Virtualisation légère – TP n° 5

DevOps, intégration et déploiement continu

Pierre-Olivier *nemunaire* Mercier

Mercredi 16 novembre 2022

Abstract

Durant ce nouveau TP, nous allons jouer les DevOps et déployer automatiquement des services !

Les exercices de ce cours sont à rendre au plus tard le mardi 29 novembre 2022 à 23 h 42. Consultez les sections matérialisées par un bandeau jaunes et un engrenage pour plus d'informations sur les éléments à rendre.

Sommaire

1 Le mouvement DevOps

Jusqu'à récemment, et encore dans de nombreuses entreprises, les développeurs et les administrateurs système faisaient partie de deux équipes différentes : les uns développant sur leurs machines avec les dernières bibliothèques, utilisant les derniers frameworks à la mode, sans se préoccuper de la sécurité (ils travaillent en `root` ou avec `sudo` ;)), tandis que les autres tentaient tant bien que mal de déployer ces services avec les contraintes opérationnelles en tête.
Ces contraintes : tant liées à la **sécurité** (il faut s'assurer qu'un service n'utilise pas une bibliothèque vulnérable par exemple, donc soit utilisé sur un système à jour, et qu'il ne tourne pas en `root`), qu'à la **disponibilité** (si le service est mal codé est contient beaucoup de fuites mémoire, il ne faut pas que les autres services présents sur la même machine en pâtissent).

Une guerre faisait donc rage entre les développeurs qui ne comprenaient pas que les administrateurs système ne pouvaient pas maintenir autant de versions d'une bibliothèque qu'il y avait de services : par exemple dans le cas de plusieurs services en PHP, on pouvait leur demander de déployer des applications utilisant la version 5.6, et la 7.2 pour d'autres, … lorsqu'il y avait des incompatibilités mineures et plus personne pour s'occuper de la maintenance d'un vieux service toujours utilisé.

Le même principe est aussi valable pour Python, Ruby, … : les développeurs ont toujours eu tendance à vouloir utiliser les dernières améliorations d'un langage, mais les administrateurs système n'ont alors pas de paquets stables dans la distribution. En effet, les distributions stables telles que Debian, RedHat ou CentOS ont des cycles de vie assez long et se concentrent plus sur la stabilité.
Cette stabilité est obtenue grâce à l'utilisation de versions éprouvées des langages et des bibliothèques, qui assurent un temps de maintenance et de recherche de bugs réduit aux équipes opérationnelles. Si un projet fonctionne bien avec une version donnée d'une de ces distributions, on peut être assez confiant sur le fait que ce sera toujours le cas (du moins tant que la distribution assure le support de sa version).

Le but du DevOps est donc de retrouver une certaine fluidité entre le développement et l'exploitation. Il s'agit d'un mouvement qui vise à ce que les développeurs, sans avoir à gérer au quotidien la maintenance des serveurs de production, soient davantage impliqués dans les opérations de déploiement. Cela passe notamment par la prise en compte de l'environnement de déploiement dès la phase de conception du projet, puis dès les premiers développements, des tests automatisés sont réalisés directement dans un environnement proche de la production.

Il en résulte moins de friction entre les deux équipes. Les développeurs étant par ailleurs amenés à écrire des recettes de déploiement, tels que des playbooks Ansible ou bien encore des conteneurs Docker.

Chez Google (et d'autres entreprises qui ont depuis repris l'idée), des équipes sont chargées de développer la fiabilité des systèmes d'information de production. Ce sont les équipes SRE, pour Site Reliability Engineering. On confie alors complètement la responsabilité de l'environnement de production aux développeurs qui sont chargés de l'automatiser. Au-delà de l'automatisation des déploiements de services, il s'agit ici de développer des mécanismes permettant aux systèmes de réagir face aux situations telles que les montées en charges, les pannes, …

! Attention par contre aux entreprises qui recrutent un profil DevOps, car cela a autant de sens que recruter un développeur Scrum ou un développeur cycle en V : DevOps est une méthodologie. Les entreprises qui recrutent un DevOps recherchent généralement quelqu'un qui fera à la fois du développement logiciel d'un côté et de l'administration système de l'autre : une situation

! généralement assez difficile à vivre. Alors qu'au contraire, la mouvance DevOps doit être prise au sérieux par l'ensemble des développeurs. Lors d'un entretien d'embauche pour ce genre de poste, assurez-vous bien de ne pas être le seul à faire du DevOps.

1.1 Intégration continue

L'**intégration continue** est la première brique à mettre en place : le but est de compiler automatiquement chaque commit dans un environnement proche de celui de production, puis de lancer les suites de tests du logiciel.

Cela permet de détecter les problèmes au plus tôt dans le cycle de développement, mais cela permet également d'améliorer la qualité du code sur le long terme, car on peut y ajouter facilement des outils qui vont se charger automatiquement de réaliser des analyses : cela peut aller de la couverture des tests, à de l'analyse statique ou dynamique de binaire, en passant par la recherche de vulnérabilités ou de mauvaises pratiques de programmation.

À la fin du processus, il est courant d'exporter les produits de compilation (tarballs, paquets, ISO, conteneurs, ...) ainsi que les journaux et rapports vers un dossier accessible. Cela permet ainsi aux développeurs de voir les problèmes et de pousser les analyses avec leurs propres outils.

Sans déploiement continu (la section suivante), c'est également ces produits de compilation que les administrateurs système vont déployer sans peine, lorsque les développeurs considéreront avoir atteint un jalon, une version stable.

1.2 Déploiement continu

Une fois tous les tests passés et les objets produits (on parle d'*artifact* ou d'*assets*), il est possible de déclencher un déploiement : il s'agit de rendre accessible aux utilisateurs finaux le service ou les objets.

