Institut **S**upérieur d'**I**nformatique, de
Modélisation et de leurs **A**pplications

Campus des Cézeaux
1 rue de la Chébarde
TSA 60125
CS 60026
63178 Aubières CEDEX

Puzle

Rapport de stage de 3^{eme} année

Filière : Génie Logiciel

Conception d'un générateur d'environnement de production

Présenté par **Maxime SIBELLAS**.

Tuteur ISIMA :
Loïc Yon

Référent ISIMA :
Claude Mazel

Soutenance :
à 11h le 10.03.2017

Remerciements

Je remercie Cédric Charière–Fiedler de m'avoir embarqué dans cette aventure.
Je remercie Loïc Yon, Christophe Duhamel, Claude Mazel et l'ISIMA de m'avoir autorisé à initier ce projet durant ma deuxième année et de nous avoir soutenu dans nos démarches entrepreneuriales.

Table des figures

Résumé

Dans le cadre d'un projet entrepreneurial, l'équipe de développeurs dont je fais partie a soumis le besoin d'une boîte à outils simplifiant et accélérant la création de contenu sous forme de preuve de concept ou prototype. C'est dans ce contexte que j'ai décidé de concevoir une suite de logiciels open-source permettant le déploiement rapide et simplifié d'outils nécessaires tant à la création technique qu'à l'organisation et la transmission d'information. Cela comprend la conception d'une structure logicielle définie par une configuration précise, la création des programmes pour l'entretien des outils déployés (sauvegardes et restaurations) ainsi qu'un système de webhook permettant de configurer un centre de notifications. Je me suis aidé pour cela de la technologie Docker qui permet le déploiement rapide d'environnements prêts à l'emploi sur système Unix et des langages de script Python et Bash .
Mots-clés : base de données, centre de notifications, webhook, Docker, Unix, Python, Bash

Abstract

In the context of a professionnal project, the technical team I was part of submits the need for a toolkit simplifying and accelerating content creation like proof of concept or prototype. That is why I decided to design an open-source software suit to deploy quickly and easily all tools necessary both for technical creation and organisation and information sharing. This means the creation of a software structure which can be defined by a specific configuration, the creation of programms for tools maintenances (backup and restore) and a configurable notifications center based on webhooks. To realize it, I used the fashionable technology Docker which allow deploying ready-to-use environments on Unix system and Python and Bash scripting languages.
Key words : database, notification center, webhook, Docker, Unix, Python, Bash

Table des matières

Introduction

De nos jours, l'entrepreneuriat est une option de plus en plus envisagée par les jeunes. C'est un moyen de *challenger* ses idées, de les confronter à la réalité et pourquoi pas de les réaliser. La plupart de ces idées apparaissent lors d'un concours : hackathon, StartUp week-end ... Ils se déroulent généralement sur 48 heures ce qui laisse peu de temps à l'organisation : choix des outils et des médias de communication... C'est une expérience facinante mais peu aisée pour les néophytes.

C'est lors de nos premiers pas dans cet univers que nous est apparu l'avantage d'avoir le plus rapidement possible tout notre attirail d'outils prêt à l'emploi et ceci dans un seul but : prendre une longueur d'avance sur nos concurrents. Nous nous confrontions à des choix techniques concernant notre organisation : il nous fallait choisir un certain nombre de logiciels pour partager les informations, d'autres pour stocker nos documents et encore d'autres pour nous partager les tâches... Nous nous sommes vite confrontés à un problème : à utiliser plein d'outils, on en oublie où ils se trouvent et où partent nos informations. Nous cherchions donc à concevoir un écosystème d'outils que nous pourrions gérer seuls, du déploiement à l'entretien, et dont nous connaîtrions chacune des entrées/sorties d'information.

C'est dans ce contexte que j'ai commencé à développer *Summoner*, notre invocateur d'environnement de travail. Pour cela, nous verrons tout d'abord quels étaient nos besoins et comment le choix des outils s'est effectué, dans une seconde partie nous expliciterons les éléments techniques utilisés par *Summoner* et ses *minions* (i.e. la solution et ses outils) et leurs implémentations et nous conclurons sur les perspectives d'évolutions et d'amélioration du travail effectué.

Chapitre 1

L'initiation d'un projet

Au fil de l'année, des événements, des concours et des rencontres, le besoin d'outils divers et variés, déployés rapidement et simplement s'est fait sentir : lors des hackathons, des projets personnels, etc. C'est donc de ce besoin qu'est née l'idée qui m'a guidé tout au long de ce projet : la création d'un *Startup-kit* permettant à n'importe qui d'obtenir un environnement *ready-to-use* en quelques clics.

Le développement du concept s'est fait en parallèle du lancement et de l'évolution de l'idée que nous avons proposée devant le jury du Startup week-end 2016 et qui a remporté le 1er prix. Les réflexions et choix explicités dans la suite ont donc été effectués en rapport avec nos besoins.

1.1 Composition de l'écosystème

Lors de la création de projet, que ce soit lors d'un concours sur un week end ou bien lors de la création d'une startup, une bonne organisation est la clé de la réussite. Cette organisation se base sur plusieurs critères : une méthodologie simple et compréhensible de tous et un partage des informations structuré et rapide. Pour cela plusieurs solutions existent : une rigueur sans faille de la part de tout les participants ou bien des outils clairs, faciles d'utilisation et bien utilisés.

1.2 Composition en stack

Notre environnement de travail est divisé en plusieurs stacks : des groupes d'outils communicant entre eux. Trois stacks principales ont été définie (Figure 1.1) :

- **La stack utilitaire** qui comprend tout les outils d'organisation, de stockage et de communication. C'est ce groupe d'outils qui constituera notre StartUp kit.
- **La stack technique** qui comprend les outils de développement, d'intégration continue et de monitoring de l'environnement.
- **L'application** est elle-même constituée de plusieurs outils. Cette architecture prend pour exemple une application suivant une architecture micro-services communicant grâce à un *message broker*. Cependant, elle est applicable à tout autre application.

Nous nous intéresserons ici à la composition des stacks utilitaire et technique.

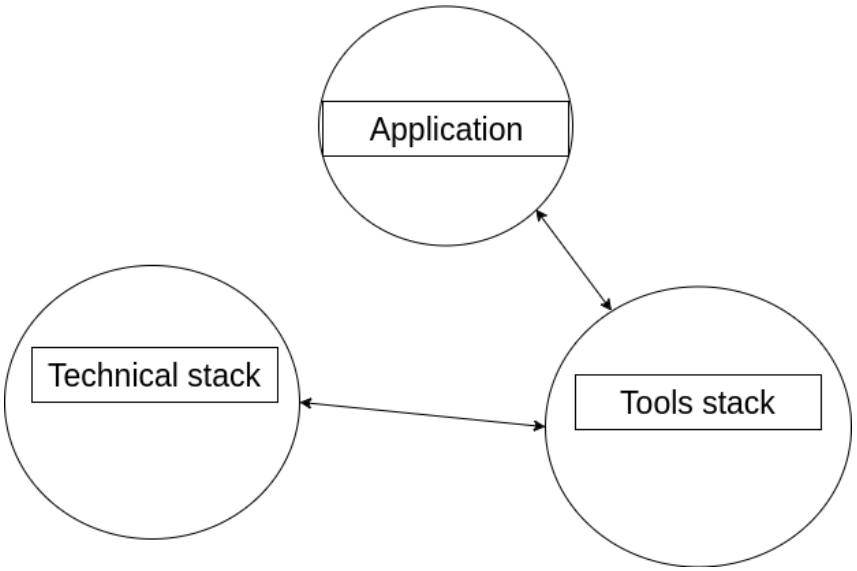

FIGURE 1.1 – Définition des stacks

1.2.1 Les indispensables

Stockage

Conscients de la problématique de l'utilisation de nos données personnelles dans un cadre ultra compétitif, le premier besoin était de posséder notre propre solution de stockage afin de contrôler l'accès à ces données. Nous avons donc rendu possible l'installation d'une solution de stockage de type **Nextcloud**[1]. Cet outil donne la possibilité de fonctionner sous différents type de base de données, ce qui permet une flexibilité dans le choix des technologies utilisées.

Visibilité

Le deuxième besoin d'un projet est sa visibilité : avoir une vitrine publique pour communiquer, faire parler de soi, etc. Pour cela, de nombreux moyens existent mais le plus simple et plus répandu est le très célèbre **Wordpress**[2]. Nous avons donc ajouté la possibilité de déployer un site propulsé par Wordpress prêt à l'usage.

Dans le même esprit mais plus sobre, nous avons fait la découverte d'une plateforme de *blogging* nommée **Ghost**[3]. Ghost est tout aussi personnalisable que Wordpress mais est plus léger et plus simple d'entretien.

1.2.2 S'organiser : Agile, SCRUM & Co

De nos jours, de nombreux principes d'organisation managériale ont vu le jour. Parmi ceux là, certains sont en vogue dans les grandes entreprises et sont applicables aux projets qui doivent évoluer rapidement et sur lesquels de nombreuses hypothèses (techniques mais pas uniquement) doivent être vérifiées. Ces principes sont fondés sur la méthodologie dite agile.

1. Site web officiel de Nextcloud - `www.nextcloud.com`
2. Site web officiel de Wordpress - `https://fr.wordpress.com/`
3. Site web officiel Ghost - `https://ghost.org/fr/`

La méthode agile est le découpage d'un projet en étapes unitaires afin de définir des périodes d'itération sur ces étapes pour les valider le plus rapidement possible. Cela permet de valider un fonctionnalité, un outil, une hypothèse technique quelconque ou même, dans notre contexte, une hypothèse business ou de marché le plus vite possible et en minimisant la perte de temps et donc d'argent. De nombreux concepts découlent de cette méthode : le SCRUM, le Lean ... Tous ont leurs avantages et leurs inconvénients mais tous se structurent de la même manière : un découpage des taches de la façon la plus unitaire possible, une évaluation de l'importance de la tache (souvent avec l'attribution d'une note) et la constitution d'un *backlog*. On définit alors un *sprint* (une itération) à partir des tâches du backlog. À chaque fin de sprint, un retour sur le travail effectué est réalisé afin d'améliorer les suivants et de choisir la meilleure orientation du travail possible pour le projet.

