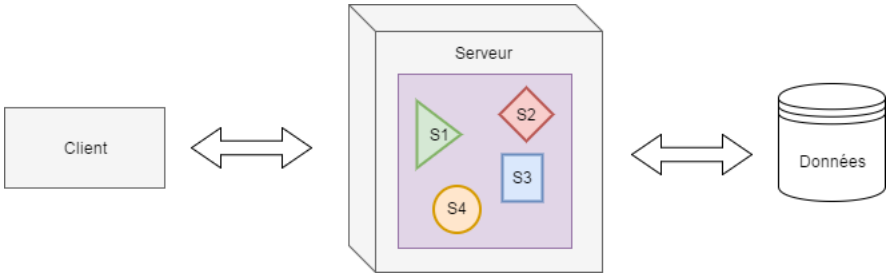

Figure 3.2. – Architecture trois tiers monolithique

Si un certain module est plus lent, il n'est pas possible de le déplacer indépendemment afin d'améliorer son exécution, il faut répliquer le monolithe entier tandis que du côté des microservices il est possible de répliquer un service en particulier et d'en redéployer un sans avoir à redéployer tout l'ensemble. De plus, dans les gros projets, la quantité de code a tendance à augmenter rapidement impliquant une hausse de la compléxité et rendant ainsi difficile l'ajout de nouvelles fonctionnalités. Le couplage entre ces dernières devient fort et les nombreux effets de bords résultant de chaque modifications rendent alors l'application moins fiable, limitant les perspectives d'évolution.

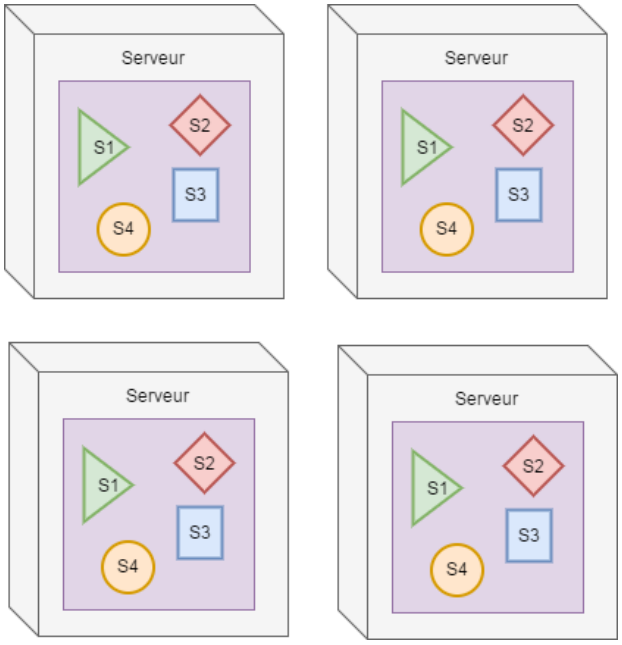

Figure 3.3. – Scalabilité horizontale d'une application monolithique

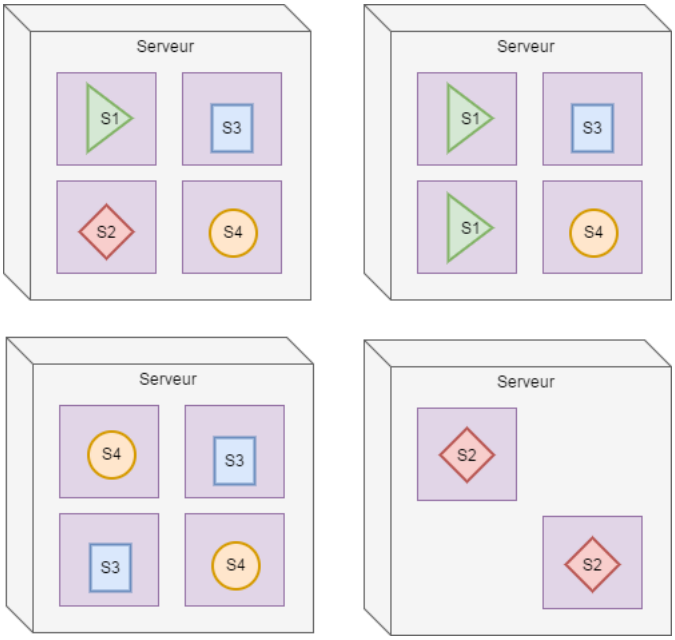

Figure 3.4. – Scalabilité horizontale des microservices

Dans notre cas, l'objectif de la couche microservices , située dans le tiers 2, est de réaliser la composition des services métiers exposés par EFS dans le but d'exposer les données pour les applications ou les partenaires (comme PBI dont nous avons parlé dans la partie 3.1) qui viendront les consommer. Cette dernière est constituée des éléments présents sur la figure 3.5. Les services EFS sont exposés via une surcouche API, située dans le tiers 2.

Spring Boot Stack est la brique applicative hébergeant les microservices. C'est dans cette dernière que la composition de services EFS est réalisée. Spring boot permet d'utiliser le framework java *Spring* en simplifiant grandement la configuration, le déploiement ou encore la sécurité et en créant des applications cloud-ready. Cette brique prend aussi en charge l'implémentation des principaux patterns à savoir :

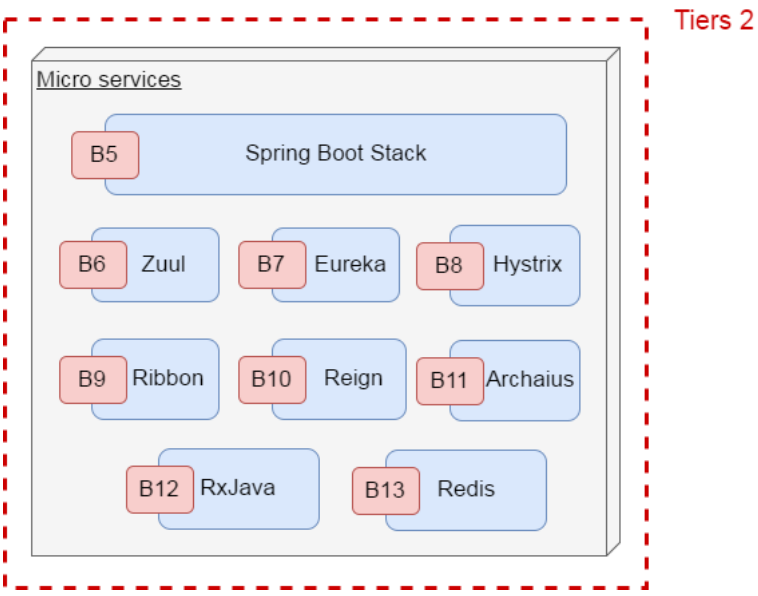

Figure 3.5. – Couche microservices

— Circuit breaker : Capacité du système à être tolérant à la panne. En cas d'erreur successive lors de l'appel d'un sous-composant, le circuit d'appel est coupé « temporairement » en adoptant un comportement par défaut. L'implémentation qui a été choisie pour ce pattern est Hystrix de la stack *Netflix OSS* [5] permettant de contrôler la latence et les erreurs dues à des appels réseaux. L'idée essentielle est d'empêcher les erreurs en cascade dans un environnement distribué. Hystrix permet de *fail-fast* mais de se rétablir rapidement créant ainsi une architecture tolérante aux erreurs capable de se remettre de manière autonome (on parle de self-heal). Ainsi, dès sa conception, le système prévoit les pannes.
— Feature toggle : Le principe est d'avoir une branche de développement et de déployer en production en continu. Ensuite, l'activation d'une « feature » est pilotée par le business. Cela permet aussi d'activer une fonctionnalité en fonction d'une population ou une stratégie particulière.

Le serveur d'annuaire, essentielle à une architceture distribuée, permet la détection automatique des instances déployées. Les instances des applications sont accédées via leur nom (par exemple account-service) plutôt que par leurs adresses physiques/IPs. Les applications n'ont plus besoin de connaitre les adresses des instances. L'implémentation de l'annuaire de service est Eureka de la stack *Netflix OSS* [4]. Les applications clientes peuvent s'enregistrer sur Eureka, via une annotation, qui fournira des metadatas telles que l'URL, le port ou encore le fil de vie (healthcheck) des instances. Eureka reçoit des messages dits "heartbeat" provenant de ces applications, si aucun message n'est reçu, en fonction d'un temps configurable, il supprimera l'instance.

Ensuite, le point d'entrée unique de l'architecture microservices est sa gateway fournissant des services de routage dynamique, surveillance, résilience et sécurité. L'implémentation choisie est Zuul de la stack *Netflix OSS* [3]. L'affichage d'une page web ou mobile peut nécessiter l'appel à une dizaine de microservices différents. Il n'est pas envisageable pour l'application cliente de connaitre l'ensemble des adresses physiques des microservices. Pour répondre à cette problématique, la gateway devient la seule adresse à connaitre pour les applications clientes. Zuul est aussi utilisé pour router les requêtes vers les services adéquats. Celui-ci retrouve les adresses des services automatiquement en interrogeant Eureka. Cependant, les services EFS et d'authentification (Axway gateway) sont paramétrés manuellement puisqu'ils ne sont pas enregistrés sur Eureka. La charge sera ensuite répartie grâce à un autre outils de Netflix, Ribbon, qui fourni des fonctionnalités de load-balancing permettant de distribuer la charge de travail.

3.3. Tests fonctionnels

Nous allons, dans cette partie, décrire les travaux concernant la mise en place d'une procédure complète permettant de gérer et réaliser des tests fonctionnels de manière automatique et fournissant des rapports complets pouvant être remis au client.

3.3.1. Enjeux et spécifications

Comme nous l'avons vu précédemment, les services de nos API rest sont destinés à être consommés par PBI, en charge du développement de la partie frontend de l'application mobile. Ainsi, nous étions souvent en contact avec ces derniers afin de prendre connaissance des différentes anomalies liées à nos services et de pouvoir répondre à l'évolution de leur besoins. Après correction de celles-ci, il était fréquent que nous soyions amenés à livrer la nouvelle version des services. Ces livraisons permettaient à PBI de pouvoir continuer le développement de l'application dans les meilleures conditions possibles. Elles s'effectuaient sur deux environnements différents à savoir *homo3* et *rgb* comme il est possible de l'observer sur la figure 3.6.

L'environnement homo3 est un serveur d'homologation sur lequel était effectué les recettes avec le client. Celui-ci permet de réaliser des tests afin de s'assurer que le produit est conforme aux spécifications. L'environnement rgb est un serveur de pré-production dont la structure, contrairement à homo3, est identique à celui de la production. Ce serveur permet de réaliser un béta test du produit ainsi que des tests de charge par le client dans des conditions réelles afin de déceler les derniers bug potentiels avant la mise en production. De son côté, PBI, possède la même structure avec un serveur de recette nommé *ST* et une préprod nommée *ET* connectés à nos environnements. Ainsi, nous partagions les mêmes données pour réaliser nos tests (identifiants utilisateur, comptes bancaires, portefeuilles, positions, transactions...) Concernant les services EFS nous étions toujours sur leur production ou sur des mocks en attendant la mise en production des services demandés.

Cependant, avant de procéder à la livraison des services sur ces environnements, il est nécessaire d'effectuer une batterie de tests fonctionnels permettant de vérifier que le comportement des API est conforme aux spécifications. Ces tests sont essentiels à la satisfaction client et permettent de gagner un temps précieux en évitant d'attendre les retours avant d'identifier les possibles anomalies. Néanmoins, dans notre cas, peu de ces tests avaient été mis en place et il n'existait aucune procédure à suivre. Les cas de tests était rédiger sous Word et le résultat de leur exécution était consigné dans des fichiers excel, peu lisibles, contenant peu d'informations et difficilement traçables. De cette manière, il est difficile de normaliser l'écriture des tests et la création d'un rapport est chronophage (créer les formules excel, etc...) pour un résultat qui ne sera pas exhaustif. En outre, il est impossible d'avoir une vue d'ensemble sur l'évolution des tests au cours du temps et est compliqué de suivre l'état d'une spécification particulière à différentes dates données. Par ailleurs, le client lui-même a souhaité un autre format plus rigoureux permettant de vérifier le comportement des services dans leur intégralités et de pallier aux inconvénients que nous avons cité.

Ainsi, j'ai été chargé de mettre en place une procédure pouvant répondre à ce besoin. Cette dernière devait :

- définir les cas de tests
- fournir des rapports d'exécution de tests détaillés
- nécessiter peu de développement, il était en effet initule de réinventer la roue
- être gratuite
- être accessible à tout moment à n'importe quel membre de l'équipe de développement qui pourrait être amené à effectuer une livraison

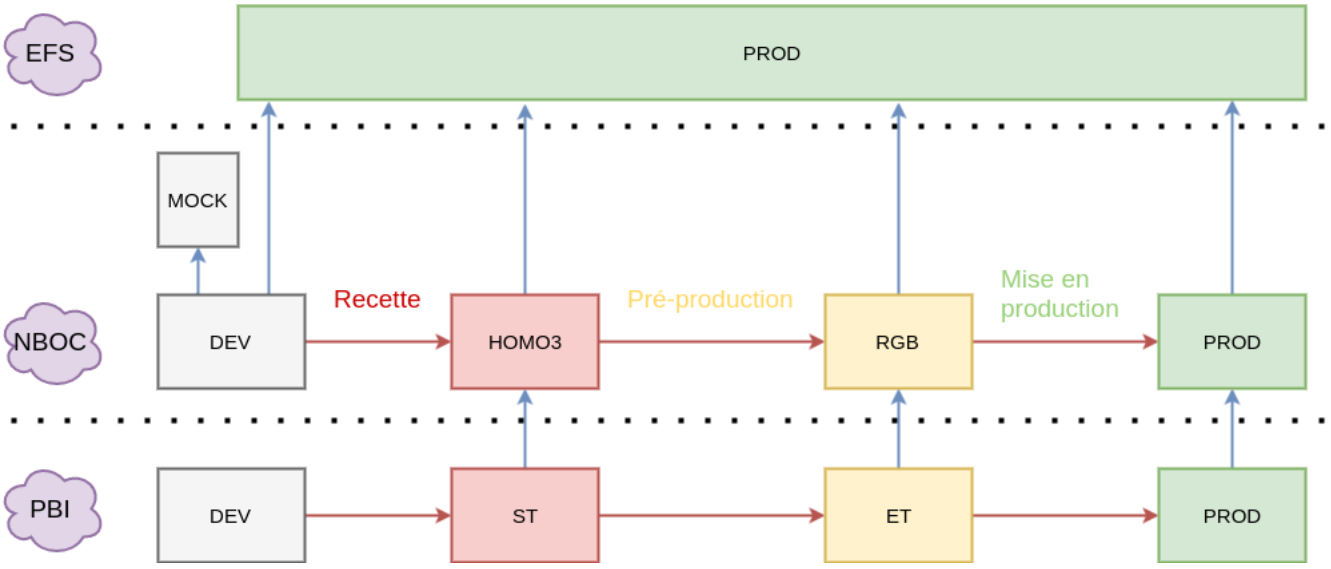

Figure 3.6. – Environnements

3.3.2. TestLink : Un gestionnaire de tests

Dans le but de répondre aux besoins explicités précedemment, j'ai décidé d'avoir recours à un gestionnaire de tests sous forme d'une application web. En effet, celle-ci pourrait être mise en place sur un seveur interne et rendue disponible pour toute l'équipe, favorisant ainsi le partage d'information et permettant de centraliser toutes les données concernant la réalisation des tests fonctionnels, assurant ainsi un versionning cohérent. De plus, cela répond à la problématique de mise en place rapide et de maintenabilité efficace (pas de mise à jour à gérer sur tous les postes, etc...).

Après avoir mené plusieurs recherches, j'ai pu constater qu'il existait différents gestionnaires open-source concurrents sur le marché répondant à nos exigences : *Salomé TMF*, *TestLink* ou encore *Squash TM*. En effet, ces derniers nous offrent tous la possibilité de créer des tests, de les lier aux exigences client ou encore de créer des campagnes de tests puis d'exporter les résultats sous forme de rapport détaillé. Ces outils proposant des fonctionnalités très proches, j'ai décidé d'en choisir un disposant d'une grande flexibilité ainsi que d'une communauté active afin de pouvoir faciliter l'adaptation à nos cas de tests. En effet, les gestionnaires génèrent des rapports regroupant des informations telles que la description des tests, leur temps d'exécution ou leur status. Cependant, dans notre cas, nous souhaitions pouvoir inclure pour chacun des tests des informations supplémentaires telles que le nom du service auquel appartient la fonctionnalité testée, l'identifiant de l'utilisateur utilisé pour le test, le numéro de compte bancaire utilisé et de manière générale, tous les paramètres utilisés pour réaliser la reqûetes testées. PBI partageant les mêmes données que nous sur les environnements d'homologatoin et de pré-production, il leur était alors possible de vérifier de leur côté que les tests passent effectivement. De plus, lorsqu'ils remarquaient une anomalie, ils pouvaient nous transmettre les paramètres qu'ils avaient utilisé afin que nous puissions reproduire celle-ci chez nous.

