PROJET PROFESSIONNEL DEVOPS

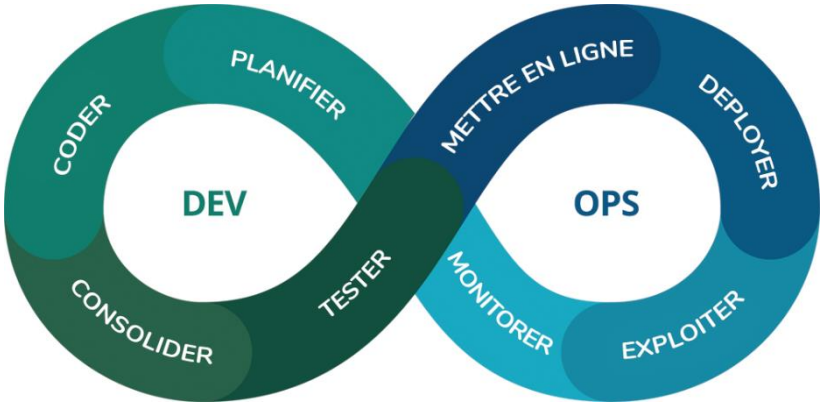
Marie CHAP DevOps 2023/2024

Marie CHAP
Mariechap501@gmail.com

Sommaire

I. Introduction DevOps 3
A. Les outils DevOps principaux 3
1. Planification et collaboration : 3
2. Gestion du code (développement) 4
3. Intégration Continue / Déploiement Continue (CI/CD) 4
4. Gestion de l'infrastructure 4
5. Surveillance et retour d'information 4
II. Présentation du projet professionnel 5
III. Initialisation du projet Part.1 : 6
A. Initialisation du projet sur gitlab 6
B. Ouverture de VPS sur ovh 6
C. Installation de Node.js et PM2 sur le VPS 6
D. Préparer le plan lié aux tâches du projet 7
E. GitFlow : 8
F. Préparer un système de tag et une branche versionning 8
IV. Mise en place CI (Continuous Integration)/CD(Continuous Delivery) 9
A. CI/CD Backend 9
B. Code coverage backend 11
C. Ajout CI/CD Frontend 13
V. Conteneurisation de l'application avec Docker Part.2 14
A. Introduction à Docker 14
1. Les images Docker 14
2. Les conteneurs 14
3. Exécuter les conteneurs avec Docker-compose 15
B. Création d'un fichier reverse proxy 15
C. Création d'image – Dockerfile 16
D. Déploiement avec Docker compose 17
E. Ajout d'un environnement de production au déploiement 17
VI. Monitoring 19
VII. Veille informatique 19

I. Introduction DevOps



DevOps est la concaténation des trois premières lettre du mot anglais *development* et de l'abréviation « ops » du mot anglais *operations* (exploitation).

C'est un terme qui à été inventé par le belge Partrick Debois en 2007.

Le DevOps est un ensemble de pratiques qui visent à réduire le fossé entre le développement logiciel (dev) et les opérations informatiques (Ops), d'où le terme.

L'idée est de favoriser une collaboration plus étroite et une meilleur communication entre ces deux entités qui opèrent souvent de manière isolée.

Les principes du DevOps sont :

- **l'intégration continue** (CI pour *continuous integration* – le code est régulièrement fusionné et testé),
- La livraison continue (CD pour *continuous delivery* – les mise à jour du logiciel sont régulièrement libérées pour la production),
- L'infrastructure en tant que code (la gestion de la provision des infrastructures informatiques via le code – *IAC* pour *Infrastructure As Code*),
- La surveillance et la journalisation (le suivi en temps réel de la performance et des erreurs du logiciel)
- La culture de la rétroaction (l'encouragement à l'amélioration constante via les retours d'information)

Schéma classique :

image

A. Les outils DevOps principaux

1. Planification et collaboration :

Ces outils permettent de créer des tâches et de gérer un projet. Les plus utilisés sont :

- GitLab
- GitHub
- Jira

2. Gestion du code (développement)

Ces outils permet d'effectuer un contrôle de version du code. Les plus utilisés sont :

- GitHub
- GitLab
- Bitbucket

3. Intégration Continue / Déploiement Continue (CI/CD)

Lorsque l'on utilise un outil de dépôt de code, par exemple GitLab, les développeurs peuvent répartir les fonctionnalités à traiter sous forme de *branches*. Une branche représente une fonctionnalité, un problème ou la réparation d'un bug. Chaque fois qu'un développeur apporte une modification, il regroupe ces changements dans ce qu'on appelle un *commit*. Le *commit* sauvegarde les modifications apportés et c'est accompagné d'un message expliquant brièvement les changements. Chaque commit déclenche automatiquement une série d'instruction d'intégration continue pouvant compiler le code, exécuter des tests unitaires, des test d'intégration et d'autres types de tests pour s'assurer que les dernières modification n'ont pas introduit de bugs. C'est instructions sont regroupé dans une fichier qu'on appelle *pipeline*.