Cette méthodologie permet d'orienter le développement de nos prototypes selon l'avis des utilisateurs : dès qu'une fonctionnalité est développée, nous la confrontons à l'utilisateur. Si il valide la fonctionnalité (par son usage, le fait qu'elle soit pratique, etc), nous la gardons, sinon nous la supprimons et redéfinissons un sprint pour développer une nouvelle fonctionnalité.

Pourquoi redéfinir ici cette méthode managériale ? Tout simplement parce que nous avons choisi de suivre ce principe pour avancer dans notre projet : cela nous offre une meilleure visibilité sur son futur. Il a donc fallu choisir l'outil correspondant.

Kanban

Le *Kanban* permet une organisation simple et efficace. Il est constitué de tableaux (*boards*) dans lesquels sont créées des listes, à la façon des *TodoLists*.

Notre choix s'est porté sur deux outils :

Taiga [4] - Un outil de travail de groupe fournissant la possibilité d'un support dédié aux sprints SCRUM et d'un Kanban. C'est un logiciel open-source développé en Python

et en CofeeScript. Ses versions conteneurisées sont malheureusement encore peu fiables et son adaptation à notre environnement est encore en cours.

Wekan [5] - Kanban open-source dans la lignée de *Trello* [6], c'est un outil simple que nous avons choisi car accessible aux néophytes. À la différence de Trello, il peut être déployé sur un serveur privé.

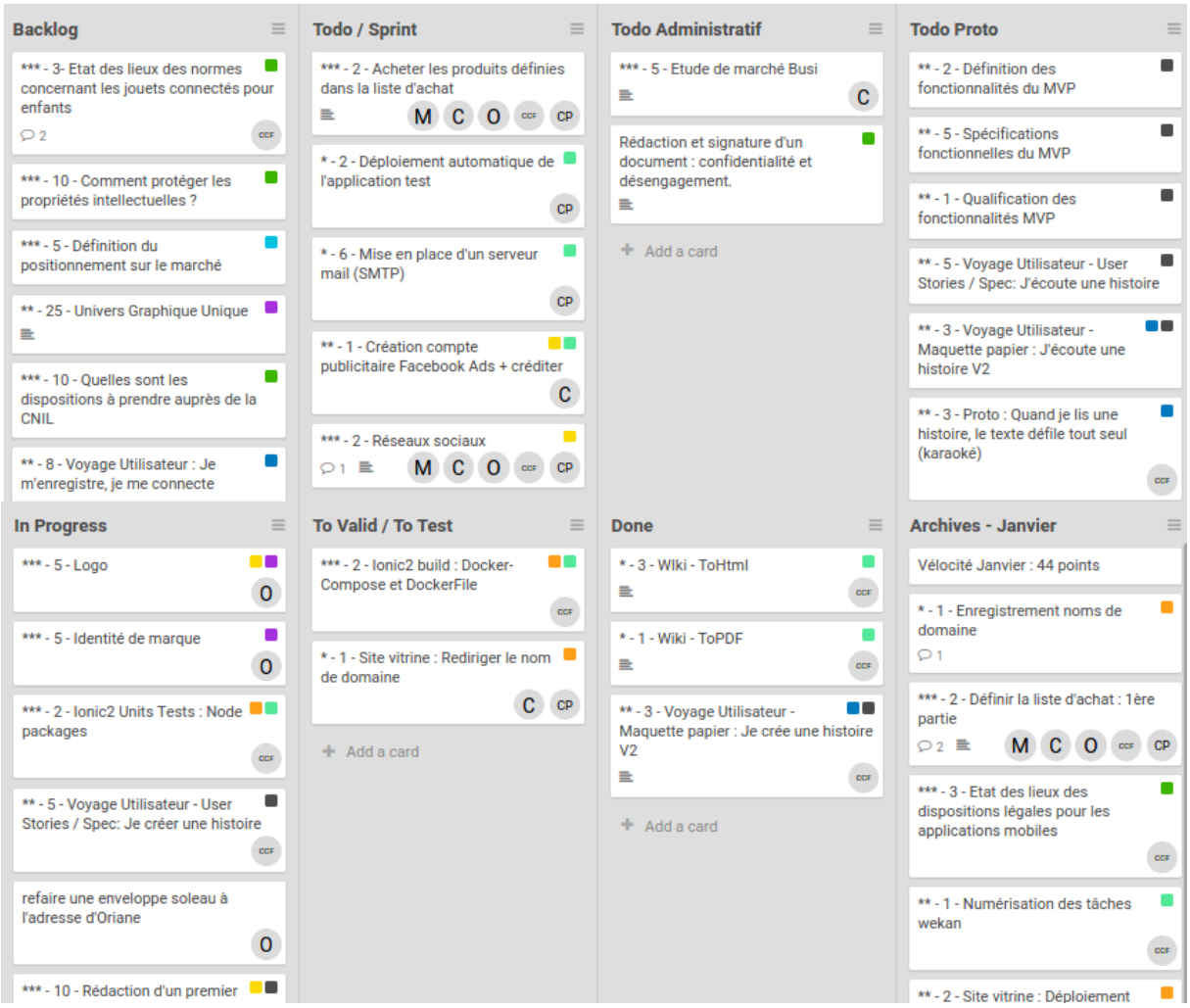

FIGURE 1.2 – Mise en place de la méthode agile via un Kanban

Chaque post-it du tableau contient une tâche dont l'importance est représentée par une valeur. Chaque tâche est attribuée à un ou plusieurs membres. Toutes ces taches

6. Site web officiel de Trello - `https://trello.com/`

ont été définies lors du premier sprint. Le tableau est divisé en plusieurs listes pour une meilleure visibilité des domaines auquels appartiennent les taches.

1.2.3 Simplifier la communication

De nombreux outils mettent à disposition des canaux de communications efficaces : *Facebook Messenger*, *Google Hangout*, etc. Ces outils ne permettent cependant que des flux uniques : toutes les informations se retrouvent mélangées. La plateforme de discussion *Slack*[7] est aujourd'hui la plus utilisée pour les travaux en équipe, autant en entreprise qu'en dehors. Cependant, Slack est une solution payante et donc impossible à utiliser dans le cadre de ce projet. Nous nous sommes donc rabattus sur la version open-source mais néanmoins (presque) aussi complète : *Mattermost*[8]. La principale différence entre ces deux applications est l'existence d'un client Slack installable directement sur l'ordinateur alors que Mattermost n'est accessible que par un navigateur. Ces deux outils possèdent en outre les mêmes services : création de canaux de discussion, modifications des droits sur chacun des canaux, intégration de Webhooks, etc.

1.2.4 Des webhook pour un écosystème vivant

Le but de l'écosystème défini est d'accéder à l'information dès qu'elle est partagée par un membre de l'équipe. Pour cela, notre idée est de relier tous ces outils en un même centre de contrôle et de notifications.

Un Webhook est un système de requêtes HTTP permettant de récupérer des informations venant d'une API ou de recevoir des flux afin de les traiter puis de les retransmettre dans un format valide pour la destination suivante (Figure 1.3). Grâce à cela, il est possible d'effectuer des actions suite au déclenchement d'un événement précis.

Un exemple :

7. Site web officiel de Slack - `https://slack.com`
8. Site web officiel de Mattermost - `https://about.mattermost.com`

- Lorsqu'un document est déposé sur notre outil de stockage, un événement est déclenché qui transmet l'information à un Webhook qui lui même postera un message dans le canal de discussion approprié.

En généralisant ce mécanisme, il est possible de pouvoir définir des "recettes" de webhook : un enchaînement d'événements définis par un événement source et une suite d'actions à effectuer. Cela permet de définir son propre centre de notifications afin de ne rien perdre des communications du groupe.

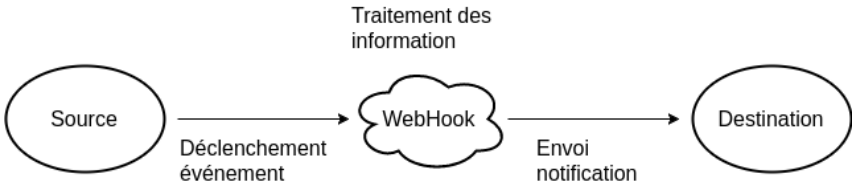

FIGURE 1.3 – Fonctionnement d'un webhook

Une fois relier à un canal de Mattermost, les messages envoyés seront archivés dans la discussion associée et tout les membres autorisés pourront les consulter(Figure 1.4).

FIGURE 1.4 – Message reçu par Mattermost

1.3 Les outils techniques

Lors du développement d'une application, certains outils sont nécessaires pour nous simplifier la vie ou bien pour nous donner des indicateurs sur le code source. Parmi eux, les outils de *versionning* de code source.

1.3.1 Versionning & CI

Notre choix s'est arrêté sur la solution **Gitlab** utilisant le système de versionnage Git. Ce produit est derrière son concurrent **Github** en terme fonctionnalités mais possède l'avantage de proposer une installation sur notre propre serveur afin de nous assurer le contrôle des accès à nos projets.

Le grand intérêt de cette outil est qu'il permet de définir un *workflow* d'intégration continue au sein d'un même outil. Plus besoin de plusieurs outils pour s'occuper du versionning, des tests puis du déploiement : Gitlab s'occupe de tout. Il utilise pour cela Docker et ses *registries* que nous verrons plus tard.

Workflow

Le workflow d'intégration continue constitue la suite d'actions effectuées lors d'une mise à jour du code source d'une application.

