



ENSIIE STRASBOURG

Rapport de stage

Auteur :

Philippe
GAULTIER,

Élève ingénieur en
troisième année à l'ENSIIE
Strasbourg

Maître de stage :

Sven REBER,

Ingénieur logiciel

Lausanne, Suisse, le 17 janvier 2016

The road is long and in the end
the journey is the destination

Unknown

Dans toute la suite du rapport, «GFP» désigne l'entreprise «Global Financial Products».

Table des matières

1	Introduction	3
2	Présentation de EdgeLab	3
2.1	Histoire	3
2.2	Les équipes	3
2.3	Clients	3
2.4	Objectif du stage	4
3	L'application	4
3.1	Architecture	4
3.2	Le projet front-end	5
3.2.1	AngularJS	5
3.2.2	Tests unitaires	6
3.2.3	Tests d'intégration	6
3.2.4	Développement continu - Intégration continue	8
3.2.5	L'environnement	9
3.3	Contraintes	10
3.4	Déroulement du stage	11
3.4.1	Refonte d'interfaces	12
3.4.2	Nouvelles fonctionnalités	12
3.4.3	Amélioration des outils	12
3.5	Développement	13
3.5.1	Bonnes pratiques	13
3.5.2	Architecture	15
3.6	Améliorations	16
3.6.1	Optimisations	16
4	Remerciements	17
5	Conclusion	17
6	Annexes	18
6.1	Outils utilisés	18
6.1.1	Langages utilisés	18
6.1.2	Bibliothèques utilisées	18
6.1.3	Outils divers utilisés	19
6.2	Glossaire	19

Table des figures

1 Introduction

La Suisse fait partie des leaders mondiaux du secteur financier, et dispose d'un statut particulier en Europe : elle fait partie de l'espace Schengen mais pas de l'Union Européenne, ainsi disposant d'un certain exotisme, comme une monnaie, des prises électriques et une législation spécifique, tout en offrant certaines facilités pour les formalités administratives aux travailleurs européens, notamment français.

Lausanne, quatrième ville de Suisse, est particulièrement attrayante de part son emplacement, au bord du lac Léman, et son investissement dans l'éducation supérieure, particulièrement en mathématiques et en informatique, de rang mondial.

Pour toutes ces raisons, et afin de découvrir un milieu du travail différent de la France, j'ai décidé de rejoindre EdgeLab, startup du management du risque financier à Lausanne pour six mois en tant que développeur web.

2 Présentation de EdgeLab

2.1 Histoire

EdgeLab est fondée en 2013 en tant que filiale de l'entreprise de gestion de portefeuilles financiers Global Financial Products (GFP), suite à la demande de leurs clients d'un outil d'automatisation de gestion de portefeuilles et de calcul de risques pour ces derniers.

Ainsi naît EdgeLab, éditeur logiciel pour le management du risque financier qui se positionne dans le domaine du Business to Business (B2B). Les premiers membres sont des docteurs en mathématiques qui conçoivent le moteur de calcul de risque, puis viennent des développeurs web pour l'application web qui permet aux clients d'interagir avec ce moteur.

2.2 Les équipes

Aujourd'hui, EdgeLab est constitué d'une quinzaine de développeurs composant trois équipes à parts approximativement égales :

- Équipe Quantitative : elle travaille sur le moteur de calcul de risque
- Équipe Back-end : elle développe le back-end de l'application web
- Équipe Front-end : elle développe le front-end de l'application web

2.3 Clients

Les clients d'EdgeLab sont des entreprises du secteur bancaire qui gèrent des instruments financiers et qui sont intéressées par des mesures de risque dans leur processus de décision. De plus, le volume de données et les montants associés requièrent l'association de l'automatisation logicielle et de la supervision par des humains de niveaux d'expertise divers.

Cela inclut un large éventail d'entreprises allant des établissements bancaires aux gestionnaires de fortune, en passant par courtiers en bourse.

Actuellement, l'application est utilisée par des clients de différentes organisations.

2.4 Objectif du stage

Ce stage était centré sur le produit proposé par EdgeLab à ses clients, l'application web, et les objectifs étaient multiples, dans plusieurs domaines :

- Développer de nouvelles fonctionnalités
- Améliorer la qualité du code
- Augmenter la couverture de tests
- Simplifier le processus de déploiement
- Améliorer les outils de développement
- Moderniser et simplifier l'interface

3 L'application

L'application web est un projet de grande taille pour les standards du web : environ un millier de fichiers et 15 000 lignes de code source (et après mon passage, 25 000 lignes de tests).

C'est en effet une application client lourd qui suit les standards du web moderne et utilise les outils les plus récents : HTML5, CSS3, et la majorité de la logique en Javascript. Le serveur fournit en fait seulement une API, au même titre que l'API Facebook ou Twitter.

Le rendu de pages se fait côté client, de même qu'une partie des calculs financiers, utilisant les capacités des navigateurs modernes au maximum, tout cela dans le but de rendre l'expérience utilisateur plus agréable, fluide et de minimiser les rechargements complets de page, ainsi que les échanges de données réseau, premier facteur de latence.

