Déploiement continu et architecture serveur

I. Introduction

1. Ressources utilisés

Organisation Github

Afin de gérer au mieux les différents projets que nous avions à réaliser, nous avons créé une organisation sur Github afin de regrouper tous les projets à un seul endroit. L'organisation s'appelle Vécolo Project https://github.com/vecolo-project

Serveur VPS

L'intégralité de nos applications et base de données sont hébergées sur un serveur VPS sur lequel pointe le domaine vecolo.fr

Le serveur possède 2 processeurs virtuels cadencés à 2.60 GHz, 2go de RAM, 30go d'espace de stockage en RAID10 ainsi qu'un connexion internet de 100 Mbps.

Raspberry PI 4

Nous avons un besoin spécifique qui est que nous devons simuler des station de recharge de vélo un peu partout dans Paris. Pour ce faire nous utilisons une Raspberry PI 4 car celle-ci possède un processeur ARM, c'est ce qui se rapproche le plus du futur composant embarqué sur de vraies stations de recharge.

Elle embarque un processeur ARM 4 cœurs 64bits cadencés à 1.5 GHz, 4go de RAM et 32go d'espace de stockage.

Nous possédons physiquement cette Raspberry et elle est hébergé physiquement chez nous.

2. Technologies phares

Docker Swarm & Portainer

Docker est un outil qui peut empaqueter une application et ses dépendances dans un conteneur isolé, qui pourra être exécuté sur n'importe quel serveur. Cela est très utile pour déployer et maintenir des applications sur un serveur sans conflits de dépendances.

Portainer est une interface web permettant d’avoir une meilleure gestion de docker sur un serveur, cela permet d’administrer toute nos instances sans avoir à utiliser un terminal.

Registre Docker privé

Un registre docker est un espace de stockage où mettons à dispositions les images docker de nos différentes application. Le fait qu’il soit privé signifie qu’il n’est pas accessible publiquement et peut être auto hébergé.

MariaDB

MariaDB est un système de gestion de base de données. Il s'agit d'un fork communautaire de MySQL. C’est dans cette base de données que nous stockons toute les données relative à Vécolo, le monitoring des applications et le Schéma BDD de l’application Java Vekanban.

Grafana

Grafana est un logiciel libre qui permet la visualisation de données. Il permet de réaliser des tableaux de bords et des graphiques principalement depuis des bases de données temporelles. Il est très souvent utilisé pour faire du monitoring d’applications serveurs.

Github Actions

Les actions Github sont des procédures automatiques qui peuvent être lancés lorsque différentes actions sont lancées sur un dépôt Github (push, merge, pull request, …). Elles permettent de lancer différentes actions en relation avec le code (exécution des tests unitaires, compilation de l’application, notifications via webhooks, …) ;