Si tout les tests passent, ces outils peuvent être configurés pour être déployer automatiquement ou manuellement selon l'environnement. Le déploiement de l'application peut être automatique sur un environnement de staging ou de test par exemple et manuel sur un environnement de production.

Des solutions très connues sont :

- Jenkins
- GitLab CI/CD
- Github actions
- AWS CodePipeline
- Azure DevOps
- CircleCI
- Travis CI

4. Gestion de l'infrastructure

Ces outils concerne la configurations de plusieurs serveurs ou d'ordinateur ayant chacun un rôle spécifique, par exemple gestion de stockage, calcul, traitement de données. Tout ces serveurs et ordinateur sont regroupés dans ce qu'on appelle un cluster.

Voici une liste de quelques outils permettant de coordonner …

5. Surveillance et retour d'information

Nous avons besoin de surveiller les applications exécutées sur des serveurs, une fois que celle-ci sont déployer. Des outils permettent la surveillance et alerte en collectant des métriques d'applications et de système en temps réel, aidant à la détection et résolution de problème, par exemple Prometheus.

Voici une liste des outils les plus courantes :

- Grafana
- ELK (Elasticsearch, logstash,kibana)
- Datadog / Nagios / New Relic / Sentry

II. Présentation du projet professionnel

Le projet dont je vais vous présenter se fera en 2 étapes. Je vais partir sur une application avec une partie pour gérer le front « *frontend* » en Vue.js et une partie « *backend* » en Node.js qui fera office d'API pour l'authentification et serveur web pour servir la construction de l'application (build) frontend Vue.js.

L'application permet de s'authentifier en utilisant une base de données MongoDB et des tokens JWT.

Je vais utiliser GitLab pour la planification, l'organisation des tâches et le CI/CD. Pour le déploiement j'ai ouvert des *Virtual Private Server (VPS)*, sur OVH, pour déployer mon application.

Je vous présenterai l'organisation lié aux tâches à effectuer ainsi que la répartition des branches de travail (GitFlow), comment j'ai préparé le « *versionning* » de l'application et comment j'ai construit mon fichier de configuration CI / CD (continuous integration – Continuous Delivery) en organisant les étapes de :

- vérification de qualité code,
- construction de code,
- tests du code
- Déploiement de l'application

L'objectif de cette première partie est de créer le fichier d'instruction CI / CD (pipeline) réaliste avec 4 étapes et quelques instructions (jobs).

Un pipeline est exécuté sur un « runner », mis à disposition par Gitlab runner, ce runner est gratuit mais est limitée. Plus tard dans le projet, j'ai du configurer mon propre Gitlab runner en ouvrant un VPS(virtual private server).

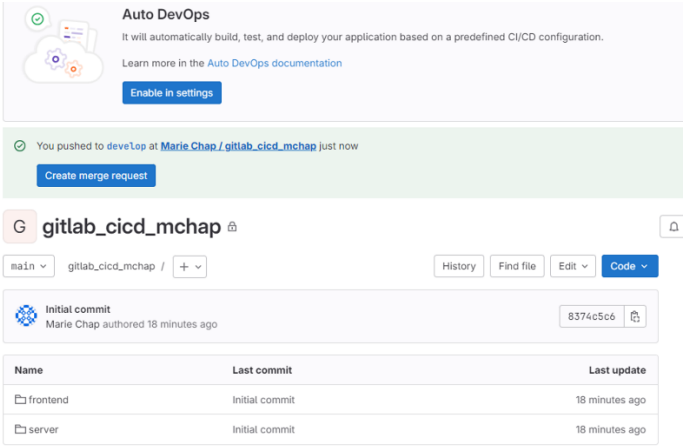
J'ai décidé d'appeler mon environnement « *staging* »(environnement de pré-production ou test), car les instructions de déploiement dessus seront automatique, c'est-à-dire que lorsque l'étape test du pipeline réussira, l'étape de déploiement sur le serveur se fera automatiquement .

Dans la seconde partie du projet, je vais « conteneurisé » la partie frontend et backend, pour séparer les concepts en utilisant Docker et je rajouterai un environnement production avec un déploiement manuel.

III. Initialisation du projet Part.1 :

A. Initialisation du projet sur gitlab

Pour initialiser le projet je créer un nouveau projet sur gitlab



B. Ouverture de VPS sur ovh

Après avoir ouvert un VPS sur OVH(virtual private server), je me connecte dessus et le but de la première connexion au serveur ici, est de créer un lien entre mon VPS et le runner Gitlab.

Pour cela je créer une paire de clé SSH, dont je vais ajouter la clé privé sur Gitlab sous forme de variables masqué et la clé public sur le serveur.

De cette manière, je vais pouvoir utilisé la variable contenant le clé privé et le retourner sur le VPS pour que l'application ait l'autorisation d'y accéder.

Cette variable sera présente sur mon fichier de configuration.

C. Installation de Node.js et PM2 sur le VPS

Maintenant que la connexion au VPS est mise en place, je vais pouvoir y accéder afin d'installer Node.js et que PM2.

Node.js permet d'exécuter du code Javascript côté serveur et gérer les packages (bibliothèque) nécessaire à une application.

