Kubernetes & GitLab

Nicolas Hordé
Université de Limoges
Service Outils Numériques
123, avenue Albert Thomas
87060 Limoges cedex

Résumé

Kubernetes et Gitlab, deux mots que l'on peut lire de façon récurrente si l'on suit l'actualité informatique. Ce duo, c'est l'aboutissement du paradigme de conteneurisation : le déploiement automatique d'applicatifs déclenché par la soumission d'une version stable de leur(s) code(s) source(s) vers la branche « Main » de votre dépôt Git !

Kubernetes est un orchestrateur de conteneur open source qui se place schématiquement au-dessus du système de conteneurisation. Son objectif est d'assurer l'exploitation de vos conteneurs. Il permet de décrire par le biais du format YAML les micro-services que vous désirez installer sur votre cluster. Kubernetes, dit « K8S » est hautement modulaire. Il s'appuie sur des briques libres pour opérer le stockage, l'interconnexion et la conteneurisation. Il est nécessaire de se familiariser avec son jargon et ses concepts pour en tirer toute la substance. K8S maintient l'état des services grâce à un système inédit de « failover ».

Gitlab dans sa version « Community Edition » est une plateforme web de développement. En conjonction avec K8S, il devient aisé de mettre en place la construction de conteneurs, des tests de qualité ou des audits de sécurité du code voire même le déploiement automatique d'infrastructures complexes. Le DevOps, ou opérateur de développement et d'opérabilité, va écrire des scripts visant à exécuter des tâches lorsque vous poussez du code vers la plateforme. Selon la branche, le type de projet, ou la nature du contenu, les scripts vont opérer différemment par le biais de pipelines qui vont enchaîner conditionnellement des tâches.

Mots-clefs

CI/CD, conteneurs, cluster, Docker, GitLab, Kubernetes, DevOps intégration, workflow, Git, Bash

1 Les conteneurs

1.1 Un peu d'histoire

La conteneurisation est un « nouveau » paradigme qui offre les nombreux avantages de la virtualisation tout en limitant la consommation de ressources associée à son exploitation. La conteneurisation est le fruit d'une l'évolution de plusieurs outils dont les fonctionnalités ont convergé : Chroot (1979), Jail (2000), Vserver (2001), Linux namespaces (2002), Process Containers (2006), LXC (2008), Warden (2011), Docker (2013). L'apparition de Docker a suscité un engouement croissant autour de la conteneurisation tant sa simplicité de mise en œuvre est déconcertante.

1.2 Comment ça fonctionne ?

Pour faire simple, la conteneurisation repose sur un ensemble d'API intégrées au noyau Linux qui permet l'exécution de processus dans un environnement partiellement isolé du système hôte.

Une arborescence partielle du système Linux est utilisée afin d'instancier un conteneur. Cette arborescence, stockée sous la forme d'un fichier image au format « OCI Image Format », est constituée préalablement par l'utilisateur à l'aide d'un outil dédié. Un fichier descriptif (Dockerfile) renseigne les différentes actions nécessaires à la génération de l'image. Cette recette de fabrication de l'image repose essentiellement sur l'installation de paquets, la copie/création de fichiers, et ce, en partant d'un modèle d'image préalablement décrit. Un point d'entrée peut être défini afin de fixer quel applicatif sera exécuté après création du conteneur. Le stockage des images utilise un système de fichier incrémentiel (tel que OverlayFS) qui est relativement peu gourmand en termes d'espace disque et qui optimise la génération successive d'images à partir d'un même modèle. Un versionnage par étiquette est associé à chaque image afin de cibler facilement une version particulière. Il existe un système performant de dépôt en ligne qui assure le téléchargement automatique d'images si celle-ci n'est pas présente sur l'ordinateur hôte.

Lorsque l'utilisateur demande l'exécution d'un conteneur s'appuyant sur un fichier image (commande « docker run » par exemple), l'utilisateur peut spécifier comment le conteneur sera relié aux ressources de l'hôte (système de fichier, ports, variables d'environnement, réseaux). Par défaut, le conteneur est exécuté dans un environnement totalement isolé du système hôte.

1.3 Structuration du moteur de conteneurisation

Sous Linux, la conteneurisation est assurée par un ensemble d'exécutables et de services dont voici l'imbrication générale.