Dans le cas d'un programme à télécharger (Python, VLC, MariaDB, ...), on va placer les paquets sur le site internet, éventuellement mettre à jour un fichier pointant vers la dernière version (pour que les utilisateurs aient la notification).

Ou bien dans le cas d'un service en ligne (GitHub, Netflix, GMail, ...), il s'agira de mettre à jour le service.

Parfois les deux seront à faire : à la fois publier un paquet ou un conteneur et mettre à jour un service en ligne : par exemple, le serveur Synapse du protocole de messagerie Matrix ou encore Gitlab, tous deux publient des paquets à destination de leurs communautés, et ils mettent à jour leur service en ligne.

Il existe pour cela de très nombreuses stratégies : lorsque l'on n'a pas beaucoup de trafic ni beaucoup de machines, on peut simplement éteindre l'ancien service et démarrer le nouveau, si ça prend quelques millisecondes en étant automatisé, cela peut être suffisant compte tenu du faible trafic.

Lorsque l'on a un trafic élevé, de nombreux clients et donc que le service est réparti sur plusieurs machines, on ne peut pas se contenter de tout éteindre et de tout rallumer. Déjà parce que trop de visiteurs vont se retrouver avec des pages d'erreur, et aussi parce qu'en cas de bug logiciel qui n'aurait pas été vu malgré les étapes précédentes, cela pourrait créer une situation catastrophique (imaginez qu'on ne puisse plus valider une commande sur Amazon à cause d'une ligne commentée par erreur !).
On va donc privilégier un déploiement progressif de la nouvelle version (que l'on va étendre sur plusieurs

minutes, heures ou mêmes jours), en éteignant tour à tour les instances, et en veillant à ce que les métriques (voir la section suivante !) soient constantes.

1.3 Monitoring et supervision

Une fois déployé, le service peut avoir des ratés, alors il convient de le surveiller afin d'être le plus proactif possible dans la résolution des problèmes. La pire situation est celle dans laquelle c'est un utilisateur qui nous informe d'un problème… (sur Twitter !?)

Nous avons réalisé précédemment une partie collecte de métriques, avec nos conteneurs TICK. Nous n'allons donc pas nous en occuper ici.

Notez tout de même qu'il y a deux grandes catégories de logiciels de supervision :

Basée sur des états comme Nagios, Zabbix, … : ces logiciels vont simplement réaliser des séries de tests définis, à intervalles réguliers et contacter l'administrateur d'astreinte dès qu'un test ne passe plus de manière persistante.

Il y a rarement beaucoup d'intelligence ou d'anticipation automatique dans ces outils.

Basée sur les métriques comme ELK, Prometheus, InfluxDB, … : dans la stack TICK que nous avons mis en place précédemment, nous avions placé un agent sur la machine que nous souhaitions analyser. Outre les graphiques présentés dans Chronograf, le dernier outil que l'on n'avait pas configuré était Kapacitor, qui permet après avoir analysé les données, d'alerter en fonction d'une évolution.

L'instrumentation d'une application est une bonne manière de faire remonter des métriques (combien de clients actuellement connectés, combien de messages/transactions traités, …). Ce sont autant d'informations que l'on peut faire remonter dans sa base de données de métriques.

La différence entre les deux techniques est que nagios va vous alerter à partir d'un certain seuil que vous aurez préalablement défini (s'il reste moins de 10 % d'espace disque par exemple), tandis que Kapacitor va tenter d'interpréter les indicateurs (et donc vous alerter seulement si la courbe représentant l'espace disque disponible augmente de telle sorte qu'il ne reste plus que quelques heures avant d'être saturé).

Sur la base de ces indicateurs, il est possible d'engager des opérations automatiques, comme par exemple la provision de nouvelles machines pour épauler un service distribuable, qui est proche de la surcharge, acheter de l'espace de stockage supplémentaire auprès du fournisseur, …

Enfin, citons le Chaos Monkey, conçu par Netflix, qui est un programme qui va casser de manière aléatoire des éléments de l'environnement de production. Le but est de provoquer sciemment des pannes, des latences, … à n'importe quel niveau du produit, afin d'en tester (brutalement certes) sa résilience. Cela oblige les développeurs, les opérationnels et les architectes à concevoir des services hautement tolérant aux pannes, ce qui fait que le jour où une véritable panne survient, elle n'a aucun impact sur la production (enfin on espère !).

2 But du TP

Nous allons nous mettre aujourd'hui dans la peau d'une équipe DevOps et réaliser une solution complète d'intégration/déploiement continu (le fameux CI/CD, pour *Continuous Integration* et *Continuous Delivery*).

Le résultat attendu d'ici la fin de cette partie sera de mettre en place toutes les briques décrites dans la section précédente.

Nous allons pour cela automatiser le projet `youp0m`.

Il est également attendu que vous rendiez un playbook Ansible, permettant de retrouver un environnement similaire. Car on pourra s'en resservir au prochain cours.

Dans un premier temps, on voudra juste compiler notre projet, pour s'assurer que chaque commit poussé ne contient pas d'erreur de compilation (dans l'environnement défini comme étant celui de production). Ensuite, on ajoutera quelques tests automatiques. Puis nous publierons automatiquement le binaire `youp0m` comme fichier associé à un tag au sein de l'interface web du gestionnaire de versions.

Enfin, `youp0m` produisant une image Docker, en plus de publier le binaire, nous publierons l'image produite sur un registre Docker. C'est à partir de cette image Docker que l'on va commencer à déployer automatiquement…

2.1 Préparer le terrain

Tous les déploiements sont à faire sur votre machine en utilisant des conteneurs Docker, qui seront regroupés au sein de réseaux Docker. Cela vous permettra d'utiliser la résolution de noms entre vos conteneurs.

