



Faustin DEWAS- Apprenti M2

L'intégration de bons principes de développement informatique au milieu d'un processus d'évolution d'une application

Tuteur Enseignant
Christine SAUVAGE

Tuteur Entreprise
Nicolas GRUEL



Remerciements

J'aimerais remercier Nicolas, Jérôme, Florian, ainsi que tous les autres développeurs et membres d'Aneo qui m'ont accompagné pendant ces 2 ans chez Aneo, en me guidant quand je me perdais, en répondant à mes interrogations, et ce, jour après jour.

A SOLID work based on simplicity	4
I – Un nouveau départ	5
II – Aneo	6
1. Présentation	6
2. Organisation et cadre de vie	6
3. Clubs	7
4. Evènements	7
III – ArmoniK, Chef d’Orchestre	8
1. Application et applications	8
2. Equipe et organisation	9
3. Mission	10
4. Interface administrateur, à quoi sers-tu ?	10
5. Un code fonctionnel, à quel prix ?	11
IV – Introduction aux principales technologies utilisées	11
1. Git, sautons d’une branche à une autre	11
2. GitHub, garde forestier	15
3. Angular	16
4. gRPC	18
5. Docker	19
6. Kubernetes	20
7. Terraform	21
V – Principes de développements	21
1. Dette technique, qu’est-elle donc ?	21
2. Mot d’ordre : simplicité	22
3. Redondance de code, comment l’éviter	22
4. Nomenclature et structure	23
5. Tout pour un code SOLID	25
6. DevOps	30
7. Tests Unitaires	32
8. Documentation Développeur	35
9. Sécurité et Programmation	35
10. Outils	38
VI – Vie d’un utilisateur	39
1. Expérience Utilisateur	39
2. Documentation Utilisateur	41
3. Retours utilisateurs	41
VII – Résultats	42
VIII – Pour finir en beauté	43

A SOLID work based on simplicity

After four years of studies at Efrei Paris, I decided to complete my formation with an apprenticeship at Aneo, a software development & consulting specialized company. Last year, I an intern in the same company, on the project ArmoniK, where I was a front-end software developer. That was the occasion to have a first look on the software development world. But the apprenticeship had a whole different goal since I became the main developer of the ArmoniK Graphical User Interface. I came to Aneo not to discover, but to apply the knowledge I gained over these last four years.

To be more precise, my mission was to get rid of the technical debt of the application I used to work on. This one has been almost entirely recreated in a short amount of time, making its code very unstable and hard to read. And since my recent software developer education was about creating and maintaining code thanks to some informatic principles, it has been decided that it would be the subject of my apprenticeship.

This report will firstly address the topic of ArmoniK, a High Throughput Computing tasks orchestrator designed for cloud providers and on-premises deployments. It will be the occasion to learn more about the ArmoniK Graphical User Interface, a tool made to monitor and operate the ArmoniK application directly in a browser. Then, it will be addressing the different technologies used to create and work on this ambitious project from Aneo. By taking a detailed look on Git, GitHub, Docker, Kubernetes, and other technologies, we will have the context to understand the main topic of this report.

We will then define technical debt, a problem that prevents a lot of software from evolving at a normal pace, sometimes preventing any kind of evolution. We will then see how to avoid technical debt, by studying a few principles: KISS, YAGNI, and the most important and complex, SOLID. While the first two are easily understandable in their principle, SOLID is a cluster of five important principles in Object Oriented development. But they do share the same goal: simplicity. A good informatic software is a simple one.

We will then go further. The principles are merely the fundamentals on how to build a good software application. Security, documentation, and testing are also tools that will increase the quality of a software. We are going to explore them all, and alongside them, introduce the DevOps philosophy. This informatic methodology aims to automate as many processes as possible, to, once again, simplify the work of developers in the making of an application.

After explaining all those principles, tools and philosophy aiming to increase code quality, the report will then address a topic as important as the last one: the user experience. Since the Graphical User Interface of ArmoniK is a Front-End solution, its use must be as convenient as possible for every user. User Experience designing is a difficult and complex job that requires a lot of knowledge and must not be put apart when creating an application.

This report will then end by giving the results and impacts of applying the different principles for both the developers and the users of the application. We will see how applying them had an impact on the global stability, number of bugs introduced, and the simplicity of modifying the code.

I – Un nouveau départ

En mars 2023, après un stage de 5 mois, je quittais mon rôle de développeur de l'interface administrateur d'ArmoniK chez Aneo. J'avais profité de ce stage forger mes connaissances dans certaines technologies et méthodologies utilisées dans des projets informatiques de grande envergure. C'était avant tout une expérience pour découvrir, pour commencer à rentrer dans ce monde du développement.

Lorsque je suis revenu en septembre 2023, cette fois-ci en tant qu'alternant, mon but n'était plus le même : je connaissais déjà les outils, les méthodes et la technique de ce projet. Cette fois-ci, je n'étais plus là pour apprendre, mais pour appliquer mes connaissances les plus théoriques. Je devais dorénavant me prouver que j'étais capable d'utiliser les connaissances que acquises durant ces 5 années d'études. Cela par la recherche de différentes solutions avant d'en choisir une, par la découverte de nouvelles technologies, ou par la réécriture d'une partie du code.

Je suis revenu en tant que développeur principal de l'application sur laquelle j'avais auparavant travaillé. Cette fois-ci avec le rôle de faire évoluer l'application, et de le faire de manière que cela soit de plus en plus simple. Cependant, à mon retour, j'ai trouvé une application totalement reconstruite, certes suivant la même logique qu'auparavant, mais totalement réécrite pour répondre plus facilement à de nouveaux besoins clients. Malheureusement, cette reconstruction s'était faite trop rapidement. De ce fait, l'interface administrateur de ArmoniK s'est endettée techniquement. Et cette dette, ce fut à moi de la payer, avant que cette corruption ne se retrouve éparpillée au travers des milliers de lignes de code que constituent l'interface administrateur.

Ces connaissances dont je parlais tout à l'heure allaient m'être utiles. Les différents principes de développements que j'ai appris ces dernières années avaient tout à fait leur place pour sauver le code de cette application. Proposer une implémentation propre de ceux-ci serait donc la preuve de ma maîtrise de ces-dits principes. C'est pour cette raison que le sujet de ce mémoire porte sur **l'intégration de bons principes de développement informatiques au milieu d'un processus d'évolution d'une application**.

Ce mémoire commencera par présenter l'entreprise qui m'a accepté pour cette alternance : Aneo. Ensuite, une courte présentation du projet ArmoniK et de son interface administrateur sera de mise. Pour continuer dans cette lancée de contextualisation, nous étudierons plus en détail les technologies et outils utilisés pour ce projet, ce qui permettra d'appréhender plus efficacement les éléments principaux de ce mémoire. Enfin, nous verrons quels principes de développements ont pu être mis en place, leur impact, ainsi que quelques cours exemples de code. Pour suivre, nous passerons côté utilisateur de l'application, et nous verrons comment faire évoluer leur expérience. Pour finir, nous ferons le point sur le produit actuel de cette année d'alternance, notamment au niveau de l'application fournie, de ses évolutions, des pistes à améliorer.

II – Aneo

Avant de parler d'ArmoniK, il est important d'apporter quelques précisions sur l'entreprise dans laquelle j'ai pu passer mon alternance : Aneo.

1. Présentation

Aneo est un cabinet de consulting spécialisé dans la transition numérique et organisationnelle des entreprises. Fondée en 2002 par Pierre Sinodinos, Aneo a subi pendant son existence plusieurs grandes restructurations organisationnelles et changements de stratégies. Aujourd'hui composée d'un peu plus de 150 employés, consultants ou développeurs internes, commerciaux et RH, Aneo est une entreprise *hybride* en plein essor. Hybride car elle est capable de toucher à tout : management, développement informatique, cloud, HTC... Tout en développant et finançant d'ambitieux projets internes.

Evidemment, Aneo possède de nombreux concurrents. Parmi lesquels Arollad, Cellenza, Finaxys, Atos, Orange Business Services... Pour ne citer que ceux qui rivalisent en matière de consulting d'information et de technologie. Au niveau de l'organisation des entreprises, la concurrence se retrouve notamment chez Wemanity, bartle ou encore julhiet sterwen. Au niveau des projets internes, les entreprises leaders des marchés dans lesquels Aneo se positionne sont capgemini, aubay, wavestone, Devoteam ou encore Sopra.

2. Organisation et cadre de vie

Un point particulier sépare Aneo de ses concurrents : la manière de laquelle l'entreprise traite ses collaborateurs. L'entreprise a veillé à aplatir sa hiérarchisation le plus possible : nous sommes loin de la pyramide avec les différents managers ayant tous du pouvoir sur l'autre. Ici, dans une équipe, tout le monde aura la même importance, que l'on soit stagiaire ou membre senior. Les avis de chacun sont écoutés, discutés et débattus. Cela permet aux employés de se sentir plus acceptés dans l'entreprise, et permet aux nouveaux arrivés de sociabiliser plus facilement. Cette hiérarchie aplatie permet aussi de simplifier les processus d'administrations en réduisant les intermédiaires. Généralement, les personnes concernées par une problématique ou un sujet quelconque vont pouvoir interagir directement et facilement.

Les locaux de l'entreprise sont aussi très adaptés pour cette organisation : il s'agit d'un open-space organisé en *flex-office*. Cela signifie simplement que personne n'a de place attitrée. Chaque jour, n'importe quel collaborateur peut changer de place sans que cela ne pose de problème à un autre. Cela permet de mélanger les personnes ayant des compétences et des cultures différentes, permettant à chacun d'apprendre de l'autre.

L'entreprise a divisé ses collaborateurs à travers des « cercles » selon leurs compétences. Ainsi, les membres possédant le plus de compétences techniques se retrouvent dans le cercle *Rouge*, tandis que ceux avec des connaissances poussées en management et organisation vont se retrouver dans le cercle *Jaune*. Ces communautés sont séparées en sous-cercles et représentent le cœur de métier d'Aneo. En dehors de ces deux cercles, nous retrouvons la vente, qui va s'assurer que l'entreprise fonctionne correctement. C'est le cœur d'Aneo, et, dans cette dernière communauté, nous pouvons retrouver le Codir, la direction et les différents services d'aide aux collaborateurs de l'entreprise.

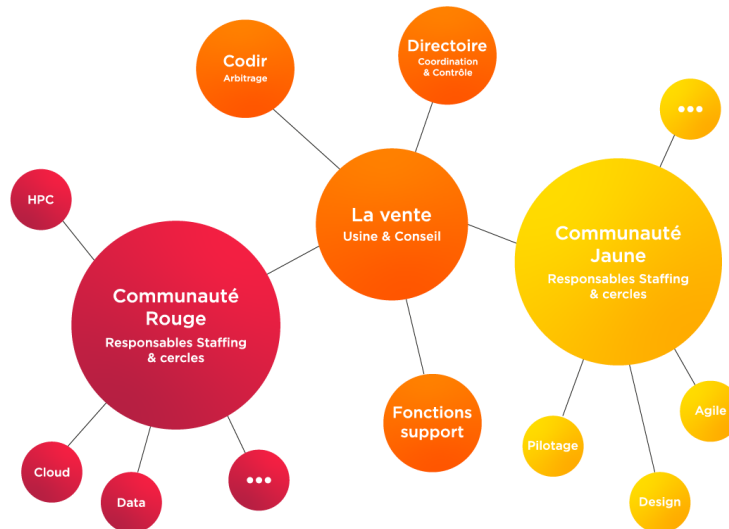


Figure 1 - Organigramme d'Aneo

3. Clubs

Aneo a mis en place un système de clubs pour ses employés. Comme pour des associations étudiantes, les collaborateurs peuvent se regrouper et former un club basé sur une de leur passion. Nous pouvons alors retrouver des clubs d'œnologie, de jeu vidéo, de volleyball, de football... Tout ça n'est pas sans rappeler les nombreuses associations de l'Efrei. En tant qu'étudiant, j'accorde une grande importance à la vie associative de mon école, et redoutais de devoir la quitter. Cela m'a positivement surpris de trouver un équivalent dans le monde professionnel, et de voir à mon retour que cet aspect est toujours aussi important.

4. Evènements

Tout entreprise souhaitant conserver ses collaborateurs se doit d'organiser différents types d'évènements pour ces derniers. Autrement, l'ennui s'installe. Chez Aneo, ces évènements ont des buts variés : de la présentation des dernières nouvelles de l'entreprise aux simples fêtes.

Commençons avec les *Aneo Lunch*. Ces évènements se produisent un vendredi sur deux et ont pour but de présenter les derniers résultats et changements de la boîte. Tous les collaborateurs de l'entreprise sont invités, soit au siège de l'entreprise, soit en visio-conférence. L'Aneo Lunch se divise généralement en deux phases. Lors de la première, les nouveaux collaborateurs, stagiaires et alternants vont se présenter et leur parcours professionnel ou scolaire. Par la même occasion, certains projets des collaborateurs sont présentés, afin de partager leur travail à l'ensemble de l'entreprise. Enfin, l'équipe RSE (Responsabilité Sociétale des Entreprises) va présenter les diverses évolutions réalisées pour augmenter la qualité de vie dans la boîte. Pour clôturer le rituel, les collaborateurs vont pouvoir partager un repas, ce qui est une nouvelle occasion de discuter et partager des connaissances.

L'entreprise organise aussi fréquemment des afterworks ouverts aux collaborateurs qui ne sont pas encore arrivés dans l'entreprise, mais aussi à d'autres représentants d'entreprise, pour pouvoir nouer des liens forts avec ces boîtes. Avant mon entrée dans la boîte l'année dernière, j'ai eu l'occasion de participer à un de ces afterworks, qui, à l'époque, m'a permis de faire la rencontre de mes futurs collègues et de commencer à me plonger dans la culture de cette entreprise.

Certains évènements présents l'année dernière, lors de ma période de stage, ont malheureusement disparus : je peux notamment citer les après-midi *knowledge*, qui avaient pour but de laisser un collaborateur présenter à la fin du mois un cours sur un sujet de son choix.

III – ArmoniK, Chef d’Orchestre

Tout au long de cette année, j’ai eu l’occasion de travailler sur une partie du projet ArmoniK, l’Admin GUI (pour Graphical User Interface), ou interface utilisateur. Avant de rentrer dans le détail des technologies utilisées pour créer et gérer un tel projet, il est important de préciser quel est son but, sa raison d’être ; mais aussi de présenter l’équipe qui l’a développé au fil des ans, ainsi que mon rôle dans celle-ci.

1. Application et applications

ArmoniK est un orchestrateur de tâches HTC, mis en production pour la première fois en juin 2022. Il a été développé par Aneo sur la base de HTC-Grid, un ancien projet HTC développé par Amazon Web Services. Construit à la fois pour être déployé *On Premise* (serveurs internes d’une entreprise) ou sur des services *Clouds*, il est capable de gérer un grand nombre de tâches de durée très variable, tout en gérant efficacement les ressources qui lui sont attribuées. Il trouve son utilité dès qu’un utilisateur a besoin d’effectuer de nombreuses tâches en peu de temps, que leurs résultats soient peu liés, et qu’elles aient une grande variation de besoins en ressource physiques (processeurs, mémoire utilisée...).

Le nom « Orchestrateur » est assez explicite. Un peu comme un chef d’orchestre, un orchestrateur va placer et gérer des éléments à travers les ressources qu’il dispose. Plus précisément, il va automatiser le processus d’organisation, de coordination, de gestion de systèmes informatiques complexes, middlewares ou/et services. Dans le cadre d’ArmoniK, ces ressources sont les tâches HTC.

HTC, ou *High Throughput Computing*, est un peu moins évocateur ; en français, il se traduit par « Calcul haut débit ». Il s’agit d’une méthode permettant à un service de calculer un très grand nombre de tâches indépendantes le plus rapidement possible, i.e. avec le plus grand débit. Le cas le plus représentatif du HTC se retrouve dans l’application de méthode dite de *Monte-Carlo*.

ArmoniK permet donc de gérer et de distribuer des calculs indépendants de manière parallélisée dans un environnement informatique, généralement sur des services cloud. Les calculs sont gérés par un algorithme complexe pour être traités le plus rapidement et efficacement possible. Aujourd’hui, ArmoniK fonctionne notamment sur AWS et GCP, son architecture particulière et hétérogène lui permet d’être déployé sur différents services de cloud providers ; en d’autres termes, il est multicloud.