1. Interface CLI ou GUI, (ex : commande « Docker ») ;
2. Runtime de haut niveau, (ex : Dockerd) ;
3. Runtime de bas niveau, (ex: Containerd) ;
4. Couche OCI. (Open Container Initiative, ex : Runc).

Il existe de nombreux moteurs (ou runtime) de conteneurisation avec des caractéristiques différentes : docker, cri-o, kata, lxc, lxd, Singularity, Containerd, Firecracker, gVisor, Rkt.

1.4 Quels bénéfices ?

Pour résumer les bénéfices induits par la conteneurisation :

- isolation du conteneur par rapport au système hôte (et entre conteneurs) ;
- découplage avec le système de paquet hôte ;
- portabilité des conteneurs assurée par une API standardisée ;
- performances quasi identiques à un fonctionnement natif ;
- stockage différentiel d'images comportant qu'une portion de l'arborescence de Linux ;

- immuabilité des images ;
- grande lisibilité du contenu d'une image et de la source permettant de la produire ;
- scalabilité horizontale (nécessite un orchestrateur).

Cependant, pour beaucoup de projets, l'usage de la conteneurisation seule ne permet pas d'acquérir un niveau d'abstraction suffisant pour remédier à la complexité et offrir une alternative élégante.

2 Kubernetes: l'orchestrateur

2.1 Présentation de Kubernetes

Kubernetes est une plate-forme modulaire qui permet d'opérer des micro-services conteneurisés. Elle favorise à la fois l'écriture de configuration déclarative et l'automatisation. C'est suite à l'ouverture de codes issus de Borg par Google en 2014 que Kubernetes est né**.** Kubernetes est traductible par « Timonier », on utilise parfois le nom contracté « K8S » ou « Kube » pour les intimes. Kubernetes occupe la couche n°4 du mille-feuille qui compose un cloud, juste au-dessus du moteur de conteneurisation.

1. Layer 1 : infrastructure ;
2. Layer 2 : OS ;
3. Layer 3 : Container Engine ;
4. Layer 4 : Orchestration ;
5. Layer 5 : CMP (Cloud Management Platform).

2.2 Principes de base

2.2.1 Le Cluster

Kubernetes peut assurer une redondance des éléments applicatifs par le biais de plusieurs nœuds qui constituent un « cluster » (grappe d'ordinateur relié par un réseau). Deux types de nœuds cohabitent dans le cluster : les nœuds maîtres (masters) et les nœuds esclaves (workers). Ce qui différencie les deux est la présence du plan de contrôle (Control plane) au sein des nœuds maîtres qui assurent le bon fonctionnement de l'ensemble du cluster.

« Kubelet » est le service qui écoute et exécute les directives données par le plan contrôle sur chaque nœud. Lors de l'instanciation d'un conteneur, c'est lui qui relaye les directives reçues vers le CRI (Container Runtime Interface) qui formalisera ensuite la demande sous une forme compréhensible par le moteur de conteneurisation.

Tous les nœuds sont reliés et communiquent par le biais d'un réseau interne dont le choix est laissé à l'initiative de l'administrateur lors de l'installation de K8S (CNI – Container Network Interface). Plusieurs plugins CNI sont disponibles, ils s'appuient sur des techniques différentes pour assurer la communication dite « interpod ». Calico, Flannel, Cilium, Contiv offrent des caractéristiques variées en ce qui concerne la consommation de ressources, les performances, les fonctionnalités ou encore la sécurité.

À tout moment, « cAdvisor » renseigne le plan de contrôle de l'état du nœud et assure ainsi la remontée de nombreuses métriques vers le cœur du système. Dans le plan de contrôle, c'est le « Scheduler » qui jugera des directives appropriées à donner pour assurer le fonctionnement optimal du cluster à la lueur des métriques dont il dispose. L'application de ces directives est réalisée par le « Controller manager » qui relaiera ensuite celles-ci vers les services « Kubelet » des nœuds concernés.

Il existe un point d'entrée unique des API, situé au sein de plan de contrôle : l'« API server ». Que vous utilisiez la commande CLI « kubectl » ou l'interface graphique par le biais du « Proxy », toutes vos actions passent par cette API REST.

