UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ÉVOLUTION ET IMPACT DES TECHNOLOGIES DE DÉPLOIEMENT AU SEIN DES ARCHITECTURES DE MICROSERVICES

RAPPORT DE PROJET DE SYNTHÈSE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE DE LA

MAÎTRISE EN GÉNIE LOGICIEL

PAR

AMINE SOUFYANI

DÉCEMBRE 2022

REMERCIEMENTS

Je tiens à remercier Sébastien Mosser, mon professeur de recherche pour m'avoir accepté au sein de son équipe et de m'avoir donné l'opportunité de travailler ce sujet ainsi que de m'avoir fourni de nombreux conseils afin mener à bien ce projet.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ABRÉVIATIONS, DES SIGLES ET DES ACRONYMES

API : Application Programming Interface

BDD : Base de données

CQRS: Command and Query Responsibility Segregation

DDD: Domain Driven Design

DOM: Document Object Model

HTTP: Hypertext Transfer Protocol

JS: JavaScript

JSON: JavaScript Object Notation

JWT: JSON Web Token

SOA: Service Oriented Architecture

URL : Uniform Resource Locator

RÉSUMÉ

Dans la dernière décennie, les architectures en microservices sont devenues de plus en plus populaires au sein des équipes d'ingénieurs logiciel. Cette variante des architectures SOA nous permet de séparer une application en plusieurs entités logicielles communiquant entre elles via des protocoles de communications légers. Cette séparation nous permet ainsi d'obtenir des architectures découplées et hautement cohésives ainsi que d'augmenter la réutilisabilité au sein d'une architecture. Cependant, cette séparation entraine une certaine complexité à la hauteur du déploiement, car chaque service devra avoir sa propre architecture de déploiement et donc son propre processus de déploiement.

C'est sur ce défi que nous portons notre attention au long de cette étude, en particulier sur l'identification des impacts et des évolutions des architectures de déploiement au sein des architectures en microservices.

Mon travail a principalement consisté en la création d'un outil d'analyse statique de fichiers Docker au sein de dépôts de code afin d'établir quelles modifications sont apportées aux fichiers Docker et la cause de ces changements au sein des architectures étudiées ainsi que l'analyse de deux dépôts de code. Ce logiciel a été développé uniquement en JavaScript et s'adresse à tous les dépôts de code public GitHub.

Mots clés : Microservices, Docker, Architecture, Déploiement, Applications logicielles, Services

ABSTRACT

In the last decade, microservices architectures are becoming increasingly popular within the software industry and software teams. This variant of SOA architecture allows us to separate an application into several software entities communicating with each other via light communication protocols. This separation allows us to obtain decoupled and highly cohesive architectures allowing us to increase the reusability within an architecture. However, this separation leads to an increased complexity at the deployment level because each service will have to have its own deployment architecture.

It is therefore on this challenge that we focus our attention throughout this study, on the identification of the impacts and evolutions of deployment architectures within microservices architectures.

My work mainly consisted in the creation of a tool for static analysis of docker files in code repositories to find what modification are proposed to these files and the cause of these changes within the architecture studied as well as the analysis of two code repositories having a microservice architecture. This software was developed only in JavaScript and is intended for all public GitHub code repositories.

Keywords: Microservices, Docker, Architecture, Deployment, Software, Services

INTRODUCTION

Étudiant au programme de maîtrise en génie logiciel à l'Université du Québec à Montréal, ce document présente la partie du travail de synthèse qui est effectué lors de ma dernière année d'étude.

Ce projet de synthèse en génie logiciel a pris place au sein de l'équipe ACE dirigée par Sébastien Mosser et s'est majoritairement déroulé en autonomie, et à distance à la suite de la pandémie mondiale. Les communications avec mon professeur de recherche se sont principalement effectuées au travers de Mattermost ainsi que Zoom.

Dans un premier temps, une présentation du contexte de l'étude ainsi que de ses objectifs sera menée. Nous allons ensuite nous attarder sur la présentation de notre outil d'analyse ainsi que sur la méthodologie d'analyse développée dans le cadre de cette synthèse pour, dans un troisième temps, procéder à l'analyse de deux projets open source disposant d'une architecture en microservices. Enfin nous procèderons à une analyse comparative entre ces deux projets avant de conclure.

CHAPITRE 1

État de l'art et état de la pratique

Dans ce chapitre, nous allons contextualiser notre projet technique afin d'obtenir une compréhension du champ d'application de notre sujet et de la portée de ce dernier.

1.1 Contexte

L'architecture en microservice a été inventée afin de résoudre certains problèmes causés par les gros projets, notamment comme les problèmes liés aux architectures monolithiques tel que celui de l'entropie logicielle[1] Qui énonce que la dette technique d'un projet augmente avec la durée de vie de ce dernier, car l'ajout de fonctionnalités augmente la taille du code et donc sa complexité.

Une architecture en microservice a pour but de séparer une architecture en une suite de services modulaires de petites tailles et indépendants communiquant entre eux à l'aide de protocoles de communication légers par le biais d'interfaces de programmation d'application (API).

[1] https://en.wikipedia.org/wiki/Software_entropy

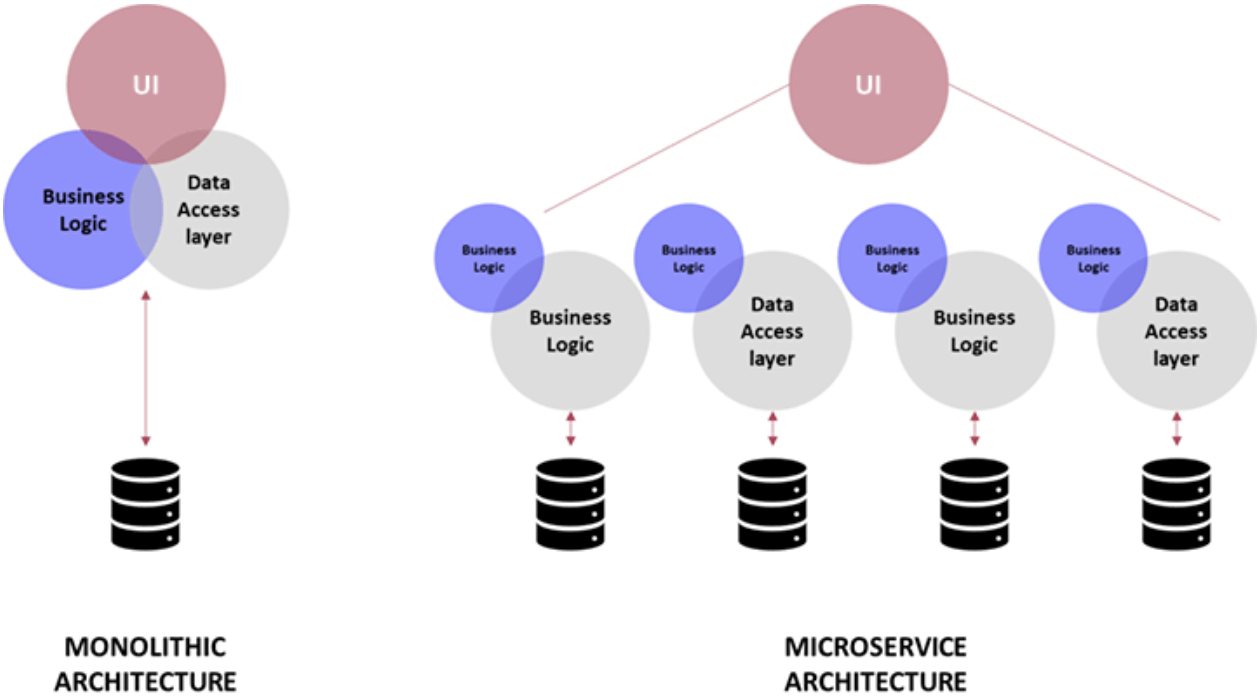


Figure 1 - Différence architecture monolithique / microservices [1]

Cette séparation en petits services indépendants nous permet d'obtenir un plus grand découplage dans notre application. Un autre avantage que nous offre une architecture en microservice est le déploiement modulaire et singulier de chaque microservice, car en effet, un microservice est déployé individuellement contrairement à une application monolithique où nous devons redéployer l'intégralité de l'application.

Déployer une architecture en microservices semble alors être bien différent en comparaison au déploiement d'une architecture monolithique en raison de l'interdépendance entre les différents services[2].

Une approche nous permettant le déploiement d'architectures en microservices est l'utilisation de la conteneurisation ainsi que de l'orchestration. Pour cela, il nous faut lancer un conteneur (processus qui va exécuter l'application sur un serveur) créé à partir d'une image sur un ensemble de serveurs vu comme une ressource unique.

[2] Building Microservices: Designing Fine-Grained Systems (Chapter on Deployment)

Afin de répondre à ce processus de déploiement, de nouvelles technologies ont émergées. Nous pouvons compter parmi elles Docker[3] Et Kubernetes[4] ; deux technologies populaires prévues à cet effet.

Docker est un outil logiciel ou plus particulièrement une plateforme logicielle permettant de simplifier le développement et le déploiement par le biais de de conteneurs virtuels, permettant ainsi exécuter les applications logicielles de la même manière indépendamment de l'environnement. La technologie docker peut aussi être utilisée pour étendre des systèmes distribués de façon qu'ils s'exécutent de manière autonome. Cela permet aux nœuds d'être déployés au fur et à mesure que les ressources sont disponibles, facilitant ainsi l'élasticité d'infrastructure d'un projet. Ici nous pouvons définir l'élasticité comme la mesure par lesquelles un système informatique est capable de s'adapter aux changements de charge de travail en provisionnant et en vidant les ressources de manière autonome, afin qu'elles correspondent au plus à la demande réelle.

Un conteur Docker est construit automatiquement au travers d'une Dockerfile qui est un fichier texte décrivant les commandes qu'un utilisateur utiliserait dans un CLI pour établir un environnement.

Une Dockerfile se présente donc sous la forme d'une suite d'instruction prenant la structure suivante :

[5]

Figure 2 - Example d'Instruction de Dockerfile

[3] https://www.docker.com/

[4] https://kubernetes.io/

[5] https://docs.docker.com/engine/reference/builder/

Ce conteneur docker sera alors déployé (artéfact de déploiement) sur un environnement physique, il existe plusieurs services permettant le déploiement de dockerfiles comme Amazon Elastic Container Service (ECS[6]), Azure ou encore Kubernetes.

Kubernetes, quant à lui, est un système open source qui vise à fournir une plateforme permettant d'automatiser le déploiement des conteneurs d'applications sur des clusters de serveurs.

Comme nous avons pu le voir, du fait de l'évolution des architectures monolithique en architectures distribuées; par exemple SOA et microservice de nouveaux besoins ont fait leurs apparitions. Ces besoins nous ont poussés à inventer de nouveaux outils tels que Docker et Kubernetes nous permettant de simplifier et d'adapter le processus de déploiement.

1.2 Enjeux et objectifs

Les enjeux de la thématique du déploiement au sein des architectures en microservices sont donc une partie extrêmement importante pour toute entreprise. En effet, les déploiements se veulent de plus en plus fréquents au sein des projets en génie logiciel afin de mettre dans les mains des utilisateurs une nouvelle version au plus vite. Cela est notamment le cas de la méthodologie DevOps [7] où les processus de déploiements ont donc été automatisés pour répondre à une fréquence de déploiement plus élevée et nécessitant alors une maintenance des pipelines de déploiement.

Une modification au niveau du processus de déploiement peut avoir un impact majeur au sein d'une organisation et de ses utilisateurs en provoquant par exemple, l'arrêt ou le non-fonctionnement d'un service ce qui, à son tour, peut s'avérer couteux pour le client ou l'entreprise déployant le logiciel, ou même dangereux dans le cas de logiciel où une haute disponibilité est critique tel qu'un logiciel de contrôle des flux de trafic aérien.

Notre objectif sera alors de déterminer quels sont les évolutions et les impacts des technologies de déploiement au sein des architectures en microservices.

[6] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html

[7] The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations - Gene Kim, Patrick Debois, Jez Humble, John Willis

1.3 Problématique

Comme nous avons pu l'établir dans nos deux précédentes sections, les architectures en microservices nous permettent de découper l'application en multiples services pouvant avoir différents usages technologiques pouvant être tous déployés sur leurs propres serveurs. Cela peut impliquer une complexité supérieure au niveau des moyens de déploiement en contraste à celle d'une architecture monolithique.

La problématique de ce projet se situera au niveau de l'identification des motifs quant aux changements des *Dockerfiles* (fichier descripteur d'images Docker) et des impacts que cela peut provoquer au niveau du processus de déploiement ainsi que sur l'architecture en microservice elle-même.

Notre travail a donc porté sur l'identification des facteurs de changements au niveau des Dockerfiles afin d'établir quelles sont les causes de changements au niveau du processus de déploiement d'une architecture en microservice et comment cette dernière évolue-t-elle.

CHAPITRE 2

Développement d'un outil d'analyse

Dans ce chapitre, nous allons développer plusieurs aspects du développement de notre outil d'analyse[8]. Dans un premier temps, nous parlerons des motivations nous menant à la création d'un tel outil. Nous discuterons ensuite des technologies utilisées et de l'architecture du logiciel, puis des fonctionnalités de ce dernier et enfin, des limitations de notre outil et des améliorations que nous aurions pu lui apporter.

2.1 Motivation

Les dépôts de code inclus dans notre corpus de référence d'architecture à étudier contiennent des centaines à des milliers de commits, et leur durée d'existence peut s'étendre sur de multiples années comme pour le projet eShopOnContainers [9] Qui contiens 3 978 commits à ce jour et se déroule sur plus de cinq ans (voir projet S04 sur la table 1). Un processus manuel d'analyse commit par commit afin d'identifier les modifications des *Dockerfiles* s'avérerait alors très long avec une chance d'introduire des erreurs lors de notre analyse.

Pour mener à bien cette analyse, nous avons donc décidé de développer un outil permettant de faciliter l'agrégation des données pertinentes au sein de ces dépôts de code afin de réduire le travail manuel d'analyse.

Dans un premier temps, nous avons déterminé que nous souhaiterions répondre aux questions suivantes par l'utilisation de notre outil d'analyse :

- Quels sont les fichiers Docker utilisés par le projet ?
- Quand les fichiers Docker ont-ils été modifiés ?
- Quels changements a-t-on d'une version d'une Dockerfile à la suivante ?

[8] https://github.com/ekiriano/DockerHistoryVisualizationApp

[9] https://github.com/dotnet-architecture/eShopOnContainers

2.2 Technologies et architecture

Pour mener à bien la construction de cette application, nous avons choisi d'effectuer le développement d'une application web où la partie back end et front end sont séparées. Pour ce faire nous avons décidé d'utiliser les technologies suivantes :

- React.js [10]Qui est une librairie open source et gratuite écrite en JavaScript et disposant de sa propre syntaxe (JSX) : cette librairie est utilisée afin de développer des interfaces utilisateur basées sur la composition de composants graphiques. Cette librairie a été créé et est actuellement maintenue par Meta.
- Node.js [11]Est un environnement d'exécution JavaScript Open source multiplateforme et backend permettant d'exécuter du JavaScript en dehors du navigateur web en conjonction avec Express.js qui est un canevas logiciel d'application web basé sur node.js, publié en tant que logiciel gratuit sous la licence MIT, conçu principalement pour le développement d'application web et d'API.
- MongoDB[12], est un système de gestion de base de données orienté document open source, sous-la. Licence Apache.
- Ainsi que Typescript [13] Qui est un sur-ensemble syntaxique strict de JavaScript développé et maintenu par Microsoft et nous permet d'ajouter des types statiques à JavaScript.

