

Formation Kubernetes



Formation Kubernetes

Apprenez Kubernetes de zéro avec des exercices pratiques !

Commencer la formation →

Voir sur GitHub ↗

Contenu de la formation [🔗](#)

Cette formation vous accompagne pas à pas dans l'apprentissage de Terraform avec AWS. Chaque partie inclut de la théorie, des exercices pratiques et des exemples concrets.



Bases de Terraform

- Installation et configuration
- Premier déploiement AWS
- Providers et ressources

- **TP1** : Serveur simple avec AWS



Configuration avancée

- Images personnalisées avec Packer
- Provisioners SSH
- VPC et networking
- **TP2-4** : Infrastructure réseau complète



Organisation des projets

- Structure des fichiers
- Variables et outputs
- Workspaces et environnements
- **TP5-6** : Projets structurés



Collaboration

- Backend S3 distant
- État partagé
- CI/CD avec Terraform
- **TP7** : Travail en équipe



Haute disponibilité

- Count et for_each
- Load balancers
- Auto-scaling
- **TP8** : Infrastructure résiliente



Modules et réutilisabilité

- Création de modules
- Gestion d'état avancée
- Refactorisation
- **TP9** : Modules personnalisés

Prérequis [🔗](#)

- Compte AWS (free tier suffisant)
- Connaissance de base de Linux/Bash
- Notions de networking (IP, DNS, ports)
- Aucune expérience Terraform requise !

Structure des TP [🔗](#)

Chaque TP suit la même structure :

1. **Théorie** : Concepts expliqués simplement
2. **Pratique** : Exercices guidés étape par étape
3. **Code** : Exemples complets commentés
4. **Vérification** : Tests de bon fonctionnement

Prêt à commencer ? [Lancez-vous avec le TP1 →](#)

Formation Kubernetes

Introduction

Bonjour à tous ! [🔗](#)

A propos de moi [🔗](#)

A propos de vous [🔗](#)

- Attentes ?

Trois transformations profondes de l'informatique [🔗](#)

Kubernetes se trouve au cœur de plusieurs transformations profondes techniques, humaines et économiques de l'informatique:

- Le mouvement DevOps et la CI/CD
- Le "Cloud"
- La conteneurisation logicielle
- Infrastructure as Code

Docker (qui est surtout la marque la plus connue pour les conteneurs en général) et Kubernetes sont des projets qui symbolisent et supportent techniquement ces transformations. D'où leur omniprésence dans les discussions informatiques actuellement.

Le mouvement DevOps [🔗](#)

- Dépasser l'opposition culturelle et de métier entre les développeurs et les administrateurs système.

- Intégrer tout le monde dans une seule équipe et ...
- Calquer les rythmes de travail sur l'organisation agile du développement logiciel
- Rapprocher techniquement la gestion de l'infrastructure du développement avec l'infrastructure as code.
 - Concrètement on écrit des fichiers de code pour gérer les éléments d'infra
 - l'état de l'infrastructure est plus claire et documentée par le code
 - la complexité est plus gérable car tout est déclaré et modifiable au fur et à mesure de façon centralisée
 - l'usage de git et des branches/tags pour la gestion de l'évolution d'infrastructure

Objectifs du DevOps

- Rapidité (**velocity**) de **déploiement** et **refactorisation** logicielle (organisation agile du développement et livraison jusqu'à plusieurs fois par jour)
 - Implique l'automatisation du déploiement et ce qu'on appelle la CI/CD c'est à dire une infrastructure de déploiement continu à partir de code.
- Passage à l'échelle (horizontal scaling) des logiciels et des équipes de développement (nécessaire pour les entreprises du cloud qui doivent servir pleins d'utilisateurs)
- Meilleure organisation des équipes
 - meilleure compréhension globale du logiciel et de son installation de production car le savoir est mieux partagé
 - organisation des équipes par thématique métier plutôt que par spécialité technique (l'équipe scale mieux)

Infrastructure as Code

On décrit en mode code un état du système:

- pas de dérive de la configuration et du système (immutabilité)
- on peut connaître de façon fiable l'état des composants du système
- on peut travailler en collaboration plus facilement (grâce à Git notamment)
- on peut faire des tests
- on facilite le déploiement de nouvelles instances

Le Cloud

Au delà du flou dans l'emploi de ce terme, le cloud est un mouvement de réorganisation technique et économique de l'informatique.

- On retourne à la consommation de "temps de calcul" et de services après une "aire du Personal Computer".
- Pour organiser cela on peut définir trois niveaux à la fois techniques et économiques de l'informatique:
 - **Software as a Service:** location de services à travers internet pour les usagers finaux
 - **Plateform as a Service:** location d'un environnement d'exécution logiciel flexible à destination des développeurs
 - **Infrastructure as a Service:** location de resources "matérielles" à la demande pour installer des logiciels sans avoir à maintenir un data center.

Le cloud permet surtout techniquement la flexibilité et la scalabilité à la demande des resources de base pour nos applications : on peut commander plein de machines et les ajouter à notre cluster. On peut également copier une infra temporairement pour faire des migrations ou tests.

Conteneurisation

La conteneurisation est permise par l'isolation au niveau du noyau du système d'exploitation du serveur : les processus sont isolés dans des namespaces au niveau du noyau. Cette innovation permet de simuler l'isolation sans ajouter une couche de virtualisation comme pour les machines virtuelles.

Ainsi les conteneurs permettent d'avoir des performances proche d'une application traditionnelle tournant directement sur le système d'exploitation hôte et ainsi d'optimiser les ressources.

Les images de conteneurs sont aussi beaucoup plus légers qu'une image de VM ce qui permet de

Les technologies de conteneurisation permettent donc de faire des boîtes isolées avec les logiciels pour apporter l'uniformisation du déploiement:

- Un façon standard de packager un logiciel (basée sur le)
- Cela permet d'assembler de grosses applications comme des legos
- Cela réduit la complexité grâce:
 - à l'intégration de toutes les dépendances déjà dans la boîte
 - au principe d'immutabilité qui implique de jeter les boîtes (automatiser pour lutter contre la culture prudence). Rend l'infrastructure prédictible.

Les conteneurs sont souvent comparés à l'innovation du porte-conteneur pour le transport de marchandise.

Apports de Docker pour le DevOps [🔗](#)

- Standardisation du déploiement : principe identique à tous les langages et environnements
- Reproductibilité / Fiabilité : lancer un conteneur se passe toujours de la même façon et limite les aléas
- Supporte l'architecture distribuée (microservices) en permettant l'isolation légère de chaque partie du logiciel

- Docker se positionne de plus en plus commercialement sur la partie amont de la conteneurisation, car sa solution d'orchestration intégrée pour la prod a perdu en popularité et glisse doucement vers le legacy.

Apports Kubernetes pour le DevOps

- Abstraction et standardisation des infrastructures:
- Langage descriptif et incrémental: on décrit ce qu'on veut plutôt que la logique complexe pour l'atteindre
- Scalabilité facilité et potentiellement illimitée
- Logique opérationnelle intégrée dans l'orchestrateur: la responsabilité des l'état du cluster est laissé au contrôleur k8s ce qui simplifie le travail

On peut alors espérer **fluidifier** la gestion des défis techniques d'un grosse application et atteindre plus ou moins la livraison logicielle continue (CD de CI/CD)

Le mouvement Cloud Native

Le mouvement **Cloud Native** est relativement récent, il remonte aux années 2010. Le terme a été popularisé en 2015 par la Cloud Native Computing Foundation (CNCF), une organisation à but non lucratif qui a été créée par la Linux Foundation pour fournir une plateforme pour la collaboration et le développement de technologies Cloud Native open source (Kubernetes et son écosystème).

La CNCF vise à promouvoir des applications qui peuvent être déployées et exécutées de manière efficace dans un environnement dynamique, pour tirer pleinement parti des avantages du cloud, tels que la scalabilité, la flexibilité et la résilience. Pour cela la CNCF promeut :

- L'usage des conteneurs qui permettent de créer des unités d'exécution indépendantes qui peuvent être facilement déployées et

orchestrées

- Une architecture d'application qui permette notamment la configuration dynamique à partir l'environnement telle que décrite ici : <https://12factor.net/>
- Les microservices permettent de découper les applications en petits services indépendants qui peuvent être déployés et gérés individuellement.

Kubernetes entre Cloud et auto-hébergement

Un des intérêts principaux de Kubernetes est de fournir un modèle de Platform as a Service (PaaS) suffisamment versatile qui permet l'interopérabilité entre des fournisseurs de clouds différents et des solutions auto-hébergées (on premise).

Cependant cette interopérabilité n'est pas automatique (pour les cas complexes) car Kubernetes permet beaucoup de variations. Concrètement il existe des variations entre les installations possibles de Kubernetes

Formation Kubernetes

Accueil

Bienvenue sur le site des supports de formation

Utilisez le menu de gauche pour parcourir les cours et TP.

Plateforme Guacamole

- Accédez à Guacamole à l'adresse fournie par le formateur, par exemple `guacamole.k8s.dopl.uk` OU `guacamole.uptime-formation.fr`.
- Accédez à votre VM via l'interface Guacamole avec le login fourni par le formateur (traditionnellement votre prenom en minuscule et sans accent et un mot de passe générique indiqué à l'oral)
- Pour accéder au copier-coller de Guacamole, il faut appuyer simultanément sur `ctrl+Alt+Shift` et utiliser la zone de texte qui s'affiche (réappuyer sur `ctrl+Alt+Shift` pour revenir à la VM).
- Cependant comme ce copier-coller est capricieux, il est conseillé d'ouvrir cette page de doc dans le navigateur à l'intérieur de guacamole et de suivre à partir de la machine distance.

Imprimer le site internet

- Est-ce vraiment nécessaire ? Normalement non car le site (un snapshot unique pour la formation) va rester en ligne pendant des

années.

- Pour le moment le site de support doit être imprimé page par page avec la fonction d'impression pdf du navigateur

Problèmes avec le snap firefox

```
$ sudo add-apt-repository ppa:mozillateam/ppa

$ echo '
Package: *
Pin: release o=LP-PPA-mozillateam
Pin-Priority: 1001

Package: firefox
Pin: version 1:1snap1-0ubuntu2
Pin-Priority: -1
' | sudo tee /etc/apt/preferences.d/mozilla-firefox

$ sudo snap remove firefox

$ sudo apt install firefox
```

Formation Kubernetes

Introduction

Bonjour à tous ! 

A propos de moi 

A propos de vous 

- Attentes ?

quelques aspects pratiques: 

- Le lab : se connecter à <https://guacamole.kube.dopl.uk/guacamole>
 - Login : votre prenom en minuscule sans accents
 - Mot de passe : fournis par le formateur
- Supports : ce site qui restera en ligne pendant les prochaines années (et imprimable en pdf joliment grâce à la fonction d'impression de chrome/chromium)
- émargement au début de chaque demi journée

Trois transformations profondes de l'informatique 



Kubernetes se trouve au coeur de plusieurs transformations profondes techniques, humaines et économiques de l'informatique:

- Le mouvement DevOps et la CI/CD
- Le "Cloud"
- La conteneurisation logicielle
- Infrastructure as Code

Docker (qui est surtout la marque la plus connue pour les conteneurs en général) et Kubernetes sont des projets qui symbolisent et supportent techniquement ces transformations. D'où leur omniprésence dans les discussions informatiques actuellement.

Le mouvement DevOps [🔗](#)

- Dépasser l'opposition culturelle et de métier entre les développeurs et les administrateurs système.
- Intégrer tout le monde dans une seule équipe et ...
- Calquer les rythmes de travail sur l'organisation agile du développement logiciel
- Rapprocher techniquement la gestion de l'infrastructure du développement avec l'infrastructure as code.
 - Concrètement on écrit des fichiers de code pour gérer les éléments d'infra
 - l'état de l'infrastructure est plus claire et documentée par le code

- la complexité est plus gérable car tout est déclaré et modifiable au fur et à mesure de façon centralisée
- l'usage de git et des branches/tags pour la gestion de l'évolution d'infrastructure

Objectifs du DevOps

- Rapidité (**velocity**) de **déploiement** et **refactorisation** logicielle (organisation agile du développement et livraison jusqu'à plusieurs fois par jour)
 - Implique l'automatisation du déploiement et ce qu'on appelle la CI/CD c'est à dire une infrastructure de déploiement continu à partir de code.
- Passage à l'échelle (horizontal scaling) des logiciels et des équipes de développement (nécessaire pour les entreprises du cloud qui doivent servir pleins d'utilisateurs)
- Meilleure organisation des équipes
 - meilleure compréhension globale du logiciel et de son installation de production car le savoir est mieux partagé
 - organisation des équipes par thématique métier plutôt que par spécialité technique (l'équipe scale mieux)

Infrastructure as Code

On décrit en mode code un état du système:

- pas de dérive de la configuration et du système (immutabilité)
- on peut connaître de façon fiable l'état des composants du système
- on peut travailler en collaboration plus facilement (grâce à Git notamment)
- on peut faire des tests
- on facilite le déploiement de nouvelles instances

Le Cloud

Au delà du flou dans l'emploi de ce terme, le cloud est un mouvement de réorganisation technique et économique de l'informatique.

- On retourne à la consommation de "temps de calcul" et de services après une "aire du Personal Computer".
- Pour organiser cela on peut définir trois niveaux à la fois techniques et économiques de l'informatique:
 - **Software as a Service:** location de services à travers internet pour les usagers finaux
 - **Platform as a Service:** location d'un environnement d'exécution logiciel flexible à destination des développeurs
 - **Infrastructure as a Service:** location de ressources "matérielles" à la demande pour installer des logiciels sans avoir à maintenir un data center.

Le cloud permet surtout techniquement la flexibilité et la scalabilité à la demande des ressources de base pour nos applications : on peut commander plein de machines et les ajouter à notre cluster. On peut également copier une infra temporairement pour faire des migrations ou tests.

Conteneurisation

La conteneurisation est permise par l'isolation au niveau du noyau du système d'exploitation du serveur : les processus sont isolés dans des namespaces au niveau du noyau. Cette innovation permet de simuler l'isolation sans ajouter une couche de virtualisation comme pour les machines virtuelles.

Ainsi les conteneurs permettent d'avoir des performances proche d'une application traditionnelle tournant directement sur le système d'exploitation hôte et ainsi d'optimiser les ressources.

Les images de conteneurs sont aussi beaucoup plus légers qu'une image de VM ce qui permet de

Les technologies de conteneurisation permettent donc de faire des boîtes isolées avec les logiciels pour apporter l'uniformisation du déploiement:

- Un façon standard de packager un logiciel (basée sur le)
- Cela permet d'assembler de grosses applications comme des legos
- Cela réduit la complexité grâce:
 - à l'intégration de toutes les dépendances déjà dans la boîte
 - au principe d'immutabilité qui implique de jeter les boîtes (automatiser pour lutter contre la culture prudence). Rend l'infrastructure prédictible.

Les conteneurs sont souvent comparés à l'innovation du porte conteneur pour le transport de marchandise.

Apports de Docker pour le DevOps [🔗](#)

- Standardisation du déploiement : principe identique à tous les langages et environnements
- Reproductibilité / Fiabilité : lancer un conteneur se passe toujours de la même façon et limite les aléas
- Supporte l'architecture distribuée (microservices) en permettant l'isolation légère de chaque partie du logiciel
- Docker se positionne de plus en plus commercialement sur la partie amont de la conteneurisation, car sa solution d'orchestration intégrée pour la prod a perdu en popularité et glisse doucement vers le legacy.

Apports Kubernetes pour le DevOps [🔗](#)

- Abstraction et standardisation des infrastructures:

- Langage descriptif et incrémental: on décrit ce qu'on veut plutôt que la logique complexe pour l'atteindre
- Scalabilité facilité et potentiellement illimitée
- Logique opérationnelle intégrée dans l'orchestrateur: la responsabilité des l'état du cluster est laissé au contrôleur k8s ce qui simplifie le travail

On peut alors espérer **fluidifier** la gestion des défis techniques d'un grosse application et atteindre plus ou moins la livraison logicielle continue (CD de CI/CD)

Le mouvement Cloud Native

Le mouvement **Cloud Native** est relativement récent, il remonte aux années 2010. Le terme a été popularisé en 2015 par la Cloud Native Computing Foundation (CNCF), une organisation à but non lucratif qui a été créée par la Linux Foundation pour fournir une plateforme pour la collaboration et le développement de technologies Cloud Native open source (Kubernetes et son écosystème).

La CNCF vise à promouvoir des applications qui peuvent être déployées et exécutées de manière efficace dans un environnement dynamique, pour tirer pleinement parti des avantages du cloud, tels que la scalabilité, la flexibilité et la résilience. Pour cela la CNCF promeut :

- L'usage des conteneurs qui permettent de créer des unités d'exécution indépendantes qui peuvent être facilement déployées et orchestrées
- Une architecture d'application qui permette notamment la configuration dynamique à partir l'environnement telle que décrite ici : <https://12factor.net/>
- Les microservices permettent de découper les applications en petits services indépendants qui peuvent être déployés et gérés individuellement.

Kubernetes entre Cloud et auto-hébergement

Un des intérêts principaux de Kubernetes est de fournir un modèle de Platform as a Service (PaaS) suffisamment versatile qui permet l'interopérabilité entre des fournisseurs de clouds différents et des solutions auto-hébergées (on premise).

Cependant cette interopérabilité n'est pas automatique (pour les cas complexes) car Kubernetes permet beaucoup de variations. Concrètement il existe des variations entre les installations possibles de Kubernetes

Formation Kubernetes

Outils de la formation

Le dépôt git “fil rouge” [🔗](#)

Un dépôt va vous fournir les corrections de TP durant la formation :

<https://github.com/e-lie/formation-kube-unified> :

Ouvrir le bureau VNC [🔗](#)

VSCode [🔗](#)

Plateforme Guacamole [🔗](#)

- Accédez à Guacamole à l'adresse fournie par le formateur, par exemple `guacamole.k8s.dopl.uk` OU `guacamole.uptime-formation.fr`.
- Accédez à votre VM via l'interface Guacamole avec le login fourni par le formateur (traditionnellement votre prenom en minuscule et sans accent et un mot de passe générique indiqué à l'oral)
- Pour accéder au copier-coller de Guacamole, il faut appuyer simultanément sur `ctrl+Alt+Shift` et utiliser la zone de texte qui s'affiche (réappuyer sur `ctrl+Alt+Shift` pour revenir à la VM).
- Cependant comme ce copier-coller est capricieux, il est conseillé d'ouvrir cette page de doc dans le navigateur à l'intérieur de guacamole et de suivre à partir de la machine distance.

Quelques soucis épisodiques et solutions [🔗](#)

avec le snap firefox [🔗](#)

```
$ sudo add-apt-repository ppa:mozillateam/ppa

$ echo '
Package: *
Pin: release o=LP-PPA-mozillateam
Pin-Priority: 1001

Package: firefox
Pin: version 1:1snap1-0ubuntu2
Pin-Priority: -1
' | sudo tee /etc/apt/preferences.d/mozilla-firefox

$ sudo snap remove firefox

$ sudo apt install firefox
```

Formation Kubernetes

TP - installer un cluster de dev

Choisir une distribution kubernetes de dev

Voir le cours “différents types de cluster”

- `minikube` est une distribution spécialisée pour le dev => avantage : la distribution la plus connue, bien documentée et utilisée dans tous les tutoriels
- `k3s` est une distribution simple à installer pour le dev ou la prod en particulier pour des contextes edge computing. => avantage : permet de faire de la prod / s’installe sur un serveur.
- `kind` (Kubernetes IN Docker) une distribution basée sur Docker. avantages : léger, rapide à démarrer, supporte le multinoeud, utilise kubeadm en arrière plan. Kind un cluster kube très standard / vanilla mais proche de la prod (multinoeud, lb). Permet de tester la plupart des solutions y compris des plugins CNI (réseau) et CSI (stockage).