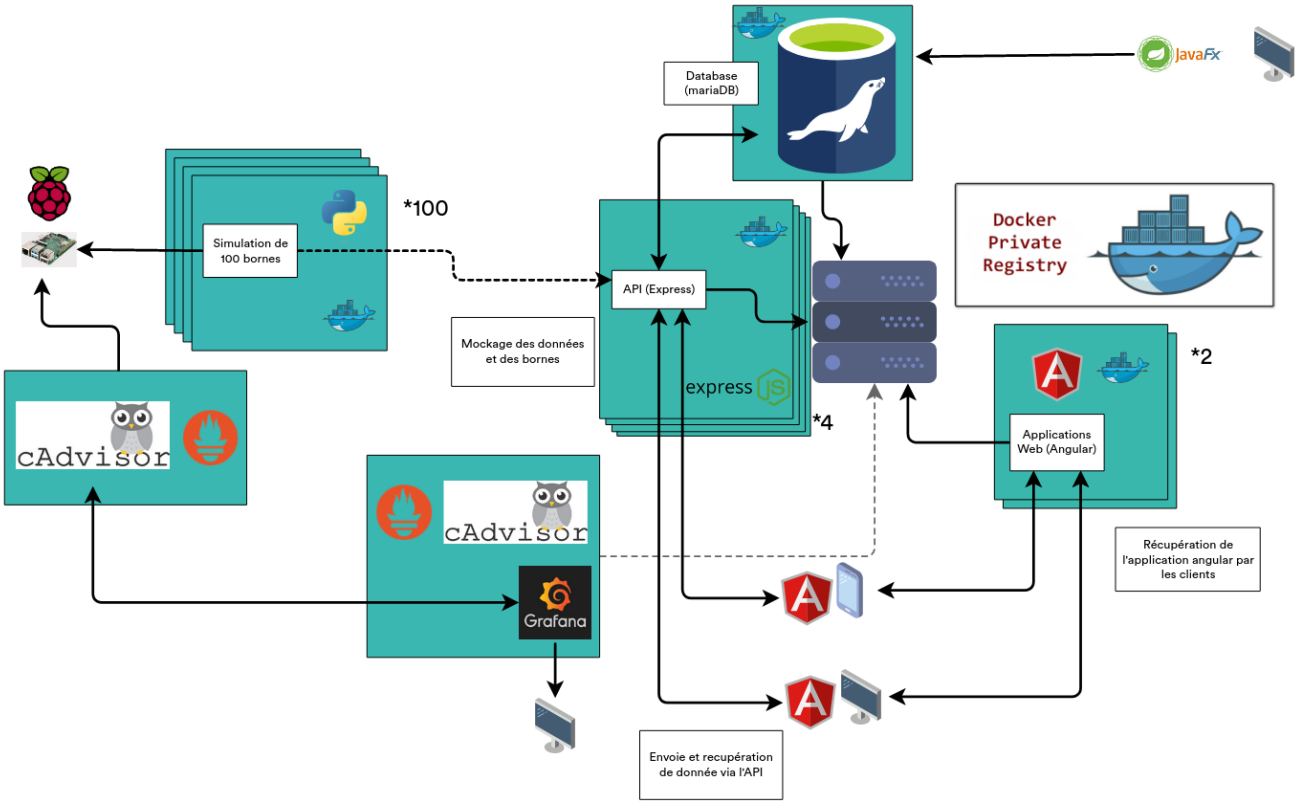
Caddy

Caddy est un serveur web écrit en G. Il a été conçu à l’ère du web sécurisé et il fonctionne ainsi par défaut et fourni du https, sauf si c’est demandé explicitement de ne pas le faire.

Il se chargera tout seul d’obtenir un certificat Let’s Encrypt et de le renouveler ensuite sans intervention manuelle. Il se chargera aussi de rediriger toutes les requêtes vers l’adresse en https, là encore sans configuration spécifique.

3. Architecture Globale

Voici le schéma d'architecture globale de notre projet que nous allons détailler dans ce rapport :



II. Docker, le cœur de notre architecture

Pour ce projet, nous avons vraiment voulu exploiter docker avec ce qu'il pouvait faire de mieux. Nous avons donc passé énormément de temps à apprendre cette technologie et l'implémenter correctement au sein de notre projet.

1. Tout est docker

Dockerisation de tous les applicatifs

Dans d'autres projets, mise à part l'application client Java, toutes nos applications possède un Dockerfile personnalisé qui permet à celle-ci de se lancer dans un container sans contraintes.

Cela nous permet d'avoir des images clé en main directement instanciable et fonctionnel.

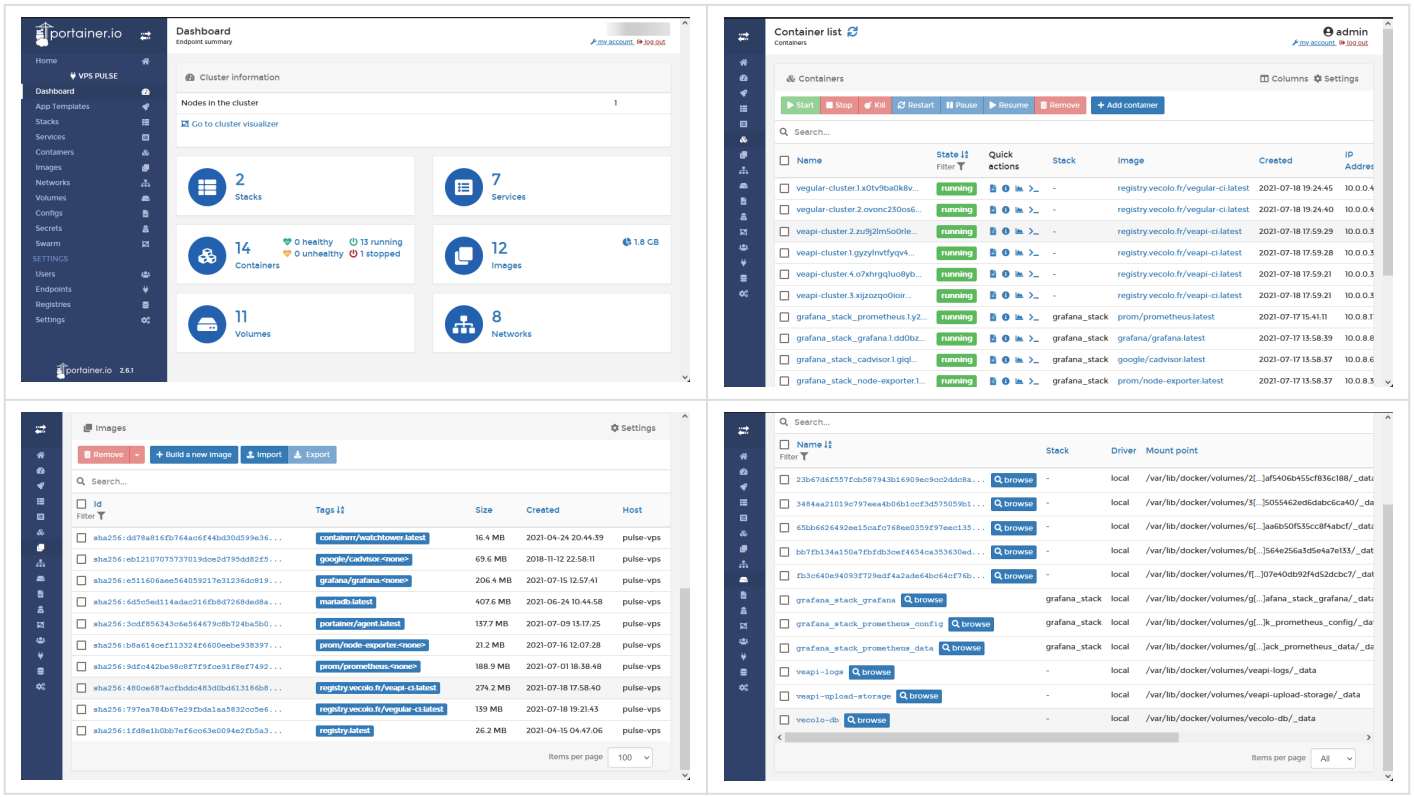
Voici par exemple Dockerfile de notre serveur backend API.

```
# Stage 1 building the code
FROM node:lts-slim as 🔨 builder
WORKDIR /usr/app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Stage 2 final stage with builded code
FROM node:lts-slim
WORKDIR /usr/app
COPY package*.json ormconfig.js ./
RUN npm ci --production

VOLUME /usr/app/upload
VOLUME /usr/app/logs

COPY --from=builder /usr/app/dist ./dist
COPY ./media ./media

ENV JWT_SECRET ='mon-token-secret' \
    DB_TYPE='mariadb' \
    DB_HOST='localhost' \
    DB_PORT=3306 \
    DB_DATABASE='dbname' \
    DB_USER='user' \
    DB_PASSWORD='' \
    PORT=3000 \
    LOG_LEVEL='debug' \
    ENDPOINT_PREFIX='' \
    NODE_ENV='production'\
    SENDGRID_API_KEY=''\
    SENDGRID_SEND_EMAIL=''\
    SENDGRID_REPLY_EMAIL=''\
    RECAPTCHA_KEY=''


EXPOSE 3000
CMD node dist/app.js
```