Le stockage dans K8S est interfacé grâce au CSI (Container Storage Interface). Il est possible pour l'administrateur ou l'opérateur de stockage de déclarer différents « backend » de stockage associés à des technologies différentes. Sur plusieurs nœuds, il est impératif d'utiliser un stockage distribué tel que Ceph ou GlusterFS ou encore S3 (solution propriétaire).

2.2.2 Les objets

Dans K8S, tout n'est qu'objet ! Ceux-ci sont stockés dans une base clé-valeur nommée « ETCD ». Cette base de données est stockée de façon distribuée au sein du plan de contrôle sur l'ensemble des nœuds maîtres. Ils assurent, si leur nombre est de 3 au minimum, une haute disponibilité. Les objets dans K8S sont de deux types.

1. Volatiles et localisables au sein d'un nœud en particulier ;
2. Immuables et de portée globale.

La plus petite unité indivisible (par K8S) au sein d'un cluster est le « Pod ». Un pod est joignable par une adresse IP unique qui est logée sur un réseau dédié aux pods. Un pod peut contenir un ou plusieurs conteneurs. Il est possible de définir des conteneurs dits « d'initialisation » qui sont instanciés avant la création des conteneurs principaux afin de réaliser des tâches qui préparent le lancement du pod. Le pod est par nature le **seul** objet qui est volatile et localisable au sein du cluster K8S : il possède un cycle de vie complet. Il est possible de déterminer les conditions de vie et de mort du pod en leur adjoignant des tests à intervalle régulier (liveness probes). Si celui-ci y échoue, il sera immédiatement tué !

On ne crée pas directement un pod dans K8S. Il faut renseigner un « deployement » qui va déterminer la configuration du pod une fois qu'il sera déployé (nombre d'instances, stockage, etc). Il existe plusieurs sous-types de « deployement » dont la finalité diffère, mais dont l'objectif principal demeure le même : instancier des pods sur des nœuds. « daemon set », « horizontal pod autoscaler », « stateful set » sont fréquemment utilisés. Dès qu'un « deployement » est créé dans Kube, il est en mesure de générer des instances de pods.

Le deuxième objet le plus fréquemment utilisé est le « service ». Celui-ci est relié via des annotations à un « deployement » afin d'aiguiller les requêtes qui le vise vers les pods qui y sont liés. Un service possède une IP sur un réseau dédié aux services.

Le stockage est réalisé par un système d'allocation dynamique ou statique de ressources. Le « Persistant Volume Claim » et le « Persistant Volume » sont les incarnations sous forme d'objet de ces deux stratégies.

Chaque nœud est connu dans Kube sous forme d'un objet « node ».

Pour terminer, je présenterais rapidement l'objet « ingress » ou « ingressroute » (selon le système utilisé) qui gère l'accès externe aux services dans le cluster, généralement du trafic HTTP(S). Il permet schématiquement de lier un nom à un ou plusieurs services par le biais interposé ou non d'un équilibrage de charge (loadbalancer).

2.2.3 Le fonctionnement de Kubernetes

Le « Controller manager » possède un rôle primordial. Son objectif est de maintenir un état stable dans le cluster. Celui-ci est la somme de tous les fichiers descriptifs de tous les objets que les opérateurs ont renseignée préalablement. À chaque instant, le « Controller manager » vérifie que l'état des objets est consistant. Le point de rupture sur lequel tout repose, c'est le pod ! Si les tests ne sont pas correctement remplis ou si le pod ne répond pas correctement aux sollicitations : il sera tué et ré-instancié selon les règles définies par le « deployement » qui le détermine.

Il existe fondamentalement deux façons d'agir sur l'état du cluster K8S. Il est possible de lancer des commandes dites impératives qui vont forcer le cluster à entrer directement un nouvel état. Et il existe la méthode déclarative s'appuyant sur un fichier descriptif au format YAML qui va indiquer à K8S, les objets et les caractéristiques associés souhaités. La méthode déclarative est celle qui sera préférée, l'autre étant plutôt réservée à des fins d'administration et le débogage.

En l'état, on comprend vite que l'administration de Kubernetes risque fort de devenir rapidement très complexe et peu intuitive. Il devient alors nécessaire de faire appel à d'autres composants afin de simplifier la création de micro-services.