Nous allons installer l’une ici.

Installer `Minikube` (facultatif)

- Pour installer minikube la méthode recommandée est indiquée ici:
<https://minikube.sigs.k8s.io/docs/start/>

Nous utiliserons classiquement `docker` comme runtime pour minikube (les noeuds k8s seront des conteneurs simulant des serveurs). Ceci est, bien sûr, une configuration de développement. Elle se comporte cependant de façon très proche d'un véritable cluster.

- Pour lancer le cluster faites simplement: `minikube start` (il est également possible de préciser le nombre de coeurs de calcul, la mémoire et d'autre paramètre pour adapter le cluster à nos besoins.)

Minikube configure automatiquement kubectl (dans le fichier `~/.kube/config`) pour qu'on puisse se connecter au cluster de développement.

- Testez la connexion avec `kubectl get nodes`.

Affichez à nouveau la version `kubectl version`. Cette fois-ci la version de Kubernetes qui tourne sur le cluster actif est également affichée. Idéalement le client et le cluster devrait être dans la même version mineure par exemple `1.20.x`.

Installer kind (facultatif) [🔗](#)

> Installer et configurer kind

Installer k3s (facultatif) [🔗](#)

> Installer et configurer kind

Ajouter les connexions à Lens [🔗](#)

Lens permet de chercher les kubeconfigs via l'interface et d'enregistrer plusieurs cluster dans la hotbar à gauche.

Le contexte de kubectl dans le terminal de Lens est automatiquement celui du cluster actuellement sélectionné. On peut s'en rendre compte en lançant `kubectl get nodes` dans deux terminaux dans chacun des deux cluster dans Lens.

Facultatif : merger la configuration kubectl [🔗](#)

- Pour dumper la configuration fusionnée des fichiers et l'exporter on peut utiliser: `kubectl config view --flatten >> ~/.kube/merged.yaml`.

Facultatif 3e installation : Un cluster K8s managé, exemple avec Scaleway

Le formateur peut louer pour vous montrer un cluster kubernetes managé. Vous pouvez également louez le votre si vous préférez en créant un compte chez ce provider de cloud.

- Créez un compte (ou récupérez un accès) sur [Scaleway](#).
- Créez un cluster Kubernetes avec [l'interface Scaleway](#)

La création prend environ 5 minutes.

- Sur la page décrivant votre cluster, un gros bouton en bas de la page vous incite à télécharger ce même fichier `kubeconfig` (*Download Kubeconfig*).

Ce fichier contient la **configuration kubectl** adaptée pour la connexion à notre cluster.

Facultatif 4e installation : un cluster avec `kubeadm` ou méthode The Hard Way

Voir le TP facultatif.

Formation Kubernetes

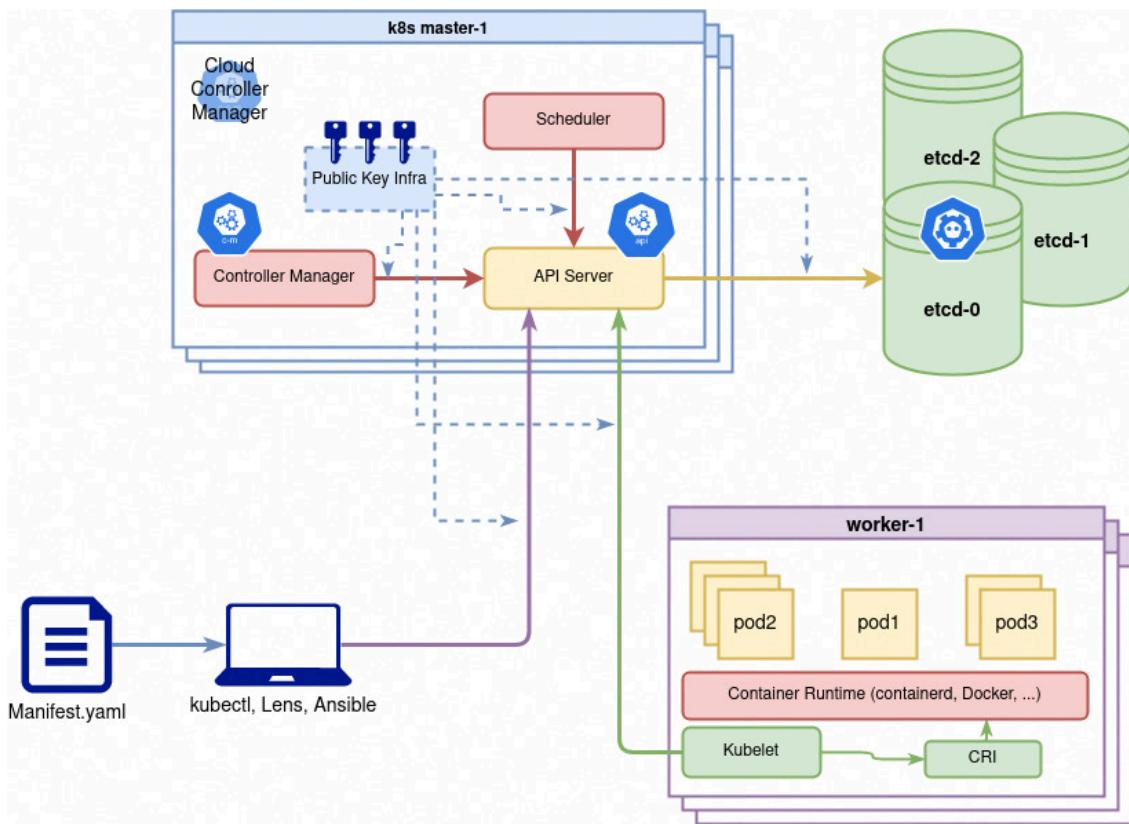
Cours - Les différents types de clusters

Kubernetes

- Kubernetes est une solution d'orchestration de conteneurs extrêmement populaire.
- Le projet est très ambitieux : une façon de considérer son ampleur est de voir Kubernetes comme un système d'exploitation (et un standard ouvert) pour les applications distribuées et le cloud.
- Le projet est développé en Open Source au sein de la Cloud Native Computing Foundation.

Architecture de Kubernetes

Kubernetes a une architecture master/worker ce qui signifie que certains certains noeuds du cluster contrôlent l'exécution par les autres noeuds de logiciels ou jobs.



Les noeuds Kubernetes [🔗](#)

Les noeuds d'un cluster sont les machines (serveurs physiques, machines virtuelles, etc. généralement Linux mais plus nécessairement aujourd'hui) qui exécutent vos applications et vos workflows. Tous les noeuds d'un cluster qui ont besoin de faire tourner des tâches (workloads) kubernetes utilisent trois services de base:

- Comme les workloads sont généralement conteneurisés chaque noeud doit avoir une *runtime de conteneur* compatible avec la norme Container Runtime Interface (CRI) : containerd, cri-o OU Docker n'est pas la plus recommandée bien que la plus connue. Il peut également s'agir d'une runtime utilisant de la virtualisation.
- Le `kubelet` composant (binaire en go, le seul composant jamais conteneurisé) qui contrôle la création et l'état des pods/conteneur sur son noeud.
- D'autres composants et drivers pour fournir fonctionnalités réseau (Container Network Interface - CNI et `kube-proxy`) ainsi que fonctionnalités de stockage (Drivers Container Storage Interface (csi))

Pour utiliser Kubernetes, on définit un état souhaité en créant des ressources (pods/conteneurs, volumes, permissions etc). Cet état souhaité et son application est géré par le `control plane` composé des noeuds master.

Les noeuds master kubernetes forment le `control plane` du Cluster

Le control plane est responsable du maintien de l'état souhaité des différents éléments de votre cluster. Lorsque vous interagissez avec Kubernetes, par exemple en utilisant l'interface en ligne de commande `kubectl`, vous communiquez avec les noeuds master de votre cluster (plus précisément l'`API Server`).

Le control plane conserve un enregistrement de tous les objets Kubernetes du système. À tout moment, des boucles de contrôle s'efforcent de faire converger l'état réel de tous les objets du système pour correspondre à l'état souhaité que vous avez fourni. Pour gérer l'état réel de ces objets sous forme de conteneurs (toujours) avec leur configuration le control plane envoie des instructions aux différents kubelets des noeuds.

Donc concrètement les noeuds du control plane Kubernetes font tourner, en plus de `kubelet` et `kube-proxy`, un ensemble de services de contrôle:

- `kube-apiserver`: expose l'API (rest) kubernetes, point d'entrée central pour la communication interne (intercomposants) et externe (`kubectl` ou autre) au cluster.
- `kube-controller-manager`: contrôle en permanence l'état des resources et essaie de le corriger s'il n'est plus conforme.
- `kube-scheduler`: surveille et cartographie les resources matérielles et les contraintes de placement des pods sur les différents noeuds pour décider ou doivent être créés ou supprimés les conteneurs/pods.
- `cloud-controller-manager`: Composant *facultatif* qui gère l'intégration avec le fournisseur de cloud comme par exemple la création automatique de loadbalancers pour exposer les applications kubernetes à l'extérieur du cluster.

L'ensemble de la configuration kubernetes est stockée de façon résiliante (consistance + haute disponibilité) dans un gestionnaire configuration distribué qui est généralement `etcd`.

`etcd` peut être installé de façon redondante sur les noeuds du control plane ou configuré comme un système externe sur un autre ensemble de serveurs.

Lien vers la documentation pour plus de détails sur les composants :

<https://kubernetes.io/docs/concepts/overview/components/>

Configuration de connexion `kubeconfig`

Pour se connecter, `kubectl` a besoin de l'adresse de l'API Kubernetes, d'un nom d'utilisateur et d'un certificat.

- Ces informations sont fournies sous forme d'un fichier YAML appelé `kubeconfig`
- Comme nous le verrons en TP ces informations sont généralement fournies directement par le fournisseur d'un cluster k8s (provider ou k8s de dev)

Le fichier `kubeconfig` par défaut se trouve sur Linux à l'emplacement

`~/.kube/config`.

On peut aussi préciser la configuration au *runtime* comme ceci: `kubectl --`

`kubeconfig=fichier_kubeconfig.yaml <commandes_k8s>`

Le même fichier `kubeconfig` peut stocker plusieurs configurations dans un fichier YAML :

Exemple :

```

apiVersion: v1

clusters:
- cluster:
  certificate-authority: /home/jacky/.minikube/ca.crt
  server: https://172.17.0.2:8443
  name: minikube
- cluster:
  certificate-authority-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURKekNDQWcrZ0F3SUJBZ0lDQm5Vd0RRWUpLb1pJaHZj
  server: https://5ba26bee-00f1-4088-ae11-22b6dd058c6e.k8s.ondigitalocean.com
  name: do-lon1-k8s-tp-cluster

contexts:
- context:
  cluster: minikube
  user: minikube
  name: minikube
- context:
  cluster: do-lon1-k8s-tp-cluster
  user: do-lon1-k8s-tp-cluster-admin
  name: do-lon1-k8s-tp-cluster
current-context: do-lon1-k8s-tp-cluster

kind: Config
preferences: {}

users:
- name: do-lon1-k8s-tp-cluster-admin
  user:
    token: 8b2d33e45b980c8642105ec827f41ad343e8185f6b4526a481e312822d634aa4
- name: minikube
  user:
    client-certificate: /home/jacky/.minikube/profiles/minikube/client.crt
    client-key: /home/jacky/.minikube/profiles/minikube/client.key

```

Ce fichier déclare 2 clusters (un local, un distant), 2 contextes et 2 users.

Créer un cluster Kubernetes [🔗](#)

Installation de développement [🔗](#)

Pour installer un cluster de développement :

- solution officielle : Minikube, tourne dans Docker par défaut (ou dans des VMs)

- solution très pratique et “vanilla”: kind
- avec Docker Desktop depuis peu (dans une VM aussi)
- un cluster léger avec `k3s`, de Rancher (simple et utilisable en production/edge)

Créer un cluster en tant que service (*managed cluster*) chez un fournisseur de cloud. [🔗](#)

Tous les principaux fournisseurs de cloud proposent depuis plus ou moins longtemps des solutions de cluster gérées par eux (KaaS, Kubernetes as a service):

- Google Cloud Plateform avec Google Kubernetes Engine (GKE) : très populaire car très flexible et l'implémentation de référence de Kubernetes.
- AWS avec EKS : Kubernetes à la sauce Amazon pour la gestion de l'accès, des loadbalancers ou du scaling.
- Azure avec AKS : Kubernetes à la sauce Microsoft.
- DigitalOcean ou Linode : un peu moins de fonctions mais plus simple à appréhender

Les gros fournisseurs proposent tous des services éprouvés, il s'agit surtout de faciliter l'intégration avec l'existant: Si vous utilisez déjà des resources AWS ou Azure il est plus commode de louer chez l'un d'eux votre cluster.

Fournisseurs français de Kubernetes as a service: OVH, Scaleway

Installer un cluster de production on premise : l'outil “officiel” `kubeadm` [🔗](#)

`kubeadm` est un utilitaire (on parle parfois d'opérateur) aider à générer les certificats et les configurations spéciques pour le control plane et connecter les noeuds au control plane. Il permet également d'assister les taches de maintenance comme la mise à jour progressive (rolling) de chaque noeud du cluster.

- Installer le dæmon `kubelet` sur tous les noeuds
- Installer l'outil de gestion de cluster `kubeadm` sur un noeud master
- Générer les bons certificats avec `kubeadm`
- Installer un réseau CNI k8s comme `flannel` (d'autres sont possible et le choix vous revient)
- Déployer la base de données `etcd` avec `kubeadm`
- Connecter les nœuds worker au master.

L'installation est décrite dans la [documentation officielle](#)

Opérer et maintenir un cluster de production Kubernetes "à la main" est très complexe et une tâche à ne pas prendre à la légère. De nombreux éléments doivent être installés et géré par une équipe opérationnelle.

- Mise à jour et passage de version de kubernetes qui doit être fait très régulièrement car une version n'est supportée que 2 ans.
- Choix d'une configuration réseau et de sécurité adaptée.
- Installation probable de système de stockage distribué comme Ceph à maintenir également dans le temps.
- Etc.

Kubespray : intégration "officielle" de `kubeadm` et Ansible pour gérer un cluster [🔗](#)

<https://kubespray.io/#/>

En réalité utiliser `kubeadm` directement en ligne de commande n'est pas la meilleure approche car cela ne respecte pas l'infrastructure as code et rend plus périlleux la maintenance/maj du cluster par la suite.

Le projet kubespray est un installer de cluster kubernetes utilisant Ansible et `kubeadm`. C'est probablement l'une des méthodes les plus populaires pour véritablement gérer un cluster de production on premise.

Mais la encore il s'agit de ne pas sous-estimer la complexité de la maintenance (comme avec `kubeadm`).

Installer un cluster complètement à la main pour s'exercer

On peut également installer Kubernetes de façon encore plus manuelle pour mieux comprendre ses rouages et composants. Ce type d'installation est décrite par exemple ici : [Kubernetes the hard way](#).

Il existe également une série de tutoriel pour faire cette installation manuelle à l'aide d'Ansible

Quelques PaaS (Plateforme as a Service) basés sur Kubernetes

- `Rancher`: Un écosystème Kubernetes très complet, assez *opinionated* et entièrement open-source, non lié à un fournisseur de cloud. Inclut l'installation de stack de monitoring (Prometheus), de logging, de réseau mesh (Istio) via une interface web agréable. Rancher maintient aussi de nombreuses solutions open source, comme par exemple Longhorn pour le stockage distribué.
- `openshift` : Une version de Kubernetes configurée et optimisée par Red Hat pour être utilisée dans son écosystème. Elle intègre notamment du monitoring et monitoring, Jenkins&Tekton pour le déploiement, un registry d'image etc. Tout est intégré avec l'inconvénient d'être un peu captif·ve de l'écosystème et des services vendus par Red Hat.

Bibliographie pour approfondir le choix d'une distribution Kubernetes :

- Chapitre 3 du livre `Cloud Native DevOps with Kubernetes` chez O'reilly

Formation Kubernetes

TP - Découvrir la CLI kubectl et déployer une application

Découverte de Kubernetes [🔗](#)

Installer le client CLI [kubectl](#) [🔗](#)

kubectl est le point d'entrée universel pour contrôler tous les types de cluster kubernetes. C'est un client en ligne de commande qui communique en REST avec l'API d'un cluster.

kubectl peut gérer plusieurs clusters/configurations et switcher entre ces configurations. Pour que cette fonctionnalité soit plus confortable on installe souvent l'addon kubectx en plus.

La méthode d'installation importe peu ici. Pour installer kubectl sur Ubuntu nous pouvons lancer: `sudo snap install kubectl --classic`.

- Faites `kubectl version` pour afficher la version du client kubectl.

Bash completion et racourcis [🔗](#)

Pour permettre à kubectl de compléter le nom des commandes et ressources avec `<Tab>` il est utile d'installer l'autocomplétion pour Bash :

```

● ● ●

sudo apt install bash-completion

source <(kubectl completion bash)

echo "source <(kubectl completion bash)" >> ${HOME}/.bashrc

```

Vous pouvez désormais appuyer sur `<Tab>` pour compléter vos commandes `kubectl`, c'est très utile !

- Notez également que pour gagner du temps en ligne de commande, la plupart des mots-clés de type Kubernetes peuvent être abrégés :
 - `services` devient `svc`
 - `deployments` devient `deploy`
 - etc.

La liste complète : <https://blog.heptio.com/kubectl-resource-short-names-heptioprotip-c8eff9fb7202>

Explorons notre cluster k8s

Notre cluster k8s est plein d'objets divers, organisés entre eux de façon dynamique pour décrire des applications, tâches de calcul, services et droits d'accès. La première étape consiste à explorer un peu le cluster :

- Listez les nodes pour récupérer le nom de l'unique node (`kubectl get nodes`) puis affichez ses caractéristiques avec `kubectl describe node/minikube` OU `kubectl describe node minikube`.

La commande `get` est générique et peut être utilisée pour récupérer la liste de tous les types de ressources ou d'afficher les informations d'une ressource précise.

Pour désigner un seul objet, il faut préfixer le nom de l'objet par son type (ex : `kubectl get nodes minikube` OU `kubectl get node/minikube`) car k8s ne peut pas

deviner ce que l'on cherche quand plusieurs ressources de types différents ont le même nom.

De même, la commande `describe` peut s'appliquer à tout objet k8s.

