Université Paris-Sud XI

Rapport de stage de fin d'études

Auteur :
Pierre DAL-PRA

Tuteurs :
François-Pierre CHALOPIN pour
eBusiness Information
Sylvain CONCHON pour
l'Université Paris-Sud XI

10 septembre 2012

Remerciements

Je tiens tout d'abord à remercier Messieurs François-Pierre CHALOPIN, Olivier DUVOID et Fabrice REBY, respectivement directeur technique du groupe Excilys, directeur de la société eBusiness Information et directeur du groupe Excilys, pour la confiance qu'ils m'ont témoigné en m'accueillant au sein du groupe.

Je remercie particulièrement Monsieur François-Pierre CHALOPIN, qui a su m'accorder le temps nécessaire en tant que tuteur de stage, ainsi que Monsieur Olivier DUVOID, grâce à qui j'ai beaucoup appris tant sur les aspects fonctionnels du métier d'ingénieur que sur le plan humain.

Je remercie Stéphane LANDELLE, qui par ses compétences, sa pédagogie et sa capacité d'écoute m'a permis de monter en compétences sur des technologies de pointe et des méthodes de développement. Je le remercie également pour l'aide précieuse qu'il m'a apporté durant mon travail sur Gatling et pour la confiance qu'il m'a accordé en me permettant de travailler sur ce projet particulièrement intéressant dont il a la charge.

Je remercie également l'ensemble des stagiaires et employés du groupe Excilys pour leur aide et la bonne humeur dont ils ont fait preuve, rendant mon intégration au sein de l'entreprise aussi simple que possible.

Table des matières

1 Résumé

1.1 Problématiques

Mon stage s'est axé autour de deux problématiques :
- Se former aux technologies Java/JEE et développer une application web d'e-banking en équipe afin de mettre à profit cette formation (2,5 mois)
- Ajouter de nouvelles fonctionnalités à Gatling, injecteur de charge open source développé par Excilys (3,5 mois)

1.2 Intégration à l'existant

Application d'e-banking

Le but de l'application d'e-banking était de mettre à profit la formation initiale et de démontrer notre maîtrise des outils et technologies auxquels nous avons été formés. Elle a fait l'objet d'un nouveau projet, afin de nous permettre de travailler sur chacun des composants de l'application mais également de nous offrir une certaine liberté quant aux choix techniques et d'implémentation.

Gatling

Le projet Gatling existant depuis plusieurs mois, les apports de mon équipe ont naturellement dû s'intégrer à la base de code existante.
Étant donné que nous avons essentiellement réalisé de nouvelles fonctionnalités, l'objectif a été de limiter autant que possible de modifier le code existant, en se rapprochant d'un système de *plugins*.

1.3 Approche adoptée

Application d'e-banking

L'approche adoptée pour développer l'application d'e-banking a été la même pour l'ensemble des stagiaires qui devaient réaliser ce projet :
- Une liste de fonctionnalités à implémenter nous a été fournie
- Nous devions réaliser l'ensemble des fonctionnalités en cinq semaines
- Nous avons travaillé en suivie la méthode de gestion de projets Scrum
- La qualité technique de notre travail a été vérifiée régulièrement

Gatling

L'approche adoptée pour Gatling a été très différente.
Nous avons dû effectuer un important travail de recherche en amont, afin de maîtriser les

nouveaux outils et technologies que nous devions utiliser et maîtriser. Nous avions toujours une liste de fonctionnalités à implémenter, mais ce travail de recherche impliquait que nous pouvions aussi bien très vite réaliser un des fonctionnalités car les concepts à maîtriser ou les librairies à utiliser s'avéraient simples comme cela pouvait prendre plusieurs semaines pour cerner le problème et le moyen de le résoudre.

1.4 Outils utilisés

Application d'e-banking

L'application d'e-banking a été développée en Java et s'est concentrée autour des trois technologies qui ont été au cœur de la formation :

- Maven, outil de build et de gestion de projet
- Spring, framework d'entreprise
- Hibernate, framework de persistance et de mapping objet-relationnel

De nombreuses autres librairies sont venues les compléter, mais l'objectif premier de ce projet restait de maîtriser ces trois technologies, omniprésentes dans les applications d'entreprise actuelles.

Nous avons également utilisé Git pour le contrôle de version et le serveur d'intégration continue Jenkins.

Gatling

Le projet Gatling utilise de nombreuses technologies :

- Scala, le langage de programmation utilisé pour développer l'application
- Akka, toolkit de développement d'application concurrentes et distribuées
- Netty & et Async HTTP Client, librairies permettant d'effectuer des requêtes HTTP asynchrones
- Metrics, toolkit facilitant l'ajout de métriques à une application et l'analyse de ces métriques via des outils de reporting

Maven a été également utilisé pour le projet Gatling.

1.5 Apport personnel

Application d'e-banking

Comme les autres membres de mon équipe, j'ai pris part à l'ensemble des phases du développement de l'application d'e-banking :

- Choix des fonctionnalités à implémenter à chaque sprint
- Choix d'architecture et des librairies à utiliser si besoin
- Développement et test du code produit

Gatling

Comme pour l'application d'e-banking, j'ai pris part à l'ensemble des phases de développement du développement des nouvelles fonctionnalités de Gatling, en concertation avec le responsable du projet Gatling.

1.6 Etat du projet à la fin du stage et perspectives

Application d'e-banking

L'application d'e-banking n'ayant qu'un but de formation, cette application est terminée et ne sera à priori plus modifiée.
Elle sert cependant toujours de référence technique, de par le nombre de technologies que nous avons mises en œuvre dans le cadre de ce projet.

Gatling

Plusieurs des fonctionnalités souhaitées sont déjà réalisées et seront très prochainement intégrées dans une version stable de Gatling,d'autres sont encore en cours d'écriture mais devraient être finies d'ici la fin de mon stage.
Je compte également poursuivre ma contribution au projet Gatling après la fin de mon stage, sur mon temps libre.

2 Entreprise et environnement de travail

2.1 Présentation de l'entreprise

Le groupe Excilys regroupe sept SSII, toutes concentrées autour des technologies Java/JEE :

- eBusiness Information
- Altendis
- SS2J
- Equitalis
- Adlys
- Visual3X
- eAdvance

2.1.1 L'excellence technique

Le groupe Excilys a fait le choix de propser à ces clients (parmi lesquels on peut des sociétés du CAC 40) des consultants à la tarification certes plus élévée, mais dont la compétence et le savoir technique sont supérieurs à la concurrence.

Cette orientation se retrouve dans le mode de recrutement du groupe Excilys : la plupart des consultants sont recrutés dans le cadre de leur stage de fin d'études.
Le groupe Excilys utilise en effet cette période pour compléter leur formation à la fois théorique et pratique et souvent leur donner en plus une première expérience de projet client.

Le groupe a également fortement investi dans un système de capitalisation des connaissances développé en interne - Capico - , pour que chaque consultant puisse partager ses connaissances techniques, consulter les fiches existantes ou encore évaluer ses compétences. Etant donné la stratégie du groupe résolument tournée vers l'excellence, ce système de partage des connaissances est fondamental et focalise un budget conséquent de recherche et développement.

Le groupe Excilys intervient auprès de ses clients aussi bien pour du développement que de l'audit ou de la formation.

2.1.2 Le Service Equitable

Au-delà de tous les outils qui sont mis à disposition des consultants du groupe, c'est avant tout la Charte qui constitue sa véritable identité et fédère à la fois ses consultants et ses clients. Le modèle des plus singuliers sur la place est pourtant des plus pragmatiques.

Des règles de bonne conduite, la volonté que chaque acteur soit respecté dans ses attentes et respecte lui aussi les autres et une meilleure répartition des rémunérations entre tous, notamment pour le consultant.

FIGURE 1 – Le Service Equitable

Pour créer cette rétroaction entre le client et les consultants, le groupe a ainsi adopté la règle dite des 60/40, qui permet de reverser aux consultants hors frais commerciaux sous forme de primes 60% des facturations. Les consultants du groupe bénéficient ainsi de rémunérations élevées, mais en accord avec ce que l'implication et la qualité de leur travail apporte.

2.2 Environnement de travail

L'environnement de travail est celui d'une SSII « classique » : l'open space, afin de faciliter la communication entre développeurs, au coeur de la méthode Scrum qui est la méthode de travail privilégiée par le Groupe.
Comme indiqué précedemment, le mode de recrutement d'Excilys se basant principalement sur le recrutement de stagiaires en fin d'études, une vingtaine de stagaires ont été pris cette année. Cela m'a conduit avec travailler surtout avec des personnes dans la même situation que moi, ce qui a grandement facilité la communication et le partage de connaissances, car nous avions tous à y gagner : la société se prépare à éventuellement engager l'ensemble des stagaires, il n'y a donc pas eu de phénomène de compétition

entre stagiaires, ce qui aurait pu conduire à un cadre de travail délétère.

Le groupe Excilys restant malgré tout une SSII de petite taille (environ une centaine de personnes), je n'y ai pas trouvé le type de hiérarchie pyramidale commun aux grandes SSII. Au contraire l'ensemble des directeurs a été très disponible et à l'écoute.

Globalement, le cadre de travail fut des plus agréables et ce fut pour moi un réel plaisir de travailler pendant ces 6 derniers mois avec l'ensemble du personnel.

2.2.1 Intégration au sein de l'entreprise

Les informations fournies au début du stage m'ont permis de comprendre et de gérer rapidement les aspects administratifs du stage (horaires, feuilles de temps, congés. . .) ainsi que de démarrer rapidement la formation de 2 mois.
Cette longue phase de formation a conduit à une intégration tardive dans une équipe, puisque celle-ci n'est intervenue qu'à la fin du premier mois, lors du démarrage du projet en interne (6 personnes). Les objectifs de travail ont été clairement énoncés dès le départ :

- Suivre la formation Capico lors du premier mois.
- Réaliser les fonctionnalités demandées durant le projet en interne le second mois.
- Une fois l'équipe Gatling intégrée, Réaliser les nouvelles fonctionnalités demandées.
- Passer la certification OCPJP durant le stage (obtenue avec 91% de bonnes réponses)

Le support a été essentiellement Capico, qui a servi à la fois de livret d'accueil et de plateforme de formation.

L'intégration au sein de l'entreprise a été grandement simplifiée par les employés d'Excilys, toujours disponibles lorsque j'ai pu rencontrer un problème technique.

3 Formation et application d'e-banking

3.1 Introduction

Comme indiqué précédemment, la phase de formation a occupé presque la moitié de mon stage.
Cela peut paraître conséquent pour une phase de formation, mais cela fait partie intégrante de l'approche du groupe Excilys concernant les stages de fin d'études.