Déroulement du workflow :

1. Modification du code source : on effectue un commit et on l'envoie sur le serveur
2. Gitlab déclenche alors le déroulement des différents tests : tests unitaires, d'intégrations, fonctionnels ...
   (a) Si les tests réussissent, un tag est créé pour sauvegarder la nouvelle version de l'application
   (b) Le déploiement dans l'environnement de production est alors déclenché.
   (c) Sinon une notification est envoyée par mail pour prévenir de l'échec des tests

Ce workflow peut être connecté à un Webhook permettant de notifier l'équipe en direct du résultat des différentes étapes du workflow.

La configuration d'un workflow permet un gain de temps par automatisation et le déploiement automatique permet de toujours avoir la dernière version de l'application déployée.

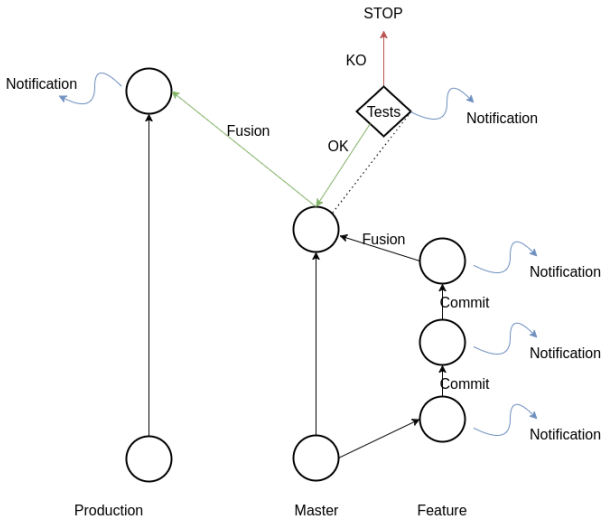

FIGURE 1.5 – Workflow d'intégration continue avec notifications

1.3.2 Qualité du code

Un des critères de qualité d'une application est la qualité de son code. Il est important d'avoir des indicateurs à ce sujet afin de prévenir des effets de bord provoqués par une trop grande réplication de code ou utilisation de procédés obsolètes. Parmi les nombreux outils existants, nous avons choisi d'utiliser *SonarQube*[9].

Sonar permet de définir des règles de programmation complémentaires aux *best practices* du langage et de leur donner des degrés d'importance (Warning, error, etc). Il analyse le code, relève les infractions aux règles, indique la couverture du code par les tests et possède encore plus d'indicateurs (Figure 1.6).

9. Plateforme de test SonarQube - `https://sonarqube.com/projects`

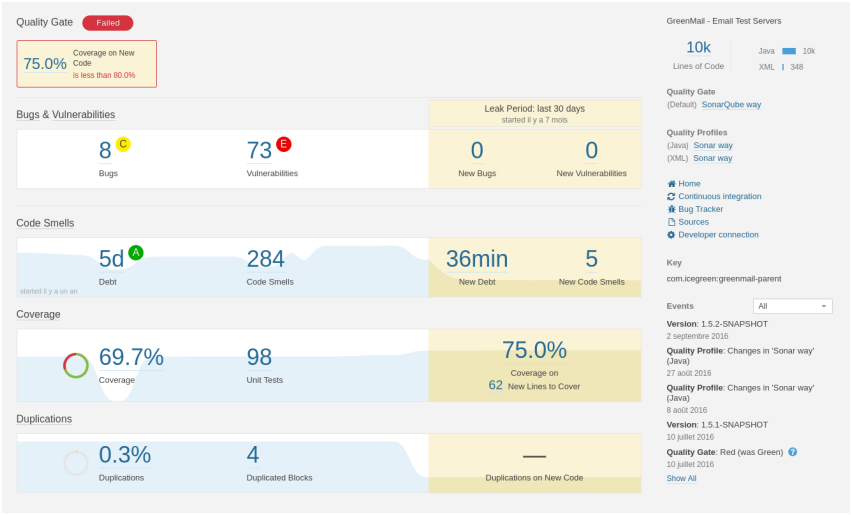

FIGURE 1.6 – Tableau de bord de Sonar

Il liste ensuite les infractions aux règles en notifiant son degré d'importance (Figure 1.7).

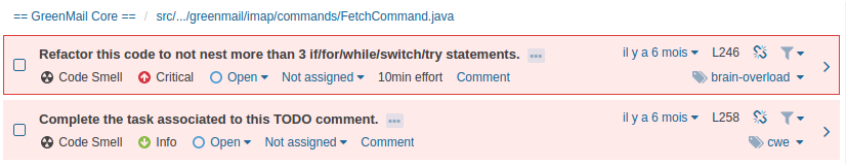

FIGURE 1.7 – Liste des infractions

1.3.3 Loggers & readers

Une application produit de nombreuses lignes de logs par jour comprenant les informations des événements significatifs, les comportement à risques, bugs etc. Une recherche dans ces logs devient fastidieuse lorsque ce fichier atteint les milliers de lignes. C'est pour cela que nous avons choisit de déployer la *stack Elastic*[10].

Cette stack comprend quatres outils complémentaires : **Beats**, **Logstash**, **Elastic-Search** et **Kibana**. Son fonctionnement est le suivant (Figure 1.8) :

10. Solution Elastic - `https://www.elastic.co/fr/`

- Beats collecte des informations sur des systèmes distants et abstraient leur transmission à une ou plusieurs destinations distantes.
- Logstash est un outil dédié à l'extraction d'informations depuis des sources de données hétérogènes, leur transformation et leur transmission.
- ElasticSearch permet de réaliser un moteur de recherche. Il fournit un service d'interrogation et gestion des données s'appuyant sur une architecture REST.
- Kibana consomme l'API d'ElasticSearch et expose une interface graphique simple d'utilisation permettant à l'utilisateur de monitorer en temps réel l'état des fichiers log.

Ainsi nos logs sont lus, étudiés et affichés dans un environnement agréable.

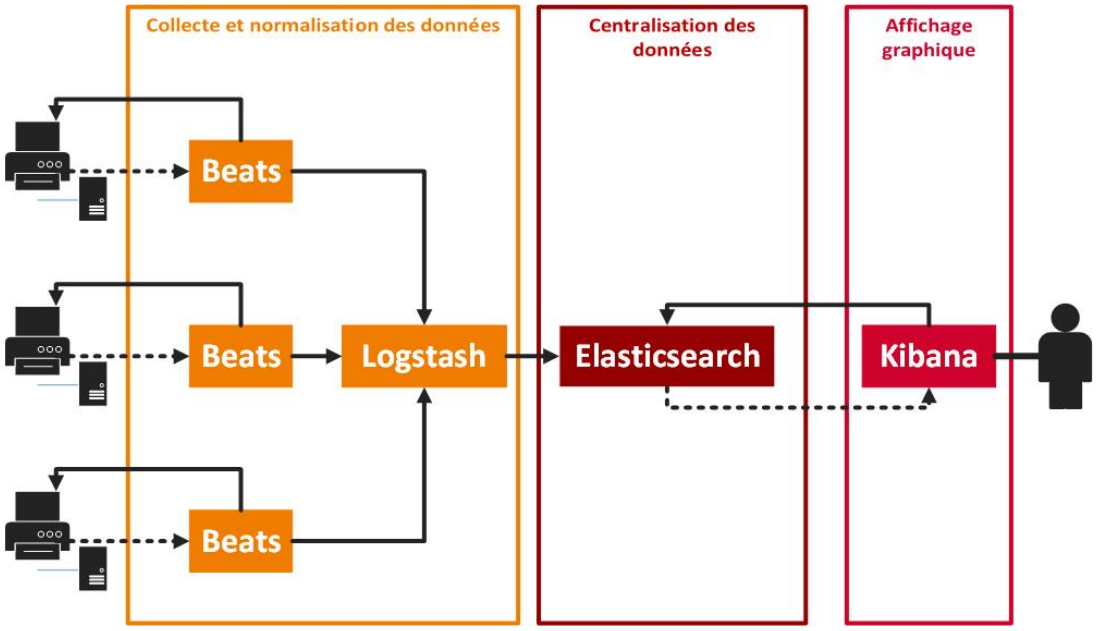

FIGURE 1.8 – Structure d'une stack Elastic [2]

Chapitre 2

Développement de Summoner

2.1 Technologie de conteneurisation

Avec l'essor du *Cloud Computing*, des services type IaaS, SaaS, PaaS, les applications sont de plus en plus empaquetées dans des machines virtuelles ou des conteneurs. Cela permet de faciliter certaines manipulations : lorsqu'un problème survient, il suffit de détruire la machine ou le conteneur et de le redémarrer ; grâce aux hyperviseurs, la gestion d'un grand nombre d'instances de ces applications est très simple : les mise à jours peuvent être appliquées à toutes les instances d'un clic, idem pour les patchs, etc.

La machine virtuelle est un outil répandu chez le développeur : il permet d'héberger un système et de simuler un ordinateur. Cela permet d'avoir un bac à sable pour tester de nouvelles distributions, pour faire des expériences qui pourraient mal tourner ou pour installer des bibliothèques sans qu'elles occupent de la place sur notre machine.

Néanmoins la création d'une machine virtuelle demande déjà une quantité de ressources conséquente : il faut allouer la mémoire vive, la mémoire physique (ou la création d'un disque virtuel), installer l'OS (et donc les bibliothèques nécessaires) ... Pratiquement, la machine virtuelle se duplique facilement et permet d'héberger plusieurs instances d'une même application sans problème. Là où la VM est peu pratique, c'est dans le partage des fichiers : la paramétrisation d'un dossier partagé entre plusieurs VMs est plutôt fastidieuse si on utilise pas d'hyperviseur.