- Pour afficher tous les types de ressources à la fois que l'on utilise :

```
kubectl get all
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	2m34s

Il semble qu'il n'y a qu'une ressource dans notre cluster. Il s'agit du service d'API Kubernetes, pour que les pods/conteneurs puissent utiliser la découverte de service pour communiquer avec le cluster.

En réalité il y en a généralement d'autres cachés dans les autres `namespaces`. En effet les éléments internes de Kubernetes tournent eux-mêmes sous forme de services et de daemons Kubernetes. Les `namespaces` sont des groupes qui servent à isoler les ressources de façon logique et en termes de droits (avec le *Role-Based Access Control* (RBAC) de Kubernetes).

Pour vérifier cela on peut :

- Afficher les `namespaces` : `kubectl get namespaces`

Un cluster Kubernetes a généralement un namespace appelé `default` dans lequel les commandes sont lancées et les ressources créées si on ne précise rien. Il a également aussi un namespace `kube-system` dans lequel résident les processus et ressources système de k8s. Pour préciser le namespace on peut rajouter l'argument `-n` à la plupart des commandes k8s.

- Pour lister les ressources liées au `kubectl get all -n kube-system`.
- Ou encore : `kubectl get all --all-namespaces` (peut être abrégé en `kubectl get all -A`) qui permet d'afficher le contenu de tous les namespaces en

même temps.

- Pour avoir des informations sur un namespace : `kubectl describe namespace/kube-system`

Déployer une application en CLI [🔗](#)

Nous allons maintenant déployer une première application conteneurisée. Le déploiement est un peu plus complexe qu'avec Docker, en particulier car il est séparé en plusieurs objets et plus configurable.

- Pour créer un déploiement en ligne de commande (par opposition au mode déclaratif que nous verrons plus loin), on peut lancer par exemple: `kubectl create deployment demonstration --image=monachus/rancher-demo.`

Cette commande crée un objet de type `deployment`. Nous pourrons étudier ce deployment avec la commande `kubectl describe deployment/demonstration`.

- Notez la liste des événements sur ce déploiement en bas de la description.
- De la même façon que dans la partie précédente, listez les `pods` avec `kubectl`. Combien y en a-t-il ?
- Agrandissons ce déploiement avec `kubectl scale deployment demonstration --replicas=5`
- `kubectl describe deployment/demonstration` permet de constater que le service est bien passé à 5 replicas.
 - Observez à nouveau la liste des événements, le scaling y est enregistré...
 - Listez les pods pour constater

A ce stade impossible d'afficher l'application : le déploiement n'est pas encore accessible de l'extérieur du cluster. Pour régler cela nous devons l'exposer grâce à un service :

- `kubectl expose deployment demonstration --type=NodePort --port=8080 --name=demonstration-service`
- Affichons la liste des services pour voir le résultat: `kubectl get services`

Un service permet de créer un point d'accès unique exposant notre déploiement. Ici nous utilisons le type Nodeport car nous voulons que le service soit accessible de l'extérieur par l'intermédiaire d'un forwarding de port.

Avec minikube ce forwarding de port doit être concrétisé avec la commande `minikube service demonstration-service`. Normalement la page s'ouvre automatiquement et nous voyons notre application.

- Sauriez-vous expliquer ce que l'app fait ?
- Pour le comprendre ou le confirmer, diminuez le nombre de réplicats à l'aide de la commande utilisée précédemment pour passer à 5 réplicats. Qu se passe-t-il ?

Une autre méthode pour accéder à un service (quel que soit son type) en mode développement est de forwarder le traffic par l'intermédiaire de kubectl (et des composants kube-proxy installés sur chaque noeuds du cluster).

- Pour cela on peut par exemple lancer: `kubectl port-forward pod demonstration-..... 8080:8080 --address 127.0.0.1` à remplacer par un de vos pods.
- Vous pouvez désormais accéder à votre app via `kubectl` sur: `http://localhost:8080`. Quelle différence avec l'exposition précédente via `minikube` ?

=> Un seul conteneur s'affiche. En effet `kubectl port-forward` sert à créer une connexion de développement/debug qui pointe toujours vers le même pod en arrière plan.

Pour exposer cette application en production sur un véritable cluster, nous devrions plutôt avoir recours à service de type un LoadBalancer. Mais

minikube ne propose pas par défaut de loadbalancer. Nous y reviendrons dans le cours sur les objets kubernetes.

CheatSheet pour kubectl et formattage de la sortie [🔗](#)

<https://kubernetes.io/docs/reference/kubectl/cheatsheet/>

Vous noterez dans cette page qu'il est possible de traiter la sortie des commandes kubectl de multiple façon (yaml, json, gotemplate, jsonpath, etc)

Le mode de sortie le plus utilisé pour filtrer une information parmis l'ensemble des caractéristiques d'une resource est `jsonpath` qui s'utilise comme ceci:

```
kubectl get pod <tab>
kubectl get pod demonstration-7645747fc6-f5z55 -o yaml # pour afficher la spécification
kubectl get pod demonstration-7645747fc6-f5z55 -o
  jsonpath='{.spec.containers[0].image}' # affiche le nom de l'image
```

Essayez de la même façon d'afficher le nombre de répliques de notre déploiement.

Des outils CLI supplémentaires pour le confort [🔗](#)

- `helm` package/template manager pour Kubernetes => voir Cours et TP
- `tanka` langage alternatif de déploiement (bonus)

`kubectl` est puissant et flexible mais il est peu confortable certaines actions courantes. Il est intéressant d'ajouter d'autres outils pour le complémenter :

- `kubectx` qui viens avec `kubens` un outil pour switcher de cluster/contexte/namespace courant confortablement
- `viddy` un watch amélioré pour visualiser en temps réel les resources du cluster et leur évolution
- `skaffold` pour développer => voir tp “développer directement dans un cluster”
- `stern` pour pouvoir afficher/tail les logs des pods correctement (notamment via un service)
- `trivy` pour des analyses de sécurité des images et du cluster

Exemple d'utilisation:

```

  •••
  viddy
  stern my-service -n my-namespace --tail 100 --include "error|warning"
  stern -t my-service ...

```

Pour installer tous ces outils il y a de nombreuses méthodes (snap/krew/installation manuelle github etc). Une façon uniforme pour avoir des version récentes et multiples sur n'importe quel OS (linux, macOS et windows avec le WSL) est le gestionnaire de dépendance de dev `asdf-vm`.

> **On peut installer tout cela avec le code suivant sous linux (via git et bash):**

Au-delà de la ligne de commande... [🔗](#)

Accéder à la dashboard Kubernetes [🔗](#)

Le moyen le plus classique pour avoir une vue d'ensemble des ressources d'un cluster est d'utiliser la Dashboard officielle. Cette Dashboard est généralement installée par défaut lorsqu'on loue un cluster chez un provider.

On peut aussi l'installer dans minikube ou k3s. Nous allons ici préférer le client lourd OpenLens

Installer OpenLens

Lens est un logiciel graphique (un client “lourd”) pour contrôler Kubernetes. Il se connecte en utilisant kubectl et la configuration `~/.kube/config` par défaut et nous permettra d'accéder à un dashboard puissant et agréable à utiliser.

Récemment Mirantis qui a racheté Lens essaye de fermer l'accès à ce logiciel open source. Il faut donc utiliser le build communautaire à la place du build officiel:

<https://github.com/MuhammedKalkan/OpenLens/releases>

Vous pouvez l'installer en lançant ces commandes :

```
## Install Lens
export LENS_VERSION=5.5.4 # change with the current stable version
curl -LO
"https://github.com/MuhammedKalkan/OpenLens/releases/download/v$LENS_VERSION/OpenLens"
sudo dpkg -i "OpenLens-$LENS_VERSION.deb"
```

- Lancez l'application `Lens` dans le menu “internet” de votre machine (VNC).
- Sélectionnez le cluster de votre choix la liste et épinglez la connection dans la barre de menu
- Explorons ensemble les ressources dans les différentes rubriques et namespaces

Formation Kubernetes

Cours - Le langage Kubernetes

L'API et les Objets Kubernetes

Utiliser Kubernetes consiste à déclarer des objets grâce à l'API Kubernetes pour décrire l'état souhaité d'un cluster : quelles applications ou autres processus exécuter, quelles images elles utilisent, le nombre de replicas, les ressources réseau et disque que vous mettez à disposition, etc.

On définit des objets généralement via l'interface en ligne de commande et `kubectl` de deux façons :

- en lançant une commande `kubectl run <conteneur> ...`, `kubectl expose ...`
- en décrivant un objet dans un fichier YAML ou JSON et en le passant au client `kubectl apply -f monpod.yaml`

Vous pouvez également écrire des programmes qui utilisent directement l'API Kubernetes pour interagir avec le cluster et définir ou modifier l'état souhaité. **Kubernetes est complètement automatisable !**

La commande `apply`

Kubernetes encourage le principe de l'infrastructure-as-code : il est recommandé d'utiliser une description YAML et versionnée des objets et configurations Kubernetes plutôt que la CLI.

Pour cela la commande de base est `kubectl apply -f object.yaml`.

La commande inverse `kubectl delete -f object.yaml` permet de détruire un objet précédemment appliqué dans le cluster à partir de sa description.

Lorsqu'on vient d'appliquer une description on peut l'afficher dans le terminal avec `kubectl apply -f myobj.yaml view-last-applied`

Parenthèse : Le YAML [🔗](#)

Kubernetes décrit ses ressources en YAML. A quoi ça ressemble, YAML ?

```
- marché:
    lieu: Marché de la Place
    jour: jeudi
    horaire:
        unité: "heure"
        min: 12
        max: 20
    fruits:
        - nom: pomme
          couleur: "verte"
          pesticide: avec

        - nom: poires
          couleur: jaune
          pesticide: sans
    légumes:
        - courgettes
        - salade
        - potiron
```

Syntaxe [🔗](#)

- Alignement ! (**2 espaces !!**)
- des listes (tirets)
- des dictionnaires composés d'un ensemble de paires **clé: valeur** (dans n'importe quel ordre !)
- Un peu comme du JSON, avec cette grosse différence que le JSON se fiche de l'alignement et met des accolades et des points-virgules
- **les extensions Kubernetes et YAML dans VSCode vous aident à repérer des erreurs**

Syntaxe de base d'une description YAML

Kubernetes [🔗](#)

Les descriptions YAML permettent de décrire de façon lisible et manipulable de nombreuses caractéristiques des ressources Kubernetes (un peu comme un *Compose file* par rapport à la CLI Docker).

Exemple [🔗](#)

Création d'un service simple :

```
kind: Service
apiVersion: v1
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kubernetes-dashboard
spec:
  ports:
    - port: 443
      targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboard
  type: NodePort
```

Organisation de la syntaxe :

- Les ressources k8s sont de dictionnaires !
- Toutes les descriptions doivent commencer par spécifier le module d'API et sa version à partir de laquelle notre objet peut être créé et manipulé.
- Il faut ensuite préciser le type d'objet avec `kind`

S'ensuit deux sections présentes dans toutes descriptions Kubernetes:

- La section `metadata` est identique pour chaque type de ressource et contient en particulier le nom, les étiquettes diverses, et

éventuellement le namespace de la ressource. Le nom dans `metadata:\nname: value` est également obligatoire.

- La section `spec` qui précise tous les paramètres désirés spécifiques à la resource en question.

Les paramètres de la spec sont pour beaucoup facultatifs et prennent leur valeur par défaut s'ils ne sont pas écrit explicitement.

On peut aller observer les paramètres en ligne de commande avec la commande `kubectl explain` exemple:

```
● ● ●

kubectl explain pod
kubectl explain pod.spec.containers
kubectl explain deploy --recursive
```

Description de plusieurs ressources

- On peut mettre plusieurs ressources à la suite dans un fichier k8s : cela permet de décrire une installation complexe en un seul fichier
 - par exemple le dashboard Kubernetes
<https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml>
- L'ordre n'importe pas car les ressources sont décrites déclarativement c'est-à-dire que:
 - Les dépendances entre les ressources sont déclarées
 - Le control plane de Kubernetes se charge de planifier l'ordre correct de création en fonction des dépendances (pods avant le déploiement, rôle avec l'utilisateur lié au rôle)
 - On préfère cependant les mettre dans un ordre logique pour que les humains puissent les lire.

- On peut sauter des lignes dans le YAML et rendre plus lisible les descriptions
- On sépare les différents objets par ...

Formation Kubernetes

TP - Déployer en utilisant des fichiers ressource yaml

Créer un pod rancher-demo à la main avec une description YAML [🔗](#)

Dans ce TP nous allons redéployer notre application `demonstration` du TP1 mais cette fois en utilisant `kubectl apply -f` et en visualisant le résultat dans Lens.

- Commencez par supprimer les ressources `demonstration` et `demonstration-service` du TP1
- Créez un dossier `TP2_deploy_using_files_and_Lens` sur le bureau de la machine distante et ouvrez le avec `vscode`.

Nous allons d'abord déployer notre application comme un simple **Pod** (non recommandé mais montré ici pour l'exercice).

- Créez un fichier `demo-pod.yaml` avec à l'intérieur le code d'exemple du cours "Objets Fondamentaux" partie "Pods".
- Appliquez le fichier avec `kubectl apply -f <fichier>`.
- Constatez dans Lens dans la partie pods que les deux conteneurs du pod sont bien démarrés (deux petits carrés vert à droite de la ligne du pod)
- Modifiez l'étiquette (`label`) du pod dans la description précédente et réappliquez la configuration. Kubernetes mets à jour le pod.
- Modifier le nom du conteneur `rancher-demo` (et pas du pod) et réappliquez la configuration. Que se passe-t-il ?

=> Kubernetes refuse d'appliquer le nouveau nom de conteneur car un pod est largement immutable. Pour changer d'une quelconque façon les conteneurs du pod il faut supprimer (`kubectl delete -f <fichier>`) et recréer le pod. Mais ce travail de mise à jour devrait être géré par un déploiement pour automatiser et pour garantir la haute disponibilité de notre application `demonstration`.

Kubernetes fournit un ensemble de commandes pour débugger des conteneurs :

- `kubectl logs <pod-name> -c <conteneur_name>` (le nom du conteneur est inutile si un seul). **Cependant** pour une consultation plus pratique des logs de conteneurs il est conseillé d'utiliser plutôt `stern -t <nomdu service qui pointe vers les pods>`
- `kubectl exec -it <pod-name> -c <conteneur_name> -- /bin/sh`
- Explorez le pod avec la commande `kubectl exec -it <pod-name> -c <conteneur_name> -- /bin/sh` écrite plus haut.
- Retrouvez les fonctions de shell et de log dans l'interface de OpenLens.
- Supprimez le pod.

Utiliser un déploiement (méthode à utiliser)

- Créez un fichier `demo-deploy.yaml` avec à l'intérieur le code suivant **A COMPLÉTER:**

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: demonstration
  labels:
    nom-app: demonstration
spec:
  selector:
    matchLabels:
      nom-app: demonstration
  strategy:
    type: Recreate
  replicas: 1
  template:
    metadata:
      labels:
        nom-app: demonstration
    spec:
      containers:
        - image: <image>
          name: <name>
          ports:
            - containerPort: <port>
              name: demo-http

```

- Appliquez ce nouvel objet avec kubectl.
- Inspectez le déploiement dans Lens.
- Changez le nom d'un conteneur et réappliquez: Cette fois le déploiement se charge créer un nouveau pod avec les bonnes caractéristiques et de supprimer l'ancien.
- Changez le nombre de réplicats.
- Que se passe-t-il si on change l'étiquette de `matchLabel` en se trompant de valeur ?

Ajoutons un service en mode NodePort pour visiter la demo [🔗](#)

- Créez un fichier `demo-svc.yaml` avec à l'intérieur le code suivant à compléter:

```

apiVersion: v1
kind: Service
metadata:
  name: demo-service
  labels:
    nom-app: demonstration
spec:
  ports:
    - port: <port>
  selector:
    nom-app: demonstration
  type: NodePort

```

- Appliquez ce nouvel objet avec kubectl.
- Inspectez le service dans Lens.
- Visitez votre application avec l'Internal ip du noeud (à trouver dans les information du node) et le nodeport (port 3xxxx) associé au service, le nombre de réplicat devrait apparaître.
- Pour tester, changez le label du selector dans le **service** (lignes `nom-app: demonstration` et partie: `les-petits-pods-demo` à remplacer dans le fichier) et réappliquez.
- Constatez que l'application n'est plus accessible dans le navigateur.
Pourquoi ?
- Allez voir la section endpoints dans lens, constatez que quand l'étiquette est la bonne la liste des ips des pods est présente et après la modification du selector la liste est vide (None)

=> Les services kubernetes redirigent le trafic basés sur les étiquettes(labels) appliquées sur les pods du cluster. Il faut donc aussi éviter d'utiliser deux fois le même label pour des parties différentes de l'application.

Correction

Le dépôt Git de la correction de ce TP est accessible ici : `git clone -b tp_rancher_demo_files https://github.com/Uptime-Formation/corrections_tp.git`

Formation Kubernetes

Cours - Objets Fondamentaux pour déployer une application

Les namespaces [🔗](#)

Tous les objets Kubernetes sont rangés dans différents espaces de travail isolés appelés [namespaces](#).

Cette isolation permet plusieurs choses :

- isoler les ressources pour éviter les conflits de nom, par exemple quand on veut déployer plusieurs fois la même application (le même code yaml) sans changer le nom des resources (comme ajouter un préfixe/sufixe).
- ne voir que ce qui concerne une tâche particulière (ne réfléchir que sur une seule chose lorsqu'on opère sur un cluster)
- créer des limites de ressources (CPU, RAM, etc.) pour le namespace
- définir des rôles et permissions sur le namespace qui s'appliquent à toutes les ressources à l'intérieur.

Lorsqu'on lit ou créé des objets sans préciser le namespace, ces objets sont liés au namespace `default`.

Pour utiliser un namespace autre que `default` avec `kubectl` il faut :

- le préciser avec l'option `-n` : `kubectl get pods -n kube-system`
- créer une nouvelle configuration dans la `kubeconfig` pour changer le namespace par défaut.

Kubernetes gère lui-même ses composants internes sous forme de pods et services.

- Si vous ne trouvez pas un objet et que votre cluster n'est pas trop rempli, essayez de lancer la commande kubectl avec l'option `-A` ou `--all-namespaces`

Les Pods

Un [Pod](#) est l'unité de base d'une application Kubernetes que vous déployez : un Pod est un groupe atomique de conteneurs, ce qui veut dire qu'il est garanti que ces conteneurs atterrissent sur le même noeud et seront toujours lancés ensemble et connectés.

Un Pod comprend en plus des conteneurs, des ressources de stockage, une IP réseau unique, et des options qui contrôlent comment les conteneurs doivent s'exécuter (ex: `restart policy`). Cette collection de conteneurs tournent ainsi dans le même environnement d'exécution mais les processus sont isolés.

Plus précisément ces conteneurs sont étroitement liés et qui partagent :

- des volumes communs
- la même interface réseau : la même IP, les mêmes noms de domaine internes
- les conteneurs peuvent se parler en IPC
- ont un nom différent et des logs différents
- ont des sondes (liveness/readiness probes) et des limites de ram et cpu différentes pour chaque conteneur

Chaque Pod est destiné à exécuter une instance unique d'un workload donné. Si vous désirez échelle votre workload, vous devez multiplier le nombre de Pods avec un déploiement.

Pour plus de détail sur la philosophie des pods, vous pouvez consulter [ce bon article](#).