En effet, le stage est orienté dans une optique d'embauche et Excilys tient à ce que les stagiaires et futurs salariés aient une très forte compétence technique sur les technologies Java/JEE, car c'est le segment de marché visé par le groupe : des clients recherchant des consultants avec un excellent niveau technique, qui mèneront à bien le projet qu'on leur confie à coup sûr.

Seulement, ce niveau de compétence ne peut s'obtenir sans difficultés ou sans y consacrer un temps important. C'est donc pour cette raison que le groupe Excilys a choisi de consacrer autant de temps à cette phase de formation, afin que les stagiaires,futurs consultants, puissent acquérir ces compétences qui leur sont demandées.
Le groupe est d'ailleurs très vigilant sur ce point : l'entraide entre stagiaires et consultants est particulièrement encouragée, afin de ne laisser personne en difficulté et les stagiaires sont soumis un mois avant la fin de leur stage à deux entretiens :

- un entretien « d'ingénieur », vérifiant que les stagiaires seront en mesure de s'exprimer et de présenter leur travail lorsqu'ils seront en mission chez un client
- un entretien technique, vérifiant que les technologies au cœur de la phase de la formation sont parfaitement maîtrisées.

3.2 Formation Capico

La première phase de la formation s'est déroulé sur Capico, plateforme d'e-learning développée par Excilys. J'ai avons été formés à de nombreuses technologies :

- Java/JEE6
- UML
- Maven
- Spring
- Hibernate

Nous avons également été formé à la méthode *Extreme Programming*, dont certains principes ont été mis en oeuvre pendant le développement de l'application d'e-banking tels que l'intégration continue ou la programmation en binôme.

Cette phase de formation a également donné lieu à plusieurs *speechs* donnés par Stéphane LANDELLE, directeur technique d'Excilys, durant lesquelles il a partagé avec nous son expérience du développement Java/JEE afin de nous parler de bonnes pratiques de développement ou de nous présenter plus en détail les technologies au coeur de la formation.

La formation Capico m'a permis de prendre en main ces technologies complexes, avant de les mettre en œuvre dans un projet réel : l'application d'e-banking.

3.3 Application d'e-banking : MF Banking

3.3.1 Fonctionnalités à implémenter

Priorité	En tant que XXX...	je peux...
1	client authentifié	Consulter sur ma page d'accueil le solde de mes comptes espèce
2	client authentifié	Consulter le détail d'un compte espèce avec les détails des opérations pour un mois donné. Les opérations carte sont cumulées sur une seule ligne.
3	client authentifié	Consulter le détail des opération carte pour un mois donné
4	client authentifié	Réaliser des virements internes
5	client authentifié	Réaliser des virement externes
6	internaute	Accéder à ma page de login
7	internaute	Me logger
8	client authentifié	exporter au format Excel le relevé des opérations pour un mois donné. Les opérations ne sont pas cumulées.
9	client authentifié	Consulter l'historique de mes virements
10	client authentifié	Consulter sur ma page d'accueil l'encours carte sur chacun de mes comptes espèce
11	client authentifié	Consulter sur ma page d'accueil le solde prévisionnel de chacun de mes comptes espèce
12	client authentifié	Accéder au détail des opérations carte par un clic sur la ligne de cumul dans le détail des opérations compte

3.3.2 Méthodologie

Scrum

Scrum est une méthode de gestion de projet, de la famille des méthodes *agiles*.
Les méthodes agiles sont nées du constat que de nombreux projets suivant les méthodes « conventionnelles » (telles que le cycle en V) n'étaient pas menés à terme ou généraient d'importants surcouts. Même quand ces projets étaient menés à terme, ils pouvaient être un échec car le besoins du client avaient changé et ces méthodes étaient peu adaptées au changement.

Les méthodes agiles telles que Scrum intègrent le client au cœur du processus de développement en se basant sur des itérations courtes(*sprints*), se concentrant sur un nombre limité de fonctionnalités, au terme desquelles les développeurs doivent pouvoir livrer au client une version incomplète mais parfaitement fonctionnelle. Celui-ci peut alors apprécier le résultat et demander des changement si besoin est. L'approche choisie par les méthodes agile est d'être réactif aux changements et de s'y adapter plutôt que d'essayer de tout prévoir parfois des années à l'avance.

Le client voit donc son projet évoluer d'itération en itération, au lieu de passer directement d'un cahier des charges à une application terminée.
La méthode Scrum repose principalement sur deux documents :
- le *product backlog*,liste priorisée des fonctionnalités(ou *items* à réaliser
- le *sprint backlog*, liste des fonctionnalités à réaliser durant le *sprint*

Il est conseillé d'avoir de petites équipes (une vingtaine de personnes maximum), afin de privilégier la communication entre membres de l'équipe et d'éliminer le besoin de réunions formelles, et auto-organisées, afin de responsabiliser l'équipe (la réussite du projet est la réussite de l'équipe et non d'un chef de projet, mais cela vaut aussi en cas d'échec !) Le déroulement d'un *sprint* est le suivant :
- en accord avec le *Product Owner* (responsable du projet chez le client), on sélectionne les fonctionnalités (qu'on découpe en tâches) qui seront réalisées durant le sprint et on estime le « coût » de chaque tâche (temps, difficulté...)
- Quotidiennement se tient le *daily scrum*, qui doit être très bref (environ 1 minute par personne) où on présente ce qu'on a fait la veille, les problèmes rencontrés et ce qu'on pense faire le jour même. Le mieux est de faire du *daily scrum* une routine, en le tenant à chaque fois au même endroit et à la même heure.
- En fin de sprint, on présente au *Product Owner* le résultat du sprint, en ne montrant que ce qui a été terminé. Se tient ensuite la *rétrospective* du sprint , où on fait le point sur l'ensemble du sprint, ce qui est allé, ce qui n'allait pas, les pistes d'amélioration...

Dans le cadre de notre application d'e-banking, MF Banking, notre application de la méthode Scrum a été la suivante :

- Notre équipe était constituée de 6 personnes
- Les *sprints* duraient une semaine, du mercredi au mercredi et le projet s'est étalé sur 5 sprints
- Le *daily scrum* se tenait à 10h
- La revue technique de notre code par le directeur technique d'Excilys avait lieu le mardi
- La revue de l'application par le *Product Owner* avait lieu le mercredi

La méthode Scrum impose également définir les conditions pour considérer qu'une tâche est terminée (Definition de « Fini Fini »). Dans le cadre de l'application d'e-banking, notre définition de *Fini Fini* était la suivante :

- Développé
- Testé (en local et sur le serveur d'intégration)
- Refactoré
- Revu par un pair
- Documenté (Javadoc)

Intégration Continue

L'intégration continue est un processus de développement issu de la méthode *Extreme Programming* (XP). L'un des principes de XP étant de pouvoir à tout moment fournir une version livrable et fonctionnelle au client et l'intégration continue contribue largement au respect de ce principe.

L'intégration continue repose sur quelques grands principes :

- Utiliser un logiciel de contrôle de versions
- Automatiser le processus de build : des outils comme Maven permettent de compiler, tester et packager le code en une seule commande. On peut également utiliser des serveurs d'intégration continue qui détectent toute modification du dépôt de code et lance automatiquement une build le cas écheant
- La build doit être « auto-testante » : Dès que le code est compilé, les tests doivent être automatiquement exécutés
- La compilation doit rester courte, pour que le résultat soit toujours rapidement visible
- Ne pas garder trop de différence entre la version locale et la version sur le dépôt : Ne pas intégrer très régulièrement les modifications peut poser de gros problèmes, l'idéal est de *commit* ses modifications tous les jours
- Le résultat de la build doit être visible par tout le monde et les livrables produits facilement accessibles

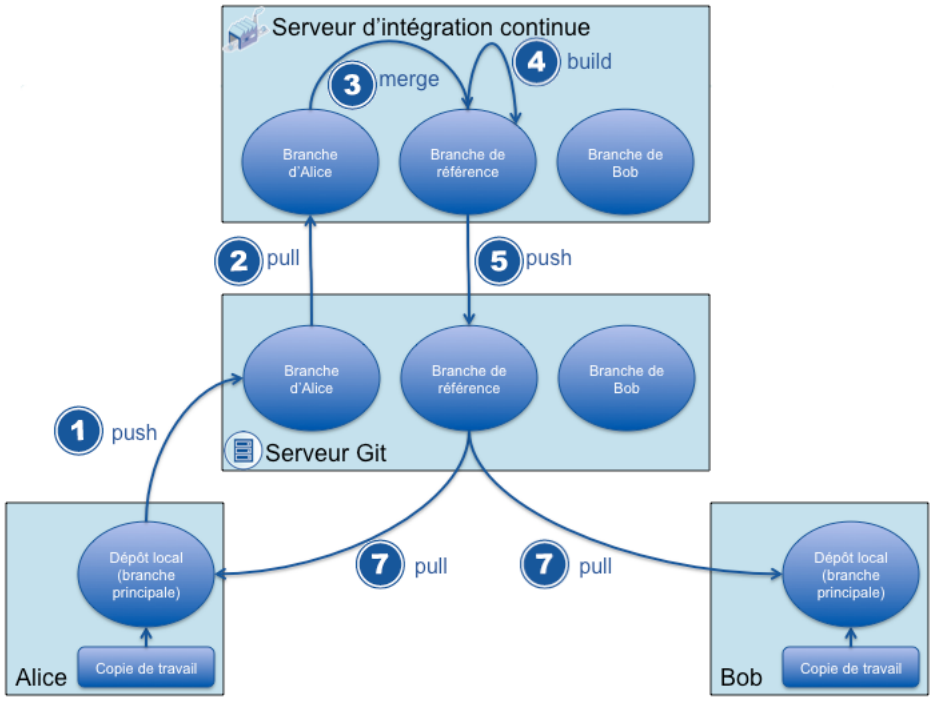

FIGURE 2 – Worflow dans un système de build incassable

Build incassable

Le principe de la build incassable est en quelque sorte une extension de l'intégration continue et repose d'ailleurs sur les outils de l'intégration continue : le contrôle de versions et le serveur d'intégration.
Le but de la build incassable est de conserver en permanence le *trunk* (la branche principale) exempt de bugs, en interdisant aux développeurs de mettre directement à jour le *trunk*.