2.3 *Vers Kube et au-delà !*

2.3.1 Helm

Helm n'est autre qu'un applicatif qui assure l'installation, la désinstallation et l'administration d'applicatifs sur le cluster tel qu'on le ferait avec un gestionnaire de paquet sous Linux. Un système de modèle et une commande CLI permettent à l'utilisateur de générer à la volée des fichiers YAML à partir d'un « chart » (paquet de Helm) et d'un fichier de configuration « values.yml » qui permettra de personnaliser l'installation. Les fichiers YAML résultants seront automatiquement appliqués au cluster. Il est possible d'écrire des charts qui s'appuient sur d'autre charts grâce à un système de dépendances très complet.

2.3.2 Grafana/Prometheus

Prometheus se charge de collecter les métriques du cluster K8S vers une base de données. Grafana offre un nombre important tableaux de bord préconfigurés, mais hautement paramétrables afin d'exposer sous forme de graphiques les données préalablement récupérées. Par le biais de Grafana, il est ainsi possible de visualiser les données récoltées par Prometheus et s'assurer le bon fonctionnement du cluster et ses services.

2.3.3 Kibana/Fluent /ES

Fluent, par le biais d'un « logging agent », vient récupérer les journaux produits par les pods afin de les stocker dans une base de données Elasticsearch. Celle-ci est ensuite exploitée par le logiciel Kibana afin de mettre en exergue certaines problématiques difficilement décelables par le seul usage des outils de métrologie.

2.3.4 Traefik/ISTIO

Par défaut, Ngnix est intégré à Kubernetes en tant que « ingress controller », c'est-à-dire pour réaliser l'aiguillage des accès externes (nom DNS) vers les services. Cependant d'autres alternatives plus performantes et plus flexibles sont offertes (service meshing) telles que Istio et Traefik.

2.3.5 SaltStack/Ansible/Chef/Puppet/pssh

Étant donné que les clusters Kubernetes s'étendent couramment sur plusieurs nœuds : je vous invite à utiliser une solution permettant d'exécuter des commandes par lot sur tout ou partie du cluster.

2.3.6 Vault

Vault permet de réaliser le stockage de données sensibles (credentials/secrets) tels que les mots de passe et identifiants utilisés par les pods. Il évite que ceux-ci soient renseignés en clair dans les fichiers YAML.

2.3.7 GlusterFS/Heketi/NFS

En stockage principal, nous avons choisi d'utiliser un GlusterFS, qui a été installé sur tous les nœuds afin de fournir des volumes persistants à la demande. Ce système de fichier distribué permet de réaliser des instantanés et il présente d'excellentes performances. Le provisionnement automatique est réalisé par le biais de Heketi qui alloue de façon transparente les ressources. Nous utilisons aussi NFS par le biais d'un serveur externe opéré par TrueNas au dessus un système de fichier ZFS. Le stockage est un point d'attention important si vous désirez profiter de la haute disponibilité (HA).

3 GitLab

Sur notre plateforme d'intégration et de déploiement continu de type cloud, il s'agit de la brique se situant sur la couche la plus élevée (Layer 5).

3.1 Présentation de GitLab

GitLab est la plateforme CI/CD par excellence, elle fournit un ensemble de fonctionnalités très large autour de l'édition de code et du travail collaboratif. GitLab s'appuie sur plusieurs logiciels libres. GitLab peut fonctionner sur Kubernetes ou Docker ou même sans aucune conteneurisation.

3.2 Structure de GitLab

GitLab est une application littéralement tentaculaire qui s'articule autour de nombreuses briques logicielles parfois séparées dans des conteneurs différents :

- Unicorn (ou Puma depuis v13.0) - serveur de l'interface web et API de GitLab ;
- PostgreSQL – base de données qui stocke les informations de l'interface web de GitLab ;

  GitLab Shell – point d'entrée des connexions SSH ;
- Redis – Cache de l'interface web ;

- Sidekiq – framework d'exécution en tâche de fond pour Ruby qui gère des travaux de façon asynchrone ;
- GitLab Workhorse – C'est un proxy inverse qui prend en charge les requêtes HTTP comme le téléchargement/dépôt et les poussages/tirage de code depuis les dépôts Git ;
- Gitaly – Service RPC qui gère les appels Git réalisés par GitLab ;
- GitLab Pages - Générateur et serveur de pages web statiques ;
- NGINX (remplacé par Traefik) – Routeur de trafic externe vers les services de Kubernetes ;
- Minio – Serveur de stockage orienté objet qui contient les données inhérentes aux « Runners » ;
- Registry – Serveur d'images de conteneurs au format OCI ;
- Runners – Exécuteur de tâches qui permettent de réaliser l'intégration et le déploiement continu.