Kubernetes fournit un ensemble de commandes pour débugger des conteneurs :

- `kubectl logs <pod-name> -c <conteneur_name>` (le nom du conteneur est inutile si un seul)
- `kubectl exec -it <pod-name> -c <conteneur_name> -- bash`
- `kubectl attach -it <pod-name>`

Enfin, pour debugger la sortie réseau d'un programme on peut rapidement forwarder un port depuis un pods vers l'extérieur du cluster :

- `kubectl port-forward <pod-name> <port_interne>:<port_externe>`
- C'est une commande de debug seulement : pour exposer correctement des processus k8s, il faut créer un service, par exemple avec `NodePort`.

Pour copier un fichier dans un pod on peut utiliser: `kubectl cp <pod-name>:</path/to/remote/file> </path/to/local/file>`

Pour monitorer rapidement les ressources consommées par un ensemble de processus il existe les commandes `kubectl top nodes` et `kubectl top pods`

Un manifeste de Pod [🔗](#)

rancher-demo-pod.yaml

```

apiVersion: v1
kind: Pod
metadata:
  name: rancher-demo-pod
  labels:
    app: rancher-demo
spec:
  containers:
    - image: monachus/rancher-demo:latest
      name: rancher-demo-container
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
    - image: redis
      name: redis-container
      ports:
        - containerPort: 6379
          name: http
          protocol: TCP

```

Rappel sur quelques concepts [🔗](#)

Haute disponibilité [🔗](#)

- Faire en sorte qu'un service ait un "uptime" élevé.

On veut que le service soit tout le temps accessible même lorsque certaines ressources manquent :

- elles tombent en panne
- elles sont sorties du service pour mise à jour, maintenance ou modification

Pour cela on doit avoir des ressources multiples...

- Plusieurs serveurs
- Plusieurs versions des données
- Plusieurs accès réseau

Il faut que les ressources disponibles prennent automatiquement le relais des ressources indisponibles. Pour cela on utilise en particulier:

- des “load balancers” : aiguillages réseau intelligents
- des “healthchecks” : une vérification de la santé des applications

Nous allons voir que Kubernetes intègre automatiquement les principes de load balancing et de healthcheck dans l’orchestration de conteneurs

Répartition de charge (load balancing) [🔗](#)

- Un load balancer : une sorte d’**“aiguillage” de trafic réseau**, typiquement HTTP(S) ou TCP.
- Un aiguillage **intelligent** qui se renseigne sur plusieurs critères avant de choisir la direction.

Cas d’usage :

- Éviter la surcharge : les requêtes sont réparties sur différents backends pour éviter de les saturer.

L’objectif est de permettre la haute disponibilité : on veut que notre service soit toujours disponible, même en période de panne/maintenance.

- Donc on va dupliquer chaque partie de notre service et mettre les différentes instances derrière un load balancer.
- Le load balancer va vérifier pour chaque backend s’il est disponible (**healthcheck**) avant de rediriger le trafic.
- Répartition géographique : en fonction de la provenance des requêtes on va rediriger vers un datacenter adapté (+ proche).

Healthchecks [🔗](#)

Fournir à l’application une façon d’indiquer qu’elle est disponible, c’est-à-dire :

- qu'elle est démarrée (*liveness*)
- qu'elle peut répondre aux requêtes (*readiness*).

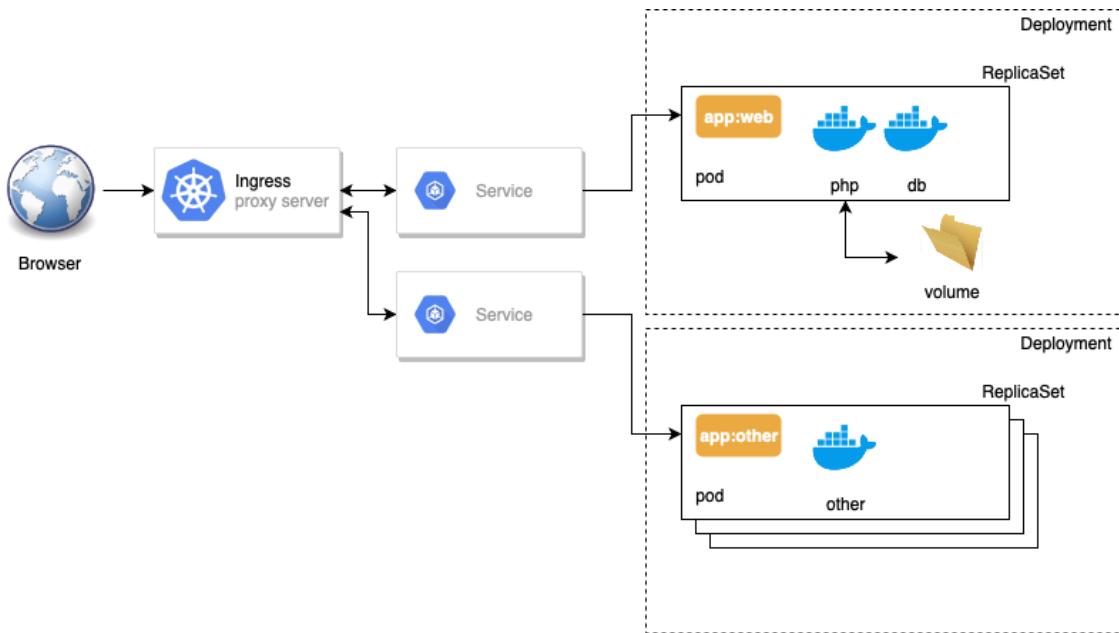
Application microservices

- Une application composée de nombreux petits services communiquant via le réseau. Le calcul pour répondre à une requête est décomposé en différentes parties distribuées entre les services.
Par exemple:
- un service est responsable de la gestion des **clients** et un autre de la gestion des **commandes**.
- Ce mode de développement implique souvent des architectures complexes pour être mis en œuvre et Kubernetes est pensé pour faciliter leur gestion à grande échelle.
- Imaginez devoir relancer manuellement des services vitaux pour une application en hébergeant des centaines d'instances : c'est en particulier à ce moment que Kubernetes devient indispensable.

2 exemples d'application microservices:

- <https://github.com/microservices-patterns/ftgo-application> -> fonctionne avec le très bon livre Microservices pattern visible sur le readme.
- <https://github.com/GoogleCloudPlatform/microservices-demo> -> Exemple d'application microservice de référence de Google pour Kubernetes.

L'architecture découpée des services Kubernetes



Comme nous l'avons vu dans le TP1, déployer une application dans Kubernetes demande plusieurs étapes. En réalité en plus des **pods** l'ensemble de la gestion d'un service applicatif se décompose dans Kubernetes en 3 à 4 objets articulés entre eux:

- **replicaset**
- **deployment**
- **service**
- **(ingress)**

Les Deployments (deploy) [🔗](#)

Les [déploiements](#) sont les objets effectivement créés manuellement lorsqu'on déploie une application. Ce sont des objets de plus haut niveau que les **pods** et **replicaset** et les pilote pour gérer un déploiement applicatif.

Deployment

Updates and Rollback

ReplicaSet

Self-healing, scalable, desired state

Pod



Pod



...

Pod



Les poupées russes Kubernetes : un Deployment contient un ReplicaSet, qui contient des Pods, qui contiennent des conteneurs

Si c'est nécessaire d'avoir ces trois types de ressources c'est parce que Kubernetes respecte un principe de découplage des responsabilités.

La responsabilité d'un déploiement est de gérer la coexistence et le **tracking de versions** multiples d'une application et d'effectuer des montées de version automatiques en haute disponibilité en suivant une **RolloutStrategy** (CF. TP optionnel).

Ainsi lors des changements de version, un seul **deployment** gère automatiquement deux **replicaset**s contenant chacun **une version** de l'application : le découplage est nécessaire.

Un *deployment* implique la création d'un ensemble de Pods désignés par une étiquette `label` et regroupé dans un **Replicaset**.

Exemple :

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80

```

- Pour les afficher : `kubectl get deployments`
- La commande `kubectl run` sert à créer un *deployment* à partir d'un modèle. Il vaut mieux utiliser `apply -f`.

Les ReplicaSets (rs) [🔗](#)

Dans notre modèle, les **ReplicaSet** servent à gérer et sont responsables pour:

- la réPLICATION (avoir le bon nombre d'instances et le scaling)
- la santé et le redémarrage automatique des pods de l'application (Self-Healing)
- `kubectl get rs` pour afficher la liste des replicas.

En général on ne les manipule pas directement (c'est déconseillé) même si il est possible de les modifier et de les créer avec un fichier de ressource.

Pour créer des groupes de conteneurs on utilise soit un [Deployment](#) soit d'autres formes de workloads ([DaemonSet](#), [StatefulSet](#), [Job](#)) adaptés à d'autres cas.

Les Services

Dans Kubernetes, un [service](#) est un objet qui :

- Désigne un ensemble de pods (grâce à des labels) généralement géré par un déploiement.
- Fournit un endpoint réseau pour les requêtes à destination de ces pods.
- Configure une politique permettant d'y accéder depuis l'intérieur ou l'extérieur du cluster.
- Configure un nom de domaine pointant sur le groupe de pods en backend.

L'ensemble des pods ciblés par un service est déterminé par un [selector](#).

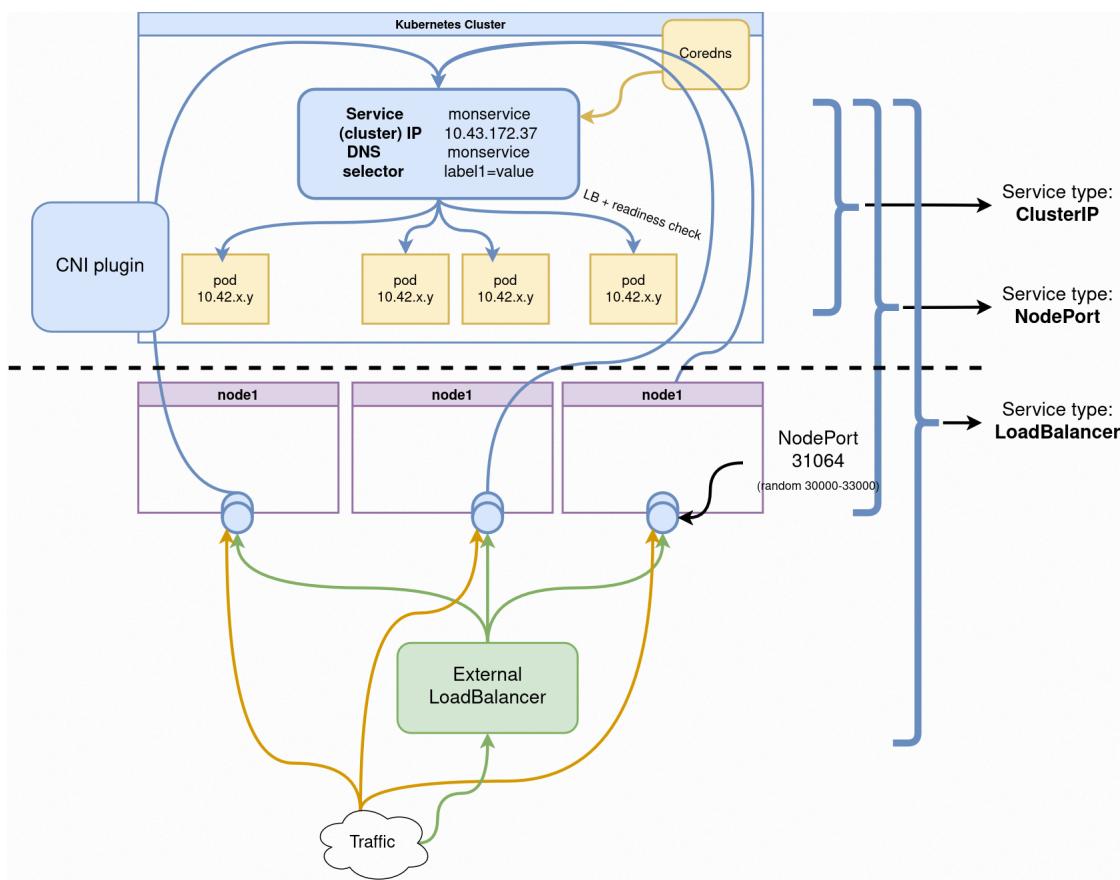
Par exemple, considérons un backend de traitement d'image (*stateless*, c'est-à-dire ici sans base de données) qui s'exécute avec 3 replicas. Ces replicas sont interchangeables et les frontends ne se soucient pas du backend qu'ils utilisent. Bien que les pods réels qui composent l'ensemble `backend` puissent changer, les clients frontends ne devraient pas avoir besoin de le savoir, pas plus qu'ils ne doivent suivre eux-mêmes l'état de l'ensemble des backends.

L'abstraction du service permet ce découplage : les clients frontend s'adressent à une seule IP avec un seul port dès qu'ils ont besoin d'avoir recours à un backend. Les backends vont recevoir la requête du frontend aléatoirement.

Les Services sont de trois types principaux :

- [clusterIP](#): expose le service **sur une IP interne** au cluster.

- [NodePort](#): expose le service depuis l'IP de **chacun des noeuds du cluster** en ouvrant un port directement sur le noeud, entre 30000 et 32767. Cela permet d'accéder aux pods internes répliqués. Comme l'IP est stable on peut faire pointer un DNS ou Loadbalancer classique dessus.
- [LoadBalancer](#): expose le service en externe à l'aide d'un Loadbalancer de fournisseur de cloud. Les services NodePort et ClusterIP, vers lesquels le Loadbalancer est dirigé sont automatiquement créés.



Deux autres types plus avancés:

- [ExternalName](#): Un service `ExternalName` est typiquement utilisé pour permettre l'accès à un service externe en le mappant à un nom DNS interne, facilitant ainsi la redirection des requêtes sans utiliser un proxy ou un load balancer.
(<https://stackoverflow.com/questions/54327697/kubernetes-externalname-services>)

- headless (avec `clusterIP: None`): est utilisé pour permettre la découverte directe des pods via leur ip au sein d'un service. Ce mode est souvent utilisé pour des applications nécessitant une connexion directe entre instances, comme les bases de données distribuées ou data broker (kafka). (<https://stackoverflow.com/questions/52707840/what-is-a-headless-service-what-does-it-do-accomplish-and-what-are-some-legiti>)

Quelques ressources plus détaillées:

<https://nigelpoulton.com/explained-kubernetes-service-ports/>

<https://nigelpoulton.com/demystifying-kubernetes-service-discovery/>

Formation Kubernetes

TP - Déployer une application multiconteneurs

Une application d'exemple en 3 parties

Récupérez le projet de base en clonant le dépôt global de la formation :

```
git clone https://github.com/e-lie/formation-kube-unified.git et en copier le dossier  
128_tp_monsterstack_base sur le bureau en tp3
```

Ce TP va consister à créer des objets Kubernetes pour déployer une application microservices (plutôt simple) : `monsterstack`. Elle est composée :

- d'un frontend en Flask (Python),
- d'un service de backend qui génère des images (un avatar de monstre correspondant à une chaîne de caractères)
- et d'un datastore `redis` servant de cache pour les images de l'application

Etudions le code et testons avec `docker-compose`

- Le frontend est une application web python (`flask`) qui propose un petit formulaire et lance une requête sur le backend pour chercher une image et l'afficher.
- Il est construit à partir du `Dockerfile` présent dans le dossier `tp3`.

- Le fichier `docker-compose.yml` est utile pour faire tourner les trois services de l'application dans docker rapidement (plus simple que kubernetes)

Pour lancer l'application il suffit d'exécuter par exemple: `docker compose up -d`

Passons maintenant à Kubernetes.

Déploiements pour le backend d'image (`imagebackend`) et le datastore `redis`

En vous inspirant du TP précédent créez des `deployments` pour `imagebackend` et `redis` sachant que:

- l'image docker pour `imagebackend` est `amouat/dnmonster:1.0` qui fonctionne sur le port 8080.
- l'image officielle `redis` est une image connue et bien documentée du docker hub (hub.docker.com). Lancez ici simplement l'image avec une configuration minimale comme dans le `docker-compose.yml`. Nous discuterons plus tard de comment déployer ce type d'application stateful de façon plus fiable et robuste.
- Combien de réplcats mettre pour l'`imagebackend` et le `redis` ?

> Correction

Déploiement du `frontend` manuellement

Comment déployer dans le cluster une image buildée en local, comme c'est le cas pour notre `frontend`? `docker build ...` ne rendra en effet pas directement l'image disponible dans le cluster car ce dernier n'a rien à voir avec le démon Docker présent sur notre machine.

La façon simple et manuelle de déployer notre image est de la pousser manuellement sur un registry (serveur d'images conteneur) par exemple Docker hub mais ça pourrait être gitlab, quay ou un registry auto-hébergé.

(cf TP développer dans Kubernetes pour de meilleures méthodes de déploiement)

- Créez un compte (gratuit et plutôt pratique) sur le Docker Hub si vous n'en avez pas encore.
- Buildez l'image `frontend` avec la ligne de commande `docker build -t frontend .` dans le dossier de projet. `docker image list` pour voir le résultat

Pour accéder à l'image dans le cluster nous allons la poussez sur le registry Docker Hub (une solution basique parmi plein d'autres) pour cela:

- Lancez la commande `docker login docker.io` et utilisez votre compte précédemment créé (ou simplement `docker login` car docker se loggue automatiquement à sa plateforme par défaut).
- Taguez l'image `frontend` avec `docker tag ...` avec un nouveau `repository:tag` précisant le serveur et l'utilisateur par exemple `docker tag frontend docker.io/myuser/frontend:1.0`
- Poussez l'image sur le registry avec `docker push <repository:tag>`
- Créez un déploiement pour le frontend en réutilisant ce `repository:tag` dans la section `image`.
- déployez avec `kubectl apply` pour tester.

➤ Correction:

Gérer la communication réseau de notre application avec des services

Les services K8s sont des endpoints réseaux qui envoient le trafic automatiquement vers un ensemble de pods désignés par certains labels. Ils sont un peu la pierre angulaire des applications microservices qui sont composées de plusieurs sous parties elles même répliquées.

Pour créer un objet `Service`, utilisons le code suivant, à compléter :

```

apiVersion: v1
kind: Service
metadata:
  name: <nom_service>
  labels:
    app: monsterstack
spec:
  ports:
    - port: <port> # port exposé en entrée par le service
      targetPort: <target-port> # port côté conteneur
  selector:
    app: <app_selector>
    partie: <tier_selector>
  type: <type>
---

```

Ajoutez le code précédent au début de chaque fichier déploiement.

Complétez pour chaque partie de notre application :

- le nom du service et le nom de la `partie` par le nom de notre programme (`frontend`, `imagebackend` et `redis`)
- le port par le port du service
- les selecteurs `app` et `partie` par ceux du groupe de pods correspondant.

Le type sera : `clusterIP` pour `imagebackend` et `redis`, car ce sont des services qui n'ont à être accédés qu'en interne, et `LoadBalancer` pour `frontend`.

- Appliquez à nouveau.
- Listez les services avec `kubectl get services`.
- Visitez votre application dans le navigateur avec `minikube service frontend`.

Exposer notre application à l'extérieur avec un `Ingress` (~ reverse proxy)

- Pour **Minikube** : Installons le contrôleur Ingress Nginx avec `minikube addons enable ingress`.

- Pour **k3s** : Installer l'ingress nginx avec la commande: `kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.1.0/deploy/static/provider/cloud/deploy.yaml` Puis vérifiez l'installation avec `kubectl get svc -n ingress-nginx ingress-nginx-controller` : le service `ingress-nginx-controller` devrait avoir une IP externe.
- Pour les autres types de cluster (**cloud** ou **manuel**), lire la documentation sur les prérequis pour les objets Ingress et installez l'ingress controller appelé `ingress-nginx` :
<https://kubernetes.io/docs/concepts/services-networking/ingress/#prerequisites>.
- Si vous êtes dans k3s, avant de continuer, vérifiez l'installation du contrôleur Ingress Nginx avec `kubectl get svc -n ingress-nginx ingress-nginx-controller` : le service `ingress-nginx-controller` devrait avoir une IP externe.

Utilisation de l'ingress 🔗

Un contrôleur Ingress (**Ingress controller**) est une implémentation de reverse proxy dynamique (car ciblant et s'adaptant directement aux objets services k8s) configurée pour s'interfacer avec un cluster k8s.

Une **ressource Ingress** est le fichier de configuration pour paramétriser le reverse proxy nativement dans Kubernetes.

- Repassez le service `frontend` en mode `clusterIP`. Le service n'est plus accessible sur un port. Nous allons utiliser l'ingress à la place pour afficher la page.
- Ajoutez également l'objet `Ingress` suivant dans le fichier `monsterstack-ingress.yaml` :

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: monsterstack
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    kubernetes.io/ingress.class: nginx
spec:
  rules:
    - host: monsterstack.local # à changer si envie/besoin
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend
                port:
                  number: 5000

```

- Récupérez l'ip de votre cluster (minikube avec `minikube ip`, pour k3 en allant observer l'objet `Ingress` dans `Lens` dans la section `Networking`. Sur cette ligne, récupérez l'ip).
- Ajoutez la ligne `<ip-cluster> monsterstack.local` au fichier `/etc/hosts` avec `sudo nano /etc/hosts` puis CRTL+S et CTRL+X pour sauver et quitter.
- Visitez la page `http://monsterstack.local` pour constater que notre Ingress (reverse proxy) est bien fonctionnel.