Le conteneur, quant à lui, est plus léger et s'ajoute à l'OS actuellement installé sur la machine. Il crée une sur-couche au système en utilisant le *file system* de la machine. Ainsi la mémoire vive et physique est partagé avec l'hôte. Un conteneur peut faire tourner une distribution différente de celle de la machine mais partagera le maximum de contenu : le système de fichier, le driver de stockage, les bibliothèques etc. Cela rend le conteneur plus rapide de mise en place et d'exécution.
Cependant, ce partage de file system est un avantage comme un inconvénient de ces conteneurs. En effet, sur les systèmes Unix, à chaque processus lancé sur la machine correspond un fichier possédant un *inode* (noeuds d'index) qui contient les méta-data du fichier (localisation sur le disque, possesseur, etc). Le partage du *file system* avec le conteneur sous entend que les processus lancés sur le conteneur possèdent un inode sur la machine. Or, le nombre d'inode sur une même machine est limité : si le conteneur lance trop de processus, il peut saturer la machine hôte.

Ces différences de fonctionnement sont décrites dans la figure 2.1.

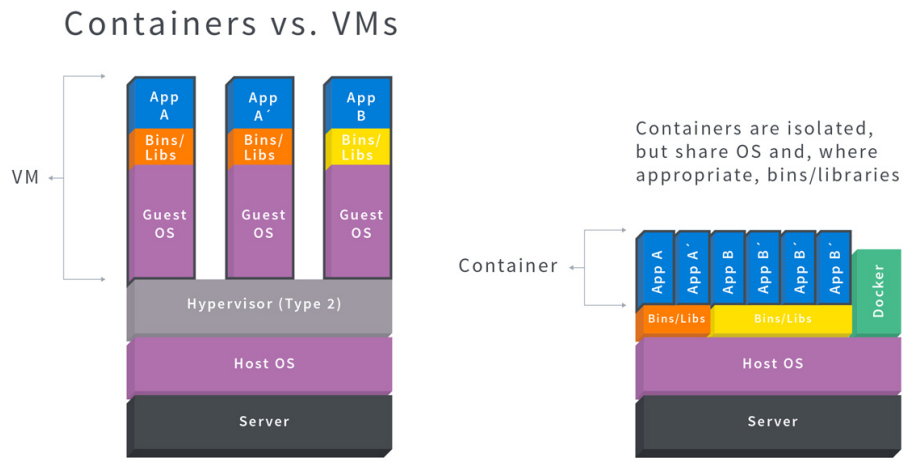

FIGURE 2.1 – Comparaison conteneur et machine virtuelle - Source : Docker Inc

2.1.1 Docker, la baleine porte-conteneurs

Docker est une solution de virtualisation par conteneur. Pour l'utiliser, il faut installer *Docker engine* qui est principalement constitué d'un démon [1], d'une API REST permettant de communiquer avec le démon et une interface en ligne de commande (Figure 2.2).

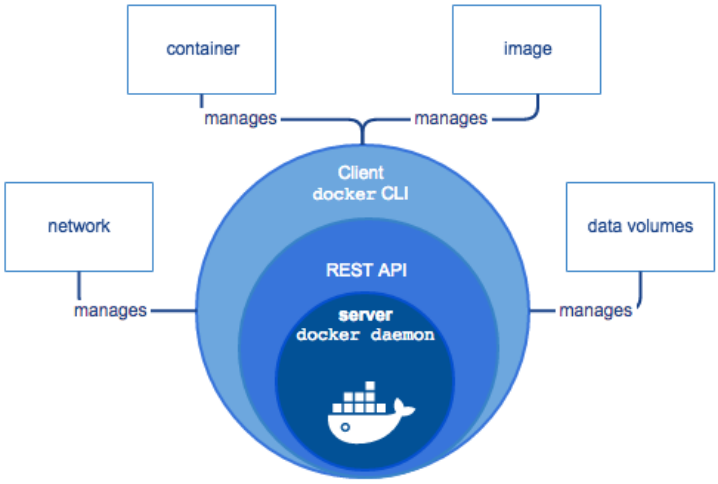

FIGURE 2.2 – Composants de Docker Engine [1]

Docker Image

Son fonctionnement est fondé sur un système d'images semblables à celles utilisées par les machines virtuelles : une image contient l'OS sur lequel va tourner l'application, les bibliothèques supplémentaires nécessaires à l'application et les fichiers sources de l'application. La construction des images se fait par superposition de couches. Chaque couche de l'image correspond à un ajout de contenu dans l'image.
Prenons l'image de l'OS **Ubuntu**, elle contiendra un certain nombre de couches correspondant aux bibliothèques nécessaires au lancement de l'OS. Si nous y créons un fichier quelconque, DockerEngine rajoutera une couche supplémentaire correspondant à cette ajout et créera une image identique à une couche prêt de l'image d'origine (Figure 2.3).

1. Processus s'exécutant en tache de fond dans un système.

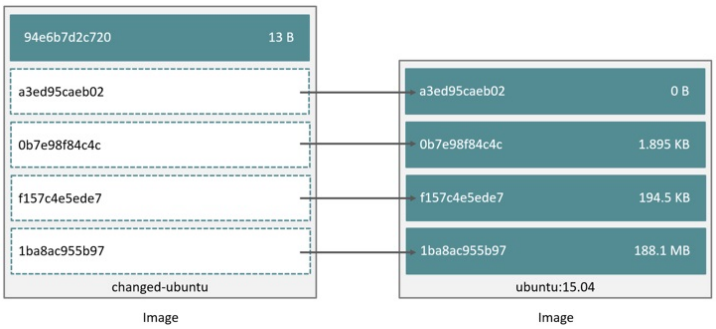

FIGURE 2.3 – Partage des couches par deux images [1]

La philosophie de Docker est que le partage simplifie l'optimisation des ressources. Ainsi, pour une image source, chaque image dérivée sera composée des mêmes sous couches originelles (Figure 2.4).

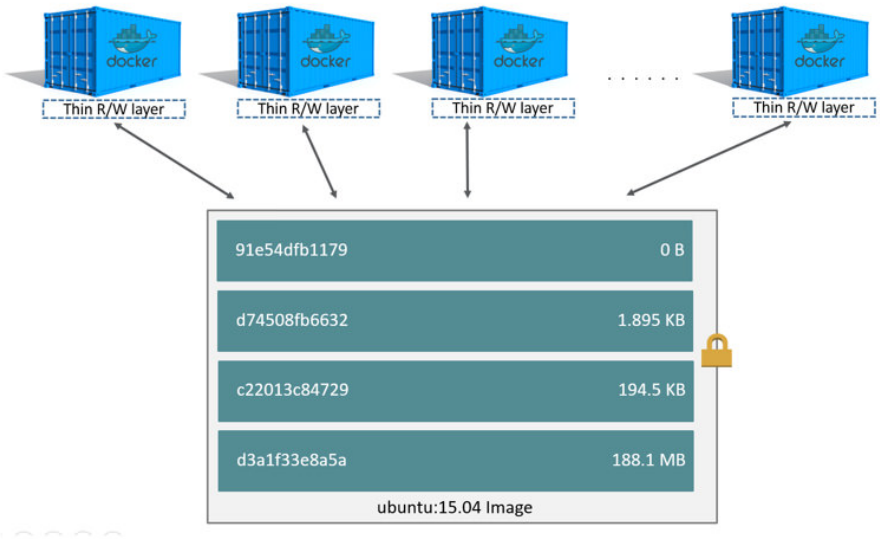

FIGURE 2.4 – Plusieurs images, mêmes couches originelles [1]

Docker Container

Un conteneur Docker est une instance d'une image Docker choisie. Lors de l'appel de la commande `$docker run <image>`, le démon Docker va créer un container sur la machine hôte qui n'est rien d'autre qu'une extension du *file system* de l'hôte. Ce conteneur partage certaines ressources avec la machine hôte :

- **L'interface avec les ports** de l'hôte qui se fait grâce à une socket contenue dans le Docker Engine. Cela permet d'associer un port du conteneur à un port de la machine. Cela est très pratique pour l'hébergement de plusieurs applications : la redirection de chacun des ports de la machine hôte peut s'effectuer sur plusieurs conteneurs différents.
- **Les volumes** de stockage. Ce sont les dossiers partagés entre le conteneur et la machine hôte : ils se trouvent dans la zone hôte du *file system* et non dans la partie conteneur. Il est important de noter que toutes les données non sauvegardées dans les volumes liés au conteneur, ne seront accessibles qu'à l'intérieur de celui-ci : s'il est détruit, les données sont perdues.

Lors du déploiement, on peut configurer plusieurs paramètres au conteneur :

- `--publish X:Y` permet d'associer le port X de l'hôte au port Y du conteneur
- `--volume dirX:dirY` défini le dossier dirX sur la machine hôte comme volume partagé associé au dossier dirY du conteneur
- `--name <name>` permet de définir un nom au conteneur pour le retrouver plus facilement.

Storage driver

La diversité des file systèmes en fonction des OS et des distributions complique l'écriture d'un conteneur vers la machine hôte. C'est pour cela qu'il existe plusieurs drivers de stockage (Figure 2.5).

Storage driver	Commonly used on	Disabled on
overlay	ext4 xfs	btrfs aufs overlay zfs eCryptfs
overlay2	ext4 xfs	btrfs aufs overlay zfs eCryptfs
aufs	ext4 xfs	btrfs aufs eCryptfs
btrfs	btrfs *only*	N/A
devicemapper	direct-lvm	N/A
vfs	debugging only	N/A
zfs	zfs *only*	N/A

FIGURE 2.5 – Driver storage [1]

Il est généralement déterminé automatiquement par Docker Engine mais il peut être modifié selon le comportement qu'on lui associe et certains besoins spécifiques (Figure 2.6). Le risque de choisir arbitrairement son driver est d'avoir une installation moins stable.