3.2.1 Git, la gestion de versions

Impossible de parler de CI/CD sans aborder Git. Git est au centre de GitLab, comme son nom l'indique. Git est un logiciel de gestion de versions décentralisé libre créé par Linus Torvalds. Git permet de soumettre une version d'un code donné (de façon plus générale de texte brut même si Git sait gérer des binaires) vers un dépôt qui contient l'historique complet de tous les changements.

Les branches

Dans Git, il est possible de travailler sur plusieurs branches dont la signification diffère selon la convention que l'on désire mettre en place. Il existe plusieurs modèles de développement avec une normalisation des numérotations de versions et des branches qui sont spécifiques aux choix de chaque groupe de développeurs travaillant sur le même dépôt, mais dont les considérations ne sont pas techniques, mais plutôt organisationnelles.

Les références

Il existe plusieurs façons de faire référence à une révision donnée du dépôt. Branche, étiquette, référence relative ou Hash permettent au développeur de cibler une révision particulière du dépôt afin de lui appliquer ultérieurement une action.

Les actions

Il est possible d'agir de diverses façons depuis/vers un dépôt :

- Créer un nouveau dépôt (git init) ;
- Récupérer un dépôt distant (git clone) ;
- Voir les différences entre 2 révisions (git diff) ;
- Voir le statut de la copie de travail actuelle (git status) ;
- Visualiser les journaux du dépôt (git log) ;
- Ajouter/détruire/déplacer des fichiers (git add/rm/mv) ;
- Mets à jour le dépôt local (git pull/fetch) ;
- Ajouter une nouvelle révision à mon dépôt local (git comit) ;

- Envoyer une branche vers le dépôt distant (git push) ;
- Annuler des modifications (git reset) ;
- Ajouter/lister des branches (git branch) ;
- Changer de branche (git checkout) ;
- Fusion de branche (git merge).

Grâce au concours de Git, il est facile de travailler à plusieurs développeurs sur un même dépôt distant en toute sécurité en résolvant les conflits de version. Il est aisé de revenir à des révisions précédentes ou de fusionner 2 versions différentes.

3.3 *Intégration et déploiement continus – La théorie*

L'intégration et déploiement/livraison continus ou CI/CD, c'est la convergence de plusieurs technologies et de méthodologies de développement, dont certaines sont liées à l'organisation du travail lui-même.

3.3.1 Intégration continue

En pratique l'intégration continue vise à garantir la production d'un code de qualité à chaque révision que les développeurs soumettent vers un système de gestion de version centralisé. Des tests récurrents sont automatiquement déclenchés selon des critères définis à l'avance par le DEVOPS (l'opérateur en charge de CI/CD). Les tests peuvent concerner des domaines variés tels que des audits de sécurité, formatage du code, optimisation du code, analyse de vulnérabilités, mesure de l'efficience du code, tests unitaires, etc.

3.3.2 Déploiement continu

Le CD consiste à produire automatiquement un applicatif à partir des sources, dès lors que ceux-ci ont été soumis au système de gestion de version centralisé. Dans les plateformes CD les plus abouties, la mise en production de l'applicatif est aussi réalisé, dès que la construction de l'applicatif se termine de façon satisfaisante. L'étape CD est presque systématiquement précédée des étapes CI, et leur exécution est conditionnée au résultat positif de leur déroulement.