Le principe est le suivant :

1. Chaque développeur dispose d'une *branche*, une version parallèle des sources du dépôt. Quand des modification ont été effectuées et doivent être synchronisées avec le *trunk*, le développeur *push* ses modifications sur sa branche.
2. Le serveur d'intégration continue détecte les modifications apportées sur la branche du développeur, et récupère ses modifications
3. Le serveur d'intégration fusionne alors ces modifications avec sa copie locale du

trunk

4. Une build est alors tentée, afin de vérifier que le code compile bien et que l'ensemble des tests passent...
5. Si la build réussit, le serveur d'intégration propage alors les modification sur le trunk
6. Les développeurs peuvent alors se synchroniser avec le *trunk*, en étant assuré que l'ensemble est parfaitement fonctionnel

L'inconvénient de la build incassable est qu'elle repose sur le respect de bonnes pratiques de codage de la part des développeurs, notamment l'écriture de tests exhaustifs.
En effet, hormis un problème de configuration, un code non testé mais qui compile est une build réussie. Mais un test qui ne passe pas provoque l'échec de la build. Si le code est trop peu ou mal testé, le système de la build incassable perd tout son intêret puisque la build réussira et que du code « incorrect » se retrouvera alors dans la branche principale, pourtant censée ne contenir que du code sûr sur lequel les autres développeurs peuvent s'appuyer sans crainte.

Dans le cadre de MF Banking, nous avons mis en place un système de build incassable (et d'intégration continue), en se basant sur le système de contrôle de versions Git et le serveur d'intégration Jenkins.
A défaut de pouvoir déclencher automatiquement une build lorsqu'une modification était propagée sur le dépôt pour des raisons de configuration réseau, une build était déclenché automatiquement toutes les 15 minutes.

La procédure que nous avons suivi pour mettre en place la build incassable est détaillée sur le blog du cabinet d'expertise informatique Octo [1].

3.3.3 Organisation des sprints

- Sprint 0 et 1 :
  - Mise en place de l'environnement de développement : Git, Jenkins, SGBD Postgresql et serveur d'applications Tomcat
  - Mise en place des conventions de codage (nom des classes et méthodes, formatage du code
  - Item 1 : *Consulter sur ma page d'accueil le solde de mes comptes espèce*
  - Item 6 : *Accéder à ma page de login*
  - Item 7 : *Me logger*
  - Mise en place d'une page d'administration basique, pour démontrer le bon fonctionnement du système de rôles
- Sprint 2 :
  - Item 2 : *Consulter le détail d'un compte espèce avec le détail des opérations pour un mois donné. Les opérations carte sont cumulées sur une seule ligne*

1. http ://blog.octo.com/gestion-de-version-distribuee-et-build-incassable/

  - Item 3 : *Consulter le détail des opérations carte pour un mois donné*
  - Item 12 : *Accéder au détail des opérations carte par un clic sur la ligne de cumul dans le détail des opérations compte*
- Sprint 3 :
  - Item 4 : *Réaliser des virements internes*
  - Item 5 : *Réaliser des virements externes*
  - Item 8 : *Exporter au format Excel le relevé des opérations pour un mois donné. Les opérations ne sont pas cumulées.*
  - Item 9 : *Consulter l'historique de mes virements*
  - Item 10 : *Consulter sur ma page d'accueil l'encours carte sur chacun de mes comptes espèce*
  - Item 11 : *Consulter sur ma page d'accueil le solde prévisionnel de chacun de mes comptes espèce*
- Sprint 4 :
  - Ajout de Web Services REST et SOAP
  - Amélioration de la zone d'administration : ajout de clients, passage de virements...
  - Mise à jour automatique du solde, du solde prévisionnel et de l'encours carte
  - Réalisation de deux applications web et d'une application Android pour tester les Web Services

3.3.4 Technologies utilisées

Maven

Maven est un outil de gestion de projet développé par la fondation Apache.
A la différence d'outils tels que Make ou de Ant, son équivalent Java, qui exige de définir précisement les étapes nécessaires à la compilation du code et à la construction du binaire qui en résulte, Maven se base sur un ensemble de conventions (notamment pour l'emplacement du code source, des tests, des ressources de l'application...) qui, si elles sont respectées, permettent de compiler les sources, éxécuter les tests et packager le code compilé avec une configuration minimale. En ce sens, configurer un projet pour qu'il puisse être construit par Maven revient à décrire le projet en lui-même plus qu'à décrire son processus de compilation.

Maven permet également de gérer les dépendances d'un projet (les librairies à utiliser) de façon simple et efficace et son système de plugins permet de l'adapter aux besoins du projet en toute circonstances.

Maven se révèle être un outil indispensable, y compris sur des projets de taille réduite. Etant donné le nombre important de librairies utilisées pour l'application MF Banking,

le développement de celle-ci se serait averé beaucoup plus complexe sans l'apport de Maven et son système de gestion de dépendances.

Spring

Spring est un framework destinée aux application d'entreprise, développé par SpringSource.Spring est né en 2003, suite à de multiples déboires avec la solution pour les applications d'entreprise de l'époque, JEE et le EJB.

Le coeur du framework Spring est son conteneur IoC (*Inversion of Control*). D'une certaine façon, l'IoC (ou, selon l'article de Martin Fowler sur IoC [2], l'injection de dépendances) est une alternative à l'opérateur *new* : on définit **par configuration** quelles classes classes devront être instanciées,configurées et gérées par le conteneur (on appelle alors ces instances des *beans* et ces *beans* peuvent être ensuite automatiquement « injectées » dans les classes de l'application à développer et être utilisées telles quelles,sans qu'il y ait besoin de créer soi-même une nouvelle instance.
Dans le cas où la classe que l'on souhaite injecter implémente une interface, il est tout à fait possible et même conseillé de ne faire référence qu'à l'interface, le conteneur Spring se chargera alors d'injecter l'implémentation automatiquement. Cela permet donc d'avoir un couplage réduit au strict minimum, puisque les classes n'ont même pas connaissance de l'implémentation sous-jacente,uniquement connue du conteneur IoC de Spring.

L'interêt du framework Spring ne se limite pas à son conteneur IoC et il propose également de nombreuses intégrations avec d'autres technologies, telles que Hibernate/JPA.

Hibernate et JPA

Hibernate est un ORM (Object-Relationnal Mapper). L'interêt d'un ORM est de s'abstraire du SQL en laissant celui-ci se charger des requêtes SQL à proprement parler. Ceci est possible en configurant la manière dont un objet Java est liée à la base de données (quelle classe correspond à quelle table ? quel variable d'instance correspond à quel attribut ?) et la manipulation de l'objet Java geré par l'ORM aboutira alors à l'éxécution automatiques des requêtes SQL appropriées.
JPA (ou Java Persistence API), API issue des spécifications JSR-224 et JSR-317, est justement une manière de configurer la façon dont une classe Java est liée à la base de données, par le biais d'annotations.

2. http ://martinfowler.com/articles/injection.html

CXF

CXF est un framework développé par la fondation Apache, simplifiant l'écriture de Web Services.

On peut voir les Web Services comme une moyen de communication entre applications. Un exemple typique d'utilisation des Web Services est le comparateur de prix. Un comparateur de prix ne va pas stocker directement toutes les informations des différents sites qu'il cible, pour des raisons évidentes : sécurité des informations, synchronisation des bases de données,etc...
La solution est de faire appel à des Web Services,exposés par les sites marchands, que le comparateur de prix requêtera afin d'obtenir les informations souhaitées. La logique métier de gestion des stocks, des tarifs...restent donc du côté de l'application requêtée.

Il nous a été demandé d'implémenter des Web Services SOAP et REST pour MF Banking, et nous avons utilisé le framework CXF pour les réaliser.

3.3.5 Architecture

L'architecture de MF Banking repose sur le modèle de l'architecture N-tiers. Le principe de cette architecture consiste à découper l'application en plusieurs couches (le plus souvent trois couches), chacune ayant un but très précis. Ce découpage présente deux avantages majeurs :

- chaque couche ayant un but bien précis, celle-ci est plus facile à maintenir et est aisément découplable du reste de l'application.
- Les couches pouvant facilement être découplées, les couches sont interchangeables

Il en résulte une application hautement modulaire, où chaque couche peut (théoriquement) être remplacée ou mise à jour en ayant un impact réduit sur le reste de l'application.

Dans le cas d'une architecture 3-tiers, que nous avons utilisé pour MF Banking, les couches sont les suivantes :

Le rôle de la couche d'accès aux données est, comme son nom l'indique, d'accès aux données de l'application. Typiquement, cette couche se charge d'accéder à une base de données et d'y récupérer les informations ou d'en stocker des nouvelles.
Dans notre cas, la couche d'accès aux données accédait à une base de données PostgresSQL, via JPA et Hibernate.

La couche de logique métier est le pivot de l'application : c'est dans cette couche qu'est gerée toute la logique propre à l'application que l'on souhaite développer, et sert donc de pont entre les données et la façon dont elles sont présentées à l'utilisateur.

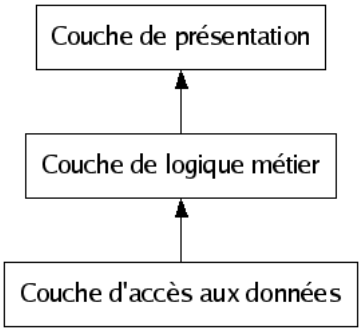

FIGURE 3 – Architecture 3-tiers

Enfin, la couche de présentation est celle qui est en charge d'afficher les données de l'application.
Dans notre cas, la couche de présentation consistait en une série de pages JSP, gérées par Spring MVC.

Les Web Services SOAP et REST que nous avons mis en place était des couches de présentation alternatives, réutilisant la couche de logique métier.

Les frameworks permettant l'injection de dépendances tels que Spring simplifient grandement la mise en place d'architecture n-tiers.
En déléguant le « câblage » automatique des implémentations au conteneur Spring et en utilisant systématiquement des interfaces (afin de ne pas dépendre directement d'une implémentation), remplacer une implémentation par une autre revient finalement à fournir la bonne archive JAR contenant les implémentations désirées.