C’est aussi un projet Open Source basé sur *Kubernetes* (orchestrateur de containers), notamment financé par Aneo et ses clients. Cela signifie que l’équipe d’Aneo est la principale source de développements de l’application, mais que le code écrit est public et ouvert aux modifications externes. Le projet n’est évidemment pas le seul dans son domaine et possède quelques concurrents : *IBM Spectrum Symphony* ou *Tibco DataSynapse*, pour ne citer qu’eux. ArmoniK se positionne sur le marché par son état d’open source et son côté cloud native, mais aussi par la possibilité d’appliquer modifications spécifiques selon les besoins de certains utilisateurs et clients.

Concrètement, ArmoniK permet aujourd’hui de faciliter des calculs de prédictions monétaires dans la banque, mais son utilisation peut être appliquée dans d’autres domaines. L’équipe d’ArmoniK a d’ailleurs mis en place un exemple d’application de RTX (Ray tracing, création d’une image à l’aide de rayons diffusés dans une scène 3D) à partir de tâches HTC sur un cluster d’anciennes *Raspberry Pi*.

Le projet en lui-même est décomposé en plusieurs sous projets : *infra*, *core*, *api*... et j’ai pu travailler sur le projet *gui*. L’interface Administrateur ou Admin GUI permet de visualiser et gérer les tâches en temps réel, tout en exposant des données importantes pour un utilisateur.

2. Equipe et organisation

L'équipe d'ArmoniK n'est pas bien grande, mais très compétente et a beaucoup évolué au fil des ans, pour finalement s'organiser en 5 sous-équipes, chacun aillant un rôle différent. Dans le tableau suivant, vous pourrez retrouver ces différentes équipes ainsi que leur responsable principal :

Direction Générale				
Product Management Wilfried Kirschenmann				
Vente	Développement Produit	Intégration	Support	Suivi Interne
Thierry Pecoud	Nicolas Gruel	Damien Dubuc	Nicolas Gruel	Thomas Lefèvre

Le nom de ces équipes est assez évocateur ; mais rentrons un peu plus en détail sur chacune d'entre elles :

- **La vente** consiste surtout à la recherche et au démarchage de nouveaux clients, notamment pour essayer et financer le développement d'ArmoniK.
- **Le développement produit** se concentre sur les nouvelles fonctionnalités, directement prévues par l'équipe de développement. C'est dans cette équipe que je me trouve.
- **L'équipe d'intégration** va aider le client à intégrer ArmoniK dans leur cycle de travail : configuration, mise en production, tests... Cette équipe peut avoir à modifier le code source d'ArmoniK pour l'adapter aux besoins du client. Ils vont aussi récupérer les demandes clients et les transmettre à l'équipe de développement.
- **L'équipe support**, quant à elle, a pour rôle d'aider un ou plusieurs clients en cas de soucis ou de besoins particuliers. Ce sont les premiers contactés en cas de problème.
- Enfin, **le suivi interne** veille à ce que le projet aille dans la bonne direction et à une vitesse acceptable, afin de s'assurer que les objectifs soient bien atteints dans le temps imparti.

L'équipe suit une méthodologie agile conçue spécialement pour ce projet et inspiré des méthodes kanban. Il est à noter que la méthode agile la plus connue, *scrum* et ses sprints de 2 à 4 semaines n'a pas été retenue. L'équipe considérant que les sprints fixes étaient une véritable épée de Damoclès suspendue sur la tête des développeurs et ralentissaient fortement les développements.

A la place, les membres les plus expérimentés de l'équipe ont pu mettre au point leur propre méthode agile. Plutôt qu'un Product Owner, les Tech Lead vont discuter directement avec le client. Les sprints n'existent plus, les livraisons sont faites quand le code est prêt, environ tous les mois. De cette manière, les développeurs vont pouvoir limiter le bouche-à-oreille entre les clients et les autres managers dû à la communication indirecte. L'équipe de développement va pouvoir accepter ou refuser les demandes des clients plus rapidement et éviter de perdre du temps en discussions inutiles. La suppression des sprints permet aux développeurs d'avancer sans pression sur le code de l'application, tout en leur permettant de s'assurer de la qualité de ce dernier. C'est une méthodologie avant tout basée sur la confiance : si un des développeurs ne travaille pas ou trop peu, il va peut-être handicaper l'ensemble de l'équipe. Ce n'est donc pas une méthode adaptée à tous ; fort heureusement, un tel phénomène ne s'est pas produit dans l'histoire d'ArmoniK.

3. Mission

Mon rôle est simple : je suis le développeur principal de l'interface administrateur d'ArmoniK. Cela signifie que je suis le principal responsable de l'intégration de nouvelles fonctionnalités, de la correction de bug, des outils de mesure de la qualité du code et de la maintenance de ce dernier.

Déjà développeur sur ce même projet l'année précédente, j'avais la connaissance nécessaire pour prendre le rôle de développeur principal. Et ce, même si l'application avait été réécrite entièrement pendant mon absence : les technologies utilisées n'ont pas changé.

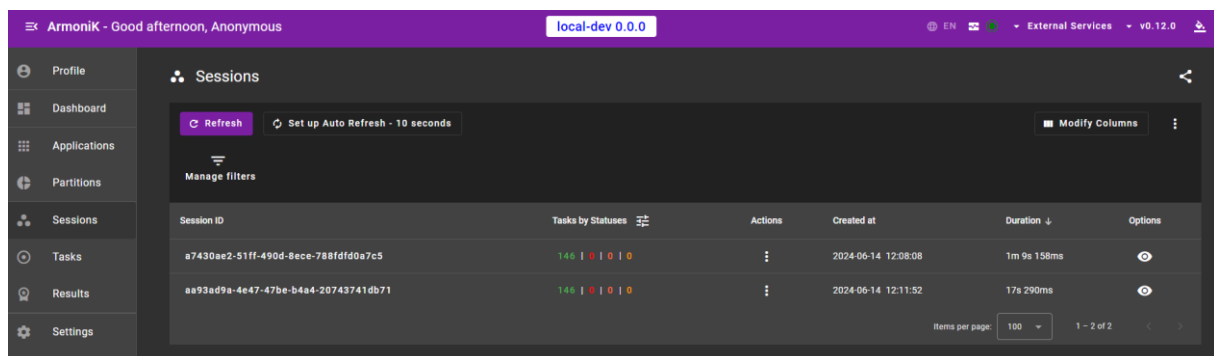
Si l'interface administrateur a été entièrement réécrite, c'est notamment à cause de raisons techniques. Les bibliothèques de code (ou package : ensemble de code téléchargeable implémentant des fonctionnalités) utilisées avaient atteint leur limite, et n'étaient plus aussi bien maintenu qu'à leurs débuts : par conséquent, elles ont cessé d'évoluer. Cela limitait techniquement toute modification de l'interface administrateur, à moins de contourner les implémentations des packages. Réécrire l'application avec des bibliothèques plus grand public et mieux maintenues était alors la solution pour sortir de cette impasse.

4. Interface administrateur, à quoi sers-tu ?

Pour comprendre à quoi sert l'interface administrateur, il faut d'abord comprendre comment ArmoniK gère les tâches HTC. Un utilisateur, lorsqu'il souhaite lancer des calculs, va soumettre une **application**. Une application peut être lancée plusieurs fois à travers différentes **sessions** de calculs. Une session va être composée d'un ensemble de **tâches**, chacune responsables d'un calcul et élément principal d'ArmoniK. Les tâches retournent des **résultats**, qui peuvent être utilisés comme entrée pour d'autres tâches. Les sessions peuvent être exécutés sur différents environnements, appelés **partitions**.

Lorsqu'une session est exécutée, il est possible qu'une ou plusieurs de ses tâches échouent, qu'elles soient en erreur. ArmoniK gère ces cas de figure, mais un utilisateur de l'application souhaite savoir pourquoi une tâche en particulier a échoué. C'est ici que l'interface administrateur d'ArmoniK trouve son utilité.

Développée avec le framework JavaScript *Angular*, son rôle est d'exposer des données des différentes applications, partitions, sessions, tâches et résultats exécutées dans ArmoniK. Les données sont affichées dans différents tableaux personnalisables et interactifs. Il est possible pour un utilisateur de trier les données par un champ spécifique, ou de les filtrer par des conditions pouvant être très complexes. Grâce à cette interface, un utilisateur est capable de retrouver les différentes tâches d'une session, en temps réel, et d'accéder à leur état et autres diverses et nombreuses informations.



Session ID	Tasks by Statuses	Actions	Created at	Duration	Options
a7430ae2-51ff-490d-8ece-788fd0a7c5	146 0 0 0		2024-06-14 12:08:08	1m 9s 158ms	
aa93ad9a-4e47-47be-b4a4-20743741db71	146 0 0 0		2024-06-14 12:11:52	17s 290ms	

Figure 2 - Interface Administrateur d'ArmoniK, page des sessions

C'est un outil qui affiche un nombre important de données : il doit donc être clair, facile à utiliser et efficace. C'est avec l'interface qu'un utilisateur d'ArmoniK va utiliser l'application. Une interface de mauvaise qualité serait synonyme d'un projet de mauvaise qualité, il est donc très important qu'elle soit le plus qualitative possible.

5. Un code fonctionnel, à quel prix ?

Lorsque je suis arrivé, j'ai pris connaissance de la réécriture de l'application, en sachant que celle-ci s'était faite en un temps record : un mois pour un unique développeur. Les demandes clients avaient été répondues, mais la qualité du code en lui-même en avait pâti. Nous avons sacrifié la qualité pour la fonctionnalité, nous avons donc accepté d'avoir une dette technique, qu'il allait falloir rembourser un moment ou un autre.

Le principal sujet de mon alternance fut donc la mise en place de principes visant à améliorer la qualité de ce code, à l'aide des connaissances que j'ai acquises au cours de ma formation de développeur. C'est ce même sujet qui est à l'honneur de ce mémoire, que nous allons explorer dans quelques temps.

IV – Introduction aux principales technologies utilisées

1. Git, sautons d'une branche à une autre

a. Petite histoire de git

Git est un outil très utilisé aujourd'hui par presque tous les développeurs, peu importe le langage de programmation sur lequel ils travaillent. Il s'agit d'un outil de versionnage de code, qui va permettre de gérer facilement la création et la modification de code tout en maintenant le bon fonctionnement de celui-ci. Il se base sur un système de *branches*, de *commits* et de *merges* pour fonctionner, que nous allons explorer d'ici un instant. Il s'agit d'un outil comprenant de nombreuses fonctionnalités, et nous aborderons ici les plus communes.

Repositories

Git utilise un système de *repositories*, de « dépôts » pour stocker le code. Un développeur qui initie un projet doit forcément le référencer à un repository. Celui-ci lui permettra de sauvegarder son code en ligne, tout en le partageant à d'autres développeurs ayant accès à ce repository. Certains sont privés, ce qui signifie donc que le code n'est accessible à personne d'autres que les développeurs. Cependant, d'autres sont publics, et l'entièreté du code est rendu disponible à n'importe qui. Les projets Git publics sont connus comme des projets *Open-Source*.

Commits & Pushs

Lorsqu'un développeur souhaite sauvegarder du nouveau code, il va créer un *commit*. Ce commit sauvegardera l'ensemble des modifications faites depuis le précédent commit : le code rajouté ainsi que celui supprimé, ligne par ligne. Ensuite, il pourra *push* le commit vers le repository afin de le rendre disponible à tous les autres développeurs du projet, et pour sauvegarder son code. N'importe quel développeur ayant accès au dépôt Git peut se déplacer à n'importe quel commit à l'aide de la commande *checkout*.

Branches

Tous les projets Git possèdent une branche principale, *main* ou *master* (généralement), sur laquelle l'ensemble des commits se retrouve. Avant de commencer à développer une nouvelle fonctionnalité ou de modifier le code, un développeur va s'isoler de la branche principale en en créant une nouvelle. Cette nouvelle branche possèdera l'ensemble des précédents commits de la branche

d'origine. Une branche permet d'isoler tous les commits qui lui seront envoyés des autres branches. De cette manière, un développeur s'assure que même si son code n'est pas fonctionnel, il peut le sauvegarder et le partager sans rendre inopérable le code stocké sur les autres branches. On peut voir les branches comme des divergences permettant aux développeurs de s'assurer que leur travail ne rendra pas le code du projet inopérant.

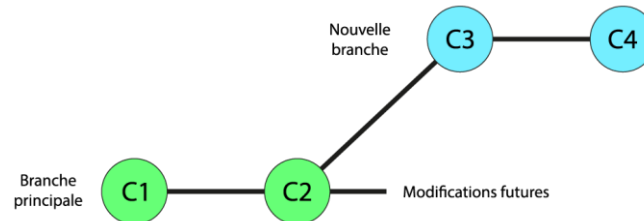


Figure 3 - Représentation Graphique d'une branche Git

Merge

Une fois que le développeur a fini son travail sur sa branche, il peut procéder à un *merge*, à la fusion de sa branche avec celle principale. L'ensemble des modifications de la branche seront alors rassemblées en un unique commit qui sera pushé comme dernier commit de la branche principale.

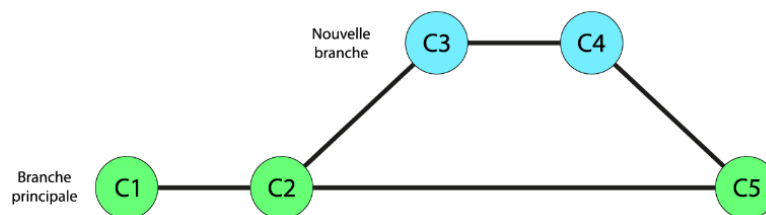


Figure 4 - Représentation Graphique d'un merge Git

Conflits

Un conflit naît quand deux commits modifient le même code de manière différente. Les modifications ne sont pas d'accord entre elles, et rentrent en conflit quand Git les compare. Il est alors impossible de pusher un commit qui rentre en conflit avec un autre avant de modifier le code adéquatement. Cela intervient notamment quand plusieurs développeurs travaillent sur la même branche. Si l'un envoie un commit modifiant une certaine ligne du code sur la branche, un autre développeur ayant modifié la même ligne sans avoir mis à jour les informations de sa branche va recevoir un conflit au moment d'envoyer son commit.

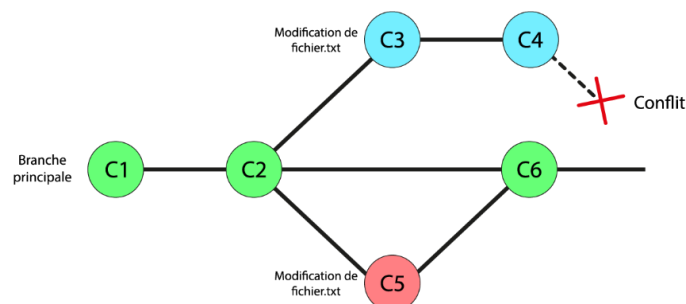


Figure 5 - Représentation Graphique d'un conflit Git

C'est un cas qui apparaît notamment au moment de merge une branche à une autre. Si d'autres merges ont eu lieu avant, cela signifie que de nouveaux commits sont présents sur la branche recevant le

merge. Si les mêmes lignes de code ont été modifiées, cela entrainera un conflit. Pour le résoudre, le développeur devra d'abord merger la branche principale vers sa branche et modifier le code afin que tout fonctionne.

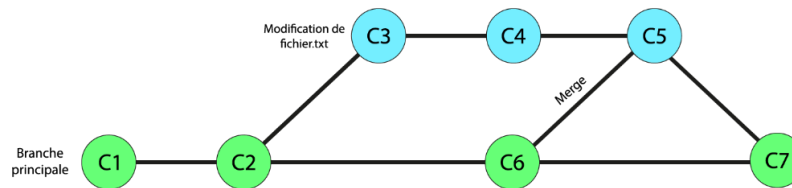


Figure 6 - Résolution d'un conflit par merge

b. Organisation et utilisations des branches

Les dépôts Git sont organisés autour des branches : comment les créer, les gérer, les détruire... Afin de faciliter la collaboration entre développeurs. Aujourd'hui, la plupart des dépôts suivent une de ces deux méthodes :

- *Git Flow*, la méthode la plus commune aujourd'hui, adaptée aux grandes équipes.
- *Trunk based development*, la plus recommandée, notamment dans le cadre du DevOps.

Chacune de ces deux méthodes présente ses avantages, ses inconvénients. Mais la majorité des problèmes rencontrés sont généralement dus à la grande rigidité au changement de l'être humain. De ce fait, certaines équipes seront plus efficaces avec l'une de ces méthodes que l'autre.

Git Flow

Git Flow se base sur deux branches principales : *main* et *develop*. La première représente l'historique officiel des versions de l'application, et est donc mise à jour uniquement lors d'une livraison. La branche *develop*, quant à elle, va intégrer toutes les nouvelles fonctionnalités, peu importe l'état de la livraison. C'est à partir de la branche *develop* que la livraison sera faite, en mergant l'ensemble de ses commits sur *main*.