3.3.3 CI/CD et pipelines

Pour assurer le cycle CI/CD dans son intégralité, un système est utilisé par la quasi-totalité des plateformes : les pipelines. Les pipelines sont des graphes orientés comportant plusieurs étapes (stage). Une étape peut être constituée d'une ou de plusieurs tâches. Par défaut, une étape est considérée comme validée si toutes les tâches (task) qui lui sont associées sont exécutées sans erreur. D'autres modes de fonctionnement sont cependant possibles avec l'introduction de sous-pipelines ou d'un système d'exécution de tâches en parallèle. Il est possible par ailleurs de définir des tâches dont l'exécution est manuelle. L'état de chaque tâche est matérialisé à sa gauche par une icône (validée, non validée ou manuelle).

Dans GitLab, chaque tâche est associée à du code écrit en Bash. La génération du pipeline est issue de l'interprétation d'un fichier au format YAML dont la syntaxe est propre à GitLab. Ce fichier (.gitlab.ci.yml) est contenu au sein même de l'arborescence du projet Gitlab, dans le dépôt git.

Helm, Docker, Git, Bash et beaucoup des binaires courants (wget, curl, sed, etc) sont sollicités pour la réalisation des différentes tâches de CI/CD.

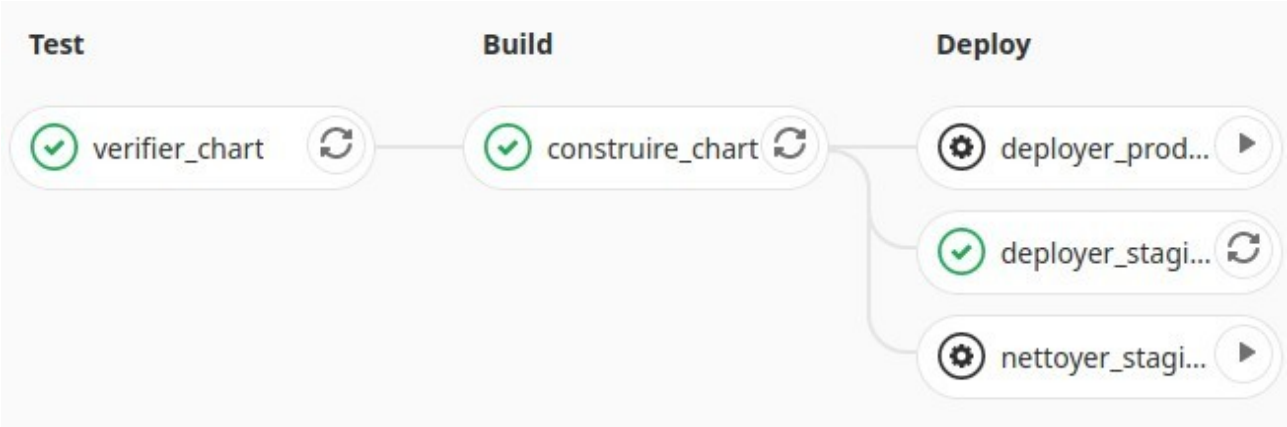

Figure 1: Pipeline dans GitLab

3.3.4 Les « triggers »

Pour rendre l'intégration continue automatique, il est impératif de faire appel aux « triggers ». Les déclencheurs permettent de définir les conditions qui doivent être remplies afin que les pipelines de CI ou de CD soient démarrés.

Il est important de comprendre que le modèle de développement et l'organisation du travail qui a été définie préalablement vont impacter la structure de notre dépôt Git ainsi que les branches qui y seront créées, ainsi que les déclencheurs qui devront être traités.

3.3.5 GitFlow/Workflow

Un workflow est une méthodologie de développement basée sur des recommandations ou de bonnes pratiques qui induisent une logique de travail visant à atteindre une cohérence et une productivité optimale.

GitFlow est un workflow Git qui met à profit les branches Git dans l'activation de déclencheurs essentiels aux processus automatisés de CI/CD. Ce modèle se base sur 4 branches qui sont associées à des versions de code dont la maturité est différente :