Afin que l'image finale soit la plus légère et performante possible, et qu'elle ne possède que l'applicatif, nous possédons dans nos instructions une image intermédiaire qui va simplement se contenter de compiler notre code avant de le faire transiter dans l'image finale qui possédera l'exécutable uniquement.

Portainer, l'interface de contrôle

Grâce à Portainer, nous possédons une interface web pour administrer nos images docker ainsi que les containers lancés. Un agent Portainer est présent aussi bien sur le serveur VPS que sur la Raspberry mais c'est le serveur VPS qui héberge l'interface web.



2. Cluster et load balancer

Nous avons décidé d'intégrer à notre projet la problématique de performance de notre application lorsque celle-ci est et soumis à une forte charge. Heureusement, docker nous propose plusieurs solutions pour répondre à cette problématique.

Docker Swarm

Docker-Swarm est un outil conçu pour enrichir Docker lui offre un "mode Swarm". Ce mode donne la possibilité de **créer des clusters de machines** exécutants des **conteneurs Docker**, qui **fonctionnent ensemble** comme une seule machine.

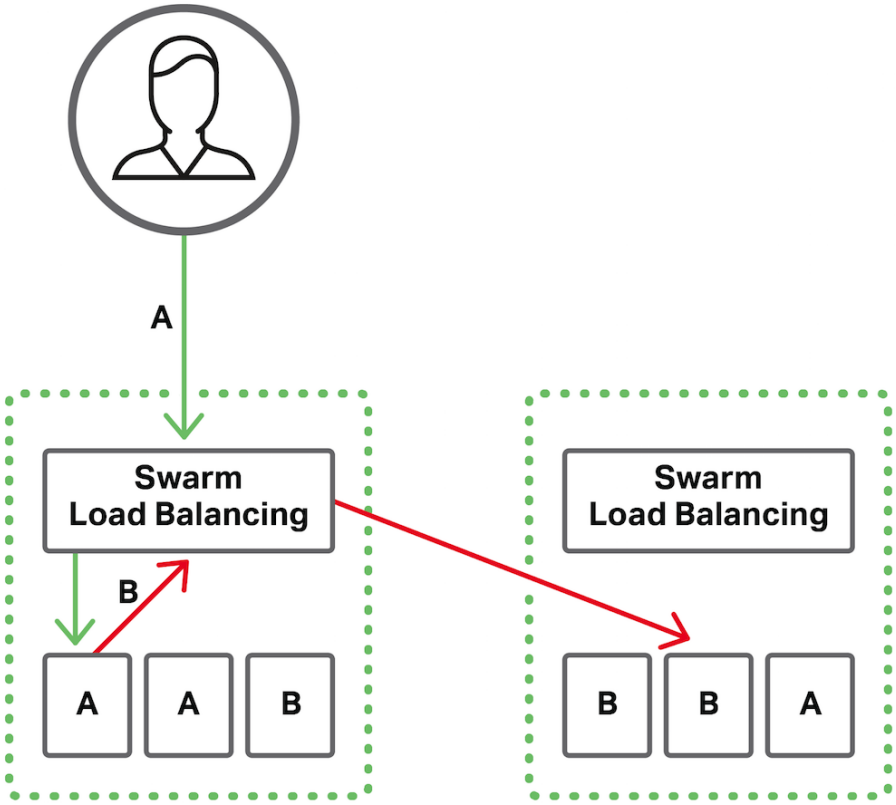
Docker-Swarm permet entre autres de :

- **Coordonner les conteneurs** et de leur **affecter des tâches** à chacun d'eux.