Notre application est très simple sur son modèle de fonctionnement et, à un haut niveau, son architecture ressemble à l'architecture présentée en figure 3.

[10] https://reactjs.org/

[11] https://nodejs.org/en/

[12] https://www.mongodb.com/

[13] https://www.typescriptlang.org/

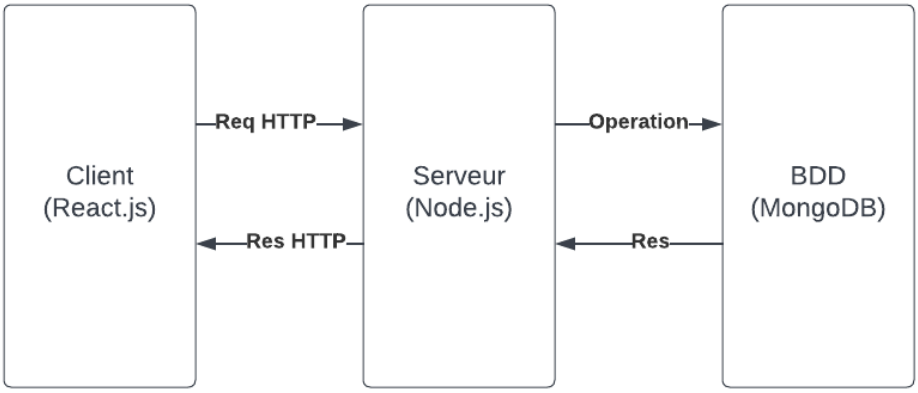


Figure 3 - Architecture haut niveau de l'outil d'analyse

Nous disposons d'un client / interface graphique communiquant à l'aide de requêtes HTTP au serveur qui lui établit une connexion à la base de données. Autrement dit, nous avons mis en place une architecture en trois couches[14].

Dans cette partie concernant l'architecture nous nous concentrerons sur l'organisation de l'API de la couche applicative, à savoir notre serveur node.js :

[14] https://www.ibm.com/cloud/learn/three-tier-architecture#:~:text=Three%2Dtier%20architecture%20is%20a,associated%20with%20the%20application%20is

1. Authentification

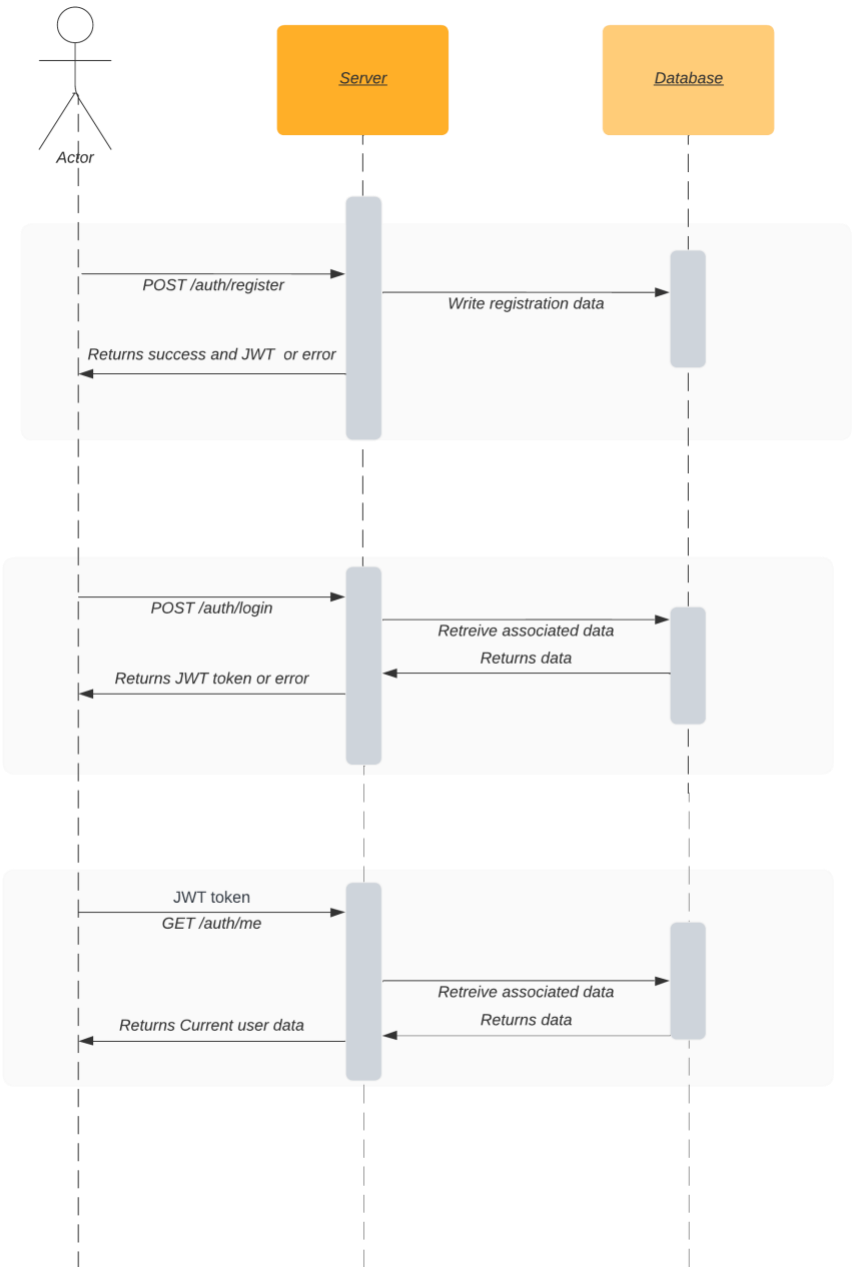


Figure 4 – Illustration API Authentification Utilisateur

Au niveau de l'API concernant l'authentification de l'utilisateur nous disposons de trois routes :

a. POST /register

Ce point d'API permet d'enregistrer un utilisateur au sein de l'application en lui passant comme données son nom d'utilisateur, son courriel et mot de passe. L'application retournera alors un code de succès ou d'erreur.

b. POST /login

   Ce point d'api permet de connecter l'utilisateur au sein de l'application en envoyant son courriel et son mot de passe. L'application nous répondra alors par un jeton d'authentification JWT si la connexion a réussi.

c. GET /me

   Ce point d'api permet d'obtenir les informations liées à l'utilisateur courant si ce dernier envoie dans les headers de la requête un jeton d'authentification JWT valide. Dans le cas contraire, un message d'erreur sera retourné.

2. Projet

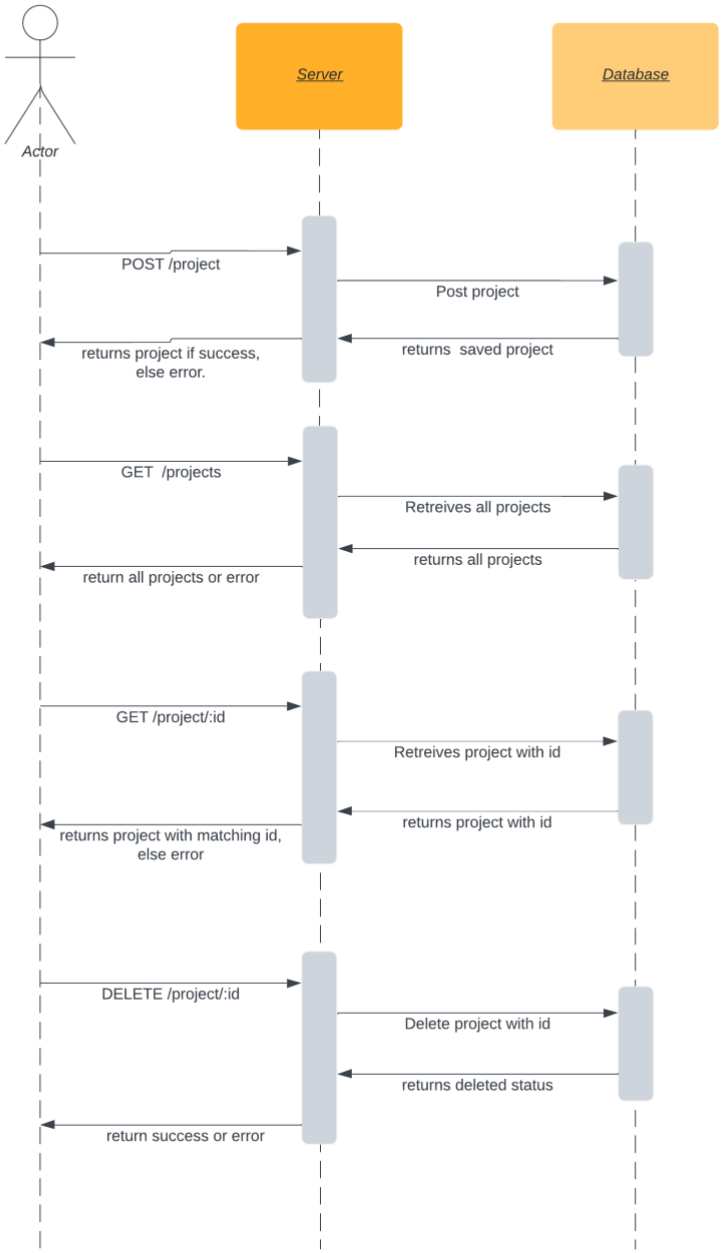


Figure 5 - illustration API projets

Au niveau de l'API concernant les projets de l'utilisateur, nous disposons de quatre routes qui toutes requiers l'authentification de l'utilisateur :

a. POST /projet

   Ce point d'API, permet d'enregistrer les données d'un projet à partir de son url GitHub,
   Les données enregistrées comportent les relations aux branches du projets, l'url git ainsi que le nombres de branches.

b. GET/projets

   Ce point d'API, permet de récupérer l'ensemble des projets pour un utilisateur.

c. GET/projets/ :id

   Ce point d'API, permet de récupérer les informations du projet dont l'identifiant est précisé

d. DEL /projet/ :id

   Ce point d'API, permet de supprimer les informations du projet dont l'identifiant est précisé

3. Branches

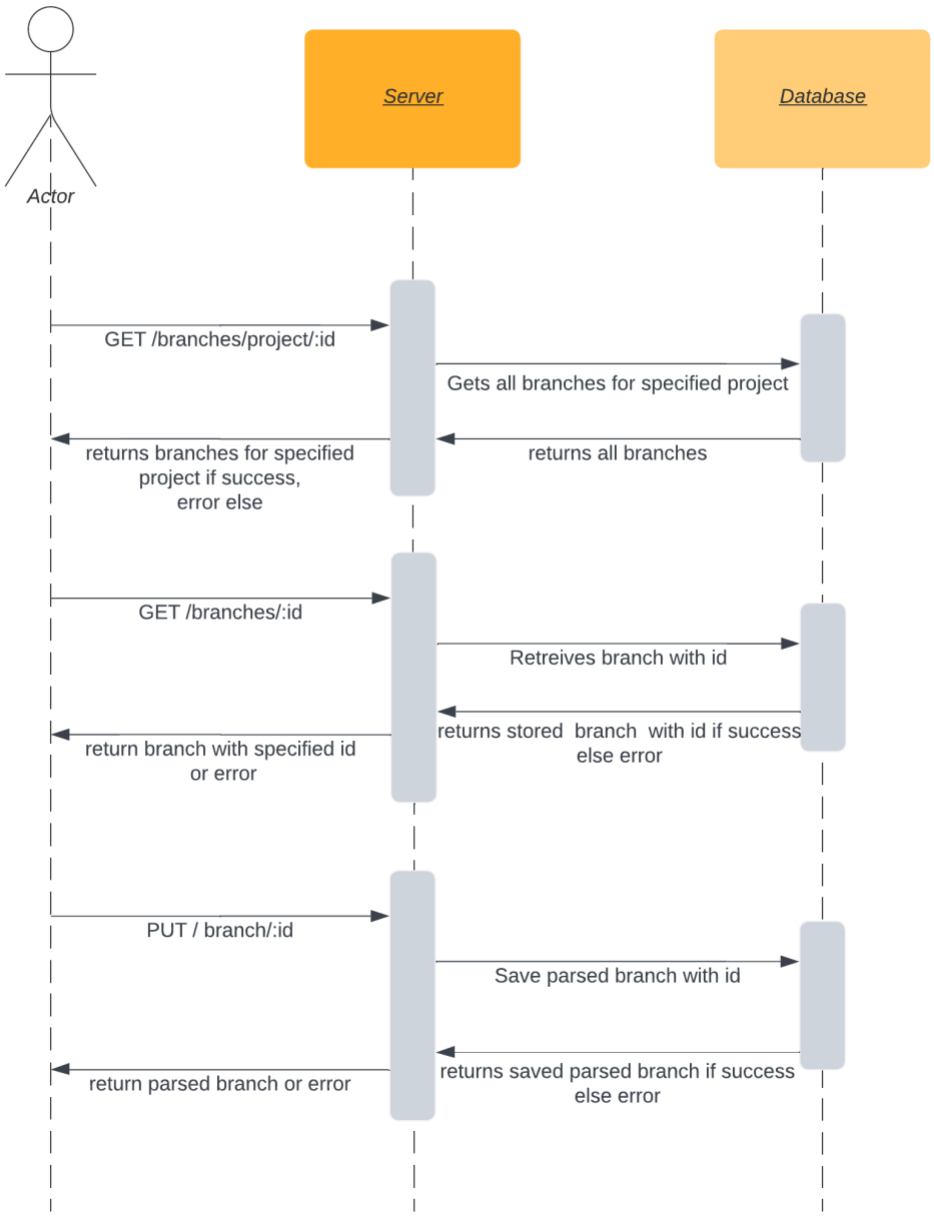


Figure 6 - illustration API branches

Au niveau de l'API concernant les projets de l'utilisateur, nous disposons de trois routes qui toutes requièrent l'authentification de l'utilisateur :

a. GET/branches/project/ :id

   Ce point d'API permet de récupérer les données des branches spécifiques à un projet.

b. GET/branches/ :id

   Ce point d'API permet de récupérer les informations de la branche dont l'identifiant est précisé.

c. PUT /branch/ :id

   Ce point d'API permet de de traiter les Dockerfiles pour la branche spécifiée et retourne les informations à l'utilisateur.

   Les informations comprennent la liste des fichiers Docker ainsi que la liste des commits où ces derniers sont modifiés ainsi que les fichiers en eux-mêmes.

2.3 Fonctionnalités

Les fonctionnalités présentées par notre outil logiciel sont nombreuses et peuvent être regroupées en deux catégories distinctes : les fonctionnalités utilisateur et les fonctionnalités propres à l'analyse des fichiers Dockers.

Dans cette première catégorisation, les fonctionnalités utilisateur regrouperont les fonctionnalités concernant les interactions de l'utilisateur avec le logiciel d'analyse :

1. Un utilisateur peut créer un compte utilisateur, se connecter à son compte ainsi que se déconnecter, et l'authentification de l'utilisateur est gérée à l'aide de jeton d'authentification JWT[15].
2. Un utilisateur peut ajouter un projet GitHub à son compte afin d'y accéder ultérieurement.
3. Un utilisateur peut alors sélectionner la branche GitHub qu'il souhaiterait analyser afin d'obtenir les informations utiles.
4. Un utilisateur peut afficher une timeline des modifications apportées aux Dockerfiles.
5. Un utilisateur peut afficher une visualisation commit par commit des fichiers Docker présents dans le dépôt de code GitHub.
6. Un utilisateur peut aussi afficher une vue différentielle entre deux mêmes Dockerfiles entre deux commits.