C'est aussi à partir de la branche *develop* que toutes les branches de fonctionnalités seront créées. Ces branches « *features* » (leur nom change en fonction de ce qu'elles apportent) sont créées par les développeurs pour rajouter, modifier ou supprimer du code. Avec une durée de vie plutôt longue (nombreux commits, nombreux fichiers modifiés), elles sont mergées avec *develop* une fois la fonctionnalité totalement terminée.

Ces branches représentent le cœur de Git Flow. Il est néanmoins possible d'étendre son utilisation en mettant en place les branches *release* et *hotfix*. Les branches *releases* sont créées à partir de *develop* afin de préparer la livraison (documentation, corrections de bugs...) avant de merger les modifications sur *main*. La branche *hotfix* est censée intervenir rarement, étant donné qu'elle permet de régler les bugs d'une version déjà livrée.

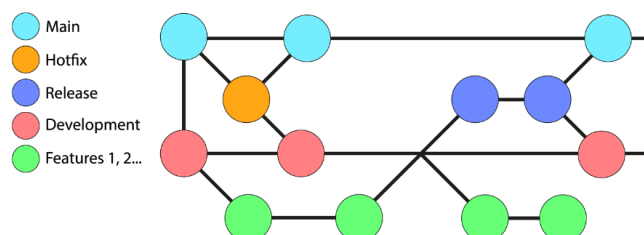


Figure 7 - Représentation schématique d'un arbre Git Flow

Git Flow est une stratégie particulièrement efficace sur un projet organisé selon des livraisons, et est suffisamment malléable pour permettre de gérer des bugs rapidement. Cependant, il nécessite de nombreuses branches afin de pouvoir fonctionner correctement. De plus, les branches à longue durée de vie ont tendance à créer plus de *merge conflicts*, ce qui peut vite se ressentir dans de grandes équipes, nécessitant de nombreuses branches.

Trunk based Development

Le trunk based development, ou développement basé sur le tronc, n'est basé que sur une seule branche : *main*. Celle-ci fait office de dépôt de code pour les nouvelles fonctionnalités, mais aussi de versions de code. Les nouvelles fonctionnalités sont écrites dans des branches à durée de vie courtes, qui sont mergées le plus vite possible. C'est un fonctionnement simple qui suit la logique du DevOps, notamment celui de l'intégration continue (méthode qui sert à automatiser l'intégration des changements de code).

Même si un bug est introduit dans *main*, l'avantage d'avoir des branches à durée de vie courte permet de très rapidement le corriger. De plus, les conflits sont moins nombreux, ce qui simplifie énormément la qualité de vie des développeurs, qui n'ont plus à se tirer autant les cheveux pour respecter l'ensemble des ajouts.

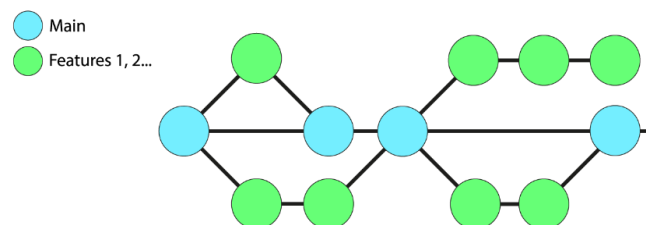


Figure 8 - Représentation schématique d'un arbre Trunk based

ArmoniK utilise le développement basé sur le tronc. De plus, comme nous le verrons plus tard, le projet crée régulièrement des *tags*, des versions chiffrées de l'application, suivant la même logique de livraison que le Git Flow. C'est une alternative qui permet de remplacer la branche des *releases* et *main* dans Git Flow.

c. Conventions de commits

Dans un projet, un commit est censé représenter une unité de modification, d'ajout ou de suppression logique du code. Dans le cas où un commit rassemble plusieurs choses qui ne le sont pas logiquement, il peut être complexe de retrouver une modification en particulier. C'est pour cela que les commits sont séparés et nommés logiquement, selon des conventions.

Les commits sont représentés par des tags dans l'historique de Git. Ces tags sont uniques : un rassemblement de caractères et de nombres qui sont difficiles à différencier par l'être humain. Ils sont donc accompagnés d'un message, pouvant faire plusieurs lignes, composé d'un en-tête, d'un contenu, et d'un pied.

Nous pouvons séparer les commits en plusieurs types : les plus basiques étant *feat*, *fix*, *chore*, *docs*, *test*... Le type doit être le premier élément à apparaître dans l'en-tête du commit. Il doit ensuite être suivi d'une courte description permettant à tout développeur de comprendre ce qui a été réalisé dans celui-ci.

Respecter les conventions de commits n'est pas toujours facile : ce n'est pas un réflexe de fractionner des modifications selon ce qu'elles font. Dans le meilleur des mondes, tous les développeurs arrivent à

respecter les conventions, et pourront facilement trouver le moment et la ligne où auront été introduit un bug. Mais en réalité, les processus mis en place pour préserver le bon fonctionnement du code aujourd'hui réduisent le nombre de bugs. Par conséquent, il est très rare qu'un développeur aille chercher un commit en particulier pour corriger un bug.

2. GitHub, garde forestier

a. Toute une forêt

GitHub est un espace de repositories Git. Accessible au public, n'importe qui peut profiter de ses services pour héberger ses dépôts de manière publique ou privée. Il propose des outils pour compléter et étendre les services de Git : *issues*, *pull request*, *actions*... Et bien d'autres encore. Il s'oppose notamment à GitLab, un concurrent moins grand public mais qui a pour avantage d'être déployable sur serveurs, permettant un contrôle total des dépôts Git hébergés.

b. Issues

Les développeurs, qu'ils soient seuls ou dans une équipe, ont besoin de savoir et retenir les tâches à faire : nouvelles fonctionnalités, refactoring de code, corrections de bug... Les issues sont là pour répondre à ce besoin. N'importe quel développeur peut en créer au moindre bug trouvé, à la moindre idée, pour proposer une solution et en discuter avec les membres du projet, avant de commencer à l'implémenter.

Ces issues peuvent ensuite être associées à un ou plusieurs *tags*, qui permettent aux développeurs d'avoir une compréhension rapide du sujet abordé par l'issue : *bug*, *feature*, *must have*...

Les issues peuvent ensuite être placées dans un *projet*, qui est une implémentation de la méthode Kanban directement intégrée à GitHub. En fonction de l'avancement de l'issue, celle-ci voyagera dans le tableau.

c. Pull Request

La sécurité et la qualité du code sont extrêmement importantes dans le monde professionnel. On ne peut pas laisser n'importe qui, même quelqu'un de confiance, merger sa branche dans la branche principale, au risque d'y introduire un bug.

Très généralement, les branches principales d'un dépôt sont protégées, c'est-à-dire qu'à moins d'être administrateur, on ne peut ni push, ni merger dessus sans l'autorisation préalable d'un développeur de l'application. Les Pull Request permettent de demander cette autorisation. Les développeurs eux-mêmes doivent ouvrir des Pull Requests lorsqu'il doivent merger leurs branches.

Lorsqu'un développeur accède à une Pull Request, il va pouvoir relire l'ensemble du code rajouté, supprimé, ou modifié. Dans le cas où un code ne lui plaît pas, il peut laisser un commentaire sur la Pull Request à l'intention du développeur de la branche, qui devra discuter et modifier le code en question avant de relancer la relecture. Une fois que la Pull Request est approuvée, elle peut être mergée avec la branche principale.

Une Pull Request est généralement liée à une issue, ce qui permet d'avoir un suivi facile des modifications faites sur le dépôt Git. Lorsque la branche est mergée, la Pull Request et l'issue qui lui est liée sont automatiquement fermées.

d. GitHub Actions

Les GitHub actions trouvent leur utilité dans le cadre de la philosophie DevOps. Elles permettent d'implémenter les processus de *Continuous Integration* et de *Continuous Delivery* dans un projet. Nous reviendrons sur ces notions plus tard dans ce rapport, car elles retrouvent tout leur intérêt dans le cadre de ce mémoire.

Les GitHub actions permettent d'automatiser l'exécution de scripts à chaque push ou Pull Request sur une branche. Utilisant un format de fichier *yaml*, très facile à lire pour un être humain, elles sont extrêmement simples à comprendre. De plus, des utilisateurs de GitHub ont la possibilité de créer des « actions », des outils facilitant l'automatisation d'un processus dans les GitHub actions.

Il s'agit d'un outil communautaire permettant à chacun de facilement automatiser des processus informatiques. ArmoniK l'utilise dans ce cadre : mais en plus, les GitHub actions vont automatiser le processus de création d'une nouvelle version d'ArmoniK. Le développeur souhaitant créer une nouvelle version, un nouveau *tag*, n'a qu'à indiquer les informations nécessaires de la version : un numéro, et une courte description des changements apportés.

e. Tags

Les tags sont un outil très simple à comprendre : ils permettent aux développeurs de créer une branche « release » où une version sera un des commits de cette branche. A la place, GitHub permet de créer un « tag », qui aura le rôle d'une nouvelle version. Chaque tag devient ainsi une référence, et de ce fait, est plus facile à trouver qu'un commit sur une branche. Les tags permettent aussi aux développeurs de ne pas s'embêter à maintenir une branche *release*, *master*, ou *develop* supplémentaire.

3. Angular

Avant de rentrer en détail dans Angular, il faut comprendre ce qu'est qu'un framework Javascript. Auparavant, les sites et applications web étaient développées en HTML, CSS et Javascript, le tout séparé. Cela avait de gros inconvénients, notamment au niveau de la lisibilité, de l'intégration et de la répétition de code : à chaque nouvelle page HTML, des éléments qui étaient déjà présents sur une autre page devaient être recopiés.

Les frameworks Javascript sont donc arrivés avec une idée principale : réduire toute cette redondance de code. Grâce à eux, les éléments HTML peuvent être séparés les uns des autres sous la forme de composants. Ceux-ci vont posséder leur propre code HTML, Javascript voire CSS. Ainsi, les éléments souvent répétés d'une page ne le sont plus. Aujourd'hui, les frameworks Javascripts sont légion et proposent tous des centaines de fonctionnalités et outils supplémentaires. Les explorer ici n'a aucun intérêt, alors concentrons-nous sur celui utilisé pour développer l'Admin GUI : Angular.

a. Fonctionnement

Angular divise son code entre différents types d'outils. Tout d'abord, nous avons les composants, qui sont les éléments visuels avec lesquels l'utilisateur de l'application va interagir. Ensuite, nous retrouvons les services, qui vont permettre de gérer les données entre les différentes parties de l'application. Ce sont les services qui gèrent le fonctionnement logique du code. Ensuite, nous avons les directives : elles permettent d'ajouter des comportements à un composant dans notre application. Enfin, nous avons les pipes : ils permettent de gérer plus facilement l'affichage d'une donnée dans le code HTML.

Une telle structure de code fait énormément penser au paradigme de l'Orienté Objet en programmation informatique. Les composants et services vont représenter nos différents objets et

dépendances que l'on va injecter tout au long du code. Les autres Framework JavaScript sont eux aussi basés sur ce paradigme, très commun et simple à comprendre.

b. TypeScript (TS)

La particularité d'Angular est qu'il impose le TypeScript plutôt que le Javascript comme langage de programmation. Le TypeScript est une simple surcouche du Javascript, permettant de rajouter un outil aussi pratique que nécessaire en développement informatique : le typage des données.

JavaScript est souvent vu comme une aberration parmi les développeurs. Et pour cause : c'est un langage qui a été créé en quelques jours, et qui comporte depuis toujours de nombreux problèmes d'implémentation. Cependant, il a été adopté par le monde d'internet, rendant son utilisation presque obligatoire lorsque l'on développe une application web.

TypeScript vient donc réparer un des plus gros défauts de Javascript : le manque de typage. Sans cet outil, un développeur peut être très vite perdu dans sa compréhension d'un code. S'il ne sait pas quel format de données il traite, cela va lui faire perdre énormément de temps, et il risque d'introduire des bugs plus facilement. Par exemple, il pourrait décider de lire une clé de données qui n'existe pas, javascript n'y verrait aucun problème, mais cela aurait pour effet d'arrêter le bon fonctionnement du code. TypeScript, quant à lui, peut prévenir le développeur si une donnée n'est pas utilisée correctement. Ainsi donc, TypeScript est un outil permettant d'augmenter drastiquement la qualité et la stabilité d'un code Javascript.

c. RxJS

En informatique, et plus particulièrement en programmation orientée objet, les développeurs font souvent face aux mêmes problèmes. Ils ont alors créé des patrons de conception, des modèles à suivre pour résoudre ces problèmes le plus simplement possible. Parmi ces patrons, nous pouvons retrouver l'*Observateur*. Angular intègre la bibliothèque RxJS, qui est une implémentation de ce patron de conception.

Le patron Observateur permet de limiter le couplage entre plusieurs modules ou composants Angular. En ayant une panoplie d'*Observateurs* observant un *Sujet* (ou un *Observateur* observant un unique *Observable*), la modification de l'état du sujet permet à tous les observateurs de se mettre à jour en suivant une méthode indiquée par le développeur.

En Angular, le patron Observateur est utilisé pour mettre à jour les données de l'application en fonction d'événements reçus : interactions de l'utilisateur, récupération de nouvelles données... De ce fait, il est énormément utilisé dans de nombreux projets Angular.

```
import {Subject, Observable, map} from 'rxjs';

let myNumber = 0;
const subject = new Subject<number>();

const observer1: Observable<number> = subject.pipe(map(n => n*2));
observer1.subscribe(n => myNumber = n);

subject.next(3);
console.log(myNymber); // 6
```

Figure 9 - Exemple d'implémentation du patron Observateur

Dans cet exemple, nous créons la variable *myNumber*, qui abritera la valeur finale de nos données. La première chose que nous allons faire, c'est créer un *Observable* avec la fonction *pipe*. Cette dernière permet d'interagir et de transformer la donnée envoyée par le sujet avant de la gérer. Ensuite, nous

créons une *subscription*, qui permet de récupérer la valeur de l'observable et de l'intégrer au reste du code : ici, en écrasant la valeur de *myNumber*. Enfin, nous mettons à jour l'état du sujet pour tester notre code. En affichant la valeur de *myNumber*, nous n'avons pas 0, mais bel et bien 6.

En faisant voyager les *Sujets* à travers les services de notre application, nous pouvons donc très simplement mettre à jour les données des tables ArmoniK en temps réel.

Dans la version la plus récente d'Angular, les *signaux* ont été introduits. Il s'agit d'une alternative au patron observable, qui lui est très similaire ; plus simple à utiliser, elle est cependant moins flexible. De ce fait, les *signaux* ne sont pas faits pour gérer des données externes à l'application. Ils vont être préférés pour modifier certains états de l'application.

d. Material

En plus des Framework Javascripts, les développeurs web ont créés les Frameworks CSS. Ce sont des outils basés sur les Frameworks Javascript créant une bibliothèque de composants basiques et avec un visuel pré-travaillé. Grâce à ces outils, les développeurs peuvent se permettre de passer un minimum de temps sur la partie visuel de l'application, et donc de concentrer leurs efforts sur la logique.

Angular Material est une bibliothèque de composants Angular développée par Google. Il a notamment été retenu par rapport à d'autres frameworks grâce à cette dernière information : Angular est aussi développé par google, il y a donc une très haute compatibilité entre les deux outils (notamment sur l'utilisation de la bibliothèque RxJS). De plus, nous pouvons être sûr qu'il y aura des évolutions fréquentes de l'outil.

4. gRPC

ArmoniK est une application gérant énormément de données. Elle doit donc être capable de les faire circuler le plus rapidement possible d'un point à un autre. Pour se faire, elle se base en partie sur le gRPC. Cette technologie va donc avoir le rôle d'une api REST, mais ne fonctionne pas de la même manière. Le gRPC est une technologie de *Remote Procedure Call* (Appel de procédure à distance, RPC) fonctionnant sur le protocole http2. Le RPC est une méthode permettant à un client de communiquer avec un serveur de manière simple, sans rentrer dans les détails de son réseau. Le protocole http2 permet de compresser les données transmises afin de limiter la taille de la requête faite, améliorant donc sa vitesse. gRPC va aussi traiter les requêtes en les transformant en *binaryCode*, pour limiter au mieux leur taille et augmenter l'efficacité du programme, jusqu'à 10x plus par rapport à une api REST.

gRPC va permettre de lier facilement un client et un serveur à l'aide de fichiers *proto*. Ces fichiers permettent de définir des schémas d'interfaces et de fonctions, qui seront ensuite traduits dans un langage informatique au choix des développeurs. De nombreux langages sont disponibles : C#, Java, Python, Javascript...