Paramétriser notre déploiement [🔗](#)

Configuration de l'application avec des variables d'environnement simples [🔗](#)

- Notre application frontend peut être configurée en mode DEV ou PROD. Pour cela elle attend une variable d'environnement `CONTEXT` pour lui indiquer si elle doit se lancer en mode `PROD` ou en mode `DEV`. Ici nous mettons l'environnement `DEV` en ajoutant (aligné à la hauteur du i de image):

```
env:  
- name: CONTEXT  
  value: DEV
```

- Généralement la valeur d'une variable est fournie au pod à l'aide d'une ressource de type `ConfigMap` OU `Secret` ce que nous verrons par la suite.
- Configurez de même les variables `IMAGEBACKEND_DOMAIN` et `REDIS_DOMAIN` comme dans le `docker-compose` et trouvez comment modifier les hostnames associés aux services.

| > Correction:

Quelques autres paramétrages...

- | > Facultatif: Santé du service avec les Probes
- | > Facultatif: Ajouter des indications de ressources

Correction du TP

La correction se trouve dans le dossier `130_tp_k8s_monsterstack` du dépôt de la formation (<https://github.com/e-lie/formation-kube-unified>)

Formation Kubernetes

TP - Développer directement dans un cluster Kubernetes

Quel workflow de développement ? [🔗](#)

Envoyer les images dans le cluster... [🔗](#)

Au moment de déployer la partie frontend de l'application, nous rencontrons un problème pratique : l'image à déployer est construite à partir d'un Dockerfile, elle est ensuite disponible dans le stock de nos images docker en local. Mais notre cluster (minikube ou autre) n'a pas accès par défaut à ces images : résultat nous pouvons bien construire et lancer l'image du frontend avec `docker build` et `docker run -p 5000:5000 frontend` mais nous ne pouvons pas utiliser cette image dans un fichier `deployment` avec `image: frontend` car le cluster ne saura pas où la trouver => `ErrImagePull`

Nous devons donc trouver comment envoyer l'image dans le cluster et idéalement de façon efficace.

- Une première méthode utilisée dans le TP précédent est de pousser l'image sur un registry par exemple DockerHub. Ensuite si nous faisons référence dans le déploiement à `<votre_hub_login>/frontend` ou `<adresse_registry>/frontend` le cluster devrait pouvoir télécharger l'image. Cette méthode est cependant trop lente pour le développement car il faudrait lancer plusieurs commandes à chaque modification du code ou des fichiers k8s.

Imaginez que vous développez une application microservice avec un dépôt de code contenant une quinzaine de conteneurs différents à builder et déployer séparément. Imaginez que vous vouliez pouvoir éditer

directement l'application, la déployer dans le cluster et pouvoir itérer rapidement sur le code logiciel et sur le code kubernetes. Il nous faut accélérer grandement la vitesse et le confort de déploiement.

- Une méthode déjà plus rapide est d'utiliser `minikube` et son intégration avec Docker tel qu'expliqué ici:
<https://minikube.sigs.k8s.io/docs/handbook/pushing/#1-pushing-directly-to-the-in-cluster-docker-daemon-docker-env>. Une fois la commande `eval $(minikube docker-env)` lancée les commandes type `docker build` construiront l'image directement dans le cluster. On peut même construire directement l'ensemble des images d'une stack avec `docker-compose build`. Inconvénients : toujours assez lent, manuel et spécifique à minikube/docker ou à chaque distribution k8s de dev.
- Une solution puissante et générique pour avoir un workflow développement confortable et compatible avec `minikube` mais aussi tout autre distribution kubernetes est `skaffold`! Combiné ici à un registry d'images, `skaffold` surveille automatiquement nos modifications de développement et reconstruira/redéploiera toutes les images à chaque fois en quelques secondes.

Méthode Minikube (facultatif)

- Lancez la commande `eval $(minikube docker-env)` qui va indiquer à la CLI Docker d'utiliser le daemon présent à l'intérieur du cluster minikube, notamment pour construire l'image.
- Lancez un build Docker `docker build -t frontend ..`. La construction va être effectuée directement dans le cluster.
- Vérifiez que l'image `frontend` est bien présente dans le cluster avec
`docker image list`

Déploiement du `frontend` via minikube

Ajoutez au fichier `frontend.yml` du dossier `k8s` le code suivant:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: monsterstack
spec:
  selector:
    matchLabels:
      app: monsterstack
      partie: frontend
  strategy:
    type: Recreate
  replicas: 3
  template:
    metadata:
      labels:
        app: monsterstack
        partie: frontend
    spec:
      containers:
        - name: frontend
          image: frontend
          imagePullPolicy: Never
          ports:
            - containerPort: 5000
```

- Appliquez ce fichier avec `kubectl` et vérifiez que le déploiement `frontend` est bien créé.

Méthode Skaffold

- Vérifiez que vous n'êtes pas dans l'environnement minikube docker-env avec `env | grep DOCKER` qui doit ne rien renvoyer.
- Installez `skaffold` en suivant les indications ici:
`https://skaffold.dev/docs/install/`
- Créez ou modifiez un fichier `skaffold.yaml` avec le contenu :

```

apiVersion: skaffold/v1
kind: Config
build:
  artifacts:
    - image: docker.io/<votre_login>/frontend # change with your registry and log
      to it with docker login
  deploy:
    kubectl:
      manifests:
        - k8s/*.yaml

```

- Identifiez-vous sur le registry avec `docker login <registry>`.
- Changez dans le fichier `frontend.yaml` du dossier `k8s` le paramètre `image` comme suit:

```

...
spec:
  containers:
    - name: frontend
      image: docker.io/<votre_login>/frontend
      ports:
        - containerPort: 5000
      # imagePullSecrets:
      #   - name: registry-credential

```

- Lancez le build, push, et le déploiement du imagebackend et du redis avec `skaffold run` et `skaffold delete` pour nettoyer.
- Le mode développement de skaffold est lié à la commande `skaffold dev` qui en plus de tout builder et déployer s'assure de surveiller (`watch`) toutes les modifications apportées aux fichiers de code logiciel / Dockerfile et code Kubernetes, etc...

Enfin une solution efficace pour développer

Registry privé avec login simple [🔗](#)

- Dans le cas d'un registry privé, pour pouvoir tirer l'image il faut:

- ajouter le login sous forme d'un secret dans Kubernetes. Par exemple : `kubectl create secret docker-registry registry-credential --docker-server=registry.kluster.ptych.net --docker-username=elie --docker-password=<thepassword>`
- ajouter ensuite à la spec du pod une section :

```
imagePullSecrets:
  - name: registry-credential
```

...cela permettra que Kubernetes connaissent le login pour se connecter au registry et télécharger l'image.

Skaffold avec Helm ou jsonnet etc. [🔗](#)

`skaffold` est un petit binaire couteau suisse et peut s'intégrer à la majorité des contextes de dev et déploiement kubernetes:

- compatible avec la plupart des méthodes de déploiement d'app Kustomize, Jsonnet, ArgoCD, ...
- incontournable pour le développement Helm
- utilisable dans des pipelines CI/CD

Pour l'usage avec Helm, voir le TP Helm.

Correction du TP [🔗](#)

La correction se trouve dans le dossier `132_tp_k8s_dev_env` du dépôt de la formation (<https://github.com/e-lie/formation-kube-unified>)

Formation Kubernetes

Cours - Les volumes pour le stockage persistant et la configuration

Le stockage et les Volumes dans Docker [🔗](#)

Les conteneurs propose un paradigme immutable : on peut les transformer pendant leur execution (ajouter des fichier, changer des configurations) mais ce n'est pas le mode d'utilisation recommandé. En particulier Kubernetes est susceptible de les supprimer et de les recréer automatiquement. Les fichiers ajoutés manuellement pendant l'execution seront alors perdu.

Se pose donc la question de la persistance des données d'une application, par exemple une base de donnée. Dans un environnement conteneurisé toute persistance est permise via des volumes, sortes de disques durs virtuels, qu'on connecte à nos conteneur. Comme un disque ces volumes sont monté à un emplacement du système de fichier du conteneur. En écrivant dans le dossier en question on écrit alors sur ce disque virtuel qui conservera ses données même si le conteneur est supprimé.

Le stockage dans Kubernetes [🔗](#)

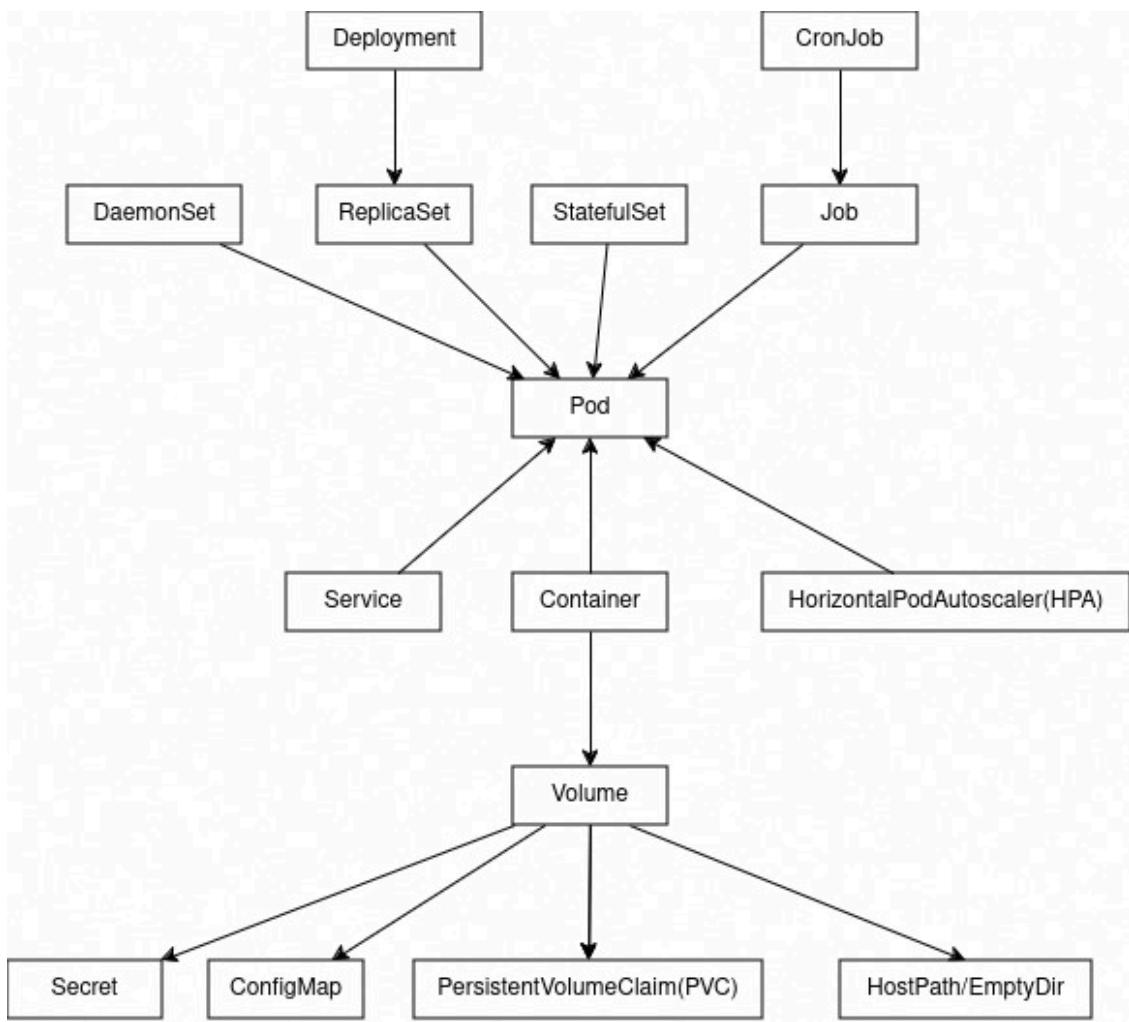
Les Volumes Kubernetes [🔗](#)

Comme dans Docker, Kubernetes fournit la possibilité de monter des volumes virtuels dans les conteneurs de nos pod. On liste séparément les

volumes de notre pod puis on les monte une ou plusieurs dans les différents conteneurs. Exemple:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
    - image: k8s.gcr.io/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /test-pd
          name: test-volume
  volumes:
    - name: test-volume
      hostPath:
        # chemin du dossier sur l'hôte
        path: /data
        # ce champ est optionnel
      type: Directory
```

La problématique des volumes et du stockage est plus compliquée dans kubernetes que dans docker car k8s cherche à répondre à de nombreux cas d'usages. [doc officielle](#). Il y a donc de nombreux types de volumes kubernetes correspondants à des usages de base et aux solutions proposées par les principaux fournisseurs de cloud.



Volumes pour la persistence

Demander des volumes et les liers aux pods

:`PersistentVolumes` et `PersistentVolumeClaims` 

Quand un conteneur a besoin d'un volume, il crée une `PersistentVolumeClaim` : une demande de volume (persistant). Si une des `StorageClass` du cluster est en capacité de le fournir, alors un `PersistentVolume` est créé et lié à ce conteneur : il devient disponible en tant que volume monté dans le conteneur.

- les conteneurs demandent du volume avec les `PersistentVolumeClaims`
- les `StorageClasses` répondent aux `PersistentVolumeClaims` en créant des objets `PersistentVolumes` : le conteneur peut accéder à son volume.

doc officielle

Le provisioning de `PersistentVolume` peut être manuel (on crée un objet `PersistentVolume` en amont ou non. Dans le second cas la création d'un `PersistentVolumeClaim` mène directement à la création d'un volume si possible)

Liens externes

- <https://developers.redhat.com/articles/2022/10/06/kubernetes-storage-concepts>
- <https://bluexp.netapp.com/blog/cvo-blg-5-types-of-kubernetes-volumes-and-how-to-work-with-them>

Backup de volume

Il existe plusieurs méthodes pour effectuer des sauvegardes de données persistantes dans Kubernetes :

- Utiliser des outils de backup Kubernetes : Certains outils de sauvegarde Kubernetes, tels que Velero, des CSI plugin comme longhorn, des opérateurs de BDD comme CloudNativePG, permettent de sauvegarder et de restaurer des données persistantes. Ces outils peuvent être configurés pour effectuer des sauvegardes régulières des volumes persistants (backup physique) ou de vos objets contenant des données (Backup logique) dans votre cluster Kubernetes, puis les stocker dans un emplacement de stockage sécurisé.
- Utiliser des outils classiques de backups planifiés depuis vos pods (push depuis le pod): Les volumes persistants Kubernetes sont généralement montés en tant que systèmes de fichiers dans les pods. En utilisant des outils de sauvegarde de fichiers tels que rsync, borg, etc. on peut sauvegarder ces volumes persistants sur des emplacements de stockage externes. On peut également utiliser des scripts basés sur kubectl planifiés depuis un serveur de backup se connectent à l'intérieur des pods pour récupérer les données.

Objets de configuration

Les ConfigMaps

D'après les recommandations de développement [12factor](#), la configuration de nos programmes doit venir de l'environnement.

Les objets ConfigMaps permettent d'injecter dans des pods des ensemble clés/valeur de configuration en tant que volumes/fichiers de configuration ou variables d'environnement.

Cela permet notamment de centraliser et découpler la configuration du déploiement des pods. Par exemple on peut stocker de façon centraliser le nom de domaine à utiliser pour une application et plusieurs de ses microservices pourront venir la récupérer dans la même configmap.

Exemple de configmap et de récupération d'une variable d'environnement

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-config
data:
  MYSQL_DATABASE: mydatabase
```

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: mysql:5.7
          env:
            - name: MYSQL_DATABASE
              valueFrom:
                configMapKeyRef:
                  name: mysql-config
                  key: MYSQL_DATABASE
          ports:
            - containerPort: 3306

```

les Secrets [🔗](#)

Les Secrets se manipulent comme des objets ConfigMaps, mais ils sont chiffrés et faits pour stocker des mots de passe, des clés privées, des certificats, des tokens, ou tout autre élément de config dont la confidentialité doit être préservée. Un secret se crée avec l'API Kubernetes, puis c'est au pod de demander à y avoir accès.

Il y a plusieurs façons de donner un accès à un secret, notamment :

- le secret est un fichier que l'on monte en tant que volume dans un conteneur (pas nécessairement disponible à l'ensemble du pod). Il est possible de ne jamais écrire ce secret sur le disque (volume `tmpfs`).
- le secret est une variable d'environnement du conteneur.

Pour définir qui et quelle app a accès à quel secret, on peut utiliser les fonctionnalités "RBAC" de Kubernetes.