- **Gérer et suivre l'état** des différents conteneurs dans le cluster, et **redistribuer les tâches** en cas de besoins.

- **Assurer la scalabilité et flexibilité** de l'infrastructure, en augmentant ou diminuant le nombre de clusters mis en services.

- **Gérer et mettre à jour les applications** sur plusieurs conteneurs.



Mise en place des services

Swarm fonctionne grâce aux services, qui sont des **descriptions de l'état qu'on souhaite garder pour les nœuds du cluster**. Pour fonctionner, un service a besoin d'un conteneur et de commandes à exécuter sur celui-ci.

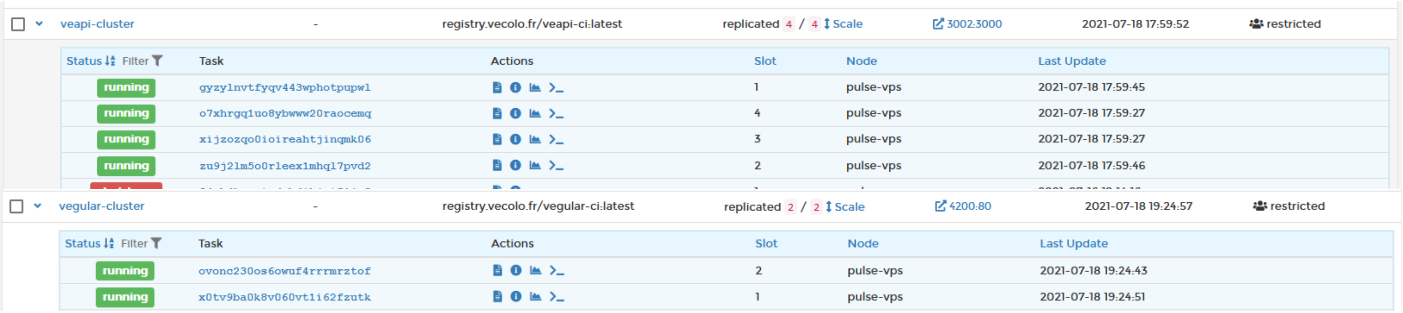
Les services exécutés sur Swarm peuvent avoir plusieurs caractéristiques, telles que :

- **Options des services** : lors de la création du service, on peut **configurer plusieurs paramètres selon les besoins de nos applications** (limites mémoire, le nombre des répliques de l'image à exécuter sur Swarm, etc.).

- **État désiré** : le déploiement du service permet de **définir l'état désiré** sur le Swarm. L'état désiré représente le **comportement normal ou la configuration idéale de l'application sur Swarm**. Par exemple, lorsqu'un problème survient et met à défaut l'état désiré, les "Manager Nodes" interviennent pour corriger le problème en affectant plus de ressources au service.

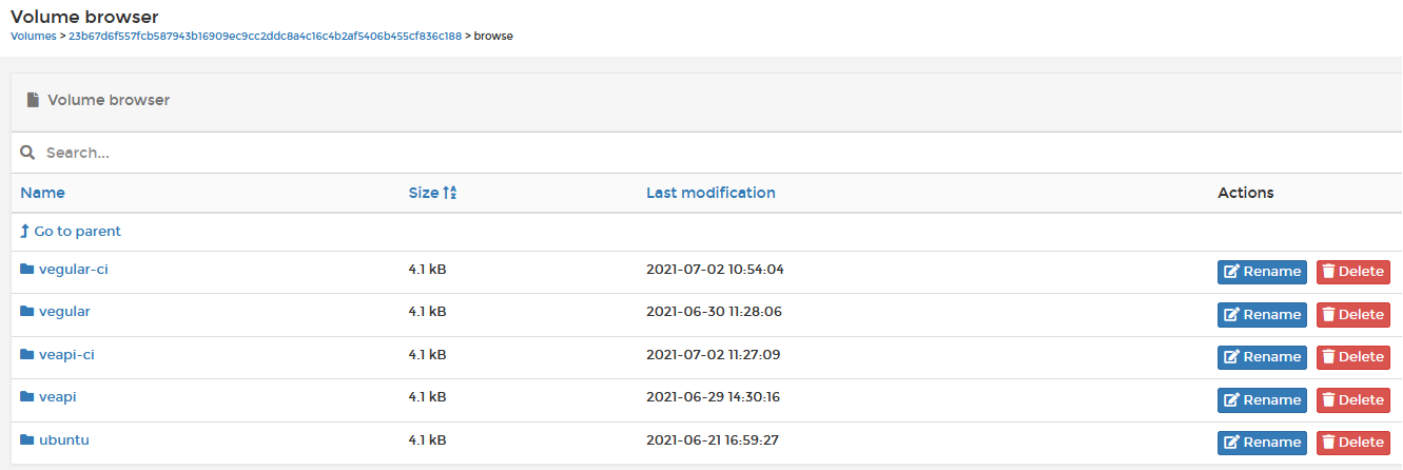
Grâce aux services, nous pouvons créer un répartiteur de charge entre plusieurs instance d'une même application. Celui-ci sera répliqué autant de fois que voulu.

Dans le cas de notre application, nous possédons un service sur le backend API qui va faire fonctionner 4 instances de notre application et répartir la charge équitablement sur chaque nœuds. Notre application WEB elle, possède un service avec 2 instances.