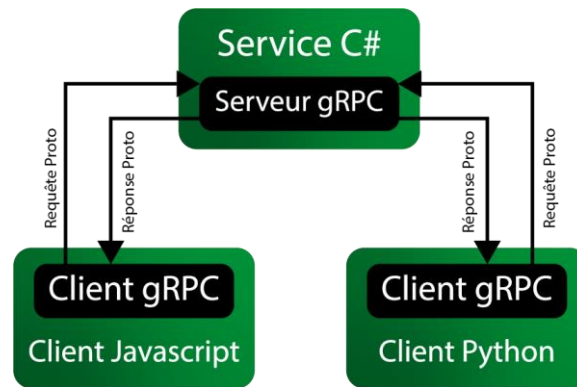


Figure 10 - Exemple d'une communication gRPC

Les schémas peuvent être traduits pour créer des Serveurs ou des Clients. Les Clients vont pouvoir appeler les fonctions générées pour accéder aux données exposées par les Serveurs qui vont implémenter le retour de ces fonctions. Comme les fonctions Serveurs/Clients sont créées par le gRPC, il n'y aura pas de problème d'incompatibilité entre les deux, ce qui simplifie le travail des développeurs. Un autre avantage certain est que les fonctions générées se comportent comme du code local pour l'application l'utilisant, ce qui permet donc d'avoir un typage automatique des données retournées.

5. Docker

L'environnement sur lequel tourne un code est un problème très fréquent en informatique. D'une machine à une autre, un code peut ne peut pas fonctionner de la même manière, et les développeurs ne peuvent pas simplement adapter le code pour une machine ou une autre. Le fameux « pourtant ça fonctionne sur ma machine » prend tout son sens, et était même à la base de l'incessant combat entre développeurs et opérateurs, qui devaient déployer le code.

La première solution pour pallier ce problème fut de créer les Machines Virtuelles (aka Virtual Machines, ou VM) : recréer numériquement une machine, en lui allouant de la mémoire vive, des processeurs, du stockage... Les configurations de VM pouvant être partagées, il était facile d'en créer une sur laquelle un programme informatique aller fonctionner parfaitement. Mais les VM ne font que déplacer le problème : elles sont elles aussi dépendantes de l'environnement sur lequel elles sont déployées, mais en plus, nécessitent énormément de ressources. Il faut imaginer qu'une VM représente un ordinateur virtuel totalement simulé, y compris les processus inutiles au bon fonctionnement du programme. Elles sont donc souvent très coûteuses pour leur utilité.

Fort heureusement, une alternative, moins coûteuse et inspirée des Machines Virtuelles est née : la containerisation des programmes informatiques. Introduit par l'entreprise Docker en 2013, elle repose sur un principe simple : limiter les processus d'une VM au strict nécessaire pour qu'un programme informatique fonctionne.

Docker se base sur un système d'*images*, qui sont des schémas utilisés par le *Docker Engine* pour créer des *containers*. Les images vont définir quelles commandes utiliser, quels processus utiliser pour que le logiciel fonctionne. Elles peuvent aussi se baser sur d'autres images qui ont déjà certains processus définis.

Le Docker Engine est ce qui crée et gère les *containers*, les instances d'une ou de plusieurs applications. Il n'est pas dépendant de l'environnement sur lequel il est installé, ce qu'il fait qu'il est utilisable absolument partout.

Grâce à ce fonctionnement, Docker a donc plusieurs avantages : les images sont très légères et extrêmement portables, les containers demandent peu de ressources en comparaison des Machines Virtuelles, et il est déployable partout.

ArmoniK est une application complexe, énormément découplée, et chaque partie de cette application est containerisée à l'aide de Docker, la GUI ne faisant pas exception. Ces images sont stockées publiquement sur *Docker Hub*, le dépôt officiel d'images Docker.

6. Kubernetes

Afin de gérer facilement un grand nombre d'images, ArmoniK utilise Kubernetes, un orchestrateur de containers. Il va gérer leurs cycles de vies en fonction des instructions des opérateurs de l'application. Un container ayant subi une erreur fatale et s'étant arrêté pourra être redémarré automatiquement, sans que les opérateurs aient besoin de faire quoi que ce soit. C'est un outil supplémentaire qui réduit de manière significative le temps de travail à fournir pour un opérateur gérant des containers. Il permet aussi de déployer facilement plus ou moins de containers en fonction de la demande utilisateur : un site utilisé par 10 utilisateurs n'aura pas les mêmes besoins qu'un autre qui l'est utilisé par 1000.

Kubernetes se repose sur le principe de *clusters*, qui sont composés d'une ou plusieurs *Nodes*, et d'un *Control Plane*. Ce dernier va permettre de contrôler tout le cluster. Les *Nodes* sont des machines virtuelles ou physiques, dépendamment du cluster, et vont gérer des *Pods*. Ce sont des groupes d'un ou plusieurs containers qui partagent un stockage et des ressources communes.

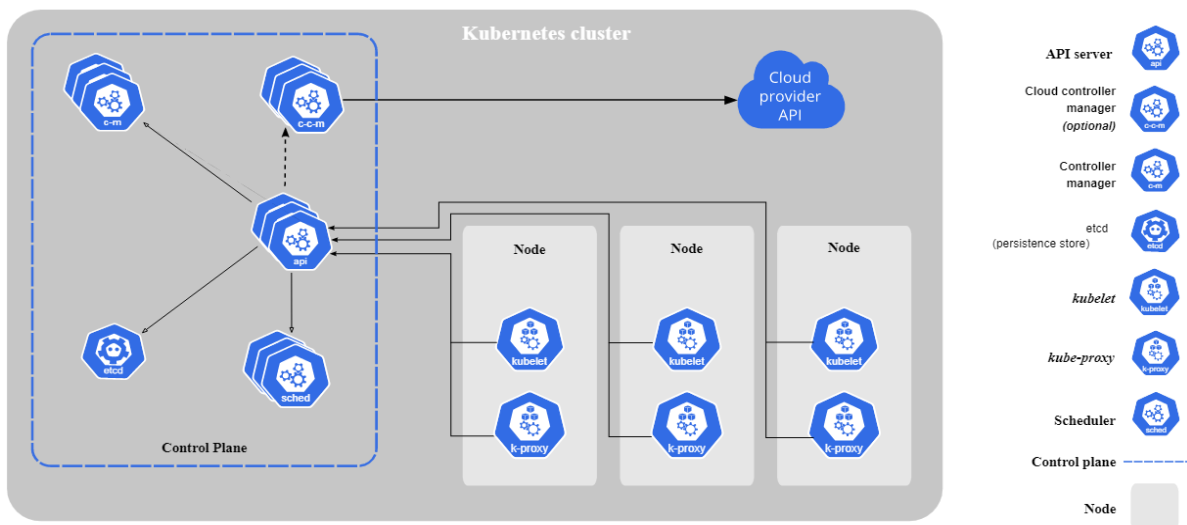


Figure 11 - Schéma d'un cluster Kubernetes

Par défaut, Kubernetes va essayer de partager les charges (les demandes en ressource physique de chaque pod) le plus possible entre les différentes Nodes disponibles afin de s'assurer que les ressources de chaque Node ne soient pas surchargées. Pour cela, il va utiliser son *Scheduler*, qui aura reçu les spécifications des ressources, les besoins des opérateurs, et d'autres facteurs internes au cluster.

De plus, Kubernetes permet de créer des *namespaces*. Ceux-ci permettent d'isoler un groupe de container d'un autre en les plaçant dans différents *namespaces*. Ils peuvent utiliser les mêmes ressources, mais seront séparés les uns des autres pour simplifier leur lisibilité par les opérateurs et leur rôle dans un même cluster Kubernetes.

7. Terraform

Terraform est un outil d'Infrastructure as Code (IaC) permettant de créer, changer et versionner des infrastructures de code en local ou auprès d'un Cloud Provider (Amazon Web Services, Google Cloud Platform, Azure...) ou sur des serveurs. C'est le squelette d'ArmoniK : c'est grâce à une configuration définie par les opérateurs de l'application que les différents conteneurs de l'application sont configurés et communiquent entre eux.

Il se base sur des fichiers HCL, très simples à lire, même sans compétences préalables. Malgré leur simplicité apparente, ils sont capables de rentrer en détail dans la configuration de l'infrastructure : pour chaque application, l'on peut définir exactement les ressources dont elle aura besoin, son adresse, ses ports de connexion...

Terraform s'adapte facilement aux changements d'infrastructure, grâce à un système de plans de modifications. Lorsque l'infrastructure est modifiée, Terraform va simplement chercher les différences avec la précédente version et appliquer uniquement le minimum de changements nécessaire pour faire fonctionner cette nouvelle version.

Terraform est totalement compatible avec Kubernetes. Il est capable de déployer, provisionner et manager un *cluster*, ses *Pods* et *namespaces*. Et ce, tout en gardant la panoplie d'options déjà disponibles auparavant. Il est donc simple de configurer et modifier un *cluster* Kubernetes, en lui indiquant les ressources physiques que doivent avoir chaque *pod*.

V – Principes de développements

1. Dette technique, qu'est-elle donc ?

Ce mémoire fait souvent référence à la dette technique. Elle est même au centre de sa problématique, étant donné que c'est elle que les principes de développements sont censés rembourser. Il est donc nécessaire de comprendre ce qu'est cette dette pour comprendre l'impact de ces principes.

Lorsqu'une date de livraison est trop proche, un projet peut accepter de s'endetter techniquement pour respecter cette date. Cela signifie que le ou les développeurs en charge de la fonctionnalité vont mettre de côté la qualité ou/et la performance d'un morceau de code pour qu'il soit fonctionnel à temps. C'est quelque chose qui arrive fréquemment dans de gros projets, mais le plus important avec la dette technique, c'est le fait de la rembourser. Rembourser une dette technique signifie retravailler la partie du code sacrifiée.

Du code de mauvaise qualité peut avoir plusieurs impacts. Le premier peut être au niveau de la performance, comme indiqué plus tôt. Les autres sont moins directs sur l'expérience utilisateur. La plupart des autres problèmes vont impacter les développeurs. Le code sera difficile à lire et à comprendre, et il sera donc difficile à modifier ou utiliser dans l'avenir. Il ralentira le processus de développement de certaines fonctionnalités, ce qui aura sans doute pour effet de rajouter encore plus de dette technique.

Dans le monde de l'informatique, il existe de nombreux projets « *legacy* » dont le code ne consiste presque plus qu'en dette technique, et sont un calvaire pour les développeurs travaillant dessus ou pour les utilisateurs qui rencontrent fréquemment des ralentissements ou bugs. Ces projets sont à un stade où la dette n'est plus remboursable que d'une seule manière : totalement réécrire le code, le recommencer de zéro.

Il est donc extrêmement important pour une équipe de développeurs de rembourser une dette technique le plus rapidement possible. Cela leur permet d'éviter le risque d'avoir du code toujours plus pourrissant dans un projet toujours plus complexe à comprendre.

2. Mot d'ordre : simplicité

Commençons par les principes les plus basiques des développeurs : KISS et YAGNI. Le premier, « *Keep It Simple, Stupid !* » (Garde le simple, stupide !), est assez clair quant à son utilité : il vise à garder un système informatique, un design, un code, le plus simple possible. La simplicité est l'ami de l'humain : c'est elle qui permet à chacun de comprendre rapidement ce qu'ils regardent. En informatique, c'est tout aussi vrai : un code simple est, généralement, un très bon code.

Pour continuer, explorons rapidement le principe YAGNI, « *You ain't gonna need it* » (Tu ne vas pas en avoir besoin). C'est un principe qui étend en quelque sorte le KISS, étant donné qu'il indique aux développeurs de ne pas garder ce qui est inutile, de ne pas polluer le projet avec du code qui ne servira à rien.

3. Redondance de code, comment l'éviter

Le premier problème auquel les développeurs sont confrontés est celui de la redondance de code. Il survient surtout pour les nouveaux développeurs de l'application ou dans de grands projets avec une énorme équipe. Dans le premier cas, les membres les plus aguerris de l'équipe peuvent guider le nouveau venu vers le code qu'il a répété, supprimant rapidement le problème. Le second cas est plus complexe. Dans des dizaines de milliers de lignes de code, il est difficile de connaître chaque ligne, chaque fonction, ainsi que son but, peu importe que le code soit structuré correctement ou non. Il faut encore une fois faire confiance aux *reviews* de code lors des *Pull Request* pour guider correctement le développeur qui a répété son code.

Ne pas répéter son code est en fait un des principes clé du développement informatique, quel que soit son paradigme. Ce principe, nommé DRY (Don't Repeat Yourself, ou Ne Vous Répétez Pas), indique qu'une connaissance dans un système informatique ne doit avoir qu'une unique représentation faisant autorité.

Même si tous les développeurs font de leur mieux pour l'appliquer, il n'est parfois pas respectable du tout, notamment dans le monde du Web. Et ce, même si les différents composants d'un framework aident à ne pas répéter certaines parties d'un code. Étant donné que l'utilisateur peut avoir accès à plusieurs pages très similaires, les quelques différences entre les deux codes peuvent mener à quelques redondances de code. Certains outils, comme l'héritage de classes, permettent de limiter la quantité de code répété, mais certaines fonctions devront forcément être répétées.

Exemple d'implémentation

Un des meilleurs exemples de redondance de code dans l'interface administrateur vient sans aucun doute des pages d'affichage des tables de données. Nous pouvons prendre la page des *partitions* et celle des *résultats*, qui initialisent toutes les deux des données de la même manière. Mais comme les données traitées sont différentes, elle doivent l'initialiser de manière différentes. Elles utilisent l'héritage de classes pour accéder aux mêmes fonctions, mais leur initialisation est différente. Le code est similaire, mais s'agit de redondance de code atteignable dans le cadre de l'interface administrateur.

```
// Partitions
export class IndexComponent extends TableHandler
implements OnInit, AfterViewInit, OnDestroy {

  readonly filtersService = inject(PartitionsFiltersService);
  readonly indexService = inject(PartitionsIndexService);

  tableType: TableType = 'Partitions';

  ngOnInit() {
    this.initTableEnvironment();
  }

  ngAfterViewInit() {
    this.mergeSubscriptions();
  }

  ngOnDestroy() {
    this.unsubscribe();
  }
}

// Results
export class IndexComponent extends TableHandler
implements OnInit, AfterViewInit, OnDestroy {

  readonly filtersService = inject(ResultsFiltersService);
  readonly indexService = inject(ResultsIndexService);

  tableType: TableType = 'Results';

  ngOnInit() {
    this.initTableEnvironment();
  }

  ngAfterViewInit() {
    this.mergeSubscriptions();
  }

  ngOnDestroy() {
    this.unsubscribe();
  }
}
```

Les fonctions *ngOnInit*, *ngAfterViewInit* et *ngOnDestroy* sont des fonctions appelées par Angular à différents moments de vie d'un composant. Il n'est donc pas recommandé de les implémenter directement dans une abstraction, car, dans le cas où une classe doit initialiser une étape de plus, elle pourrait avoir besoin d'une de ces fonctions. C'est pour cette raison que le code est ici répété.

4. Nomenclature et structure

En informatique, la clarté est un mot d'ordre : nous l'avons vu avec le principe KISS, les choses doivent rester simples, claires. Et pour de la clarté, il faut pouvoir nommer correctement les différentes variables d'un code, et l'organiser correctement. Et pour que les noms et l'organisation soient le plus clair possible, il est nécessaire que les développeurs posent des normes au début du projet.

Ces normes peuvent être déjà préétablies par les langages de programmation utilisés : l'organisation et la nomenclature des projets python et java changent drastiquement. L'idée générale est de respecter les conventions d'un langage, de manière à fluidifier la transition d'un projet à un autre pour un développeur. En retrouvant les mêmes normes partout, tout le monde s'y retrouve.

a. Notions de nomenclature

Pour la nomenclature, les développeurs doivent fournir un effort supplémentaire. Les conventions de nommage ne concernent presque que le type de syntaxe utilisé (camelCase, KebabCase, snake_case...) et la taille maximale que peut atteindre une ligne. Ce n'est cependant pas suffisant pour garder en clarté. Un code informatique se décompose en variables, fonctions, classes. Elles ont toutes un rôle, elles vont toutes stocker, gérer, modifier, supprimer des données. Il est donc nécessaire qu'elles soient nommées en fonction de leurs rôles.