3.1 Architecture

L'architecture est assez simple et suit le modèle d'une entreprise comme Google. L'idée est d'utiliser l'outil le plus adapté à la tâche. Ainsi, le serveur web (back-end) est développé en Java 8 et utilise le framework réputé pour le développement web en entreprise, Spring.

Les calculs lourds sont délégués au moteur de calcul financier écrit en C++, focalisé sur la performance. Ces calculs sont soit effectués en temps réels pour les plus courts (durée inférieure à 30 secondes), soit lus dans une base de données qui est remplie la nuit, pour les calculs longs et prévisibles (plusieurs heures).

Mais sur quelles données financières s'appuie ce moteur de calcul, dans un contexte de volatilité des marchés ? Les données proviennent en fait de différents fournisseurs de données reconnus, par exemple Bloomberg, et sont actualisées chaque jour. Cependant, certaines données sont parfois contradictoires avec d'autres provenant d'un fournisseur différent. Rarement, le

cas de données incomplètes, utilisant un format singulier, ou simplement erronées peut se poser.

On comprend alors la nécessité de récupérer, valider, et agréger ces données avant de les utiliser ou des les afficher à l'utilisateur. C'est la responsabilité d'un service à part, écrit en Java, qui va lui aussi chaque nuit effectuer ce long travail portant parfois sur des centaines de milliers de données. Il va rejeter les données suspectes puis va remplir la base de données avec les bonnes données, qui pourront alors être utilisées par le moteur de calcul, ou bien tout simplement par le back-end web qui les transmettra au front-end afin de les afficher à l'utilisateur.

À l'heure du trading haute fréquence, de l'ordre de la milliseconde, l'actualisation des données quotidienne peut ne pas sembler être assez. Au contraire, dans le contexte de la prévision et de l'analyse du risque à moyen ou long terme, et en tant qu'outil d'aide à la décision, cette fréquence est en fait suffisante.

Si un événement perturbe les marchés, il faut simplement de nouveau effectuer le calcul de risque pour les mois ou années à venir, et réévaluer ses décisions stratégiques.

3.2 Le projet front-end

3.2.1 AngularJS

L'application front-end utilise le framework open-source AngularJS, initialement développé par Google. Il permet d'organiser son projet de manière cohérente, en séparant :

- Vues : l'interface avec laquelle l'utilisateur interagit
- Contrôleurs : la logique des vues et la gestion des événements
- Services : les fonctionnalités génériques utilisées dans les contrôleurs
- Directives : les petits composants réutilisables (par exemple un `datetimepicker` sur un formulaire)
- Configuration

L'utilisation d'un framework bien connu est nécessaire, sinon indispensable, dans ce domaine assez peu structuré et évoluant très rapidement qu'est le développement front-end, ainsi que dans le contexte d'utilisation du très flexible et piégeux langage Javascript.

En sus, AngularJS automatise de nombreuses tâches indispensables à une expérience utilisateur riche :

- Animations
- Cryptage
- Cookies
- etc

De plus, un avantage majeur d'AngularJS comparé aux autres frameworks, est le fait qu'il ait été conçu avec les tests en tête. Il fournit de nombreuses aides pour tester son code, allant

même jusqu'à proposer une librairie de tests d'intégration, chose inédite dans l'écosystème Javascript.

Enfin, la communauté a travaillé dur pour procurer tout un assortiment de modules open sources pour répondre aux besoins courants du développement web.

Ma contribution pour ce sujet a consisté à appliquer les conventions préconisées par la communauté (John Papa's style guide). Cette uniformisation du nommage, de l'indentation, et des pratiques a nécessité un refactoring du projet, chose représentant un défi particulier pour le langage dynamique, faiblement typé et manquant de réelles capacités orienté objet qu'est Javascript.

3.2.2 Tests unitaires

Un code non testé doit être considéré invalide jusqu'à preuve du contraire. Partant de ce constat, nous avons considérablement développé les tests unitaires existant, couvrant un nombre réduit de cas, passant ainsi de 250 à un millier de tests, et de quelques milliers de lignes de tests à 25 000.

En plus d'écrire des tests pour le code existant, ce qui a permis de mettre en lumière certains bugs, ou simplement de considérer certains cas pas forcément pris en compte, l'écriture de tests pour le nouveau code a été institutionnalisée.

Cela a eu comme résultat visible des dizaines de fichiers couverts à 100%, pour une moyenne de couverture du projet de 40%, expliquée en partie par une minorité de fichiers ne possédant aucun tests.

Enfin, cette décision a aussi porté ses fruits pour l'épineux sujet des régressions, qui consiste en une perte ou détérioration des fonctionnalités de l'application à la suite d'un changement. Sans tests, une régression passe souvent inaperçue jusqu'à ce qu'un utilisateur s'y heurte.

Avec une bonne couverture de tests, c'est de l'histoire ancienne : le changement provoque l'échec d'un ou plusieurs tests qui étaient valides auparavant. Ainsi la régression est signalée instantanément au développeur, qui peut cerner son ampleur et la résoudre, tout cela avant qu'elle touche l'utilisateur final.