Ainsi, j'ai décidé d'utiliser l'outil *TestLink* dont la communauté avait mis à disposition de tous des templates permettant de modifier le code source afin de customiser la génération des rapports de tests. Celui-ci est une application web développée en PHP et utilisant le système de gestion de base de données MySQL. Il permet de centraliser toute la gestion des tests fonctionnels du projet en les organisant par le biais des structures présentées dans le tableau 3.1.

Cas de test	Test fonctionnel définissant un scénario spécifique
Suite de tests	Collection de cas de test validant une même fonctionnalité
Plan de tests	Collection de suite de tests contenant toutes les informations telles que la portée, les étapes, la version etc... Un plan est exécuté pour un build particulier
Build	Une release spécifique des APIs testées

Table 3.1. – Structures fournies par TestLink

Cet outil présente de nombreux avantages qui m'ont conforté dans mon choix :
— Campagnes de tests versionnées dont l'historique est enregistré en base de données.
— Export et import de cas de test et de leur résultats
— Connection avec *Mantis*, un tracker de bug utilisé dans notre projet
— Gestion de rôles sur les tests (qui effectues le test, qui valide etc...)
— Rapport complet dans différents formats
— Accessible à toute l'équipe n'importe quand
— Simplicité d'utilisation et de mise en place

Ensuite, avant de procéder à la création des cas de test sur TestLink, j'ai commencé par définir la structure du futur plan de test qui serait exécuté avant chaque livraison. Ainsi, j'ai décidé de séparer l'ensemble des tests en deux grandes familles : ceux concernant les fonctionnalités de consultation et ceux concernant les fonctionnalités de transaction, ce qui m'amena à la création de deux suites de tests. Après cela, j'ai créé autant de suites de tests qu'il y avait de fonctionnalités décrites dans l'annexe A.2. Il est possible d'observer sur la figure 3.7 l'organisation d'un plan de test type. Afin de garder le même formalisme tout le long de la réalisation des plans de tests et pour assurer une certaine cohérence, j'ai décidé de mettre en place plusieurs conventions définissant une stratégie de test :

Cas de test

Les cas de tests doivent avoir un nom de la forme [id]-[titre] où
— id désigne un ID unique permettant de les identifier rapidement et de faciliter leur organisation. Celui-ci est *NOBC-API-XX*, où XX représente le numéro du test.
— titre désigne de manière clair et concise l'objectif du test

De plus, chaque cas de test possède en attribut un numéro de version de la forme *vX* où X est incrémenter de 1 chaque fois que le cas de test est modifié.

Plan de test

Les plans de tests doivent avoir un nom de la forme [scope]-[environnement]-[version] où
— scope désigne la portée du plan de test : "complete" pour tous les tests, "transaction service" pour les tests du service de transaction, "transaction overview" pour les tests de la fonctionnalité transaction overview etc...
— environnement désigne le serveur sur lequel sont effectués les tests : homo3 ou rgb
— version est de la forme vX.Y.Z où
  — X est incrémenté de 1 lorsqu'un nouveau cas de test est ajouté ou supprimé du plan
  — Y est incrémenté de 1 lorsqu'un cas de test existant du plan a été modifié
  — Z est incrémenté de 1 à chaque exécution du plan

Build

Les builds doivent avoir un nom de la forme : [version] où
— version désigne la version de l'API microservices testée

La stratégie de test étant définie, il fallait maintenant déterminer quelles informations devaient être transmises au sein des rapports de tests. Les gestionnaires de tests proposent de remplir des formulaires afin consigner le résultat des tests une fois ceux-ci effectués, ce qui servira par la suite à générer un rapport. Cependant, ces derniers sont plutôt génériques et ne permettent pas de renseigner des données spécifiques à un projet particulier. Comme nous l'avons dit plus haut, nous souhaitions être en mesure de passer des paramètres supplémentaires afin de faciliter nos échanges avec PBI, comme :

— url et paramètres utilisés pour la requête
— service testé
— identifiants utilisés

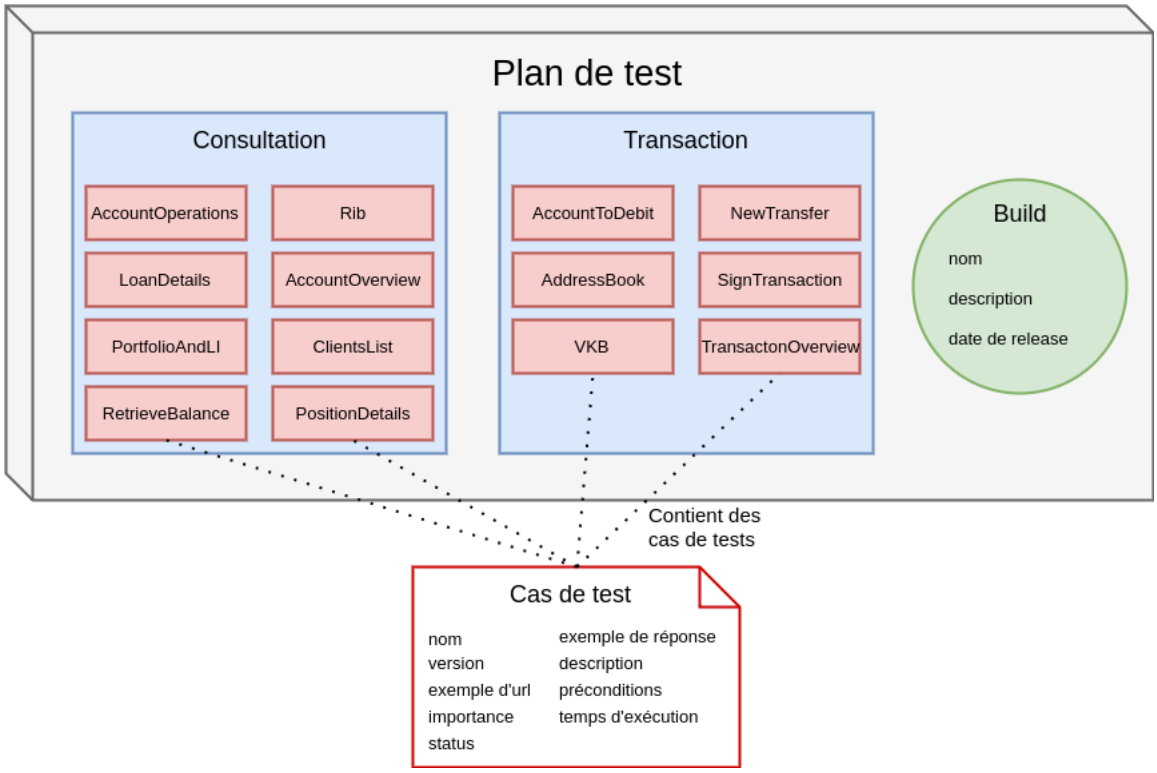

Figure 3.7. – Configuration d'un plan de test type

Ainsi, j'ai décidé de modifier le code source de TestLink afin de rajouter les champs dont nous avions besoin aux formulaires de tests. J'ai réalisé cette modification en deux étapes dont la première consistait à ajouter les champs aux formulaires puis à gérer la partie front en PHP. La seconde concernait la récupération des données et leur sauvegarde en base de données, ce qui a impliqué la création de nouvelles requêtes SQL. Une fois les champs mis en place, j'ai créé un cas de test puis générer un premier rapport au format PDF en guide de POC (proof of concept). Celui-ci ayant été jugé satisfaisant, j'ai dû étudier l'ensemble des spécifications du projet concernant chacun des services mis en place afin de procéder à l'écriture de tous les cas de tests. Cela m'a permis de mieux comprendre le besoin du client, d'avoir une bien meilleure vision sur le projet dans sa globalité en m'apportant des informations sur l'utilité de chaque service et de pouvoir me former sur le projet en restant productif.

Ces cas de tests permettaient de vérifier que les réponses des requêtes émises vers l'API microservices étaient en accord avec les attentes de PBI. Par exemple, les réponses étant au format JSON, ils permettaient la vérification de la présence de tous les champs obligatoires, la cohérence des valeurs des champs (par exemple un compte de type emprunt aura un champ "montant emprunté" alors qu'un compte épargne n'en aura pas) ou encore la vérification de la cohérence des données de notre backend avec celles de l'application web.

3.3.3. Rapports de tests obtenus

Une fois tous les cas de tests rédigés, j'ai procédé à la réalisation d'une campagne de tests complète aboutissant à la génération d'un rapport. Afin de procéder à cela, pour chacun des tests j'ai utilisé l'outil *Postman* qui est une plateforme proposant une interface graphique facilitant la construction de requêtes. Cet outil est conçu pour faciliter le développement des APIs en permettant de les intéroger de manière très rapide et simple sans avoir à développer un client pour les consommer. Dans l'optique de mettre à profit les tests effectués, nous avons connecté TestLink à Mantis, une application web permettant d'assurer le suivi des anomalies dans laquelle il est possible d'ouvrir des tickets concernant des bugs qui seront alors pris en charge par les développeurs jusqu'à leur clôture. Ainsi, lorsqu'un test échouait il était possible, d'un simple clique sur TestLink, de générer automatiquement un ticket sur Mantis avec toutes les informations rajoutées précédemment. Les développeurs avaient donc toutes les données (url, service etc...) pour reproduire l'anomalie en locale et la corriger.

TestLink permet de personnaliser la génération des rapports de tests après l'exécution d'un plan complet. De plus, il est possible de générer le rapport dans différents formats. Nous avons donc décidé que chaque livraison comporterait un rapport complet contenant toutes les informations au format PDF. Cependant celui-ci pouvant être très conséquent, nous avons choisi de l'accompagner d'un rapport léger généré au format excel ne contenant que le nom des tests et leur statut (succès, echec, bloqué) avec un code couleur permettant a PBI de rapidement repérer les tests en échec et leur nombre. Ces derniers ont déclaré être très satisfait des nouveaux rapports fournis, c'est pourquoi il a été décidé que l'exécution des tests fonctionnels sur la plateforme TestLink serrait obligatoire avant chaque livraison. En attendant l'attribution d'un serveur pour héberger le gestionnaire de test, celui-ci a été mis en place sur le serveur personnel d'un des architecte du projet afin de le rendre accessible à toute l'équipe.

La réalisation de ces tests et leur exécution m'a permis de relever certains écarts entre le produit réalisé et les spécifications. Par exemple, le service LoanDetails ne fournissait pas, dans sa réponse, certains champs demandé par le client. C'est pourquoi, après avoir centralisé tous les écarts que j'avais relevé, j'ai organisé une réunion avec mon chef de projet ainsi que certains développeurs dans le but de déterminer lesquels pourraient être corriger et lesquels ne le pourraient pas, nécessitant de prendre contact avec le client afin de clarifier la situation. Au terme de cette réunion, j'ai pu créer une nouvelle version des spécifications afin de les mettre à jour puis j'ai fait part de ces modifications à PBI à travers des mails rédigés en anglais. Ensuite, la mise en place de ces outils a permi à l'équipe de pouvoir effectuer les livraisons dans de meilleurs conditions puisque les anomalies pouvaient être détectées avant d'attendre les retours du client. De plus, tous les tests étaient centralisés, organisés et pouvaient être exécutés en parallèle par différentes personnes ce qui permettait un gain de temps à la fois sur l'exécution des tests mais aussi sur leur gestion (archivage, formalisme, perte de documents etc...).

Néanmoins, afin d'exécuter les tests, les développeurs utilsaient Postman pour construire et envoyer les requêtes. Si cet outil est extrèmement utile pour développer un nouveau service ou tester un nouveau endpoint lors du développement, il n'est pas conçu pour exécuter un grand nombre de requête les unes après les autres à des fins de tests. En effet, comme nous l'avons expliqué dans la partie 3.2.1 les requêtes doivent posséder un token d'authentification dans leur header pour espérer passer la gateway et atteindre la couche microservices. Or, pour cela, il est nécessaire d'envoyer, toujours avec Postman, une requête de génération de ce token à l'API gateway. Après cela, il faut se connecter sur l'interface web fourni par Axway, retrouver la requête ainsi que sa réponse et copier le token. Cependant, pour accéder à cette interface il faut d'abord être en mesure d'accéder à l'environnement testé, Homo3 ou RGB, qui, rappelons le, n'ont pas les mêmes instances de l'API Gateway et ne sont pas accessibles de l'extérieur. Pour cela, il faut se connecter en RDP sur TSE puis ensuite utiliser les bonnes adresses et identifiants pour accéder à l'interface désirée. De retour sur Postman, il faut coller le token dans le header de la requête que l'on souhaitais tester. Cette démarche est illustrée sur l'annexe ?? qui décrit la procédure d'authentification gérée par la gateway. Et il faut répéter cette démarche **à chaque fois que le token expire**, ce qui ne pose pas de problèmes lorsque l'on souhaite envoyer une requête pour tester

son code mais devient rapidement très fastidieux lorsque l'on a des centaines de requêtes à envoyer pour exécuter tous les tests. Un exemple classique serait de tester le service LoanDetails. Pour cela il faut générer un token pour un abonné. Ensuite, on appelle le service ClientList pour afficher les produits de l'abonné puis on regarde si celui-ci possède un produit de type *loan* pour récupérer son id (crédit en français). Enfin, on appelle le service LoanDetails avec l'id du loan obtenu précédemment. Si le client ne possède pas de loan il faut recommencer le processus avec divers abonnés jusqu'à trouver ce que l'on cherche. Maintenant, certains services nécessitent d'en appeler trois autres différents ce qui implique de faire de nombreuses combinaisons avant de tomber sur celle qui nous permet de réaliser le test. Il résulte de cela une perte considérable de temps qui aurrait pu être mis au profit du développement, de l'amélioration des points jugés sensibles ou encore de la correction des anomalies. En effet, il n'était pas rare que l'ensemble des tests puissent occuper une personne presque **une demi journée**, ce qui se révèle être énorme sur des sprints de deux ou trois semaines. Il fallait donc trouver le moyen de conserver la procédure de test et la génération de rapports tout en réduisant drastiquement le temps que cela demandait, c'est pourquoi nous avons décidé d'automatiser les tests fonctionnels.

3.3.4. Automatisation des tests fonctionnels

La fin de la phase de développement approchant, l'équipe avait fait ressentir le besoin de procéder à la réalisation de tests de charge afin de vérifier la capacité des APIs à soutenir le trafic attendu. Pour cela, l'un des outils open source les plus performant et rapide à mettre en place du marché est **Apache JMeter**. Ainsi, afin de centraliser tous les tests et de gagner du temps sur la formation aux outils et leur installation, j'ai décidé d'utiliser JMeter pour aussi automatiser les tests fonctionnels.

Ce logiciel, développé en java avec l'API Swing par la fondation Apache, permet de réaliser aussi bien des tests de performance que de charge ou encore fonctionnel compatibles avec un grand nombre de protocoles et technologies. Il permet de créer des requêtes et de les exécuter de manière automatique sur des serveurs web, des base de données via JDBC, sur des LDAP et bien d'autres. Celui-ci exporte le résultat des tests au format XML ce qui permet aisément de les réimporter sous TestLink. En effet, ce dernier nous offre la possibilité d'importer les résultats de tests via des fichiers aussi au format XML. Ainsi, il était possible de jouer les tests automatiquement dans un premier temps sur JMeter puis de générer, dans un second temps, un rapport à destination du client via TestLink. En outre, JMeter met à notre disposition de nombreux avantages :

— Logiciel open source disponible gratuitement
— Interface en Swing très intuitive
— Multithreading permettant de simuler la concurrence des requêtes lors des tests de charges via la mise en place de différents groupes de threads représentant des utilisateurs fictifs
— Possibilité d'installer de nombreux plugins développés par une communauté active afin de customiser les tests si besoin est

Il propose de construire les plans de tests de manière interactive en ayant recours à des composants préconçus qui interagiront entre eux. Il en existe différents types pouvant être utilisés comme nous le montre le tableau figure 3.2 :

Les tests étant nombreux, j'ai décidé de les classer par service afin de rapidement en retrouver un en particulier et de pouvoir les jouer selon le microservice désiré. Cela s'est avéré utile par la suite, lorsqu'un service était défaillant, pour pouvoir facilement lancer les tests le concernant, étudier les réponses obtenues et cibler l'origine des anomalies. Il était, en effet, beaucoup plus rapide d'avoir recours à JMeter qui permettait d'envoyer de nombreuses requêtes avec différents paramètres plutôt que d'exécuter des requêtes manuellement via Postman jusqu'à réussir à reproduire un bug pour l'analyser.