Dans votre playbook Ansible, vous pourrez procéder ainsi (il s'agit d'un exemple qu'il faut comprendre et adapter par rapport à la suite) :

```
- name: Create virli network
  docker_network:
    name: virli3
```

Étant donné que votre machine ne dispose pas de domaine sur Internet et que l'on va essayer de simplifier au maximum l'installation, vous devriez ajouter cette ligne à votre fichier `/etc/hosts` (ou `\Windows\System32\drivers\etc\hosts`) :

```
127.0.0.1   gitea droneci
```

Cette ligne va vous permettre de résoudre les noms des conteneurs. Cela permettra aux requêtes OAuth de se faire de manière transparente pour vous lorsque vous serez dans votre navigateur.

2.2 Gestionnaire de versions

Avant de pouvoir commencer notre aventure, il est nécessaire d'avoir un gestionnaire de versions. Nous allons ici utiliser Git.

2.2.1 Problématique du stockage des produits de compilation

Outre les interfaces rudimentaires fournies au-dessus de Git (gitweb[1], gitolite[2], ...), il y a de nombreux projets qui offrent davantage que le simple hébergement de dépôts. Vous pouvez voir sur GitHub notamment qu'il est possible d'attacher à un tag un certain nombre de fichiers. Mais cela ne s'arrête pas là puisque depuis 2020 pour GitHub et 2016 pour GitLab, ces gestionnaires de versions intégrent carrément un registre Docker.

En effet, la problématique du stockage des produits de compilation est vaste. Si au début on peut se satisfaire d'un simple serveur web/FTP/SSH pour les récupérer manuellement, on a vite envie de pouvoir utiliser les outils standards directement : `docker pull ...`, `npm install ...`, ...

Des programmes et services se sont spécialisés là-dedans, citons notamment Artifactory ou Nexus Repository et bien d'autres.

2.2.2 Installation et configuration

Aller c'est parti ! première chose à faire : installer et configurer Gitea.

Nous allons utiliser l'image : `gitea/gitea`.

Votre playbook ressemblera à quelque chose comme ça :

```
- name: Create a volume for storing repositories
  docker_volume:
    name: gitea-data

- name: launch gitea container
  docker_container:
    name: gitea
    image: "gitea/gitea:{{ version }}"
    volumes:
      - gitea-data:/data
      - /etc/localtime:/etc/localtime:ro
      - /etc/timezone:/etc/timezone:ro
    state: started
    restart_policy: unless-stopped
    memory: 1G
    memory_swap: 1G
    networks:
      - name: gitea_net
    published_ports:
      - "2222:22"
      - "3000:3000"
    env:
      RUN_MODE: "prod"
      DOMAIN: "gitea"
      SSH_DOMAIN: "gitea"
      INSTALL_LOCK: "true"
      GITEA__security__SECRET_KEY: "{{ secret_key }}"
      GITEA__security__INTERNAL_TOKEN: "{{ internal_token }}"
```

[1] https://git.wiki.kernel.org/index.php/Gitweb
[2] https://github.com/sitaramc/gitolite

Vous trouverez plus d'infos sur cette page : https://docs.gitea.io/en-us/install-with-docker/.

Une fois le conteneur lancé, vous pouvez accéder à l'interface de votre gestionnaire de versions sur le port 3000 de votre machine (à moins que vous n'ayez opté pour un autre port).

Vous pouvez ajouter un nouvel administrateur avec la commande suivante :

```
docker exec -u git gitea gitea admin user create --username "${USER}" --admin \
  --must-change-password=false --random-password --email "${USER}@exmpl.tf"
```

Notez le mot de passe généré pour ensuite vous y connecter.

(Vous n'êtes pas tenus d'intégrer la création de l'administrateur initial dans votre playbook.)

2.3 Logiciel d'intégration continue

De nombreuses solutions sont disponibles sur Internet, la plupart du temps gratuites pour les projets libres (Travis CI, CircleCI, ...).

Mais nous allons déployer notre propre solution, en utilisant Drone CI. C'est une solution d'intégration continue libre et moderne, conçue tout autour de Docker. Idéale pour nous !

Deux conteneurs sont à lancer : nous aurons d'un côté l'interface de contrôle et de l'autre un agent (*runner* dans le vocabulaire de Drone) chargé d'exécuter les tests. Dans un environnement de production, on aura généralement plusieurs agents, et ceux-ci seront situés sur des machines distinctes.

2.3.1 Interface de contrôle et de dispatch des tâches

La documentation du projet est extrêmement bien faite, suivons la marche à suivre pour relier Gitea à Drone.

Drone va avoir besoin d'authentifier les utilisateurs afin d'accéder aux dépôts privés (avec l'autorisation des utilisateurs). Pour cela, comme l'indique la documentation de Drone, on va utiliser OAuth2 : dans Gitea, il va falloir créer une *application OAuth2*. Le formulaire de création se trouve dans la configuration du compte utilisateur, sous l'onglet *Applications*.

Drone aura également besoin d'une URL de redirection. Dans notre cas, ce sera : `http://droneci/login`.

Voici à quoi pourrait ressembler le playbook Ansible démarrant notre conteneur Drone :

```
- name: Launch drone container
  docker_container:
    name: droneci
    image: drone/drone:2
    volumes:
      - /var/lib/drone:/data
    state: started
    restart_policy: unless-stopped
    memory: 1G
    memory_swap: 1G
```

```
    networks:
      - name: drone_net
      - name: gitea_net
    published_ports:
      - "80:80"
    env:
      DRONE_GITEA_CLIENT_ID: "{{ client.id }}"
      DRONE_GITEA_CLIENT_SECRET: "{{ client.secret }}"
      DRONE_GITEA_SERVER: "http://gitea:3000"
      DRONE_RPC_SECRET: "{{ shared_secret }}"
      DRONE_SERVER_HOST: "droneci"
      DRONE_SERVER_PROTO: "http"
```

C'est à vous de définir un `shared_secret`, il s'agit d'une chaîne aléatoire qui permettra aux *Runner*s (section suivante) de s'authentifier.