3. Registre docker privé

Les applicatifs étant la propriété de Vécolo, nous ne pouvons pas les héberger publiquement sur docker hub. Il nous faut notre propre registre, le VPS possède donc un registre docker privé avec une authentification. C'est sur ce registre que nous poussons les images de nos applicatifs. Nous avons ainsi un contrôle total sur l'hébergement de nos images Docker.



III. Déploiement continu

Les étapes de déploiement d'un projet web sont souvent fastidieuses et répétitif. De plus, il faut parfois que cela se fasse rapidement et il n'y a pas toujours quelqu'un de disponible pour le faire.

Nous avons donc fait en sorte que à partir du moment où nous poussons du code sur des branches spécifiques de nos projets, l'application soit automatiquement mise à jour sur le serveur principal sans intervention de notre part.

1. Builds automatique & Webhooks

Grâce aux actions Github, nous avons pu mettre en place sur nos dépôts de l'application Backend API et Front end Angular des évènements déclenchés lorsque la branche « **dev** » est mis à jour (dernière version fonctionnelle de nos applicatifs). Cet évènement va se charger de construire l'image Docker associé à la dernière version du projet, avant de l'envoyer à notre registre privé.

Une fois cela fait avec succès, un signal est envoyé au service (API ou Angular) présent sur le serveur à l'aide d'un Webhook.

```yaml
name: cd

on:
  push:
    branches:
      - 'dev'

jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2

      - name: Set up QEMU
        uses: docker/setup-qemu-action@v1

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1

      - name: Login to Vecolo Registry
        uses: docker/login-action@v1
        with:
          registry: registry.vecolo.fr
          username: vecolo
          password: ${{ secrets.REGISTRY_PASSWORD }}

      - name: Build and push
        uses: docker/build-push-action@v2
        with:
          context: .
          push: true
          tags: registry.vecolo.fr/vegular-ci:latest

      - name: Trigger API Webhook reload service
        uses: joelwmale/webhook-action@2.1.0
        with:
          url: ${{ secrets.WEBHOOK_URL }}
```

2. Mise à jour des services incrémental

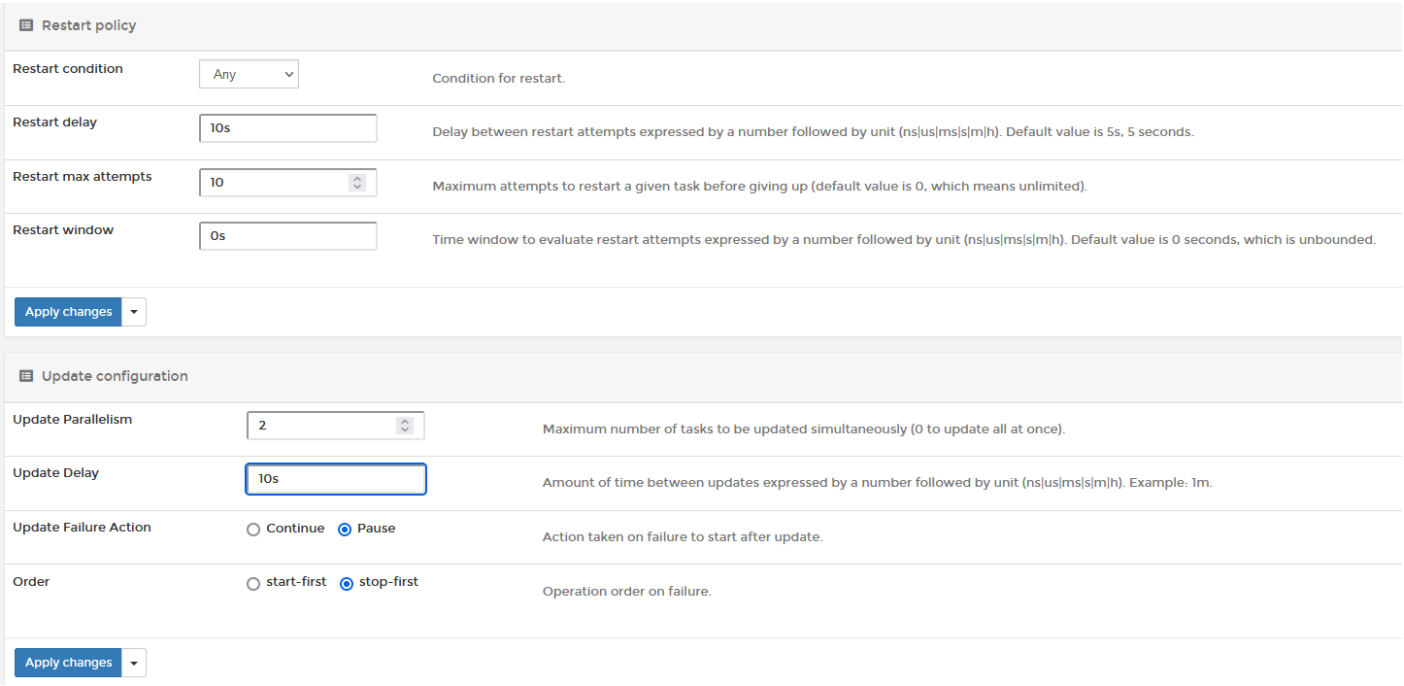
Grâce à la configuration de nos service, celui-ci met à disposition une URL pouvant être appelé depuis un Webhook. Quand celle-ci est requêté, le service sait qu'il doit redémarrer en mettant à jour l'image Docker de ses instances.