[15] https://jwt.io/

La seconde partie de notre catégorisation nous permet de cloner un dépôt de code en local et d'en extraire les branches, mais aussi de traverser tous les commits d'une branche afin d'en extraire le contenu des fichiers Docker présent ainsi que les changements de ces derniers.

Notre logiciel nous permet aussi d'obtenir la différence entre les différentes commandes d'un Dockerfile en nous basant sur une heuristique.

Le processus afin d'extraire les différences entre les différentes commandes est la suivante :

Nous utilisons un outil nous permettant d'obtenir les informations de la Dockerfile sous le format de la figure.7 :


```
[
 {
  "position" : 1,
  "operation" : "REMOVAL",
  "type" :  "FROM"
 },
 {
  "position" : 1,
  "operation" : "ADDITION",
  "type" :  "FROM"
 },
 {
  "position" : 2,
  "operation" : "REMOVAL",
  "type" :  "ENV"
 }
]
```


Figure 7 - diff output example

Cela nous permet ensuite de calculer la différence entre les deux fichiers. Un des problèmes que nous rencontrons alors pour calculer la différence est le déplacement de ligne de code au long du Dockerfile. Notre heuristique de calcul suppose que la position d'une opération au sein du Dockerfile ne change pas au long du projet. Cela nous permet alors d'obtenir une approximation des changements au sein d'une Dockerfile. Une visualisation des fichiers côte à côte nous permettra ensuite de réduire les erreurs introduites par notre heuristique.

Nous obtenons alors les différences sous le format démontré en figure 8 entre deux mêmes Dockerfiles, nous permettant ainsi d'obtenir une idée des changements ayant eu lieu dans cette Dockerfiles.


```
{

 "ADDITION" : {"FROM" : 0, "ENV" : 0 , "RUN" : 0 , "COPY" : 0 , "VOLUME" : 0,
"USER" : 0, "WORKDIR" : 0},

 "REMOVAL" : {"FROM" : 0, "ENV" : 1 , "RUN" : 0 , "COPY" : 0 , "VOLUME" : 0,
"USER" : 0, "WORKDIR" : 0},

 "MODIFICATION" : {"FROM" : 1, "ENV" : 0 , "RUN" : 0 , "COPY" : 0 , "VOLUME" :
0, "USER" : 0, "WORKDIR" : 0}

}
```


Figure 8 - Result process of dockerfile diff

2.4 Limitations et Améliorations

Durant la mise à l'épreuve de notre logiciel lors de l'analyse des dépôts de codes, nous nous sommes rendu compte que ce dernier a des limitations dues à un problème d'alignement des exigences que l'on a définies par rapport aux exigences dont nous avons actuellement besoin.

En effet, lors de l'analyse, notre conception du problème a changé en plus que de connaitre les changements qui ont été apportés aux Dockerfiles, il est tout aussi important voir plus de remettre en contexte ces changements au sein du commit en question, à savoir quels autres fichiers en dehors des Dockerfiles ont aussi été modifiés pour un certain commit.

Notre logiciel pourrait alors être amélioré selon deux pans :

D’une part, notre logiciel pourrait être amélioré au niveau de sa robustesse, celui-ci ayant été développé initialement en tant que prototype d’analyse. De fait, les vérifications de données utilisateur ainsi que la gestion des erreurs ont été quelque peu mises à l’écart pour nous permettre d’itérer au plus vite vers une version utilisable. Les points mentionnés précédemment devront être mis en place afin que le logiciel puisse être utilisé par le grand public.

D’autre part, notre logiciel pourrait être amélioré par l’ajout de fonctionnalités pertinentes à l’analyse même des Dockerfiles. L’une des fonctionnalités les plus importantes dont nous aurions besoin concerne la contextualisation des changements des Dockerfiles. Pour ce faire, notre logiciel devrait être capable de récupérer tous les fichiers pertinents liés à une modification d’une Dockerfile, par exemple en nous indiquant les changements d’un fichier référencé dans une commande RUN.

CHAPITRE 3

Méthodologie d'analyse

Dans les chapitres précédents, nous avons établi notre objectif d'identification des changements au niveau des Dockerfiles afin de déterminer les causes de ces changements et leurs effets en tant qu'artéfact de déploiement sur les architectures en microservices. Nous avons aussi remis en cause les capacités de notre logiciel d'analyse dans le cadre du manque de contextualisation que ce dernier nous offre pour effectuer une analyse quantitative.

Dans ce chapitre, nous élaborerons une méthodologie d'analyse en fonction de critères permettant d'atteindre nos dits objectifs, tout en prenant en compte les limitations de notre outil.

3.1 Choix des projets à étudier

Étant donné les limitations de notre outil d'analyse explicitées précédemment, nous avons décidé de basculer d'une étude quantitative à une étude plus qualitative et, en effectuant ce choix, notre processus d'analyse comportera une composante manuelle qui pourrait s'avérer être longue. C'est pour cela que nous avons choisi de réduire la sélection des projets étudiés.

Initialement, notre corpus est composé de treize projets open source disponibles publiquement sur GitHub. Ces projets nous ont été fournis par l'équipe de recherche car ils servent d'architectures de référence à cette dernière.

La taille des projets étant disparate (voir en table.1), notre objectif premier fut d'effectuer une sélection de deux dépôts de code comparables nous permettant de réaliser une étude représentative.

Id	Origine	Contributeurs	Dockerfiles présents	# commits
S01	https://github.com/DescartesResearch/TeaStore	13	✓	1647
S02	https://github.com/sitewhere/sitewhere	15	X	2659
S03	https://github.com/spring-petclinic/spring-petclinic-microservices	31	✓	710
S04	https://github.com/dotnet-architecture/eShopOnContainers	99	✓	3982
S05	https://github.com/GoogleCloudPlatform/microservices-demo	90	✓	680
S06	https://github.com/mspnp/microservices-reference-implementation	12	✓	415
S07	https://github.com/sqshq/PiggyMetrics	13	✓	290
S08	https://github.com/EdwinVW/pitstop	13	✓	337
S09	https://github.com/instana/robot-shop	14	✓	362
S10	https://github.com/digota/digota	1	✓	89
S11	https://github.com/microsoft/PartsUnlimitedMRPmicro	7	✓	25
S12	https://github.com/sczyh30/vertx-blueprint-microservice	3	✓	86
S13	https://github.com/microservices-demo/microservices-demo	49	✓	1704

Table 1 - Table des projets

Taille échantillon	**13**
Moyenne	**998.923**
Médiane	**415**
Valeur minimum	**25**
Valeur maximum	**3982**
Spectre	**3957**
Intervalle interquartile	**1486**
Premier quartile	**189.5**
Troisième Quartile	**1675.5**

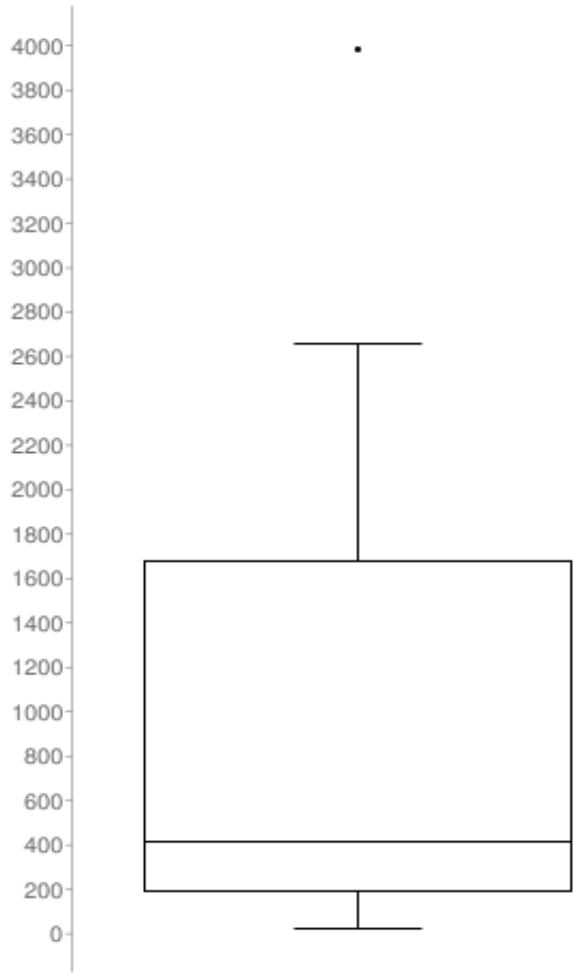


Table 2 – Boxplot de distribution des commits

Afin d'effectuer la sélection des deux dépôts de codes que nous avons évaluer trois critères afin de filtrer les projets :

- La présence de Dockerfiles.
- La durée de vie du projet (par nombre de commits).
- Ainsi que l'orientation du projet vers la technicité ou vers le domaine.

En ce qui concerne le premier critère; la présence de Dockerfiles; nous avons écarté le projet *S02* car celui-ci ne contient pas de Dockerfiles.

Pour ce qui est du second critère, nous avons dans un premier temps établi la médiane de commits au sein des treize projets de notre corpus. Cette médiane se situe à 415 commits. Nous avons ensuite extrait une sélection de projets autour de cette médiane. Trois projets se sont ainsi distingués du fait de leur proximité à cette médiane, à savoir les projets *S06* (415 commits), *S08* (337 commits) et *S09* (362 commits).

À partir de ces trois projets, nous avons écarté le projet *S06*, malgré une correspondance au niveau de la taille des commits, car trop complexe à étudier. En effet, ce projet est lui-même composé de références à d'autres dépôts de code, rendant le processus d'analyse moins direct. Nous avons donc opté pour des dépôts de code où toutes les informations sont situées au sein du même dépôt de code.

Ainsi, nous nous sommes calqués sur le projet S08 d'une taille de 337 commits et sur le projet S09 d'une taille de 362 commits.

De plus, le projet s09 (robot-shop) est orienté vers la démonstration de la mise en place des technologies derrière les architectures en microservices tandis que le projet S08 (pit-stop) est plus centré sur l'organisation et la séparation de la logique d'affaire au sein des architectures en microservices.

Notre objectif en effectuant cette dichotomie est de repérer des motifs et des différences entre ces deux types de projets.

3.2 Méthodologie d'analyse

Notre analyse se doit d'être répétable entre tous les projets de notre corpus d'étude. À ces fins, nous devons être capables d'établir un processus reproductible pour tous les projets.

3.2.1 Point de départ

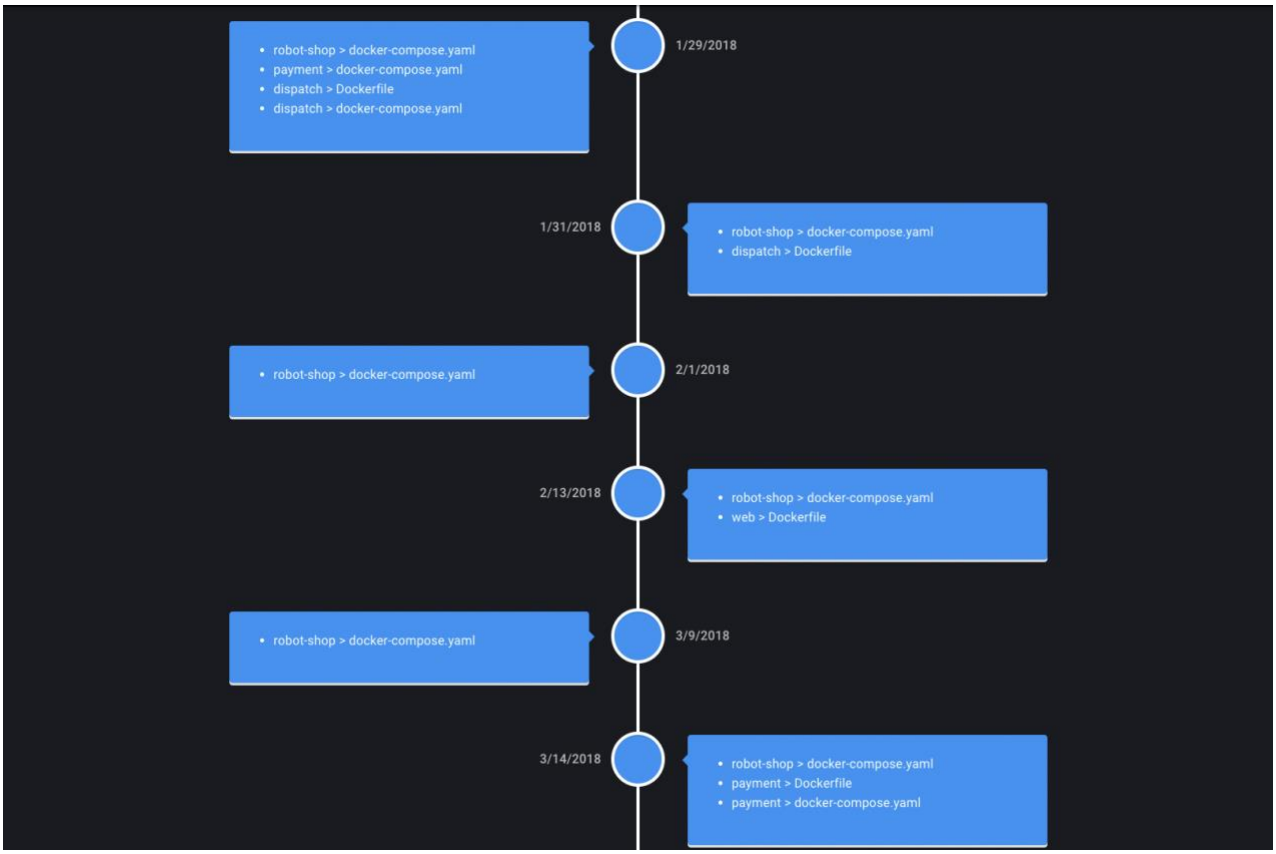


Figure 9 - Timeline outil d'analyse

Nous prendrons comme point de départ à notre analyse l'URL GitHub d'un projet public contenant des Dockerfiles. Cette URL sera ensuite renseignée dans notre outil d'analyse afin de pouvoir en extraire certaines informations.

Une fois l'URL renseignée, nous sélectionnerons la carte correspondant au dépôt de code concerné. Cela nous affichera la liste des branches composant ce dépôt et, ainsi, nous pourrons sélectionner la branche que nous souhaitons étudier. Cette sélection lancera l'étape de récupération de données pertinentes par le serveur. Une fois ce processus asynchrone terminé, nous pouvons accéder aux informations des Dockerfiles au sein de cette branche.

Nous nous dirigerons ensuite sur l'outil Timeline de notre outil d'analyse afin d'obtenir les dates de modification des Dockerfiles ainsi que les groupements de Dockerfiles modifiés ; voir figure.9.

Nous avons choisi ce point de départ car il nous permet d'obtenir la liste concise des Dockerfiles et Docker-compose[16] Au cours de la durée du projet, ce qui est un point crucial pour pouvoir établir une analyse chronologique afin de déceler l'évolution de ces derniers.

À la fin de cette étape, nous aurons un document listant les dates de modifications ainsi que les fichiers concernés par des modifications.

3.2.2 Analyse des Dockerfiles

Lors de cette étape, nous nous servirons de la liste datée des modifications afin de connaitre l'espace de temps à étudier entre chaque modification. Nous chercherons donc à comprendre ce qu'il s'est passé de manière générale dans cet intervalle de temps, mais aussi qu'est-ce qui s'est passé au sein des fichiers Docker.

Pour cela, nous allons reprendre notre document listant les dates de modifications, ainsi que les fichiers impactés par ces modifications. Nous allons ensuite utiliser un outil nommé GitKraken. GitKraken est un client git sous forme d'interface graphique facilitant l'utilisation et l'efficience de git.