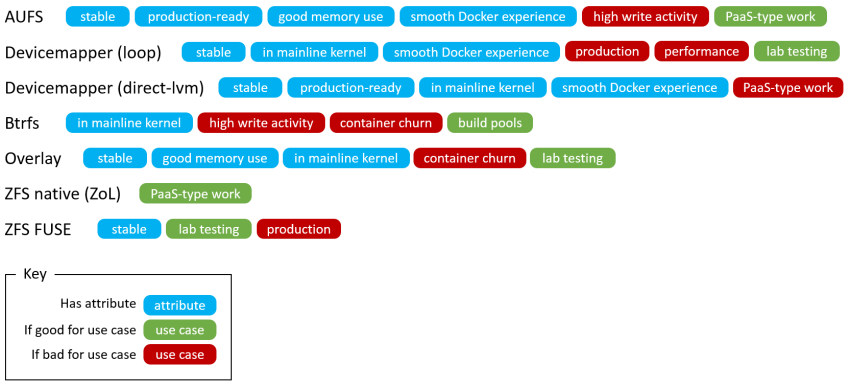

FIGURE 2.6 – Efficacité des drivers selon les besoins [1]

2.1.2 Liens entre conteneurs & réseaux

On l'a vu, il est possible d'associer le port d'un conteneur à un de la machine hôte. Cependant, dans le cadre d'une application conteneuriser, il est fort probable que nous ayons à y connecter une base de donnée, un serveur de log, un message broker etc. Ceux-ci peuvent aussi être lancés dans un autre conteneur.
Comment les relier ? Encore une fois, c'est Docker Engine qui s'en occupe : lors de l'appel de la commande `'$docker run --link containerA <image>'`, un lien va se créer entre le `containerA` et le conteneur créé.

De plus, Docker permet la définition de réseaux afin d'isoler les conteneurs entre eux. À l'installation, Docker Engine crée trois réseaux : `bridge`, `none`, `host`. `Bridge` correspond au réseau par défaut des conteneurs créés et `host` à un réseau supplémentaire de la machine.

Il est possible de créer des réseaux supplémentaires et d'associer des conteneurs à un ou plusieurs de ces réseaux. Un conteneur ne peux voir et atteindre des conteneurs des mêmes réseaux que lui. Cela permet un découpage des stacks de notre installation.

2.1.3 Configuration descriptive

Pour pouvoir déployer nos conteneurs, il faut d'abord définir et configurer nos images. Nous avons vu comment les images sont structurées d'un point de vue du file system, regardons comment les configurer et y installer notre application.

Pour cela, il y a deux solutions : la solution intelligente et simple et la bête et compliquée. La seconde consiste en la création d'un conteneur à partir de l'image d'un OS, à accéder à l'intérieur du conteneur via le bash puis en l'installation de notre application en ligne de commande. À partir de là, il suffit de sauvegarder le conteneur en une nouvelle image.

Dockerfile

La façon simple est de décrire notre image dans un fichier `Dockerfile` puis de le compiler (Figure 2.7). Ce fichier contient un descriptif de chacune des couches de notre image :

- **FROM :** l'image source. Ce peut être l'image d'un OS ou bien une image pré-existante.
- **RUN :** décrit la liste des commandes à exécuter pour l'installation des bibliothèques, des fichiers sources etc. Les bonnes pratiques veulent que chaque commande soit écrite sur une ligne différente pour qu'elle soit considérée comme une couche à part entière de l'image et surtout pour simplifier le "débuggage" lorsqu'une commande échoue.
- **USR :** permet de spécifier un utilisateur pour les traitements suivants.
- **ENV :** défini le *workind directory* de l'image. Il sera utilisé comme dossier d'entrée du conteneur.
- **ENV :** permet de définir toutes les variables d'environnements nécessaires au fonctionnement du conteneur.

- **COPY :** permet de copier des fichiers de l'hôte vers l'image lors de sa compilation.
- **EXPOSE :** liste les ports qui seront exposés par le conteneur.
- **ENTRYPOINT :** c'est le point d'entrée du conteneur, la commande qui sera exécutée à son lancement. La syntaxe est celle d'un tableau contenant la commande suivie de ses paramètres.

Dans le même dossier que le fichier, la commande `'$ docker build .'` créera l'image correspondante. Il ne reste plus qu'à l'instancier dans un conteneur.

```
FROM ubuntu:latest

RUN useradd -m -d /home/web web && mkdir /home/web/.venv &&\

apt-get update && apt-get upgrade -y && \
apt-get install -y libc6 libc6-dev libpython2.7-dev libpq-dev libexpat1-dev && \
apt-get install -y libffi-dev libssl-dev python2.7-dev python-pip && \
pip install virtualenv && \
virtualenv -p python2.7 /home/web/.venv/default && \
/home/web/.venv/default/bin/pip install cython && \
/home/web/.venv/default/bin/pip install cherrypy==3.6.0 pyopenssl mako && \
/home/web/.venv/default/bin/pip install psycopg2 python-memcached sqlalchemy && \
apt-get autoclean -y && \
apt-get autoremove -y && \
chown -R web.web /home/web/.venv

USER web
WORKDIR /home/web
ENV PYTHONPATH /home/web/webapp

COPY webapp /home/web/webapp

EXPOSE 8080

ENTRYPOINT ["/home/web/.venv/default/bin/cherryd", "-i", "server"]
```

FIGURE 2.7 – Exemple de Dockerfile pour une application Python

Docker-compose

Lorsque nos images sont créées, il y a une méthode plus simple et plus paramétrable que la ligne de commande pour instancier nos conteneurs : docker-compose. C'est un outil permettant le déploiement des applications multi-conteneurs. Il consiste lui aussi en

la rédaction d'un fichier décrivant l'environnement à déployer (Figure 2.8). On y retrouve la spécification de tout les éléments nécessaires au déploiement d'un conteneur.

Au lancement de la commande `'$ docker-compose up'`, l'outil déploiera les conteneurs dans l'ordre de description. L'outil Compose possède un équivalent pour toutes les commandes de Docker Engine, cela simplifie l'entretien d'application multi-conteneur et notamment les mises à jour.

```
mysql:
  image: mysql:<version>
  environment:
    - <env variables>
  volumes:
    - <volume on host>:/var/lib/mysql
  container_name: mysql

application:
  image: <application>:<version>
  links:
    - mysql:mysql
  ports:
    - 8080:80
  environment:
    - <env variables>
  volumes:
    - <volume on host>:/opt/mattermost/data
  container_name: <name>
```

FIGURE 2.8 – Exemple générique d'un fichier compose

Compose et variables d'environnement Le fichier compose peut être écrit dynamiquement avec des variables d'environnement. Il suffit pour cela de remplir un fichier `.env` dans le même répertoire et Docker Engine s'occupe de le charger.

Cette méthode permet d'écrire des fichiers compose configurable pour n'importe quel environnement. Ce mécanisme est utilisé dans Summoner pour configurer, entre autre, les noms de domaines et de sous domaine des outils, les ports, les noms de conteneurs et bien d'autres données.

```
ghost:
  image: ghost:${GHOST_VERSION}
  container_name: ${GHOST_SUBDOMAIN}-${DOMAIN}
  ports:
    - 127.0.0.1:${GHOST_WEB_PORT}:2368
  environment:
    - VIRTUAL_HOST=${GHOST_SUBDOMAIN}.${DOMAIN}
    - LETSENCRYPT_HOST=${GHOST_SUBDOMAIN}.${DOMAIN}
    - LETSENCRYPT_EMAIL=${GHOST_SUBDOMAIN}@${DOMAIN}
  volumes:
    - ${VOLUME_STORAGE_ROOT}/${GHOST_SUBDOMAIN}/blog:/var/lib/ghost/
```

FIGURE 2.9 – Exemple générique d'un fichier compose dynamique

2.1.4 Docker registry

Lorsque nous manipulons plusieurs images différentes et que nous souhaitons les utiliser sur plusieurs serveurs, il est avantageux de ne pas devoir les compiler sur chacun des serveurs. Pour éviter cela, Docker possède son propre gestionnaire d'image : le *registry*. Docker l'utilise pour stocker les images de membres de la communauté sur le site DockerHub.

De la même façon, il est possible de déployer un registry privée gérer nos propres images via la commande `'$ docker push <image>'`. Ainsi, de n'importe quel endroit, nous pourrons récupérer nos images grâce à la commande `'$ docker pull <image>'`.

2.2 Travail effectué

Nous l'avons vu, pour bien gérer les applications multi-conteneurs, il faut être rigoureux sur la gestion des volumes pour éviter les risques de perte de données. Pour cela, l'arborescence des dossiers dans Summoner est simple et sans ambiguïté (Figure 2.10).

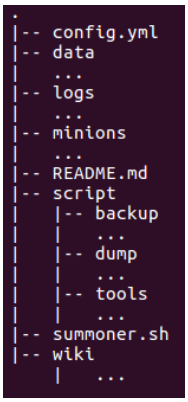

FIGURE 2.10 – Arborescence de Summoner

Le dossier `minions` contient tout les dossiers relatifs aux applications déployées, le dossier `data` contient les volumes de ces applications ainsi que ceux des bases de données, le dossier `scripts` tout les scripts nécessaires à la maintenance du système et le wiki quelques conseils et documentations sur la solution.

2.2.1 Installation de la solution

L'installation de la solution se fait grâce à un script shell au sommet de l'arborescence. Ce script se charge d'installer les outils relatifs à Docker selon la distribution de la machine hôte (actuellement soit Debian soit Ubuntu).