Cependant nous ne voulons pas d'interruption de service. C'est pourquoi les instances ne se mettent pas toutes à jour en même temps.

Dans le cadre du back end API (4 instances) les serveurs se mettent à jour 2 par 2. Les 2 derniers serveurs ne se mettent à jour que si les 2 premiers ont réussi à se lancer correctement et continue de tourner depuis au moins 10 secondes.

Pour l'application Angular qui ne contient que 2 instances le service va les mettre à jour une par une

En cas d'échec, le service va retenter plusieurs fois de lancer les instances jusqu'à un maximum de 10 fois. Si au bout de 10 fois le service n'arrive toujours pas à lancer les instances, il effectue un rollback sur les précédentes versions fonctionnelles de l'application. Ainsi, avons en permanence une instance fonctionnelle de lancée.



IV. Raspberry VS 100 Bornes

1. Objectif recherché

Notre projet Vemock consiste à simuler le comportement d'une station. Or dans notre application, nous souhaitons avoir bien plus qu'une seule station d'active afin que le comportement soit réaliste. De plus, nous souhaitons simuler ce comportement depuis un appareil ayant des composants matériels prévus pour de l'embarqué. Cela permettra dans le futur une meilleure compatibilité sur de vrais stations.

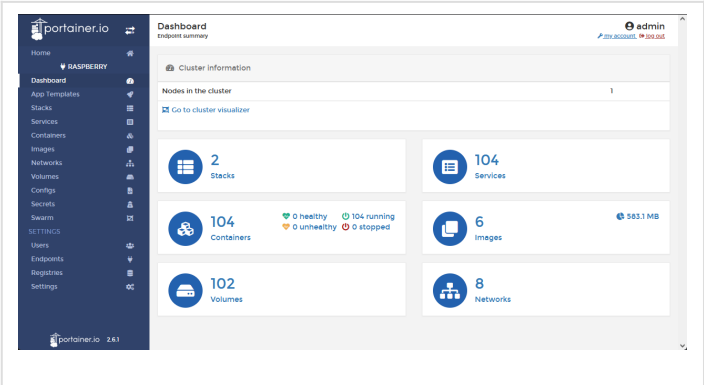
Le choix de la Raspberry était donc tout indiqué. Il fallait maintenant trouver un moyen correct de faire tourner plusieurs instance de station différente sur cette Raspberry.

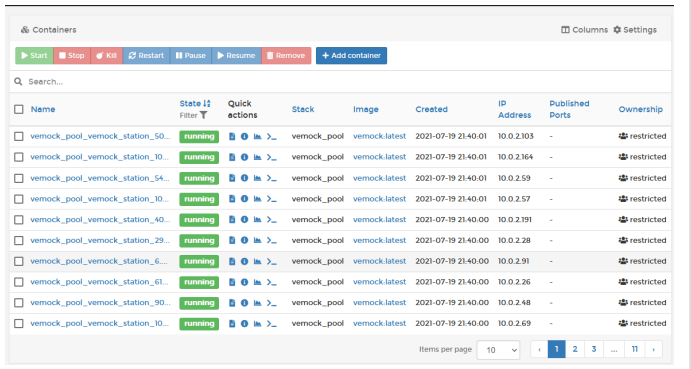
2. Stack de 100 bornes

Docker Compose est un outil qui permet de décrire (dans un fichier YAML) et gérer (en ligne de commande) plusieurs conteneurs. Il est alors possible de démarrer un ensemble de conteneurs en une seule commande. Dans le fichier **docker-compose.yml**, chaque conteneur est décrit avec un ensemble de paramètres qui correspondent aux options disponibles lorsqu'on lance une instance normalement.

La solution d'utiliser docker compose pour monter plusieurs bornes en même temps était donc très intéressante.

Notre docker-compose final permet de lever en une seule commande 100 instances ! elles sont toutes identifiées sur l'API grâce à un token unique et permet donc de simuler le comportement de 100 stations à Paris.





3. Générateur du docker-compose

Pour simuler le comportement de 100 stations, le fichier docker compose **docker-compose.yml** fais plus de 2400 lignes. Bien sûr il serait trop long d'écrire toutes ces lignes à la main...

Nous avons donc mis au point un script python qui permet à partir d'un fichier de configuration JSON beaucoup moins gros de générer notre **docker-compose.yml** final.