Une fois lancé, rendez-vous sur l'interface de DroneCI : http://droneci/

Vous serez automatiquement redirigé vers la page d'authentification de Gitea, puis vers l'autorisation OAuth d'accès de Drone à Gitea. Il faut bien évidemment valider cette demande, afin que Drone ait accès à nos dépôts.

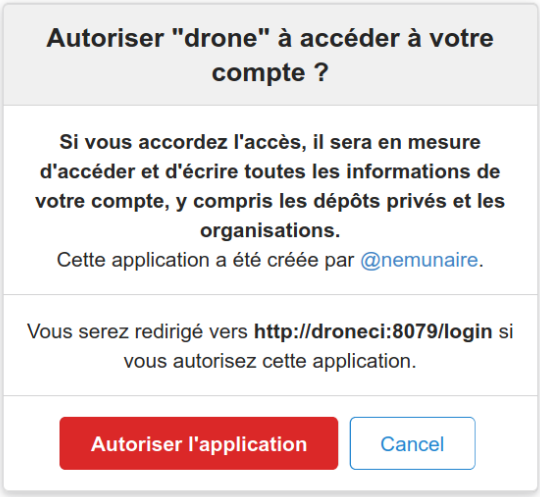

Figure 1: OAuth Drone

2.3.2 Runner

Notre conteneur `droneci` est uniquement une interface graphique qui va centraliser d'un côté les nouveaux commits à traiter, et de l'autre les résultats retournés par les agents chargés d'exécuter les tâches.

Il serait impensable d'exécuter arbitrairement du code en parallèle d'une application privilégiée (ici, notre conteneur `droneci` a accès aux dépôts potentiellement privés de Gitea). Les agents qui sont amenés à traiter du code arbitraire s'exécutent à part et peuvent être de différents types.

Nous allons lancer un *runner* Docker : il s'agit d'un type d'agent qui va exécuter nos étapes de compilation dans des conteneurs Docker (oui, quand on disait que Drone était conçu autour de Docker, ce n'était pas pour rire !)

Voici à quoi pourrait ressembler le playbook Ansible démarrant notre agent Drone :

```
- name: Launch drone runer
  docker_container:
    name: droneci-runner
    image: "drone/drone-runner-docker:1"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    state: started
    restart_policy: unless-stopped
    memory: 2G
    memory_swap: 2G
    networks:
      - name: drone_net
    env:
      DRONE_RPC_PROTO: "http"
      DRONE_RPC_HOST: "droneci"
      DRONE_RPC_SECRET: "{{ shared_secret }}"
      DRONE_RUNNER_CAPACITY: "2"
      DRONE_RUNNER_NAME: "my-runner"
      DRONE_RUNNER_NETWORKS: "drone_net,gitea_net"
```

On remarquera que l'on partage notre socket Docker : l'exécution de code arbitraire n'aura pas lieu directement dans le conteneur, en fait il s'agit d'un petit orchestrateur qui lancera d'autres conteneurs en fonction des tâches qu'on lui demandera. Le code arbitraire sera donc toujours exécuté dans un conteneur moins privilégié.

L'environnement étant prêt, il ne reste plus qu'à nous lancer dans nos projets !

2.4 Intégration continue

Une fois Gitea et Drone installés et configurés, nous allons pouvoir rentrer dans le vif du sujet : faire de l'intégration continue sur notre premier projet !

L'idée est qu'à chaque nouveau *commit* envoyé sur le dépôt, Drone fasse une série de tests, le compile et publie les produits de compilation.

2.4.1 Créez un dépôt pour `youp0m`

Reprenons les travaux déjà réalisés : nous allons notamment avoir besoin du `Dockerfile` que nous avons réalisé pour le projet `youp0m`.

Après avoir créé (ou migré pour les plus malins !) le dépôt `youp0m`[a] dans gitea, synchronisez les dépôts dans Drone, puis activez la surveillance de `youp0m`.

[a] https://git.nemunai.re/nemunaire/youp0m

Que fait Drone pour « surveiller » un dépôt ?

Grâce aux permissions de que Drone a récupéré lors de la connexion OAuth à Gitea, il peut non seulement lire et récupérer le code des différents dépôts auxquels vous avez accès, mais il peut aussi changer certains paramètres.
L'activation d'un dépôt dans Drone se traduit par la configuration d'un *webhook* sur le dépôt en question. On peut le voir dans les paramètres du dépôt, sous l'onglet *Déclencheurs Web*.

Figure 2: L'onglet des *webhooks* dans Gitea

À chaque fois qu'un événement va se produire sur le dépôt, Gitea va prévenir Drone qui décidera si l'évènement doit conduire à lancer l'intégration continue ou non, selon les instructions qu'on lui a donné dans la configuration du dépôt.

2.4.2 Définir les étapes d'intégration

Nous allons devoir rédiger un fichier `drone.yml`, que l'on placera à la racine du dépôt. C'est ce fichier qui sera traité par DroneCI pour savoir comment compiler et tester le projet.

Un fichier `drone.yml` existe déjà à la racine du dépôt. Celui-ci pourra vous servir d'inspiration, mais il ne fonctionnera pas directement dans votre installation.
Vous rencontrerez des problèmes inattendus si vous utilisez le fichier `.drone.yml` du dépôt.
Vous **DEVEZ** partir d'un fichier vide et suivre la documentation pour obtenir un `.drone.yml` fonctionnel.