[16] https://docs.docker.com/compose/

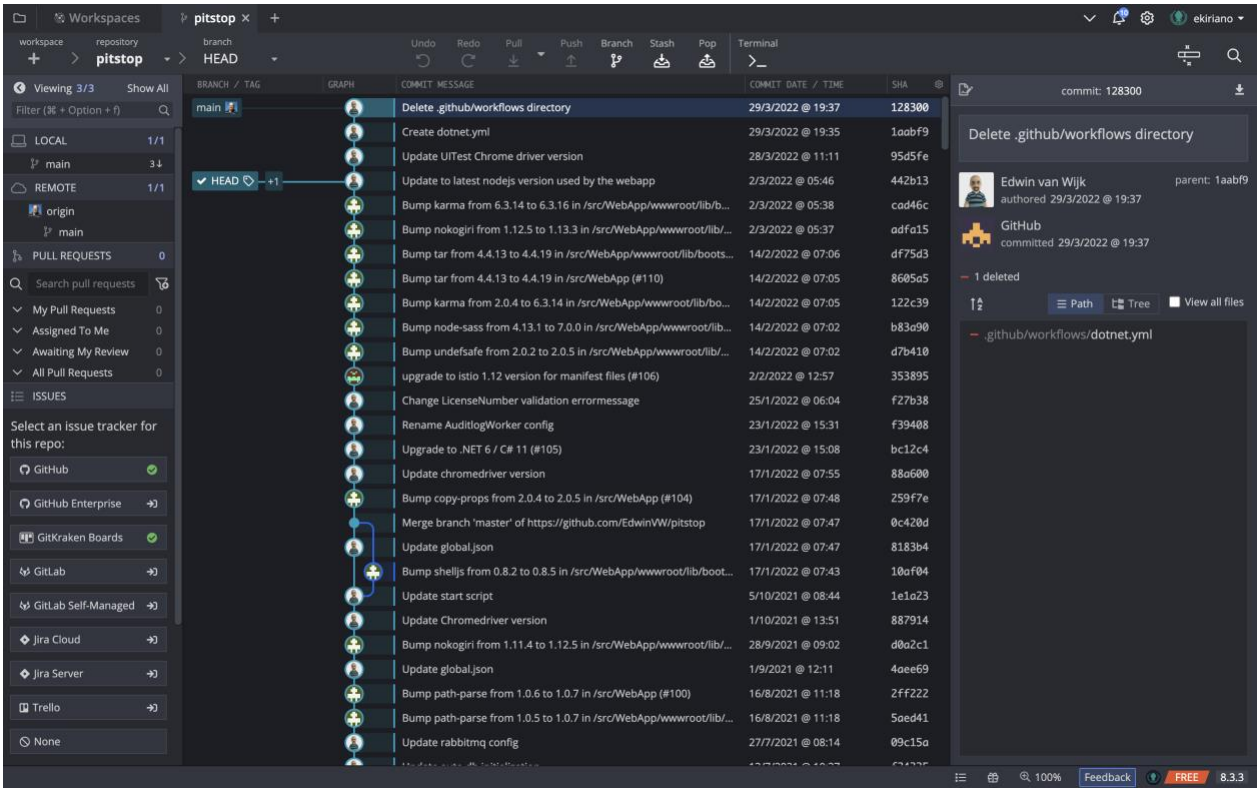


Figure 10 - Interface GitKraken

Nous allons configurer notre client Git de manière à afficher uniquement les commits de la branche qui nous intéresse dans la partie gérant les branches distantes du logiciel, puis d'afficher les commits par ordre de date descendante, ainsi que de décocher la case « Voir tous les fichiers ». Les données alors présentées par l'application nous permettrons de calquer notre analyse sur notre ligne chronologique.

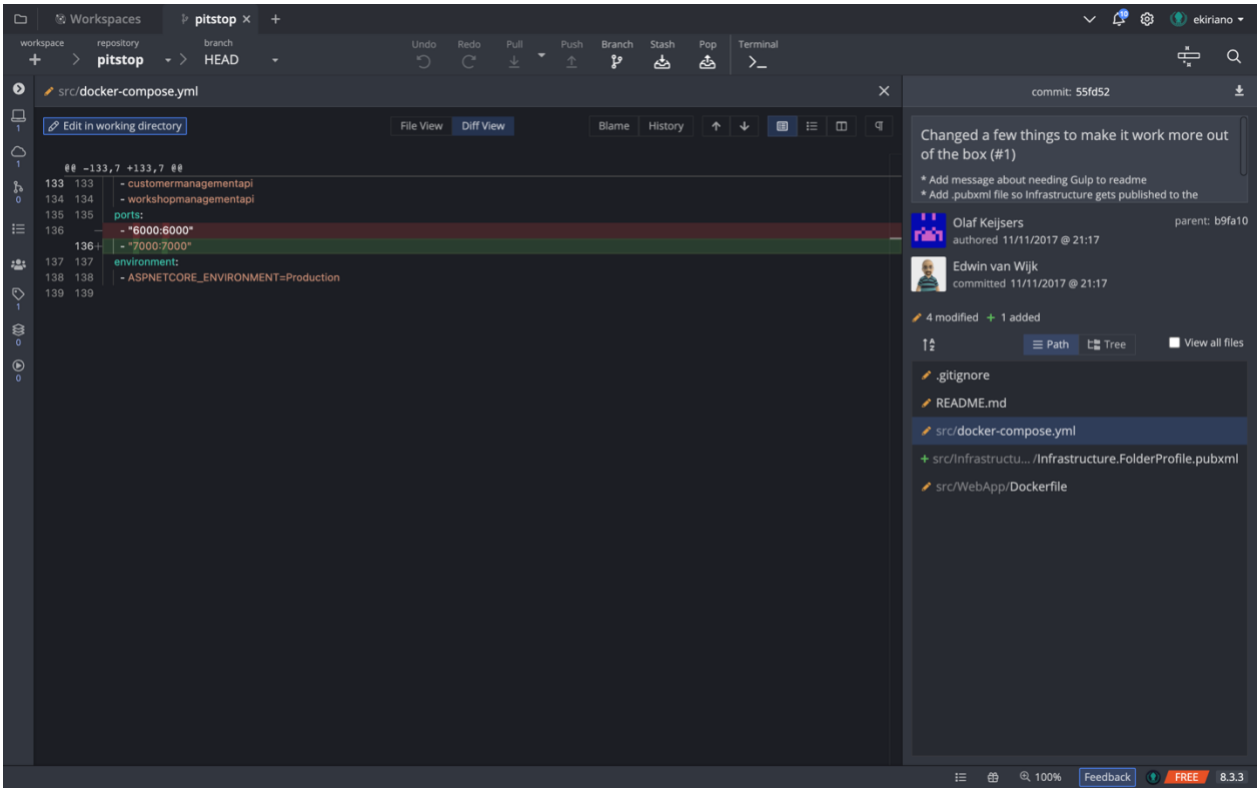


Figure 11 – Vue différentielle de GitKraken.

GitKraken nous permettra d'accéder à la liste des fichiers modifiés par les commits et d'en obtenir une visualisation différentielle comme montré sur la figure 11.

Nous « remonterons » ensuite tous les commits du dépôt de code afin d'étudier le contexte dans lequel ces derniers ont eu lieu.

Ici, nous définissons le contexte comme un ensemble de paramètres nous permettant de comprendre l'origine des modifications associées aux Dockerfiles.

Lors de notre étude, nous avons trouvé les paramètres suivants comme étant les plus indicatifs afin de comprendre les modifications :

- Les fichiers modifiés lors des commits
- Les fichiers référencés dans les Dockerfiles ayant subi une modification
- Les modifications des Dockerfiles en elles-mêmes
- Les commentaires dans le code source

- Les modifications de documentation telles que les ReadMe.md
- Les commentaires descriptifs des commits

3.2.3 Documents récapitulatifs (logs)

Lors de la remontée des commits du dépôt de code, nous procédons à la fabrication d'un document intermédiaire récapitulatif des changements apportés aux Dockerfiles en suivant la structure du document établi à l'étape 1.

Cette étape de construction de document récapitulatif nous servira ensuite comme base afin d'élaborer une catégorisation des changements entre les différents projets sélectionnés.

3.2.4 Catégorisation des logs

Une fois l'ensemble des dépôts de code analysé par nos soins, nous nous munirons de leurs documents récapitulatifs afin d'édifier une catégorisation des paramètres à étudier.

Pour cela nous allons adopter une approche « Bottom-Up », autrement dit, nous allons prendre comme point de départ notre ensemble (E) de documents récapitulatifs afin d'en extraire une liste de catégorisation granulaire des modifications ayant eu lieu au sein des Dockerfiles (Ln), pour ensuite pouvoir en extraire des groupements de types de modifications ayant eu lieu dans les Dockerfiles étudiés.

En partant de notre ensemble de documents, nous avons alors choisi d'établir un document CSV, par souci de simplicité, car ce format nous permet de placer toutes les informations des documents récapitulatifs afin de simplifier leur traitement ultérieur.

Le document CSV que nous avons décidé de construire à la structure suivante : la première colonne contient les dates des modifications des Dockerfiles tandis que la première rangée contient les catégories granulaires nous servant d'étape intermédiaire. Ensuite, chaque case contient quelles modifications a eu lieu dans la Dockerfile ainsi que le décompte des modifications pour cette dernière. Voir exemple en annexe.

Une fois les deux documents CSV établis, nous effectuons une union au niveau des paramètres des deux projets afin de constituer la liste de tous les paramètres ($L = L1\ U\ L2\ U\ \ldots\ Ln$).

À partir de cette liste, nous allons alors établir des groupements plus larges qui nous serviront de base afin de fonder notre analyse.

3.3 Choix de nos critères d'analyse

Après la catégorisation effectuée précédemment ainsi que nos connaissances individuelles, nous avons extrait les catégories suivantes de ces deux projets :

- La création initiale des Dockerfiles et Docker-compose (l'initialisation du projet) : cette dimension est importante car elle se rapporte au déploiement initial de l'architecture en microservices. Bien que cette dernière soit un critère ayant une occurrence unique dans le projet, elle nous permet d'établir un point de départ à notre analyse, ce qui est fondamental à la compréhension de l'évolution de l'architecture dans son ensemble.

- Les changements de ports : cette dimension se rapporte à la communication inter-service et plus précisément à la configuration de cette communication. Nous avons choisi cette dimension du fait de l'importance de la configuration de port. En effet, un port mal exposé dans un conteneur peut mener à une erreur de connexion à une base de données ou bien, à une mauvaise communication avec un service, résultant alors à la non-utilisation d'un service ou à la perte de données.

- Les modifications de commandes liées aux scripts est un point qui nous a semblé crucial d'inclure car elles se rapportent à la configuration des images Dockers ainsi que de leurs dépendances et donc de la configuration applicative des artéfacts de déploiement. En effet, les scripts nous permettent d'installer des dépendances aux images ou de configurer une image; par exemple, lors la configuration d'une image linux, des permissions peuvent être accordées ou encore des librairies nécessaires à l'exécution de l'artefact peuvent être installées.

- Les mises à jour d'images et changements d'images, se rapportant ainsi à la dimension de la mise à jour et de la maintenance des conteneurs et donc des artéfacts de déploiement. Ce point nous semble important car il peut aussi avoir attrait à la sécurité et l'optimisation, selon les mises à jour d'images.

- La configuration de variables d'environnements car elle se rapporte à la configuration modulaire des conteneurs nous permettant d'injecter des variables dans la configuration des artéfacts de déploiement tel que des clés d'API permettant ainsi une configuration simple par un tiers.

- L'ajout de dockerfiles et modifications de Docker-compose se rapporte à l'évolution de l'architecture de déploiement. En effet, chaque ajout de Dockerfile est crucial afin de comprendre l'évolution de l'architecture. De même l'évolution des Docker-compose nous indiquent l'évolution des services ainsi que de leurs dépendances et interdépendances, nous permettant ainsi de dépeindre l'évolution de l'architecture de déploiement.

Cette taxonomie des changements s'opérant au sein des Dockerfiles est propre aux deux projets que nous avons étudiés. En effet, cette dernière a été constituée par le biais des documents récapitulatifs des changements et donc n'est pas généralisable à tous les Dockerfiles. Cependant, la méthodologie menant à la production de la taxonomie l'est.

Il est aussi important de souligner que la taxonomie produite est issue d'une analyse et d'un regroupement des données d'origine manuelle. Cette composante humaine au sein de la production de la taxonomie peut engendrer des erreurs et donc, dans certains cas, une taxonomie incorrecte. Il est ainsi fortement recommandé de croiser les références afin d'obtenir une taxonomie plus fiable.

Dans le cas de notre taxonomie, nous n'avons pas eu l'occasion de faire valider les documents récapitulatifs des changements ou de collaborer lors de l'élaboration de la taxonomie. Nous avons tout de même établi les documents récapitulatifs à plusieurs occasions afin de nous assurer de leurs cohérences.

CHAPITRE 4

Présentation et analyse de RobotShop

Dans ce chapitre nous allons dans un premier temps introduire le projet RobotShop, puis nous nommerons les technologies utilisées au sein de ce projet ainsi que l'architecture mise en place. Enfin, nous conduirons une analyse selon les critères définis au préalable dans notre méthodologie d'analyse.

4.1 Introduction

L'application Robot Shop est un exemple d'application de microservices servant de bac à sable afin de tester et d'apprendre les techniques d'orchestration et de surveillance des applications conteneurisées.

4.2 Technologies

De nombreuses technologies sont utilisées au sein du projet robot shop et nous pouvons y retrouver plusieurs langages d'implémentations : Node.js, Java, Python, Golang, PHP ainsi qu'AngularJS. Au niveau de la persistance des données, nous retrouvons MongoDB, MySQL ainsi que Redis. Nous observons aussi l'utilisation de RabbitMQ pour la gestion des communications et Instana pour effectuer la surveillance de l'application. Enfin, nous retrouvons une utilisation de Kubernetes afin de mettre en place l'orchestration de conteneurs ainsi que Docker pour la création d'images.

4.3 Architecture

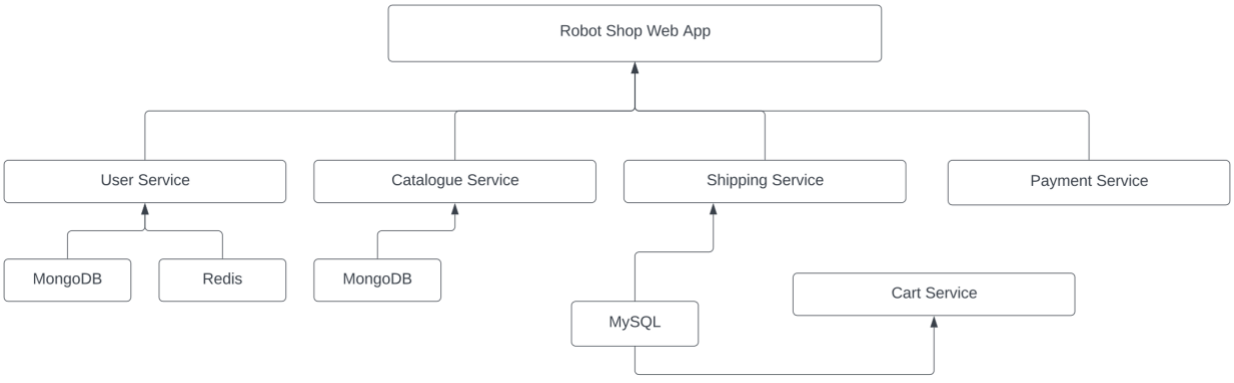


Figure 12 - illustration Architecture RobotShop

Ci-dessus, nous pouvons retrouver un schéma simplifié des différents services composant l'architecture en microservice de robot shop. Nous pouvons y observer les services suivants :

L'application web, qui est la partie exposée à l'utilisateur du système. C'est par le biais de ce service que les utilisateurs utilisent les fonctionnalités offertes par l'e-commerce. Cette partie de l'application communiquera avec les différentes API.