Exemple de secret pour un certificat SSL et son montage comme fichier dans un pods [🔗](#)

Création du secret en ligne de commande à partir d'un fichier:

```
kubectl create secret generic my-cert --from-file=mycert.pem
```

Cela donne par exemple le secret suivant (les données d'un secret sont encodé en base64, un mode de sérialisation qui permet notamment de stocker des données binaires sous forme de texte):

```
apiVersion: v1
data:
  mycert.pem:
    LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURhekNDQWxPZ0F3SUJBZ0lVRXZsVXVyT3RTelN0cFlwA
kind: Secret
metadata:
  creationTimestamp: "2023-03-08T16:29:52Z"
  name: my-cert
  namespace: default
  resourceVersion: "5543"
  uid: dd27cd3f-1779-47f4-a821-25d8139b21a4
type: Opaque
```

On peut ensuite monter le secret sous forme d'un fichier dans des pods via un volume comme suit :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: mycontainer
          image: myimage
          ports:
            - containerPort: 443
          volumeMounts:
            - name: my-cert
              mountPath: /etc/mycert.pem
              readOnly: true
      volumes:
        - name: my-cert
          secret:
            secretName: my-cert
```

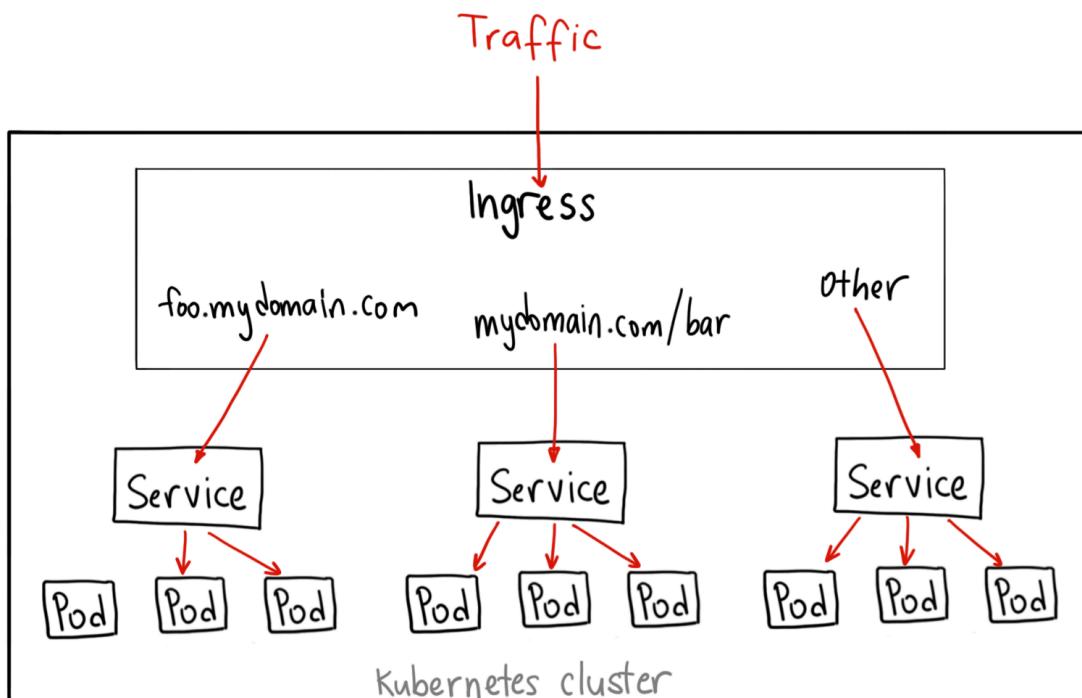
Formation Kubernetes

Cours - Le réseau dans Kubernetes

Les services [🔗](#)

Les services sont les objets réseau de base. Voir cours précédent sur les objets fondamentaux.

Les objets Ingresses [🔗](#)



Crédits [Ahmet Alp Balkan](#)

Un Ingress est un objet pour gérer dynamiquement le **reverse proxy** HTTP/HTTPS dans Kubernetes. Documentation:

<https://kubernetes.io/docs/concepts/services-networking/ingress/#what-is-ingress>

Exemple de syntaxe d'un ingress:

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
    - host: "domain1.bar.com"
      http:
        paths:
          - pathType: Prefix
            path: "/bar"
            backend:
              service:
                name: service1
                port:
                  number: 80
          - pathType: Prefix
            path: "/foo"
            backend:
              service:
                name: service2
                port:
                  number: 80
    - host: "domain2.foo.com"
      http:
        paths:
          - pathType: Prefix
            path: "/"
            backend:
              service:
                name: service3
                port:
                  number: 80

```

Pour pouvoir créer des objets ingress il est d'abord nécessaire d'installer un **ingress controller** dans le cluster:

- Il s'agit d'un déploiement conteneurisé d'un logiciel de reverse proxy (comme nginx) et intégré avec l'API de kubernetes
- Le contrôleur agit donc au niveau du protocole HTTP et doit lui-même être exposé (port 80 et 443) à l'extérieur, généralement via un service de type LoadBalancer.

- Le contrôleur redirige ensuite vers différents services (généralement configurés en ClusterIP) qui à leur tour redirigent vers différents ports sur les pods selon l'URL de la requête.

Il existe plusieurs variantes d'**ingress controller**:

- Un ingress basé sur Nginx plus ou moins officiel à Kubernetes et très utilisé: <https://kubernetes.github.io/ingress-nginx/>
- Un ingress Traefik optimisé pour k8s.
- Il en existe d'autres : celui de payant l'entreprise Nginx, Contour, HAProxy...

Chaque provider de cloud et flavour de Kubernetes est légèrement différent au niveau de la configuration du contrôleur ce qui peut être déroutant au départ:

- minikube permet d'activer l'ingress nginx simplement (voir TP)
- autre exemple: k3s est fourni avec traefik configuré par défaut
- On peut installer plusieurs ingress controllers correspondant à plusieurs IngressClasses

Comparaison des contrôleurs: <https://medium.com/flant-com/comparing-ingress-controllers-for-kubernetes-9b397483b46b>

La nouvelle API Gateway

- <https://gateway-api.sigs.k8s.io/>

Gestion dynamique des certificats à l'aide de certmanager

certmanager est une application Kubernetes (un operator) capable de générer automatiquement des certificats TLS/HTTPS pour nos ingresses.

- Documentation d'installation: <https://cert-manager.io/docs/installation/kubernetes/>
- Tutorial pas à pas pour générer un certificat automatiquement avec un ingress et letsencrypt: <https://cert-manager.io/docs/tutorials/acme/ingress/>

Exemple de syntaxe d'un ingress utilisant certmanager:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: kuard
  annotations:
    kubernetes.io/ingress.class: "nginx"
    cert-manager.io/issuer: "letsencrypt-prod"
spec:
  tls:
    - hosts:
        - example.example.com
      secretName: quickstart-example-tls
    rules:
      - host: example.example.com
        http:
          paths:
            - path: /
              pathType: Exact
              backend:
                service:
                  name: kuard
                  port:
                    number: 80
```

Architecture réseau Kubernetes [🔗](#)

Réseau standard de Kubernetes [🔗](#)

La configuration réseau standard pour Kubernetes implique l'utilisation de Flannel comme CNI plugin (solution de réseau virtuel compatible Container Network Interface) et de Kube-proxy en mode iptables (configuration par défaut de k3s par exemple mais aussi la configuration la plus simple avec kubeadm etc).

Dans cette configuration, Flannel est responsable de la mise en place d'un réseau virtuel (on parle de network fabric) qui permet aux pods de communiquer entre eux sur différents nœuds dans le cluster. Flannel assigne une adresse IP unique à chaque pod et crée un tunnel VXLAN ou UDP entre les nœuds pour permettre la communication entre les pods.

Kube-proxy, configuré en mode iptables, utilise l'outil iptables (de filtrage de paquet dans le noyaux Linux) pour gérer le trafic réseau dans le cluster. Il crée des règles iptables pour faire suivre le trafic vers les endpoints des pods ou des services appropriés en fonction de leurs adresses IP et des ports. Kube-proxy maintient également une table NAT pour gérer le trafic entrant vers le cluster.

CNI (container network interface) : Les implémentations du réseau Kubernetes [🔗](#)

Beaucoup de solutions de réseau qui se concurrencent, demandant un comparatif un peu fastidieux.

- plusieurs solutions toutes robustes
- diffèrent sur l'implémentation : BGP, réseau overlay ou non (encapsulation VXLAN, IPinIP, autre)
- toutes ne permettent pas d'appliquer des **NetworkPolicies** : l'isolement et la sécurité réseau
- peuvent parfois s'hybrider entre elles (Canal = Calico + Flannel)
- Calico, Flannel, Weave ou Cilium sont très employées et souvent proposées en option par les fournisseurs de cloud
- Flannel est simple et éprouvé mais sans network policies ou observabilité avancée

- Cilium a la particularité d'utiliser la technologie eBPF de Linux qui permet une sécurité et une rapidité accrue

Comparaisons :

- <https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/>

Vidéos

Quelques vidéos assez complète sur le réseau :

- [Kubernetes Ingress networking](#)
- [Kubernetes Services networking](#)
- Vidéo sur le fonctionnement détaillé du réseau d'un pod :
<https://www.youtube.com/watch?v=5cNrTU6o3Fw>
- Vidéo sur le fonctionnement détaillé des services (pas du réseau sous jacent comme la précédente mais des objets) :
<https://www.youtube.com/watch?v=T4Z7visMM4E>

Formation Kubernetes

TP - Ajouter une persistance à Redis avec un volume

Persistir les données de Redis [🔗](#)

Actuellement le Redis de notre application ne persiste aucune donnée. On peut par exemple constater que le compteur de visite de la page est réinitialisé à chaque réinstallation.

Nous allons maintenant utiliser un volume pour résoudre simplement ce problème.

- Clonez le cas échéant la correction du TP précédent avec `git clone -b tp_monsterstack_final https://github.com/Uptime-Formation/corrections_tp.git tp_monsterstack_correction1.`

En vous inspirant du code du tutoriel officiel suivant

(<https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/>) :

- Créez un nouveau fichier `redis-data-pvc.yaml` contenant un `PersistentVolumeClaim` de nom `redis-data` de 2Go.
- Ajoutez au déploiement/pod-template de redis une section `volumes` avec un volume correspondant au pvc
- Ajoutez un mount au conteneur redis au niveau du chemin `/data`

Installons notre application avec `kubectl` et testons en visitant la page

Observer la persistence [🔗](#)

- Supprimez uniquement les trois déploiements.
- Redéployez à nouveau avec `kubectl apply -f ..`, les deux déploiements sont recréés.
- En rechargeant le site on constate que les données ont été conservées (nombre de visite conservé).
- Allez observer la section stockage dans `Lens`. Commentons ensemble.
- Supprimer tout avec `kubectl delete -f ..`. Que s'est-il passé ? (côté storage)

En l'état les `PersistentVolumes` générés par la combinaison du `PersistentVolumeClaim` et de la `StorageClass` de minikube sont également supprimés en même temps que les PVCs. Les données sont donc perdues et au chargement du site le nombre de visites retombe à 0.

Pour éviter cela il faut avec une `Reclaim Policy` à `retain` (conserver) et non `delete` comme suit <https://kubernetes.io/docs/tasks/administer-cluster/change-pv-reclaim-policy/>. Les volumes sont alors conservés et les données peuvent être récupérées manuellement. Mais les volumes ne peuvent pas être reconnectés à des PVCs automatiquement.

Pour récupérer les données on peut:

- récupérer les données à la main sur le disque/volume réseau indépendamment de kubernetes
- utiliser la nouvelle fonctionnalité de clone de volume

Correction du TP

La correction se trouve dans le dossier `146_tp_k8s_monsterstack_redis_pvc` du dépôt de la formation (<https://github.com/e-lie/formation-kube-unified>)

Formation Kubernetes

TP - Ajouter une Configuration dynamique ConfigMap

Gérer la configuration avec des ConfigMaps [🔗](#)

Actuellement notre application frontend utilise des variables d'environnement codées en dur directement dans le fichier de déploiement. Cette approche n'est pas idéale car elle mélange la configuration applicative avec la définition de l'infrastructure.

Nous allons maintenant externaliser cette configuration dans une **ConfigMap** Kubernetes, ce qui permettra de :

- Modifier la configuration sans toucher aux fichiers de déploiement
- Réutiliser la même configuration pour plusieurs déploiements
- Gérer différentes configurations pour différents environnements (dev, prod)

État actuel [🔗](#)

Dans le fichier `frontend.yaml`, nous avons actuellement trois variables d'environnement définies directement :

```
env:
  - name: CONTEXT
    value: PROD
  - name: REDIS_DOMAIN
    value: redis
  - name: IMAGEBACKEND_DOMAIN
    value: imagebackend
```

Créer une ConfigMap [🔗](#)

Une **ConfigMap** est un objet Kubernetes qui permet de stocker des données de configuration sous forme de paires clé-valeur.

- Créez un nouveau fichier `frontend-config.yaml` dans le dossier `k8s` avec le contenu suivant :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: frontend-config
data:
  CONTEXT: "DEV"
  IMAGEBACKEND_DOMAIN: "imagebackend"
  REDIS_DOMAIN: "redis"
```

Modifier le déploiement pour utiliser la ConfigMap [🔗](#)

Maintenant que nous avons créé notre ConfigMap, nous devons modifier le déploiement `frontend` pour qu'il utilise les valeurs de cette ConfigMap au lieu des valeurs codées en dur.

- Modifiez la section `env` dans le fichier `frontend.yaml` comme suit :

```

env:
  - name: CONTEXT
    valueFrom:
      configMapKeyRef:
        name: frontend-config          # Le nom de la ConfigMap
        key: CONTEXT                  # La clé dans la ConfigMap
  - name: REDIS_DOMAIN
    valueFrom:
      configMapKeyRef:
        name: frontend-config
        key: REDIS_DOMAIN
  - name: IMAGEBACKEND_DOMAIN
    valueFrom:
      configMapKeyRef:
        name: frontend-config
        key: IMAGEBACKEND_DOMAIN

```

- `valueFrom`: Indique que la valeur provient d'une source externe (ici une ConfigMap)
- `configMapKeyRef`: Référence une clé spécifique dans une ConfigMap
- `name`: Le nom de la ConfigMap (doit correspondre au `metadata.name` de la ConfigMap)
- `key`: La clé dans la section `data` de la ConfigMap

Déployer et tester

- Déployez la ConfigMap et le déploiement modifié :

```

● ● ●

kubectl apply -f k8s/frontend-config.yaml
kubectl apply -f k8s/frontend.yaml

```

- Vérifiez que la ConfigMap a été créée :

```
kubectl get configmap frontend-config  
kubectl describe configmap frontend-config
```

- Vérifiez que les pods frontend ont bien redémarré avec la nouvelle configuration :

```
kubectl get pods -l partie=frontend
```

- Testez l'application pour vérifier qu'elle fonctionne toujours correctement.

Avantages de cette approche [🔗](#)

- **Séparation des “préoccupations”** : La configuration est séparée du code de déploiement donc on peut changer `CONTEXT` de `DEV` à `PROD` sans modifier le déploiement
- **Réutilisabilité** : La même ConfigMap peut être utilisée par plusieurs déploiements
- **Versionning** : Les ConfigMaps peuvent être versionnées dans Git
- **Gestion multi-environnements** : Créez différentes ConfigMaps pour dev, staging, prod

Pour aller plus loin (facultatif) [🔗](#)

Vous pouvez aussi injecter toute la ConfigMap comme variables d'environnement d'un coup avec `envFrom` :

```
envFrom:  
- configMapRef:  
  name: frontend-config
```

Cette approche injecte automatiquement toutes les clés de la ConfigMap comme variables d'environnement.

Documentation officielle :

<https://kubernetes.io/docs/concepts/configuration/configmap/>

Correction du TP

La correction se trouve dans le dossier `146_tp_k8s_monsterstack_redis_pvc` du dépôt de la formation (<https://github.com/e-lie/formation-kube-unified>)

Formation Kubernetes

Cours 6 - Méthodes pour installer des applications Kubernetes

Kustomize

L'outil `kustomize` sert à paramétrer et faire varier la configuration d'une installation Kubernetes en fonction des cas.

- Intégré directement dans `kubectl` depuis quelques années il s'agit de la façon la plus simple et respectueuse de la philosophie déclarative de Kubernetes de le faire.

Par exemple lorsqu'on a besoin de déployer une même application dans 3 environnements de `dev`, `prod` et `staging` il serait dommage de ne pas factoriser le code. On écrit alors une version de base des manifestes kubernetes commune aux différents environnements puis on utilise `kustomize` pour appliquer des patches sur les valeurs.

Plus généralement cet outil rassemble plein de fonctionnalité pour supporter les variations de manifestes :

- ajout de préfixes ou suffixes aux noms de resources
- mise à jour de l'image et sa version utilisée pour les pods
- génération de secrets et autres configurations
- etc.

Documentation : <https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/>

Kustomize est très adapté pour une variabilité pas trop importante des installations d'une application, par exemple une entreprise qui voudrait

déployer son application dans quelques environnements internes avec un dispositif de Continuous Delivery. Il a l'avantage de garder le code de base lisible et maintenable et d'éviter les manipulations impératives/séquentielles.

- Pour utiliser kustomise on écrit un fichier `kustomization.yaml` à côté des manifestes et patchs et on l'applique avec `kubectl -k chemin_vers_kustomization`.
- Il est aussi très utile de pouvoir visualiser le résultat du patching avant de l'appliquer avec : `kubectl kustomize chemin_vers_kustomization`

Mais lorsqu'on a besoin de faire varier énormément les manifestes selon de nombreux cas, par exemple lorsqu'on distribue une application publiquement et qu'on veut permettre à l'utilisateur de configurer dynamiquement à peu près tous les aspects d'une installation, kustomize n'est pas adapté.

Helm, package manager pour Kubernetes [🔗](#)

Helm permet de déployer des applications / stacks complètes en utilisant un système de templating pour générer dynamiquement les manifestes kubernetes et les appliquer intelligemment.

C'est en quelque sorte le package manager le plus utilisé par Kubernetes.

- Un package Helm est appelé **Chart**.
- Une installation particulière d'un chart est appelée **Release**.

Helm peut également gérer les dépendances d'une application en installant automatiquement d'autres chart liés et effectuer les mises à jour d'une installation précautionneusement s'il le **Chart** a été prévu pour.

En effet en plus de templater et appliquer les manifestes kubernetes, Helm peut exécuter des hooks, c'est à dire des actions personnalisées avant ou après l'installation, la mise à jour et la suppression d'un paquet.

Il existe des *stores* de charts Helm, le plus conséquent d'entre eux est <https://artifacthub.io>.

Observons un exemple de Chart :

<https://artifacthub.io/packages/helm/minecraft-server-charts/minecraft>

Un des aspects les plus visibles côté utilisateur d'un chart est la liste, souvent très étendue, des paramètres d'installation du chart. Il s'agit d'un dictionnaire YAML de paramètres sur plusieurs niveaux. Ils ont presque tous une valeur par défaut qui peut être surchargée à l'installation.