Toutes les informations nécessaires à l'écriture du fichier `.drone.yml` se trouvent dans l'excellente documentation du projet :
https://docs.drone.io/pipeline/docker/examples/languages/golang/.

Les étapes sont sensiblement les mêmes que dans le `Dockerfile` que nous avons écrit précédemment.

Commencez à partir de l'exemple donné dans la documentation de Drone. Par rapport au `Dockerfile`, n'ajoutez pas `tags dev`, cela permettra d'embarquer tout le contenu statique (pages

HTML, feuilles de style CSS, Javascript, ...) directement dans le binaire, ce qui simplifiera la distribution.

Committons puis poussons notre travail. Dès qu'il sera reçu par Gitea, nous devrions voir l'interface de Drone lancer les étapes décrites dans le fichier.

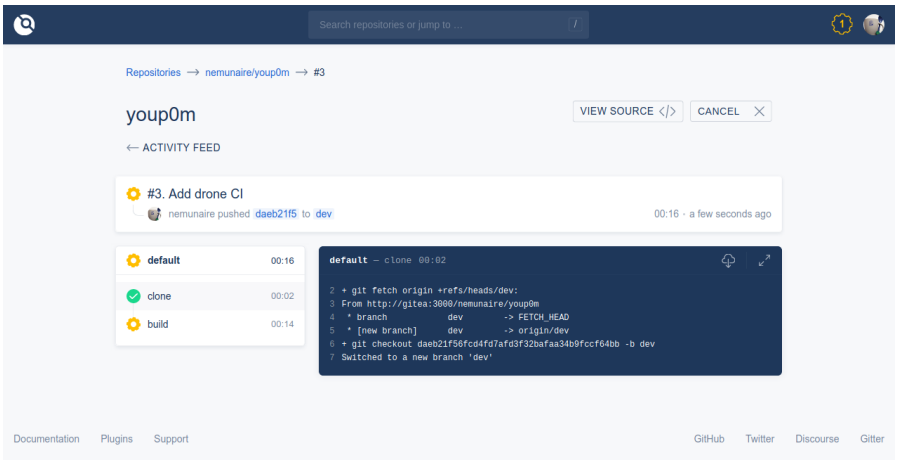

Figure 3: Drone en action

IMPORTANT : si vous avez l'impression que ça ne marche pas et que vous avez réutilisé le fichier présent sur le dépôt au lieu de partir de l'exemple donné dans la documentation, **commencez en partant de l'exemple de la documentation** ! Le fichier présent sur le dépôt **ne fonctionnera pas** dans votre situation !

Lorsqu'apparaît enfin la ligne `git.nemunai.re/youp0m`, le projet est compilé !

2.4.3 Inspection qualité

`youp0m` n'a pas de suite de tests fonctionnels, mais nous allons utiliser Sonarqube pour faire une revue qualité du code !

Tout d'abord, il faut lancer le conteneur Sonarqube :

```
docker run --rm -d --name sonarqube --network drone -p 9000:9000 sonarqube
```

Le service met un bon moment avant de démarrer, dès qu'il se sera initialisé, nous pourrons accéder à l'interface sur http://localhost:9000.

En attendant qu'il démarre, nous pouvons commencer à ajouter le nécessaire à notre `.drone.yml` : http://plugins.drone.io/aosapps/drone-sonar-plugin/.

Après s'être connecté à Sonarqube (`admin:admin`), nous pouvons aller générer un token, tel que décrit dans la documentation du plugin Drone.

Une fois la modification *commitée* et poussée, Drone enverra le code à Sonarqube qui en fera une analyse minutieuse. Rendez-vous sur http://127.0.0.1:9000/projects pour admirer le résultat.

2.4.4 Publier le binaire correspondant aux tags/jalons

Nous savons maintenant que notre projet compile bien dans un environnement différent de celui du développeur ! Néanmoins, le binaire produit est perdu dès lors que la compilation est terminée, car nous n'en faisons rien.

Ajoutons donc une nouvelle règle à notre `.droneci.yml` pour placer le binaire au sein de la liste des fichiers téléchargeables aux côtés des tags.

Vous aurez sans doute besoin de :

- https://docs.drone.io/pipeline/conditions/
- http://plugins.drone.io/drone-plugins/drone-gitea-release/

Attention à ne pas stocker votre clef d'API dans le fichier YAML !

Lorsque l'on est plusieurs à travailler sur le projet ou pour accroître la sécurité, il convient de créer, un compte *bot* qui sera responsable de la création des *releases*. Ce sera donc sa clef d'API que l'on indiquera dans l'interface de Drone.
Pour notre exercice, nous n'avons pas besoin de créer un utilisateur dédié, mais il est impératif d'utiliser les *secrets* de Drone pour ne pas que la clef d'API apparaisse dans l'historique du dépôt.