Le service client ; ce service offre une API qui est utilisée pour gérer les clients dans le système.

Le service catalogue ; ce service offre une API qui est utilisée pour gérer les catalogues de l'e-commerce.

Le service d'envoi ; ce service offre une API qui est utilisée gérer les envois de l'e-commerce.

Le service de payement ; ce service offre une API qui est utilisée gérer les payements de l'e-commerce.

Le service du panier ; ce service offre une API qui est utilisée gérer les paniers de l'e-commerce.

4.4 Analyse

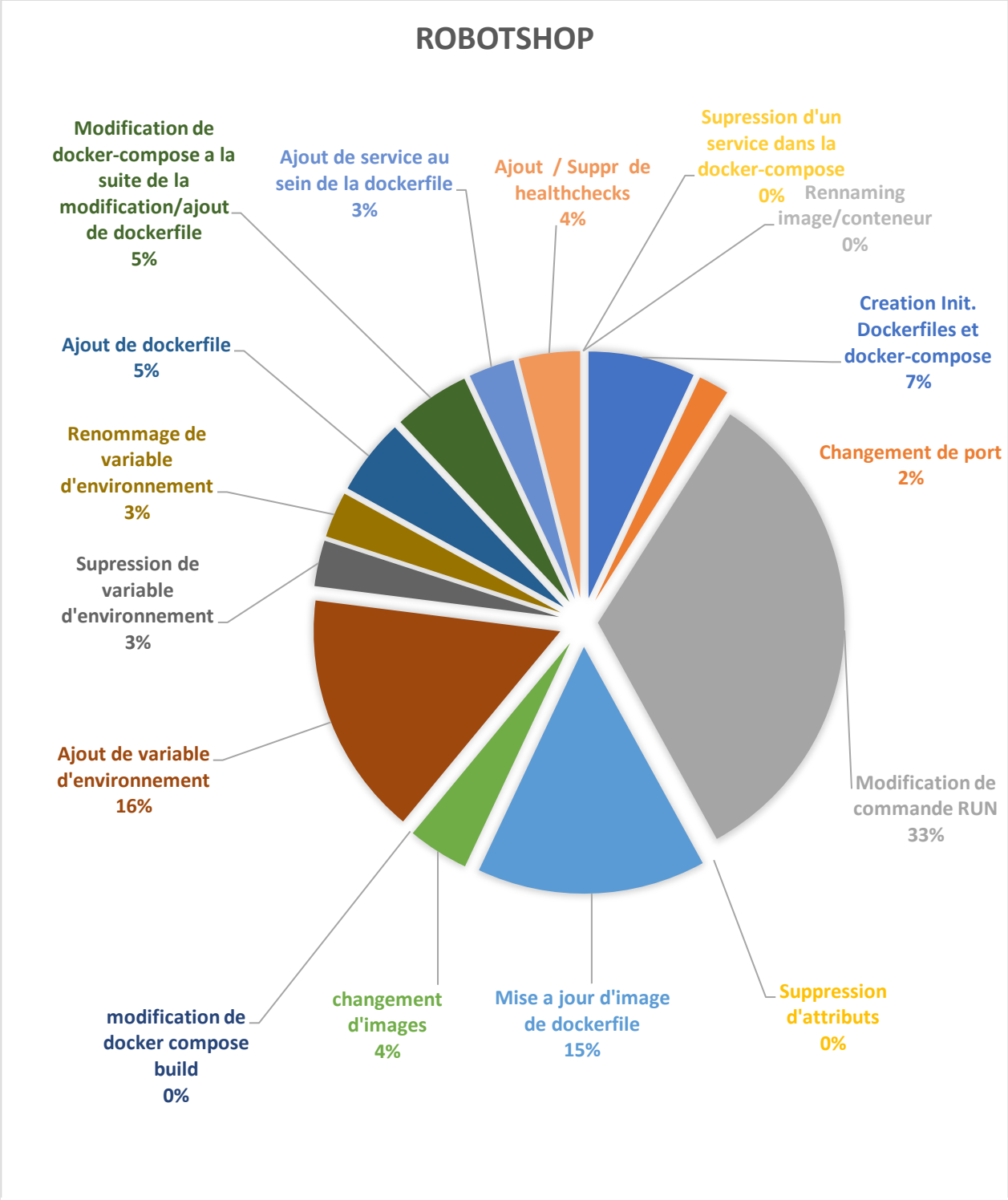


Figure 13 - Vue globale des modifications de robot-shop pour Docker

4.4.1 Création initiale de Dockerfiles et Docker-compose

Nous pouvons noter une absence d'information partielle au niveau de la création initiale des Dockerfiles et Docker-compose au sein du projet. En effet, dès le premier commit du projet, nous avons six Dockerfiles et un Docker-compose à notre disposition. Nous pouvons remarquer que ces Dockerfiles sont hétéroclites au niveau des images utilisées ainsi qu'au niveau des langages d'implémentations et de la persistance des données. En effet, nous avons une image MongoDB, une image java, une image Nginx ainsi que trois images node.js.

Nous pouvons noter que bien qu'il y ait une disparité au niveau des images des Dockerfiles, ces derniers exposent tous leur port 8080 vers l'extérieur. De plus, nous remarquons que les Dockerfiles disposant de l'image Node disposent aussi de la même configuration, même version d'image Node, même processus d'installation de dépendances et même exécution de commande

Le fichier Docker-compose, quant à lui, rassemblera les services au sein d'un seul fichier afin de grouper notre application multi-conteneur au démarrage. Ce dernier établi les dépendances entre les différents conteneurs.

Pour ce point nous avons pu observer un manque partiel d'information quant à la mise en place des artéfacts de déploiement étudié qui ont été créés avant d'être répertoriés au sein de l'outil de gestion de versions. Toutes informations sur la création des Dockerfiles avant ce point ne sont donc pas accessibles par nos soins. Nous avons aussi pu observer la mise en place de l'organisation des dépendances entre les différents conteneurs ; ici le service web semble être le point de contact des différents conteneurs disposant d'une logique métier, ce dernier dépends des services catalogue et utilisateur. De plus, bien qu'il y ait des langages d'implémentations disparates, les Dockerfiles partageant la même image semblent aussi partager la même configuration.

4.4.2 Changements de ports

Au niveau des changements de ports, nous observons peu de modifications au long de ce projet. Nous notons cependant que les changements de ports ayant lieu peuvent être catégorisés de deux manières.

La première catégorie est l'ajout d'un changement de port à la suite d'une modification d'image. En effet, nous pouvons par exemple observer l'ajout du port 15672 :15672 à la suite du changement d'image concernant le passage de rabbitmq à rabbitmq :3.7-alpine management. Le port introduit correspond à l'addition de l'interface graphique de l'outil de management de rabbitmq pouvant être accéder à l'url correspondant au format suivant : http://{node-hostname} : 15672/. Ce motif d'addition de ce port spécifique se répétera pour toute image de RabbitMQ passant à la version management, cela est dû à la configuration spécifique de l'image RabbitMQ management.

La deuxième catégorie concerne la suppression des attributs d'exposition de port dans le Docker-compose, afin de comprendre la suppression des ports de le Docker-compose. Nous devons d'abord nous attarder sur la différence entre l'attribut port du Docker-compose et l'attribut expose des Dockerfiles.

L'attribut port permet d'exposer le port spécifié a l'hôte, permettant ainsi à ce dernier d'y accéder. À contrario, l'attribut expose des Dockerfiles permet d'exposer le port uniquement dans le conteneur, le rendant, de fait, accessible uniquement au sein des conteneurs.

La suppression des ports du Docker-compose nous permet donc de supprimer l'exposition à l'hôte et d'isoler les communications entre les conteneurs.

De manière générale, nous pouvons observer que les changements de ports régulent l'exposition aux différents services et à l'hôte comme démontré par la suppression des ports. Nous observons aussi que la configuration des ports nous permet de supporter la configuration d'images afin d'accéder à certains outils de cette dernière comme démontré avec le passage à l'image rabbitmq :3.7-alpine-management.

4.4.3 Modification de scripts

Nous pouvons noter plusieurs modifications au niveau des scripts des Dockerfiles et ce, tout au long du projet à intervalles fréquents.

Nous constatons que ces modifications n'ont pas lieu de manière groupées mais semble être mise en place afin d'adapter l'image du service en question à l'ajout de fonctionnalités ou de dépendances du dit service : par exemple, lors de l'ajout de commandes RUN au niveau du service de ratings, nous avons eu un ajout de trois commandes RUN afin de mettre en place les commandes nécessaires à l'accord de permission afin

que le canevas logiciel Symphony fonctionne correctement, ou encore pour ajouter la fonctionnalité du profilage dans le service de ratings.

Nous observons aussi que certaines modifications ont lieu au sein des commandes RUN et COPY pour faciliter la lisibilité et la compréhension de la Dockerfile et du code, comme les effets de bords liées au renommage d'un scripts précédemment importé comme nous pouvons le voir au niveau du script run.sh qui a été renommé en entrypoint.sh par convention dans la Dockerfile du service load-gen.

D'autres modifications telles que les suppressions de scripts ont lieu lorsqu'une dépendance ou un artefact logiciel n'est plus utilisé. Nous pouvons alors observer cela dans la Dockerfile du service web lors de la suppression du script njs config.

Nous pouvons établir que les raisons de modification des commandes RUN et COPY liées aux scripts des Dockerfiles peuvent être multiples. Cependant, elles semblent introduites en vue de supporter la configuration de l'image et des dépendances liées au service en question et ne semblent donc pas provoquer de changements d'architecture. La suppression de configurations non utilisée dans la Dockerfile, par exemple, ne provoque pas d'effets de bord dans l'implémentation de la logique métier (Par exemple pour le script njs config). Ces commandes semblent permettre de configurer donc correctement les dépendances de l'image afin de pouvoir créer un conteneur ayant toutes les configurations nécessaires à l'exécution du service lors du déploiement de cet artéfact.

4.4.4 Mise à jour d'images et changements d'images

Nous avons pu observer plusieurs mise à jours tout au long du projet et ainsi remarquer que certaines mises à jour ont été opérées par groupement, notamment au niveau des services cart, user et catalogue qui ont tous la même image de départ (Node v8) ; leur mise à jour à Node v10, c'est donc faite de manière uniforme et cette stratégie de mise à jour permet de garder un ensemble cohérent au niveau des conteneurs utilisant les mêmes environnements tout en permettant de réduire les problèmes de dépendances (l'installation de la même dépendance d'un service à l'autre peut s'effectuer de manière uniforme, ce qui réduit aussi le problème de disponibilité des dépendances entre chaque versions).

Quant aux autres mises à jour d'images, celles-ci furent éparses au long du projet et ne semblent pas être corrélées l'une de l'autre.

Nous pouvons cependant noter un rétrogradage de l'image python3 à python2 dans le service de payement car cette version a provoqué des erreurs avec des dépendances non compatibles. Cette image passera tout de même à la version python3 une fois le problème de dépendance résolu.

De manière générale, nous observons que les mises à jour d'images s'effectuent de manière régulière et de manière groupée lorsque cela est possible comme pour les services cart, user et catalogue. Nous suspectons que la raison des mises à jour est que ces dernières nous permettent d'obtenir les différents correctifs appliqués aux versions les plus récentes tel que des correctifs importants liés à la sécurité (nous pouvons ici facilement penser à un conteneur utilisant une image log4j pour gérer ses logs qui aurait pu subir un exploit Log4shell) ou un correctif ayant attrait à la performance.

4.4.5 Variables d'environnements

Nous avons recensé une vingtaine de modification ayant attrait aux variables d'environnement au long de l'analyse de ce projet.

Nous avons remarqué dans un premier temps que les variables d'environnement peuvent avoir un impact majeur selon leur utilisation comme, par exemple, dans la docker-compose.yaml ou nous pouvons activer l'option End User Monitoring (EUM) par l'ajout de variables d'environnement ou encore au niveau des services cart , catalogue et profile où la variable d'environnement INSTANA_AUTO_PROFILE se charge de l'activation automatique du profilage de l'application au démarrage de l'application ; ces éléments ont pour impact de nous offrir une flexibilité simpliste au niveau de la configuration de nos conteneurs.

De plus, nous avons remarqué que les variables d'environnements nous permettent d'obtenir une flexibilité de configuration au niveau des images elles-mêmes et de leurs sources, comme lors de l'interpolation des informations des images pertinentes au registry dans le Docker-compose. Cela permet également une configuration classique telle que la spécification de l'hôte de base de données dans un service comme présent dans le service de shipping.

Globalement, nous pouvons remarquer que les applications de l'usage de variables d'environnement au sein de la configuration des Dockerfile peux être variée, passant de la configuration d'outils d'installation de dépendance à des éléments facilitant l'activation de certaines fonctionnalités comme nous avons pu le voir avec le profilage d'Instana. Cet usage des variables d'environnements nous permet donc alors de

configurer l'artefact de déploiement en y passant les variables d'environnement a l'exécution de l'application dans le conteneur.

4.4.6 Ajouts de Dockerfiles et modifications de Docker-compose

L'ajout de Dockerfiles ainsi que les modifications liées au Docker-compose représentent environ 20% de la totalité des modifications du projet, en faisant alors une part de changements non négligeable étant donné leur proportion.

Nous avons pu observer que l'ajout de Dockerfile semble se répercuter dans les Docker-compose au niveau de leurs déclaration dans le champ des services.

Les Dockerfiles, quant à elles, semblent supporter l'environnement dans lequel les artefacts logiciel, en l'occurrence les services, seront exécutés.

Les deux constatations précédentes sont représentées par l'ajout du service de payement au sein de l'architecture. Cette addition s'est illustrée, dans un premier temps, par la création du Dockerfile du service (image servant à la création du conteneur), et ensuite par sa déclaration au sein du Docker-compose puis son ajout en tant que dépendance du service web (permettant à l'outil compose d'organiser les conteneurs entre eux).

De manière générale nous avons pu constater qu'une part des modifications de Docker-compose sont reliées aux ajouts de Dockerfiles. Les Docker-compose permettent ainsi la déclaration au sein de l'architecture globale et permet de spécifier les relations de dépendances entre les services. Une autre part de modification concerne les changements dus à la configuration des Dockerfiles, comme nous avons pu l'observer avec l'ajout de port pour supporter le changement d'image de certains services à rabbitmq-alpine-management.

4.5 Conclusion

Lors de l'analyse du projet robot-shop, nous avons observé dès le départ que nous disposions d'un projet avec des langages de programmation hétéroclites. Nous avons alors pensé de prime abord que cette disparité se répercuterait au sein de la configuration des conteneurs à savoir des Dockerfiles. Cette supposition c'est avéré confirmée en partie du fait de l'observation que les services disposant d'un même langage d'implémentation, disposent aussi d'une configuration similaire au niveau de leur Dockerfile, comme au niveau de l'image utilisée et évoluent ensemble de manière groupée.

Nous avons aussi pu remarquer que la gestion des ports nous permet d'isoler les conteneurs de l'hôte, mais aussi d'exposer les services natifs à un conteneur. Les changements de ports permettent donc de gérer les communications du conteneur vers l'extérieur mais aussi du conteneur en lui-même. Cette dimension ne semble pas avoir d'effet de bord concernant le déploiement du conteneur, mais est tout de même important lorsque l'on considère d'autres dimensions telle que la télémétrie du logiciel déployé, par exemple nous pouvons imaginer une configuration de port défectueuse correspondant à la page de l'outil de télémétrie pour certain service ce qui entrainerai l'incapacité d'obtenir les informations de télémétrie.