Summoner est installé selon la configuration chargée dans le fichier `config.yml` qui contient les meta-données nécessaires à l'installation : le nom de domaine du serveur, le volume racine des données et le volume racine des bases de données ainsi que la liste des applications à déployer à l'initialisation(Figure 2.11).

```
summoner:
  domain: virtualdonkey.fr
  vsroot: ~/Summoner/data
  dbsroot: ~/Summoner/data/database
  applications: nginx ghost nextcloud wekan
```

FIGURE 2.11 – Fichier de configuration de Summoner

La création de ce fichier de méta-données était nécessaire pour palier la problématique de généricité des chemins selon les différentes applications. Cette problématique se retrouve lors du lancement des scripts d'invocation des minions ou encore de maintenance : comment accéder au dossier de données d'une application donnée ?
Pour cela, un second fichier de méta-données est créé à l'installation et comprend les chemins absolus vers le dossier de Summoner ainsi que vers le dossier des Minions (Figure 2.12). Ce fichier se situe dans le *home directory* de l'utilisateur et est chargé à chaque manipulation afin d'avoir accès rapidement à chacune des informations nécessaires.

```
SUMMONER_HOME=~/Summoner
MINIONS_DIR=~/Summoner/minions
VOLUME_STORAGE_ROOT=~/Summoner/data
DATABASE_STORAGE_ROOT=~/Summoner/data/database
SUMMONER_CONFIG_FILE=~/Summoner/config.yml
```

FIGURE 2.12 – Fichier de méta-données

Lors de l'installation, les scripts de maintenance (de sauvegarde notamment) sont enregistrés dans le Cron [2] afin d'automatiser les sauvegardes.

2. Programme qui permet aux utilisateurs des systèmes Unix d'exécuter automatiquement des scripts, des commandes ou des logiciels à une date et une heure spécifiées, ou selon un cycle pré défini (Wikipédia)

2.2.2 Invocation d'un minion

La structure définissant un outil est composée de deux fichiers :

- Le fichier docker-compose définissant les conteneurs à déployer.
- Un fichier de configuration contenant les variables d'environnement nécessaires au déploiement.

Un script d'installation fourni permet la création du fichier d'environnement (`.env`) à partir de la configuration fournie. Son rôle est de vérifier que la configuration nécessaire au lancement de l'outil soit vérifiée : reverse proxy et certification TLS par exemple. De plus, il s'occupe d'effectuer les configurations manuelles nécessaires pour certains outils : autorisation de l'utilisation de fichiers de grande taille (stockage Cloud) ...

Architecture multi-conteneurs

Actuellement, la plupart des outils utilisés sont composés d'un conteneur d'application et d'un conteneur de base de donnée. Cette architecture est discutable sur deux points.

- **Multiples bases de données**
  La multiplication des outils provoque donc une duplication des types de bases de données. Le déploiement de deux outils utilisant une base MySQL créera deux conteneurs de base MySQL. La duplication du conteneur de base MySQL provoque une utilisation double d'espace disque. La première amélioration à effectuer serait de rendre unique les conteneurs de base.
- **La conteneurisation de bases de données**
  L'utilisation de conteneur pour stocker et utiliser les bases de données est déconseillée. Le risque de perte de données dans cette configuration est élevé : si un problème survient à l'intérieur du conteneur et provoque sa destruction, la perte sera plus conséquente qu'une base de donnée qui s'éteint. La pratique à mettre en place sera l'installation des bases de données sur le serveur.

Schéma de nommage

À des fins de simplicité pour l'automatisation des sauvegardes de données, nous avons défini une règle de nommage des conteneurs : cela permet, lors de la maintenance, de différencier les conteneurs de bases de données des conteneurs d'application et aussi de récupérer les applications déployées.

Par défaut, Docker nomme les conteneurs construits grâce à Docker Compose selon le motif suivant : `<directory>_<service>_1`. Selon la règle de nommage utilisée, les conteneurs d'application sont nommés `<application>-<domain>` et les conteneurs de base de données `<database_type>-<application>-<domain>`.

Cela permet en plus de pouvoir déterminer à quel nom de domaine est associé le conteneur dans le cadre de l'utilisation de plusieurs domaines.

2.2.3 Interaction entre minions

Sécurité SSL & sous domaines

Lors de la création de notre environnement de travail, la question s'est posée de l'accès à nos différents outils avec un seul nom de domaine. La réponse semblait évidente : les sous-domaines.

Pour cela, nous pouvions les gérer via l'interface de notre hébergeur et de notre DNS, cependant, comme l'idée est d'avoir un outil se déployant automatiquement avec une configuration valide avec le minimum (voir aucune) d'action humaine. De plus, l'instanciation d'un container docker attribut une adresse IP aléatoire au container et les ports attribués peuvent différer : cela complique la configuration via un hébergeur. Le but était de simplifier au maximum la redirection vers les outils contenus dans les conteneurs.
Nous avons donc décidés d'utiliser un premier conteneur Docker afin de l'utiliser comme reverse proxy pour rediriger les appels sur les sous-domaines vers l'outil correspondant. Pour cela, Nginx semblait être le plus utilisé et le mieux documenté.

L'image docker Nginx utilisée expose les ports 80 pour le protocole HTTP, et le port 443 pour le protocole HTTPS (Figure 2.13). Nginx nécessite aussi les droits de lecture sur la socket Docker pour pouvoir gérer la redirection des sous-domaines sur les conteneurs. Une fois ce container déployé, il suffit alors de rajouter la variable d'environnement `VIRTUAL_HOST` dans chacun des containers qui devront être accessibles par un sous-domaine.

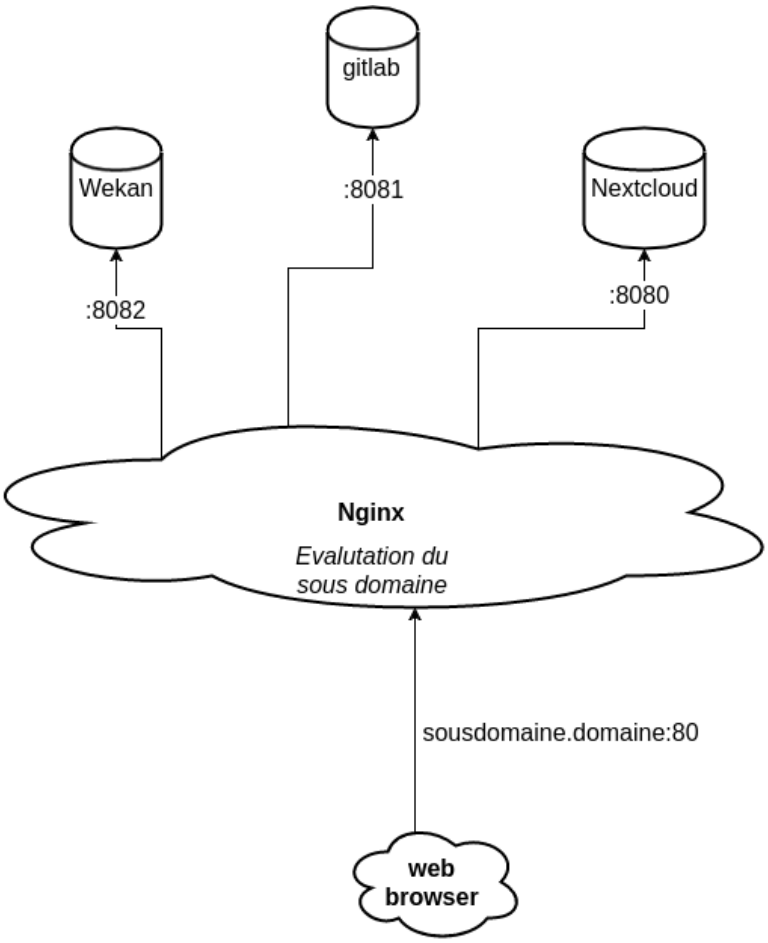

FIGURE 2.13 – Fonctionnement de Nginx comme reverse proxy

Malgré la configuration du reverse proxy Nginx, nous nous sommes confrontés à un autre problème : nos sous-domaines n'étaient pas atteignables par le protocole HTTPS. Or certains outils nécessitent absolument ce protocole (Gitlab, Mattermost ...). Pour cela, il fallait trouver une solution d'authentification des sites web. L'auto certification n'étant pas sûre et la plupart des certifications étant payantes, la solution étaient d'utiliser un

deuxième outils : LetsEncrypt.
LetsEncrypt est un organisme de certification gratuit et automatique. Parfaitement interfaçable avec notre reverse proxy pour valider nos conteneurs : la certification s'effectue à la volée à chaque déclaration de sous-domaine au prêt de Nginx.

Pour faire fonctionner Nginx et Letsencrypt en parallèle, il suffit d'ajouter deux nouvelles variables d'environnement : `LETSENCRYPT_HOST` et `LETSENCRYPT_EMAIL` (Figure 2.14) afin de valider la génération de certificat au prêt de Letsencrypt. Ainsi, à chaque déclaration d'un nouveau container Docker nécessitant la définition d'un sous-domaine, la configuration Nginx est alors rechargée à l'intérieur du docker (`nginx -s reload`), un certificat est généré par LetsEncrypt et le container est accessible de façon sécurisé.

```
statik:
  container_name: ${STATIK_SUBDOMAIN}-${DOMAIN}
  image: hanzel/nginx-html:${STATIK_VERSION}
  environment:
    - VIRTUAL_HOST=${STATIK_SUBDOMAIN}.${DOMAIN}
    - LETSENCRYPT_HOST=${STATIK_SUBDOMAIN}.${DOMAIN}
    - LETSENCRYPT_EMAIL=contact@${STATIK_SUBDOMAIN}.${DOMAIN}
  volumes:
    - ${VOLUME_STORAGE_ROOT}/${STATIK_DATA_DIR}:/etc/nginx/html
  ports:
    - ${STATIK_WEB_PORT}:80
```

FIGURE 2.14 – Fichier compose d'un conteneur accessible par un sous domaine et certifié

Solution de compte unique

Chaque outil utilisé nécessite un compte utilisateur pour accéder au contenu. Pour simplifier leur utilisation et éviter de se créer un compte utilisateur par outil, des solutions ont été étudiées : l'authentification par **OAuth** et l'utilisation d'un LDAP.