Après cela, j'ai défini la structure du plan de test général qui devais contenir tous les tests fonctionnels en choisissant les composants qui allaient le constituer puis en procédant au paramétrage des différents éléments de configuration. La capture d'écran de JMeter figure 3.8 montre la structure que

Composant	Description
Moteur d'utilisateurs	Il s'agit de l'élément qui définira le niveau de la charge à appliquer (nombre d'utilisateurs, d'itération, temps de montée en charge)
Contrôleur logique	Permet de structurer les tests
Configuration	Permet de définir les configurations communes à plusieurs éléments
Compteur de temps	Permet de gérer le temps d'attente avant l'exécution d'un composant
Echantillons	Réalise les opérations de tests (ici des requêtes HTTP)
Pré-processeurs	Agi avant les échantillons pour réaliser des opérations
Post-processeurs	Agi sur le résultat des échantillons
Récepteurs	Traite et formatent le résultat des tests
Assertions	Permet de vérifier le résultat des échantillons

Table 3.2. – Composants JMeter

j'ai établi pour le plan de test final et chacun des services. Cette structure est explicitée sous cette figure via une explication du détail de chaque élément employé ainsi que ses intéractions avec les autres.

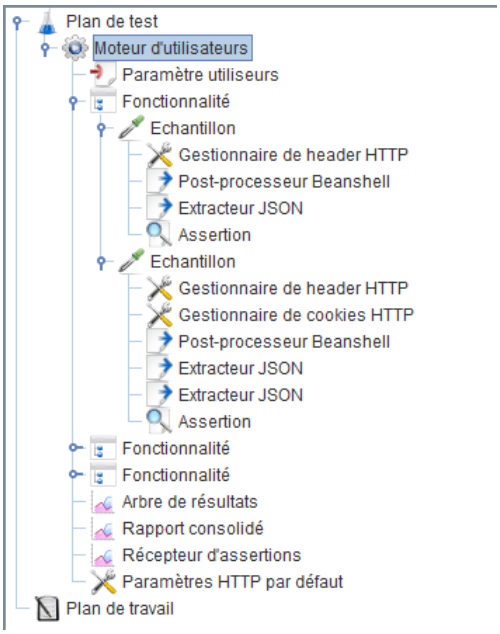

Figure 3.8. – Plan de test JMeter

1 - Moteur d'utilisateur Le moteur d'utilisateurs est l'élément englobant tous les composants, permettant de définir le niveau de charge affecté au plan de tests. Dans notre cas, ces derniers étant fonctionnels, le nombre de threads (utilisateurs) ainsi que le nombre d'itération des requêtes est de 1. De plus, le temps de montée en charge est laissé par défaut afin que les tests soient exécutés dans les meilleures conditions possibles.

2 - Contrôleur logique Les contrôleurs logiques permettent de définir la structure du plan de tests. Ici, il y en a un par service testé. Ils contiennent tous les tests d'un service particulier, les opérations à effectuer pour les mener à bien et les assertions.

Par exemple, il y a le controleur NewTransfer regroupant les tests du service du même nom permettant de réaliser des transactions bancaires.

2.1 - Echantillon Les échantillons représentent les requêtes HTTP qui seront exécutées par JMeter. Afin de maintenir la cohérence avec TestLink, toutes les requêtes testées portent un nom de la forme NOBC-API-XX, ce qui correspond aux ID des cas de tests défini dans TestLink. Celles-ci sont alors vérifiées grâce à des assertions. Celles dont le nom n'est pas de cette forme ne sont pas vérifiées et n'ont pas d'assertions, elles ne sont exécutées que pour remplir les préconditions nécessaires à l'exécution des tests.

Par exemple, pour pouvoir appeler le service NewTransfer, il faut l'identifiant d'un compte débiteur et créditeur, et comme pour tout service il faut un token d'authentification. Ainsi, il faut donc au minimum appeler les services AccountToDebit, Addessbook et TokenInfo, extraire les informations souhaitées puis les utiliser en paramètre de la requête vers NewTransfer.

De manière générale, toutes les requêtes testées seront toujours au minimum précédée d'une requête de précondition permettant de générer le token d'authentification. Il s'agit de la raison majeure pour laquelle JMeter a permi un gain de temps considérable, il n'y avait en effet plus à se soucier d'aller chercher un token en passant par le TSE à chaque exécution de requête.

2.2 - Json extractor Cet élément est de type post-processeur et permet d'extraire la valeur d'un champ d'une réponse au format JSON en utilisant JsonPath, un langage permettant d'adresser une partie d'un document JSON, puis de la stocker dans une variable globale. Positionné dans un échantillon de type précondition, il permet d'extraire une information qui pourra ensuite être passée en paramètre d'une requpete testée.

Par exemple, après l'appel à AccountToDebit, il permet de récupérer l'identifiant d'un compte débiteur qui sera utilisé comme paramètre dans la requête NOBC-API-28 vers NewTransfer.

Cet élément est aussi utilisé sur les requêtes testées pour extraire les valeurs de tous les champs de la réponse afin qu'ils soient ensuite vérifiés par le biais d'assertions.

2.3 - Beanshell extractor Cet élément a le même objectif que json extractor mais utilise un script Beanshell à la place de JsonPath. Beanshell est un langage de script dont la syntaxe est très proche de Java, interprété et dynamiquement typé. JsonPath permet d'accéder à un champ JSON particulier mais ne permet pas de stipuler des conditions complexes, c'est pourquoi dans certains cas j'ai dû recourir à des scripts Beanshell pour récupérer une valeur bien précise dans un champ d'une réponse.

Par exemple, supposons maintenant que l'on souhaite appeler le service NewTransfer. Il faut dans un premier temps récupérer l'id d'un compte débiteur via un appel à AccountToDebit, ce qui est faisable facilement via JsonPath. Ensuite, il faut récupérer un compte créditeur via un appel à AddressBook. Or, le compte créditeur doit obligatoirement être différent du compte débiteur, condition qu'il n'est pas possible de préciser via JsonPath, c'est pourquoi il faut recourir ici à un script Beanshell pour récupérer une valeur cohérente.

2.4 - Assertion Les éléments assertions sont aussi de type post-processeur et sont des scripts Beanshell dont l'objectif est de vérifier des conditions puis de retourner une réponse qui déterminera le status du test après son exécution (succès ou échec). Lorsque la requête à tester est exécutée, les éléments json extractor stockent les valeurs des champs de la réponse dans des variables qui pourront ensuite être utilisées dans ces scripts. Il ne reste plus qu'à écrire le script et les conditions de son succès/échec.

2.5 - HTTP cookie manager Elément de type pré-processeur permettant de configurer les cookies d'une requête avant son exécution. Ici, chaque requête doit au minimum posséder un cookie contenant le token d'authentification qui est récupéré automatiquement au préalable via l'exécution d'une requête permettant de générer ce dernier.

2.6 - HTTP header manager Elément de type pré-processeur permettant de configurer le header de la requête si besoin est. (par exemple le content-type)

3 - View result tree Elément permettant d'afficher, entre autre, le résultat de l'exécution de toutes les requêtes (succès/échec) ainsi que leur réponse. Celui-ci permet aussi de spécifier si l'on souhaite exporter le résultat du plan de test, les informations à exporter et leur format. J'ai décidé d'exporter les données au format XML afin de pouvoir les réimporter par la suite sous TestLink.

4 - Summary report Rapport permettant de fournir des informations plus détaillées sur l'exécution des tests comme le temps moyen par requête, le nombre de succès/échec, la bande passante consommée etc...

5 - HTTP request defaults Permet de configurer les paramètres HTTP par défaut (nom de domaine, port) ainsi que les informations concernant le proxy interne.

6 - Users variables Permet de définir des variables globales qui pourront être accessible depuis n'importe quel endroit. Une seule variable a été défini, il s'agit du nom de l'environnement sur lequel les tests devaient être effectués (Homo3 ou RGB). En effet, il serait extrêmement fastidieux de devoir changer l'url de toutes les requêtes manuellement à chaque fois que l'on change d'environnement ou de faire un deuxième plan de test, c'est pourquoi le nom de domaine de celui-ci a été externalisé dans une variable pouvant ensuite être appelée dans l'url de chacune des requêtes.

7 - Assertions listener Elément permettant d'afficher le résultat de toutes les assertions appliquées sur les requêtes exécutées dans le plan de tests.

La structure du plan de test étant défini, j'ai ensuite procédé à l'écriture de toutes les requêtes, scripts Beanshell et autres assertions afin d'automatiser le plus de cas de test présent sur TestLink possible. Environ une centaine de ces cas ont pu être ainsi automatisés. Après l'exécution du plan de test dans son intégralité sur JMeter, un fichier XML contenant tous les résultats était généré.

Cependant, JMeter et TestLink étant deux outils totalement indépendants, il est évident qu'ils n'attendaient pas de fichiers XML ayant la même structure. Or, modifier ces fichiers manuellement serait contre productif et n'aurait alors aucun intérêt. Dès lors, j'ai décidé d'avoir recours au langage *XSLT* ou *eXtensible Stylesheet Langage Transformation*. Ce dernier, est un langage permettant de transformer un document XML en un autre en modifiant sa structure. Pour cela, il se base sur la manipulation de modèle ou template afin d'altérer le fichier XML source en remplaçant ses éléments d'origines par des éléments donnés. J'ai donc utilisé ce langage afin de construire des feuilles de style XSL dont l'objectif était de décrire les transformations à effectuer pour obtenir un fichier XML prêt à être importé sur TestLink à partir du fichier XML généré par JMeter. Afin d'appliquer cette feuille de style, j'ai mis en place un processeur XSLT nommé *Saxon XSLT*. Celui-ci est un moteur prenant un document XSLT en entrée ainsi qu'un fichier XML source à transformer pour produire en sortie un nouveau document au format souhaité. Pour finir, j'ai mis en place un programme léger en Batch prenant en paramètre tous les documents impliqués ainsi que les données TestLink requises comme le titre du plan de test ou le nom du testeur et générant le fichier de résultat pour l'import.

3.3.5. Procédure finale et perspectives

Tous les éléments étant mis en place, j'ai rédigé un guide décrivant les actions à effectuer concernant les tests lors d'une livraison afin de générer les rapports à envoyer à PBI. Celui-ci décrit toutes les étapes pour la création ou modification des tests automatiques ainsi que le travail que j'ai effectué. De plus, celui-ci décrit comment installer et configurer l'ensemble des outils utilisés (TestLink, Apache JMeter et Saxon XSLT).

J'ai ensuite pu mettre le guide à disposition des développeurs et leur ai exlicité la marche à suivre finale qui est la suivante :

— Lancer les tests automatiques sur JMeter
— Récupérer le fichier XML généré contenant les résultats du plan de test

— Exécuter le programme Batch pour généré le fichier XML prêt à l'import pour TestLink
— Créer un nouveau plan de test sur TestLink
— Importer le fichier XML
— Repérer les tests en échecs et créer un ticket Mantis pour ces derniers
— Générer les rapports PDF et Excel

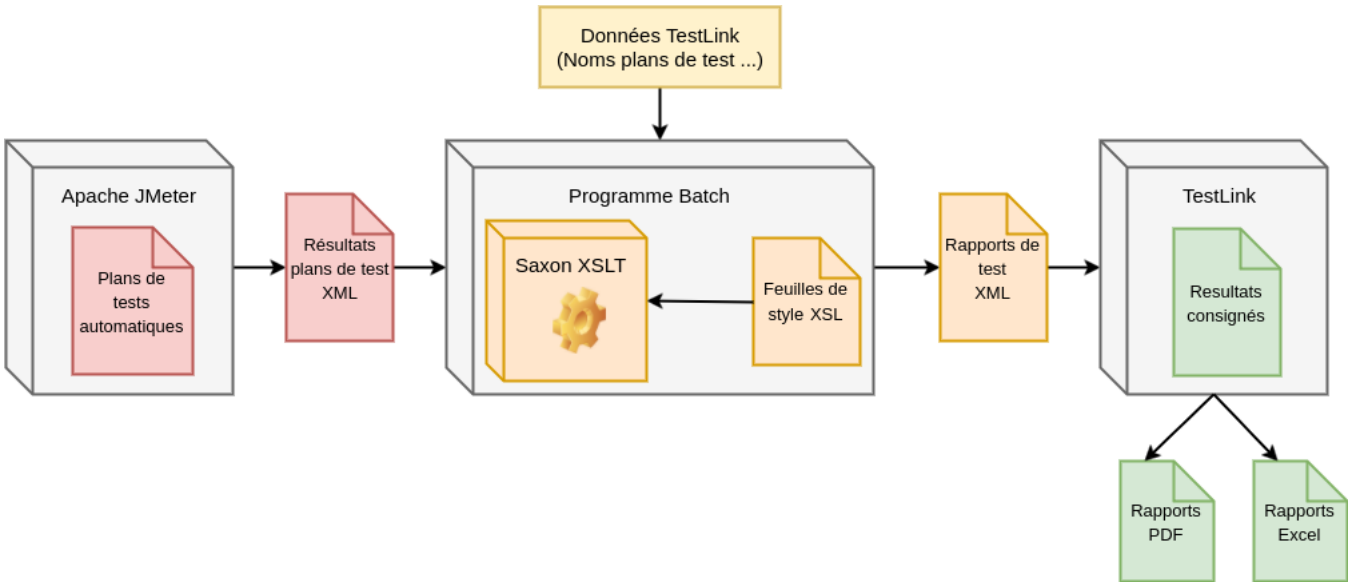

Figure 3.9. – Procédure de test finale

Lors d'une livraison, une ressource pouvait parfois être occupé jusqu'à une demi-journée rien que pour exécuter l'ensemble des tests et consigner leur résultats, pour les raisons que nous avions évoquées partie 3.3.3. Avec cette nouvelle procédure, la génération des documents à destination du client prenait environ **10 minutes**, le plus long étant d'attendre la fin de l'exécution des tests automatiques. Ceci représente un gain de temps considérable pour l'équipe et permet d'obtenir des rapports conformes avec les besoins et attentes du client. De plus, cela a permis de soulager les développeurs d'un bon nombre de tâches répétitives et peu intéressantes d'un point de vue technique bien qu'essentielles dans le processus de satisfaction du client, dégageant ainsi du temps supplémentaire dans les sprints pour accomplir les objectifs fixés.

Cependant, malgré les bénéfices apportés, certains points restent perfectibles. En effet, l'application web TestLink est actuellement hébergée sur le serveur personnel d'un des architectes de l'équipe. Le projet touchant à sa fin, celui-ci sera repris par une équipe de maintenance qui aura besoin d'accéder à tous les outils et qui devra consigner les résultats de ses tests quotidiens c'est pourqoi il aurait été recommandé de pouvoir procéder à l'installation de TestLink sur un serveur dédié.

De plus, il aurait pu être intéressant de pousser l'automatisation des tâches à son maximum en ayant recours à Jenkins. En effet, il s'agit d'un outil d'intégration continue permettant d'automatiser différentes procédures. J'ai appris par la suite qu'un Jenkins aurait dû être mis en place au début du projet mais que cela n'avait pas été effectué faute de temps, il est prévu qu'un architecte se charge de cette installation. JMeter et TestLink sont tous deux compatibles avce cet outil tout comme les programmes Batch, il serait ainsi possible par la suite de programmer l'exécution des tests et la création des rapports de manière entièrement automatique à un intervalle de temps souhaité.

3.3.6. Aller plus loin : Tests de charge

L'équipe avait émis le besoin de mettre en place des tests de charge, JMeter étant installé et y étant formé j'ai pris en charge la réalisation de ces tests. Ces derniers consistent à mesurer le temps de réponse d'un système lorsque celui-ci est soumis à des conditions particulières afin de vérifier s'il

est capable de soutenir le traffic attendu. Les conditions regroupent différents paramètres comme le temps de réponse, le volume du traffic, le nombre de requêtes effectuées en parallèle, la configuration du matériel ou encore la stabilité des serveurs. Afin de réaliser de tels tests, il faut dans un premier temps configurer l'environnement sur lequel ils seront exécutés. Dans notre cas, nous avions à notre disposition un environnement de pré-production (RGB) dont les caractéristiques étaient identiques à celles de la production, ce qui était approprié pour réaliser ce genre de tests. De plus, nous avions aussi l'environnement HOMO3 avec des capacités inférieures, idéal pour exécuter les tests dans des conditions un peu plus extrêmes dans le but mesurer la robustesse des APIs.

Après cela, j'ai pris soin de définir différents scénarios de tests dont l'objectif étaient de simuler l'activité utilisateur afin de se rapprocher le plus possible des cas d'utilisation réels. Par exemple, un des scénario classique pourrait décrire la procédure suivie pour effectuer une transaction bancaire d'un compte vers un autre. Pour cela, l'utilisateur pourrait se connecter à l'application mobile puis initier sa transaction en choisissant des comptes débiteur et créditeur et il enfin il la signerait numériquement. De plus, celui-ci pourrait d'abord consulter ses comptes avant de prendre la décision de réaliser cette transaction. Les scénarios sont représentés par des diagrammes de cas d'utilisation comme celui présent sur la figure B.1, décrivant la réalisation d'une transaction.