Cerise sur le gâteau, lorsqu'un bug est détecté, il suffit au développeur d'ajouter le test correspondant, qui est invalide avant la résolution, et valide après. On s'assure ainsi que le bug a réellement disparu, et une éventuelle réapparition sera automatiquement détectée.

3.2.3 Tests d'intégration

Les tests d'intégration, ou end-to-end, consistent à tester toute la chaîne logicielle, simulant les action d'un utilisateur, afin de vérifier que chaque composant s'imbrique sans problème avec les autres. Ils complètent les tests unitaires car disposent d'une granularité plus grande et deux composants peuvent être unitairement valides tout en s'intégrant de façon invalide, dans le cas par exemple de données échangées sous un format différent pour les deux parties.

Il est donc apparu comme indispensable de combler l'absence de tests d'intégration, surtout en disposant de la librairie Protractor mise à disposition par Angular dans cet objectif.

Pour le front-end, les tests d'intégration consistent à automatiser les interactions d'un éventuel utilisateur : mouvement de souris, clics, défilement vertical, etc... C'est un travail titanesque à l'échelle d'une application complexe, disposant de multiples pages, avec de très nombreuses possibilités.

Pourtant, partant du principe énoncé par Lao-Tseu, "Un voyage d'un millier de lieues commence par un premier pas", nous avons entamé ce gros chantier, et écrit de nombreux tests d'intégrations.

L'énorme avantage d'un test d'intégration de ce genre est qu'il permet d'écrire des scénarios. Par exemple, pour l'authentification d'un utilisateur, on peut écrire le scénario correspondant, en explorant plusieurs cas se produisant en réalité :

- J'entre une mauvaise combinaison d'email/mot de passe
- Je n'entre rien dans les champs, je clique directement sur "Login"
- J'entre la bonne combinaison d'email/mot de passe, puis je me déconnecte
- ...

Le code suivant illustre le premier cas :

```
it('should not change page if the login fails', function() {  
    nameInput.sendKeys('toto@edgelab.ch');  
    passwordInput.sendKeys('toto');  
  
    loginButton.click();  
  
    expect(isLoggedIn()).toBe(false);  
});
```

Listing 1 – Test de connexion à son compte avec Protractor

Ainsi, ils remplacent les tests manuels, fastidieux, et non-exhaustifs que le développeur est amené à faire à longueur de journée dans son navigateur.

Pour leur exécution, ces tests tournent dans un vrai navigateur, collant ainsi au plus près à la réalité. Encore mieux, au même titre que les tests unitaires, on peut les exécuter dans une multitude de navigateurs et de versions différentes d'un même navigateur, détectant ainsi une éventuelle régression dans une version spécifique d'un navigateur spécifique.

Dans notre cas, nous nous sommes limités à la dernière version de Chrome, Firefox, et Internet Explorer.

3.2.4 Développement continu - Intégration continue

Le développement continu est une bonne pratique de développement qui consiste, dans le cadre d'un projet disposant d'une grande couverture de tests, à automatiser le lancement de tests dans un environnement de test le plus proche possible de l'environnement de production final, lors de tout changement dans le code. Cela s'accompagne d'outils de visualisation de la progression et du résultat des tests, afin que le développeur soit informé le plus vite possible, idéalement de façon instantané, de l'apparition d'un bug dans le code à la suite d'une modification.

Ce modèle de développement se distingue du modèle traditionnel où le développeur développe, et des testeurs, appartenant souvent à une autre équipe, et parfois plusieurs jours plus tard, testent le code et alerte le développeur d'éventuels bugs. En résumé, le cycle de développement est réduit à une granularité beaucoup plus faible : l'entièreté des étapes développement, test, détection de bugs, correction s'effectue dès lors en quelques minutes, voire quelques secondes.

Enfin, puisque la détection de bugs est automatisée par l'exécution de tests dans un environnement ressemblant à celui de production, si tous les tests passent avec succès, on peut considérer que le code est valide. Pourquoi donc ne pas déployer en production le nouveau code ? Et ce automatiquement ? C'est le principe de l'intégration continue, qui prolonge le modèle du développement continu en ajoutant une étape finale.

Une nécessité de ce modèle est cependant d'avoir une grande couverture de tests (proche de 100%). Dans le cas contraire, le fait que tous les tests passent n'est en soit pas un signe que le code est valide. Mais on peut voir cette nécessité comme un avantage : le développeur se voit forcé d'écrire les tests attendant au code qu'il vient de produire, et l'on maintient à l'échelle du projet une bonne couverture de tests.