OAuth permet l'utilisation d'une API d'authentification d'un site web extérieur pour un compte utilisateur. C'est une méthode utilisée par de nombreux sites qui nous permettant de se connecter grâce à un compte Google, Github ou Facebook par exemple. L'inconvénient de cette méthode est la nécessité de partager des informations avec un site externe.

L'utilisation d'un LDAP permet la création d'un unique compte pour chaque utilisateur qu'il peut utiliser sur toute la suite d'outil. Le problème auquel nous faisons face actuellement est que tout les outils choisis ne supporte pas l'utilisation d'un LDAP. De plus, l'activation d'une authentification grâce à un LDAP nécessite une configuration manuelle non automatisable.
La solution désormais envisagée serait la création d'une méthode d'authentification donnant accès à l'ensemble des outils.

2.3 Entretien et maintenance

Presque tout les outils implémentés de Summoner utilisent une base de données et possèdent des fichiers de configuration propre. Pour palier toute mauvaise surprise, une politique de sauvegarde des données devait être mise en place. Cela comprend les sauvegardes à chaud (*dump*) et à froid (*backup*) des bases de données et les sauvegardes de données des applications.

2.3.1 Différentes sauvegardes de base de données

Dump de base de données

La sauvegarde à chaud des bases de données est la plus simple à réaliser puisque de nombreux utilitaires existent. Il consiste en une sauvegarde à un instant t des informations stockées en base en bloquant leur modification.

Un utilitaire existe pour chaque type de base de donnée : il s'occupe de bloquer les modifications, de sauvegarder les données et de les débloquer.

- **MySQL et MariaDB** possèdent le même utilitaire de dump `mysqldump` puisque MariaDB a été développée à partir de MySQL. L'utilitaire crée un fichier SQL permettant la reconstruction du contenu de la base.
- **MongoDB** possède son utilitaire `mongodump` qui effectue une sauvegarde binaire dans un format BSON (*binary JSON*).
- **PostgreSQL** et son utilitaire `pg_dump` se comporte comme `mysqldump` en créant un fichier de reconstitution. Cet utilitaire gère la compression du fichier de sortie.

Tout ces utilitaires ont leurs inconvénients : ils ralentissent l'utilisation de la base en bloquant l'écriture des données, en augmentant les entrées-sorties etc. C'est pour cela qu'il est conseillé de les utiliser lors d'une faible activité des services.

Backup de la base et de l'application

La sauvegarde à froid des bases de données consiste à effectuer une sauvegarde du *file system* de la base de donnée lorsque celle-ci est éteinte. C'est le même processus pour les données liées à l'application.
Pour cela, nous stoppons les conteneurs et nous sauvegardons les volumes de données dans une partition du *file system* de la machine hôte.

2.3.2 Fonctionnement de l'utilitaire

L'utilitaire de sauvegarde créé pour Summoner gère la sauvegarde des bases de données les plus répandues : MySQL, MongoDB, MariaDB et PostgreSQL.

Lors du lancement de l'utilitaire, il récupère les bases de données et leurs types via l'interface de commande de Docker Engine grâce à la règle de nommage appliquée aux conteneurs (Figure 2.15).

```
DATABASE_TYPE="mongodb" "mysql" "postgresql" "mariadb"

echo -e "\033[33m[‘date +%F_%H_%M_%S‘] Analyse online containers ...\033[0m"
# Getting all the DB containers running
for TYPE in ${DATABASE_TYPE[@]}
do
  DOCKER_LIST+=" ‘docker ps -f "name=$TYPE" --format "{{.Names}}"‘"
done
```

FIGURE 2.15 – Récupération des conteneurs de données

Puis le script exécute la sauvegarde à l'intérieur même du conteneur grâce à la méthode `'$ docker exec <conteneur> bash -c <cmd>'`. Cela permet d'y appliquer l'utilitaire associé à la base.

```
case "$TYPE" in
  "mongodb"    ...
    ;;

  "mysql"    APPLICATION_DUMP_FILE=$APPLICATION_DUMP_DIR/$APPLICATION-$DATE-dump.sql
    # Dumping the database
    docker exec $CONTAINER bash -c 'mysqldump [...]' > $APPLICATION_DUMP_FILE
    echo -e "\033[32m[‘date +%F_%H_%M_%S‘] $CONTAINER dump finished. \033[0m"
    ;;
  "postgresql"    ...
    ;;
  "mariadb"    ...
    ;;
      echo -e "\033[32m[‘date +%F_%H_%M_%S‘] Wrong case - Jump to next container"
    ;;
esac
```

FIGURE 2.16 – Exécution d'une commande dans un conteneur

Versionning des backup

Sauvegarder les données pour ne pas les perdre nécessite de stocker les sauvegardes dans un support externe afin d'éviter la perte de ces dernières. N'ayant pas de second serveur afin de mettre en place ce mécanisme, nous téléchargeons une copie des sauvegardes sur un compte Dropbox. Ceci n'est pas une fonctionnalité vouée à être utilisée dans la version finale de Summoner.

Chapitre 3

Perspective d'évolution

3.1 Amélioration de la structure

La structure actuelle de Summoner utilise des scripts bash pour installer et maintenir la solution. Ce choix a été fait pour nous permettre de nous focaliser sur les technologies de conteneurisation et les configurations des outils dont nous avions besoin sans avoir à choisir une technologie précise. Néanmoins, Summoner aura rapidement besoin d'une interface graphique d'installation d'outil ainsi que d'un gestionnaire de source. Pour cela, une grande partie de l'infrastructure de Summoner sera programmée en Python, langage de script ayant une gestion du *file system* très avancée.

Le catalogue d'outils sera élargi et la possibilité d'intégrer ses propres outils sera ajoutée. L'intérêt est de permettre aux développeurs d'y ajouter les environnements qu'ils développent : soit en ajoutant les fichiers nécessaires à la description du minion c'est à dire le fichier compose et le fichier de configuration correspondant ou en ajoutant le fichier Dockerfile et les sources nécessaires.

L'interface web de Summoner permettra de plus d'accéder à une tableau de bord permettant le monitoring de chacun des conteneurs. Il permettra aussi la configuration du centre de notification : la définition des Webhooks et la construction de recettes.

3.1.1 Webhook

Le centre de notification n'est pas encore développé. Des premiers tests de Webhook ont été réalisés en Python grâce à la bibliothèque CherryPy permettant la génération d'un simple serveur Web. Néanmoins, une réflexion poussée sera nécessaire à l'élaboration d'une fabrique de Webhook la plus généraliste possible ainsi qu'au choix de la structure de donnée la plus appropriée pour définir les recettes ou enchaînements de Webhook.

La possibilité d'exécuter ce centre de notification à l'intérieur d'un conteneur devra être envisagée. Aujourd'hui, les tentatives de conteneurisation d'un serveur web Python n'ont pas abouti.

3.1.2 Versions & mises à jour

Lors de la soumission d'une image Docker sur un *registry*, un nom lui est donné selon le schéma suivant : `<user_name>/<image>:<tag>`. La version de l'image est définie par la valeur du paramètre `<tag>`. Par défaut, celui-ci vaut `latest`, cependant ce n'est pas conseillé d'utiliser ce tag car l'image peut être reconstruite à la suite d'une modification ponctuelle par l'auteur. Cela peut aboutir à des problèmes de compatibilité non suspecté. Ce tag prend généralement comme valeur la version de l'outil installé dans l'image. Ainsi `bob/nextcloud:11.0` sera l'image déposé par bob sur le *registry* contenant la version 11.0 de l'outil Nextcloud.

Actuellement, la version de l'outil déployé est définie par le fichier de configuration de ce dernier. Par défaut, la version déployée est la plus récente déployée lors de l'élaboration du fichier compose. Afin de mettre à jour proprement les conteneurs, il suffit de modifier la version de l'image utilisée et de relancer le conteneur, l'outil se mettra à jour automatiquement. Un gestionnaire de version devra être implémenté afin de permettre à l'utilisateur de choisir la version qu'il souhaite installer. Aucune fonctionnalité de l'interface en ligne de commande ne permet cela. Il sera alors nécessaire d'interroger le *registry* grâce à une requête HTTP.

3.1.3 Best practices

Dans son état actuel, Summoner est encore un prototype. La première étape de son évolution sera la mise en pratique des *best practices* liées à la conteneurisation.

Nous avons parlé du fait d'utiliser des conteneurs de base de données. Ce fonctionnement n'est pas conseillé en environnement de production afin de limiter les risques de perte de données. Cependant, si l'utilisation des conteneurs de base est conservée, il faudra changer le fonctionnement des sauvegardes. En effet, les pratiques habituelles recommandent l'utilisation d'un second conteneur pour effectuer les sauvegardes. Ce second conteneur n'a qu'un seul but : lancer les utilitaires lors de son déploiement, sauvegarder les fichiers sur son volume puis s'éteindre.

3.2 Orchestration

L'utilisation des conteneurs comme remplaçant des machines virtuelles possède le même avantage d'être associés à des outils d'orchestration. Ces outils permettent de déployer et agir sur des conteneurs à distance et de façon globale. Mettre à jour des conteurs n'impose plus d'accéder à chacun un par un : l'orchestrateur s'en charge, de même lorsqu'il faut déployer des conteneurs sur de nouvelles machines : l'orchestrateur déploie directement les conteneurs avec la configuration souhaitée.

Pour Summoner, une solution d'orchestration peut devenir utile pour simplifier la gestion des nombreux conteneurs déployés. Ou bien pour une gestion d'environnement de développement, de production etc.