Voici un exemple des fichiers de configuration avec seulement 2 stations :

config.json × docker-compose.yml ×

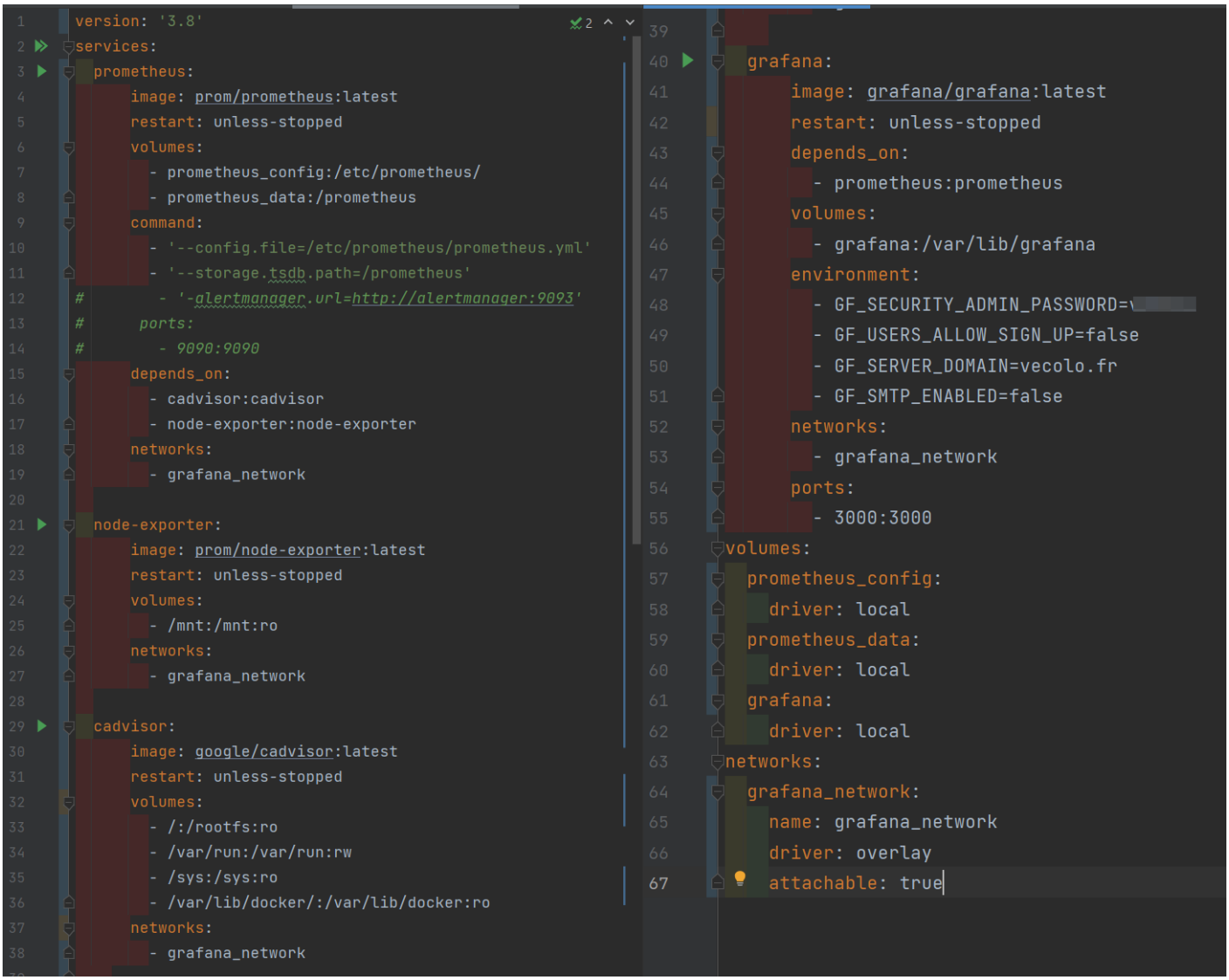
```json
{
  "config": {
    "frequency_wait": 5,
    "history_path": "history.json",
    "max_log_size": "30m",
    "max_log_file": 10,
    "api_endpoint": "[redacted]",
    "debug":  "True"
  },
  "station_list": [
    {
      "station_id": 1,
      "auto_off": "True",
      "max_seats": 12,
      "max_charge": 20,
      "jwt_token": "[redacted]"
    },
    {
      "station_id": 2,
      "auto_off": "True",
      "max_seats": 6,
      "max_charge": 20,
      "jwt_token": "[redacted]"
    }
  ]
}
```

```yaml
version: '3.8'
services:
  vemock_station_1:
    environment:
      - station_id=1
      - auto_off=True
      - max_seats=12
      - history_path=history.json
      - frequency_wait=5
      - max_charge=20
      - jwt_token=eyJ[redacted]
      - api_endpoint=https://a[redacted]
      - debug=True
    volumes:
      - vemock_station_1:/usr/src/app/history
    networks:
      - vemock_network
    restart: unless-stopped
    logging:
      driver: "json-file"
      options:
        max-size: "30m"
        max-file: "10"
    image: vemock
  vemock_station_2:
    environment:
      - station_id=2
      - auto_off=True
      - max_seats=6
      - history_path=history.json
      - frequency_wait=5
      - max_charge=20
      - jwt_token=eyJhb[redacted]
      - api_endpoint=https[redacted]
      - debug=True
    volumes:
      - vemock_station_2:/usr/src/app/history
    networks:
      - vemock_network
    restart: unless-stopped
    logging:
      driver: "json-file"
      options:
        max-size: "30m"
        max-file: "10"
    image: vemock
volumes:
  vemock_station_1:
    driver: local
  vemock_station_2:
    driver: local
networks:
  vemock_network:
```

V. Monitoring

Comme nous pouvons le constater, notre architecture serveurs est assez rempli et beaucoup d'applications différentes. Il est donc compliqué de tout surveiller sans aide.