Plutôt que d'installer un chart à l'aveugle il est préférable d'effectuer un templating/dry-run du chart avec un ensemble de paramètre pour étudier les ressources Kubernetes qui seront créées à son installation: voir dans la suite et le TP. (ou d'utiliser un outil de déploiement et supervision d'applications comme ArgoCD)

Quelques commandes Helm: [🔗](#)

Voici quelques commandes de bases pour Helm :

- `helm repo add bitnami https://charts.bitnami.com/bitnami`: ajouter un repo contenant des charts
- `helm search repo bitnami`: rechercher un chart en particulier
- `helm install my-release my-chart --values=myvalues.yaml`: permet d'installer le chart my-chart avec le nom my-release et les valeurs de variable contenues dans myvalues.yaml (elles écrasent les variables par défaut)
- `helm upgrade my-release my-chart`: permet de mettre à jour notre release avec une nouvelle version.
- `helm plugin install https://github.com/databus23/helm-diff` pour télécharger le plugin helm diff important avant de lancer un upgrade
- **Ensuite** `helm diff upgrade my-release mychart --values values.yaml`

- `helm list`: Permet de lister les Charts installés sur votre Cluster

Pour lister les resources d'une release helm n'a pas de fonction préconçue : il faut bricoler un peu:

- `helm get manifest release-name | yq '(.kind + "/" + .metadata.name)'`
- `kubectl get all --all-namespaces -l='app.kubernetes.io/managed-by=Helm,app.kubernetes.io/instance=release-name'`
- `kubectl api-resources --verbs=list -o name | xargs -n 1 kubectl get --show-kind -l release=awesome-nginx --ignore-not-found -o name`
- `helm delete my-release`: Permet de désinstaller la release `my-release` de Kubernetes

Les Operateurs et les Custom Resources Definitions (CRD) [🔗](#)

Kubernetes étant modulaire et ouvert, en particulier son API et ses processus de contrôle il est possible et même encouragé d'étendre son fonctionnement depuis l'intérieur en respectant ses principes natifs de conception:

- Exprimer les objets/resources à manipuler avec des descriptions de haut niveau sous forme de manifestes YAML fournies à l'API
- Gérer ces resources à l'aide de boucles de contrôle/réconciliation qui vont s'assurer automatiquement de maintenir l'état désiré exprimé dans les descriptions.

Un opérateur désigne toute extension de Kubernetes qui respecte ces principes.

Les Custom Resources Definitions (CRDs) sont les nouveaux types de resources ajoutés pour étendre l'API

- On peut lister toutes les resources (custom ou non) dans kubectl avec `kubectl api-resources -o wide`. les CRDs sont aussi affichées dans la

dernière section du menu Lens.

- On peut utiliser `kubectl explain` sur ces noms de resources pour découvrir les types qu'on ne connaît pas

doc: <https://kubernetes.io/docs/concepts/extend-kubernetes/operator/>

Quelques exemples d'opérateurs:

- L'application `Certmanager` qui permet de générer et manipuler les certificats x509/TLS comme des resources Kubernetes
- L'opérateur de déploiement et supervision d'application `ArgoCD`
- L'ingress `Traefik` ou le service mesh `Istio` qui proposent des fonctionnalités réseaux avancés exprimées avec des resources custom.
- L'opérateur `Prometheus` permet d'automatiser le monitoring d'un cluster et ses opérations de maintenance.
- la chart officielle de la suite `Elastic (ELK)` définit des objets de type `elasticsearch`
- `KubeVirt` permet de rajouter des objets de type VM pour les piloter depuis Kubernetes
- Azure propose des objets correspondant à ses ressources du cloud Azure, pour pouvoir créer et paramétrier des ressources Azure directement via la logique de Kubernetes.

Les opérateurs sont souvent répertoriés sur le site: <https://operatorhub.io/>



Avec les opérateurs il est possible d'ajouter des nouvelles fonctionnalités quasi-natives à notre Cluster. Ce mode d'extensibilité est un des points qui fait la force et la relative universalité de Kubernetes.

Écrire un opérateur

Plus concrètement un opérateur est:

- un morceau de logique opérationnelle de votre infrastructure (par exemple: la mise à jour votre logiciel de base de donnée stateful comme cassandra ou elasticsearch) ...
- ... implémentée dans kubernetes par un/plusieurs conteneur(s) "controller" ...
- ... contrôlé grâce à une extension de l'API Kubernetes sous forme de nouveaux type d'objets kubernetes personnalisés (de haut niveau) appelés *CustomResourcesDefinition* ...
- ... qui manipule d'autre resources Kubernetes.

L'écriture d'opérateurs est un sujet avancé mais très intéressant de Kubernetes.

- Ils peuvent être développés avec un framework Go ou Ansible

Il est important de comprendre que le développement et la maintenance d'un opérateur est une tâche très lourde. Elle est probablement superflue pour la plupart des cas. Écrire un chart fait principalement sens pour une entreprise ou un fournisseur de solution qui voudrait optimiser un morceau de logique opérationnelle crucial et éventuellement vendre cette nouvelle solution a de nombreux clients.

Voir : <https://thenewstack.io/kubernetes-when-to-use-and-when-to-avoid-the-operator-pattern/>

Formation Kubernetes

TP - Déployer Wordpress avec Helm

Helm est un “gestionnaire de paquet” ou vu autrement un “outil de templating avancé” pour k8s qui permet d’installer des applications plus complexe de façon paramétrable :

- Pas de duplication de code
- Possibilité de créer du code générique et flexible avec pleins de paramètres pour le déploiement.
- Des déploiements avancés avec plusieurs étapes

Inconvénient: Helm ajoute souvent de la complexité non nécessaire car les Charts sur internet sont très paramétrables pour de multiples cas d’usage (plein de code qui n’est utile que dans des situations spécifiques).

Helm ne dispense pas de maîtriser l’administration de son cluster.

Installer Helm [🔗](#)

Helm est un binaire installable de nombreuse façons :

<https://helm.sh/docs/intro/install/>

Nous l’avons installé automatiquement avec asdf et les script dans `/opt`

Utiliser un chart Helm pour installer Wordpress [🔗](#)

- Cherchez Wordpress sur <https://artifacthub.io/>.

- Prenez la version de **Bitnami** et ajoutez le dépôt avec la première commande à droite (ajouter le dépôt et déployer une release).
- Installer une “**release**” `wordpress-tp` de cette application (ce chart) avec

```
helm install wordpress-tp bitnami/wordpress
```
- Des instructions sont affichées dans le terminal pour trouver l'IP et afficher le login et password de notre installation. La commande pour récupérer l'IP ne fonctionne que dans les cluster proposant une intégration avec un loadbalancer et fournissant donc des IP externe. Dans minikube (qui ne fournit pas de loadbalancer) il faut à la place lancer `minikube service wordpress-tp` pour y accéder avec le NodePort.
- Notre Wordpress est prêt. Connectez-vous-y avec les identifiants affichés (il faut passer les commandes indiquées pour récupérer le mot de passe stocké dans un secret k8s).

Vous pouvez constater que l'utilisateur est par default `user` ce qui n'est pas très pertinent. Un chart prend de nombreux paramètres de configuration qui sont toujours listés dans le fichier `values.yaml` à la racine du Chart.

On peut écraser certains de ces paramètres dans un nouveau fichier par exemple `myvalues.yaml` et installer la release avec l'option `--values=myvalues.yaml`.

- Désinstallez Wordpress avec `helm uninstall wordpress-tp`

Utiliser la fonction `template` de Helm pour étudier les ressources d'un Chart [🔗](#)

- Visitez le code des charts de votre choix en clonant le répertoire Git des Charts officielles Bitnami et en l'explorant avec VSCode :

```
git clone https://github.com/bitnami/charts/ --depth 1
code charts
```

- Regardez en particulier les fichiers `templates` et le fichier de paramètres `values.yaml`.
- Comment modifier l'username et le password wordpress à l'installation ? il faut donner comme paramètres le yaml suivant:

```
wordpressUsername: <votrenom>
wordpressPassword: <easytoguesspasswd>
```

- Nous allons paramétrer plus encore l'installation. Créez un dossier TP5 avec à l'intérieur un fichier `values.yaml` contenant:

```
wordpressUsername: <stagiaire> # replace
wordpressPassword: myunsecurepassword
wordpressBlogName: Kubernetes example blog

replicaCount: 1

service:
  type: ClusterIP

ingress:
  enabled: true
  hostname: wordpress.<stagiaire>.<labdomain> # replace with your hostname
  pointing on the cluster ingress loadbalancer IP
  tls: true
  certManager: true
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
    kubernetes.io/ingress.class: nginx
```

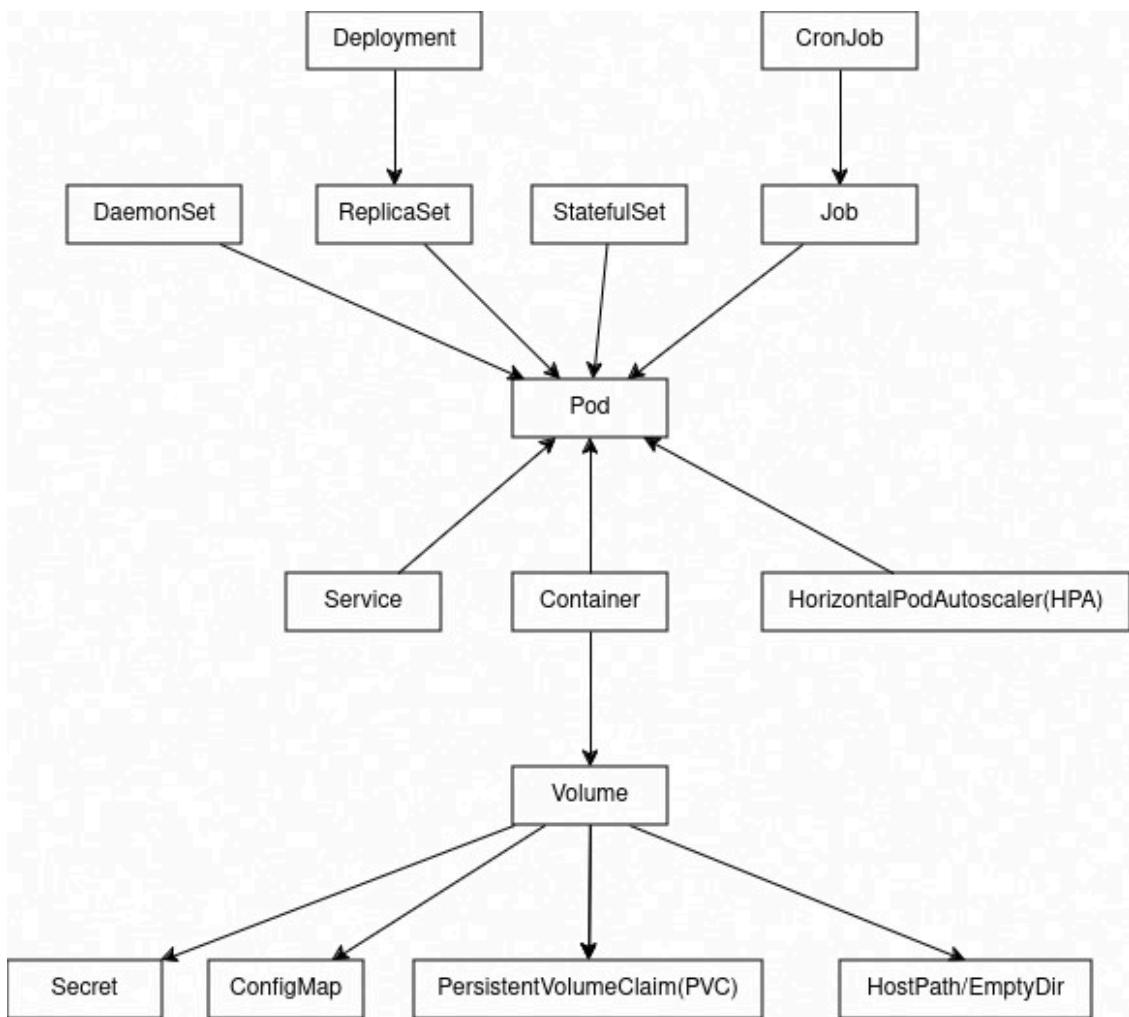
- En utilisant ces paramètres, plutôt que d'installer le chart, nous allons faire le rendu (templating) des fichiers ressource générés par le chart:

```
helm template wordpress-tp bitnami/wordpress --values=values.yaml > wordpress-tp-manifests.yaml.
```

On peut maintenant lire dans ce fichier les objets kubernetes déployés par le chart et ainsi apprendre de nouvelles techniques et syntaxes. En le parcourant on peut constater que la plupart des objets abordés pendant cette formation y sont présent plus certains autres.

Formation Kubernetes

Cours - Les différents contrôleurs pour les pods



En plus du déploiement d'un application, Il existe pleins d'autre raisons de créer un ensemble de Pods:

- Le **DaemonSet**: Faire tourner un agent ou démon sur chaque nœud, par exemple pour des besoins de monitoring, ou pour configurer le réseau sur chacun des nœuds.

- **Le Job** : Effectuer une tache unique de durée limitée et ponctuelle, par exemple de nettoyage d'un volume ou la préparation initiale d'une application, etc.
- **Le CronJob** : Effectuer une tache unique de durée limitée et récurrente, par exemple de backup ou de régénération de certificat, etc.

De plus même pour faire tourner une application, les déploiements ne sont pas toujours suffisants. En effet ils sont peu adaptés à des applications statefull comme les bases de données de toutes sortes qui ont besoin de persister des données critiques. Pour cela on utilise un **StatefulSet** que nous verrons par la suite.

Étant donné les similitudes entre les DaemonSets, les StatefulSets et les Deployments, il est important de comprendre un peu précisément quand les utiliser.

Les **Deployments** (liés à des ReplicaSets) doivent être utilisés :

- lorsque votre application est complètement découplée du nœud
- que vous pouvez en exécuter plusieurs copies sur un nœud donné sans considération particulière
- que l'ordre de création des replicas et le nom des pods n'est pas important
- lorsqu'on fait des opérations *stateless*

Les **DaemonSets** doivent être utilisés :

- lorsqu'au moins une copie de votre application doit être exécutée sur tous les nœuds du cluster (ou sur un sous-ensemble de ces nœuds).

Les **StatefulSets** doivent être utilisés :

- lorsque l'ordre de création des replicas et le nom des pods est important
- lorsqu'on fait des opérations *stateful* (écrire dans une base de données)

Jobs [🔗](#)

Les jobs sont utiles pour les choses que vous ne voulez faire qu'une seule fois, comme les migrations de bases de données ou les travaux par lots. Si vous exécutez une migration en tant que Pod dans un deployment:

- Dès que la migration se finit le processus du pod s'arrête.
- Le **replicaset** qui détecte que l'"application" s'est arrêter va tenter de la redémarrer en recréant le pod.
- Votre tâche de migration de base de données se déroulera donc en boucle, en repeuplant continuellement la base de données.

CronJobs [🔗](#)

Comme des jobs, mais se lancent à un intervalle régulier, comme les `cron` sur les systèmes unix.

Exemple de Cronjob pour un backup avec Velero [🔗](#)

Voici un exemple de Job Kubernetes pour effectuer un Job régulier. Par exemple un backup Velero.

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: daily-job
spec:
  schedule: "0 0 * * *" # Planifie la tâche tous les jours à minuit
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: my-container
              image: my-image:latest
              command: ["/bin/sh"]
              args: ["-c", "echo 'Hello, World!'"]
          restartPolicy: OnFailure
```

Des déploiements plus stables et précautionneux : les StatefulSets [🔗](#)

L'objet `statefulset` est relativement récent dans Kubernetes.

On utilise les `statefulsets` pour répliquer un ensemble de pods dont l'état est important : par exemple, des pods dont le rôle est d'être une base de données, manipulant des données sur un disque.

Un objet `StatefulSet` représente un ensemble de pods dotés d'identités uniques et de noms d'hôtes stables. Quand on supprime un StatefulSet, par défaut les volumes liés ne sont pas supprimés.

Les StatefulSets utilisent un nom en commun suivi de numéros qui se suivent. Par exemple, un StatefulSet nommé `web` comporte des pods nommés `web-0`, `web-1` et `web-2`. Par défaut, les pods StatefulSet sont déployés dans l'ordre et arrêtés dans l'ordre inverse (`web-2`, `web-1` puis `web-0`).

En général, on utilise des StatefulSets quand on veut :

- des identifiants réseau stables et uniques
- du stockage stable et persistant
- des déploiements et du scaling contrôlés et dans un ordre défini
- des rolling updates dans un ordre défini et automatisées

Article récapitulatif des fonctionnalités de base pour applications stateful:

<https://medium.com/capital-one-tech/conquering-statefulness-on-kubernetes-26336d5f4f17>

Exemple très minimal avec Cassandra:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
spec:
  selector:
    matchLabels:
      app: cassandra
  serviceName: cassandra
  replicas: 3
  template:
    metadata:
      labels:
        app: cassandra
    spec:
      containers:
        - name: cassandra
          image: cassandra:3.11.10
          env:
            - name: CASSANDRA_SEEDS
              value: "cassandra-0.cassandra.default.svc.cluster.local,cassandra-1.cassandra.default.svc.cluster.local,cassandra-2.cassandra.default.svc.cluster.local"
            - name: CASSANDRA_CLUSTER_NAME
              value: "my-cassandra-cluster"
            - name: CASSANDRA_DC
              value: "dc1"
            - name: CASSANDRA_RACK
              value: "rack1"
          ports:
            - containerPort: 9042
              name: cql
          volumeMounts:
            - name: data
              mountPath: /var/lib/cassandra/data
  volumeClaimTemplates:
    - metadata:
        name: data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 10Gi
```

Formation Kubernetes

TP - Paramétrer monsterstack pour plusieurs environnements avec Kustomize

Gérer plusieurs environnements avec Kustomize [🔗](#)

Kustomize est un outil de paramétrage et de mutation d'applications Kubernetes intégré directement dans `kubectl`. Il permet de gérer plusieurs variantes d'une même application (dev, staging, production) sans dupliquer les fichiers de configuration.

Le problème à résoudre [🔗](#)

Actuellement, notre application monsterstack utilise une ConfigMap pour sa configuration. Mais comment gérer différentes configurations pour différents environnements (dev, prod) sans dupliquer tous nos fichiers YAML ?

Kustomize résout ce problème en utilisant :

- Une **base** commune contenant les ressources partagées
- Des **overlays** (surcouches) pour chaque environnement qui modifient la base selon les besoins

Architecture Kustomize [🔗](#)

Nous allons créer une structure de dossiers comme suit :

```

k8s/
└── base/                      # Configuration de base (commune)
    ├── kustomization.yaml
    ├── frontend.yaml
    ├── imagebackend.yaml
    ├── redis.yaml
    ├── redis-data-pvc.yaml
    └── monsterstack-ingress.yaml
└── dev/                         # Overlay pour l'environnement dev
    ├── kustomization.yaml
    └── ingress-domain.yaml
└── prod/                         # Overlay pour l'environnement prod
    ├── kustomization.yaml
    ├── ingress-domain.yaml
    └── frontend-replicas.yaml

```

Étape 1 : Créer la base [🔗](#)

Copier les fichiers existants [🔗](#)

- Créez la structure de dossiers :

```
mkdir -p k8s/base k8s/dev k8s/prod
```

- Copiez tous vos fichiers YAML actuels (du TP monsterstack configmap) dans `k8s/base/`, **sauf** le fichier `frontend-config.yaml` que nous allons générer automatiquement avec Kustomize.