Un autre point important, souvent soulevé par les détracteurs de ce modèle, modèle largement adopté dans le monde de l'informatique, est la crainte de voir les développeurs coder en circuit fermé sans que les managers n'aient de contrôle ou même d'assurance sur la qualité du code. Ce faisant, un développeur mal intentionné pourrait introduire du code malicieux en toute discrétion. C'est pour celà qu'il est important de faire des revues de code, pratique collégiale largement introduite par les projets open-source ayant des membres aux quatre coins du globe. Cette pratique consiste à relire les modifications faites par un développeur, avant qu'elles soient incorporées dans le code officiel, seul ou en duo composé du développeur original et d'un autre développeur expérimenté. Le but est d'encourager les bonnes pratiques grâce à une présence physique d'une part, et d'autre part d'utiliser un levier psychologique bien connu : si l'on sait que l'on a des invités à la maison, on range et nettoie ladite maison avant de les recevoir. Et ainsi, si le développeur sait que son code sera relu par ses pairs, il est plus enclin à mettre tout en oeuvre pour que ledit code soit le plus lisible, clair, et de qualité que possible. Enfin un dernier avantage est que l'équipe a une vision plus globale et complète du projet au lieu de ne voir que la petite partie qui les concerne.

Nous avons vu comment maintenir un projet qui bénéficie de nombreux tests, dans un état aussi bon, voire meilleur, grâce aux pratiques citées ci-dessus. Nous avons adoptées toutes ces dernières dans la deuxième moitié de mon stage, cela résultant en un essor remarquable du

projet et de la motivation des développeurs.

Mais comment adapter un projet non conçu avec ces bonnes pratiques en tête ? C'est ce qui a constitué la majeure partie de la première moitié de mon stage. Il est tout à fait possible de le faire, même sur un projet de grande taille. Les étapes que nous avons effectuée sont les suivantes, de façon chronologique :

- Augmenter drastiquement le nombre de tests pour le code existant
- Forcer l'écriture de tests avec tout nouveau code
- Mettre en place les revues de code
- Automatiser le lancement des tests sur la version principale du code
- Insister pour que l'environnement de test soit toujours (ou le plus possible) valide, c'est-à-dire que tous les tests passent avec la dernière version du code
- Automatiser le lancement des tests sur les version alternatives du code (branches)
- Automatiser le déploiement du code en environnement de production
- Former les équipes aux bonnes pratiques de développement et outils de tests

Dans notre cas, nous avons utilisé un serveur de développement continu appelé Jenkins, qui permet de lancer des tests automatiquement quand un changement dans le code est fait, et ce quelque soit le projet ou le langage. Cela contribue à une compétition saine et productive entre équipes et projets pour atteindre un état stable et valide avec tous les tests qui passent avec succès.

3.2.5 L'environnement

On juge un artisan à ses outils. C'est aussi le cas pour un développeur. Ses outils sont au moins aussi important que le code qu'il produit, car il passera probablement plus de temps à compiler, tester, déboguer et déployer son code qu'à l'écrire. En conséquence, ces outils conditionneront sa productivité et la qualité du code qu'il engendre.

Pour toutes ces raisons, il nous est apparu vital d'améliorer la chaîne d'outils existants, qui bien qu'efficace, nous a semblé complexe à utiliser, à maintenir, et à améliorer. Les attributions de ces outils de métadéveloppement sont divers mais tous visent à faciliter le travail du développeur, avec une devise : tout ce qui peut être automatisé doit être automatisé. Voici les principales attributions de cet environnement :

- Installer les dépendances grâce à un gestionnaire de paquets
- Lancer le projet en mode déboguage
- Optimiser, minifier, concaténer et compresser le code pour le déploiement : le résultat est un "artefact" qui sera exécuté
- Déployer l'artefact sur un serveur distant
- Lancer l'analyse statique et vérifier les conventions de code
- Lancer les tests unitaires et/ou d'intégration
- Afficher une page d'aide expliquant les différentes actions possibles

Tout cela pour quatre environnements différents (développement, serveur de test, pré-production, production), avec dans l'idéal une seule commande pour chaque tâche.

Après ce travail de refonte de cette chaîne d'outils, il est désormais possible de lancer l'une ou l'autre tâche avec une seule commande, dans l'environnement de notre choix, ce qui est un grand pas en avant non seulement pour la productivité, mais aussi pour la qualité du projet.

Les implications peuvent paraître minimales, mais elles sont en fait énormes. Il s'agit de bousculer les habitudes de travail des développeurs, en menant la vie dure aux mauvaises pratiques, et en encourageant et facilitant les bonnes.

Une autre conséquence est l'abaissement de la barrière d'entrée du projet : un nouveau développeur peut plus vite se focaliser sur le code en étant libéré des autres tâches "ingrates", qui sont automatisées, et en suivant tout de suite les bonnes conventions, son travail s'intégrera plus facilement avec le reste de l'équipe.

3.3 Contraintes

Contraintes générales

Le projet est une application web et bénéficie donc des avantages de cette technologie : présence multi-plateforme, légèreté et accessibilité. Cependant cela signifie aussi des contraintes intrinsèques au web, dans un contexte d'entreprises ne possédant pas toujours la dernière version de leur navigateur web :

- Délai de réponse par page inférieur à 1 seconde, idéalement inférieur à 100 ms
- Support des anciennes versions d'Internet Explorer (9+)
- Pas d'utilisation de fonctionnalités très récentes d'HTML5 (3D, etc)

Ces contraintes sont somme toute assez légères et font sens dans un contexte d'entreprise : l'utilisation de la 3D ou le contrôle de la webcam n'ont que peu d'utilité dans notre cas.