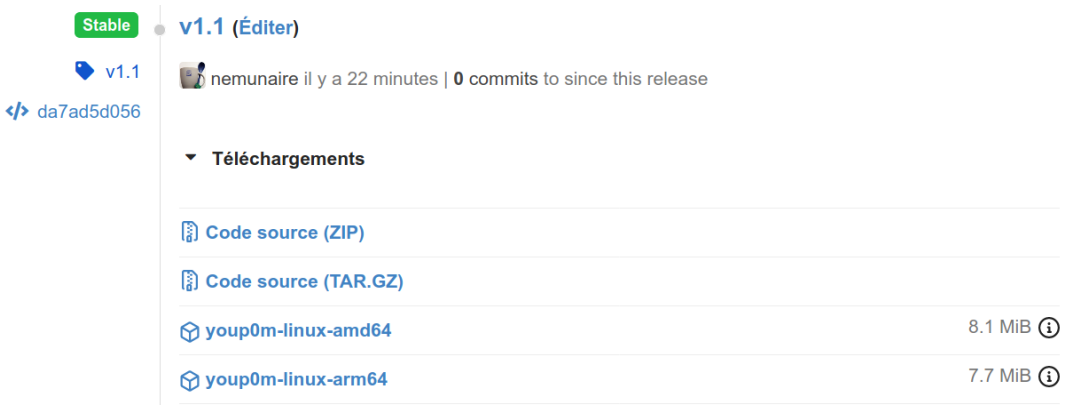

Figure 4: Binaire publié automatiquement sur Gitea

2.4.5 Publier une image Docker

Toutes les tâches de publication peuvent s'assimiler à des tâches de déploiement continu. C'est en particulier le cas lorsque le produit de compilation sera simplement publié et qu'il n'y a pas de service à mettre à jour ensuite (par exemple, dans le cas de Firefox ou de LibreOffice, une fois testés, les paquets sont envoyés sur le serveur d'où ils seront distribués ; il n'y a pas de service/docker à relancer).

À l'inverse, youp0m est à la fois un programme que l'on peut télécharger et un service qu'il faut déployer pour le mettre à jour. Afin de simplifier son déploiement en production, nous utiliserons une image Docker.

2.4.5.1 Fonctionnement du registre de Gitea

Gitea intègre un registre d'images Docker (depuis la version 1.17, mi-2022). Pour le moment, les images sont uniquement liées à un compte utilisateur ou à une organisation, pas directement à un dépôt. Une page permet de rattacher l'image envoyée à un dépôt, ce que l'on fera dans un deuxième temps.

Afin de pouvoir envoyer une image nous-même, nous devons nous connecter au registre :

```
docker login localhost:3000
```

N'a-t-on pas besoin d'un certificat TLS pour utiliser un registre Docker ?

Ceci est possible exclusivement parce que le registre `localhost` est considéré non-sûr par défaut. C'est-à-dire qu'il n'a pas besoin de certificat TLS valide sur sa connexion HTTP pour être utilisé.

Si on avait dû utiliser un autre nom de domaine, il aurait fallu l'ajouter à la liste des `insecure-registries`.

Nous pouvons ensuite envoyer une image pour s'assurer que tout va bien :

```
docker build -t localhost:3000/${USER}/youp0m .
docker push localhost:3000/${USER}/youp0m
```

Rendez-vous ensuite sur la page http://gitea:3000/$%7BUSER%7D/-/packages pour attribuer l'image au dépôt youp0m (voir pour cela dans les paramètres de l'image).

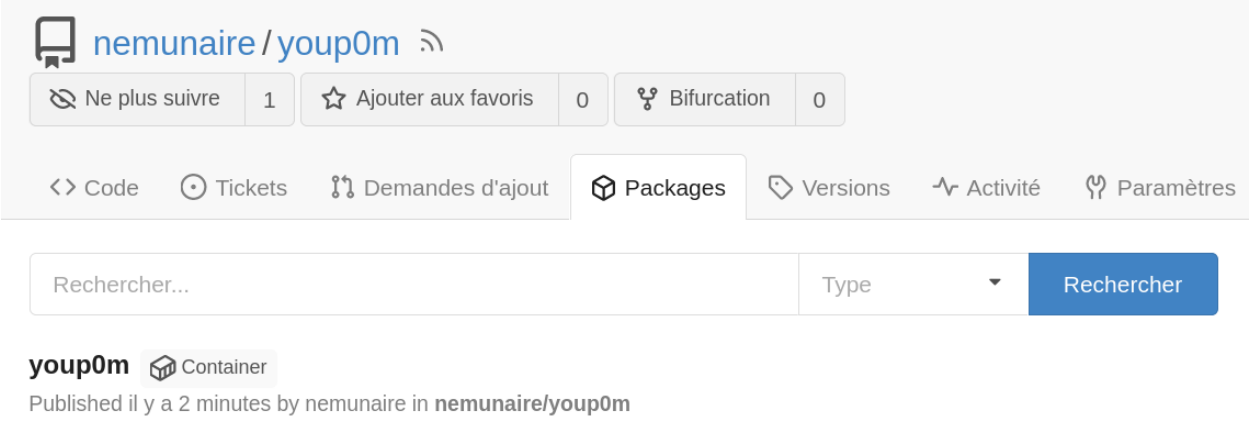

Figure 5: Notre image youp0m dans Gitea !

2.4.5.2 Publication de l'image

Une fois le registre testé, il ne nous reste plus qu'à ajouter une étape de publication de l'image Docker. Cela se fait au moyen du plugin suivant :
http://plugins.drone.io/drone-plugins/drone-docker/.

Continuons d'éditer le fichier `.drone.yml` du dépôt pour faire générer à Drone l'image Docker, et la publier.

Attention dans Drone, le domaine à utiliser pour contacter Gitea (et donc le registre), n'est pas `localhost`, mais `gitea`. Comme notre registre n'a pas de certificat TLS pour utiliser `https`, il est nécessaire de définir l'option `insecure` à `true`. Utilisez les *secrets* de Drone pour stocker le nom d'utilisateur et le mot de passe d'accès au registre.

2.4.5.3 Test de l'image

Sur notre hôte, nous pouvons tester que l'image a bien été publiée grâce à la commande suivante :

```
docker pull localhost:3000/${USER}/youp0m
docker run --rm -p 8080:8080 localhost:3000/${USER}/youp0m
```

Si une nouvelle image est bien récupérée du dépôt, bravo, vous avez réussi !