3.3.6 Schéma de la base de données

Rôles de chaque table :
- Person : liste des clients de MF Banking
- Role : liste des rôles que peuvent avoir les utilisateurs (client ou administrateur)
- Authority : table de jointure entre Person et Role. Il était nécessaire d'avoir une table de jointure car un utilisateur peut être à la fois utilisateur et administrateur,
- Compte : liste des comptes des utilisateurs
- Person_Compte : table de jointure entre Person et Compte. Ici aussi, il était nécessaire d'utiliser une table de jointure car un utilisateur peut avoir plusieurs comptes et les comptes peuvent être des comptes joints
- Operation : liste des opérations bancaires effectuées sur les comptes
- OperationType : liste des types d'opérations bancaires (chèque, carte bleue, virement et espèce)

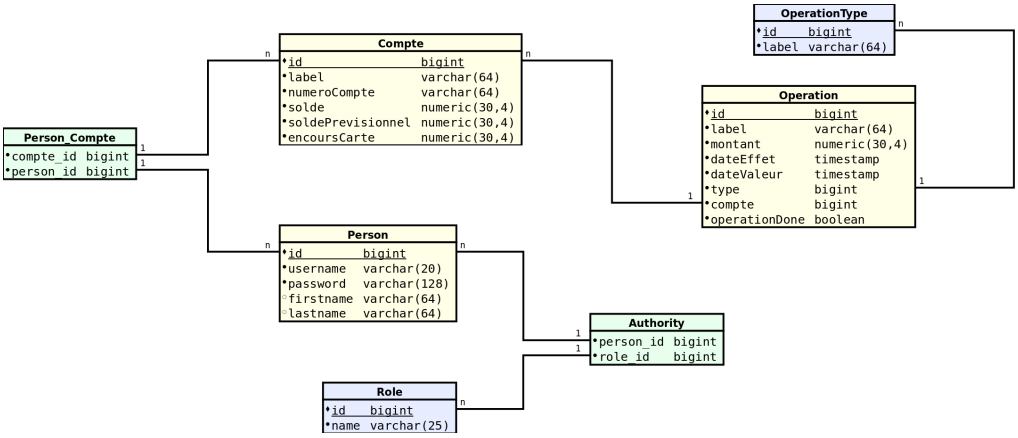

FIGURE 4 – Modèle E/A de la base de données

3.3.7 Structure

Par le biais de Maven, nous avons découpé notre application en plusieurs modules :
- Entities
- DAO
- Services et Services API
- Batch
- Web
- Webservices
- Application Android

module Entities

Ce module regroupe les différentes entités de notre application, miroir objet de notre base de données ainsi que les scripts Liquibase de mise à jour du schéma de base de données.
La librairie Liquibase nous a permis de construire notre schéma de base de données de manière incrémentale au fil des ajouts de nouvelles entités ou de nouvelles colonnes. Les mises à jour du schéma étaient appliqués automatiquement à chaque déploiement de l'application, assurant l'ensemble de l'équipe d'avoir automatiquement un schéma à jour. Cette approche est également plus compatibles avec les méthodes agiles, en créeant uniquement les tables et colonnes nécessaires pour la fonctionnalité en cours d'écriture, avec la possibilité d'effectuer des modifications quand le besoin s'en fera sentir.
Les tables de jointure n'étant pas représentées par des entités, nous avons donc cinq entités :
- Person, un utilisateur de MF Banking (qu'il soit simple utilisateur et/ou administrateur)

- Role, les rôles que peut avoir un utilisateur de l'application (utilisateur et administrateur)
- Compte, un compte bancaire d'un utilisateur (avec possibilités d'avoir plusieurs comptes par utilisateur et des comptes joints)
- Operation, une opération (crédit ou débit) sur un compte bancaire
- OperationType, le type d'une opération (virement, chèque, espèce ou carte bleue)

module DAO

Le module DAO correspond à la couche d'accès aux données.
DAO est l'acronyme de *Data Access Object* : un DAO est une classe Java dont le but est d'accéder et de modifier les données stockées par l'application. Ce terme est parfaitement générique : que les données soient stockées dans un fichier ou dans une base de données, la classe se chargeant de récupérer ces données reste un DAO.
Sont centralisés dans ce module l'ensemble de la configuration Spring ayant trait à l'accès aux bases de données (connexion, infrastructure Hibernate/JPA, gestion des transactions...) ainsi que le code en charge de la récupération des données depuis la base de données, en se basant sur les entités définies dans le module Entities.

En production, l'application utilisait une base de données PostgresSQL mais une base de données H2 a été également utilisée pour effectuer des tests unitaires.Cette base, comme les bases Derby et HSQLDB, présente l'avantage de pouvoir fonctionner uniquement en mémoire, ce qui est très adapté dans le cadre de tests unitaires : la base de données est crée, initialisée grâce à Liquibase et des données sont ajoutées le temps d'effectuer les tests unitaires et la base est détruite une fois le test terminé. On a donc des tests unitaires faciles à reproduire, puisqu'ils sont à chaque fois exécutés sur une base de données vierge.

Afin de réaliser les tests unitaires, nous nous sommes aidées d'une librairie développée par Stéphane appelée Spring DB-Unit simplifiant grandement l'insertion de données de tests dans la base en mémoire.

Idéalement, ce module aurait dû être découpé en deux modules, DAO et DAO API, afin de découpler complètement les interfaces et des DAO et leurs implémentations. En l'état, il est théoriquement possible de faire directement référence aux implémentations, bien que cela n'ait pas été fait.

Services et Services API

Le module Services est le cœur de l'application, où se situe l'ensemble de la logique métier (comment effectuer les virements, quelles vérifications effectuer, comment gérer

l'authentification des utilisateurs...).
Comme expliqué précédemment, cette couche est le pivot de l'application et c'est donc typiquement la seule couche qui n'est pas remplacée, surtout si le modèle de l'architecture N-tiers est correctement implémentée, ce qui est le cas pour MF Banking : le module Service ne dépend aucunement de la façon dont l'accès aux données est implémenté, ni de la façon dont les informations sont présentées à l'utilisateur.

Afin de respecter au maximum les bonnes pratiques de codage lié à l'architecture N-tiers, nous avons dû tester les services en isolation, sans qu'ils dépendent de s implémentations de la couche DAO. Pour ce faire, nous avons utilisé Mockito, une librairie permettant de créer de « faux » objets, dont nous déterminions le comportement, par exemple : *si la méthode X d'un DAO est appelée, alors le résultat Y doit être retourné.* Les tests unitaires des services ne testaient donc QUE les services et nous étions donc assurés qu'en cas de problème, le problème ne pouvait venir en aucun cas de la couche inférieure.

Batch

Comme pour une véritable banque, les opérations de MF Banking dispose d'une date d'effet (date à laquelle l'opération a été enregistrée) et d'une date de valeur (date à laquelle l'opération est prise en compte et le solde modifié en conséquence).

Cette modélisation impose cependant de mettre à jour automatiquement le solde du compte lorsque la date de valeur d'une ou plusieurs opérations correspond au jour courant.
Nous avons mis en place cette mise à jour automatique par le biais de Spring Batch : une tâche de mise à jour du solde est exécutée à intervalles réguliers (dans une véritable application d'e-banking, l'exécution devrait avoir lieu toutes les 24 heures, nous avons utilisé des intervalles beaucoup plus courts afin de pouvoir tester plus rapidement la modification).
Un flag indique si une opération a été prise en compte ou non et, le cas échéant, le solde est mis à jour et l'opération est marquée comme traitée.

Web

Le module Web correspond à la couche de présentation de l'architecture N-tiers.
Nous avons utilisé le framework Spring MVC pour bâtir notre couche web, car il offrait de nombreux avantages : Spring MVC est parfaitement intégré à Spring et son modèle de programmation est simple et permet de développer rapidement des interactions complexes avec l'utilisateur. Comme son nom le laisse suggérer, Spring MVC tend à se rapprocher d'une architecture Modèle-Vue-Contrôleur.

Le module Web est très nettement le module qui a le plus bénéficié de librairies tierces-parties, toujours dans le but de simplifier autant que possible le développement en déléguant les tâches complexes ou répétitives à une librairie dont c'est la spécialité. Nous avons utilisé entre autres :

- Bootstrap, développé par Twitter, offre un ensemble cohérent de feuilles de style CSS et de code JS permettant de développer simplement une interface web agréable à utiliser. Etant donné que le but de cette formation n'était pas de tester nos compétences HTML/CSS/JS, Bootstrap nous a permis d'avoir une application web élégante sans consacrer trop de temps à l'écriture de feuilles de style CSS.
- Tiles, librairie de *templating* développée par la fondation Apache. Notre interface web incluait plusieurs éléments se retrouvant sur l'ensemble des pages : les feuilles de style et les scripts JS de Boostrap, une barre de navigation...Afin d'éviter de réécrire à chaque fois ces mêmes lignes de code, avec le risque de les oublier à l'occasion, Tiles nous a permis de définir un « modèle » de page incluant systématiquement ces éléments et nous nous chargions alors d'écrire uniquement les éléments de la page qui différaient.
- Hibernate Validator est l'implémentation de référence de la Java Validation API, spécifiée par la JSR-303.Les fonctionnalités offertes par la JSR-303 nous ont permis de simplifier grandement la validation des formulaires de MF Banking. Là où une validation « classique » des données consisterait à vérifier manuellement tous les champs du formulaire à valider, la JSR-303 permet par le biais d'annotations de définir les contraintes que doivent respecter un champ d'un formulaire et le support de la JSR-303 par Spring permet de déclencher automatiquement cette validation. Et toujours grâce aux fonctionnalités offertes par Spring, il a été également très simple d'afficher des messages d'erreur internationalisés à chaque fois qu'une erreur de validation du formulaire était survenue.
- POI, API gérant l'écriture de documents Word,Excel et Powerpoint, développé par la fondation Apache. Une des fonctionnalités à réaliser était l'export du rélevé des opérations au format Excel. Grâce aux fonctionnalités offertes par POI, la manipulation de documents Excel s'est avérée très aisée et nous a évité d'avoir à manipuler directement le format Excel, ce qui aurait pu s'avérer très complexe.

Au même titre que les autres couches de notre application, la couche Web n'a pas échappé à des tests automatisés.La nature même des pages web rendant compliqué l'écriture de tests unitaires, nous n'avons pas effectué de tests unitaires vérifiant le comportement de chaque pages de façon isolée.
L'approche que nous avons adopté est celle des tests d'intégration : nous avons vérifié le comportement global de l'application en nous assurant qu'un utilisateur pourrait naviguer entre les pages et effectuer toutes les actions qu'il souhaite comme prévu.

Pour ce faire, nous avons utilisé l'outil Selenium. Pour citer les auteurs de Selenium, cet outil « automatise les navigateurs ». On crée un scénario de test en naviguant sur les différentes pages du site et cette succession de pages est ensuite enregistrée par Selenium, qui convertit ce scénario en code Java.
Le principe de Selenium est ensuite assez simple : Selenium démarre un navigateur Web et exécute le scénario de test en simulant les clics et les saisies clavier. Si Selenium réussit à reproduire le scénario de test, le test est réussi. Cela prouve donc qu'aucune fonctionnalité testée n'a été impactée par une modification du code.