Nous avons donc mis en place une stack docker compose regroupant Grafana, CAdvisor, Node-Exporter et Prometheus. Afin de pouvoir monitorer tout ce qui se passe sur nos serveurs.



```yaml
version: '3.8'
services:
  prometheus:
    image: prom/prometheus:latest
    restart: unless-stopped
    volumes:
      - prometheus_config:/etc/prometheus/
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
#       - '-alertmanager.url=http://alertmanager:9093'
#     ports:
#       - 9090:9090
    depends_on:
      - cadvisor:cadvisor
      - node-exporter:node-exporter
    networks:
      - grafana_network

  node-exporter:
    image: prom/node-exporter:latest
    restart: unless-stopped
    volumes:
      - /mnt:/mnt:ro
    networks:
      - grafana_network

  cadvisor:
    image: google/cadvisor:latest
    restart: unless-stopped
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    networks:
      - grafana_network

  grafana:
    image: grafana/grafana:latest
    restart: unless-stopped
    depends_on:
      - prometheus:prometheus
    volumes:
      - grafana:/var/lib/grafana
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=
      - GF_USERS_ALLOW_SIGN_UP=false
      - GF_SERVER_DOMAIN=vecolo.fr
      - GF_SMTP_ENABLED=false
    networks:
      - grafana_network
    ports:
      - 3000:3000
volumes:
  prometheus_config:
    driver: local
  prometheus_data:
    driver: local
  grafana:
    driver: local
networks:
  grafana_network:
    name: grafana_network
    driver: overlay
    attachable: true
```

1. Grafana & Prometheus

Grafana est un outil open source de monitoring informatique orienté data visualisation. Il est conçu pour générer des tableaux de bord sur la base de métriques et données basées dans le temps.

On peut y connecter différentes bases de données, orientées Time Series (base de données optimisée pour le stockage de données horodatées) pouvant être alimentées grâce à une grande variété d’agents de monitoring. Dans notre cas il s’agira d'une base donnée Prometheus.

2. Node exporter, les métriques systèmes

Prometheus ne sait pas collecter d'informations tout seul. Il lui faut des agents. Le plus simple et le plus complet pour avoir une vision global de la situation d'une machine (CPU, RAM, Load, traffic, etc.) est le bien nommé Prometheus Node Exporter. Il nous permettra d'avoir les métriques globales de la machine.

3. Cadvisor, les métriques Docker

Les containers sont aujourd'hui largement utilisés du développement jusqu'en production. Cependant un **docker stats** en SSH ne permet pas de gérer correctement son environnement de production.

Cadvisor est une solution rendue open-source par Google qui permet de fournir les ressources utilisées par des containers docker. Cela nous permettra d'avoir des métriques personnalisées pour chaque instance tournant sur nos serveurs

4. Tableaux de bords

Grâce à cette composition, nous possédons 2 tableaux de bord.

- Un pour le serveur principal
- Un pour la Raspberry

Les 2 tableau de bord sont quasi similaires la différence près que nous surveillons également la température de la machine sur la Raspberry car c'est un composant qui chauffe vite.

VI. Bilan du projet

1. Problèmes rencontrés

Docker Swarm et les services

Nous n'avons pas eu de cours de docker cette année et il a donc fallu partir de 0.

Docker Swarm est un concept que nous avons eu énormément de mal à comprendre et nous n'avons pas tout de suite implémenté des services car nous n'y arrivions tout simplement pas.

C'est plutôt vers la fin du projet que nous sommes enfin parvenus à faire ce que nous voulions avec.

Déploiement continue

Nous avons tenté d'abord d'utiliser Jenkins pour le déploiement continu mais nous nous sommes vite retrouvés perdus devant l'immensité de l'outil et le peu d'informations que nous avions à ce sujet.

Ce sont sur les derniers cours de l'année que nous avons pu discuter avec certains de nos professeurs afin de trouver des solutions adéquates pour avoir un développement continu qui tient la route

Mockage de 100 stations

Le mockage des 100 stations était un des plus grands défis de ce projet. Il a fallu beaucoup travailler sur l'optimisation du code pour que cela prenne le moins de ressources possible afin de faire tourner 100 instances simultanées sur une Raspberry.

Le premier montage a été long et fastidieux car il fallait créer 100 stations à Paris à des endroits géographiques cohérents est spécifié dans le docker compose la configuration de chacune.

2. Conclusion

Pour conclure, c'est sans doute la plus grande architecture serveur que nous ayons eu à mettre en place depuis le début de notre scolarité.

Le fait d'intégrer docker dans cela a été une grande nouveauté et nous a permis d'aller beaucoup plus loin que les années précédentes.

Nous sommes très fiers du déploiement continu que nous avons puis mettre en place vers la fin, ainsi que les fameuses stations qui constituent un des gros objectifs du projet.

Enfin, il a été très agréable d'avoir pu monitorer tout ça et surveiller le comportement de nos applications et nos serveurs.

Nous comptons sans doute réutiliser cette architecture pour nos futurs projets

---

Revision #1
Created 27 September 2022 18:50:01 by Noé Larrieu-Lacoste
Updated 27 September 2022 19:00:24 by Noé Larrieu-Lacoste