2.4.6 Vers le déploiement

Nous n'allons pas faire le déploiement aujourd'hui, car nous n'avons pas d'environnement de production sur lequel déployer notre service.

Vous pouvez néanmoins tester les plugins `scp` ou `ansible` si vous avez une machine virtuelle avec une connexion SSH. N'hésitez pas à l'ajouter à votre `.droneci.yml`.

2.4.7 Profitons !

Sonarqube a repéré quelques erreurs dans le code de `youp0m`, essayez de les corriger, et publiez une nouvelle version, pour observer toute la chaîne en action !

2.5 Autres outils indispensables

2.5.1 Maintient à jour des dépendances

Une opération fastidieuse, souvent oubliée sitôt le projet envoyé en production, c'est la mise à jour des dépendances applicatives. Fastidieux car il faut d'une part être informé qu'une mise à jour est disponible, c'est-à-dire qu'il faut suivre les mails, parfois nombreux, informant des nouvelles *releases*, parfois il s'agir de newslettre, ou encore parfois aucune notification ne peut être programmée, il faut se rendre régulièrement sur un site pour savoir si oui ou non une mise à jour est disponible.

Et ce n'est pas tout puisqu'une fois la nouvelle version identifiée, il faut la récupérer, vérifier que tout compile et que les tests passent… On peut vite comprendre que personne ne veut avoir cette tâche.

Heureusement pour nous, des outils existent pour faire tout cela ! Dependabot et Renovatebot sont deux projets qui vont nous aider à maintenir nos projets. Que ce soit pour corriger une vulnérabilité dans un module NodeJS, un nouvelle version d'un conteneur Docker ou une évolution majeure d'un framework, nous serons alerté et pourrons agir en conséquence, en sachant si les tests passent ou pas, avec l'aide de notre système d'intégration continue.

Pour la suite de notre expérience, nous allons prendre en main Renovatebot qui semble aujourd'hui la solution la plus aboutie des deux.

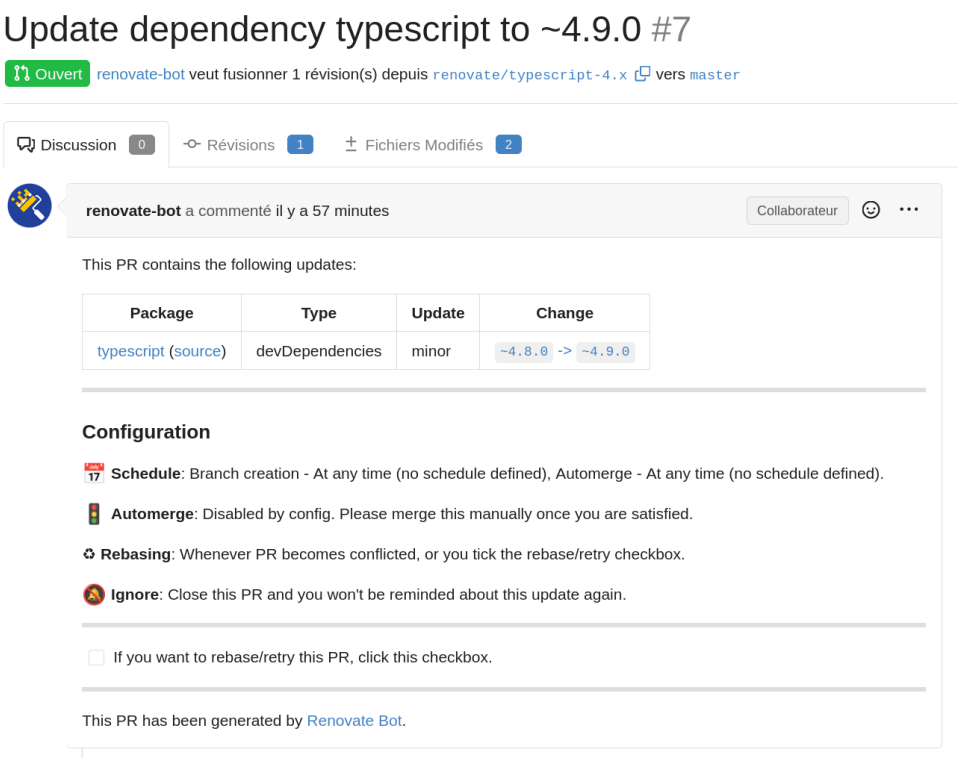

Figure 6: L'activité favorite de renovatebot : créer des *pull-requests*

2.5.2 Installation de Renovatebot

Une fois de plus, nous allons déployer … un conteneur Docker ! Mais avant cela, il va falloir créer un utilisateur dédié à Renovatebot dans notre forge. Cet utilisateur sera utilisé par le *bot* pour participer à vos projets : il va régulièrement cloner vos dépôts, rechercher les dépendances pour les outils qu'il maîtrise, puis préparer des *pull-requests* dès qu'il détectera une nouvelle version d'une dépendance.

Dès lors que vous aurez créé le nouvel utilisateur dans Gitea, il faudra générer un jeton d'accès, car aussi doué soit Renovatebot, il préfère utiliser l'API HTTP plutôt que de cliquer sur les boutons. Dans les paramètres de l'utilisateur que vous aurez créé, sous l'onglet « Applications », vous trouverez un encadré « Gérer les jetons d'accès ». Contrairement à DroneCI pour lequel il fallait créer une application OAuth2, ici il s'agit bien du formulaire en haut de la page.

> **!** Il est bien question de créer un jeton d'accès pour l'utilisateur renovatebot que vous avez créé, et non pas pour votre compte utilisateur.
> Bien que cela fonctionnerait parfaitement, le bot parlerait en votre nom, il serait alors difficile de suivre les requêtes.