Nous nous attendions à ce que la gestion des dépendances et la configuration des images passe par l'exécution de scripts et de variables d'environnement ce qui a pu s'avérer être le cas au sein de notre projet, donnant un rôle de configuration du conteneur aux commandes et aux scripts permettant ainsi d'exécuter le logiciel au sein de ce dernier.

Au niveau des modifications d'images, nous avons déceler que ces dernières peuvent être reliées à une facilité de configuration ou encore à une mise à jour de l'image supportant le langage d'implémentation comme avec les changements d'images node.js. Ces mises à jour semblent être motivées premièrement par une nécessité au niveau du logiciel, ce qui se traduit par ce changement afin que l'environnement d'exécution du logiciel soit au plus proche que celui du déploiement.

De plus nous avons pu observer que le Docker-compose est un élément central de la composition des services en un ensemble cohésif et permet d'établir les déclarations de nos Dockerfiles ainsi qu'une relation de hiérarchisation entre les services, ce qui semble concorder avec notre intuition première.

La Docker-compose nous a donc ici permis donc de contextualiser la relation entre différents conteneurs ainsi que d'avoir un aperçu sur leur hiérarchisation.

Nous avons donc pu de manière globale observer au travers du projet robot-shop, que le rôle des Dockerfiles à attrait a la configuration de la plateforme d'exécution du logiciel et que la manière de déployer un conteneur ne semble pas changer.

CHAPITRE 5: Présentation et analyse de PitStop

Dans ce chapitre nous allons tout d'abord introduire le projet PitStop, et nous nommerons ainsi les technologies utilisées au sein de ce projet, de même que l'architecture mise en place. Enfin, nous conduirons une analyse selon les critères définis au préalable dans notre méthodologie d'analyse.

5.1 Introduction

Le projet PitStop est un exemple d'application basé sur un système de gestion de garage ; dans le cas présent, un garage fictif. L'objectif de ce projet est de démontrer plusieurs concepts d'architecture logicielle tels que les microservices, CQRS, Domain Driven Design (DDD) et l'utilisation de certaines technologies telles que Docker, Kubernetes, Istio et Linkerd.

L'application PitStop prend en charge les tâches quotidiennes des employées et doit offrir les fonctionnalités suivantes : la gestion des clients, la gestion des véhicules appartenant aux clients ainsi que la gestion du planning de l'atelier.

5.2 Technologies

Concernant les technologies utilisées au sein du projet PitStop, nous retrouvons un Canvas logiciel unique comme outil d'implémentation : .NET. Au niveau de la persistance de données, nous retrouvons MongoDB, MySQL ainsi que Redis. Nous avons aussi une utilisation de Kubernetes afin de mettre en place l'orchestration de conteneurs ainsi que Docker pour la création des images. Nous avons enfin Istio et Linkerd comme maillage de service[17].

[17] https://en.wikipedia.org/wiki/Service_mesh

5.3 Architecture

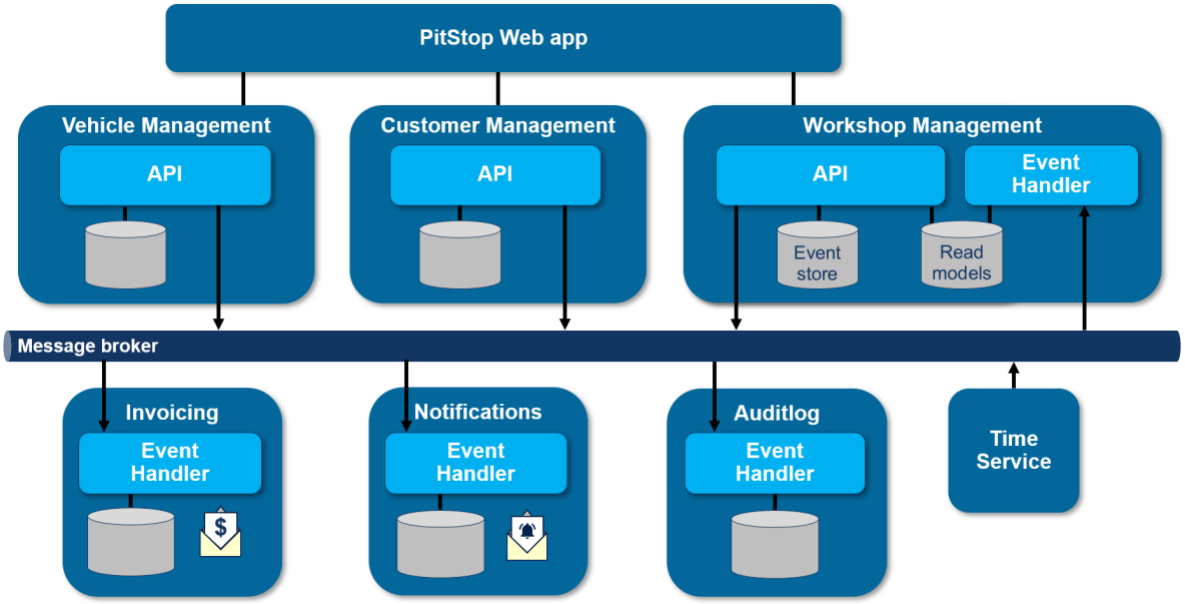


Figure 14 - Architecture PitStop [2]

Nous pouvons retrouver le diagramme d'architecture de l'application en figure 14, et y observer les services suivants :

L'application web, qui est la partie exposée à l'utilisateur du système. C'est par le biais de ce service que les utilisateurs peuvent gérer les clients, les véhicules et la planification de l'atelier. Cette partie de l'application communiquera avec les différentes API.

Le service de gestion des clients ; ce service offre une API qui est utilisée pour gérer les clients dans le système.

Le service de gestion des véhicules ; ce service offre une API qui est utilisée pour gérer les véhicules dans le système.

Le service de gestion de workshops qui expose une API servant à gérer les workshops ainsi qu'un gestionnaire d'évènement.

Le gestionnaire de notification ; ce service envoie une notification à chaque client à une tâche de maintenance planifiée le jour en cours. Il n'offre aucune API.

Le service de facturation ; ce service crée une facture pour tous les travaux de maintenance terminés et non facturés.

Le service temps est un service lorsqu'une certaine période s'est écoulée.

Le service de journal d'audit ; ce service récupère tous les événements du courtier de message et les stocke afin de permettre une référence ultérieure.

5.4 Analyse

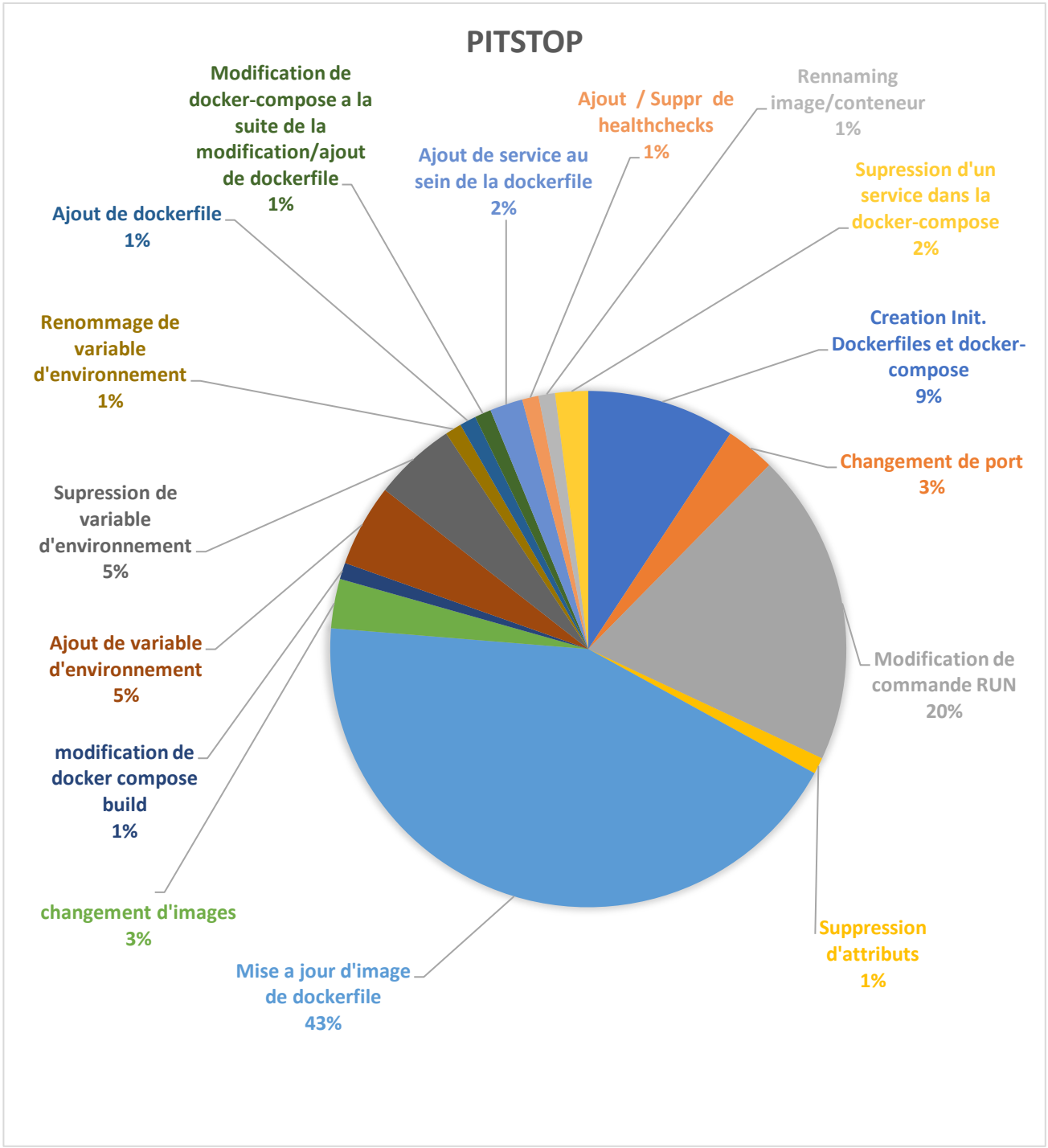


Figure 15 - Vue globale des modifications de PitStop pour Docker

5.4.1 Création initiale de Dockerfile et Docker-compose

Nous notons une absence d'information au niveau de la création initiale de Dockerfile et Docker-compose dans ce projet. En effet, dès le commencement du projet, ces derniers ont déjà été mis en place lors du commit d'initialisation et nous apercevons un commentaire laissant à penser qu'une grande partie du projet a été développée avant d'être migrée sur le dépôt de code de GitHub.

Concernant l'apparition des fichiers Docker nous observons une similarité frappante au niveau de tous les Dockerfiles : en effet, ces derniers ne diffèrent majoritairement que du fait de leur ENTRYPOINT. Cela s'explique de manière relativement simple, car nous observons que la technologie utilisée au niveau de l'implémentation est la même pour tous les Dockerfiles, à savoir ASP .NET CORE. Les Dockerfiles ont, par conséquent la même configuration.

Utiliser le même langage d'implémentation pour son architecture en microservices peut s'avérer judicieux et être un avantage puisque, en ayant une cohérence au niveau des technologies utilisées entre les différents microservices, nous pouvons simplement nous assurer d'une cohérence plus élevée au niveau de notre processus de déploiement.

Nous avons pu observer un manque de contextualisation par rapport à la création des Dockerfiles au sein de ce projet. En effet, ces derniers ont été créés avant d'être répertoriés au sein de l'outil de gestion de version toutes informations sur la création des Dockerfiles et, avant ce point, ne sont donc pas accessibles par nos soins. Nous avons aussi pu remarquer que les services du projet PitStop utilisent tous le même langage d'implémentation et par conséquent ont des Dockerfiles très similaires car le tout service nécessite la même version du même langage afin de s'exécuter. Dans le cas de notre projet cette similitude permet de partager certains aspects liés à la configuration des conteneurs.

5.4.2 Changements de ports

Nous avons remarqué deux changements de ports à dans les Dockerfiles du projet. Ces deux changements de ports ont respectivement lieu au niveau du service de l'application web et au niveau du service SQL et ont toutes deux eu pour la même raison technique la non-disponibilité du port associé.

En effet dans le service web, le port exposé de manière première fut le port 6000. Or, ce dernier est bloqué par Chrome pour des raisons de sécurité, puisque le port 6000 fait effectivement partie d'une liste de port bloqués par le navigateur Chrome afin d'éviter à Chrome d'être utilisé comme un proxy ouvert vers un réseau sécurisé, nous protégeant ainsi de potentielles attaques.

Au niveau du service SQL, le port originellement exposé a été le port 1433 qui est le port TCP exposé par défaut au niveau de SQL, le changement de port à 1434 aurait été motivé par la volonté d'éviter des conflits potentiels avec une version SQL préalablement installée.

Nous constatons globalement que les changements de port ayant eu lieu au sein du projet PitStop ont tous deux comme objectifs d'éviter des conflits par rapport aux services externes qui seraient déployés. C'est-à-dire entre le conteneur et l'extérieur du conteneur, mais aussi au sein du conteneur en lui-même.

5.4.3 Modification de commande RUN/COPY (Scripts)

Nous pouvons remarquer de multiples changements de commandes RUN et COPY tout au long de notre projet, et nous décomptons ainsi une vingtaine de changements liés à ces dernières modifications.

À la suite de l'analyse de ces modifications, nous pouvons établir que les changements de commandes RUN et COPY au sein du projet PitStop ont lieu de manière groupée et pouvons dénoter 18 changements groupés. Cela peut s'expliquer par la similarité de la configuration des Dockerfiles qui supportent les mêmes versions de Canevas logiciel.

Nous pouvons aussi noter dans le cas du projet PitStop que toutes les commandes ayant eu lieu de manière groupée ont lieu à propos de la commande : *dotnet restore*, qui est une commande permettant de restaurer les dépendances et les outils d'un projet .NET Core. Cette dernière commande est exécutée avec l'option -s permettant de spécifier une source, dans notre cas la source est un document JSON hébergé en ligne. Ce fichier hébergé aurait donc pu être modifié sans pour autant apparaitre comme tel au niveau de l'outil de gestion de version.

Nous pouvons également observer, du fait de l'importance de la commande dotnet restore, que les Dockerfiles partagent en plus de l'image et de la configuration Docker, les mêmes dépendances et outils.

Enfin, une autre modification a lieu au niveau des commandes RUN du service web, consistant en l'installation des dépendances situées dans le package. JSON ainsi que l'utilisation de GULP ; Nous pouvons donc établir à la suite de cette analyse que pour le projet PitStop, la grande partie des utilisations de ces commandes consistent en l'installation de dépendances projet.

Globalement, nous avons pu observer que les modifications concernant les scripts ont eu lieu dans un esprit de configuration du conteneur afin de permettre d'installer des dépendances ou de configurer l'image initiale afin que l'application puisse s'exécuter comme attendu lors de son déploiement. Un exemple de ce point serait au niveau des installation des dépendances situées dans les fichiers package.json ou encore des commandes dotnet restore qui vont permettre d'installer la dépendance nécessaire à l'exécution du logiciel au sein du conteneur.

5.4.4 Mises à jour d'images et changements d'images

La mise à jour d'images est l'une des catégories sur laquelle nous pouvons remarquer une prédominance au niveau du nombre de modifications au cours de l'évolution de l'architecture de déploiement.