Un exemple fréquent de mauvaise nomenclature en informatique est le fait de nommer ses variables avec des chiffres et des lettres. Ainsi, des codes se retrouvent pollués de variables nommées A1, b_2, c3_1, qui ne veulent rien dire. Ce sont des noms très peu évocateurs, qui ne signifient rien pour les développeurs lisant ce code. La personne ayant introduit ce code et ces noms aura vite oublié leur signification et elle aussi sera perdue au moment de relire ce code. Un code avec un nommage peu clair est comparable à de la dette technique, et doit être modifié le plus vite possible. Il est recommandé de nommer une variable, une fonction, selon son rôle. Ce nommage doit être fait dans la langue du langage informatique, donc généralement en anglais. Cela permet d'éviter les confusions à base de français. Pour donner un exemple concret, une fonction dont le rôle est de vérifier si un chargement est en cours devrait se nommer « *isLoading* ». Le « *is* » indique que la fonction renvoie l'état d'un élément. Le « *loading* » peut être soit le nom d'une variable, soit le nom de ce qui est vérifié, ici un chargement. Ici, le nom de la fonction respecte la syntaxe imposée, le « camelCase ».

Exemple de nomenclature avec le service « IconsService ».

```
export class IconsService {  
  readonly icons: Record<string, string> = {  
    'refresh': 'refresh',  
    'auto-refresh': 'autorenew',  
    // ...  
  };  
  
  getIcon(name: string) {  
    // Implémentation  
  }  
  
  getAllIcons() {  
    // Implémentation  
  }  
}
```

Nous avons ici l'exemple d'un service très simple de l'interface administrateur d'ArmoniK. Nous voyons dans que le code représente la classe « IconsService », qui indique clairement son rôle. A l'intérieur, nous pouvons retrouver la variable « icons », qui va stocker toutes les icônes de l'application. Nous voyons ensuite la fonction « getIcon », au singulier. Nous pouvons comprendre rapidement que son rôle est de retourner une icône à l'aide de son nom. Pour finir, nous pouvons jeter un coup d'œil à la fonction « getAllIcons », qui a pour but de retourner toutes les icônes disponibles de l'application.

Du premier coup d'œil, n'importe quel développeur peut comprendre le rôle de ce service. Il est tout de même important de noter qu'il est difficile de rester aussi clair sur des dizaines de milliers de lignes de code, notamment sur les fonctions les plus complexes.

b. Structure d'un code Angular

Comme relevé précédemment, un projet informatique a besoin d'organiser logiquement ses fichiers. Les différents morceaux de code qui le composent ont tous un rôle différent : ils ne sont pas tous du même type. Mêmes les composants les plus similaires ne sont pas tous appelés de la même manière dans toute l'application. Il est donc important de les séparer selon leurs rôles et leurs types. Nous avons déjà pu en apercevoir deux : les composants et les services. Ils sont bien plus nombreux, il est donc nécessaire de les organiser correctement. Angular propose ainsi la structure suivante :



Toute la partie logique du code va se retrouver dans la partie « app » du projet. A l'intérieur, nous allons pouvoir retrouver nos différents types de code : composants, services, directives...

Les *directives* sont des outils permettant de simplifier l'utilisation du code HTML, tandis que les *pipes* traitent les données qui leur sont envoyées pour l'afficher dans le code HTML.

Les *Tokens* sont des clés permettant de partager une référence à un même service partout dans le code de l'application. Le dossier *type* permet de définir les formats des différentes données de l'application pour TypeScript. Nous avons ensuite un dossier par page. Comme elles sont uniques, elles sont considérées comme des « modules », des ensembles de code suffisamment complexe pour être isolées du reste.

Les fichiers stockés dans le dossier *assets* sont les images, les vidéos, les sons utilisés au travers du code. Le dossier *locales* regroupe les différentes traductions de l'application.

Aussitôt que du code HTML doit être affiché et du CSS modifié, Angular propose deux manières différentes de l'organiser. La première consiste à tout mettre dans le même fichier TS. Au moment de définir un composant, un développeur a la possibilité de lui donner un attribut « template » correspondant au code HTML lié au composant. De la même manière, l'attribut « style » lui est proposé. Mais avec ArmoniK, c'est la deuxième méthode que nous avons préférée : séparer le HTML, le CSS et le TypeScript dans 3 fichiers différents. Cela permet de séparer totalement l'affichage de la logique, et de profiter à 100% des nombreux outils vérifiant la syntaxe d'un code.

Nous avons déjà eu un peu l'occasion de voir comment était organisé l'intérieur des fichiers TypeScript de Angular. Globalement, pour les services et les composants que l'on va définir, tout cela se ressemblera : Il y a généralement une classe, qui englobe des méthodes (ou fonctions) et des attributs (variables). Il s'agit du code logique de l'application, c'est celui qui doit être le plus organisé. Le HTML, quant à lui, ne change pas vraiment, il est toujours organisé de la même manière. Cependant, Angular permet de rajouter des éléments fort pratiques pour éviter la redondance de code HTML : les conditions « @if » et « @else », les « @switch » pour gérer les différents états d'une variable, ainsi que les « @for » pour répéter automatiquement une partie du code.

```
@if (actions.length > 1) {
  <button mat-icon-button [matMenuTriggerFor]="menu" aria-label="Actions" i18n-aria-label>
    <mat-icon fontIcon="more_vert" />
  </button>
} @else if (actions[0].condition && actions[0].condition(element) || !actions[0].condition) {
  <button mat-button (click)="actions[0].action$.next(element)">
    <mat-icon>{{ actions[0].icon }}</mat-icon>
    <p>{{ actions[0].label }}</p>
  </button>
}

<mat-menu #menu="matMenu">
  @for (action of actions; track action.label) {
    @if (action.condition && action.condition(element) || !action.condition) {
      <button mat-menu-item (click)="action.action$.next(element)">
        <mat-icon [fontIcon]="getIcon(action.icon)"/>
        <p>{{ action.label }}</p>
      </button>
    }
  }
</mat-menu>
```

Figure 12 - Exemple d'un code HTML Angular de l'interface administrateur

5. Tout pour un code SOLID

a. Contextualisation

SOLID est un regroupement de plusieurs principes de développements relativement complexes. Il est aujourd'hui recommandé de les appliquer autant que possible. C'est pour cette raison que j'ai cherché à les appliquer dans le code d'ArmoniK et qu'ils ont une grande importance dans ce mémoire. Les principes SOLID touchent à des notions de programmation orienté objet qu'il est nécessaire de connaître pour pouvoir les appréhender pleinement. Une courte définition de ses principes est nécessaire.

Classes

En programmation, les classes permettent de rassembler les méthodes et les propriétés d'un objet. Une classe vélo possède des roues, des pédales, un guidon, et peut avancer. Quand on instancie une classe, on crée un objet de cette classe. C'est une des bases de la programmation orienté objet. Nous avons déjà pu en apercevoir quelques-unes dans ce mémoire.

Hérité

Une classe peut hériter d'une autre classe, c'est-à-dire qu'elle peut récupérer tous les membres et fonctions de la classe « parent ». C'est une pratique courante qui sert à éviter le code redondant tout en créant du nouveau code. Par exemple, nous pourrions faire en sorte que le *solex* hérite du *vélo*. En TypeScript, le mot clé permettant à une classe d'hériter d'une autre classe est « *extends* », nous avons pu voir un exemple dans la partie *redondance de code*.

Abstraction

Les classes dites « abstraites » sont des classes qui ne peuvent être instanciées. Elles sont utilisées comme base pour créer des classes concrètes qui vont « l'implémenter ». Par exemple, *véhicule* est une classe abstraite implémentée par *vélo*, *voiture*, *bateau*... Une classe ne peut

implémenter qu'une seule abstraction à la fois. Dans les langages de programmations, elles sont très souvent référencées par le mot-clé « *abstract* ».

Interfaces

Une interface est un ensemble de signatures de méthodes d'une classe. En d'autres termes, une interface va définir le nom, l'entrée et la sortie de ses fonctions sans les implémenter. C'est un outil qui permet de créer des normes tout en laissant de la flexibilité aux développeurs pour implémenter les fonctions. Une classe peut implémenter plusieurs interfaces. Elles possèdent généralement leur propre mot-clé, « *interface* ».

b. Principes SOLID

Maintenant que les bases de la programmation orienté objet sont posées, concentrons-nous sur les principes SOLID. Au nombre de 5, ils sont aujourd'hui considérés comme des lignes de conduite à suivre impérativement pour un développeur.

Single Responsibility

Le principe de responsabilité unique note qu'une classe, une fonction ou un objet doit n'avoir qu'un seul rôle. Par exemple, un programme visant à additionner deux nombres aura une fonction pour l'addition et une autre pour afficher le résultat. Cette séparation de la logique permet d'isoler les changements dans un code, et donc réduit les risques de bugs introduits.

Open-Closed

Ce principe dicte surtout aux développeurs de faire attention à avoir un code **Ouvert** à l'extension, mais **Fermé** à la modification. Reprenons le programme d'addition du point précédent. L'on décide maintenant d'y ajouter une soustraction. Si rajouter cette fonctionnalité nécessite de modifier le code déjà présent, c'est qu'il ne respecte pas ce principe ouvert-fermé. Pour faire simple, ce principe aide à faciliter l'implémentation de nouvelles fonctionnalités, avec un code n'ayant (en principe) jamais besoin d'être modifié.

Listkov Substitution Principle

Le Principe de Substitution de ListKov indique qu'une classe de type T doit pouvoir être remplacée par une classe de type G sous-type de T de telle manière à ce que le comportement du programme ne soit pas modifié. En suivant ce principe, le développeur peut utiliser une sous-classe dans un programme sans que cela n'interfère en rien sur son code, permettant une meilleure interchangeabilité du code. Prenons l'exemple d'une voiture conduite par un automobiliste. Une Ferrari est un sous-type de voiture. Il faut que l'automobiliste puisse changer de voiture sans qu'il ait besoin de s'adapter à ce changement. La Ferrari doit être utilisable exactement comme la classe voiture, même si son fonctionnement interne diffère.

Interface Segregation

Une classe ne devrait pas dépendre d'interfaces qu'elle n'implémente pas totalement. En d'autres termes, il est préférable qu'une classe dépende de plusieurs petites interfaces plutôt que d'une grande interface, où plusieurs fonctions seront inutiles. Cela réduit donc énormément le nombre de couplages inutiles ; et donc, augmente la maintenabilité d'un code. Par exemple, un automobiliste n'a pas besoin des mêmes voyants et indicateurs que ceux d'un avion ; un altimètre ne lui serait d'aucune utilité, il n'a donc pas besoin de l'avoir.

Dependency Inversion

Une classe doit dépendre d'abstraction et non pas d'implémentation. Plus une classe dépendra d'implémentations, plus elle sera difficile à modifier. En dépendant d'abstractions, elle sera donc beaucoup plus modulable. On peut voir ça comme une manière d'éviter un *spaghetti code*¹, en faisant en sorte que les classes implémentées ne dépendent pas les unes des autres.

c. Intérêt

On peut se demander si appliquer SOLID a un réel intérêt en Web, particulièrement en *Front-End*. Aujourd'hui, la plupart des applications web sont souvent totalement réécrites ; pour implémenter une technologie plus efficace, un changement de direction visuel et d'expérience... L'interface Administrateur d'ArmoniK n'a pas échappé à cette règle, ayant déjà été partiellement réécrite une fois et totalement une autre fois. A chaque fois, des changements drastiques ont été observés.

Cependant, si une application est totalement réécrite, on peut se demander pourquoi elle doit l'être. Peut-être ne respectait-elle pas les bons principes de développements ; dans le cas où elle le ferait, la partie logique de l'application n'aurait pas besoin d'être réécrite, peu importe la technologie utilisée, si le langage de programmation est le même. Certains services fonctionneront de la même manière, les données reçues ne changent pas non plus ; ce sont des bases de code qui peuvent être reprises si elles sont de qualité industrielle.

Les principes SOLID visent à rendre un code le plus maintenable et extensible possible. Un code qui respecte ces principes peut être réutilisé sans aucune modification particulière, et c'est là qu'appliquer ces principes dans le monde du web est intéressant.

De plus, un développeur se forçant à appliquer ces principes les comprendra de mieux en mieux, et saura les implémenter de plus en plus facilement dans un code. Il est donc dans son intérêt personnel de respecter le plus tôt possible ces principes, afin de non seulement avoir un code solide, mais aussi de solides compétences de développement.

Dans le cadre d'ArmoniK, en plus de ces différents avantages, appliquer les principes SOLID a eu d'énormes impacts sur la qualité du code logique. Auparavant, les différentes pages possédaient chacune une copie du même code. Seules quelques pages possédaient des fonctionnalités supplémentaires, notamment les pages d'affichage des tâches et sessions. Mais en faisant abstraction du code implémenté dans ces différentes pages, la redondance de code de ces composants fut divisée par 5, aussi bien dans les fichiers HTML que TypeScript. De plus, le code est plus flexible ; il est beaucoup plus simple d'apporter une modification à la configuration d'une colonne de l'interface, mais aussi plus facile de rajouter du code ou une fonction unique à une page. Cela fut cependant au détriment de la redondance du code « non logique », celui qui permet de configurer l'interface administrateur. Mais c'est une redondance nécessaire, qui ne peut qu'exister. La transformation du code par les principes SOLID aura eu pour effet d'augmenter la taille de cette redondance. Cela n'aura cependant pas d'impacts sur la clarté du code, qui, lui, est énormément simplifié.

d. Exemples d'implémentation

Single Responsibility

Dans l'exemple suivant, nous avons une simplification de la classe abstraite *TableHandler*. Nous nous intéressons à la fonction *initTableEnvironment*. Cette fonction sera appelée par les classes enfant de *TableHandler* lors de leur initialisation. Cette fonction a pour but d'initialiser une table d'affichage, et va le faire en appelant des fonctions dont c'est le rôle. Les fonctions *initColumns*, *initFilters* et

initOptions ont chacun un rôle bien défini, en d'autres termes : une responsabilité unique. Un développeur jetant un coup d'œil à cette fonction aura plus de facilité à la lire que si l'entièreté du code était écrite au même endroit.

```
abstract class TableHandler {
  initTableEnvironment() {
    this.initColumns();
    this.initFilters();
    this.initOptions();
    this.intervalValue = this.indexService.restoreIntervalValue();
    this.shareableUrl = this.shareUrlService.generateShareableUrl(this.options, this.filters);
  }

  protected initColumns() {
    // ...
  }

  protected initFilters() {
    // ...
  }

  protected initOptions() {
    // ...
  }
}
```

Open-Closed

De plus, ce code respecte aussi le principe *Open/Closed*. Le prochain extrait de code représente l'implémentation d'un enfant de *TableHandler*. Nous pouvons voir que la fonction *initTableEnvironment* est appelée lors de l'initialisation du composant dans la fonction *ngOnInit*. Dans ce cas précis, si un développeur en a besoin, il n'a pas besoin de modifier *initTableEnvironment* pour rajouter une étape d'initialisation supplémentaire : il n'a qu'à la rajouter dans *ngOnInit*. La fonction n'a pas besoin d'être modifiée, mais son utilisation est ouverte à l'extension.

```
class IndexComponent extends TableHandler implements OnInit {
  ngOnInit() {
    this.initTableEnvironment();
  }
}
```

Listkov Substitution

Pour le troisième principe SOLID, entrons dans la classe abstraite *AbstractTableComponent*. Auparavant, nous regardions la classe qui initialisait l'environnement dans lequel évoluait une table de données. Ici, nous sommes dans la classe abstraite définissant le fonctionnement logique des tables de données.

Ce qui nous intéresse ici, c'est l'appel de sous-types. Nous avons la classe *GrpcTableService*, qui est l'abstraction parent des services gRPC, que nous avons défini plus tôt dans ce mémoire. Il en existe cinq, un par type de données exposées par l'API (applications, partitions, sessions, tâches, résultats). C'est à partir de ces services que les tables récupèrent les données qu'elles affichent. Ces services implémentent tous la fonction *list\$* dans ce cadre.

Nous utilisons alors l'abstraction de ces classes comme un type pour référencer ces enfants. Comme ils possèdent tous la fonction *list\$*, remplacer *GrpcTableService* par un de ses enfants n'a aucune incidence sur le fonctionnement du code.

```
export abstract class AbstractTableComponent<O extends IndexListOptions> {  
  abstract readonly grpcService: GrpcTableService<O>;  
  
  list$(options: O, filters: FiltersOr): Observable<GrpcResponse> {  
    return this.grpcService.list$(options, filters);  
  }  
}
```

Notons aussi la présence des mots clés *extends* de la première ligne de cet extrait. Cette fois-ci, ce ne sont pas des classes qui sont étendus, mais des types TypeScript. En les utilisant de la même manière que les classes, il est possible d'étendre ce principe à la Substitution de Listkov et d'avoir un très bon typage dans le code. Ici, ils sont aussi partagés à *GrpcTableService* pour que chaque table ait le client gRPC adapté.