PM2 est un gestionnaire de processus pour les applications Node.JS, permettant de démarrer, arrêter et redémarrer les processus. Il permet également de surveiller les performances de l'application en temps réel (monitoring), il offre encore beaucoup de fonctionnalités.

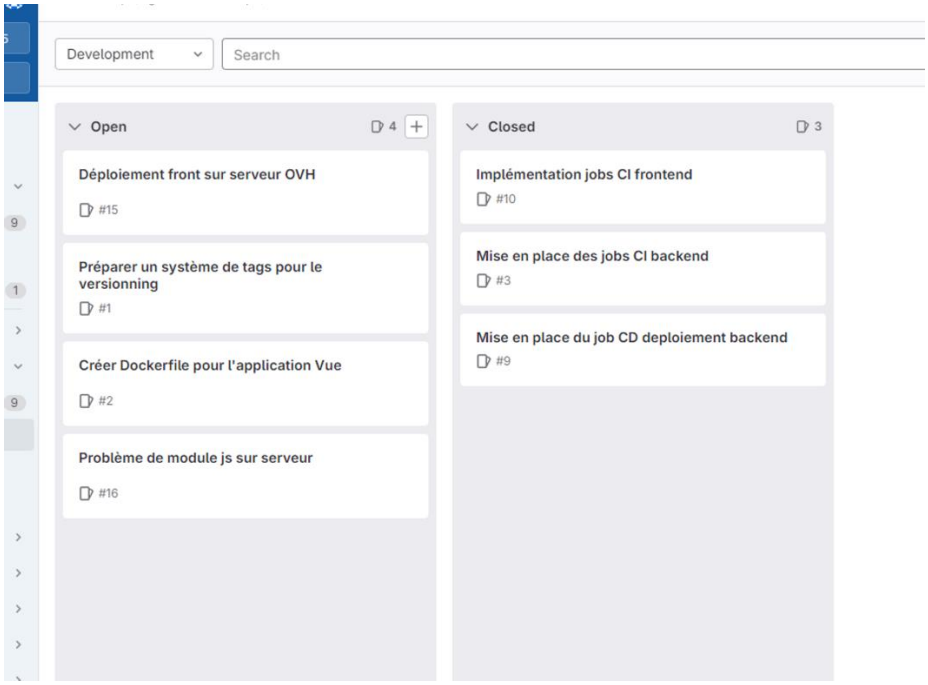
D. Préparer le plan lié aux tâches du projet

Je vais utiliser Gitlab pour la planification de tâche qui met à disposition une interface avec :

- “issues” pour planifier les tâches et les fonctionnalités à venir, -
- “board”, qui permet l’utilisation de tableau de type kanban pour la gestion agile des projets
- “milestones”, qui permet de définir et suivre des objectifs à court et à long terme.

Je vais principalement utilisé les issues et la partie board.

Tâches pour la partie 1 de l’application avant d’implémenter Docker



E. GitFlow :

Voici l'organisation de branche que j'ai choisi d'utiliser sur mon projet :

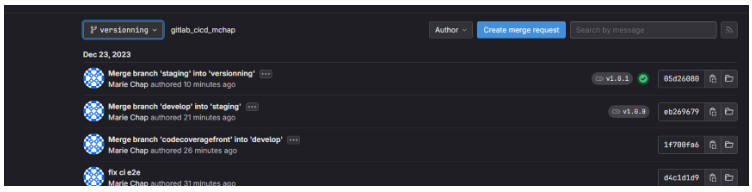
la branche *develop* va me servir à récupérer le code de toutes les branche lié à des fonctionnalités spécifique, le *staging* est une branche qui aura vocation à être déployer dans un environnment dîtes pre-prod ou test avant de pousser sur la branche *main*, qui est une version de ce qui sera mis en production. Une branche *versionning* est mis en place pour garder des version stable des étapes de l'application. Ces version seront reconnaissable avec un système de tags.

F. Préparer un système de tag et une branche versionning

Je vais initialiser un premier tag, pour pouvoir sauvegarder plus tard dans le projet toutes les version qui seront stable, pour cela je me place sur la branche ayant une version qui me convient et le lui donne une numéro de tag version :

```
chuck@ChuckyMaster MINGW64 ~/OneDrive/Bureau/cicd-docker (develop)
$ git tag -a v0.0.0 -m"preparation de tag -versionning"

chuck@ChuckyMaster MINGW64 ~/OneDrive/Bureau/cicd-docker (develop)
$ git push origin v0.0.0
```

Une fois cela fait je stock une première version avec le tag v1.0.0.



Ici nous sommes un peu plus loin dans le projet, j'ai décidé de taguer la version 1.0.0 une fois que le fichier de configuration « pipeline » ait été terminé.

IV. Mise en place CI (Continuous Integration)/CD(Continuous Delivery)

A. CI/CD Backend

Je vais commencer par créer un fichier Gitlab.ci.yml, ce fichier sera lu automatiquement sur GitLab comme fichier de configuration principale CI/CD en tant que pipeline.