Afin d'effectuer les tests Selenium dans des conditions optimales, les tests étaient effectuées sur une base de données PostgresSQL à part, remplie avant chaque le lancement des tests et vidée lorsqu'ils terminent. Les résultats étaient donc parfaitement reproductibles, sans risque qu'une mise à jour des données effectuée par les tests risque de faire échouer un test après de multiples exécutions. On peut notamment citer le cas des virements, diminuant le solde des comptes à chaque fois, jusqu'au point où le solde pourrait ne plus être suffisant et le virement par conséquent refusé.

Afin de ne pas permettre à un utilisateur anonyme d'accéder aux comptes de n'importe quel client de MF Banking, notre application a été évidemment sécurisée.Une fois de plus, nous avons utilisé une librairie développée par Spring : Spring Security.
Spring Security permet de s'abstraire d'une large part de la complexité de la sécurisation d'une application, en ne demandant au développeur qu'à préciser ce qui doit être sécurisé,comment cela doit être protégé et qui peut y accéder.

WebServices

Comme indiqué précedemment, on peut voir les Web Services comme une couche de présentation alternative.
Bien qu'ils ne constituent pas un moyen de présenter les données visuellement à l'utilisateur, ils sont néanmoins pour l'application un moyen de communiquer avec l'extérieur, au même titre que des pages web. Au même titre que le module Web, le module Web Services vient réutiliser le modules Services.

Nous avons mis en place deux types de Web Services pour MF Banking : des Web Services SOAP et des Web Services REST.

Les Web Services SOAP se basent sur l'échange de fichiers XML au format SOAP contenant les informations que les applications doivent se transmettre,en l'occurence les arguments à passer aux méthodes des Web Services et les valeurs qui sont retournées par ces mêmes méthodes.
Les différents services exposés, les types de données acceptés en entrée et retournés sont

exposés via un fichier descripteur, au format WSDL (Web Service Definition Language). L'avantage et à la fois l'inconvénient du SOAP est qu'il est basé sur le XML : le WSDL est avant tout un schéma XML et les données fournies au Web Service peuvent être validées grâce au WSDL, évitant ainsi d'envoyer des données incohérentes. Par contre, XML est un format de fichier très verbeux, où la quantité de données « utiles » est finalement assez faible à cause des balises XML.
L'API Java centré autour de la création de Web Services SOAP se nomme JAX-WS (Java API for XML Web Services) et est standardisé via la spécification JSR-224. Apache CXF,le framework que nous avons utilisé pour gérer les Web Services, propose deux approches pour générer de nouveaux Web Services ou le code y accédant :

- *code first* : on écrit d'abord le code du Web Service, et les annotations de JAX-WS fournissent les métadonnées nécessaires à la génération du fichier WSDL correspondant à ce Web Service
- *contract first* : on écrit d'abord le WSDL et des outils de génération de code se chargent d'écrire le code du Web Service correspondant. On peut également se servir d'un WSDL existant pour générer le code accédant au Web Service correspondant, épargnant ainsi la manipulation d'un fichier XML au format complexe.

Nous avons utilisé ces deux approches pour MF Banking :

- les Web Services en eux-mêmes étaient *code first*, car nous disposions déjà du code des services (ce qui réduisaient considérablement la quantité de code à écrire) et que l'écriture d'un fichier WSDL s'avérait être une tâche trop complexe à réaliser dans le temps qui nous était imparti.
- les WSDL générés ont été réutilisés pour générer le code des clients des Web Services

Les Web Services REST suivent une approche très différente de celle des Web Services SOAP. L'idée de départ des Web Services REST (REpresentational State Transfer) est de se baser sur les différents « verbes » du protocole HTTP pour définir le type d'action réalisée par une méthode d'un Web Service : une requête GET récupère des données, une requête POST ajoute ou met à jour un élement, une requête DELETE supprime un élément....
Contrairement à SOAP, REST ne met pas de contrainte sur le format des données échangées (qui peuvent donc être au format XML si on le souhaite), bien que le format le plus souvent utilisé soit le JSON (Javascript Simple Object Notation), car c'est un format simple et très compact (et qui peut être lu sans utiliser de librairie supplémentaire par du code Javascript).
L'exploitation des spécificités du protocole HTTP font que l'accès à un Web Service REST se limite à envoyer une requête HTTP à une URL précise et l'ensemble des services exposés se limite alors à une simple collection d'URLs.

Nous avons également utilisé CXF pour créer les Web Services REST de notre application en *code first*. L'approche *contract first* n'a pas été exploitée pour deux raisons :

- Bien qu'il existe un schéma XML pour les Web Services REST au format WADL (Web Application Description Language), il n'est que rarement utilisé et s'avère de toute façon sans intêret lorsque JSON est utilisé comme format d'échange
- Etant donné qu'interroger un Web Service REST se limite à effectuer une requête HTTP sur une URL, une librairie simplifiant la construction et l'émission de requêtes HTTP suffit amplement pour interroger un Web Service REST efficacement.

Application Android

L'application Android n'était pas une demande explicite du Product Owner, mais celle nous a permis de mettre en oeuvre les Web Services REST que nous avons implémenté.

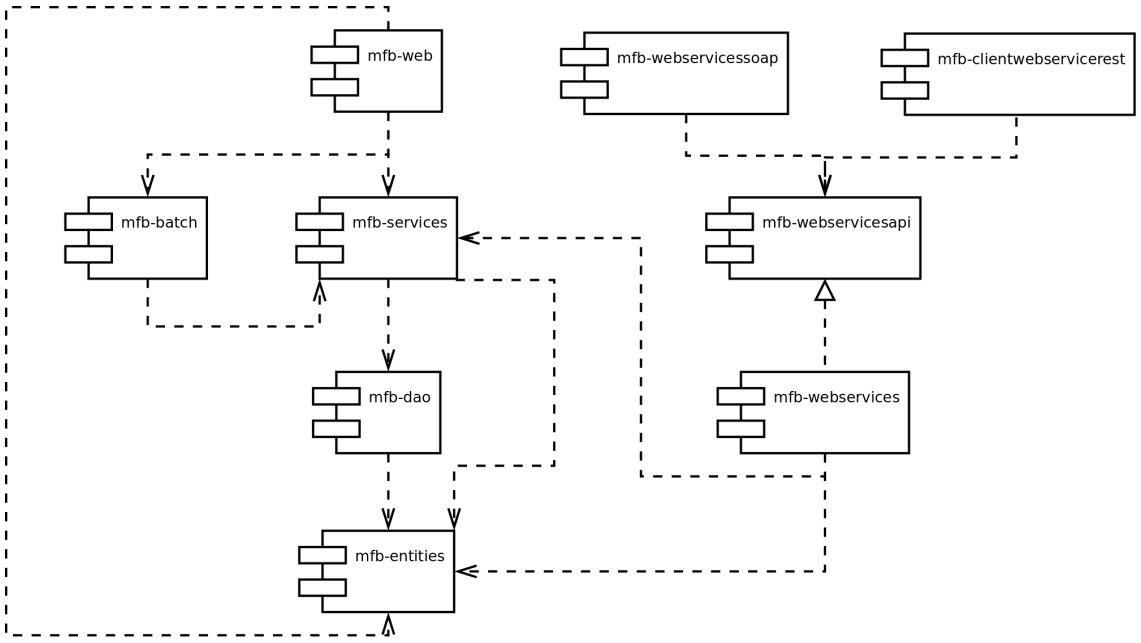

FIGURE 5 – Interaction entre les différents modules de MF Banking

3.3.8 Apport personnel

Comme tous les autres membres de mon équipe, j'ai participé au développement de chacune des fonctionnalités de MF Banking.
Ma contribution a pris différentes formes :

- Séléction des *items* à implémenter au sprint suivant
- Modélisation de la base de données quand cela était nécessaire
- Recherche de librairies pouvant nous aider dans le développement d'une fonctionnalité et quand la recherche s'est avérée fructueuse, lecture de documentation sur la librarie choisie afin de comprendre comment s'en servir
- Développement des fonctionnalités en binôme, que ça soit en tant que codeur ou en tant qu'observateur.
- Validation d'une tâche avant que celle-ci soit considerée comme terminée.

J'ai pris part au développement de chacun des modules et ai donc eu l'occasion de travailler sur l'ensemble des technologies mises en oeuvre dans MF Banking.

Si je devais mettre en avant un rôle particulier dans que j'ai joué durant le développement de MF Banking, cela serait la recherche de librairies et leur intégration dans notre projet.
Stéphane, durant un de ses *speechs*, a insisté sur l'intérêt d'utiliser des librairies afin d'éviter de réimplémenter des fonctionnalités déjà existantes et bien mieux réalisées que nous aurions pu le faire, ceci afin de se concentrer uniquement sur le code spécifique à notre application.
J'ai donc mis ce principe en oeuvre au maximum et, avant de commencer à développer une nouvelle fonctionnalité, mon réflexe fut de chercher systèmatiquement si une librairie ou plusieurs pourraient nous être utiles, les étudier afin de séléctionner la plus adaptée puis de comprendre son fonctionnement pour la mettre en oeuvre dans MF Banking.
J'ai entre autres incité mes collègues à ce que Liquibase, Spring-DBUnit ou Hibernate Validator soient utilisées.

3.3.9 MF Banking en chiffres

En résumé, MF Banking, c'est :
- 6 développeurs
- 5 semaines de développement
- 444 commits
- 8000 lignes de code

3.3.10 Tests de charge sur MF Banking avec Gatling

Durant mon travail sur Gatling, et afin de le prendre en main, j'ai été amené à écrire un test de charge de MF Banking.
Ce test de charge, impliquant l'utilisation simultanée de l'application par 1000 utilisateurs, a montré que notre application résistait bien au test et que notre implémentation était donc suffisamment performante.

Ce scénario de test a effectué 110000 requêtes sur MF Banking, donc aucune n'a échoué. Le temps moyen de réponse est de 191 ms, avec un temps de réponse maximal de 9,6 secondes.
Les résultats détaillés du tests sont présents en annexe.

3.3.11 Accès à MF Banking

MF Banking a été mis en ligne et est disponible à l'adresse suivante : http ://mf-banking.j.layershift.co.uk
Trois comptes utilisateurs sont disponibles :

- Utilisateur normal : nom d'utilisateur : *user*, mot de passe : *user*
- Administrateur : nom d'utilisateur : *admin*, mot de passe : *admin*
- Utilisateur et administrateur : *useradmin*, mot de passe : *useradmin*

4 Gatling

4.1 Présentation de Gatling

4.1.1 Qu'est-ce que Gatling ?

Gatling est un injecteur de charge open source, développé par Excilys et disponible depuis Janvier 2012.
Le but d'un injecteur de charge est d'effectuer des tests de charge sur une application web, en simulant l'activité des utilisateurs et leur navigation à travers les différentes pages.