3.2.1 Puppet, Chef & co

Parmi les orchestrateurs on retrouve de grands noms : **Puppet** [1], **Chief** [2]. Ceux sont des outils très complets permettant d'automatiser les méthodes de *scalabilité* lors des montées en charge des serveurs. Ils nécessitent une installation longue et complexe.
D'autres ciblent principalement les conteneurs comme **Kubernetes** [3] ou son fork **Super-Giant** [4]. Eux permettent simplement le déploiement et le management des conteneurs sur machines distantes.
Enfin, des outils très léger permettent l'orchestration sur une échelle plus petite comme **Ansible**.

Leur fonctionnement est le suivant : la machine principale (appelée *master*) possède la configuration des nœuds. Elle est représentée par des fichiers descriptifs contenant les services à mettre en place sur les machines distantes. Cette configuration est poussée sur chacun des nœuds par la machine *master* de façon coordonnée. C'est de cette façon qu'est réalisée la maintenance de parc de machines.

3.2.2 Outils Docker

Docker Machine

Plus simplement, Docker fourni un outil pour déployer ses conteneurs à distance : Docker Machine. Il permet de définir un serveur distant comme destinataire des conteneurs déployés. Ainsi, le déploiement et la gestion des outils pourraient être simplifiés. Cependant, cela n'est valable que sur une seule machine distante : pour en manager plusieurs, il faut itéré plusieurs fois les commandes et Docker Machine ne propose pas de monitoring.

1. `https://puppet.com/`
2. `https://www.chef.io/`
3. `https://kubernetes.io/`
4. `https://supergiant.io/`

Docker Swarm

Dans le but d'obtenir une application fonctionnant sur un cluster de machines, Docker met à disposition l'outil Docker Swarm. Il permet la mise en place d'un cluster, la création de nœud. Swarm réunit plusieurs Docker Engine afin de créer un Docker Engine virtuel communicant avec tous. Les services déployés par Swarm sont répartis automatiquement sur les nœuds disponibles et peuvent être mis à l'échelle simplement.

La mise en place d'un *load balancer* est alors nécessaire pour aiguiller les requêtes vers les services déployés sur les différents nœuds (Figure 3.1).

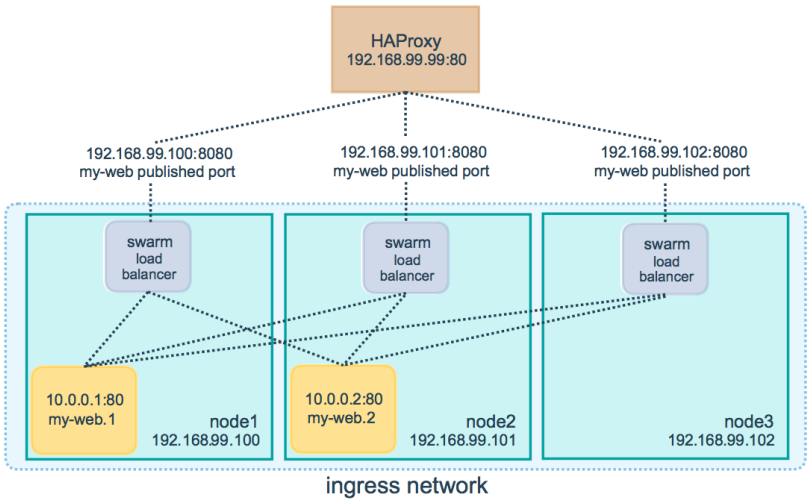

FIGURE 3.1 – Utilisation d'un *load balancer*.[16]

3.3 CoreOS, l'auto pilote

Parmi les outils développés pour aider le développement des conteneurs, un OS a été développé afin de simplifier encore plus l'utilisation des conteneurs : CoreOS. Cette distribution Linux veut simplifier la définition des files system pour supprimer le partage des bibliothèques entre le système hôte et les conteneur (Figure 3.2 & Figure 3.3).

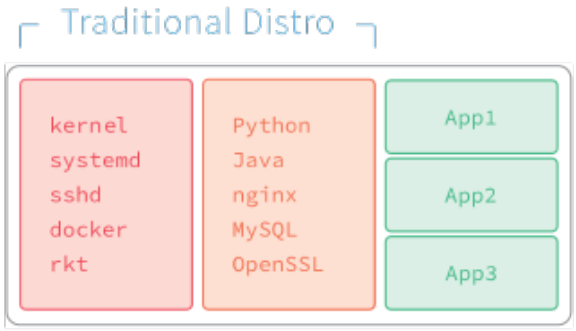

FIGURE 3.2 – File system de distribution Linux[17]

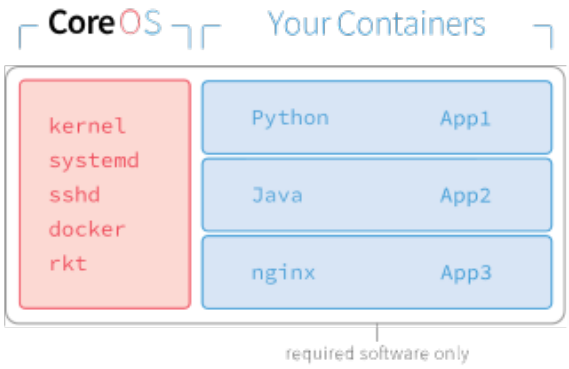

FIGURE 3.3 – File system de la distribution CoreOS[17]

Cette distribution innove dans la manière de maintenir à jour son système. Plus besoin d'administrateur ou de script de maintenance du système : les mises à jours sont poussées vers la machine. Ce fonctionnement nécessite moins de travail mais la question peut être posée quand à la sécurité de mises à jours automatique de ce genre.

La communauté autour de ce nouveau système l'a pourvu de nombreux projets annexes : au total plus de 90 projets open-source ont été créés en rapport avec CoreOS. Parmi eux un *registry* servant de gestionnaire de source, des plugins Kubernetes pour le monitoring, des outils de stockage clé-valeur développés pour la clusterisation, etc.

Conclusion

L'aboutissement de ce projet s'est traduit par la mise en production d'une solution permettant le déploiement d'outils conteneurisés. L'environnement est sauvegardé régulièrement et les sauvegardes sont versionnées et stockées sur un système externe. Cette solution reste cependant très statique : les outils pouvant être déployés sont limités et l'importation de ses propres outils n'est pas encore possible. L'ajout de ces fonctionnalités et la création d'un gestionnaire d'outils sont des pistes d'amélioration de la solution. De plus, l'utilisation d'un langage plus sophistiqué permettra une évolution et une maintenance plus aisées.

Ce projet fut l'occasion d'appréhender une technologie de conteneurisation en vogue, ses principes et son fonctionnement, ainsi que de se confronter à des problématiques de gestion d'environnement système : sécurité, politiques de sauvegarde, mises à jours, etc.

Outre l'aspect technique, ce projet a permis de nous faire réfléchir quant à l'efficacité de la communication au sein d'un groupe de travail et de nous confronter aux problématiques de l'expérience utilisateur. De nombreuses idées concernant ce projet se sont formées durant cette année, chacune d'elles sera étudiée, challenger et peut être implémentée dans la suite de ce projet. Parmi elles, l'utilisation de Hubot, le robot de Github, comme interface entre notre centre de notification et notre outil de déploiement.
Ce projet a été initié dans le cadre du développement du projet entrepreneurial **MyStoryStudio**, lauréat du Startup week-end. Nous espérons pouvoir exposer un prototype de cette solution sous forme de kit de démarrage lors d'un futur hackathon pour tester son efficacité.

Bibliographie

[1] **Documentation Docker**, *Définition du fonctionnement des conteneurs Docker et des drivers de stockage*
`https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/`

[2] **Lemon Cake**, *Descriptif de la stack Elastic*
`http://lemon-cake.fr/presentation-de-la-stack-elastic/#more-858`

[3] **Version d'essai SonarQube**, *Version d'essai en live de SonarQube*
`https://sonarqube.com/projects`

[4] **Documentation Docker**, *Documentation officielle Docker - Docker compose*
`https://docs.docker.com/compose/`

[5] **Documentation Docker**, *Documentation officielle Docker - Dockerfile*
`https://docs.docker.com/engine/reference/builder/`

[6] **DockerHub**, *DockerHub - Dépôt officiel d'image Docker*
`https://hub.docker.com/`

[7] **Jason Wilder blog**, *Automatisation de la combinaison Nginx et Letsencrypt*
http://jasonwilder.com/blog/2014/03/25/automated-nginx-reverse-proxy-for-docker/

[8] **Taiga.io**, *Documentation de l'application Taiga*
http://taigaio.github.io/taiga-doc/dist/setup-alternatives.html

[9] **Documentation Gitlab**, *Documentation des webhooks Gitlab*
https://docs.gitlab.com/ce/user/project/integrations/webhooks.html

[10] **Documentation Mattermost**, *Documentation de l'utilisation des webhooks avec Mattermost*
https://www.mattermost.org/community-applications/

[11] **Documentation MariaDB**, *Documentation officiel MySQLDump*
https://mariadb.com/kb/en/mariadb/mysqldump/

[12] **Documentation MariaDB**, *Bonnes pratiques et description des méthodes de sauvegarde MariaDB*
https://mariadb.com/kb/en/mariadb/backup-and-restore-overview/

[13] **Documentation officielle MongoDB**, *Utilisation des outils de sauvegarde MongoDB*
https://docs.mongodb.com/manual/tutorial/backup-and-restore-tools/

[14] **Documentation officielle PostgreSQL**, *Utilisation de l'utilitaire pgdump*
http://docs.postgresql.fr/8.1/app-pgdump.html

[15] **Github**, *Présentation du robot Hubot*
`https://hubot.github.com/`

[16] **Documentation Docker**, *Mise en place d'un routage entre nœuds de Docker Swarm*
`https://docs.docker.com/engine/swarm/ingress/#publish-a-port-for-a-service`

[17] **CoreOS**, *Description des fonctionnalités principales de CoreOS*
`https://coreos.com/why/`