On peut ainsi remarquer rapidement les différents services (listés en annexe A.2) qui seront appelés lors de l'exécution de ce scénario :
— VKB pour générer le clavier virtuel permettant l'authentification et la signature de la transaction
— tokenInfo pour générer le token d'authentification
— newTransfer pour initier la transaction
— accountToDebit pour sélectionner le compte débiteur
— addressBook pour sélectionner le compte créditeur
— signByVKB pour signer la transaction
— clientList pour lister tous les comptes
— accountOverview pour consulter un compte particulier

Une fois les services identifiés, il ne restait plus qu'à construire le plan de test JMeter permettant de réaliser le scénario. Pour cela, j'ai réemployé la structure mise en place précédemment pour les tests fonctionnels. Cependant, j'ai supprimé l'ensemble des assertions puisque ces dernières peuvent influer sur les temps de réponse calculés par JMeter et n'ont pas d'intérêt dans un test de charge. Il ne restait donc qu'à modifier les scripts Beanshell déjà écrits pour les adapter à la situation. La principale différence réside dans la configuration du moteur d'utilisateur. En effet, sur les tests fonctionnels j'avais placé un tel moteur à la racine des plans afin qu'il englobe tous les composants. Ainsi, pour les tests de charges il suffisait de modifier les paramètres du moteur à savoir le nombre de thread, d'itération et le temps de montée en charge pour adapter les conditions d'exécution des scénarios sans avoir à modifier la structure des plans.

Ces tests de charges ont par la suite été exécutés dans différentes conditions ce qui a permis de déceler certaines anomalies. Par exemple, après avoir essayé d'envoyer un grand nombre de requêtes par seconde, nous nous sommes aperçus que beaucoup d'entre elles échouaient. Après une analyse approfondie, il s'est avéré qu'une grande partie retournaient un "timed out" ; Cela provenait d'un problème de configuration de la gateway Zuul au sein de la couche microservices.

Nous allons utiliser le schéma figure 3.10 afin d'expliciter le problème rencontré. Bien que le raisonnement expliqué soit réel, les chiffres ne le sont pas et ne sont présents que pour illustrer ce qui n'empêche pas la compréhension. JMeter envoie 1000 requêtes à la seconde qui passeront dans un premier temps par l'API Gateway d'Axway qui les traite sans soucis. Celle-ci peut donc forwarder l'ensemble des requêtes à l'API microservices qui seront alors interceptées par la gateway Zuul. Elle était configurée pour gérer 100 threads par seconde avec un timeout de 1 seconde. Cela signifie qu'elle peut traiter 100 requêtes par seconde et qu'elle rejette les requêtes donc le temps de traitement est supérieur à 1 seconde en tuant le processus. Il fallait donc revoir cette configuration à la hausse afin de résoudre le problème

rencontré. Cependant, augmenter la taille du pool de threads que la gateway est capable de traiter par seconde aura forcément un impact sur le temps de réponse moyen des requêtes. Ainsi, une petite étude a été menée afin de déterminer une taille optimale permettant de traiter un nombre satisfaisant de requêtes tout en conservant un temps de réponse moyen correct. Pour cela plusieurs tests de charge ont été effectués avec des configurations différentes et nous avons procédé par dichotomie pour trouver les valeurs les plus satisfaisantes (exemple tableau 3.3). De plus, nous avons attendu confirmation d'EFS sur la capacité de ses API afin de ne pas risquer une congestion à la sortie de la couche microservices.

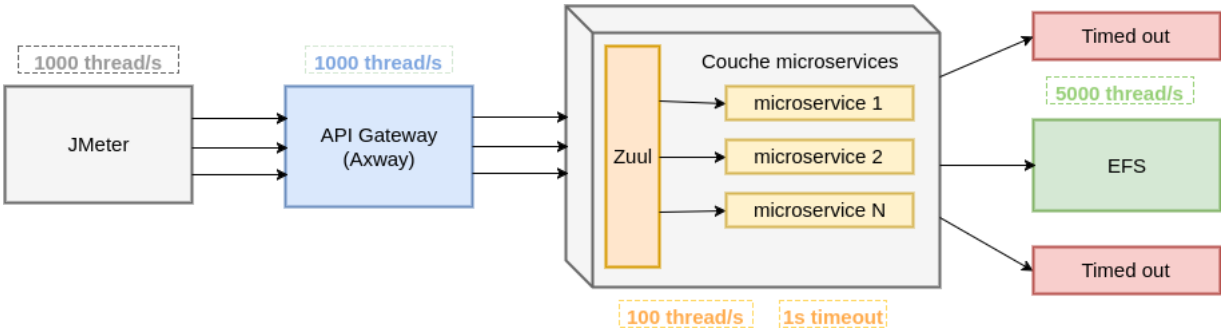

Figure 3.10. – Traitement des threads par les différentes couches

Threads/s	Temps de réponse moyen (ms)
100	300
300	350
500	420
700	750
1000	1250

Table 3.3. – Temps de réponse moyen en fonction de la taille des pools de threads

3.4. Gestion des doubles relations

3.4.1. Définition du besoin

La version 1.0 de l'application mobile est réservée à la clientèle *private* de NOBC, ne possédant que des produits de type PP (Personnes Physiques). Cependant, Neuflize ayant la volonté d'étendre la portée de ses services numériques afin de toucher un public toujours plus large, elle a souhaité rendre l'application disponible pour les PM (Personnes Morales). Ainsi, elle souhaitait inclure très rapidement dans une version 1.1 la clientèle *double relation*, c'est-à-dire la clientèle de type "entreprise" mais qui dispose en même temps dans ses accès des produits "particuliers" ; typiquement un entrepreneur qui en plus de ses comptes "entreprises" peut aussi voir/faire de la transaction sur ses comptes "privés". Or, d'un point de vue légal, les abonnés double relation n'ont pas les mêmes droits que les clients simple c'est pourquoi des modifications ont dû être apportées aux API. Les méthodes agiles nous apportant une certaine flexibilité, nous avons organisé une réunion avec les métiers afin d'étudier le besoin en détail et déterminer quelles modifications seraient appropriées.

A l'issue de cette réunion avec le client, nous avons émis différentes problématiques afin de déterminer les actions à mettre en place au niveau de la couche microservices. Sur le site internet, un abonné double relation dispose d'un bouton lui permettant de visualiser soit ses comptes "entreprises" soit ses comptes "privés". Néanmoins, il n'est pas prévu à ce jour de tel bouton permettant à l'abonné de passer de ses comptes "entreprises" à ses comptes "privés" et inversement sur la version 1.0 de l'application mobile. En effet, celle-ci est réservée à la clientèle "private". De ce fait les produits (comptes, cartes, crédits, titres, etc.) à présenter doivent être des produits PP. De plus, EFS s'est engagé à fournir toutes les

informations dans ses API pour que le niveau API microservices puisse faire la distinction entre les comptes et produits PM et les comptes et produits PP. Ainsi, le besoin au niveau de la couche API microservices consistait donc à mettre en place des filtres pour présenter/filtrer les données de/vers la couche API de PBI.

3.4.2. Mise en œuvre

Avant d'aller plus loin, il est nécessaire de définir ce que sont les *racines*. Une racine est en réalité un id client en base de données. Un abonné utilisant l'application peut avoir la main sur plusieurs racines (la sienne et d'autres via mandat par exemple). Elles contiennent l'ensemble des produits qui appartiennent au client (comptes courant, épargne, portefeuille, crédit, position etc...).

La mise en place de la gestion des doubles relations m'a été confiée et je me suis donc chargé de construire la solution. J'ai d'abord présupposé que les actions menées par EFS avaient été réalisées. Afin de les définir de manière précise, j'ai organisé un point avec une personne métier. Cette dernière a donc pu me les expliciter et elles étaient les suivantes :

— Véhiculer l'information <type de racine> au niveau du back office EFS
— Enrichir l'API pour ajouter l'information <type de racine>
— Modifier l'API pour remonter les abonnés dont le profil est "particulier" ou "particulier bourse" ainsi que les abonnés dont le profil est "entreprise" ou "entreprise bourse" et qui ont des racines typées "privé"

De plus, une table de décision a été mise en place côté EFS (en SQL) dans le but de décider si un type de racine est de type entreprise ou de type particulier afin de le remonter ou non depuis EFS et de préciser l'ordre de présentation (colonne "poids") à exploiter le cas échéant. Cette table était de la forme suivante (les termes métiers importent peu pour la compréhension) :

Type de racine	Entreprise/Particulier	Poids
PARF	Particulier	1
PARH	Particulier	2
STE	Entreprise	10
...	...	...

Table 3.4. – Table de décision pour les doubles relations

Les règles, issues des spécifications, à respecter concernant l'identification du type des racines étaient les suivantes :

— Si une racine est liée à plusieurs produits dont des comptes :
  — Son type entreprise/particulier est porté par la nature du compte
  — Tous les autres produits (crédit, titre, carte bleue, assurance vie) rattachés à cette racine auront la même nature que le compte
— Si une racine est liée à plusieurs produits sans comptes :
  — L'API côté EFS devra transmettre l'origine : P (particulier), E (entreprise)
  — Cette API est cependant susceptible de transmettre <blanc> : alors ce sera le type de la racine qui déterminera si l'ensemble de ses produits est de type particulier ou entreprise
— Le cas où une racine est liée à un compte entreprise et un compte particulier n'est pas considéré et est traité comme une anomalie à corriger

En outre, un cahier des charges m'a été fourni dans lequel il était stipulé toutes les modifications à effectuer concernant l'appel des services EFS. En effet, comme nous l'avons déjà expliqué, nos services consomment ceux d'EFS afin de la faire de la composition.

Ainsi, il était par exemple expliqué que le service EFS permettant d'obtenir les assurances vie liées à une racine ne devrait être appelé uniquement pour les clients ne possédant que des produits privés (pas entreprises).

Pour un autre exemple, EFS expose un service permettant de récupérer les informations de toutes les cartes bleues liées à une racines. Aucun changement n'était demandé concernant l'appel de ce service mais il était nécessaire de filtrer les cartes obtenues afin de ne garder que celles liées à un profil privé et non entreprise.

Mon objectif était de déterminer quels seraient les microservices impactés et dans quelle mesure par la gestion des doubles relations. Après analyse du cahier fourni, j'ai synthétisé l'ensemble des informations dans le diagramme présent sur la figure 3.11.

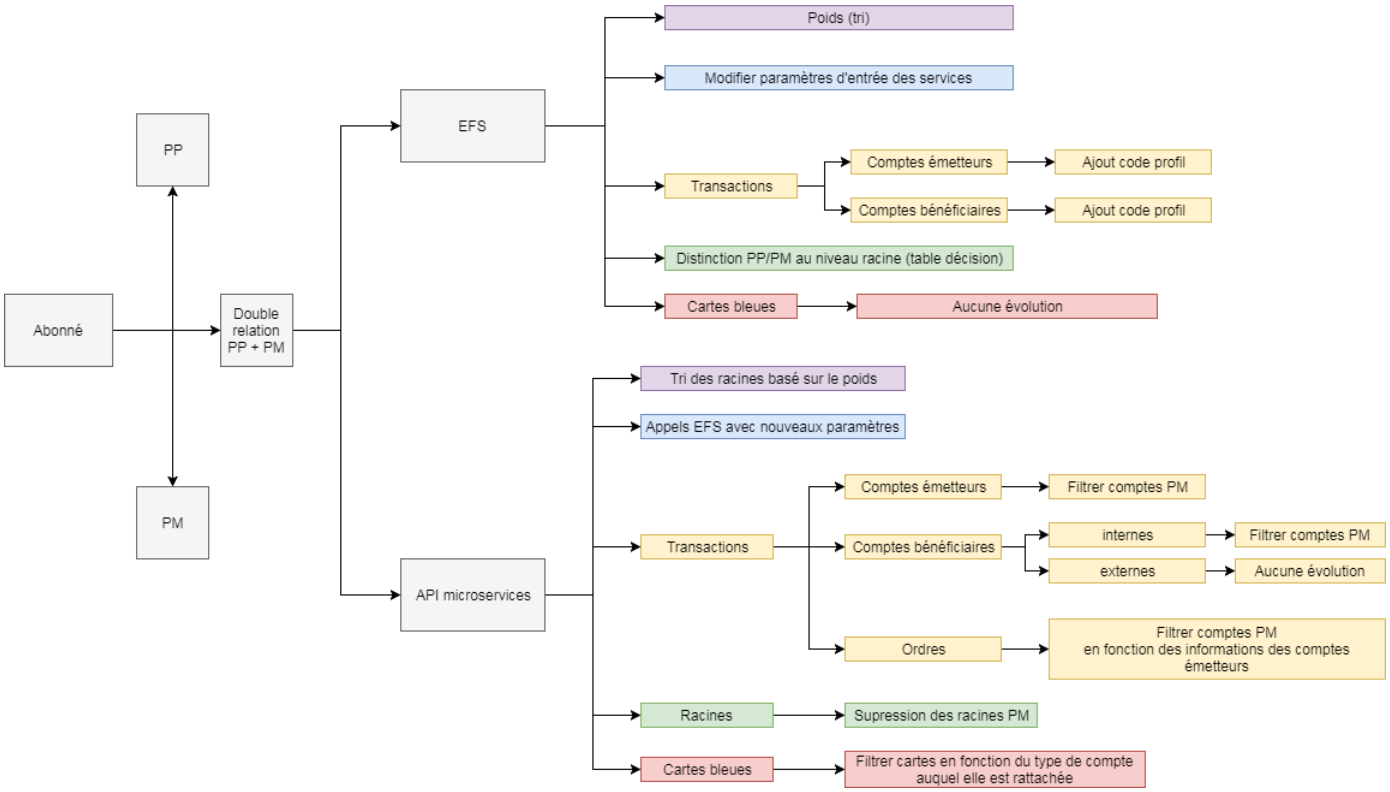

Figure 3.11. – Objets impactés par les doubles relations

Il était maintenant possible de déterminer les services qui aurraient besoin de développement supplémentaire. D'après ce schéma, les services concernés étaient ceux présent dans le tableau figure 3.5. Une fois cela établi il ne restait plus qu'à procéder au développement et à effectuer les modifications nécessaires afin d'assurer la gestion des doubles relations dans la couche API microservices.

Service	Raison
accountToDebit	Créer un filtre sur les comptes émetteurs
addressBook	Créer un filtre sur les comptes bénéficiaires internes
transactionOverview	Créer un filtre sur les ordres de transaction
clientList	Créer un filtre sur les racines récupérées et les trier selon leur poids
accountOverview Tous	Créer un filtre sur les cartes bleues Rajouter le paramètre "codeProfil=PP" dans les appels aux services EFS

Table 3.5. – Modifications par services pour la gestion des doubles relations

3.4.3. Résultats

Comme nous l'avons vu, les modifications à apporter sont mineures et les nouveaux développements sont loin d'être conséquents, mais ces derniers impactent de très nombreux endroits du code et nécessite d'altérer plusieurs services ce qui aurraient pu gêner les autres développeurs dans leur travail. De plus, EFS n'étant pas prêt de son côté, j'ai travaillé en utilisant *WireMock*, un outil permettant de créer des *mocks* qui sont des objets simulés reproduisant le comportement d'objets réels. Ainsi, cela m'a conforté dans l'idée de créer une nouvelle branche sur le github du projet sur laquelle j'ai poussé le code permettant la gestion des doubles relations. Une fois EFS prêt, il sera alors facile de simplement réaliser un merge de cette branche avec la branche master.

Bien que la réalisation de cette tâche ne présentait pas de grandes difficultés d'un point de vue de technique, celle-ci m'a permis de me familiariser avec l'aspect relationnel du projet. En effet, nous travaillons en employant des méthodes agiles ce qui m'a conduit a rencontrer le client c'est-à-dire les métiers travaillant dans le même openspace que nous. Ceci m'a permis d'avoir une première expérience complète, bien qu'à petite échelle, allant de la définition du besoin avec le client jusqu'à la réalisation de la fonctionnalité en accompagnant celui-ci tout au long du processus via diverses réunions afin de fixer les points cruciaux. Ensuite, j'ai pu réaliser cette tâche de manière autonome et donc eu la chance de pouvoir mettre en place mes propres solutions tout comme pour la partie concernant les tests fonctionnels, à ceci prêt que dans ce cas il ne s'agissait pas d'un travail indépendant mais en relation direct avec le code source et une livraison, impliquant une certaine confiance. En outre, j'ai été contraint d'apporter des modifications à travers toute l'API ce qui m'a permis de pouvoir mieux m'approprier le code et de monter ainsi en compétence sur la stack utilisée et surtout sur le développement de "microservices".