Les développeurs d'une telle application peuvent ainsi tester et vérifier la qualité de leur design et de leur implémentation en analysant la résistance de celle-ci à une utilisation simultanée par un grand nombre d'utilisateurs et met éventuellement en évidences les points chauds de leur application où une baisse de performances est observable, les aidant ainsi à le corriger, avant la mise en production.
Les rapports produits par Gatling constituent également une preuve concrète de la qualité du travail des développeurs pour leur hiérarchie ou leurs clients.

4.1.2 Pourquoi avoir créé Gatling ?

Gatling n'est pas le seul ni le premier injecteur de charge disponible sur le marché. Il existe de nombreux concurrents :

- JMeter
- LoadUI
- LoadRunner
- The Grinder
- etc...

Cependant tous ces outils présentent différents inconvénients :

- Certains sont payants, et la licence peut être très coûteuse (10000$ dans le cas de LoadUI
- Certains sont codés dans d'autres langages que Java, compliquant leur compréhension ou leur debuggage par des développeurs Java
- Certains ne fonctionnent que sur Windows
- Enfin, leur performance ne permettent pas toujours de simuler des tests de charge intensifs (plusieurs milliers d'utilisateurs, parfois sur de longues durées)
- L'écriture de leur scénario de tests peut être complexe ou les scénarios peuvent être difficiles à maintenir

Gatling a donc été mis au point afin d'offrir une alternative, gratuite, à ces différentes solutions.

4.1.3 Qu'est ce qui différencie Gatling de ces autres solutions ?

Le modèle d'*acteurs* sur lequel repose Gatling ainsi que son utilisation des communications asynchrones font que Gatling est peu gourmand en ressources (CPU ou mémoire). JMeter, par exemple, exigera une machine très performante voire d'un ensemble de machines pour réaliser des tests que Gatling peut effectuer sur une machine aux performances modérées.

Une autre différence importante entre Gatling et ses concurrents est le format des scénarios de tests (ou *simulations*).
Alors que de nombreux outils utilisent un fichier au format XML afin de décrire leur simulations, ce qui les rend plus difficilement manipulables sans passer par l'outil et également plus difficiles à maintenir, les simulations de Gatling sont écrits sous forme de code Scala, simples à modifier, à mettre à jour et à intégrer au reste du projet (dans le cadre d'un système de contrôle de version par exemple).

4.2 Technologies utilisées

4.2.1 Scala

Scala est un langage de programmation multi-paradigme (objet et fonctionnel).

La principale différence entre Scala et d'autres langages fonctionnels tels que OCaml, Haskell ou Lisp est qu'il est conçu pour fonctionner sur une JVM (machine virtuelle Java).
Ceci permet à n'importe quel code Scala d'utiliser du code Java existant sans manipulation particulière, offrant ainsi aux développeurs la possibilité d'utiliser toutes les librairies et frameworks écrits en Java. Il est également tout à fait possible d'intégrer d'utiliser du code Scala dans du code Scala.

Bien que moins répandu que Java, Scala est un langage qui prend petit à petit de l'ampleur et est déjà au coeur de certaines applications parmi les exigeantes (le moteur de gestion des tweets de Twitter est codé en Scala).

4.2.2 Akka

Akka est un toolkit permet de réaliser simplement des applications concurrentes et distribuées.

Ceci est permis par le modèle de communication utilisé par Akka basé sur les **acteurs**.

Dans le cas de Gatling, plutôt que de représenter chaque utilisateur par un thread, il est représenté par un acteur.

Chaque acteur dispose d'une **mailbox**, stockant les messages envoyés par les autres acteurs du système. Lorsqu'un acteur a reçu des message et que ceux-ci doivent être traités, un thread se charge alors de « prendre en charge » cet acteur et ses messages le temps de les traiter et redevient par la suite disponible pour un autre acteur. Il s'agit donc d'un modèle asynchrone.

Par exemple, là où un modèle 1 utilisateur = 1 thread aurait donc nécessité la création de 1000 threads pour 1000 utilisateurs, une application s'appuyant sur le modèle d'acteur d'Akka peut très bien se contenter de créer quelques dizaines de threads pour gérer ces mêmes 1000 utilisateurs.

Akka dispose également de fonctionnalités avancés de tolérance aux pannes, de supervision des acteurs par d'autres acteurs (via une hiérarchisation de ceux-ci), d'acteurs distribués...

Akka est par exemple au cœur du *Play ! Framework*, framework destinée aux applications web et écrit en Scala.

Une des raisons ayant poussé le choix de Scala pour Gatling est qu'Akka est lui-même développé dans ce langage et que celui-ci est le plus adapté pour développer des applications reposant sur Akka.

4.2.3 Netty & Async HTTP Handler

Async HTTP Handler est un client HTTP reposant sur le moteur Netty qui permet de gérer l'envoi de requêtes (et le traitement des réponses) HTTP de manière asynchrone, complétant ainsi le caractère asynchrone d'Akka, ce qui permet à Gatling de disposer d'un moteur intégralement asynchrone.

4.2.4 Metrics

Metrics est une librairie développée par Codahale et créée à l'origine pour le réseau social Yammer.

Metrics offre un panel de métriques (compteurs, histogrammes, taux par seconde...) qu'il est possible d'ajouter au code d'une application afin de mesurer certains évènements.
Metrics offre également la possibilité d'exporter ces métriques vers des outils de reporting tels que Graphite ou Ganglia, offrant ainsi une solution complète de monitoring de

l'application cible.

4.3 Architecture de Gatling

4.4 Encadrement du projet Gatling et méthodes de travail

Ma participation au projet Gatling a été encadrée par Stéphane LANDELLE, *Product Owner* de Gatling et qui le développe actuellement à plein temps.
En tant que *Product Owner*, Stéphane décide des fonctionnalités que nous devions ajouter à Gatling. Quand nous avons une piste pour implémenter une des fonctionnalités souhaitées, nous vérifions avec lui que cette piste lui convient et le cas échéant, nous commencons à l'implémenter.
Une fois l'implémentation achevée et testée, nous vérifions la qualité du code avant d'intégrer définitivement la fonctionnalité à Gatling.

4.5 Fonctionnalitées réalisées ou en cours de réalisation

4.5.1 Monitoring côté serveur

Actuellement, les mesures prises par Gatling ne concernent que ce qu'un utilisateur de l'application observerait : le temps que met une page web à répondre ou tout simplement si celle répond ou non.
La première fonctionnalité que Stéphane souhaitait intégrer à Gatling est le monitoring de l'application testée, côté serveur.

Ce choix était motivé par deux raisons :
- Bien qu'étant open source (et donc sans volonté de réaliser de profits par le biais de Gatling), Gatling reste en concurrence avec d'autres produits gratuits ou payants et plusieurs produits payants proposent du monitoring côté serveur ou comptent en intégrer dans un futur proche. Intégrer du monitoring côté serveur dans Gatling permettrait donc à celui-ci de ne pas prendre de retard sur la concurrence
- Monitorer l'application testée permettrait d'obtenir de précieuses informations sur le comportement de l'application durant le test que ça soit en terme d'utilisation CPU, de mémoire, d'activité des bases de données...En recoupant ces informations avec celles déjà existantes, cela permettrait d'obtenir des rapports encore plus pertinents : On pourrait observer par exemple que, lors d'une baisse de performances de l'application, l'activité CPU explose ou la mémoire utilisée par l'application augmente fortement, ce qui pourrait mettre les développeurs de l'application sur la piste d'un problème dans l'implémentation.

J'ai donc, avec mes collègues, exploré les solutions possibles pour apporter cette fonctionnalité à Gatling.

Cela nous a très rapidement conduit à étudier JMX (Java Management Extensions). JMX est une technologie disponible dans n'importe quel JDK standard qui offre la possibilité de créer des classes (les *MBeans* gérant un ensemble de statistiques et de pouvoir exposer celle sur le réseau par le biais de protocoles de type RMI (Remote Method Invocation). Plusieurs MBeans sont d'ailleurs disponibles dans l'API standard Java, afin de monitorer le fonctionnement de la JVM (mémoire utilisée, statistiques sur le garbage collector...).
JMX étant la solution la plus utilisée pour exposer des statistiques d'une application au monde éxterieur, il nous a semblé évident que le monitoring côté serveur passerait par l'exploitation de JMX.

Deux problèmes se posaient cependant avec JMX :
- Comment gérer efficacement la remontée de plusieurs dizaines, voire centaines de

statistiques via JMX, le tout restant configurable par un utilisateur de Gatling ?
- JMX présente de potentiels problèmes de performance, de par son utilisation du protocole RMI. Si des nombreuses statistiques sont remontées et qu'il y a donc une forte activité du côté de RMI, ne risque-t-on pas d'impacter, éventuellement de manière importante, les performances de l'application dont on cherche justement à évaluer les performances ?

Cela nous a donc conduit à rechercher des outils existants permettant de régler ces problèmes.
Le premier problème est en mesure d'être résolu grâce à l'outil Jmxtrans, spécialement conçu pour simplifier le requêtage répété de nombreux MBeans.
Le deuxième peut être résolu en utilisant un agen Java. Un agent est un éxécutable Java venant se greffer sur une JVM afin d'interagir avec le code s'éxécutant sur celle-ci. Le problème de performance était ainsi reglé car, au lieu de faire appel aux MBeans à requêter à travers RMI, la requête est directement effectuée sur les MBeans en local, au sein même de la JVM.

Nous avons alors créé plusieurs programmes afin de tester chacune des solutions, avec des résultats encourageants.
Malgré tout, d'autres choix techniques importants se posaient qui nécessiteraient un arbitrage de Stéphane et, étant donné que d'autres fonctionnalités devaient être rapidement implémentées, nous avons pour l'instant laissé de côté le monitoring côté serveur afin de se concentrer sur ces autres fonctionnalités.

4.5.2 Support de DataWriters optionnels

Les DataWriters sont les classes de Gatling qui recoivent les résultats des requêtes effectuées et qui sont en charge de les traiter.
Deux DataWriters sont présents en standard dans Gatling :
- ConsoleDataWriter, en charge d'afficher régulièrement sur la console l'état d'avancement du test de charge : durée écoulée, nombres d'utilisateurs actifs, nombre de requêtes effectuées pour chacune des requêtes du scénario...
- FileDataWriter, en charge d'écrire les résultats des requêtes dans un fichier de log afin qu'il puisse analysés par la suite pour produire le rapport de test.