Interface Segregation

De nombreuses interfaces sont injectées dans les services du code, plutôt que d'avoir à chaque fois une gigantesque interface. Certains services gRPC ne font pas que lister des données ; ils peuvent aussi interagir avec elles. Afin de respecter la nomenclature et la logique du code, les services ayant besoins de ces fonctions vont implémenter les différentes interfaces correspondant à leurs besoins.

```
export interface GrpcGetInterface<R extends GetResponse> {  
  readonly grpcClient: GrpcClient;  
  get$(id: string): Observable<R>;  
}  
  
export interface GrpcCancelInterface<R extends CancelResponse> {  
  readonly grpcClient: GrpcClient;  
  cancel$(id: string): Observable<R>;  
}  
  
export interface GrpcCancelManyInterface<R extends CancelResponse> {  
  readonly grpcClient: GrpcClient;  
  cancel$(ids: string[]): Observable<R>;  
}
```

Ces interfaces sont simples : plutôt que d'avoir une interface ayant toutes les responsabilités, nous priorisons plusieurs petites interfaces. De cette manière, la classe implémentant ces interfaces se limitera à ce qu'elles déclarent, et il n'y aura pas de déclaration de fonction inutile.

```
class TasksGrpcService extends GrpcTableService implements GrpcGetInterface, GrpcCancelManyInterface {  
  get$(taskId: string): Observable<GetTaskResponse> {  
    // ...  
  }  
  
  cancel$(taskIds: string[]): Observable<CancelTasksResponse> {  
    // ...  
  }  
}
```

Ici, la classe *TasksGrpcService* n'implémente que les interfaces *GrpcGetInterface* et *GrpcCancelManyInterface*, et n'a qu'à implémenter les fonctions associées. Si nous avions une unique interface qui implémentait *GrpcGetInterface*, *GrpcCancelManyInterface*, *GrpcCancelInterface*, la classe *TasksGrpcService* aurait dû implémenter une méthode dont elle n'a pas besoin. Respecter le principe de la ségrégation d'interfaces nous a donc permis d'éviter du code inutile.

Dependency Injection

Le but de ce principe est d'avoir des classes qui dépendent uniquement des abstractions. Nous l'avons vu à plusieurs reprises, nous avons quelques abstractions dans le code. Cependant, uniquement des classes abstraites sont les enfants d'autres classes abstraites.

Il n’y a donc pas d’exemple parlant de ce principe dans l’application. Et il n’y en a pas besoin : la logique se découplant entre des services et des composants très similaires les uns aux autres permettent d’avoir une unique abstraction pour chacun d’entre eux. En d’autres termes : le principe a été appliqué implicitement dans les exemples que nous avons pu citer.

6. DevOps

a. Introduction au DevOps

Plus tôt dans ce rapport, j’ai pu faire référence au DevOps, cette philosophie de travail informatique qui a grandement pris en popularité ces dernières années. Elle est née du besoin de fusionner les équipes de développement informatique, les *Devs* (pour *Développeurs*), avec l’équipe d’opérations informatiques, les *Ops* (pour *Opérateurs*). Ces derniers avaient pour rôle de mettre en production un code, mais la distance les séparant des développeurs rendait le processus généralement instable, complexe. Mettre en production une application était devenu un calvaire, les développeurs faisaient en sorte que le code fonctionne sur leur machine tandis que les opérateurs souhaitaient qu’il fonctionne pour les leurs.

Le DevOps est donc la fusion entre ces deux équipes : les développeurs sont maintenant ceux qui déploient leur application vers la production et s’assurent de son bon fonctionnement. Pour ce faire, l’équipe DevOps va s’occuper de l’ensemble du cycle de vie de l’application : intégration de code, de tests, déploiements vers la production... Ce cycle est répété à chaque nouvelle itération de l’application, à chaque nouvelle version.

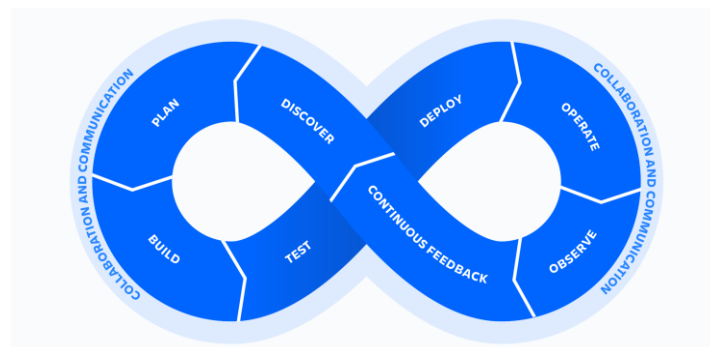


Figure 13 - Cycle de vie DevOps

Généralement, l’équipe DevOps va mettre en place des outils pour automatiser le plus de processus possible : c’est là que viennent les pratiques du *Continuous Integration* (CI) et son extension logique, *Continuous Deployment* (ou *Delivery*, pour CD).

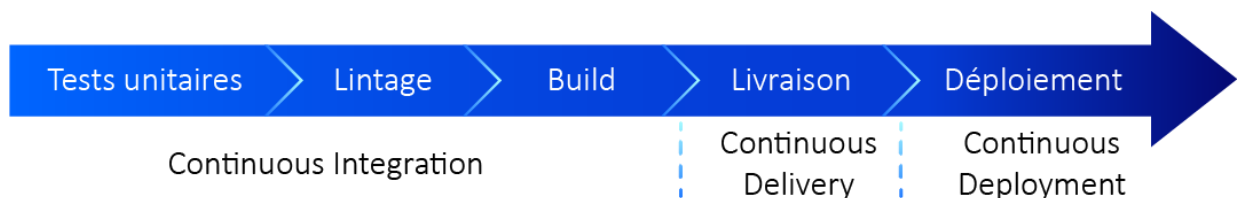


Figure 14 - Continuous Integration, Delivery, Deployment

Le rôle du *Continuous Integration* est de s’assurer qu’à chaque ajout de code, une série de tests soient lancés automatiquement. Ces tests ont pour but de vérifier le fonctionnement et la qualité de ce code. Sur les projets ArmoniK, les CI sont lancées à chaque *push* sur une *branche*, dès lors qu’une *Pull Request* est ouverte. Elle est aussi lancée à chaque fois qu’une *branche* est *merge* sur la branche principale.

Le *Continuous Delivery* est sa suite logique. C'est un processus qui est lancé lorsque l'équipe souhaite sortir une nouvelle version de l'application, d'où le terme *delivery* (signifiant livraison). Elle peut reprendre les différentes étapes automatisées de la CI, et va simplement rajouter le processus de livraison de la version. Dans le cadre d'ArmoniK, tous les processus de CI vont se terminer avec un processus de CD, permettant d'avoir une nouvelle version par branche, disponible et testable par les développeurs à tout moment.

Le *Continuous Deployment* est une extension du *Continuous Delivery*. En plus d'effectuer la livraison de la nouvelle version de l'application, elle va automatiser son déploiement vers la production. Cela consiste notamment à tester sa disponibilité dans un environnement simulant la production, puis de remplacer le déploiement de la précédente version avec la nouvelle.

Certaines applications n'ont pas besoin de *Continuous Deployment*, c'est notamment le cas d'ArmoniK. Etant donné qu'il s'agit d'une application déployée sur cloud par ses clients, et que les collaborateurs d'Aneo ne sont pas les utilisateurs finaux de l'application, la déployer sur ses propres serveurs n'a aucun intérêt. C'est pour cela que l'automatisation des processus s'arrête à la livraison de nouvelles versions.

b. Avantages du DevOps

Il est difficile pour moi de comparer une approche DevOps à une autre dans le cadre d'un projet informatique ; je n'ai jamais été confronté qu'à cette méthode. Cependant, j'ai pu remarquer les avantages qu'elle propose, des avantages dont je ne saurais me séparer.

Tout d'abord, le DevOps permet de mettre en place des normes : les processus de tests, de livraison, de qualité, de déploiement sont les mêmes, et sont basés sur des environnements fixes. De ce fait, le DevOps empêche les mauvaises surprises dues à certains facteurs inconnus. De plus, il est globalement très simple : rapide à comprendre, on peut retrouver facilement un problème souligné par un des pipelines.

Le DevOps permet de s'assurer de la qualité d'une application : en automatisant les processus de vérifications de performances et de disponibilité d'une application, on retire l'erreur humaine de l'équation. Aucun biais n'est là ne peut gêner les mesures de qualité.

Le DevOps permet aussi de simplifier l'assimilation des bonnes pratiques. La plupart des CI vont vérifier la qualité d'un code, le nombre de tests, sa syntaxe... Lorsqu'une erreur sur ces points est faite, la CI ne passe pas, le développeur doit alors corriger ses problèmes. A force de le faire, il va pouvoir corriger ses biais et fournir un code qualitatif de plus en plus fréquemment.

C'est aussi une pratique en constante évolution : les techniques utilisées doivent toujours être plus rapides et précises, il faut donc souvent essayer de nouvelles méthodes d'automatisation des processus, pour encore plus d'efficacité. L'implémentation doit constamment grandir, pour être toujours plus à jour et efficace.

De plus, l'approche DevOps colle parfaitement avec l'utilisation du *Trunk-based development* de git. Pour rappel, l'approche consiste à minimiser les modifications faites sur un code à chaque *Pull Request*, afin de minimiser les erreurs introduites. Avec de rapides pipelines de *Continuous Integration*, qui vérifient le bon fonctionnement et la qualité du code, le nombre d'erreurs introduites dans le code à chaque modification est minime.

c. Kubernetes et Terraform

Si l'interface administrateur n'utilise pas ses technologies directement. Son déploiement et sa disponibilité dans un environnement ArmoniK en dépend. Ils sont très intéressants dans le cadre de l'approche DevOps. Kubernetes et Terraform sont deux outils qui trouvent particulièrement leur place dans cette philosophie. Ils ne lui sont pas directement liés, mais leur utilisation entre dans son cadre. Kubernetes permet de gérer facilement des containers docker, des applications. C'est un outil simple à utiliser, et donc, à automatiser.

Terraform est un outil d'Infrastructure as Code. Il permet de définir rapidement l'infrastructure d'une application en local, sur serveurs, ou auprès des cloud providers. Extrêmement facile à mettre en place et très élastique, il est aussi automatisable. En l'alliant à Kubernetes, il est possible de définir une infrastructure complexe de containers Docker dans un cluster, sur n'importe quelle machine. Il est aussi simple de déployer cette infrastructure. Ce sont deux outils facilitant énormément l'étape de déploiement d'un ensemble d'application conteneurisées.

ArmoniK utilise ses deux outils de cette manière : cependant, rien n'est déployé. Des tests sont cependant réalisés pour assurer le bon fonctionnement de l'application. Mais l'équipe de développement, contrairement aux clients, n'a pas besoin d'avoir une version déployée de l'application. Ils n'ont pas besoin de l'utiliser, la déployer serait donc contre-productif. Mais utiliser Kubernetes et Terraform permet de simplifier énormément le processus pour les utilisateurs d'ArmoniK. On peut imaginer l'impact positif qu'a une application aussi complexe déployable en une unique ligne de commande.

d. Evolutions

Les équipes DevOps commencent déjà à évoluer vers une méthodologie intégrant les processus de vérification de sécurité du code : le *DevSecOps*. Les données informatiques prennent de plus en plus de place aujourd'hui. Il est nécessaire qu'elles soient le plus protégées possible. Plutôt que de tester la sécurité d'une application une fois déployée, les processus de vérifications sont lancés le plus tôt possible.

Il est impossible de s'assurer qu'un code ne contient aucune faille de sécurité. Aujourd'hui, la plupart des projets informatiques se reposent sur de nombreuses bibliothèques de code, toutes potentiellement vulnérables. L'intérêt de rajouter un aspect sécurité au DevOps est de minimiser le plus possible les vulnérabilités.

7. Tests Unitaires

Les principes de développement sont bien pratiques, mais ne suffisent pas pour s'assurer du bon fonctionnement d'un code. Nous l'avons vu, ils visent à simplifier le code. Ce dernier est plus facile à comprendre, les développeurs ont donc moins de chance d'introduire des bugs. Un bug peut être défini comme un comportement indésirable d'un code informatique. Si l'on s'attend à avoir en sortie d'une fonction 10 et que l'on retrouve 11, c'est un bug.

Les tests unitaires sont des outils permettant de vérifier le comportement d'un code. Généralement, l'unité logique la plus commune d'un code, les fonctions (ou méthodes), est composée d'instructions et de conditions, qui, ensemble, permettent de définir un comportement. Les tests unitaires vont vérifier que ces éléments fonctionnent de manière logique.

a. Exemple d'implémentation

Prenons un simple exemple de test unitaire de l'application, à nouveau avec le service *IconsService* :

```
export class IconsService {
  readonly icons: Record<string, string> = {
    // values...
  }

  getIcon(name: string | null | undefined): string {
    if (name) {
      const icon = this.icons[name];
      if (icon) {
        return icon;
      }
    }
    return this.icons['default'];
  }
}
```

Nous nous intéressons ici à la fonction *getIcon*, qui prend en paramètre un nom et qui retourne l'icône associée.

La première instruction s'assure que le nom d'icône donné à la fonction est bel et bien défini. Si c'est le cas, la fonction utilise ce nom pour stocker dans la constante *icon* l'icône associée. Si cette icône existe, alors elle est retournée dans par la fonction.

Dans le cas où aucune condition n'est respectée, l'icône par défaut est renvoyée.

Maintenant que nous connaissons le fonctionnement de cette fonction, regardons comment elle est testée :

```
describe('IconsService', () => {
  const service = new IconsService();

  describe('getIcon', () => {
    it('Should return the correct icon (3 tests)', () => {
      expect(service.getIcon('refresh')).toEqual('refresh');
    });

    it('Should return the default icon if the icon is invalid', () => {
      expect(service.getIcon('invalid-icon')).toEqual('radio_button_unchecked');
    });

    it('should return the default icon if the icon is undefined', () => {
      expect(service.getIcon(undefined)).toEqual('radio_button_unchecked');
    });
  });
});
```

Attardons-nous d'abord sur les mots clés du code : *describe* et *it*. Le premier se traduit littéralement en « décrire ». Il permet d'indiquer l'objet que nous allons tester. Il est écrit deux fois : la première, pour décrire la classe *IconsService*, et ensuite la méthode *getIcon*. Ensuite, nous avons *it*. Son rôle est de vérifier un unique comportement de l'objet testé. Dans les fonctions *it*, nous décrivons le comportement testé (*it should* est traduisible en *devrait*), puis nous implémentons le test.

Dans ce cas particulier, nous avons deux tests pour la méthode *getIcon* : un par retour. Dans le premier cas, nous essayons de récupérer la valeur d'une icône avec son nom associé, dans la fonction *expect*. Ensuite, nous vérifions le résultat retourné avec le résultat attendu, à l'aide de la fonction *toEqual*. Si nous recevons la bonne valeur, le comportement de la fonction est assuré et nous saurons alors que nous n'avons pas de bug dans ce cas.

Le deuxième test consiste à passer un nom d'icône invalide. Nous savons que quoiqu'il arrive, la fonction *getIcon* doit retourner la valeur de l'icône « *default* » si elle ne trouve pas l'icône, ce que nous faisons avec la fonction *toEqual*. Si nous ne recevons pas cette valeur, nous saurons qu'il y a un bug dans le code.

Le troisième test suit la même logique, mais cette fois-ci, nous lui passons un nom d'icône indéfini. Encore une fois, nous sommes censés retrouver la valeur de l'icône par défaut, et c'est ce que nous testons avec la fonction *toEqual*.

b. Mesures

Il est possible de mesurer la couverture en tests d'un code. Ces mesures permettent de s'assurer que le comportement des méthodes du code sont bien testés et qu'il n'y a pas de bugs. Pour mesurer efficacement un code, il faut se concentrer sur quatre éléments : le nombre de lignes couvertes, le nombre de branches couvertes, le nombre de fonctions couvertes ainsi que le nombre d'instructions exécutées.

En testant une fonction, on peut parcourir l'ensemble des lignes qui la compose. C'est une statistique basique mais très peu intéressante, tout comme pour le nombre de fonctions couvertes. Le nombre d'instructions couvertes l'est déjà plus : on peut rapidement savoir quelle instruction n'est pas exécutée lorsqu'on teste un fichier. Le nombre de branches couvertes est une métrique très importante : c'est généralement des branches, des conditions, que viennent les erreurs comportementales. Si une condition est mal écrite, elle pourra être vite détectée par un test couvrant correctement toutes les branches d'une fonction.