3.5. Monitoring applicatif

3.5.1. Définition du besoin

La mise en production de l'application mobile approchant, le client a émis le souhait de pouvoir collecter des données sur la manière dont les utilisateurs l'utiliseraient. Il souhaitait avoir à sa disposition un outil lui permettant de monitorer l'application afin de pouvoir évaluer la satisfaction des utilisateurs et de relever les axes d'amélioration.

Par exemple, pour réaliser une transaction bancaire via l'application mobile, l'utilisateur doit se connecter puis initier la transaction en choisissant des comptes émetteurs/receveurs ainsi qu'un montant. Après cela, un clavier virtuel apparait sur lequel il doit rentrer son code secret, ce qui aura pour effet de *signer* la transaction. Une des requêtes de Neuflize était de pouvoir avoir un ratio entre le nombre de transaction initiée et le nombre de transaction signée afin de déterminer les nombres de transaction abandonnée en cours de création et donc si ce système de signature ne rebutait pas les utilisateurs dans leur démarche. En effet, un grand nombre de transaction initiée mais non signée pourrait indiquer que cette procédure ne plait pas aux clients de la banque parce qu'elle serait trop longue ou peu pratique. De plus, cela permettrait aussi de suivre les potentielles erreurs qui apparaitraient entre l'initiation et la signature afin de prendre des mesures adéquates.

Afin de suivre ces informations, Neuflize souhaitait pouvoir disposer d'un *dashboard* (tableau de bord), qui centraliserait l'ensemble des données collectées sous la forme de graphiques. Le dashboard contiendrait alors tous les graphiques et permettrait de naviguer facilement afin de retrouver les informations recherchées, et ce en fonction du temps. En effet, le critère de temps est indispensable dans l'optique de suivre l'évolution de l'applicaton et pour situer les événements.

Ainsi, avant que la réalisation d'un tel dashboard me soit confiée, les outils qui seraient utilisés avaient été décidé suite à une réunion entre l'un des architectes du projet et les métiers. Il m'a donc été demandé de répondre à ce besoin en mettant en place, dans un premier temps, la stack ELK (Elasticsearch, Logstash et Kibana) dont nous parlerons dans la partie suivante puis en créant une

première ébauche des graphiques en local à l'aide de données factices. En outre, une deadline avait été fixée puisque le dashboard, ou du moins la collecte des données, devait être prêt pour la mise en production de l'application. En effet, les premières informations sont essentielles pour déterminer les évolution à apporter à l'application et permettent d'avoir une idée de la satisfaction des utilisateurs.

Il m'a été remis un document réalisé par Neuflize, dont un extrait est disponible figure 3.12, dans lequel il était décrit l'ensemble des besoins c'est-à-dire toutes les informations qui étaient souhaitées, comment elles devaient être traitées et la manière dont elles devaient être gérées. Le premier travail à effectuer a été de déterminer quelles informations pouvaient être remontées par l'API microservices et lesquelles ne le pouvaient pas. Par exemple, nous pouvons voir qu'il était demandé le nombre de transaction initiée sur une période donnée, chiffre qu'il est possible pour nous de fournir. Cependant, il est aussi demandé le temps passé sur chaque écran de l'application par l'utilisateur ou encore le nombre de page que celui-ci a visité, ce qui est impossible à fournir de notre côté. En effet, la couche API microservices est bien évidemment situé dans la partie backend du projet et est une API REST exposant des services. Ainsi, il nous est totalement impossible d'obtenir des informations si nos services ne sont pas appelés. Le temps passé sur chaque écran n'est pas calculable puisque certains écrans peuvent ne pas appeler de services, certains en appeleront plusieurs etc... Il a donc fallu analyser l'ensemble des besoins dans le but de déterminer ce qui était réalisable de notre côté, quant au reste, il était possible de l'obtenir du côté frontend via d'autres outils qui avaient été mis en place par l'équipe en charge de cette partie tel qu'*Omniture*.

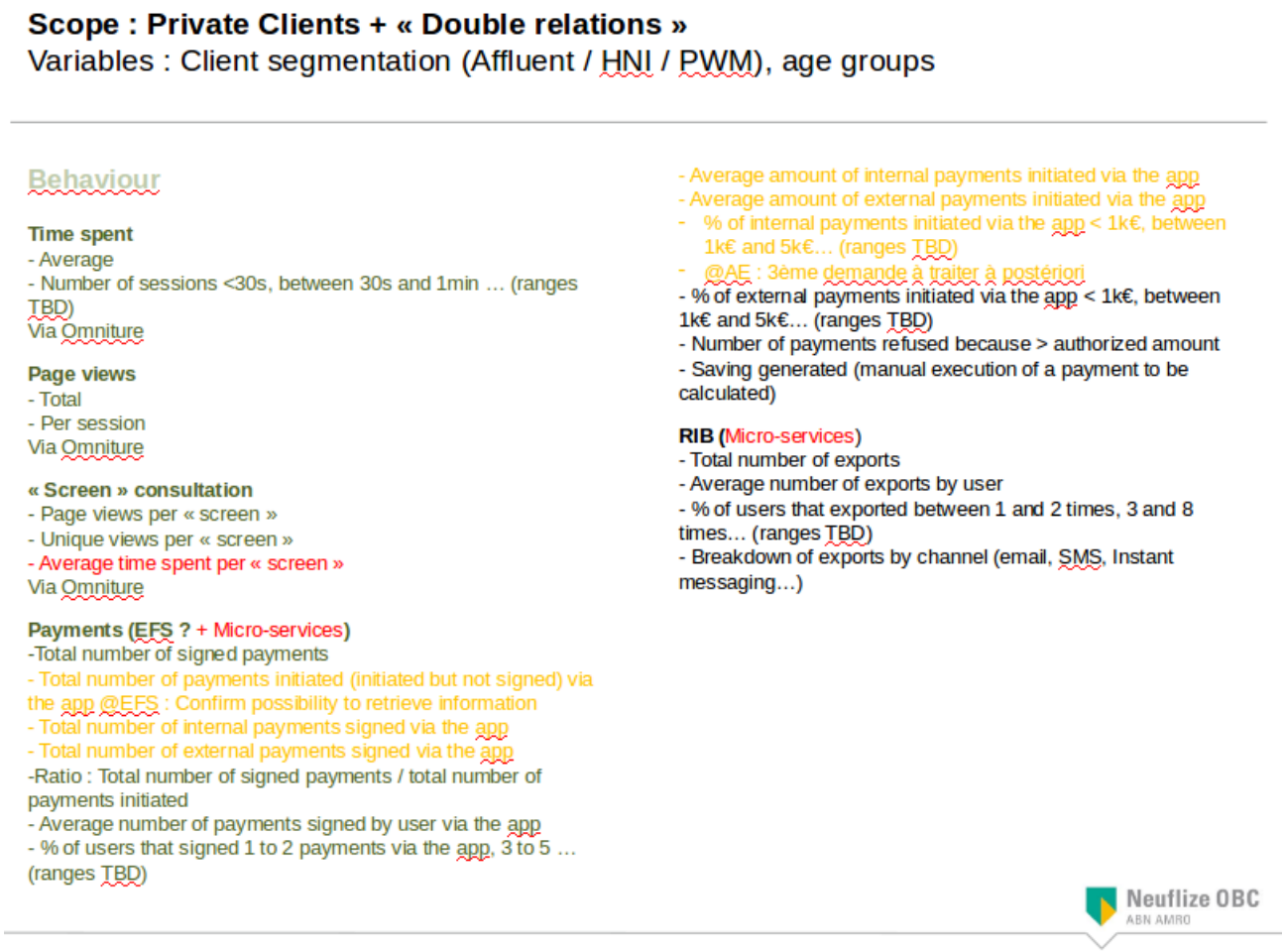

Figure 3.12. – Extrait du besoin client pour le dashboard

Une fois cette étude terminée j'ai effectué un rapport auprès d'un architecte du projet qui a validé mon analyse puis j'ai organisé une réunion avec les métiers afin de clarifier la situation et leur faire savoir

tout ce qui était faisable par l'API microservices. Au terme de cette réunion et après discussion sur la manière de remplacer certaines informations inaccessibles par d'autres s'en rapprochant ou permettant de les retrouver, j'ai pu me lancer dans la mise en place de la stack ELK et la réalisation du premier jet du dashboard.

3.5.2. ELK : ElasticSearch, Logstash et Kibana

Comme nous l'avons dit précédemment, pour réaliser le dashboard il a été décidé que la stack ELK serait utilisée. J'ai donc, dans un premier temps, du me former à l'utilisation de cette dernière qui est constituée de trois produits de la société Elastic que sont Elasticsearch, Logstash et Kibana. Son objectif premier est de stocker, d'analyse et de lire les logs générer par un projet tel qu'une API REST dans l'optique de faire du monitoring, de suivre l'activité d'un service ou de prévenir des dysfonctionnements. Nous allons maintenant décrire en quoi chacun des outils est intervenu dans le processus de réalisation du dashboard. Pour cela, il est possible de se référer à la figure 3.14 afin de pouvoir situer chaque composant.

Elasticsearch

La quantité de logs générés par une application de cette ampleur peut rapidement devenir très conséquente et de nombreuses problématiques sont alors soulevées concernant le stockages des messages importants, leur recherche ou leur analyse. Les bases données relationnelles classiques ont montré leurs limites concernant le travail avec de tels volumes de données sur le long terme que ce soit pour le stockage ou les temps de réponse lors d'une recherche, et ce malgré des requêtes bien construites avec des jointures judicieuses entre les tables. Elasticsearch a pour objectif de répondre à cette problématique.

Il s'agit d'un moteur d'indexation, de stockage et de recherche de données développé en Java basé sur *Lucene*, une bibliothèque d'indexation et de recherche de texte créée par la fondation Apache. Son principal atout est qu'il permet de stocker de large volume de données tout en permettant aux utilisateurs d'effectuer des recherches en temps réel. Les données sont stockées sous la forme de *document*. Il s'agit d'une unité basique d'information qui peut être indexée. Les index sont des collections de documents possédant des caractéristiques similaires. Les index peuvent aussi posséder un type afin de les diviser en plusieurs parties "logiques". On peut, par exemple, créer un index *api-microservices-2017.07* avec un type *log*.

On pourra alors récupérer le document d'id 1 avec la requête suivante :

```
GET api-microservices-2017.07/log/1?pretty
```

La réponse contiendrait ledit document avec les champs définis par nos soins (ici timestamp, logLevel, service et message) :

```
{
 "_index": "api-microservices-2017.07",
 "_type": "log",
 "_id": "1",
 "_version": 1,
 "found": true,
 "_source" : {
  "timestamp": "2017-07-07T14:15:45",
  "logLevel": "INFO",
  "service": "account-service"
  "message": "elasticsearch example"
 }
}
```

Dans notre cas, Elasticsearch est utilisé comme support de stockage qui sera par la suite interrogé par Kibana. Cependant, il peut tout à fait être interroger depuis une API, c'est pourquoi il ne dispose pas de client dédié.

Logstash

Logstash est un *ETL* ou *Extract-transform-load*. Cela désigne les outils capable de synchroniser des volumes de données massif depuis une source précise vers une autre. Ainsi, Logstash agit un peu à la manière d'un pipeline capable de prendre un grand nombre de messages en entrée depuis plusieurs sources pour les traiter, effectuer des transformations ou encore les filtrer avant de les renvoyer vers un support de stockage. Le schéma figure 3.13 explicite son principe de fonctionnement. La liste des technologies mentionnées dans celui-ci n'est pas exhaustive, il s'agit purement d'exemples de ce qu'il est possible de faire à titre démonstratif.

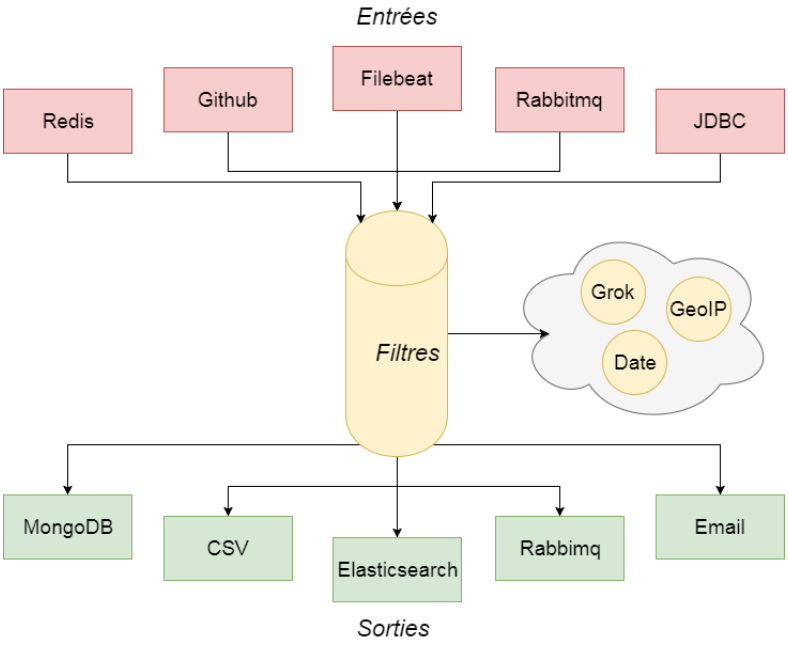

Figure 3.13. – Logstash

Logstash accepte en entrée un nombre impressionnant de formats différents comme tout ce qui est représenté sous forme de chaîne de caractères mais aussi les événements issus de sources différentes et ce de manière simultanée. Dans notre cas, logstash prendra en entrée des fichiers logs fournit par Filebeat dont nous parlerons plus en détails peu après. Ces logs auront été générés au préalable par l'API microservices à l'aide de *Logback*.

Une fois les données acquises, celles-ci sont analysées puis transformer par les filtres de Logstash. Il est ainsi possible d'ajouter, modifier ou supprimer de l'information, extraire des valeurs depuis des messages, analyser des événement et bien d'autres. Le filtre le plus puissant mis a disposition par Logstash est *Grok* que nous utiliserons ici et qui permet de transformer dynamiquement des données non structurées en données structurées en découpant une ligne de log en un ensemble de champs prédéfinis. Nous pourrons donc utiliser ce filtre afin de structurer les informations de telle manière qu'elles soient facilement exploitable par Kibana. Par exemple, la ligne de log suivante :

```
1 [2017-07-07T14:15:45] [INFO] [account-service] - elasticsearch
    example
```

peut être transformer sous la forme du document structuré montré en exemple plus haut à l'aide d'un filtre adéquat définissant les champs (comme timestamp ou loglevel).

Enfin, une fois les données formatées, il est possible de les envoyer vers la destinaiton de notre choix, et là encore Logstash offre la possibilité de transmettre les données vers un grand nombre de sources différentes. Il sera bien évidemment ici connecté à Elasticsearch et permettra donc de stocker les logs de manière structurée et indexée au sein de la base de données. En outre, il offre une grande flexibilité puisque chacune des entrées/sorties ou mêmes les filtres sont implémentés sous la forme de plugin. Par ailleurs, la société Elastic a mis à disposition une API permettant de faciliter le développement de ces plugins, ce qui permet d'ajouter de nombreuses nouvelles possibilités à Logstash.

Kibana

Avec les outils précédent nous avons de quoi stocker nos logs, les analyser, les transformer pour structurer les informations qu'ils contiennent et rechercher ce que nous souhaitons parmis eux à l'aide de requêtes Elasticsearch. Cependant, cela n'est absolument pas "user friendly" et nécessite des connaissances techniques ce qui ne suffit pas à satisfaire le besoin client. Kibana permet palier cette faiblesse en nous fournissant une interface nous permettant de choisir la mise en forme de nos données. Celui-ci permet de construire des graphiques, nommés *visualisations*, tels que des histogrammes, des camemberts ou encore des tableaux. Ensuite, Kibana permet de choisir une plage de date pour ne traiter que les données situées dans cette dernière. Une fois le dashboard et les visualisations créés, il suffit de changer le temps et tous les graphiques s'actualiseront automatiquement ce qui permet de facilement retrouver des informations. Par exemple, il suffit d'un simple clic pour connaitre le nombre de connexion qu'il y a eu hier, le mois dernier ou entre le 23 juillet et le 15 août. De plus, Kibana vient avec une nouvelle composante nommée *Timelion* permettant d'effectuer des analyses chronologiques approfondies.

Ici, Kibana alimentera les visualisations avec les données stockées dans la base d'Elasticsearch et sera l'unique interface homme machine utilisée. En effet, il offre une console développeur permettant de faciliter la construction des requêtes Elasticsearch et permet de visualiser les données stockées en base à l'aide une option *discovery* affichant pour chaque ligne de log brute les champs structurés définis dans logstash.