De plus, certaines fonctionnalités très utiles, comme l'exécution de calculs parallèles avec l'utilisation de threads dans le navigateur pour effectuer des gros calculs ("Web Workers") sont fournies depuis un certain nombre de versions par tous les navigateurs web du marché, et nous sont donc accessibles.

Enfin, AngularJS permet d'abstraire les petites différences entre navigateurs en fournissant une API commune, et l'utilisation de modules externes pallie aux manquements de l'un ou l'autre navigateur.

Une contrainte qui aurait pu être gênante est la disparité de support de la nouvelle version de Javascript (ES6 ou ES2015), qui offre une pléthore d'améliorations, comme de nouvelles fonctions mathématiques, une vraie programmation orientée objet, ou encore la possibilité de marquer une variable constante ("const").

Toutefois, il est possible de profiter de la plupart de ces fonctionnalités grâce à des bibliothèques externes ("transpilers", cf TypeScript), ce qui est la voie que nous avons choisie.

Taille et diversité des données

The function of good software is to make the complex appear to be simple.

Grady Booch

Le domaine de la finance a comme caractéristique d'offrir une très grande diversité d'entités (largeur du spectre des données), allant des obligations aux produits structurés en passant par toutes les combinaisons possibles et imaginables (on pensera aux "Perpetual Callable Convertible Bond"), mais aussi en grandes quantités (profondeur du spectre des données) : des centaines de milliers d'instruments ou de transactions par exemple.

Ces deux contraintes orthogonales influent donc à la fois sur le code (comment importer un fichier csv comportant des centaines de milliers de lignes, chacune représentant une transaction, en un temps raisonnable ? Que ce passe-t-il si le nombre de types d'instruments différents augmente de 3 à 15 dans un futur proche ?), sur l'interface (combien d'entités dois-je proposer à l'utilisateur pour faire son choix ? Cette autre entité ne contient-elle pas trop de champs pour être affichée dans un tableau ? Qu'arrive-t-il au design de la page si trois champs sont ajoutés dans le futur ?), et sur l'architecture (est-ce que ma solution "scale", c'est-à-dire reste pertinente si la quantité de données augmente considérablement ? Cet algorithme doit-il être réimplémenté dans un langage plus rapide ? Doit-il s'exécuter côté serveur ou côté client ?), le tout en tenant compte des évolutions futures (le nombre de pays ou de monnaies différents reste relativement stable dans le temps, mais qu'en est-il des émetteurs d'obligations ? Quel est leur ordre de grandeur ? Cet ordre de grandeur peut-il changer subitement ? Comment doit-on s'y préparer ?).

Toutes ces questions sont des problématiques réelles que j'ai rencontré au cours de mon stage et auxquelles il a fallu répondre.

3.4 Déroutement du stage

Le stage a débuté par une phase nécessaire de découverte du projet et des notions métiers spécifiques au domaine financier, qui a duré environ un mois. Le reste du stage s'est ensuite divisé en trois lignes directrices : refonte d'interfaces existantes, création de nouvelles fonctionnalités, et amélioration des outils de développement.

3.4.1 Refonte d’interfaces

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

C.A.R. Hoare

Certaines interfaces utilisateur se sont révélées soit difficiles d’utilisation, soit plus adaptées aux évolutions des demandes métiers, notamment concernant la diversité et la quantité des données. Mon travail a alors consisté à comprendre les spécifications, c’est-à-dire les fonctionnalités existantes de la page, les questionner pour savoir si elles correspondaient aux attentes et usages des utilisateurs ainsi qu’à l’évolution des données, puis déterminer quelles fonctionnalités additionnelles seraient bienvenues sur la même page, et enfin implémenter ces fonctionnalités en simplifiant l’interface.

En résumé, il s’agit de faire au moins autant voire plus, mieux, et plus simplement, le tout faisant émerger un code plus clair et stable. L’amélioration décisive a enfin été de tester la page refondue pour s’assurer de sa qualité et de pouvoir détecter les régressions.

3.4.2 Nouvelles fonctionnalités

Une autre ligne directrice de stage a été l’ajout de nouvelles pages, ou vues, de l’application. Cela a bien sûr été l’occasion d’appliquer dès le début les bonnes pratiques, et lors de l’adoption de TypeScript, d’utiliser ce dernier.

En sus, les nouvelles pages ont été testées avec une couverture de code approchant 100%.

3.4.3 Amélioration des outils

Comme cité à d’autres endroits, une part significative du stage a été consacré à l’amélioration des outils, que ce soit au niveau du développement (analyseurs statiques, transpilers), des tests (deux types de tests dans deux environnements différents, à lancer individuellement ou ensemble), ou du déploiement (deux environnements différents de déploiement), dans une perspective de facilité d’utilisation, de rapidité, et de fiabilité, à l’échelle d’une équipe de plusieurs développeurs.