Nous pouvons donc observer au long du projet PitStop environ quarante-deux (42) mises à jour d'image Docker au sein des neuf Dockerfiles. À la suite de l'analyse des logs liée aux mises à jour d'image Docker, nous pouvons faire deux constats :

Premièrement, les mises à jour arrivent de manière successive et incrémentale. Nous notons que nous n'assistons pas à un passage à une version inférieure, cela semble nous indiquer que lors des multiples déploiements aucune erreur n'a eu lieu du passage d'une version a supérieure.

Deuxièmement, nous pouvons observer que les mises à jour arrivent par groupements uniformes : plusieurs Dockerfiles passent à une version supérieure équivalente au même instant. Cela s'explique par la similarité de configuration de tous les Dockerfiles, dans la mesure où les Dockerfiles utilisent tous la même image d'origine celle de .NET core, facilitant ainsi les passages à une version supérieure.

Au niveau des changements d'images, nous remarquons d'entrée que ces derniers sont beaucoup moins fréquents. Nous avons deux changements d'images concernant uniquement la source de l'image ce qui

n’a pas provoqué d’effet de bord ; un autre concerne un changement d’image pour une autre passant de RabbitMQ à RabbitMQ alpine management, dont la raison la plus probable concernant ce changement est d’obtenir nativement et sans configuration additionnelle le plug-in management de RabbitMQ qui permet de fournir une API basée sur le protocole HTTP pour la gestion et la surveillance des clusters RabbitMQ, ainsi qu’une interface utilisateur et un CLI**.**

A quelques exceptions près, nous pouvons noter que les modifications concernant les mises à jour d’images ont lieu afin de pouvoir suivre avec les évolutions du Canevas logiciel utilisé. Cela a pour effet d’améliorer la maintenabilité ainsi que la sécurité du produit logiciel. Nous observons aussi que les changements concernant le remplacement d’image ont principalement eu lieu afin de faciliter la configuration de l’image au niveau des commandes exécutées dans la Dockerfile comme le remplacement de rabbitmq à rabbitmq alpine management, permettant de simplifier le processus de configuration de l’image.

5.4.5 Variables d’environnement

Les modifications de variables d’environnements (Ajout, Suppression, Modifications) semblent être l’un des points de modification important au sein des Dockerfiles du projet PitStop comptant approximativement onze modifications.

Les modifications des variables d’environnements, bien que diverses, semblent être établies pour une raison bien précise : le partage d’éléments de configuration au sein des Dockerfiles.

Notons cet usage de variables d’environnement n’est pas propre à l’utilisation de Docker, mais est une utilisation partagée dans le cadre de développement d’application logicielle.

En résumé, nous pouvons observer que les modifications au niveau des différentes variables d’environnements répondent à un besoin de configuration de l’application, notamment au besoin de flexibilité de cette configuration.

5.4.6 Ajouts de Dockerfiles et modifications de Docker-compose

Lorsque nous nous attardons sur l'ajout de Dockerfiles au sein du projet de PitStop, nous remarquons un seul ajout de Dockerfile pour une passerelle d'API. Ce dernier sera supprimé par la suite dans le projet.

Nous constatons ici particulièrement les conséquences au niveau du changement de l'architecture de déploiement à la suite de l'ajout de ce Dockerfile. En effet, ce Dockerfile a été ajouté dans le but d'être le point de séparation vers trois autres services. Ce Dockerfile regroupe alors ces trois services à son niveau, permettant ainsi de spécifier la passerelle à la place de ces trois services. Cet ajout de Dockerfiles s'est donc répercuté au niveau du Docker-compose pour ces raisons. Aussi, lors de sa suppression, l'inverse s'est alors produit.

Globalement, lorsque nous observons le Docker-compose de ce projet, nous pouvons constater que certaines modifications au niveau des Dockerfiles entrainent des modifications au niveau du Docker-compose, notamment lors d'ajout et de surpression de services au sein du Docker-compose, mais aussi des dépendances au sein de ces services. L'ajout de Dockerfiles ou de services en dépendances change les liens de hiérarchisation au sein de l'application, ayant de ce fait un impact sur son architecture.

5.5 Conclusion

Lors de cette analyse nous nous attendions à avoir une configuration de Dockerfiles différentes l'une de l'autre malgré l'utilisation du même Canvas logiciel au niveau de l'implémentation. Cependant, nous avons remarqué une uniformité au niveau de la configuration des Dockerfiles, partageant les mêmes images jusqu'au mêmes dépendances. Cela est intéressant à constater au niveau du déploiement car cela permet de partager la configuration des conteneurs, permettant donc une meilleure maintenabilité de ces artéfacts.

Par notre précédente analyse du projet robot-shop, nous nous attendions à retrouver des mises à jour uniforme et groupées en ce qui concerne les Dockerfiles du fait de leur configuration similaire. Nous avons pu remarquer qu'effectivement les mises à jour au niveau des Dockerfiles se sont effectuées par

groupement, toujours en utilisant les mêmes modifications. Ce point semble être influé par les services en eux-mêmes, car l'implémentation dépends de la version du langage utilisé. La version d'un langage peut influer sur les fonctionnalités que ce dernier présente comme pour Java avec l'introduction des lambdas ou encore de la syntaxe tel que le passage de Python 2 à Python 3.

Nous nous attendions aussi à ce que les variables d'environnement jouent le même rôle que pour l'implémentation, c'est-à-dire, stocker des valeurs dynamiques ayant un rapport avec l'environnement ou la configuration du projet. De ce fait, nous avons pu remarquer que cette hypothèse c'est avéré correcte notamment par le partage des clés de configuration au sein du projet. Cela a notamment un attrait à la configuration du déploiement ou ces variables peuvent être spécifiées au sein du conteneur afin de changer de clé d'API selon l'environnement de déploiement (développement, production, tests).

Nous avons supposé que les changements de ports aient lieu au sein des Dockerfiles afin d'exposer correctement les services entre eux. Cependant nous avons remarqué qu'en plus de se charger correctement de l'exposition interservices, la configuration des ports se charge aussi de gérer les conflits potentiels avec la machine hôte, comme nous avons pu le voir avec les contraintes des ports MySQL et Chrome.

Enfin, nous avons pu observer que les Dockerfiles semblent supporter l'implémentation de la logique d'affaires, n'induisant pas de modification d'architecture. Toutefois, les modifications ayant lieu au niveau de la Docker-compose permettent de nous indiquer les changements ayant un attrait à la composition, la hiérarchisation ainsi qu'à la dépendance de ses conteneurs à un impact au niveau de l'architecture comme l'exemple de la passerelle d'API a pu nous le démontrer.

CHAPITRE 6 : Analyse comparative

6.1 Création initiale de Dockerfile et Docker-compose

À la suite de l'analyse de PitStop, nous avons constaté que les images du projet ont eu droit à une configuration très similaire ne différant majoritairement que par leur point d'entrée (entrypoint), cela est dû à la cohérence du Canvas technologique d'une part par l'image utilisée mais aussi à la structure similaire des services (commande COPY). Le projet PitStop usage d'un fichier Docker Compose afin de hiérarchiser la relation envers les différents conteneurs permettant d'exécuter l'application conteneurisé plus simplement.

D'un autre coté pour le projet RobotShop, nous avons observé six Dockerfiles dont trois Dockerfiles ayant comme image Node.js, nous avons remarqué que pour ces dernières qu'elles disposaient elles aussi de la même configuration. De même que pour PitStop, RobotShop fait usage d'un fichier Docker Compose afin de hiérarchiser la relation envers les différents conteneurs.

En somme, nous remarquons qu'au niveau de la création initiale des Dockerfiles, les Dockerfiles ayant la même image semblent dispose de la même configuration. De plus, les deux projets utilisent l'outil Compose de Docker ce qui va permettre de définir et d'exécuter l'application multi-conteneur (ici l'ensemble des services de l'architecture). Nous remarquons aussi que chaque service dispose de sa propre Dockerfile, donc de son propre conteneur lors de l'exécution. L'utilisation du Docker Compose nous permet alors de gérer l'organisation de l'application multi conteneur de manière plus simple quel que soit l'environnement (développement, production, …) mais aussi dans les pipelines d'intégration continue.

6.2 Changements de ports

Après l'analyse de PitStop, nous avons constaté deux modifications de ports afin d'éviter les conflits avec les environnements extérieurs tel que MySQL ainsi que Google Chrome.

Tandis que lors de l'analyse de RobotShop, nous avons eu plusieurs ajouts d'un port au sein des Dockerfiles afin de supporter un nouveau service dû au changement de l'image RabbitMQ à l'image RabbitMQ Management. Nous avons aussi dénoté des suppressions au sein des attributs ports du Docker Compose permettant d'isoler l'exposition des conteneurs Dockers.

En conclusion, il nous semble que les modifications liées à l'usage de ports semblent être utilisées de la même manière entre les deux projets c'est-à-dire à des fins de configuration de l'architecture réseau des services contenus dans l'architecture de façon qu'elle soit correcte et non conflictuelle avec l'environnement extérieur. De plus, l'exposition de ports, permet aussi d'exposer des outils à l'extérieur du conteneur comme l'interface graphique servant à la gestion de RabbitMQ. Lors du déploiement de l'application il est fondamental que l'architecture réseau ait bien été configurée au préalable afin de s'assurer que toute les parties composant l'architecture en microservice puisse communiquer entre elles. Les modifications des technologies utilisées dans l'architecture en microservice peuvent donc entrainer des modifications au niveau de l'exposition des ports des conteneurs.

6.3 Modification de scripts

Après avoir analysé le projet PitStop nous nous sommes intéressés aux changements au niveau des modifications de scripts groupés. Nous avons aussi pu noter que l'installation des dépendances pour nos conteneurs s'effectue par une source de dépendance distante qui semble cohérente au travers de tous les services. Cette cohérence au niveau de l'installation des dépendances et des modifications groupées est proche de celle rencontrée au niveau des mise à jour d'image comme nous le verrons dans la prochaine partie. Cette cohérence permet au projet PitStop d'obtenir une prévisibilité lors des changements de scripts sur l'ensemble de conteneurs.

A l'inverse, les modifications de scripts au sein de RobotShop sont plus disparates. Du fait d'une configuration d'image différente, les conteneurs reflètent un besoin de supporter la configuration de l'image attachée au service et, dans le cas de RobotShop, ces images varient grandement d'un service à l'autre.

En conclusion, les modifications de scripts au sein des deux projets jouent un rôle de support à la configuration de l'image nécessaire afin de supporter le service. Bien que le rôle joué par les modifications de scripts soit le même, la façon dont ces derniers sont amenés au sein des descripteurs de conteneurs est bien différente; du côté de PitStop les changements s'opèrent de manière groupée du fait de l'utilisation d'une configuration similaire au sein des images, tandis qu'au sein de RobotShop, à l'inverse, la différence d'image rend la configuration de chaque conteneur différent. Les changements de commandes liées aux scripts et ont donc pour rôle de s'assurer que le service puisse s'exécuter au sein du conteneur tout en conservant les fonctionnalités attendues par ce dernier. Cela a donc attrait à la configuration interne du conteneur mais ne semble pas affecter la manière dont le conteneur sera déployé.

6.4 Mise à jour d'images et changements d'images

À la suite de notre analyse de PitStop, nous avons remarqué au niveau des mises à jour d'image que ces derniers s'effectuées de manière incrémentale et de manière groupée. Comme nous avons pu observer précédemment, cela s'explique par une configuration d'image et une configuration de scripts similaire. De part cette cohérence, les changements peuvent s'opérer par groupements réduisant ainsi la friction aux modifications des descripteurs de conteneurs.

Du coté de RobotShop, les mise à jour d'images s'effectuent de manière non groupée, sauf concernant les images semblables tel qu'avec les images Node. Nous avons assisté à un downgrade lors de l'évolution du projet à la suite de dépendances non compatibles.

Nous pouvons aussi noter pour les deux projets, un changement de l'image concernant Rabbitmq, passant de rabbitmq à rabbitmq management afin de simplifier la configuration de ce dernier.

En conclusion, les mises à jour d'images semblent plus simples lorsque la configuration d'image entre les services est similaire. Les mises à jour et les changements d'images semblent jouer un rôle servant de base à la configuration du conteneur et donc à l'exécution du service.

6.5 Variables d'environnements

Ultérieurement à l'analyse de PitStop, nous avons observé que les variables d'environnements au sein du projet semblent véhiculer une intention de partage des variables au sein des Dockerfiles permettant une configuration uniforme.

Pour donner suite à l'analyse de RobotShop, nous avons constaté que les variables d'environnement sont aussi partagées et interpolées au sein des Dockerfiles et fichiers de configurations. Dans le cas de RobotShop, une variable d'environnement a aussi servi de condition à l'activation d'une fonctionnalité au sein de la Dockerfile.

En conclusion, au sein des deux projets les variables d'environnement jouent un rôle de variables partagées à des fins de configuration des Dockerfiles, entre autres, elles suivent une utilisation classique des variables d'environnement en génie logiciel : le découplage de la configuration. L'étude de ces deux projets, nous ont permis de déduire que les variables d'environnements jouent un rôle de configuration des Dockerfiles, permettant de spécifier l'environnement ou d'autres conditions rendant la Dockerfile configurable par l'extérieur. Ces variables peuvent être ainsi spécifiées lors du déploiement, comme lors de la création des pipelines de déploiement. L'usage des variables d'environnements semblent dans notre cas avoir plus attrait au logiciel, n'entrainant pas de modifications de l'architecture de déploiement.

6.6 Ajouts de Dockerfiles et modification de Docker-compose

Par la suite de l'étude de PitStop, nous avons noté l'ajout d'un service de passerelle d'API qui sera ensuite supprimée. Bien que temporaire, cet ajout a eu des conséquences intéressantes au niveau de la restructuration de la hiérarchisation des services au sein des fichiers Docker Compose. En effet, une passerelle d'API est principalement utilisée afin d'offrir un point d'entrée unique dans un système utilisant plusieurs services d'API.

Concernant le projet RobotShop, l'ajout des Dockerfiles se répercutent au niveau du Docker Compose où ils sont renseignés. L'ajout d'un service a une répercussion sur l'architecture de déploiement car le nouveau service devra être déployé à son tour. De plus, les changements au niveau de la hiérarchisation de conteneurs (depends_on) dans les Docker Compose traduit aussi un impact au sein de l'architecture de déploiement comme nous avons pu le voir lors de l'ajout de dépendances au niveau du service web.

En somme, nous pouvons déduire qu'au sein des projets PitStop et RobotShop que l'ajout d'un service entraine l'ajout d'un Dockerfile afin de spécifier la configuration du conteneur et donc entraine l'ajout d'un Dockerfile afin de spécifier la configuration du conteneur et par conséquent entraine l'ajout de l'artéfact de déploiement, mais cependant nous n'avons pas pu observer l'inverse au sein des deux projets, de fait, l'ajout d'un nouveau service semble motivé par la logique du domaine d'affaire plus que par un besoin d'architecture de déploiement.

6.7 Conclusion

Tous ces éléments semblent nous porter vers le fait qu'au sein des deux projets nous avons observé peu de modifications profondes de l'architecture de déploiement. En effet, comme nous l'a démontré l'analyse comparative, la majorité des changements de l'architecture de déploiement et des Dockerfiles ont été introduits à la suite de modifications de l'architecture en microservices, notamment au niveau de l'ajout et la suppression de services comme démontré notamment avec l'ajout de la passerelle d'API.

Les fichiers descripteurs de conteneurs ne semblent pas influer et impliquer de changements sur l'architecture logicielle, cependant l'inverse semble se produire. Nos Dockerfiles semblent supporter notre logiciel plus applicativement qu'au niveau du déploiement en s'assurant que le conteneur supporte d'une part l'exécution de l'application mais aussi sa bonne communication avec les autres services.