Cependant, les mesures de tests ne sont pas totalement preuve de qualité. Même si certains tests couvrent à 100% les lignes, branches et instructions d'une fonction, ces tests peuvent être biaisés. Lorsqu'un test est difficile à écrire, il est parfois plus simple d'en écrire un réussissant sur un cas particulier. Dans ce cas-là, le test passe, les mesures de couverture le décrivent comme très performant, mais en réalité, il ne teste absolument pas le comportement de la fonction.

Il faut donc faire très attention aux mesures. Elles sont un bon indicateur de la qualité du code, mais ne doivent pas servir de référence absolue. Il est nécessaire que dessous, les tests soient correctement écrits.

c. Guidé par les Tests

Le Test Driven Development (TDD) est une méthode de développement basée sur les tests qui est fortement recommandée par de nombreux développeurs. Son principe est simple : il suffit de commencer à écrire les tests d'une fonction avant d'écrire la fonction. De cette manière, un développeur peut fixer le comportement d'une fonction avant même de l'écrire. Cela permet d'augmenter la qualité des tests du code.

Cependant, écrire les tests avant le code nécessite un immense effort organisationnel et mental pour un développeur. S'il doit finir une fonctionnalité rapidement, il peut ne pas avoir le temps de commencer par écrire les tests. Une mauvaise habitude s'installe alors, et c'est pour cela que la majorité des développeurs n'appliquent pas, ou peu, le TDD.

C'est d'ailleurs le cas de l'interface Administrateur d'ArmoniK, qui, à mon arrivée, n'avait aucun test écrit. Une grande partie de mon travail, en plus de réécrire le code, était d'intégrer des tests unitaires dans l'application. Pour cette unique raison, je n'ai pas pris l'habitude d'appliquer cette méthodologie de travail dans mon quotidien.

d. Tests et DevOps

Etant donné qu'à mon arrivée, les tests unitaires n'étaient pas implémentés, le pipeline CI de l'interface administrateur n'était pas complète. La validation du code se concentrait uniquement sur les problèmes de syntaxe, et le fait que l'application fonctionne. Mais en aucun cas un bug ne pouvait être détecté.

L'implémentation de tests unitaires a permis de détecter et de corriger de nombreux bugs tout au long de l'année. Et ce, même si certains tests ne sont pas très qualitatifs. Ces derniers restent un cas rare qui n'ont pas eu d'effet dramatique sur le fonctionnement de l'interface.

Maintenant que tous les services et composants de l'application sont testés, la pipeline CI peut facilement détecter si un bug a été introduit. Les Pull Requests dans ce cas sont alors bloquées, jusqu'à ce que l'ensemble des tests unitaires passent. De cette manière, nous nous assurons toujours plus d'avoir une très bonne qualité de code.

8. Documentation Développeur

Plusieurs fois dans ce mémoire, j'ai pu faire référence au fait qu'un long code est difficile à connaître par cœur. Et ce, même si le code est simple, correctement nommé et structuré. Il est donc nécessaire d'écrire de la documentation développeur. Cela consiste à donner une description courte des paramètres d'entrée d'une fonction ou d'une classe, de son rôle, et de sa sortie si elle en a une. Des exemples d'utilisations peuvent aussi être fournis.

La documentation d'une fonction ou classe est généralement fournie avant son implémentation, comme nous pouvons le voir. La manière de l'écrire dépend d'un langage à l'autre, voire d'un outil à un autre. Il existe des outils capables de compiler la documentation développeur pour la rendre plus lisible, généralement au travers d'une application web.

Il est nécessaire que la documentation d'une fonction ou d'une classe soit le plus clair possible sur son utilisation ; si elle ne l'est pas, les développeurs peu familiers avec le code peuvent perdre beaucoup de temps à l'utiliser, voire introduire des bugs si les comportements ne sont pas bien testés.

La documentation développeur vise surtout à simplifier le travail des développeurs, et les autorise à ne pas devoir connaître l'ensemble du projet. C'est un outil très pratique, même s'il n'est pas nécessaire, et il peut toujours aider un développeur à résoudre un problème.

```
export class TasksByStatusService {
  readonly #key = 'tasks-by-status';

  // ...

  /**
   * Save colors corresponding to tasks statuses for applications, sessions or partitions
   * @param table a type TableTasksByStatus object
   * @param statuses array of TaskStatusColored objects
   */
  saveStatuses(table: TableTasksByStatus, statuses: TasksStatusesGroup[]): void {
    this.#storageService.setItem( `${table}-${this.#key}` , statuses);
  }
}
```

Dans cet exemple, nous pouvons voir l'exemple de documentation pour une fonction de la classe *TaskByStatusService*. Nous avons sur la première ligne une courte description de la fonction, suivie d'une description des paramètres (ici présentés par le tag *@param*). La plupart des outils actuels permettent d'accéder à la définition d'une fonction en survolant son nom, ce qui permet à un développeur de comprendre rapidement son rôle, en faisant abstraction de son code.

Malheureusement, l'interface administrateur ne possède de la documentation que pour quelques fonctions. En rajouter fait partie du travail que je vais devoir accomplir avant la fin de mon alternance, il s'agit même d'une des priorités actuelles.

9. Sécurité et Programmation

Les premières personnes concernées par la sécurité d'une application sont ceux qui l'utilisent : des entreprises, des particuliers, des familles... Une faille de sécurité au niveau d'un code a déjà eu des impacts considérables sur de nombreuses vies. Les données sont très importantes, et, comme précisé plus tôt, il faut les protéger.

Dans le cadre d'ArmoniK, aucune donnée n'est personnelle, si bien que pour le moment, aucune mesure de sécurité du code n'a été implémentée côté interface administrateur. Cependant, des utilisateurs essayant d'interagir avec des fonctions auxquelles ils n'ont pas accès verront apparaître un message d'erreur.

Cependant, les bibliothèques de code peuvent encore être la source des problèmes. Il est donc nécessaire de toujours faire attention à se documenter sur leurs failles de sécurité ou à utiliser des outils permettant d'être avertis de problèmes liés aux dépendances.

a. Critères de sécurité

En informatique, la sécurité se distingue en 4 principes fondamentaux. Une application doit toujours être hautement **confidentielle**, c'est-à-dire que les données ne doivent être accessibles qu'aux personnes autorisées. De plus, elle se doit d'être **authentique**, c'est-à-dire de fournir la preuve que la source d'une information ou donnée est viable. Elle doit aussi être **intègre**, donc s'assurer que les données enregistrées soient fiables et complètes. Enfin, elle doit être constamment et totalement **disponible**.

Les failles de sécurité d'une application vont donc être calculées en fonction de ces critères. Mais toutes les failles ne se valent pas : certaines sont faciles à utiliser, d'autres non. Certaines ont un impact mineur sur un des 4 critères, quand d'autres vont être dangereuses sur les 4.

En fonction de toutes ces critères, une note est attribuée à une faille de sécurité, allant de 0 à 10. La note d'une faille de sécurité permet de la ranger dans une catégorie. De 0.1 à 3.9, elles sont *faibles*. De 4.0 à 6.9, elles sont *moyennes*. De 7.0 à 8.9, elles sont *majeures*. Enfin, de 9.0 à 10, elles sont *critiques*.

Plus une faille appartient à une catégorie élevée, plus vite l'équipe de développeurs doit la régler. Les failles critiques doivent être réglées aussitôt qu'elles sont découvertes, tandis que les faibles et moyennes peuvent être ignorées. Si une faille concerne une bibliothèque utilisée, il suffit d'attendre une nouvelle version et de mettre à jour cette dernière. Cependant, cela peut être plus complexe pour des problèmes venant du code interne d'une application.

b. Outils de détection

Aucun outil n'est capable de détecter en temps réel une faille de sécurité. Certains, que nous verrons plus tard, peuvent aider à détecter les erreurs de code menant à des failles de sécurité. Cependant, il existe des outils de recensement des failles de sécurité. Ces outils utilisent des bases de données publiques, constamment alimentées par des entreprises ou particuliers, à la recherche de la moindre faille.

Certains logiciels vont être capables de détecter les dépendances de code d'une application et de remonter les failles de sécurité associées aux versions de chaque bibliothèque. Parmi eux, nous pouvons retrouver Snyk, que j'ai notamment eu l'occasion d'utiliser à l'école. Celui-ci n'est capable que de mesurer les bibliothèques importées dans le code, ainsi que leurs dépendances.

Docker Scout est un autre exemple d'outil de recensement de failles, mais cette fois-ci, spécialisé dans les images Docker. Généralement, ces dernières sont des surcouches d'autres images. Dans ce cas précis, une image peut être comparée à une bibliothèque de dépendance, étant donné qu'elle possède le minimum nécessaire pour faire fonctionner une application.

c. Cas pratiques

Plusieurs fois pendant mon alternance, j'ai dû mettre à jour d'anciennes images Docker de l'interface administrateur d'ArmoniK. Malgré leur âge (parfois vieilles de plus d'un an) elles étaient encore utilisées chez le client, et des failles majeures avaient été détectées.

Etant donné que les versions des images que nous utilisons sont mises à jour automatiquement par le pipeline de *Continuous Delivery*, nous n'avions théoriquement pas besoin de nous occuper des versions les plus récentes d'ArmoniK. Malgré tout, une faille de sécurité majeure fut détectée sur la version la plus récente de l'image de l'interface administrateur. Cette faille concernait l'image *Nginx*, qui avait pour but de rendre l'application accessible en dehors du container docker. J'ai donc dû chercher une autre version de l'image, où aucune faille de sécurité n'a été détectée.

TAG				
1.25.3-alpine3.18-perl				
Last pushed 4 months ago by doijanky				
Digest	OS/ARCH	Vulnerabilities		
29afbe24473d	linux/386	0	2	6

Figure 15 - Analyse Docker Scout d'une version de l'image Nginx

10. Outils

Certains outils très particuliers ont pu permettre d'appliquer de bonnes pratiques de développement dans ce projet.

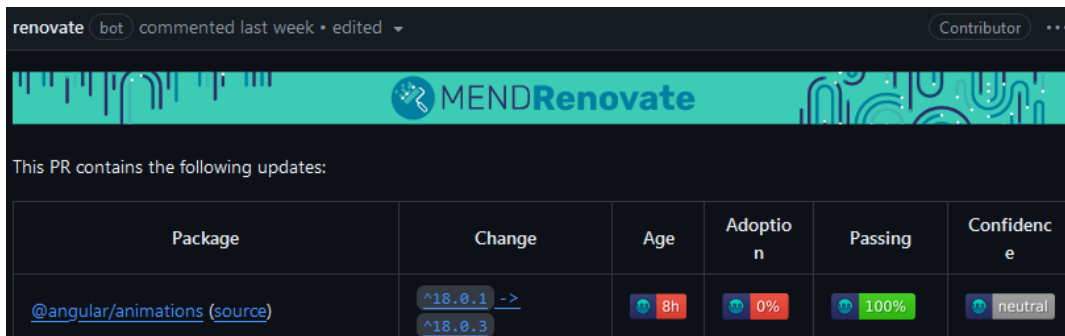
a. Eslint, qualité syntaxique

Eslint est, sans surprise, un linter. Il s'agit un outil qui va analyser la syntaxe du code et indiquer où sont les problèmes. Ils se base sur des règles définies par les développeurs pour ce projet, et permet aux développeurs d'avoir un retour en temps réel sur la qualité du code. Les linters se retrouvent dans la majorité des projets informatiques, et sont des outils très pratiques pour appliquer les bons principes de syntaxe. Eslint est un linter adapté au code JavaScript et TypeScript, et peut même analyser du code HTML.

b. Renovate, mise à jour des dépendances

Renovate va s'attaquer directement à la mise à jour automatique des bibliothèques de code d'un projet informatique. C'est un outil gratuit adapté à de nombreux langages et technologies. Renovate est un « bot » qui, à chaque mise à jour de la branche principale, va regarder les dernières versions des bibliothèques. En fonction du degré de modification (mineure ou majeure) de ces dernières, Renovate va créer une branche dans laquelle il met à jour les dépendances. Depuis cette branche, une Pull Request sera ouverte, et les développeurs pourront accepter ou non les changements apportés.

C'est un outil très pratique pour rester à jour. Nous l'avons vu, plus les bibliothèques de code sont âgées, plus elles sont vulnérables à des failles de sécurité. Cependant, cela ne signifie pas qu'il faut accepter chaque nouvelle version proposée par Renovate. Le bot affiche des métriques sur les versions qu'il propose : leur âge, leur pourcentage d'adoption, le pourcentage de mises à jour qui fonctionnent ainsi que le niveau de confiance sur cette version.



The screenshot shows a Pull Request interface for the Renovate bot. At the top, it says 'renovate bot commented last week • edited'. Below this is a header with the 'MEND Renovate' logo. The main content area states 'This PR contains the following updates:' and displays a table with the following data:

Package	Change	Age	Adoption	Passing	Confidence
@angular/animations (source)	^18.0.1 -> ^18.0.3	8h	0%	100%	neutral

Figure 16 - Pull Request de Renovate

Ces métriques permettent aux développeurs de choisir si les versions peuvent être acceptées. Il peut être dangereux d'avoir une version trop jeune, qui est peu adoptée. Ne pas avoir de données sur une version peut être dangereux : elle pourrait être porteuse d'une faille de sécurité critique. Il faut donc choisir astucieusement les versions que l'on souhaite mettre à jour.

c. SonarCloud, les bonnes pratiques

SonarCloud est un outil d'analyse de bonnes pratiques. Il possède une base de données sur les pratiques à ne pas avoir dans un code, dans de nombreux langages de programmation. A chaque commit sur une Pull Request, il va analyser le code et fournir une note aux problèmes qu'il trouve.

Ces problèmes se séparent en 3 domaines : sécurité, fiabilité et maintenabilité du code. Chaque problème trouvé dans le code se divise ensuite en 3 catégories : *faible*, *moyen* ou *majeur*. Une équipe

de développeur peut donc facilement retrouver quel type de problème est introduit, et sa dangerosité. Mais ce n'est pas tout, SonarCloud permet aussi d'avoir le nombre de ligne dupliquées dans le code introduit. De cette manière, un développeur peut savoir si son code est qualitatif ou non.

Ainsi, sur l'interface ArmoniK, de nombreux problèmes, notamment de maintenabilité de code, ont pu être observés et corrigés. Il fut très rare que d'autres types problèmes aient été observés, et jamais aucun n'a dépassé la catégorie moyenne. Aujourd'hui, tous les problèmes sont volontaires, ou liés à des tâches non encore réalisées.

VI – Vie d'un utilisateur

1. Expérience Utilisateur

Les principes de développement informatiques permettent d'avoir un code très qualitatif, simple et logique, mais, au détriment de tous les développeurs, ce n'est pas le code qui importe à un utilisateur. C'est ce qu'il a sous les yeux, et ce qu'il peut faire avec. Un code qualitatif ne vaut rien si une application est difficile à utiliser. Un développeur d'une application graphique doit donc faire autant attention à son code qu'à la qualité de l'expérience utilisateur.

L'interface graphique d'ArmoniK suit aussi cette règle. En parallèle de la réécriture de certaines parties du code, j'ai dû intégrer de nouvelles fonctionnalités à l'application, tout en retravaillant d'autres. Il fallait donc que je fasse expressément attention à ce que l'application soit facile d'utilisation. C'est un travail compliqué : un développeur connaît son application sur le bout des doigts, tout est facile à trouver pour lui. Il a donc un point de vue biaisé : il connaît toutes les manipulations, des plus simples au plus complexes ; il ne sait donc pas comment un nouvel utilisateur va appréhender son application. Là où le développeur prend 3 secondes à effectuer une manipulation, un utilisateur ne va peut-être pas trouver comment la faire. C'est ce qui définit l'expérience utilisateur : la facilité d'utilisation et d'appréhension d'une application.

Aussi appelée UX, il ne faut pas la confondre avec UI : l'Interface Utilisateur. Cette dernière n'est pas à propos de la facilité d'utilisation, mais plutôt de la partie visuelle, design de l'application. Elle est beaucoup moins importante pour un développeur, et peut être changée très rapidement et simplement. Néanmoins, elle ne doit pas être négligée pour autant : une application jolie à voir est toujours plus agréable à utiliser. L'UI ne doit simplement pas être une haute priorité.