Filebeat

Elastic a mis à notre disposition, en plus de la stack ELK, des agents de transfert nommés les *Beats*. Il en existe pour tout type de données mais dans notre cas nous utiliserons Filebeat conçu spécialement pour le transfert des logs et fichiers. Celui-ci permet de collecter l'ensemble des logs, qu'importe où ils se trouvent, pour les envoyer ensuite à Logstash. Il est possible de le configurer pour qu'il ne transfert pas les fichiers de logs entiers mais seulement les lignes qui nous intéressent. Celui-ci ne nécessite aucune intervention humaine une fois qu'il est mis en place, à chaque ligne de log générée il en est informé et effectue le transfert automatiquement. En outre, si Logstash est occupé et possède déjà un gros volume de données à traiter, Filebeat en sera également informé et pourra ralentir son flux d'envoi afin d'éviter tout problème de congestion.

Les intéractions en résumé

Toutes les intéractions entre les différents composants sont résumées sur le schéma figure 3.14. Pour commencer, les logs sont générés par l'API microservices grâce à Logback qui permet de configurer quelle ligne ira dans quel fichier et où ces derniers seront créés. Dès que des lignes de logs sont générées, Filebeat est averti et va donc lire le fichier de logs concerné, détecter les nouvelles lignes puis les envoie à Logstash si elles remplissent les critères. Ce dernier récupère ces lignes non structurées en entrée et agit comme un pipeline qui produira en sortie des données structurés à l'aide de champs prédéfinis. Ensuite, Elasticsearch reçoit ces données de la part de Logstash, les indexe puis les stocke en base de données. Enfin, Kibana interroge Elasticsearch à l'aide de requêtes en fonction du temps sélectionné pour alimenter les visualisations créées au sein du dashboard.

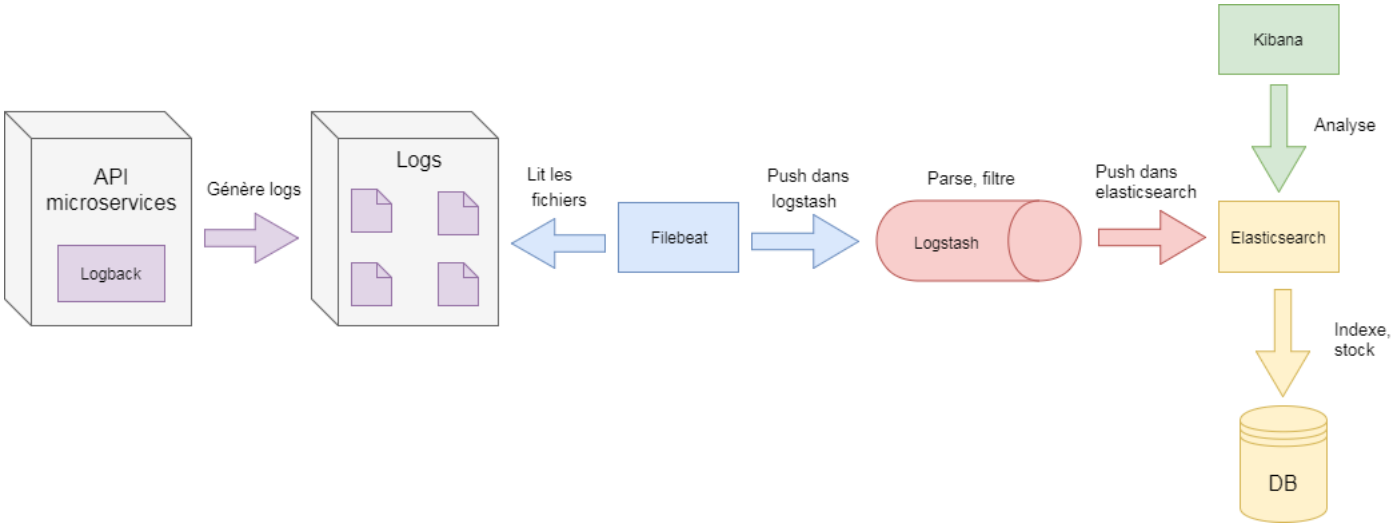

Figure 3.14. – Intéractions au sein de la stack ELK

3.5.3. Réalisation du dashboard

Comme nous l'avons vu dans la partie précedente, les données qui seront exploitées sont des logs. Ainsi, après avoir installé et configuré les différents éléments de la stack ELK, mon objectif était de générer des logs afin que puisse tester chacune des composantes entrant en jeu. Cependant, avant d'aller plus loin, il faut savoir que l'API microservices génère déjà de nombreux logs informatifs permettant d'analyser les actions effectuées, l'état des services ou encore les erreurs. De plus, il existe d'autres outils comme RabbitMQ, un message broker, qui génère une grande quantité de logs. Les lignes qui nous intéressent pour construire les visualisations sont donc noyées dans un flot d'information qui ne cesse de croître à mesure que le temps s'écoule. La première étape a donc été de définir un modèle permettant de différencier les lignes de logs qui seraient destinées au dashboard de celles qui ne le sont pas.

Mise en place du modèle

Grâce à Logback, tous les logs de l'API suivaient le même pattern constitué d'une première partie auto-générée et d'une seconde partie contenant le message effectivement loggé. Par exemple, dans le code la ligne suivante :

```
1 LOGGER.info("message");
```

aurait pour effet de générer la ligne de log suivante :

```
1 [timestamp] [host] [thread] logLevel class@method:line - message
```

avec :

timestamp	Date d'émission du log
host	VM sur laquelle le log a été émis
thread	Thread depuis lequel le log a été émis
loglevel	Niveau du log (INFO, WARNING, ERROR etc...)
class	Classe depuis laquelle le log a été émis
method	Méthode depuis laquelle le log a été émis
line	Ligne à laquelle le log a été émis
message	Le contenu du log

Table 3.6. – Log - première partie

J'ai décidé que les logs à destination de Kibana contiendraient un message commençant par le mot-clé *ANALYTICS*. Ainsi, j'ai configuré FileBeat pour qu'il n'envoie que les logs dont le message commençait par ce mot à Logstash et qu'il ignore les autres. La suite du message contiendrait toutes les informations nécessaires pour élaborer les graphiques. Parmis ces informations, les suivantes seraient obligatoires et entourées de crochet :

username	Login de l'abonné envoyant les requêtes
microservice	Microservice depuis lequel le log a été émis
service	Service depuis lequel le log a été émis

Table 3.7. – Log - deuxième partie

Une ligne finale ressemblerait donc a :

```
[timestamp] [host] [application] logLevel class@method:line - [
   ANALYTICS] [username] [microservice] [service] info1=valeur1 info2
   =valeur2 ... infoN=valeurN
```

J'ai ensuite placé la génération de log dans tous les services de l'API. Pour le service NewTransfer permettant d'initialiser une transaction bancaire on pouvait par exemple obtenir :

```
[2016-06-26 08:11:34.086] [FR7WSPAR54464] [main] INFO c.n.a.m.t.
   logstash.LogStashTests@testNewTransfer:36 - [ANALYTICS] [COCKZAO72]
    [Transaction] [NewTransfer] transactionId=13072409724 totalAmount=
   1000 paymentMode=SINGLE currency=EUR internal=true
```

Les informations optionnels transactionId, totalAmount, paymentMode, currency et internal sont propres au service NewTransfer et ne seront pas présentes dans les logs des autres services.

Configuration de Logstash

Tous ces logs seront envoyé par Filebeat à Logstash qui aura pour objectif de les structurer à l'aide de filtres. Il est possible de définir plusieurs filtres qui seront alors appliquer dans l'ordre les uns après les autres. Ainsi, j'ai créé un premier filtre permettant de parser la première partie des logs depuis le timestamp jusqu'au nom du service. Cette partie étant commune à tous les logs, ce filtre est toujours appliqué et en premier. Celui-ci se présente sous cette forme :

```
filter {
 grok {
  match => {
   "message" => "(?m)\[%{TIMESTAMP_ISO8601:timestamp}\] \[%{HOSTNAME:
      host}\] \[%{DATA:thread}\] %{LOGLEVEL:logLevel} %{DATA:class}@%{
      DATA:method}:%{DATA:line} \- \[ANALYTICS\] \[%{DATA:username}\]
      \[%{DATA:microservice}\] \[%{DATA:service}\]"
  }
 }

 date {
  match => ["timestamp" , "YYYY-MM-dd HH:mm:ss.SSS"]
  target => "@timestamp"
 }
}
```

Nous pouvons observer la présence d'un filtre Grok avec une instruction *match*. Si ce filtre est placé en premier, il récupère la ligne de log et essaie de la faire matcher avec le pattern défini dans l'instruction match. Les chaînes en majuscule désigne des constantes représentant les types de données

supportés par Logstash. Par exemple, la première valeur entre crochet du log sera stockée dans le champ timestamp et sera de type TIMESTAMP_ISO8601 alors que la dernière sera stockée dans le champ service et sera de type DATA (c'est-à-dire tout type de caractères). Une fois tous les champs créés, un second filtre *date* permet de parser le timestamp précédemment recueilli pour l'utiliser comme date pour l'événement logstash. Kibana se basera par la suite sur cette date pour construire les visualisations.

Si l'on reprend notre example de log avec NewTransfer, on peut s'apercevoir ici que les informations optionnels ne sont pas encore matchées. Pour ça il faut rajouter un second filtre qui s'en chargera. De manière générale, j'ai créé de nombreux filtres pour chacun des services de l'API. Pour chacun de ceux-là j'ai placé une condition afin qu'ils ne soient appliqués que pour le service correspondant. Ainsi, dans notre exemple, le champ service prend la valeur "NewTransfer" et donc seuls les filtres NewTransfer seront par la suite appliqués.

Une fois tous les filtres exécutés, les données sont structurées et sont envoyées à ElasticSearch qui se charge de les indexer sous la forme de document puis de les stocker en base. Il est possible de préciser, juste après les filtres, l'index qui sera utilisé. Ici j'ai choisi : *api-microservices-YYYY.MM* (où YYYY.MM désigne la date). De ce fait, un index sera créé tous les mois afin de faciliter la recherche des informations sur Kibana.

Création des visualisations sur Kibana

A ce stade, la configuration Logstash est terminée et nos logs sont bien envoyés à ElasticSearch. La dernière étape est donc de créer les différents graphiques exploitant nos données sur Kibana. Le client a seulement spécifié les données qu'il souhaitait pouvoir consulter et non la forme sous laquelle elles devaient apparaître c'est pourquoi j'ai à chaque fois essayé de choisir des graphiques adaptés parmis les types qui existaient (table, camembert etc...). Kibana vient avec une interface web sur laquelle il est possible de choisir une visualisation puis d'y associer une requête ElasticSearch afin de l'alimenter. Afin de pouvoir avoir un aperçu des graphiques rapidement j'ai créé des tests unitaires qui ne faisaient que générer des logs. Ensuite, il était possible d'exécuter ces tests via Spring pour générer autant de logs que j'en avais besoin pour construire mes visualisations sans avoir a réellement envoyer des requêtes vers l'API.

Lorsque l'on créé une visualisation, Kibana met à notre disposition des outils permettant de choisir, par exemple, quel type d'opération on souhaite effectuer sur les données : les compter, prendre le maximum, les additionner etc... ou encore quel type d'agrégation on veut mettre en place. Cela permet de construire les requêtes ElasticSearch qui fourniront les données à afficher. Toujours sur l'exemple de NewTransfer, l'un des besoins était de pouvoir suivre le montant des virements effectués via l'application mobile. Pour cela, j'ai décidé de faire un diagramme de type *donut* dont plusieurs examples sont disponibles en annexe C.1. Ce diagramme est construit sur la requête suivante :

```
{
 "size": 0,
 "query": {
  "match": {
   "service": "NewTransfer"
  }
 },
 "aggs": {
  "aggMontant": {
   "range": {
    "field": "montant",
        "ranges": [{
         "from": 0,
         "to": 200
```

```
        },
        {
          "from": 200,
          "to": 400
        },
        {
          "to": 600,
          "from": 400
        },
        {
          "to": 800,
          "from": 600
        },
        {
          "to": 1000,
          "from": 800
        },
        {
          "from": 1000
        }
        ],
          "keyed": true
    }
   }
  }
}
```

On commence par préciser que l'on ne match que les documents dont le champ *service* a pour valeur "NewTransfer". Ensuite, on réalise une *agrégation* sur les données récupérées. Les agrégations sont des opérations permettant de résumer les données de manière complexe. Ici, elle est de type *range* et a pour but de classer les données selon des intervalles de valeur défini en dessous. On précise ici que le classement s'effectuera sur la valeur du champ montant.

Malheureusement, j'ai rencontré quelques problèmes durant la création des visualisations. En effet, il s'est avéré que certains besoins ne pouvaient finalement pas être satisfait. Par exemple, les métiers souhaitaient pouvoir suivre le nombre d'abonnés qui s'étaient connectés à l'application et trier cela sur des intervals (10 abonnés se sont connectés entre 0 et 50 fois, 15 abonnées entre 50 et 100 fois etc...). Si j'ai finalement pu réussir à construire une requête ElasticSearch pour obtenir de telles données, il n'y avait aucune visualisation Kibana qui pouvait l'accueillir ou la reconstituer. Pour pallier cela j'ai essayé de fournir d'autres visualisations répondant à un besoin se rapprochant le plus possible de celui d'origine afin que le client puisse quand même accéder à certaines données même si cela était moins pratique.

Je pense que la principale erreur était que si la stack ELK permet de effectivement de faire du monitoring, elle n'est pas faite pour du monitoring métier mais plutôt pour du monitoring technique. Par exemple, surveiller les temps de réponses, les erreurs au sein des services, observer les performances etc... De plus, je pense aussi que j'aurais du aller un peu plus en profondeur dans mon étude de Kibana avant d'organiser la réunion avec le client au cours de laquelle nous avons défini les besoins qui seraient satisfait. J'ai découvert par la suite que Kibana propose une option avancée sur ses visualisations avec laquelle il est possible de passer des paramètres en JSON qui seront par la suite mergé avec l'agrégation qu'il réalise. Cependant, je n'ai pas pu appronfondir cette possibilité faute de temps et de documentation à son sujet.

En outre, depuis la création du dashboard, de nouvelles versions de Kibana sont sorties, apportant

de nouvelles visualisations ainsi que de nouveaux outils plus performant pour construire les graphiques. Il pourrait être intéressant de vérifier si certains besoins pourraient maintenant être comblés et, le cas échéant, mettre Kibana à jour. Une autre solution, bien que peu envisageable maintenant, aurait pu être d'utiliser un autre outil que Kibana, compatible avec ElasticSearch comme par exemple Grafana.

3.5.4. Recette et déploiement

Le dashboard étant maintenant terminé, une phase de recette a été organisée avec deux métiers ainsi que mon chef de projet afin que je puisse présenter le travail que j'avais effectué. Des extraits sont disponibles en annexe C.2. Tous les graphiques ont été passés en revue et les métiers ont pu vérifier que chacun des besoins que je m'étais engagé à satisfaire était effectivement présent sur ledit dashboard. Au terme de cette réunion, le dashboard a été approuvé malgré les quelques spécifications qui n'avaient pas pu être satisfaites et nous avons donc décidé de le mettre en place sur tous les environnements à savoir Homo3, RGB et la production. Nous avions déjà schématisé ces environnements figure 3.6 mais ces derniers étaient incomplets. En effet, en réalité, chacun d'entre-eux possède différentes VM dont une dite de *supervision* contenant les outils de monitoring et de supervision et une appelée *microservice* sur laquelle se trouve le code source de l'API microservices. Les environnements RGB et de prod possèdent même deux VM microservices pour le load-balancing. La stack ELK a été installé sur supervision alors que FileBeat a été installé sur microservices (a et b) puisqu'il devait récupérer les logs générés par l'API. J'ai eu la chance de pouvoir procéder moi-même à l'installation de tous les éléments sur RGB. Après cela, je me suis occupé de la maintenance de ces outils afin de corriger les éventuels problèmes qui apparaissaient.