Ces points s'illustrent respectivement pour l'un lors de la modification de commandes COPY et RUN notamment lors de la copie du code source au sein du conteneur, l'installation des dépendances et la spécification du point d'entrée, et pour l'autre au niveau de la modification de l'exposition des ports comme nous l'avons vu au sein des deux projets afin que les services puissent communiquer entre eux correctement, exposer certains outils à l'extérieur du conteneur ainsi qu'éviter des conflits de ports.

Cela nous mène à conclure que les modifications au sein des fichiers descripteurs de conteneurs entrainent peu de modification profonde de l'architecture en microservices mais semblent supporter cette dernière en tant que support exécutif.

CHAPITRE 7 - Rétrospective

Dans ce chapitre établir la rétrospective concernant notre étude. Dans un premier temps nous aborderons les obstacles rencontrés ainsi que les solutions menées pour les dépasser. Nous allons ensuite développer les apprentissages que nous avons pu tirer de cette synthèse ainsi qu'enfin effectuer un rapprochement avec les cours dispensées au programme de génie logiciel avant de conclure

7.1 Obstacles et Solutions

L'un de premiers obstacles à notre analyse a été de trouvé sous quel angle mener notre étude, nous avons donc dû extraire dans un premier temps quelle technologie étudier au sein des treize projets du corpus. Pour ce faire nous avons lister toutes les technologies utilisées dans chaque projet du corpus, et nous avons filtré les technologies selon le groupe auquel ils appartiennent (implémentation , déploiement, persistances des données) puis parmi les technologies de déploiement , nous avons dû faire un choix, ce dernier s'est porté sur Docker, car il est commun à tous les projets , à une grammaire facilement compréhensible et est une technologie qui a gagné en popularité dans les dernières années changeant l'approche que nous prenons lors du déploiement.

Un autre obstacle que nous avons rencontré a été l'évolution de notre compréhension de la question de recherche au fil de l'étude, cela nous a poussé à devoir changer notre manière d'aborder la question. Effectivement, au fil du de l'analyse du logiciel développé, nous nous sommes rendu compte que les données obtenues à l'aide de ce dernier ne suffisaient pas afin de répondre à la question, le temps manquant pour développer une autre version du logiciel nous avons dû réfléchir et changer notre approche afin de pouvoir mener la synthèse à bout. Nous avons donc consciemment décidé de passer à une approche qualitative sur un échantillon plus petit, ce qui nous a permis de tirer une analyse comparative dans les temps impartis. Une amélioration dans notre processus concernant l'outil d'analyse aurait été de le mettre en application au plus vite possiblement sous forme de script dans le cadre d'une analyse afin de vérifier que les réponses obtenues sous satisfassent.

Finalement l'un des autres obstacles reliés au précédent a été la limite de connaissance initiale par rapport aux architectures en microservices et aux technologies de déploiement. Afin de mitiger à ce point, nous avons dû faire de la recherche en amont afin de contextualiser le sujet dans le domaine du génie logiciel.

7.2 Apprentissages

En prenant du recul, ce sujet de synthèse m'a permis d'approfondir les connaissances dispensées dans la maitrise de génie logiciel de l'UQAM, en particulier par rapport à deux cours, qui ont été celui de Principes et Applications de la conception de Logiciels, ainsi que celui de Réalisation et Maintenance de Logiciels.

Ces deux cours m'ont permis respectivement d'une part de replacer les origines des architectures en microservices et de connaitre les avantages et inconvénients de ces derniers de l'autre de renforcer mes acquis de programmation et d'utilisation des outils liées au génie logiciel.

De plus, cette synthèse m'a permis de me familiariser avec les technologies de containerisation, notamment avec l'outil Docker, ce qui autre les architectures en Microservices met permet d'être plus effectif et de participer à des discussions d'ingénierie au sein de mon équipe de développement.

Cette introduction au domaine de la recherche en génie logiciel, m'a également permis de renforcer mon esprit de synthèse et d'analyse me permettant de porter un regard critique et de comparer multiples technologies dans mon rôle d'ingénieur.

CONCLUSION

Notre objectif lors de ce projet de synthèse a été de trouver des motifs de changements au sein des fichiers descripteurs de conteneurs Docker afin d'étudier quels impacts ces changements peuvent t'ils avoir sur une architecture en microservices.

Nous avons pu explorer cette problématique sur un échantillon de treize architectures open source communément étudiées par la communauté scientifique, par la création d'un outil d'analyse quantitative, puis par l'établissement d'une méthodologie d'analyse qualitative et finalement en appliquant cette méthodologie à une sélection de deux projets du corpus d'étude nous permettant d'obtenir une taxonomie des changements liées aux Dockerfiles.

À la suite de notre analyse, nous n'avons pas réussi à repérer des motifs majeurs prouvant que des modifications de Dockerfiles entrainent des changements de l'architecture en microservices. Mais que les Dockerfiles servent de support aux exécutables et évoluent au fil de ces derniers.

Bien que nos conclusions à la suite de cette étude ne nous semblent pas généralisables à toutes les architectures en microservices disposant de Dockerfiles d'une part par la taille de l'échantillon des projets étudiées qui est trop petit pour en extraire des résultats généralisables, mais aussi d'une part car notre taxonomie des changements d'opérant au sein des Dockerfiles n'a pas bénéficié des retours et d'un regard critique d'une tierce partie.

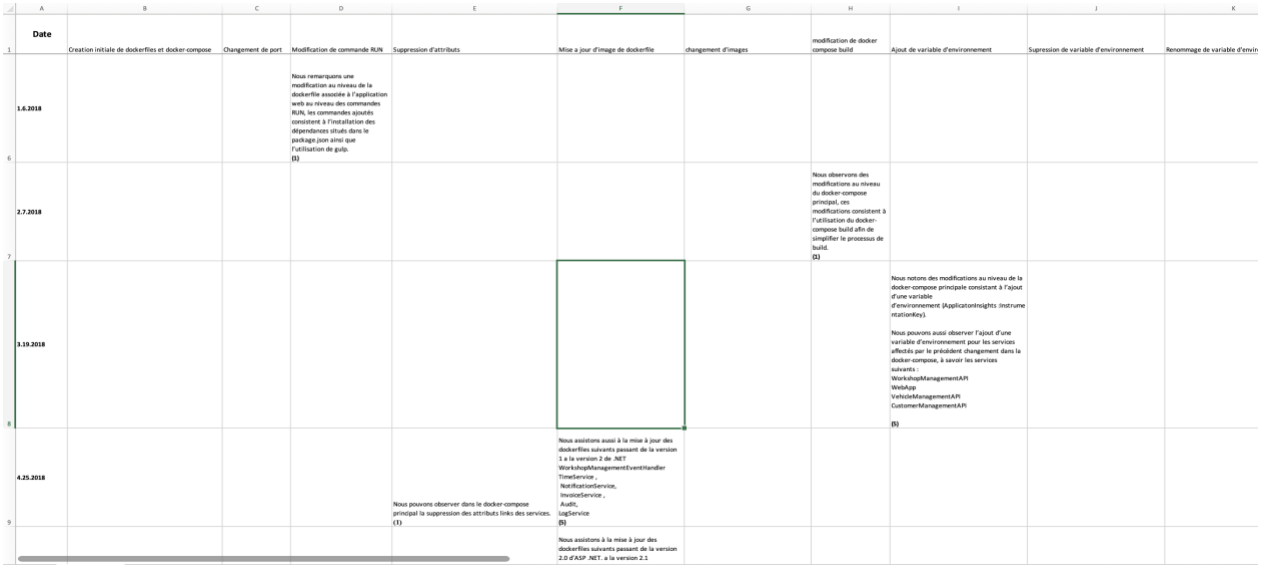
Afin d'approfondir cette question de recherche, nous pourrions élargir l'échantillon de dépôts de code étudiées afin d'obtenir une étude et une taxonomie plus représentative, ainsi que de discuter avec une tierce partie des résultats obtenus afin d'obtenir un regard critique et non biaisé permettant donc de réduire l'atteinte à la validité de cette étude. Une autre question de recherche qui nécessitera son propre approfondissement est l'impact des changements d'architecture sur les fichiers descripteurs de Dockerfiles.

En somme, ce projet de recherche m'a permis de familiariser sur le plan personnel et intellectuel avec une thématique de recherche au sein du domaine du génie logiciel. Ce projet m'a permis de consolider certains acquis dispensés lors de la maîtrise en génie logiciel tel que le cours d'architecture logiciel, mais aussi de

développer et d'étendre les connaissances liées à la conteneurisation et sur les architectures en microservices, me permettant désormais de pouvoir discuter de manière informée de ces sujets au sein de divers projets logiciels.

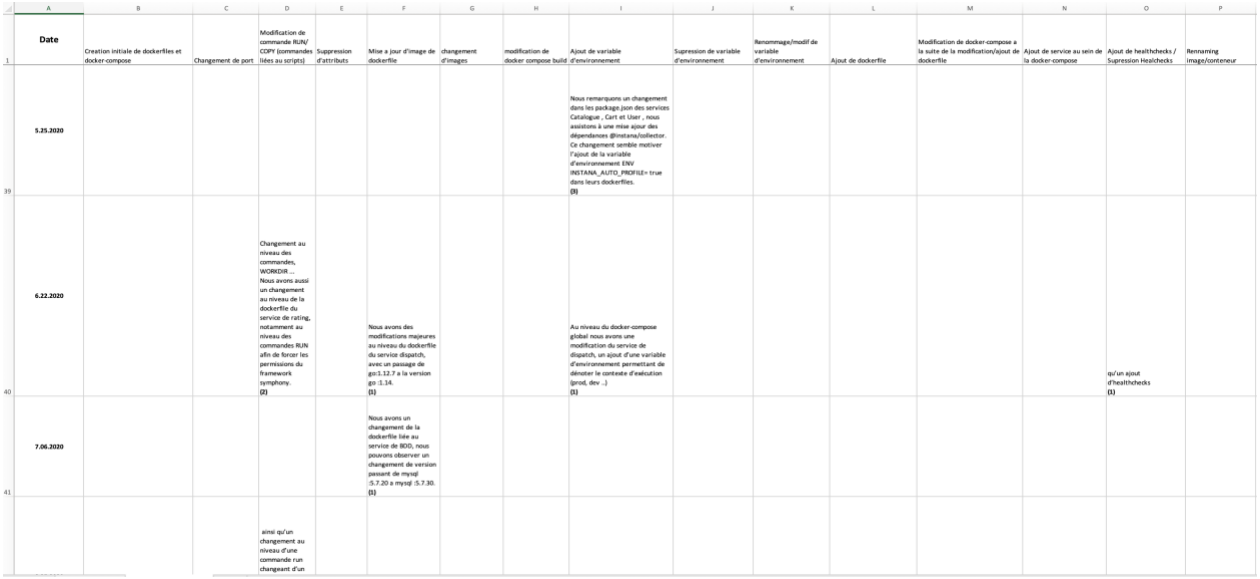
ANNEXE A

Document Logs PitStop et RobotShop



Date	Creation initiale de dockerfiles et docker-compose	Changement de port	Modification de commande RUN	Suppression d'attributs	Mise a jour d'image de dockerfile	changement d'images	modification de docker compose build	Ajout de variable d'environnement	Supression de variable d'environnement	Renommage de variable d'envir
1.6.2018			Nous remarquons une modification au niveau de la dockerfile associée à l'application web au niveau des commandes RUN, les commandes ajoutés consistent à l'installation des dépendances situés dans le package.json ainsi que l'utilisation de gulp. (1)							
2.7.2018							Nous observons des modifications au niveau du docker-compose principal, ces modifications consistent à l'utilisation du docker-compose build afin de simplifier le processus de build. (1)			
3.19.2018								Nous notons des modifications au niveau de la docker-compose principale consistant à l'ajout d'une variable d'environnement (ApplicatonInsights :Instrumentationkey). Nous pouvons aussi observer l'ajout d'une variable d'environnement pour les services affectés par le précédent changement dans la docker-compose, à savoir les services suivants : WorkshopManagementAPI WebApp VehicleManagementAPI CustomerManagementAPI (5)		
4.25.2018				Nous pouvons observer dans le docker-compose principal la suppression des attributs links des services. (1)	Nous assistons aussi à la mise à jour des dockerfiles suivants passant de la version 1 a la version 2 de .NET WorkshopManagementEventHandler TimeService , NotificationService, InvoiceService , Audit, LogService (5)					
					Nous assistons à la mise à jour des dockerfiles suivants passant de la version 2.0 d'ASP .NET. a la version 2.1					

Figure 16 - Extrait document logs pitstop[18]



Date	Creation initiale de dockerfiles et docker-compose	Changement de port	Modification de commande RUN/ COPY (commandes liées au scripts)	Suppression d'attributs	Mise a jour d'image de dockerfile	changement d'images	modification de docker compose build	Ajout de variable d'environnement	Supression de variable d'environnement	Renommage/modif de variable d'environnement	Ajout de dockerfile	Modification de docker-compose a la suite de la modification/ajout de dockerfile	Ajout de service au sein de la docker-compose	Ajout de healthchecks / Supression Healchecks	Renaming image/conteneur
5.25.2020								Nous remarquons un changement dans les package.json des services Catalogue , Cart et User , nous assistons à une mise ajour des dépendances @instana/collector. Ce changement semble motiver l'ajout de la variable d'environnement ENV INSTANA_AUTO_PROFILE= true dans leurs dockerfiles. (3)							
6.22.2020			Changement au niveau des commandes, WORKDIR ... Nous avons aussi un changement au niveau de la dockerfile du service de rating, notamment au niveau des commandes RUN afin de forcer les permissions du framework symphony. (2)		Nous avons des modifications majeures au niveau du dockerfile du service dispatch, avec un passage de go:1.12.7 a la version go :1.14. (1)			Au niveau du docker-compose global nous avons une modification du service de dispatch, un ajout d'une variable d'environnement permettant de dénoter le contexte d'exécution (prod, dev ..) (1)						qu'un ajout d'healthchecks (1)	
7.06.2020					Nous avons un changement de la dockerfile liée au service de BDD, nous pouvons observer un changement de version passant de mysql :5.7.20 a mysql :5.7.30. (1)										
			ainsi qu'un changement au niveau d'une commande run changeant d'un												

Figure 17 - Extrait document logs RobotShop [19]

[18] https://github.com/ekiriano/csv_categorisation_microservices

[19] https://github.com/ekiriano/csv_categorisation_microservices

RÉFÉRENCES

[1] Sterling (2020) *Microservices vs Monolithic* - https://sterling.com/microservices-vs-monolithic/

[2] Edwin Van Wijk (2019) *PitStop Solution Architecture-*
https://github.com/EdwinVW/pitstop/wiki/Solution%20Architecture

BIBLIOGRAPHIE

[1] Inconnu (2019), L'architecture de microservices au service des plateformes d'expérience https://www.sqli.com/fr-fr/insights-news/blog/larchitecture-de-microservices-au-service-des-plateformes-dexperience

[2] Inconnu (2021), *Docker (logiciel)*, https://fr.wikipedia.org/wiki/Docker_(logiciel)

[3] Inconnu (2021), *Kubernetes*, https://en.wikipedia.org/wiki/Kubernetes

[4] Inconnu (2021), *Node.js*, https://en.wikipedia.org/wiki/Node.js

[5] Inconnu (2021), *React (JavaScript Library)*, https://en.wikipedia.org/wiki/React_(JavaScript_library)

[6] Inconnu (2021), *Express.js*, https://en.wikipedia.org/wiki/Express.js

[7] Sam, N 2021, *Building Microservices: Designing Fine-Grained Systems* ,2nd Edition, O'Reilly Media, USA

[8] Jeff, N & Stephen, K 2019, *Docker in Action*, Manning Publications Co, USA