Nous allons nous intéresser aux éléments clés de l'expérience utilisateur d'ArmoniK ; il s'agit d'un sujet large et complexe, qui a même donné naissance au métier de designer UX.

a. Interactions

Les utilisateurs de l'interface graphique d'ArmoniK vont souvent avoir besoin d'interagir avec les données. Que cela soit pour les trier, les filtrer, les afficher... Il est nécessaire qu'ils soient capables de le comprendre et de le réaliser facilement. Les interactions se font donc généralement par des boutons, qui permettent d'accéder à des espaces de configuration des tables de données. Nous pouvons citer par exemple le *Modify Column*, *Manage filters*, ou encore *Set up AutoRefresh*.

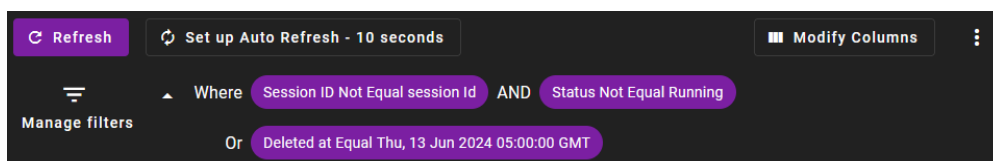


Figure 17 - Boutons supérieurs d'une table de données ArmoniK

Les boutons sont donc les éléments principaux des interactions : ils doivent attirer l'œil, être uniques, et clairs quant à leurs fonctions. Ici, les boutons les plus importants possèdent leur propre logo, ainsi qu'un nom définissant rapidement leur rôle. Les moins importants n'ont pas de nom ; cependant, en les survolant, une courte description de leur but doit apparaître. De cette manière, un utilisateur un tantinet curieux comprendra rapidement leur utilité, les maîtrisera rapidement, et évitera une guerre contre les boutons.

Certaines conséquences d'interactions ne sont pas visibles instantanément. C'est notamment le cas de celles qui ont un impact direct sur les données : suppression de sessions, annulations de tâches... Ce genre d'actions peut prendre quelques instants à se finaliser, et il faut donc que l'utilisateur soit mis au courant soit à la réussite, soit à l'échec de son action. Cela ne doit pas non plus gêner la navigation. Pour ce faire, une notification (de couleur verte ou rouge respectivement pour succès ou échec) lui est envoyée.



Figure 17 – Exemples de notifications d'ArmoniK

b. Navigation

Etant donné que ArmoniK expose plusieurs types de données, il est important de les séparer entre différentes pages. Pour un utilisateur souhaitant accéder à tel ou tel tableau de données, la navigation doit être simple et rapide. Si elle ne l'est pas, l'application sera frustrante à utiliser. Sur l'interface administrateur, elle est très basique : une barre sur la gauche de l'écran permet de passer d'une page à l'autre. De plus, l'optimisation du code a permis de fluidifier le chargement des données, si bien que la navigation est très rapide.

Mais une application doit proposer plus d'une manière de naviguer d'une page à une autre. Dans ArmoniK, des types de données sont reliées à d'autres. Les tâches sont toutes reliées à une session, les sessions à une application... Certains champs affichés dans les tableaux de données peuvent être utilisés comme outils de navigation, appliquant par la même occasion une configuration spécifique à la nouvelle page, permettant de filtrer les données selon l'originale. Par exemple, dans la table sessions, si un utilisateur clique sur le nom d'une session, il pourra accéder à la page des tâches filtrées par l'identifiant de la session.

c. Performances

Nous l'avons déjà rapidement abordé dans la partie précédente, et encore plus implicitement avant, mais la performance d'une application est un des points d'expérience utilisateur les plus évidents. Une application lente sera rapidement abandonnée par un utilisateur. Aujourd'hui, créer une application web performante n'est pas bien complexe : les machines sont de plus en plus puissantes et les technologies toujours plus efficaces et simples à utiliser.

Les optimisations de code apportés à l'interface utilisateur ont certes stabilisés quelques fonctionnalités, mais n'ont sans doute pas eu un grand impact sur ses performances. Cependant, à l'avenir, elles permettront d'éviter d'avoir des baisses de performances lors de l'ajout ou la modification de code.

d. Personnalisation

Un élément d'UX plus discret, car peu présent dans nos outils quotidiens, est celui de pouvoir personnaliser son expérience. Une application peut proposer une vue par défaut, mais qui de mieux que l'utilisateur lui-même pour savoir de quoi il a besoin ? L'interface administrateur implémente cette approche, notamment à l'aide de la page *dashboard*. Cette page est le point d'entrée de l'application,

là où les utilisateurs arrivent en premier en la démarrant. Ici, ils peuvent rajouter la table de donnée qu'ils souhaitent, avec la configuration qu'ils souhaitent. Petit à petit, des éléments supplémentaires viendront accompagner les tables ; le but est de donner la main à l'utilisateur sur comment il souhaite les utiliser.

Aussi, l'utilisateur peut modifier à son aise les éléments de la barre de navigation ; changer leur ordre, les modifier, ou les supprimer. Encore d'autres éléments sont personnalisables dans l'application. Mais en plus de pouvoir modifier leur configuration, les utilisateurs peuvent l'exporter et la passer à un autre. Comme ArmoniK est déployé sur des serveurs, il est simple d'exposer une de ces configurations exportées pour qu'elle soit disponible pour tous les utilisateurs. La personnalisation devient alors facilement partageable entre tous.

2. Documentation Utilisateur

Beaucoup de technologies et applications industrielles possèdent aujourd'hui un espace de documentation utilisateur. Il s'agit généralement d'un site internet regroupant les différentes informations, conseils et exemples à propos de l'application. Git, Kubernetes, gRPC, Angular possèdent tous leur propre documentation, et sont fréquemment utilisées par de nombreux développeurs pour apprendre et comprendre. Une documentation se doit d'être simple et correctement organisée. Une page doit correspondre à un élément particulier, et être décomposées en sous-parties. Ces parties vont permettre à un utilisateur de trouver facilement ce qu'il cherche. Si nécessaire, des exemples d'utilisations peuvent être affichés.

Pour ArmoniK, beaucoup de documentation utilisateur est déjà disponible ; cependant, celle de l'interface administrateur n'a pas été mise à jour depuis plus d'un an, avant même qu'elle ait été réécrite. Autant dire qu'il ne s'agit plus de la même application ; cette ancienne documentation, déjà incomplète, est devenue obsolète. Il m'a donc fallu rédiger une nouvelle documentation, ce qui est un exercice difficile. En tant que développeur de l'application, je connais cette dernière dans les moindres recoins. Il m'a fallu fragmenter mes connaissances en des éléments les plus simples, pour les décrire et les imaginer le plus facilement possible.

Au moment où ce mémoire est écrit, la documentation utilisateur n'est qu'à l'état de *Pull Request*. Elle nécessitera beaucoup de revues avant d'être acceptée et déployée.

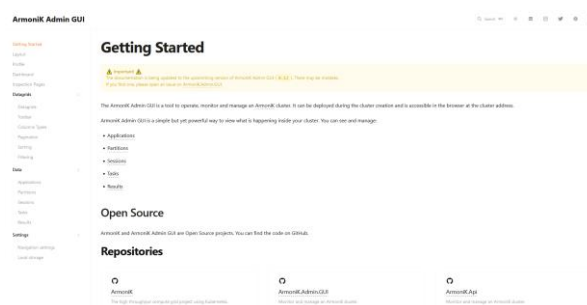


Figure 18 - Documentation utilisateur de l'interface administrateur

3. Retours utilisateurs

Quelques mois après le début de mon alternance, j'ai pu présenter la nouvelle version de l'Admin GUI aux utilisateurs de CACIB (Crédit Agricole). Cet exercice fut pour moi une première : je suis directement allé chez le client, dans un complexe impressionnant, le temps d'une après-midi. A ce moment-là, l'application était dans un état très instable, fortement buggée. Cependant, la présentation s'est déroulée correctement, et les utilisateurs finaux étaient convaincus. Mais le plus important sont

les retours que j'ai pu recueillir sur le travail de mes derniers mois et sur les différents besoins qu'avaient ces utilisateurs.

Cela m'a permis de comprendre davantage ce dont ils avaient besoin, et donc de me concentrer sur les besoins les plus utiles pour eux. C'était aussi un moment intéressant pour discuter de ces besoins : certains voulaient beaucoup, trop techniquement par rapport à ce que ArmoniK dans son ensemble puisse faire sans risquer de perdre en performances. Il a donc fallu leur faire comprendre que certains de leurs besoins n'étaient pas importants et n'auraient aucun impact sur leur expérience. D'autres points, au contraire, étaient des remarques très pertinentes, aussi bien sur des fonctionnalités supplémentaires que pourrait apporter la GUI que sur des détails très importants pour un utilisateur.

En tant que développeur, il est facile d'oublier de faire attention à l'expérience utilisateur. Ce genre d'exercice est donc le bienvenu. Il rappelle le plus important dans un projet informatique de ce type : les utilisateurs. Ce n'est pas un exercice qui doit être trop fréquent, car il pourrait vite fatiguer les utilisateurs et les développeurs. Cette discussion directe avec l'utilisateur final est une des nombreuses caractéristiques des projets agiles, et j'ai pu en comprendre l'utilité ce jour-là.

VII – Résultats

Après plusieurs mois de développement et plus de 600 Pull Requests, l'application a énormément évolué. Aujourd'hui, l'interface administrateur est une application au fonctionnement simple et rapide. Les tables de données sont ergonomiques et ne comportent presque plus aucun bug. Le nombre de fonctionnalités a rapidement grimpé et, des retours utilisateurs, sont relativement simples à utiliser. En clair : l'application a grandi dans la bonne direction. La documentation utilisateur est prête à être déployée pour éclaircir les dernières zones d'ombres d'utilisation.

L'impact des principes SOLID, à l'échelle de l'application, est conséquent, aussi bien d'un côté développeur qu'utilisateur. Pour les premiers, par exemple, il est maintenant plus simple de changer le type d'une colonne d'une certaine table. Auparavant, il fallait modifier au minimum 3 fichiers TypeScript et 1 fichier HTML pour réaliser une telle chose ; maintenant, modifier une ligne suffit. Pour la majorité du code, les développeurs peuvent maintenant arrêter de réfléchir à réimplémenter l'existant pour se concentrer sur le nouveau. Pour les utilisateurs, j'ai déjà pu y faire référence : l'application est beaucoup plus stable, et les bugs moins fréquents.

Mais appliquer ces principes ne fut pas une simple affaire. Pour certains cas très complexes, il m'a fallu plusieurs essais pour pouvoir mettre en place correctement les outils dont j'avais besoin pour appliquer les principes. Parfois, je ne pouvais pas modifier un code sans d'abord me plonger dans un autre. Pour les tables de données, l'exemple est flagrant : il m'a fallu modifier près de 3000 lignes de code dans plus de 60 fichiers, sans compter le travail préalable de réduction de la redondance de code. C'est aussi un exemple très révélateur quant à la profondeur et à l'impact de la dette technique, sur un projet qui est pourtant relativement jeune.

La réduction de bug est aussi amenée par l'implémentation de tests unitaires. Pour 14000 lignes de code TypeScript, nous avons 1500 tests sur plus de 16000 lignes. Ce nombre effarant a permis à l'application d'atteindre une couverture de tests de presque 100% ; même si, pour rappel, ce nombre est peu significatif quant à la qualité de ces derniers. Les mettre en place aura permis de détecter des bugs et de les corriger, de plus que de valider la qualité du nouveau code écrit.

Mais la qualité du code de l'interface administrateur n'a pas fini d'évoluer. Je reste un jeune développeur découvrant encore les bases de son métier : il est normal que mon implémentation de ces points ne soit pas complète, ou partiellement fautive. Mais les résultats parlent : j'ai pris en

expérience. Je dois remettre mes connaissances en question afin de constater mes erreurs et apprendre de ces dernières.

Maintenant que l'application est stabilisée et la dette technique presque réduite à néant, il est nécessaire de se concentrer sur d'autres aspects de l'interface administrateur, notamment la sécurité. Elle n'a pas été nécessaire jusqu'ici, car les services gRPC sont déjà protégés ; cependant, elles sont importantes d'un point de vue utilisateur. Aujourd'hui, si l'un d'entre eux utilise une fonction à laquelle il n'est pas censé avoir accès, il recevra un message d'erreur, sans aucune autre explication. Il sera perdu et pourra peut-être même penser qu'il s'agit d'un bug. Mais si l'application ne lui affiche pas l'option d'utiliser la fonction s'il n'en a pas le droit, alors il n'y aura aucune confusion.

VIII – Pour finir en beauté

Presque un an après, qu'y a-t-il à retenir ? Beaucoup de choses, comme nous avons pu le voir. Que ce soit ArmoniK, l'utilisation de technologies dans le cadre de la philosophie DevOps, de l'organisation d'un dépôt Git ou par l'application de principes de développement et de sécurité, énormément de sujets ont été abordés et analysés dans ce mémoire.

La première chose à retenir est sans aucun doute la dette technique, ce qu'elle représente et comment elle apparaît. C'est un problème qui ne concerne pas un morceau de code, mais plutôt l'implémentation d'une réponse à un problème. Si la réponse n'est pas correctement pensée, si elle ne prend pas en compte tous les cas possibles, si elle n'est pas capable d'appréhender le problème comme il le faut et de manière simple, alors il y aura de la dette technique. Un code est avant tout algorithmique, logique. Et la logique ne s'invente pas. Il faut toujours réfléchir, trouver plusieurs approches, les comparer...

Ce qui est évident, c'est que jusqu'ici, l'intégration de ces principes de développement, notamment SOLID, est une réussite. J'ai de nombreuses fois essayé de les remettre en question, d'essayer de trouver des cas d'utilisation où il était impossible ou contre-productif de les utiliser. Je n'ai cependant pas trouvé. Je pense néanmoins qu'un développeur capable de justifier son choix de ne pas les implémenter est totalement acceptable. Certains composants de l'interface administrateur sont déjà très simples à utiliser et comprendre dans un code, et lorsqu'on sait que ces composants ne vont pas évoluer, on peut comprendre le besoin de ne pas appliquer les SOLID. Ce sont des cas qui restent néanmoins rares et ne doivent en aucun être abusés. Sinon, l'on se retrouve rapidement avec plus de dette que l'on en avait auparavant. La morale est assurément de ne pas jouer, au risque de s'endetter.

Les méthodologies de travail comme le DevOps ou l'organisation des dépôts Git en *Trunk-based development* sont aussi extrêmement difficiles à réfuter. Le simple fait de vouloir simplifier toute partie d'un processus, jusqu'à leur automatisation totale, est compréhensible et de plus en plus accessible. La simplicité est d'ailleurs le principe phare de ce mémoire : elle est à la base de presque tous les principes abordés.

Lorsque j'ai commencé à penser au sujet de ce mémoire, je ne m'attendais pas à aborder certains sujets : l'expérience utilisateur et la sécurité en font partie. Mais au cours de l'année, j'ai pris conscience de leur importance et du besoin de développer mes connaissances dans ces sujets. J'ai beau être un jeune développeur, je ne le resterais pas éternellement. Il est nécessaire que j'apprenne et me renseigne constamment.

La sécurité est sans doute le sujet le plus important ; j'ai souvent rappelé son importance et les différents impacts possibles dus à un manque de sécurité sur une application. C'est aujourd'hui un domaine important en informatique, où de nombreux métiers apparaissent chaque année.

L'expérience utilisateur, quant à elle, est nécessaire pour comprendre facilement les besoins potentiels d'un utilisateur. C'est un outil très pratique pour prioriser l'implémentation de certaines fonctionnalités par rapport à d'autres, mais aussi pour s'assurer d'avoir une application facilement utilisable.

Finalement, l'intégration des principes de développement aura eu impact autre que la simple application de connaissances. J'ai dû dépasser ce sujet, l'étendre ; cela m'a forcé à me renseigner sur de nouveaux sujets, tels que la sécurité, l'expérience utilisateur... En d'autres termes, le sujet de mon alternance ne fut qu'une occasion pour *apprendre à apprendre*. Cette année n'aura donc pas seulement été l'occasion de développer mes compétences techniques, mais aussi mes *soft skills*, des compétences pluridisciplinaires qui me seront fortement utiles à l'avenir.

Annexe

Sources

Définition d'orchestrateur : https://fr.wikipedia.org/wiki/Orchestration_informatique

Conventions de commits Git : <https://www.conventionalcommits.org/fr/v1.0.0/>.

Philosophie et schéma du cycle de vie DevOps : <https://www.atlassian.com/fr/devops>.

Définition de Ray Tracing : https://fr.wikipedia.org/wiki/Ray_tracing

Outils

Schémas réalisés avec Adobe Illustrator.

Extraits de code réalisés avec la plateforme <https://carbon.now.sh/>.