Juste avant de lancer le conteneur, assurez-vous que votre utilisateur Renovatebot a bien accès en écriture aux dépôts que vous souhaitez qu'il surveille. Sans quoi il ne sera pas en mesure de vous aider.

Voici à quoi pourrait ressembler le playbook Ansible démarrant notre conteneur Renovatebot :

```
- name: Launch renovate container
  docker_container:
    name: renovate
    image: renovate/renovate
```

```
    state: started
    restart_policy: no
    memory: 2G
    networks:
      - name: gitea_net
    env:
      RENOVATE_ENDPOINT: "http://gitea:3000/api/v1/"
      RENOVATE_PLATFORM: "gitea"
      RENOVATE_TOKEN: "{{ renovate_token }}"
      RENOVATE_GIT_AUTHOR: "Renovatebot <renovate@gitea.sample>"
      RENOVATE_AUTODISCOVER: "true"
      RENOVATE_LOG_LEVEL: "info"
```

Dès que le conteneur sera lancé, nous devrions voir apparaître une ou plusieurs *pull-requests* pour le projet `youp0m`. Si votre CI est configurée correctement, des tests automatiques seront lancés.

Le conteneur s'arrête dès qu'il a terminé d'analysé tous les dépôts. Vous devrez le relancer si vous attendez une nouvelle action de la part de Renovatebot. Il est courant de le lancer entre chaque heure et 2 ou 4 fois par jour.

De très nombreuses options permettent d'adapter le comportement de Renovatebot aux besoins de chaque projet. La configuration se fait au travers d'un fichier `renovate.json` qui se trouve généralement à la racine du dépôt.

Pour avoir un aperçu de toutes les possibilités offertes par renovatebot, consultez la liste des éléments de configuration : https://docs.renovatebot.com/configuration-options/

Ne soyez pas effrayé par la liste interminable d'options. Il est vrai que la première fois, on peut se sentir submergé de possibilités, mais il faut noter que le projet arriver avec des options par défaut plutôt correctes, et que l'on peut facilement avoir une configuration commune pour tous nos dépôts, à travers les *presets*.

Un certain nombre de *presets* sont distribués par défaut, voici la liste (humainement lisible cette fois) : https://docs.renovatebot.com/presets-default/

Voici un exemple de configuration que vous pouvez utilisé comme base de tous vos projets :

```
{
  "$schema": "https://docs.renovatebot.com/renovate-schema.json",
  "extends": [
    "local>renovate/renovate-config"
  ]
}
```

Ceci ira chercher une configuration dans le fichier `default.json` du dépôt `renovate/renovate-config`. À condition que votre utilisateur Renovatebot s'appelle effectivement `renovate`.

Voici un exemple de fichier `default.json` que vous pourriez vouloir utiliser :

```
{
  "$schema": "https://docs.renovatebot.com/renovate-schema.json",
  "packageRules": [
    {
      "description": "Opt-out minimum Go version updates",
      "matchManagers": ["gomod"],
      "matchDepTypes": ["golang"],
```

```
      "enabled": false
    }
  ],
  "extends": [
    "config:base",
    ":dependencyDashboard",
    ":enableVulnerabilityAlertsWithLabel('security')",
    "group:recommended"
  ]
}
```

Attention, on ne le répétera jamais assez, mais Renovatebot peut vite devenir infernal, car il va créer de nombreuses *pull-requests*, inlassablement. Il convient de rapidement activer la fusion automatique des mises à jour pour lesquelles vous avez confiances et pour lesquelles vous ne feriez qu'appuyer sur le bouton de fusion, sans même tester vous-même. La fonctionnalité est décrite en détail dans la documentation[3] et explique les différentes stratégies. Néanmoins, il est nécessaire d'avoir une bonne suite de tests avant d'envisager d'utiliser une telle fonctionnalité.

[3] https://docs.renovatebot.com/key-concepts/automerge/

3 Rendu

Est attendu d'ici le cours suivant :

— vos réponses à l'évaluation du cours,
— tous les exercices de ce TP.

3.1 Arborescence attendue

Tous les fichiers identifiés comme étant à rendre sont à placer dans un dépôt Git privé, que vous partagerez avec votre professeur.

Voici une arborescence type (vous pourriez avoir des fichiers supplémentaires) :

```
./cicd-playbook/
./cicd-playbook/cicd-setup.yml
./cicd-playbook/roles/...
./youp0m/
./youp0m/.drone.yml
./youp0m/.ansible/...        # Pour ceux qui auraient fait le 5.4 optionnel
./youp0m/Dockerfile
./youp0m/entrypoint.sh
./youp0m/.dockerignore
./youp0m/renovate.json
./youp0m/...                 # Seuls les fichiers modifiés du dépôt original sont
   attendus
```

Votre rendu sera pris en compte en faisant un tag **signé par votre clef PGP**. Consultez les détails du rendu (nom du tag, ...) sur la page dédiée au projet sur la plateforme de rendu.

?

Si vous utilisez un seul dépôt pour tous vos rendus, vous **DEVRIEZ** créer une branche distincte pour chaque rendu :

```
42sh$ git checkout --orphan renduX
42sh$ git reset
42sh$ rm -r *
42sh$   # Créer l'arborescence de rendu ici
```

Pour retrouver ensuite vos rendus des travaux précédents :

```
42sh$ git checkout renduY
-- ou --
42sh$ git checkout master
...
```

Chaque branche est complètement indépendante l'une de l'autre. Vous pouvez avoir les exercices du TP1 sur `master`, les exercices du TP5 sur `rendu5`, ... ce qui vous permet d'avoir une arborescence correspondant à ce qui est demandé, sans pour autant perdre votre travail (ou le rendre plus difficile d'accès).