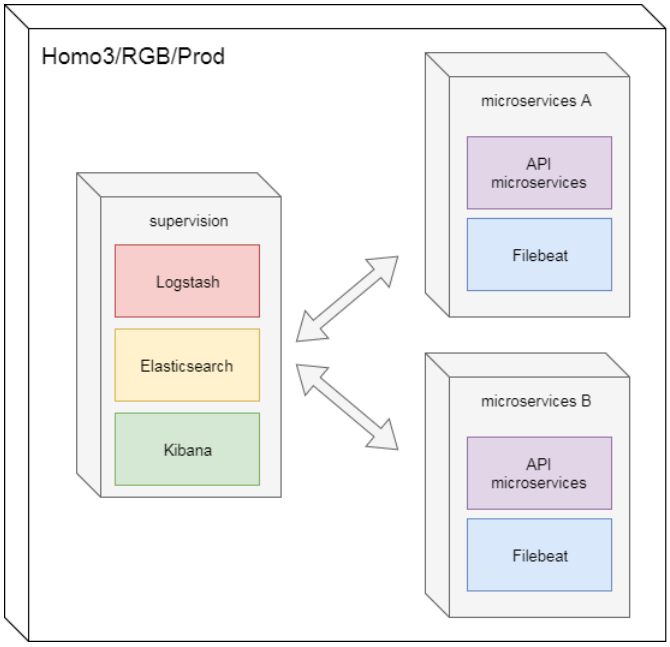

Figure 3.15. – Architecture de la stack ELK sur tous les environnements

J'ai décrit jusqu'ici les principaux travaux que j'ai mené chez Neuflize OBC. Cependant, il reste de nombreuses tâches que j'ai réalisé comme la rédaction d'un guide complet concernant le projet destiné à l'équipe de maintenance qui s'occupera prochainement de nos API puisque celles-ci sont passées en production. J'ai aussi procédé à la réalisation de certaines tâches dites "architectes" via la mise en place de cache grâce à *Redis* ou encore la configuration de RabbitMQ, le message broker, sur les différents environnements. Il s'agissait de petites tâches que j'ai pu mener conjointement avec l'un des architecte du projet, ce qui m'a permis de découvrir une facette de plus du projet (adminsitration via les consoles, TSE etc...) Nous allons maintenant nous intéresser plus en détails au travail que j'ai réalisé chez la Banque Privée 1818.

4. Banque Privée 1818

4.1. Présentation du projet

— Trois canaux : banquier/selection/cgpi
— Entrée en relation, ouverture de comptes titres, souscription AV
— La MIF
— Méthodes agile (sprint, sprint planning, scrum poker, sprint review, TU, qualif, ateliers)
— Daily meeting

4.2. Architecture logicielles

Dans cette partie nous allons présenter l'architecture de l'application web développée chez BP1818 et représentée sur la figure 4.2. Notre équipe est en charge de développer aussi bien la partie frontend que backend de ce projet. Pour cela, différentes technologies ont été choisies peu avant mon arrivée.

Notre backend est constitué de plusieurs couches dont une API Rest, nommée Fronting-Business, développée à l'aide du framework Spring en ayant recours à Spring Boot et au langage Java. Cette API regroupe l'ensemble du code source et expose tous les services qui seront consommés par notre frontend. Une seconde couche API Gateway est présente et assure la sécurité ou encore le routage en implémentant la gateway Zuul de la stack Netflix OSS, exactement comme pour le projet chez Neuflize OBC (3.2.2). Cette gateway est temporairement utilisée durant la phase de développement puis sera remplacée par l'API Gateway d'Axway (3.2.1) tout comme chez NOBC. Il existe une troisième API développée pour les besoins du projet à savoir le moteur de scoring MIF. Il permet de calculer le score d'un client de la banque en fonction de ses réponses aux questionnaires MIF et n'a pas été conçu par nos soins mais par une équipe interne de chez Natixis. Nous parlerons plus en détails de ce moteur et de son intérêt dans la partie 4.3.

Dans le but de persister nos données, nous avons recours à une base Couchbase. Cette dernière est une base de données non structurées de type NoSQL dans laquelle tout est persisté sous forme de document au format JSON. Ces documents sont distribués au travers du cluster Couchbase et stockés dans des conteneurs nommés "buckets". Il s'agit de partitions logiques au sein du cluster permettant de grouper différents documents auxquelles on alloue de la mémoire RAM lorsqu'on les crée. Tous les documents se voient attribuer un ID qui est hashé lorsqu'il sont ajoutés en base ensuite de quoi la table de hashage est mise à jour puis stockée dans la mémoire allouée avec les métadonnées du document ce qui permet de garantir des temps d'accès très faibles. Dans notre cas nous avons un cluster contenant trois buckets :

— references : Contient toutes les données de références. Il s'agit des données métiers qui ne sont pas vouées à évoluer (liste de pays, départements en France, pièces justificatives à fournir en cas d'ouverture de compte, etc...)
— customers : Contient toutes les données des clients (réponses au questionnaire MIF, données personnelles, contrats souscrits etc...)
— customers-test : Bucket créé pour les test unitaires de l'application

Durant ce projet, nous avons adopté un paradigme de programmation réactive. Il s'agit d'un type de programmation à la mode et de plus en plus populaire qui a sans doute représenté, pour moi, la principale difficulté que j'ai rencontré sur ce projet. Il est possible d'avoir recours à des flux d'événements

asynchrones (comme des clics souris) que l'on peut observer pour y réagir. Avec la programmation réactive, les flux sont présents partout, et **tout** peut être un flux, que ce soit les variables, les inputs ou les réponses de requêtes. Ils sont une séquence d'événements ordonnés dans le temps qu'il faut traiter de manière asynchrone en appliquant des fonctions de callback sur les valeurs qu'ils émettent. Cela se rapproche grandement du pattern observer/observable. En effet, afin de pouvoir traiter les événements il faut d'abord écouter le flux (on parle de souscrire à celui-ci) qui est donc l'*observable* puis lui appliquer des fonctions qui sont les *observer*. Les flux sont immutables et peuvent être utilisés de manière chaînée ce qui rend le code très concis et beaucoup plus clair. Concernant les outils, nous avons recours aux bibliothèques *ReactiveX* disponibles dans de nombreux langages (dont java pour backend avec RxJava et javascript pour notre frontend avec RxJS). ReactiveX fournit des méthodes et met en place un modèle basé sur les observables pour manipuler les flux de la même manière que les collections à la mode Java 8. La figure 4.1 illustre un example de programmation réactive.

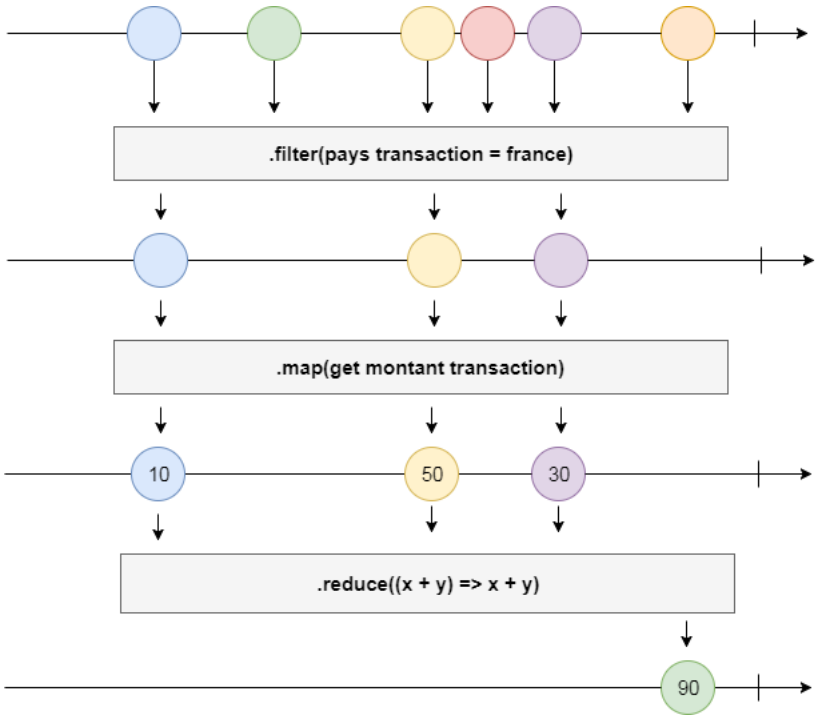

Figure 4.1. – Example de programmation réactive

Les rectangles gris sont les méthodes mises à disposition par ReactiveX pour manipuler les flux. Supposons qu'après un appel vers un service on récupère la liste des transactions bancaires effectuées par un client. Ici nous avons recours à ReactiveX et la programmation réactive pour construire un premier flux contenant lesdites transactions (symbolisées par les boules de couleurs). Une première fonction *filter* filtre le flux afin de ne garder que les transactions effectuées en France, puis les émet dans un second flux. Après cela la fonction *map* transforme chaque objet de type transaction en un entier qui est le montant en euros puis émet ces entiers dans un nouveau flux. Enfin, la méthode *reduce* applique une fonction aux deux premiers éléments puis l'applique entre le résultat obtenu et l'élément suivant. Le résultat final contient le montant total des transactions effectuées en France par le client et ne nécessite que quelques lignes de code. De plus, les flux sont asynchrones et pendant qu'ils émettent le reste du code n'est pas bloqué et continu à être exécuté. Si la page web a souscrit au résultat du montant, elle sera notifiée lorsque celui-ci sera calculé et la vue sera mise à jour sans actions de la part du client.

Enfin, du côté frontend, la technologie choisie se prête à merveille à la programmation réactive puisqu'il s'agit d'Angular4. Ce dernier est un framework très puissant utilisant le langage *Typescript*, une surcouche de Javascript et se base sur certains concepts clés tels qu'une architecture orientée composant, le data binding bidirectionnelle (lier une partie de la vue au contrôleur) ou encore l'injection

de dépendances.

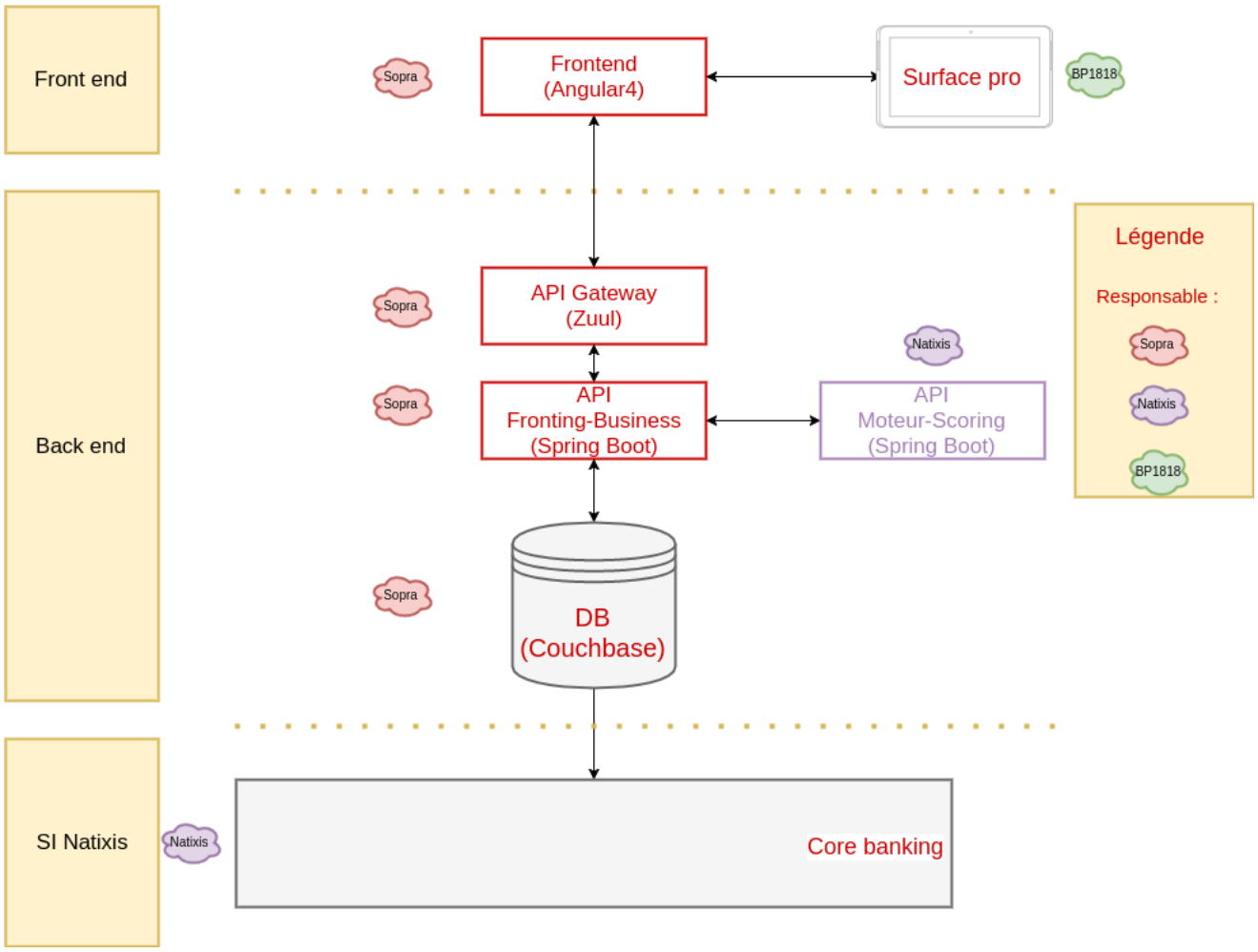

Figure 4.2. – Architecture du Fronting Digital

4.3. Moteur de scoring MIF

4.3.1. Etude du besoin

4.3.2. Backend

4.3.3. Frontend

4.4. Recherche client

4.5. Pièces justificatives

4.6. Sprintance

Sprint 5 ————————————-

correction de bugs sur données perso PP et données perso PM front + back pour la partie objectifs financiers de la mif

issues :

— 52

— 24

— 71
— 69
Sprint 6 ———————————-
issues :
— 62
— 66
Sprint 7 ———————————-
correction de bugs sur données perso PP et données perso PM back pour la restitution du scoring mif mock pour l'api de scoring
issues :
— 94
— 106
— 74
Sprint 8 ———————————-
front + back pour la recherche client
issues :
— 177
— 156
— 153
— 152
— 151
— 149
— 147
— 135
— 134
Sprint 9 ———————————-
reliquat sprint 8 back pour les pièces justificatives
issues :
— 170
— 171
Sprint 10 ———————————-
fin backend pièces justificatives
Moteur de calcul des pièces justificatives côtés frontend

Conclusion et perspectives

TODO

Bibliographie

[1] Article de MARTIN FOWLER - Explications sur l'architecture microservices
`https://martinfowler.com/articles/microservices.html`

[2] NETFLIX OSS - Explications sur la stack Netflix OSS
`https://netflix.github.io/`

[3] Github de ZUUL - Documentation sur Zuul de la stack Netflix OSS
`https://github.com/Netflix/zuul`

[4] Github de EUREKA - Documentation sur Eureka de la stack Netflix OSS
`https://github.com/Netflix/eureka`

[5] Github de HYSTRIX - Documentation sur Hystrix de la stack Netflix OSS
`https://github.com/Netflix/hystrix`

A. Architecture du projet - Neuflize OBC

A.1. Services

Service	Fonction	Description
Account-service	AccountOperations	Liste les opérations effectuées sur un compte bancaire
Account-service	AccountOverview	Liste les informations d'un compte
Account-service	LoanDetails	Liste les informations d'un crédit
Account-service	PositionDetails	Liste les informations d'une position
Account-service	Rib	Liste les informations contenu dans un rib
Profile-service	ClientList	Liste les informations relative au client et à ces racines
Profile-service	TokenInfos	Liste les informations nécessaires à la génération d'un token
Account-service	PortfolioAndLI	Liste les portefeuilles et assurances vie
Account-service	RetrieveBalance	Liste les soldes d'un ou plusieurs comptes d'un client
Transaction-service	AddressBook	Permet de lister les comptes créditeurs d'un client
Transaction-service	NewTransfer	Permet d'initier une transaction
Transaction-service	AccountToDebit	Permet de lister les comptes débiteurs d'un client
Transaction-service	VKB	Permet de générer un clavier numérique virtuel
Transaction-service	SignTransaction	Permet de signer numériquement une transaction
Transaction-service	TransactionOverview	Liste les transactions effectuées par le client