Le besoin se faisait sentir de pouvoir supporter des DataWriters optionnels, notamment pour l'intégration de Metrics.
Etant donné que chaque DataWriter est un acteur Akka, j'ai retravaillé le code en charge du dispatch des résultats des requêtes aux différents DataWriters.
Là le code existant envoyait explicitement aux deux DataWriters le résultat des requêtes,

j'ai mis en place un routeur[3] Akka, en charge de renvoyer un message à l'ensemble de ses acteurs routés à qui je fournis la liste de tous les DataWriters actifs (gérés par un fichier de configuration).

4.5.3 Intégration Jenkins

L'ajout d'une intégration de Gatling au serveur d'intégration continue Jenkins est motivée par l'interêt que peut avoir des tests de charge réguliers sur une application en cours de développement.
En effet, en exécutant à chaque build un test de charge (mis à jour pour tester les nouvelles fonctionnalités ajoutées par le code mis à jour) permet de surveiller constamment les performances de l'application et d'observer l'impact des nouveaux ajouts sur les fonctionnalités pré-existantes.

Deux voies ont été envisagées pour réaliser cette intégration, en se basant sur le système de *plugins* de Jenkins :

- Utiliser le HTML Publisher Plugin, afin de publier automatiquement les résultats des simulations de Gatling(qui prennent la forme d'une page HTML)
- Utiliser le Performance Plugin, et lui fournir les données en entrée pour qu'il puisse générer ses graphiques

La solution se basant sur le HTML Publisher Plugin n'a pas été retenue car l'intégration qui en résultait n'était pas assez poussée (on disposait tout au mieux d'un lien vers la pages des résultats).

La solution se basant sur le Performance Plugin a donc été privilégiée, car les fonctionnalités offertes par ce plugin sont bien supérieures,entre autres :

- Intégration plus poussée de la page de résultats des simulations au sein de l'interface de Jenkins
- Graphiques montrant l'évolution des résultats des simulations de build en build

Le problème auquel j'ai été confronté est que le Performance Plugin ne supporte que deux formats en entrée : le format de logs de Junit et celui de JMeter, et le format des logs de Gatling n'était compatible avec aucun d'entre eux. En effet, les formats de logs de Junit et de JMeter sont tous deux basés sur XML, alors que Gatling écrit ses résultats au format TSV (*tab−separated values*).

Une première solution à ce problème a été proposée par un développeur de Gatling : modifier le Performance Plugin (qui est open-source) afin qu'il supporte le format des logs de Gatling.

3. http ://doc.akka.io/docs/akka/2.0.3/scala/routing.html

Cependant, cette solution n'était pas pérenne, car elle reposait sur une version « alternative » du Performance Plugin, et tous les utilisateurs de Gatling souhaitant bénéficier de cette intégration auraient dû installer ce nouveau plugin depuis les sources, alors que le Performance Plugin de base peut être installé en quelques clics depuis Jenkins. L'ajout pourrait également être proposé aux développeurs en charge du Performance Plugin, mais rien ne garantit qu'ils auraient accepté d'intégrer cette ajout, surtout que Gatling reste un outil relativement jeune.

La solution a été de prendre le problème dans l'autre sens : plutôt que de forcer le Performance Plugin à pouvoir lire le format de logs de Gatling, c'est à Gatling d'écrire ses logs dans un format supporté par le Performance Plugin.
Comme évoqué précedemment, deux formats sont actuellement supportés :

- Le format de logs de Junit
- Le format de logs de JMeter

J'ai étudié en détail ces deux formats, afin de définir celui qui sera le plus adapté mais également le plus simple à écrire.

Le format de logs de Junit a été rapidement écarté, car il ne correspond absolument pas aux besoins de Gatling : de nombreux informations non pertinentes dans le cadre d'un test de charge doivent être indiquées et peu d'informations peuvent être précisées concernant les tests à proprement parler.
Plus simplement, le format de logs de Junit est uniquement pensé...pour des logs de tests unitaires Junit.

Mon choix s'est donc porté sur le format de logs de JMeter. Pour les besoins de Gatling, le format est extrêmement simple :

- La déclaration XML
- Le résultat de chaque requête est représenté par un élements ayant comme tag *httpSample*, dont les détails sont représentés par une série d'attributs

Gérer l'écriture des logs dans ce nouveau format n'a pas posé de problème particulier,étant donné que toutes les informations nécessaires étaient accessibles et qu'assurer un formatage correct des balises fut suffisant.

Les premiers tests furent très convaincants : le format était respecté, les logs étaient correctement traités par le Performance Plugin générait correctement les graphiques.
Mais des test complémentaires demandés par Stéphane démontrèrent que le Performance Plugin n'est pas une solution viable quand il est utilisé avec Gatling : la combinaison d'Akka et de Netty permettant d'effectuer avec des ressources modestes des tests de charge avec plusieurs milliers d'utilisateurs, les logs des simulation ont tendance à attendre rapidement des tailles de plusieurs dizaines de méga-octets.
Les tests initiaux ne portaient que sur des simulations courtes avec peu d'utilisateurs

(200 utilisateurs pendant 3 minutes environ). Mais un test avec 8000 utilisateurs pendant 15 minutes, générant un log de simulation d'une cinquaintaine de méga-octets, ont montré les limites du Performance Plugin : la génération des graphiques est beaucoup plus lente et l'affichage de résultats détaillés bloquait Jenkins pendant plusieurs dizaines de secondes.

Étant donné que les solutions se basant sur des plugins Jenkins existants se sont toutes avérées être des impasses, la décision a été prise de créer directement un plugin à Gatling.
La solution se basant sur le Performance Plugin étant abandonné, il n'est donc plus nécessaire de se baser sur le format JTL et le nouveau plugin va plutôt se baser sur les données déjà produites par Gatling.

Ce nouveau plugin offrira également plus de fonctionnalités que le Performance Plugin :

- Le Performance Plugin permet de faire échouer une build ou de considérer comme instable en fonction à partir d'un certain pourcentage d'erreurs. Le nouveau plugin proposera plus de critères : temps de réponse moyen, temps de réponse maximal,95/99èmes percentiles sur le temps de réponse...
- Les graphiques fournis par le nouveau plugin seront plus nombreux et plus détaillés

Le plugin est actuellement en cours d'écriture et sera terminé courant septembre, afin qu'il puisse être intégré à la version 1.3 de Gatling, la prochaine version stable.

4.5.4 Ajout de métriques en temps réel à Gatling

Les rapports créés par Gatling détaillant les résultats du test de charge sont très complets, mais ne peuvent être fournis qu'une fois le test terminé.Plusieurs utilisateurs de Gatling ont fait part de leur interêt pour une solution permettant d'avoir des résultats en temps réel, en se basant notamment sur des solutions populaires de monitoring telles que Graphite ou Ganglia.
La demande pour cette fonctionnalité ayant été assez forte, Stéphane a décidé que cette fonctionnalité serait implémentée.

Plusieurs pistes et librairies ont été envisagées mais la solution retenue a été l'intégration de la librairie Metrics [4] écrite par Codahale au sein de Gatling.

4. http ://metrics.codahale.com

Deux problèmes ont cependant compliqué l'intégration « directe » de Metrics.

Tous les métriques proposées par *thread-safe* et donc accessibles en toute sécurité par plusieurs threads simultanément, mais dans le cadre de Gatling et de l'utilisation d'Akka, un seul thread accéderait aux métriques à un instant T.
De plus rendre du code Java *thread-safe* passe par l'utilisation de la synchronisations et par la prise et le rendu de verrous sur un objet, ce qui impacte les performances. les métriques étant mise à jour à chaque requête terminée, on arrive rapidement à plusieurs milliers de mises à jour par seconde et l'impact de la synchronisation sur les performances risquait de se faire sentir.

L'autre problème posé par Metrics est que nous avions besoin que certaines métriques soit calculées uniquement sur un intervalle d'une seconde, ce qui impliquait que les valeurs stockées par les métriques soient réinitialisés à chaque seconde, ce qui n'était pas proposé par les métriques par défaut de Metrics.

J'ai donc dû réimplementer certaines métriques afin que celles-ci ne soient plus thread-safe mais qu'elles soient en contrepartie plus performantes. J'ai également dû implémenter des métriques dont les valeurs sont réinitialisés à chaque seconde.
Ces modifications ont conduit à réimplementer une bonne partie de l'architecture de Metrics car certains choix d'implémentations de Metrics (visibilité *default* des classes, classes internes essentielles privées...) rendent difficile une extension simple des fonctionnalités de Metrics (Metrics est en quelque sorte conçu pour être utilisé « tel quel »).

Le schéma ci-dessous décrit plus en détail la gestion des résultats des requêtes par mon intégration de Metrics à Gatling :

- Quand une requête termine, un objet RequestRecord contenant les résultats de la requête (temps mis pour éxécuter la requête, nom de la requête concernée dans la simulation, status OK/KO...) est produit et transmis aux DataWriters, chargés de traiter ce RequestRecord.
- MetricsDataWriter est le point d'entrée du RequestRecord. Cette classe gère une Map associant une requête (via son nom) à un acteur Akka,RequestMetrics, en charge de conserver les métriques propres à cette requête et de les mettre à jour
- MetricsDataWriter ausculte le RequestRecord afin d'obtenir le nom de la requête et transmet le RequestRecord à l'instance RequestMetrics correspondante
- L'instance de RequestMetrics qui a reçu le RequestRecord le traite et met à jour les métriques qui doivent l'être.

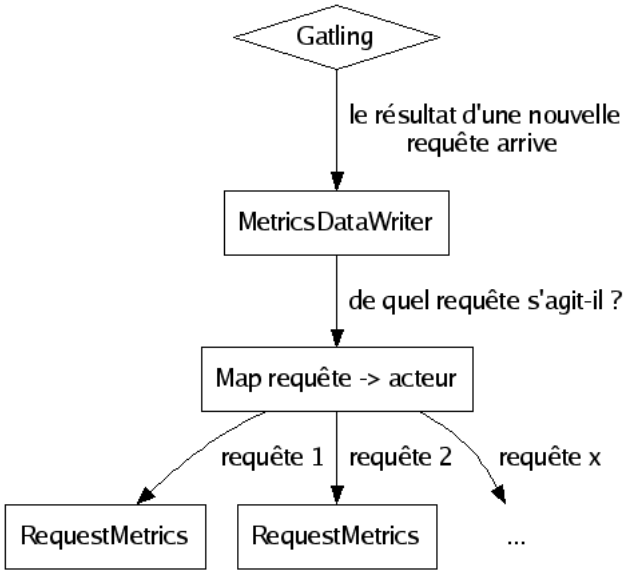

FIGURE 6 – Dispatch d'un résultat de requête

Actuellement, seul Graphite est supporté pour l'export des métriques, mais le support de Ganglia est envisagé si suffisament d'utilisateurs sont interessés.
L'intégration de Metrics ayant été finalisée et acceptée par Stéphane, elle est déjà disponible dans la version de développement et sera présente dans la prochaine version stable.