Pour débuter je vais lui donner des instructions (jobs) lié à la partie « server » de mon application (backend). Je lui déclaret dans un premier temps quelles sont les étapes :

Ici je vais avoir 3 étapes :

- Check
- Test
- Deploy

Ces étapes seront exécutées dans l'ordre.

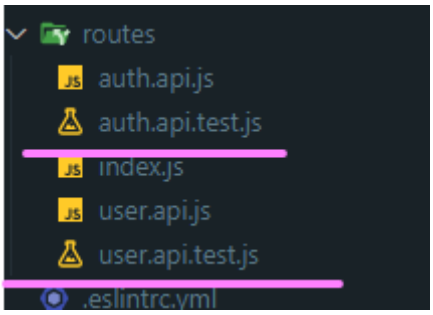
Check contient des tâches pour vérifier le code, dans mon cas j'utilise ESlint un outil de linting pour Javascript, qui analyse le code pour y trouver des problèmes de style de codage.

Test contient les instructions pour lancer les tests unitaires disponible dans l'application,

Ici Jest prépare un environnement de test MongoDB propre pour chaque test unitaire ou d'intégration. Il établit une nouvelle connexion MongoDB pour l'ensemble des tests, nettoie la base de donnée avant chaque test individuel et ferme proprement les ressources après que tous les tests aient été effectué.

```
const { MongoMemoryServer } = require('mongodb-memory-server');
const mongoose = require('mongoose');

let mongo;

beforeAll(async () => {
  mongo = await MongoMemoryServer.create();
  const uri = mongo.getUri();
  await mongoose.connect(uri);
});

beforeEach(async () => {
  const collections = mongoose.connection.collections;
  for (const key in collections) {
    const collection = collections[key];
    await collection.deleteMany();
  }
});

afterAll(async () => {
  await mongoose.disconnect();
  await mongo.stop();
});
```

Ci-dessous, deux tests unitaires avec Supertest testant les routes API.



Pour lancer les test et aller chercher les fichier de test, je lui donne le script à exécuter « npm run test :ci » cette commande est défini dans un fichier package.json de l'application et figure en tant que script disponible

```
"scripts": {
  "start": "node --watch index.js",
  "test:dev": "jest --watch",
  "test:ci": "jest --coverage --ci",
  "lint": "eslint . -c .eslintrc.yml"
},
```

Lorsque que le script est executé, ca lancera la commande de test « jest –coverage –ci », qui est ici une commande propre à la bilbiothèque de test Jest.

J'indique ensuite sur le pipeline comment executé les tests.

```
    <<: *before_script_definition
    script: npm run lint

  security:backend:
    stage: check
    <<: *before_script_definition
    script: npm audit

  test:backend:
    image: node:lts-bullseye
    stage: test
    <<: *before_script_definition
    script: npm run test:ci
```

Deploy contient des instructions pour déployer le code sur un serveur.

Pour mettre en place les instruction de déploiement sur le pipeline, voici ce que font les instructions :

Établi une connexion avec notre serveur distant VPS (je vais me servir de la variables masqué faite en amont contenant la clé privé).

Une fois la connexion établie, il créer un dossier ssh et lui indique que l'hôte est reconnu.

Ensuite je lui indique quelle répertoire est à copié sur le serveur distant VPS, en indiquant à chaque fois un chemin.

Une commande installe ensuite les dépendances nécessaire en omettant les dépendances de développement, la dernière instruction se connecte au serveur et utilise PM2 pour démarrer ou redémarrer l'application.

Une règle est ajouté stipulant que l'étape de déploiement ne doit s'opérer que lorsque le pipeline est lancé sur la branche par défaut. Dans mon cas, l'étape Deploy ne sera disponible que sur la branche par défaut « *staging* ».

Ici le déploiement se fera automatiquement.

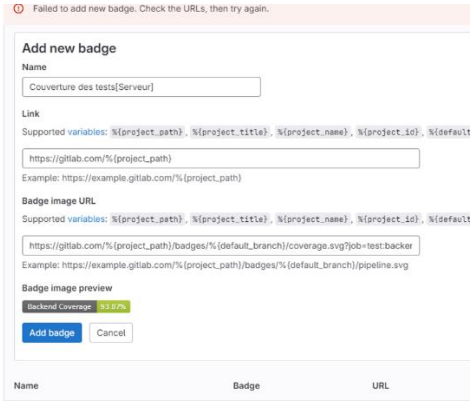
```yaml
deploy:
  stage: deploy
  image: node:lts-bullseye
  environment:
    name: staging
  inherit:
    default: false
  before_script:
    - eval $(ssh-agent -s)
    - echo $SSH_KEY_VPS | base64 -d | ssh-add -
  script:
    - mkdir -p ~/.ssh
    - touch ~/.ssh/known_hosts
    - ssh-keyscan -H $SERVER_IP >> ~/.ssh/known_hosts
    - scp -r ./server $SERVER_USER@$SERVER_IP:/var/www
    - ssh $SERVER_USER@$SERVER_IP "cd /var/www/server && npm install --omit=dev"
    - ssh $SERVER_USER@$SERVER_IP "cd /var/www/server && pm2 startOrRestart ecosystem.config.js --env production && pm2 save"
  rules:
    - if: "$CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH"
```

B. Code coverage backend

Après avoir mis en place des test unitaires, je mets en place une couverture de code.