Table A.1. – Fonctionnalités de l'API microservices

A.2. Architecture logicielle

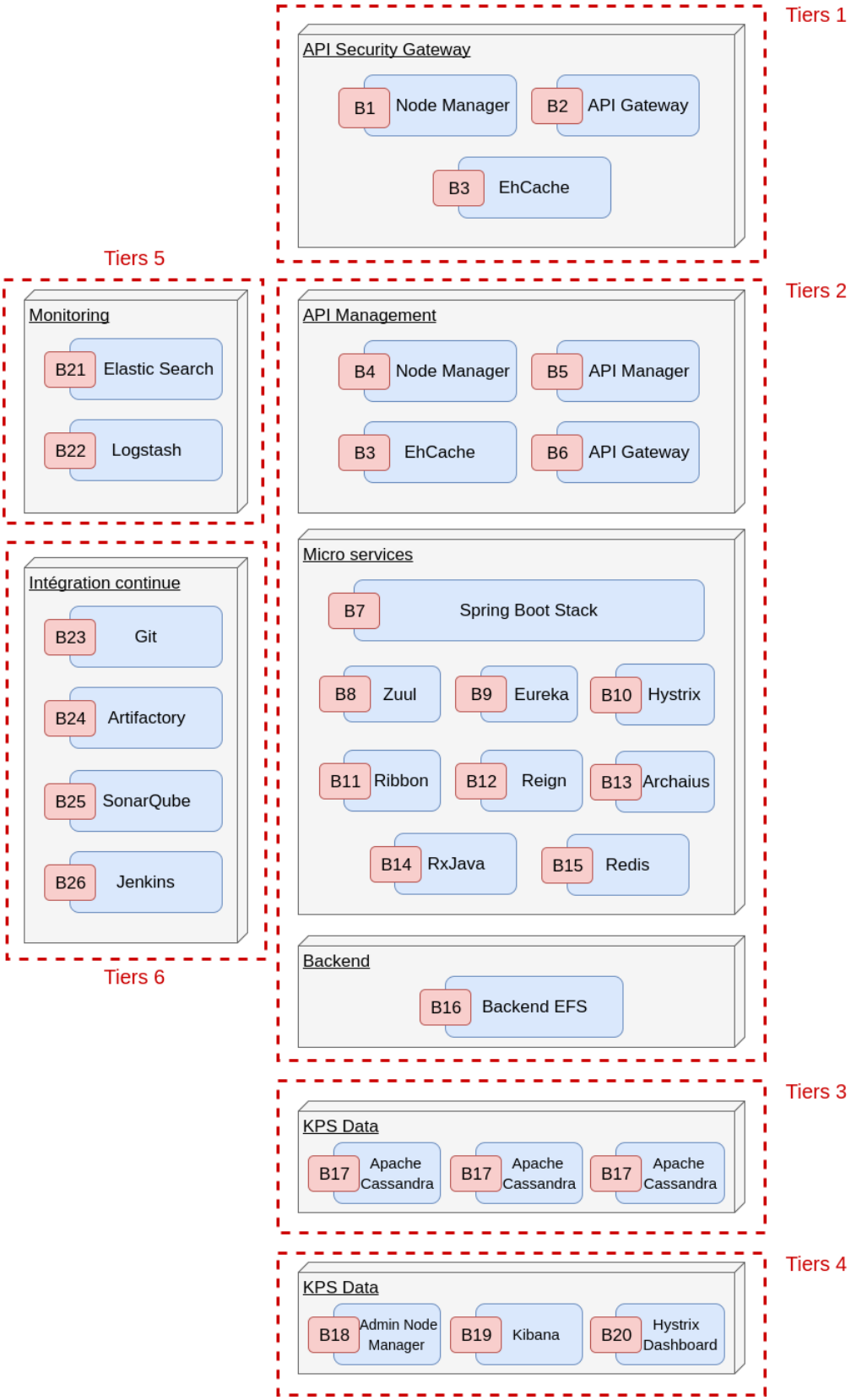

Figure A.1. – Architecture logicielle - Neuflize OBC

B. Tests - Neuflize OBC

B.1. Exemple de scénario de test

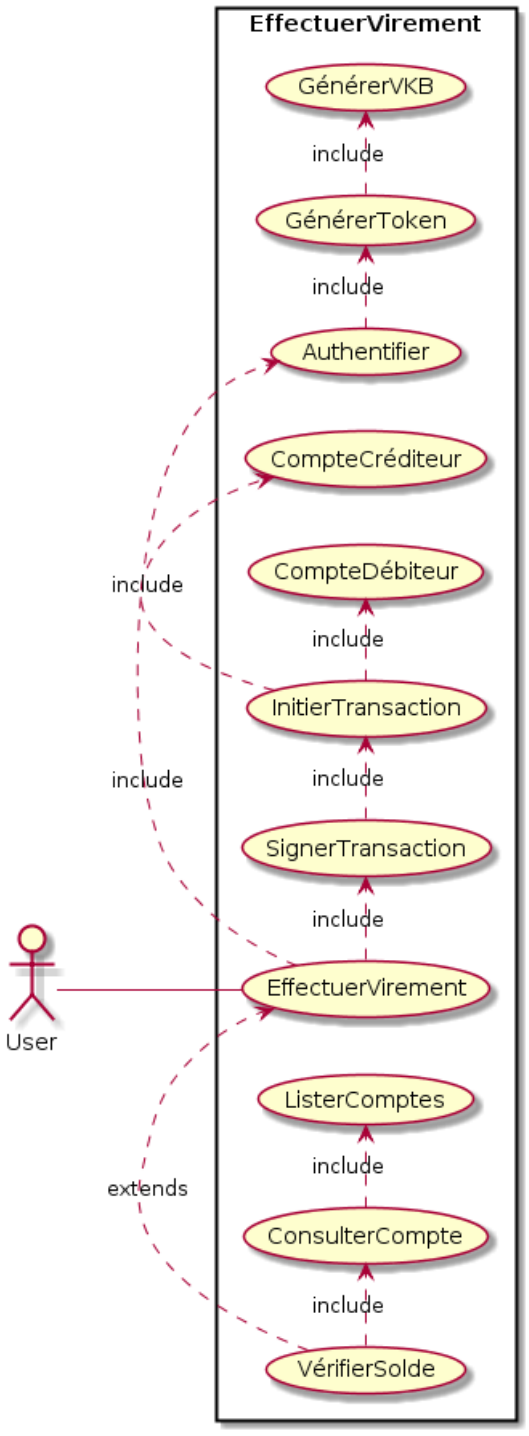

Figure B.1. – Scénario : Transaction bancaire

C. Monitoring applicatif - Neuflize OBC

C.1. Visualisations des transactions

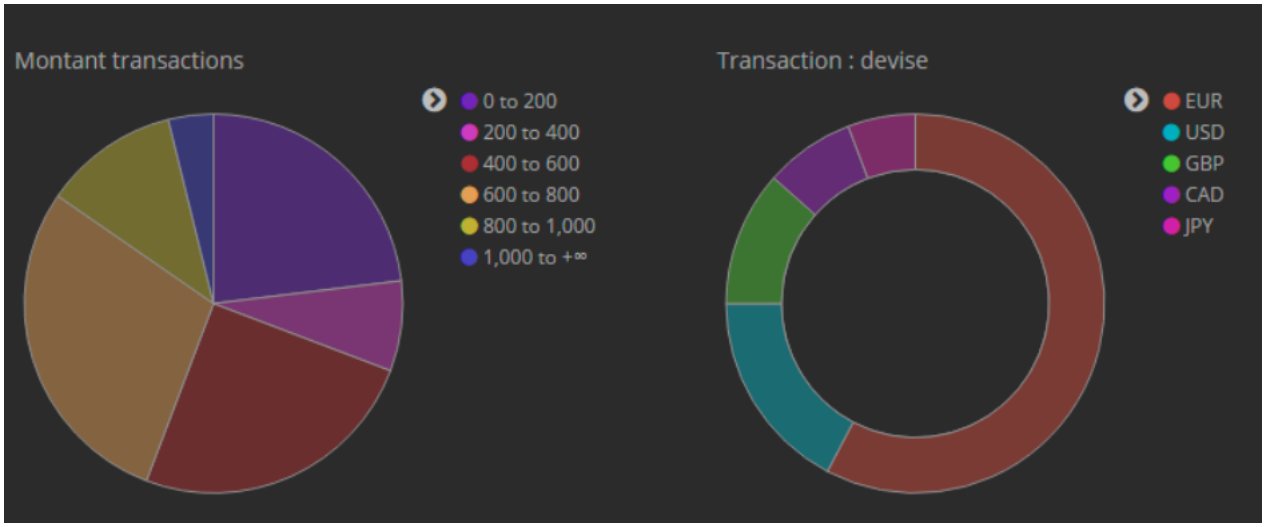

Figure C.1. – Visualisations des transactions

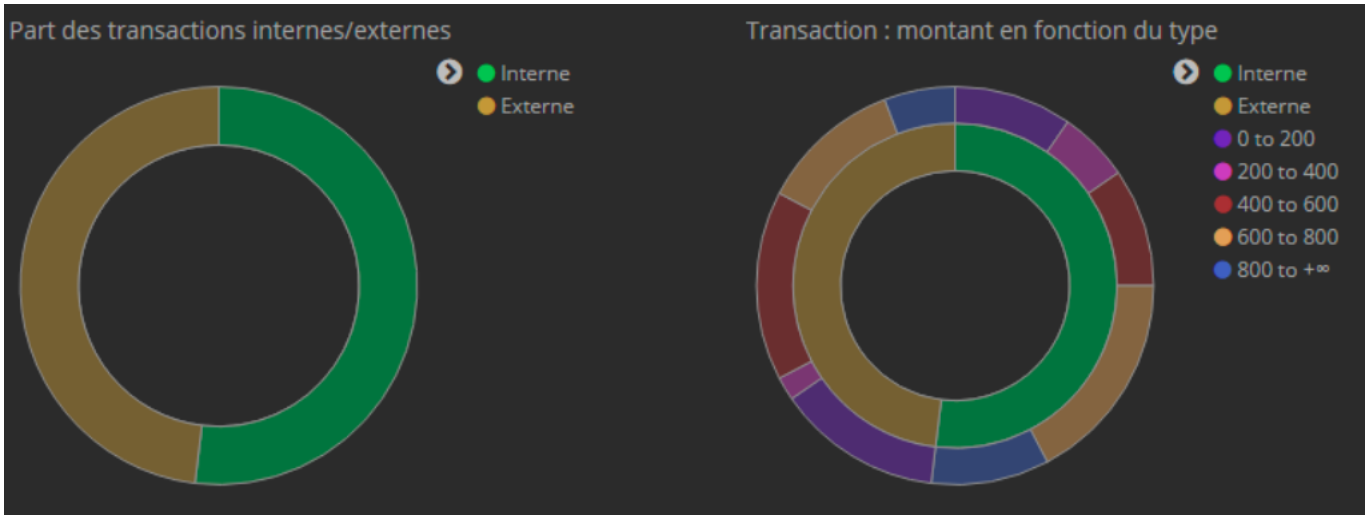

Figure C.2. – Visualisations des transactions

C.2. Extrait du dashboard final

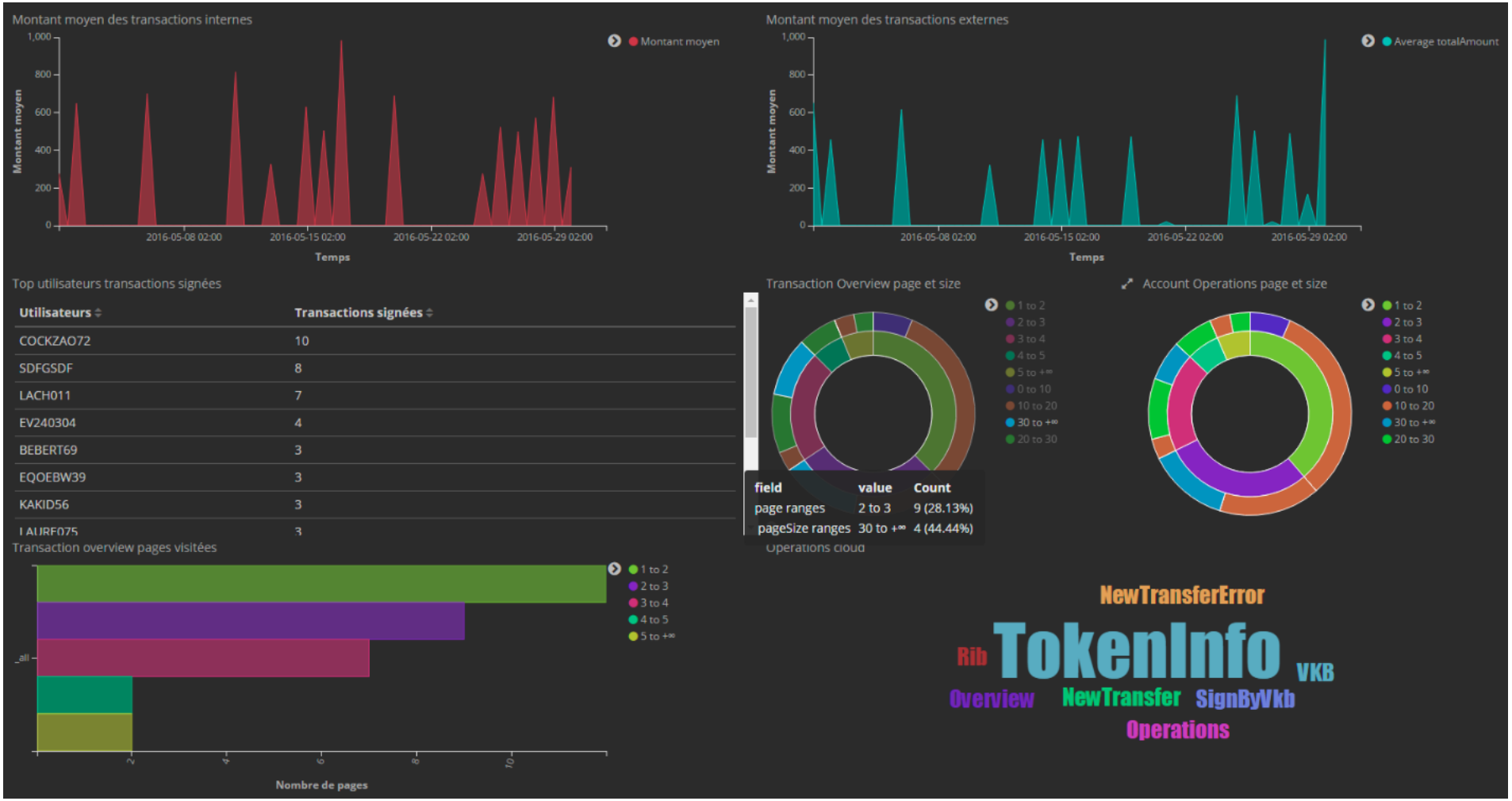

Figure C.3. – Extrait du dashboard n°1

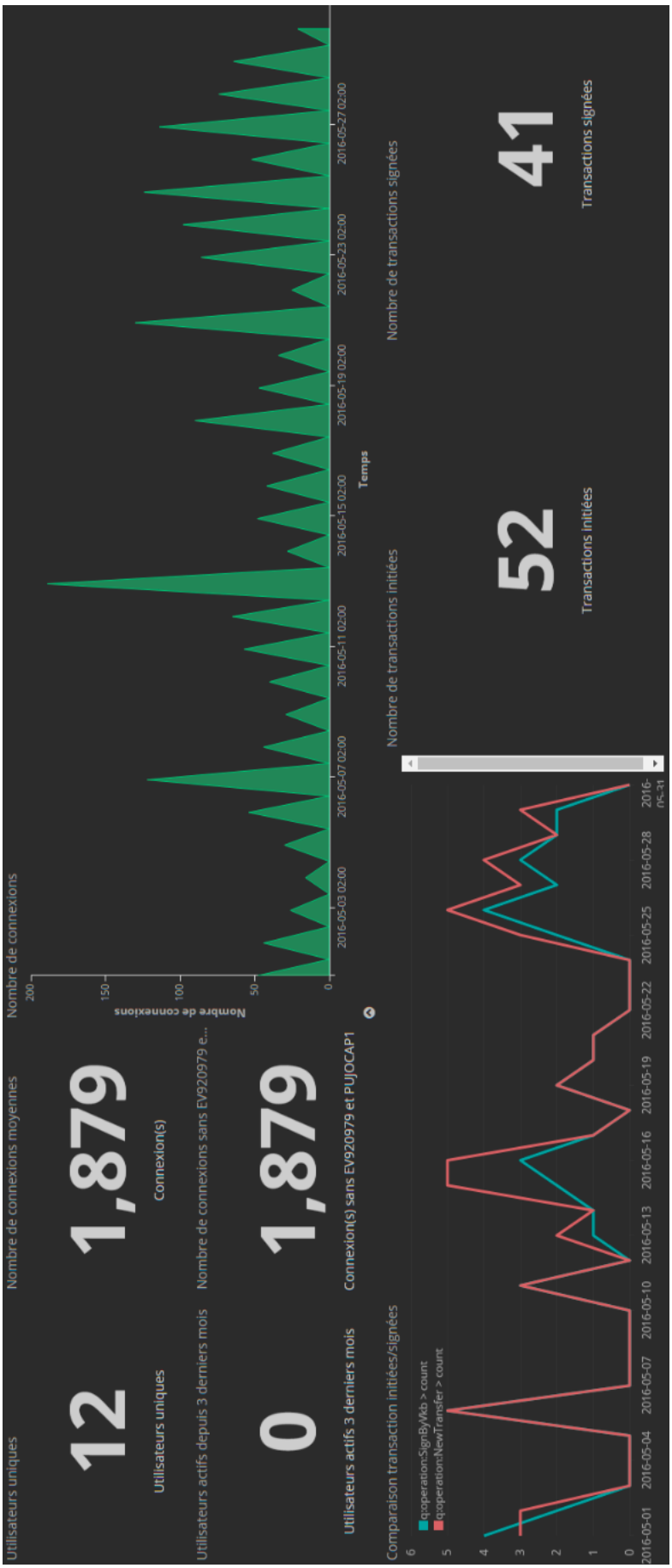

Figure C.4. – Extrait du dashboard n°2