5 Annexes

5.1 Rapport détaillé du test de charge Gatling sur MF Banking

STATISTICS

Executions	Total	OK	KO
	110000	110000	0

Response Time (ms)	Total	OK	KO
Min	0	0	-
Max	9636	9636	-
Mean	191	191	-
Std Deviation	285	285	-
95th percentile	3603	3603	-
99th percentile	3829	3829	-

FIGURE 7 – Stastiques globales du test : nombre de requêtes réussies et echouées et détails du temps de réponse

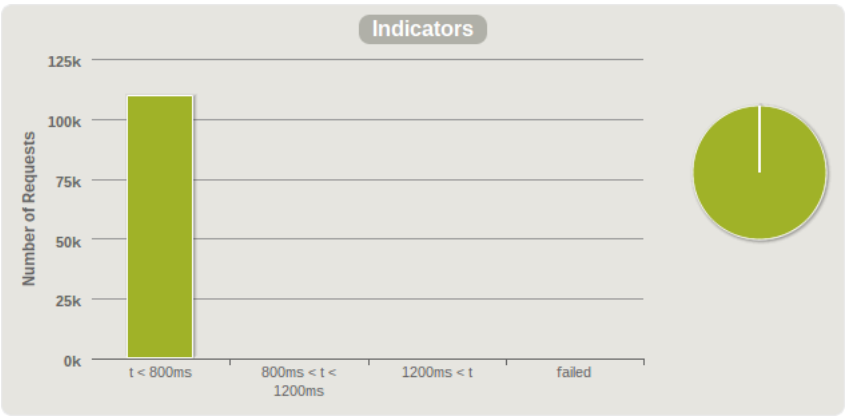

FIGURE 8 – On peut voir que presque toutes les requêtes ont été effectués en moins de 800 ms

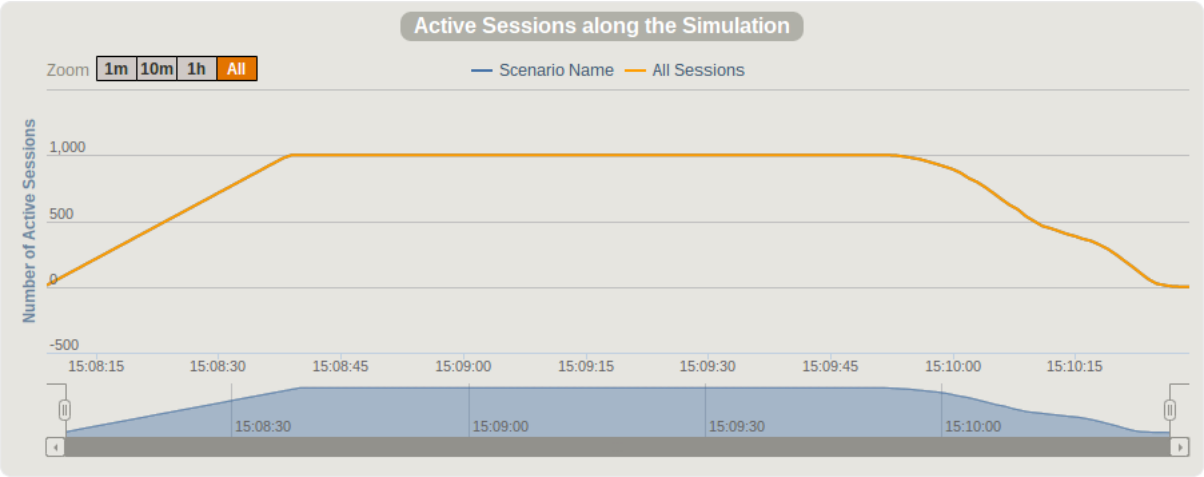

FIGURE 9 – Nombres d'utilisateurs actifs simultanement. La montée progressive du nombres d'utilisateurs au début est dû à une *rampe* : on demande à Gatling de prendre 30 secondes pour lancer progressivement les 1000 utilisateurs du test

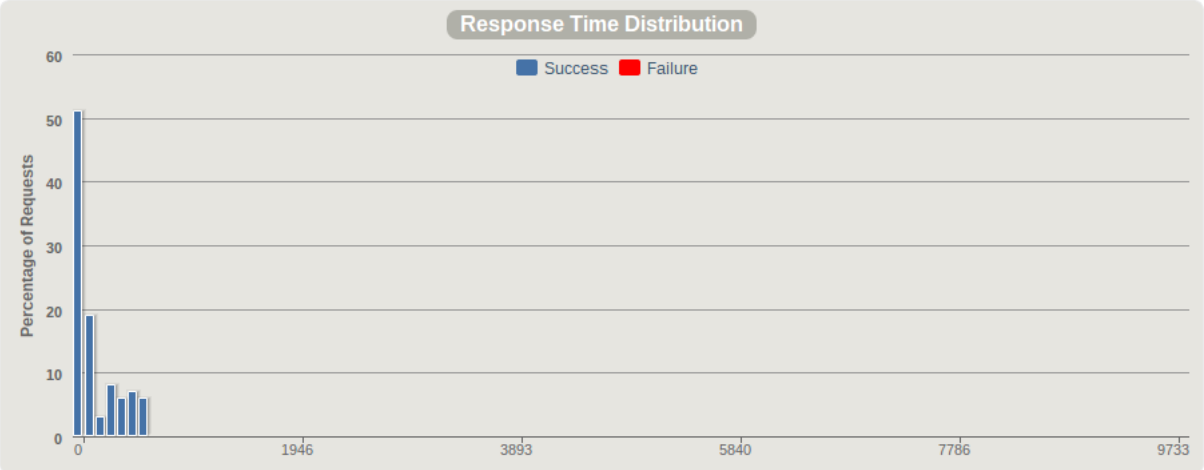

FIGURE 10 – Distribution du temps de réponse : on peut voir que la majorité des requêtes ont été été effectués quasi instantanément et que le nombre de requêtes lentes est tellement faible qu'elles n'apparaissent même pas sur le graphe

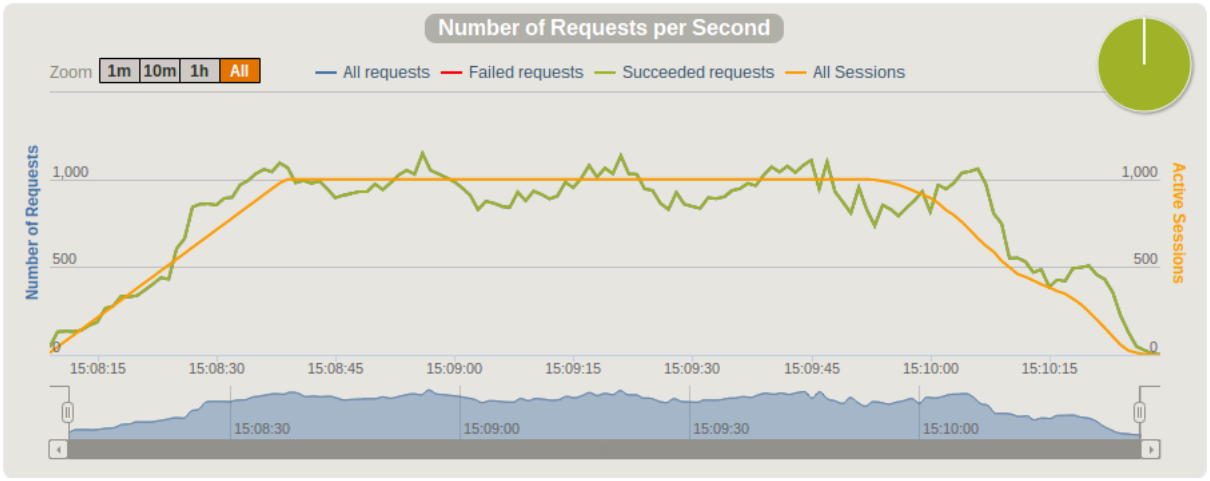

FIGURE 11 – Nombre de requêtes qui démarrent par seconde : on observe qu'une fois la rampe terminée, le système se maintient sans difficulté entre 800 et 1200 requêtes par seconde

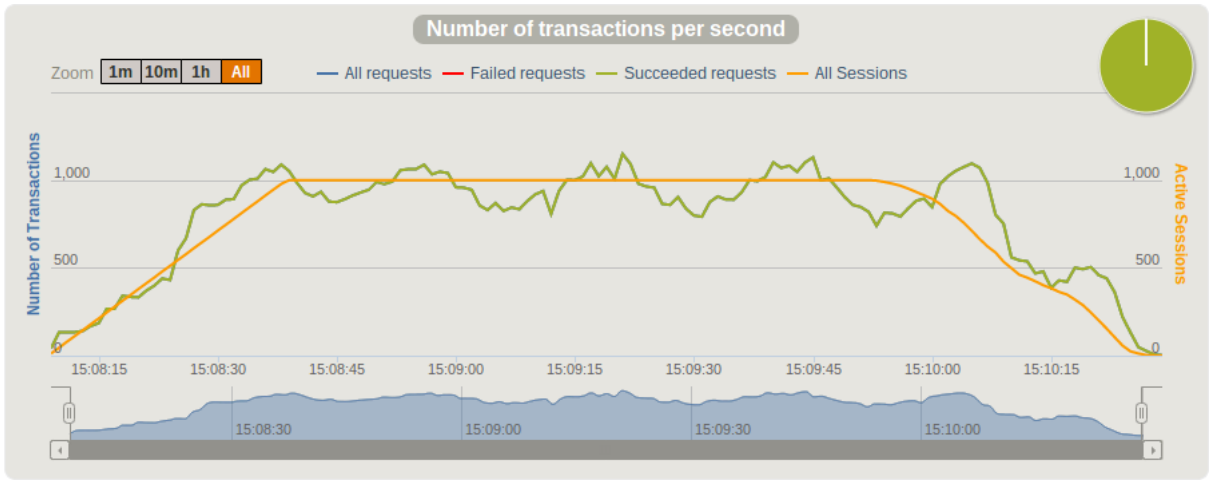

FIGURE 12 – Nombre de transactions (requêtes qui s'achèvent) par seconde : mêmes observations que pour les requêtes, le système se maintient durant tout le test entre 800 et 1200 transactions par seconde