Pour ce faire, au niveau du fichier de configuration à la partie de test, j'ajoute une regex qui permettra d'extraire des données spécifique. Je lui indique ensuite le format de sortie avec Cobertura qui analyse les résultats et fourni des rapports détaillé sur la couverture du code par les tests.

Pour avoir un aperçu au premier coup d'œil sur GitLab il est possible de créer des badges, ces badges offrent un visuel en temps réel lorsqu'une pipeline contenant des informations concernant la couverture de code est en cours d'exécution par exemple.



Sur les logs du pipeline nous avons également accès à une interface plus détaillés sur le code coverage :

```
File             | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------|---------|----------|---------|---------|-------------------
All files        |   93.87 |       75 |     100 |   93.87 |
 database/models |     100 |      100 |     100 |     100 |
  user.model.js  |     100 |      100 |     100 |     100 |
 env/keys        |     100 |      100 |     100 |     100 |
  index.js       |     100 |      100 |     100 |     100 |
 routes          |   93.02 |       75 |     100 |   93.02 |
  auth.api.js    |   94.73 |       75 |     100 |   94.73 | 20
  user.api.js    |   91.66 |       75 |     100 |   91.66 | 18,33
-----------------|---------|----------|---------|---------|-------------------
================================ Coverage summary ================================
Statements   : 93.87% ( 46/49 )
Branches     : 75% ( 6/8 )
Functions    : 100% ( 4/4 )
Lines        : 93.87% ( 46/49 )
==================================================================================
Test Suites: 2 passed, 2 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        1.781 s
Ran all test suites.
```

Voici comment comprendre le code coverage :

Statements : Indique 93.87% (46/49) - des déclarations du code ont été exécuté au moins une fois. Il y avait 49 déclaration au total et 46 d'entre elles ont été exécutées.

Branches : 75% (6/8) – 75% des branches de contrôle de flux (if , else etc.) ont été traversées pendant les tests. Il y avait 8 branches au total, 6 ont été exécutées.

Functions : 100% (4/4) 100% des fonctions ont été appelées pendant les test et 4/4 ont été appelées.

Lines : 93.87% (46/49) – 93.87% des lignes de code ont été exécutées pendant les tests.

C. Ajout CI/CD Frontend

Pour le frontend, je vais rajouter l'étape (stage) de construction sur le pipeline (build) et ajouter un job « build :frontend ».

Ce job, est écrit de la manière suivante :

```
build:frontend:
  stage: build
  extends:
    - .before_script_template_frontend
    - .cache_dependencies_frontend
  cache:
    policy: pull
  script: npm run build
  artifacts:
    when: on_success
    expire_in: 1 day
    paths:
      - /frontend/dist
```

Je lui indique des jobs cachés fait en amont, gérant les cache, le stage et le script qu'il va devoir exécuté : « npm run build ».

Une fois le script de build exécuté, l'application est disponible sur un dossier « dist » (pour distribution).

Ce dossier correspond à la version « build » de l'application, celle qui est prête à être déployer. J'indique ici sur le pipeline qu'il faut donc récupéré les dossiers et fichiesr à l'intérieur de dist.

Ce qui correspond ici à la clé artifacts.

Je vais ajouter un test e2e avec Cypress dans le pipeline.

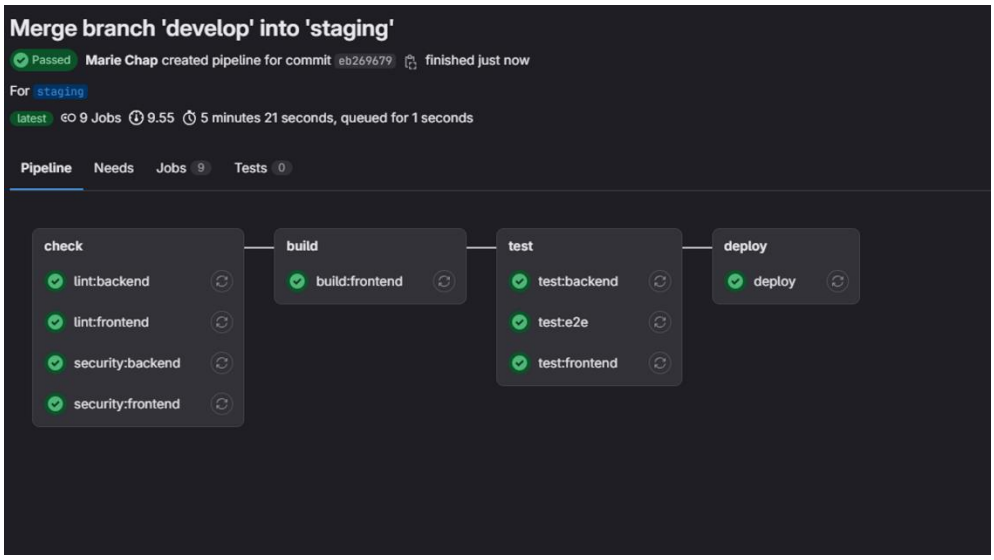
Les test e2e ne peuvent être exécuté que lorsque le build frontend s'est bien déroulé et est réussie car ce test doit être au plus proche de l'expérience utilisateur.