- « main », toute révision de cette branche est fonctionnelle, testée en pré-production et permet une mise en production. Si un développeur pousse du code (fusionnée depuis « release ») vers cette branche, celui-ci va d'abord passer tous les tests CI, et s'il en réchappe, le code sera construit/compilé puis mis en production immédiatement. Toute erreur dans cette chaîne interrompra immédiatement le processus ;
- « release », version de pré-production issue d'une version de développement qu'on jugera suffisamment mature pour entrer dans un cycle l'amenant à une livraison (une révision « main »). Une version est normalement gelée, aucune nouvelle fonction n'y sera adjointe ultérieurement. Un déclenchement sur cette branche va lancer un processus complet de CI/CD menant jusqu'au déploiement d'une version de pré-production distincte (url différente) de la version de production, mais autorisant de tester l'applicatif sans risque ;
- « develop » ici se trouvent les révisions de développement qui ne présente pas un niveau de maturité suffisant pour passer le cap de la construction et du déploiement. Seuls certains tests d'intégration seront réalisés si du code est poussé vers cette branche ;
- « hotfix » est une branche qui permet de réaliser des corrections (par le biais de patchs) de bogues impactant parfois plusieurs branches à la fois ;
- « nom de la fonctionnalité» sont des branches créées au nom de la fonctionnalité à implémenter afin d'alléger la branche « develop » et améliorer la lisibilité des gros projets ;

3.3.6 CI/CD dans GitLab

Après avoir rattaché le cluster K8S à la plateforme GitLab, j'ai mis en place un script unique et centralisé. Tous les projets GitLab incluent ce script au moyen d'une directive rajoutée dans l'en-tête du fichier optionnel .gitlab.ci.yml. Ce script personnalisé détermine la nature du projet en fonctions des fichiers qui constituent le dépôt. Un pipeline est généré en fonction du type de projet puis exécuté.Pour le moment, notre plate-forme CI/CD personnalisée gère ces différents types de projets :

- Chart Helm, il déploie l'applicatif dans le cluster et génère une adresse pour l'atteindre ;
- Dockerfile, il construit le ou les images de conteneurs et les pousse sur le dépôt ;
- PHP, il réalise un certain nombre de tests sur le code (vérification de syntaxe, formatage du code, audit de sécurité...etc) afin de s'assurer que celui-ci répond bien aux critères de qualité.

3.4 *Runners, le cœur de la CI/CD*

Afin d'exécuter les tâches de CI/CD, GitLab peut s'appuyer sur plusieurs types d'« executor » qui vont prendre en charge l'environnement d'exécution du processus de CI/CD. En voici quelques exemples : SSH, Shell, VM, Docker et Kubernetes. Il est ainsi possible par ce biais de déléguer les tâches d'intégration et de déploiement vers différentes plateformes. Dans notre infrastructure, l'usage de Kubernetes en tant qu' « exector » a été choisi.

Voici brièvement les étapes qui se déroulent lorsque l'on déclenche un processus de CI/CD :

1. Gitaly qui gère l'accès aux dépôts Git, génère un événement qui doit être traité ;
2. GitLab Shell instancie un environnement d'exécution Bash ;
3. lancement des services décrits dans le fichier de CI/CD (étape « prepare ») ;
4. instanciation d'un conteneur spécial en tâche de fond par « Sidekiq » dans lequel toutes les opérations de l'étape « pre-job » (puis de « post-job ») vont être exécutées ;
5. le dépôt Git du projet est cloné vers un répertoire /build (étape « pre_job ») ;
6. exécution du script CI/CD sous forme d'un pipeline généré à la volée depuis le fichier « .gitlab.ci.yml ». Si une image est spécifiée par le développeur/opérateur dans le fichier de CI/CD, l'étape se déroule dans ce conteneur qui est instancié avant l'exécution des étapes (étape « job ») ;
7. création du cache et chargement des « artifacts» depuis le conteneur spécial. (étape « post-job »).

Par le biais de Minio, les caches et les « artifacts » permettent de récupérer des éléments produits par une tâche afin soit, d'éviter de réitérer la procédure, soit de réutiliser ces fichiers dans un autre pipeline ou dans une autre étape/tâche.

4 Conclusion

Gitlab est une plateforme de travail collaborative qui peut s'appuyer sur l'orchestrateur de cluster Kubernetes afin de réaliser l'exécution de tâches CI/CD. Git assure l'hébergement des dépôts de code et son versionnage. L'association des deux outils, articulée par le biais d'une gestion événementielle adossée aux dépôts Git autorise l'automatisation de tests d'intégration et l'exécution de déploiements continus. L'usage d'un tel système implique des changements organisationnels.