Comme tout investissement, ce n’est pas du temps perdu, mais bien du temps gagné, au long terme, comme l’a montré l’intégration couronnée de succès au projet d’un nouveau langage comme TypeScript.

Enfin, les nouveaux outils sont plus évolutifs, pour répondre à de futurs besoins, par exemple la redondance en production (si une machine cesse de fonctionner, une autre prend la relève automatiquement) ou le changement de fournisseur de machines dans le nuage.

3.5 Développement

3.5.1 Bonnes pratiques

Always code as if the guy who
ends up maintaining your code will
be a violent psychopath who
knows where you live

Martin Golding

Programs must be written for
people to read, and only
incidentally for machines to
execute

Arold Abelson

Git

Comme mentionné plus tôt, le projet est versionné avec git, ce qui procure moult avantages bien connus :

- Historique des modifications
- Annulation de changements
- Identification de l'auteur d'un changement
- Partage du code
- Statistiques
- Exécution automatique de scripts à différentes étapes du développement ("Git hooks")

Mais d'autres avantages moins connus sont aussi présents en utilisant le processus de développement par pull request, qui consiste pour chaque nouvelle tâche à faire une "branche" (copie indépendante du projet à un instant T). Nous avons adopté ce processus de façon systématique ce qui nous a procuré les avantages suivants :

- Gestion des versions et retour facile à une version antérieure
- Identification et comparaison des changements ("diff")
- Revue de code
- Application systématique des bonnes pratiques de développement
- Déploiement indépendant du développement
- Cycle itératif de développement d'une fonctionnalité

Analyseurs statiques

There are only two kinds of programming languages : those people always bitch about and those nobody uses.

Bjarne Stroustrup

Javascript est un langage aussi puissant qu'il est piégeux, et ce de par quatre caractéristiques principales :

- Faiblement typé (les types sont implicites, ils ne sont pas spécifiés par le développeur ; certaines conversions implicites peuvent passer inaperçues)
- Ensemble de fonctionnalités dépréciées mais pas supprimées par souci de rétro-compatibilité
- Dynamique (une variable peut changer de type, un objet peut se voir ajouter de nouveaux membres)
- Coexistence de types primitifs (passés par valeur) et non primitifs (passés par référence, ou plus exactement la référence est passée par valeur), le tout implicitement

Il se trouve que les deux premiers points sont facilement résolus par l'analyse statique, qui consiste à détecter erreurs et mauvaises pratiques avant l'exécution du code, en parcourant le code source. Ils sont particulièrement utiles sur de gros projets et remplacent en fait partiellement l'étape de compilation, qui est la force et la faiblesse des langages compilés.

Nous avons donc utilisés deux analyseurs statiques ("linters") de façon systématique, grâce aux "Git hooks") qui permettent l'exécution automatique d'un script ou d'un programme à certaines étapes du développement, dans notre cas avant un commit et un push.

Le développeur est ainsi alerté de quel extrait de code ne répond pas aux critères de qualité du projet avant que ledit code soit incorporé dans le projet, et il peut ainsi résoudre le problème en amont.

TypeScript

TypeScript est un langage développé par Microsoft qui est converti ("transpilé") en pur Javascript avant d'être exécuté. Il peut être vu comme une surcouche à Javascript et apporte de nombreux avantages, dont comme son nom l'indique, des types, plus précisément un typage fort : le développeur peut spécifier explicitement le type de telle variable ou fonction. Il répond donc à nos premières et troisièmes problématiques (typage faible et nature dynamique).

On peut s'interroger sur la nécessité de rajouter des types apparents à un langage qui a fait le choix de les cacher, surtout si cela ajoute une étape (la transpilation) au processus de build. Pour répondre à cette question légitime, il faut se replonger dans les origines de Javascript, qui fut créé il y a 20 ans, aux débuts du world wide web, comme un langage de script et d'animation pour des pages web simples à destination de non-programmeurs.

Aujourd'hui, le contexte est bien différent puisqu'il est utilisé autant côté serveur que côté client dans des applications riches, complexes, et imposantes (plusieurs milliers voire centaines

de milliers de lignes de code), sur de nombreuses plateformes allant du navigateur à l'ordinateur de bureau en passant par le téléphone portable et la tablette.

Si des types peuvent sembler être une contrainte dans une petite application, ils sont tout simplement indispensables dans une grande. Ils permettent bien sûr d'améliorer la lisibilité et la maintenabilité, mais aussi la qualité du code : est-il bien normal qu'un nombre se transforme en chaîne de caractères, et inversement, de façon impromptue, et ce plusieurs fois dans le même bloc de code ?

Le développeur est poussé à remettre son code en question et à avoir une vision à plus long terme, sans parler du coût en performances que requièrent ces nombreuses conversions, parfois implicites.

Enfin, le compilateur TypeScript peut déduire seul le type d'une variable sous certaines conditions ("type inference"), allégeant ainsi la charge du développeur.

Parmi les autres avantages de TypeScript, on pourra citer le support de tous les ajouts de la nouvelle version de JavaScript (ES6 ou ES2015), la présence d'énumération, d'interfaces, de modules, de paramètres optionnels ou par défaut dans une fonction... En bref, des fonctionnalités indispensables au développement et présentes dans la plupart des langages mais qui manquaient jusqu'à ce jour à JavaScript (ou étaient émulées de façon fastidieuse et/ou cryptiques).