Déploiement :

Pour le déploiement du front je vais indiquer l'artéfact contenant le build du front et le copier sur le serveur en rajoutant cette ligne dans le job « deploy ».

```
- ssh-keyscan -H $SERVER_IP >> ~/.ssh/known_hosts
- scp -r ./server $SERVER_USER@$SERVER_IP:/var/www
- scp -r ./frontend $SERVER_USER@$SERVER_IP:/var/www/dist
- ssh $SERVER_USER@$SERVER_IP "cd /var/www/server && npm install --omit=
- ssh $SERVER_USER@$SERVER_IP "cd /var/www/server && pm2 startOrRestart
ules:
- if: "$CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH"
```

Voilà un exemple du pipeline exécuté avec les 4 stages (étapes)



V. Conteneurisation de l'application avec Docker Part.2

A. Introduction à Docker

Docker est un outil open-source écrit en langage Go et créé en 2012 par trois ingénieurs français. Il permet d'empaqueter une application et ses dépendances dans un conteneur isolé, qui pourra être exécuté sur n'importe quel environnement.

Voici quelques avantages de Docker :

- Permet une utilisation plus efficiente des ressources d'un système
- Les applications conteneurisée utilisent beaucoup moins de ressource que des machines virtuelles
- Un conteneur Docker est garanti d'être identique quel que soit le système
- Les dépendances ou les configurations d'un conteneur sont isolés et n'affectera aucun des autres conteneurs.

1. Les images Docker

Une image Docker est un schéma qui contient les instructions pour créer un conteneur Docker. Pour créer une image, il faut utiliser un fichier spécifique appelé Dockerfile qui a une syntaxe particulière permettant de définir les étapes nécessaire à la création de l'image.

Chaque instruction dans un Dockerfile permet de créer une couche dans l'image.

2. Les conteneurs

Un conteneur est une instance d'une image en cours d'exécution.

Par défaut un conteneur est isolé des autres conteneurs et de la machine hôte.

Ils permettent d'isoler une ou plusieurs applications sur le même environnement quel que soit la machine physique et le système d'exploitation utilisé (Windows,

linux). Ils peuvent containerisée les applications les plus complexes, ils sont léger, utilisables sur n'importe quel environnement, autonomes car ils sont isolés. Ils sont scalables, Des outils permettent de d éployer des centaines ou milliers de conteneurs en quelques minutes et enfin ils sont sécurisé, l'isolation par défaut des conteneurs apporte une grande sécurité des environnements déployés.

3. Exécuter les conteneurs avec Docker-compose

Docker Compose est l'outil Docker permettant de lancer des applications multi-conteneurs. Il permet de lancer tous les services (*sur la configuration de docker compose un service représente un conteneur, par exemple : un conteneur ayant les services d'authentification, un autre pour la base de données …etc*) d'une application en une commande.

B. Création d'un fichier reverse proxy

Avant de créer mes images Docker, j'ai besoin d'une certaine configuration qui va devoir me servir à la création des images Docker.

Cette configuration déclare un serveur qui écoute sur le port 80 et si on essaie d'atteindre /api/ le serveur redirigera sur la route http://server, si ça ne commence pas par /api/ l'application Vue.js (frontend) sera retournée. On appelle cela un reverse-proxy.

Nginx.conf

C. Création d'image – Dockerfile

Pour cette deuxième partie, afin de conteneurisé d'un côté le frontend et de l'autre le backend, je créer pour chaque partie un Dockerfile.

Ici le Dockerfile pour le frontend :

```
FROM node:lts-alpine as build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build-only

FROM nginx:stable-alpine
COPY --from=build /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
ENTRYPOINT []
CMD ["nginx", "-g", "daemon off;"]
```

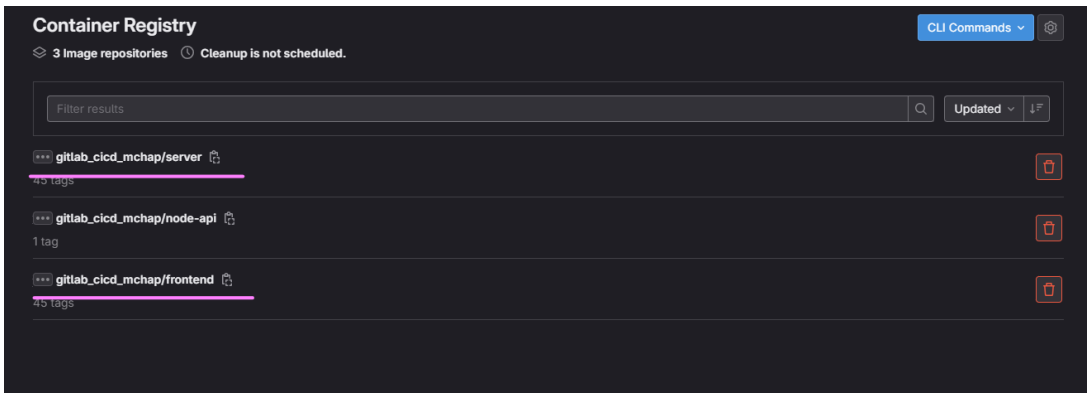
Ce fichier contient 2 étapes (à partir de « FROM »).

Il récupère une image lts-alpine de Node.js, (sur cette ligne je place un alias « as build » pour pouvoir le récupérer sur la seconde étape) se positionne dans le dossier /app, on copy le package json, installe les dépendances puis copy le reste du dossier et lance le build.

La seconde étape réutilise le build de la première étape, récupère le contenu de app/dist et le place dans /usr/share/nginx/html. Ensuite il récupère un fichier de configuration nginx.conf (reverse-proxy).

Ces images, une fois créer devront être pousser dans un registre de conteneur. Je vais utiliser le registre de conteneur mis à disposition par Gitlab.

Ce registre me permet de stocké mes propres image et de les réutiliser sur mon application avec Docker.



D. Déploiement avec Docker compose

Je vais maintenant préparer le déploiement sur le serveur avec docker-compose et ajouter une nouvelle étape (stage) et une nouvelle instruction (job).

Voilà comment va fonctionner le job « deploy », le lui indique :

- L'image docker à récupérer
- L'environnement (de production ou staging)
- Une règle pour lancer le job seulement si l'on est sur une certaine branche (staging)
- Une commande pour lire ma clé SSH(créer plus tôt, lorsque j'ai établie une connexion entre mon runner Gitlab et le serveur VPS) et l'ajouter au trousseau de l'agent SSH

Ensuite un script copie le fichier docker-compose.prod.yml sur le serveur, se connecte au registre des conteneur de Gitlab et récupère les images les plus récentes, puis les conteneur sont créé et démarrer.

```
deploy_staging:
  stage: deploy
  image: alpine
  environment:
    name: staging
    url: http://$SERVER_STAGING_IP
  before_script:
    - apk add --no-cache openssh-client
    - eval $(ssh-agent -s)
    - echo $SSH_KEY_PRODUCTION | base64 -d | ssh-add -
  script:
    - mkdir -p ~/.ssh
    - touch ~/.ssh/known_hosts
    - ssh-keyscan -H $SERVER_STAGING_IP >> ~/.ssh/known_hosts
    - scp docker-compose.prod.yml root@$SERVER_STAGING_IP:/#docker-compose.prod.yml
    - >
      ssh root@$SERVER_STAGING_IP "
      echo $CI_JOB_TOKEN | docker login -u "$CI_REGISTRY_USER" --password-stdin "$CI_REGISTRY" &&
      docker-compose -f docker-compose.prod.yml pull &&
      docker-compose -f docker-compose.prod.yml up -d --force-recreate
      "
  rules:
    - if: $CI_COMMIT_BRANCH == 'staging'
```

E. Ajout d'un environnement de production au déploiement

Pour ajouter un environnement de production, j'ai également ouvert un VPS. J'ai répéter toutes mes premières opérations pour pouvoir me connecter dessus avec une clé SSH.

Sur le pipeline, maintenant j'ai deux « jobs » de déploiement.

J'ai ajouté quelques nouvelles instruction :

Une nouvelle variables contenant l'adresse du serveur VPS

```
variables:
  FRONTEND_IMAGE: $CI_REGISTRY_IMAGE/frontend
  BACKEND_IMAGE: $CI_REGISTRY_IMAGE/server
  SERVER_STAGING_IP: 51.254.32.14
  SERVER_PRODUCTION_IP: 151.80.133.130
```

Ainsi qu'une instruction :

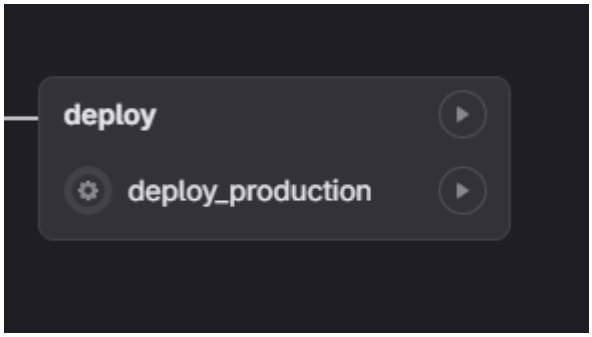
« when : manual » qui indique que le déploiement se fera uniquement manuellement.

```
deploy_production:
  stage: deploy
  image: alpine
  when: manual
  environment:
    name: production
    url:http://$SERVER_PRODUCTION_IP
```

J'ai également ajouté une nouvelle règle pour afficher ce job seulement si on est sur la branche « main » est également ajouté.

```
rules:
  - if: $CI_COMMIT_BRANCH == 'main'
```