Documentation

Tous les langages de programmations un tant soit peu répandus offrent la possibilité d'écrire des commentaires, ce qui montre bien la nécessité dans un projet de documenter son code, à des fins encore une fois de lisibilité et maintenabilité.

Nous documentons profusément le code en utilisant JSDoc qui est un outil de formattage des commentaires (similaire à JavaDoc) et qui permet de produire un document HTML à partir des commentaires.

Ce format permet notamment de préciser quels sont les arguments d'une fonction et quelle est sa valeur de retour (particulièrement utile en Javascript sans typage fort) mais aussi les préconditions, postconditions, complexité, etc de cette fonction, facilitant ainsi le travail à plusieurs sur le même projet ainsi que la pérennité de ce dernier.

3.5.2 Architecture

Perfection [in design] is achieved,
not when there is nothing more to
add, but when there is nothing left
to take away

Antoine de Saint-Exupéry

AngularJS suit le principe MVVM (Model View - View Model) et comme tout framework, impose une architecture conventionnelle. Un projet se divise donc habituellement en plusieurs

parties bien distinctes :

Services

Un service est chargé de récupérer des données, que ce soit depuis un serveur distant, via une API externe, ou localement (cookie ou local storage). C'est une abstraction qui permet de changer le fournisseur de données sans affecter le reste de l'application.

Un service peut aussi définir des fonctionnalités communes réutilisables à différents endroits de l'application.

Contrôleur

Le contrôleur fait la liaison entre un service et une vue. Il prépare les données récupérées depuis le service, les transforme éventuellement, et définit des comportements spécifique à une page, par exemple que faire lorsque l'utilisateur clique sur tel bouton. Il y a un contrôleur par page.

Vue

Une vue définit comment les données envoyées par le contrôleur vont être affichées sur la page. Il y a une vue par page.

Directive

Une directive est une sous-vue, destinée à être incluse et réutilisée dans plusieurs vues (i.e plusieurs pages).

Filtre

Un filtre définit une transformation de donnée destinée à être utilisée sur une vue. Un filtre est facile à mettre en place mais est coûteuse car la transformation de données se réapplique à chaque action de l'utilisateur, au lieu d'une seule fois comme dans le contrôleur.

3.6 Améliorations

3.6.1 Optimisations

Un cas intéressant d'optimisation qui impacte directement l'utilisateur est la page d'importation de données au format CSV. L'ordre de grandeur des données dans un cas d'utilisation réel est de 100 000 à 200 000. Or pour autant de données, le temps initial de chargement était de 45 secondes. L'objectif fut donc de diminuer ce temps autant que possible.

La première étape fut de refondre le code, notamment en enlevant certains motifs connus pour ralentir l'exécution, tels que déclarer une fonction à l'intérieur d'une boucle, au lieu de le faire à l'extérieur, résultant en autant de déclarations de la même fonction que de tours de boucle (ici, 100 000 !). Cette étape fut guidée par les analyseurs statiques et l'étude de multiples mesures de performance en ligne pour décider du construct optimal, choix au final dicté par la comparaison de mesures de performances de chacun au sein de l'application.

La seconde étape fut d'optimiser le tri des données. Chaque donnée (ligne dans le csv) contenant une dizaine de champs (colonnes dans le csv), chacun demandant à être validé, parfois en fonctions d'autres champs, le tri consiste en une part significative du temps d'exécution total (environ 25%). Au lieu d'effectuer de nouveau la validation de multiples fois comme le

tri le requiert, cette validation fut effectuée une seule fois au début puis enregistrée au sein de chaque donnée. Puis, puisque la meilleure façon d'accélérer un programme est de lui faire faire moins d'opérations, nous nous sommes posés la question du bien fondé du tri. Cela est-il requis pour l'utilisateur ? Ceci améliore-t-il son expérience ? Il est apparu que non et nous avons donc supprimé complètement ce tri superflu.

Grâce aux outils de mesures du navigateur, nous avons pu précisément évaluer le temps pris par chaque opération, et l'optimiser en conséquence. Par exemple, la lecture du fichier csv est infinitésimale (< 1 seconde) par rapport au temps total. Il est donc superflu d'optimiser cette partie.

Au final, le temps d'exécution après optimisation est environ de 10 secondes, ce qui signifie un gain d'un facteur 4, avec une expérience utilisateur plus fluide et plus agréable, une validation accrue, et de nouvelles fonctionnalités (telle que le choix de la précision de l'arrondi pour les nombres réels, quantité variant d'un système client à un autre).