Je vérifie que le déploiement se fera bien manuellement, si l'on est sur la branche « main »



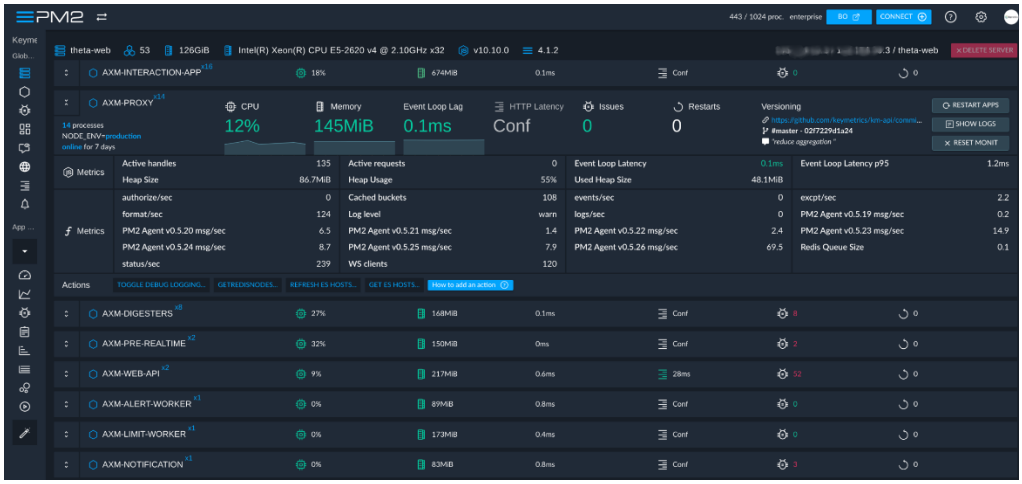
VI. Monitoring

Le monitoring permet de surveiller la performance des applications, le monitoring a pour but d'assurer la disponibilité ainsi que la performance et équipements physiques (réseaux serveurs, routeurs, onduleur, bornes wifi, etc…).

Dans mon cas, au début de mon projet j'ai du installer PM2 afin de lancer mon application Node.JS.

PM2 est un gestionnaire de processus, permettant également le monitoring. Il met à disposition une interface détaillé de l'état de l'application. Cette interface fournit des graphiques et des métriques pour comprendre les performances.

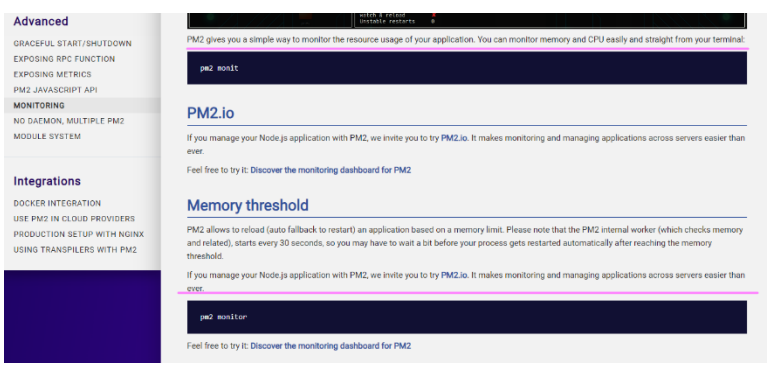
Voilà à quoi ressemble l'interface monitoring PM2 :



VII. Veille informatique

Durant la création de ce projet j'ai été amené à faire plusieurs veille, d'une part pour ajouter des tags automatique en rapport avec le versionning et d'une autre part, une recherche sur PM2 car le sujet m'a beaucoup intéressée.

Je vais donc vous présenter ma veille sur PM2.



La documentation PM2 est très agréable à lire, ici la documentation pour avoir accès à son interface graphique PM2

J'ai de mon côté suivi les étapes afin de me rendre sur mon interface graphique .

Voici un extrait d'article sur PM2

PM2 is a popular daemon process manager for **Nodejs** with a complete feature set for a production environment, that will help you manage and keep your application online 24/7.

A process manager is a "**container**" for applications that facilitates deployment, enables you to manage (start, restart, stop, etc..) the application at runtime, and provides for high availability.

In this article, we will show how to monitor **Nodejs** applications using **PM2** from the command line and on the web. This guide assumes you already have **PM2** installed on your Linux system and you are already running your Nodejs application using it. Otherwise, check out:

« PM2 est un gestionnaire de processus pour Node.JS ayant beaucoup de fonctionnalités pour un environnement de production, il vous aide à gérer et avoir l'œil sur votre application 24/7.

Un gestionnaire de processus est un conteneur pour les application simplifiant le déploiement, vous permettant de gérer le démarrage, le redémarrage, stop etc.

Dans cet article nous montrerons comment utilisé le monitor sur une application Nodejs en utilisant PM2 en ligne de command et sur le web. Ce guide part du principe que vous ayez PM2 installé sur votre system et que Node.js tourne sur votre application. »