4 Remerciements

Plusieurs personnes m'ont apporté une aide significative sur ce projet et je tiens à les remercier chaleureusement ici :

- Fabrice DOUCHANT, mon maître de stage pendant la première partie de mon stage
- Sven REBER, mon maître de stage pendant la deuxième partie de mon stage
- Mathieu CAMBOU, pour ses conseils et sa vision à long terme du projet
- Martin SOUBEYRAND, pour ses conseils techniques
- Jonathan BALLE, pour son aide technique

5 Conclusion

Les entreprises offrant la liberté de remettre en question des pans entiers d'un projet tout en faisant en sorte que le stagiaire soit force de proposition ne sont pas légion, et cette chance m'a été offerte tout au long de ce stage de six mois. J'ai eu la chance de vivre le travail collaboratif en participant à un gros projet existant, mais aussi d'expérimenter, de proposer de nouvelles idées et de les voir se réaliser, de questionner, d'améliorer, et d'apprendre au contact de développeurs et mathématiciens brillants.

J'ai pu approfondir mes connaissances sur des sujets génériques et réutilisables tels que le génie logiciel, l'organisation d'un projet, la collaboration, et l'élaboration d'une interface utilisateur. J'ai également pu apprendre une myriade de notions dans un domaine qui m'était inconnu, la finance, à l'intersection des mathématiques, de l'économie, et de l'informatique.

En bref, ce stage m'a apporté énormément. L'intérêt du stage était à la hauteur des défis rencontrés. Je suis reconnaissant à EdgeLab et à mon maître de stage de m'avoir fait confiance et donné cette chance.

Au final, je finis ces six mois satisfait de ce qui a été accompli et avide de continuer à explorer ce domaine fascinant de la finance.

6 Annexes

6.1 Outils utilisés

6.1.1 Langages utilisés

CSS

CSS est un langage spécifique à un domaine : le style d'une page web. CSS3 en est la dernière version et permet notamment des animations complexes et fluides.

HTML

HTML est le langage de balisage standard pour définir quels éléments sont présents sur un page web. HTML5 en est la dernière version. Cette dernière se voit enrichie de constructs appartenant normalement à un langage de programmation tels que conditions, boucles, déclaration et définition de variables, etc.

Javascript

Javascript est un langage de script créé initialement pour définir des comportements additionnels sur une page web, et est devenu un langage multiplateforme utilisé autant côté serveur que côté client, et s'est enrichi pour répondre aux besoins nouveaux du web tels que manipulation d'images, simulation 3D, audio, etc.

Sass

Sass est une surcouche à CSS : il permet de définir des variables, des conditions, des boucles, élevant quasiment CSS au rang de véritable langage de programmation.

Typescript

Typescript est un langage de programmation développé par Microsoft qui enrichit Javascript avec des fonctionnalités jugées par beaucoup comme indispensable : typage fort, véritable programmation orienté objet, etc, autorisant un analyseur statique à détecter des erreurs de programmation avant l'exécution du programme.

6.1.2 Bibliothèques utilisées

Les principales bibliothèques utilisées dans le projet sont les suivantes :

AngularJS

AngularJS, souvent abrégé Angular, est un framework créé par Google pour développer des applications web riches et structurées. Il enrichit HTML5, fournit des API uniformes selon les navigateurs et leurs différentes versions, et dispose d'un large écosystème de modules provenant de la communauté. Il s'est de facto imposé comme l'un des frameworks web les plus utilisés des cinq dernières années. La version 2 est en bêta.

JQuery

JQuery est probablement la librairie web la plus utilisée des dix dernières années avec pour vocation première la manipulation du DOM de page web avec une API uniforme selon les navigateurs et les versions. Bien qu'AngularJS soit un sérieux concurrent, cette librairie complète ce framework.

Lodash

Lodash est une librairie Javascript qui complète et enrichit la librairie standard notamment dans le paradigme fonctionnel. Elle a été téléchargée plus d'un milliard de fois depuis sa création.

6.1.3 Outils divers utilisés

Grunt

Grunt est un outil configurable pour gérer toutes les étapes du cycle de développement d'un projet : rechargement automatique de la page lors d'un changement dans le code, lancement des tests, build du projet (minification, compression des images, etc), et déploiement sur serveur distant (avec mise en cache des fichiers), tout cela avec une seule commande.

Jscs

Analyseur statique de formattage de code similaire à Jshint.

Jshint

Jshint est un analyseur statique de style du code, pour que l'indentation, le nombre d'accolades ou de parenthèses ou encore les retours à la ligne soient uniformes dans le projet.

Tslint

Tslint est le pendant de Jshint pour Typescript. De part le typage fort de Typescript, il est de plus capable d'alerter si une variable n'existe pas dans une classe, si le nom comporte une typo, ou si la variable est utilisée de façon contraire à son type.

Tsc

Le transpiler qui transforme le code Typescript en code Javascript.

6.2 Glossaire

Analyseur statique

Outil analysant le code source, sans avoir besoin de l'exécuter, signalant les mauvaises pratiques et/ou les différences de style (indentation, retour à la ligne, etc).

API

«Application Programming Interface», interface de programmation. Ensemble normalisé de classes et de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels.

Framework

Ensemble cohérent de composants logiciels structurels.

Transpiler

Outil analogue à un compilateur, qui transforme un code source en un code source dans un autre langage (contrairement au compilateur qui transforme un code source en format binaire).