Créer le fichier `kustomization.yaml` de base [🔗](#)

- Créez le fichier `k8s/base/kustomization.yaml` :

```

apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- frontend.yaml
- imagebackend.yaml
- redis.yaml
- redis-data-pvc.yaml
- monsterstack-ingress.yaml

configMapGenerator:
- name: frontend-config
  literals:
  - CONTEXT=PROD
  - IMAGEBACKEND_DOMAIN=imagebackend
  - REDIS_DOMAIN=redis

```

Explications :

- `resources`: Liste des fichiers YAML à inclure
- `configMapGenerator`: Génère automatiquement une ConfigMap au lieu de la définir dans un fichier séparé
- Les `literals` sont les paires clé-valeur de la ConfigMap

Tester la base [🔗](#)

- Générez et visualisez le résultat sans déployer :

```

● ● ●

kubectl kustomize k8s/base

```

Cette commande affiche tous les manifestes générés. Notez que la ConfigMap a un nom avec un hash ajouté automatiquement (ex: `frontend-config-abc123`). Cela permet de déclencher un rolling update quand la configuration change.

Étape 2 : Créer l'overlay dev [🔗](#)

Créer le kustomization.yaml pour dev [🔗](#)

- Créez le fichier `k8s/dev/kustomization.yaml` :

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- ./base

nameSuffix: -dev

configMapGenerator:
- name: frontend-config
  behavior: replace
  literals:
  - CONTEXT=DEV
  - IMAGEBACKEND_DOMAIN=imagebackend
  - REDIS_DOMAIN=redis

patches:
- target:
  kind: Ingress
  name: monsterstack
  path: ./ingress-domain.yaml
```

Explications :

- `resources: ./base`: Hérite de toutes les ressources de la base
- `nameSuffix: -dev`: Ajoute `-dev` à tous les noms de ressources
- `configMapGenerator` avec `behavior: replace`: Remplace la ConfigMap de base avec des valeurs pour dev
- `patches`: Modifie l'Ingress pour utiliser un domaine dev

Créer le patch pour l'Ingress dev [🔗](#)

- Créez le fichier `k8s/dev/ingress-domain.yaml` :

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: monsterstack
spec:
  tls:
  - hosts:
    - dev.elie.formation.dopl.uk
    secretName: monsterstack-tls-dev
  rules:
  - host: dev.elie.formation.dopl.uk
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend
            port:
              number: 5000

```

Comment ça marche ?

Kustomize utilise une **strategic merge** : il fusionne intelligemment ce patch avec l'Ingress de base. Les champs spécifiés ici remplacent ceux de la base, les autres sont conservés.

Tester l'overlay dev [🔗](#)

- Visualisez le résultat pour dev :

```

● ● ●
kubectl kustomize k8s/dev

```

Remarquez :

- Tous les noms ont le suffixe `-dev`
- La ConfigMap contient `CONTEXT=DEV`
- L'Ingress utilise `dev.elie.formation.dopl.uk`

Étape 3 : Créer l'overlay prod [🔗](#)

Créer le kustomization.yaml pour prod [🔗](#)

- Créez le fichier `k8s/prod/kustomization.yaml`:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- ./base

nameSuffix: -prod

configMapGenerator:
- name: frontend-config
  behavior: replace
  literals:
  - CONTEXT=PROD
  - IMAGEBACKEND_DOMAIN=imagebackend
  - REDIS_DOMAIN=redis

patches:
- target:
  kind: Ingress
  name: monsterstack
  path: ./ingress-domain.yaml
- path: ./frontend-replicas.yaml
```

Créer les patches pour prod [🔗](#)

- Créez le fichier `k8s/prod/ingress-domain.yaml`:

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: monsterstack
spec:
  tls:
  - hosts:
    - prod.monsterstack.local
    secretName: monsterstack-tls-prod
  rules:
  - host: prod.monsterstack.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend
            port:
              number: 5000

```

- Créez le fichier `k8s/prod/frontend-replicas.yaml` pour augmenter le nombre de répliques en prod :

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 7

```

Ce patch remplace simplement le nombre de replicas (3 dans la base → 7 en prod).

Tester l'overlay prod [🔗](#)

- Visualisez le résultat pour prod :

```
● ● ●  
kubectl kustomize k8s/prod
```

Remarquez :

- Tous les noms ont le suffixe `-prod`
- La ConfigMap contient `CONTEXT=PROD`
- L'Ingress utilise `prod.monsterstack.local`
- Le frontend a 7 replicas au lieu de 3

Étape 4 : Déployer avec Kustomize [🔗](#)

Déployer l'environnement dev [🔗](#)

```
● ● ●  
kubectl apply -k k8s/dev
```

L'option `-k` indique à `kubectl` d'utiliser Kustomize.

- Vérifiez les ressources créées :

```
● ● ●  
kubectl get all -l app=monsterstack  
kubectl get configmap | grep frontend-config  
kubectl get ingress
```

Toutes les ressources devraient avoir le suffixe `-dev`.

Déployer l'environnement prod (dans un autre namespace) [🔗](#)

Pour isoler complètement les environnements, on peut les déployer dans des namespaces différents.

- Créez un namespace pour prod :

```
kubectl create namespace prod
```

- Déployez dans le namespace prod :

```
kubectl apply -k k8s/prod -n prod
```

- Vérifiez les ressources dans les deux namespaces :

```
kubectl get all -n default -l app=monsterstack  
kubectl get all -n prod -l app=monsterstack
```

Mettre à jour la configuration

Un des grands avantages de Kustomize : modifier facilement la configuration.

- Modifiez `k8s/dev/kustomization.yaml` pour changer `CONTEXT=DEV` en `CONTEXT=STAGING`
- Redéployez :

```
kubectl apply -k k8s/dev
```

Le hash de la ConfigMap change automatiquement, ce qui déclenche un rolling update des pods frontend.

Avantages de Kustomize [🔗](#)

- **DRY (Don't Repeat Yourself)** : Pas de duplication de configuration
- **Versionning facile** : Tout est en YAML, versionnable dans Git
- **Natif dans kubectl** : Pas d'outil externe à installer
- **ConfigMap avec hash** : Les changements de config déclenchent automatiquement des rolling updates
- **Composabilité** : On peut créer des overlays d'overlays
- **Pas de templating** : Contrairement à Helm, Kustomize travaille avec des fichiers YAML valides

Pour aller plus loin [🔗](#)

Intégration avec Skaffold [🔗](#)

Skaffold supporte nativement Kustomize. Modifiez votre `skaffold.yaml` :

```
apiVersion: skaffold/v2beta20
kind: Config
build:
  artifacts:
    - image: registry.example.com/frontend
deploy:
  kustomize:
    paths:
      - k8s/dev # ou k8s/prod
```

Autres fonctionnalités Kustomize [🔗](#)

- **commonLabels** : Ajouter des labels à toutes les ressources
- **commonAnnotations** : Ajouter des annotations à toutes les ressources

- **images** : Modifier les tags d'images sans patcher
- **replicas** : Modifier le nombre de replicas directement dans kustomization.yaml
- **secretGenerator** : Générer des Secrets comme les ConfigMaps

Ressources

- Documentation officielle :
<https://kubectl.docs.kubernetes.io/guides/introduction/kustomize/>
- Kustomize GitHub : <https://github.com/kubernetes-sigs/kustomize>
- Article sur les patches : <https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/>

Formation Kubernetes

TP - MonsterStack avec Redis StatefulSet

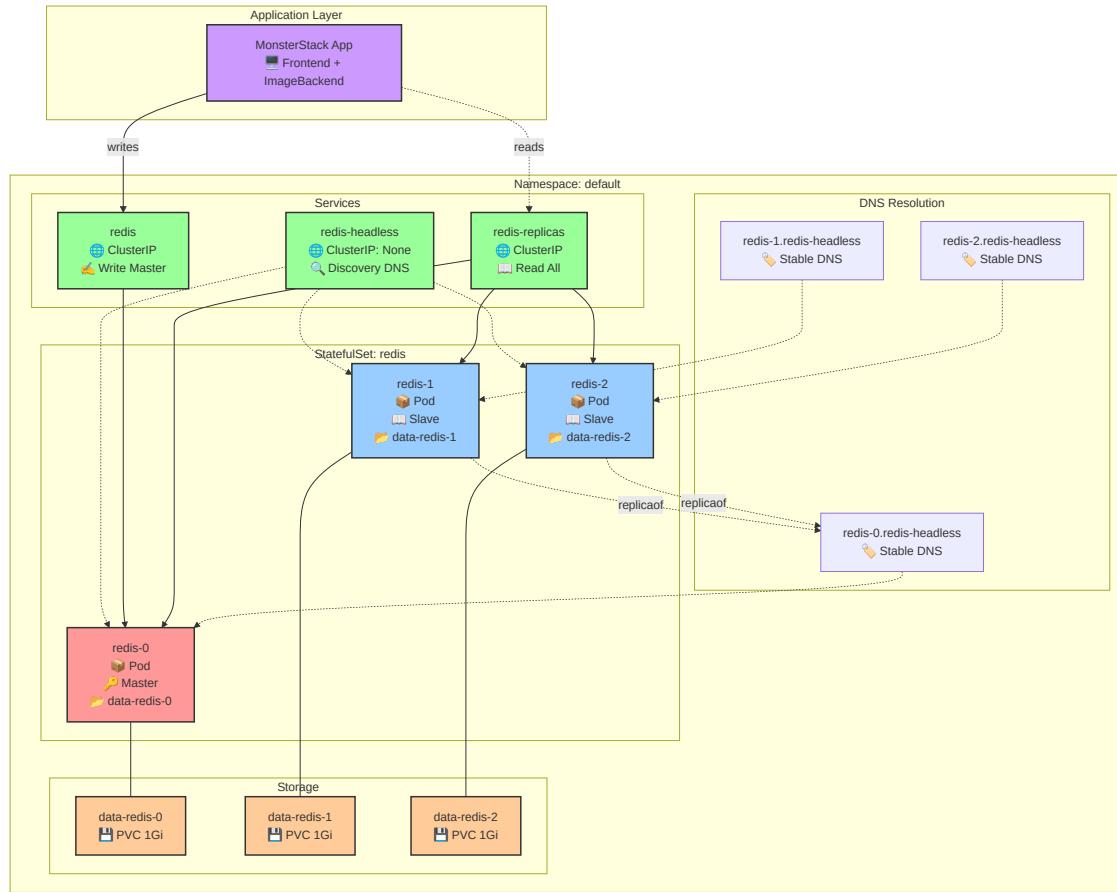
TP : Déployer MonsterStack avec Redis en mode répliqué (StatefulSet) [🔗](#)

Objectifs [🔗](#)

Dans ce TP, nous allons améliorer notre déploiement MonsterStack en remplaçant le Redis simple par un cluster Redis répliqué utilisant un StatefulSet.

Ce qui illustrera un cas classique de déploiement stateful simple.

Architecture [🔗](#)



Comprendre les StatefulSets [🔗](#)

Un StatefulSet est un contrôleur Kubernetes qui gère le déploiement et la mise à l'échelle d'un ensemble de Pods, avec les garanties suivantes :

- **Identité stable** : Chaque pod a un nom persistant (redis-0, redis-1, redis-2)
- **Ordre de déploiement** : Les pods sont créés et supprimés dans l'ordre
- **Stockage persistant** : Chaque pod peut avoir son propre volume persistant
- **Réseau stable** : Nom DNS stable pour chaque pod

Pourquoi utiliser un StatefulSet pour Redis ? [🔗](#)

Redis en mode réPLICATION nécessite :

- Une identification stable du master (redis-0)
- Une persistance des données même après redémarrage
- Une configuration différente pour master et slaves

Créer la ConfigMap Redis [🔗](#)

Créez un fichier `redis-configmap.yaml` :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: redis-config
  namespace: default
data:
  master.conf: |
    # Configuration du Master Redis
    bind 0.0.0.0
    protected-mode no
    port 6379
    tcp-keepalive 300

    daemonize no
    loglevel notice
    databases 16

    # Persistance
    save 900 1
    save 300 10
    save 60 10000
    stop-writes-on-bgsave-error yes
    rdbcompression yes
    rdbchecksum yes
    dbfilename dump.rdb
    dir /data

    # Mémoire
    maxmemory 256mb
    maxmemory-policy allkeys-lru

  slave.conf: |
    # Configuration des Replicas Redis
    bind 0.0.0.0
    protected-mode no
    port 6379
    tcp-keepalive 300

    daemonize no
    loglevel notice
    databases 16

    # RéPLICATION - le master est toujours redis-0
    replicaof redis-0.redis-headless 6379
    slave-read-only yes

    dir /data

    # Mémoire
    maxmemory 256mb
    maxmemory-policy allkeys-lru
```

En résumé :

- **Deux configurations distinctes** : Une pour le master (`master.conf`) et une pour les slaves (`slave.conf`)
- **Configuration master** : Active la persistance RDB avec sauvegarde automatique (`save 900 1, save 300 10, etc.`) et stockage dans `/data`
- **Configuration slaves** : Contient `replicaof redis-0.redis-headless 6379` pour se connecter automatiquement au master et `slave-read-only yes` pour empêcher les écritures
- **Gestion dynamique** : L'initContainer choisit automatiquement la bonne configuration selon le nom du pod (redis-0 = master, autres = slaves)

Créer les Services [🔗](#)

Service Headless pour la découverte [🔗](#)

Créez un fichier `redis-services.yaml` :

```
# Service Headless pour la découverte des pods
apiVersion: v1
kind: Service
metadata:
  name: redis-headless
  namespace: default
  labels:
    app: redis
spec:
  type: ClusterIP
  clusterIP: None # Headless service
  ports:
    - port: 6379
      targetPort: 6379
      name: redis
  selector:
    app: redis
---
# Service pour accéder au master uniquement
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  namespace: default
  labels:
    app: redis
spec:
  type: ClusterIP
  ports:
    - port: 6379
      targetPort: 6379
      name: redis
  selector:
    app: redis
    statefulset.kubernetes.io/pod-name: redis-0
---
# Service pour accéder à tous les pods (lecture répartie)
apiVersion: v1
kind: Service
metadata:
  name: redis-replicas
  namespace: default
  labels:
    app: redis
spec:
  type: ClusterIP
  ports:
    - port: 6379
      targetPort: 6379
      name: redis
  selector:
    app: redis
```

Explications de nos services :

- `redis-headless` : Service headless (`clusterIP: None`) qui permet la découverte DNS. Chaque pod Redis a son propre nom DNS stable (`redis-0.redis-headless`, `redis-1.redis-headless`, etc.). Essentiel pour la réPLICATION car les slaves doivent pouvoir contacter directement le master par son nom DNS.
- `redis` : Service principal qui pointe uniquement vers `redis-0` (master) via le sélecteur `statefulset.kubernetes.io/pod-name: redis-0`. Nommé simplement "redis" pour assurer la compatibilité avec l'application existante du TP 147. Toutes les écritures passent par ce service.
- `redis-replicas` : Service qui load-balance sur TOUS les pods Redis (master + slaves) via le sélecteur `app: redis`. Utile pour distribuer les lectures sur l'ensemble du cluster et améliorer les performances.

Créer le StatefulSet

Créez un fichier `redis-statefulset.yaml` :

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
  namespace: default
spec:
  serviceName: redis-headless # Service headless associé
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      initContainers:
        - name: config
          image: redis:7.2-alpine
          command:
            - sh
            - -c
            - |
              set -ex
              # Déterminer si c'est le master ou un slave
              if [ "$(hostname)" = "redis-0" ]; then
                echo "Initializing as MASTER"
                cp /mnt/config-map/master.conf /etc/redis/redis.conf
              else
                echo "Initializing as REPLICA"
                cp /mnt/config-map/slave.conf /etc/redis/redis.conf
              fi
      volumeMounts:
        - name: redis-config
          mountPath: /mnt/config-map
        - name: config
          mountPath: /etc/redis/
  containers:
    - name: redis
      image: redis:7.2-alpine
      command:
        - redis-server
        - /etc/redis/redis.conf
      ports:
        - containerPort: 6379
          name: redis
      volumeMounts:
        - name: data
          mountPath: /data
        - name: config
          mountPath: /etc/redis
  resources:
    requests:

```

```

    cpu: 100m
    memory: 128Mi
  limits:
    cpu: 500m
    memory: 512Mi
  livenessProbe:
    tcpSocket:
      port: redis
    initialDelaySeconds: 30
    periodSeconds: 10
  readinessProbe:
    exec:
      command:
        - sh
        - -c
        - redis-cli ping | grep PONG
  initialDelaySeconds: 15
  periodSeconds: 5
  volumes:
    - name: redis-config
  configMap:
    name: redis-config
    - name: config
    emptyDir: {}
  volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi

```

Quelques précisions sur le code:

- `initContainer` : Conteneur d'initialisation qui s'exécute avant Redis. Il examine le hostname du pod (`hostname` = redis-0, redis-1, etc.) et copie la configuration appropriée (master.conf pour redis-0, slave.conf pour les autres) dans un volume partagé que le conteneur principal utilisera.
- `volumeClaimTemplates` : Template qui crée automatiquement un PersistentVolumeClaim (PVC) unique pour chaque pod du StatefulSet. Contrairement aux Deployments, chaque pod garde son propre stockage même après redémarrage, garantissant la persistance des données.

- `serviceName: redis-headless` : Relie le StatefulSet au service headless pour activer les noms DNS stables.
- **Probes de santé** : `livenessProbe` (TCP) vérifie que Redis répond, `readinessProbe` (`redis-cli ping`) s'assure que Redis est opérationnel avant de recevoir du trafic.

Mettre à jour l'application 🔗

L'application peut continuer à utiliser le service `redis` sans modification, car il pointe automatiquement vers le master :

```
# Dans frontend.yaml et imagebackend.yaml (inchangé depuis le TP 147)
env:
  - name: REDIS_HOST
    value: "redis" # Pointe vers le master (redis-0) pour les écritures
  - name: REDIS_PORT
    value: "6379"
```

Déployer l'application 🔗

```
# Créer le namespace si nécessaire
kubectl create namespace default

# Déployer Redis StatefulSet
kubectl apply -f redis-configmap.yaml
kubectl apply -f redis-services.yaml
kubectl apply -f redis-statefulset.yaml

# Attendre que les pods Redis soient prêts
kubectl wait --for=condition=ready pod -l app=redis --timeout=300s

# Déployer le reste de l'application
kubectl apply -f frontend.yaml
kubectl apply -f imagebackend.yaml
kubectl apply -f monsterstack-ingress.yaml
```

Vérifier le déploiement

Vérifier les pods

```
# Voir l'ordre de création des pods  
kubectl get pods -l app=redis -w  
  
# Vérifier les noms stables  
kubectl get pods -l app=redis  
# Doit afficher : redis-0, redis-1, redis-2
```

Vérifier la réPLICATION

```
# Se connecter au master  
kubectl exec -it redis-0 -- redis-cli  
  
# Dans redis-cli, vérifier le statut  
127.0.0.1:6379> INFO replication  
# Doit montrer role:master et 2 connected_slaves  
  
# Tester l'écriture sur le master  
127.0.0.1:6379> SET test "Hello from master"  
127.0.0.1:6379> exit  
  
# Vérifier la réPLICATION sur un slave  
kubectl exec -it redis-1 -- redis-cli GET test  
# Doit retourner "Hello from master"
```

Vérifier les services

```
# Vérifier les endpoints  
kubectl get endpoints redis-headless  
kubectl get endpoints redis  
kubectl get endpoints redis-replicas  
  
# Tester la résolution DNS  
kubectl run -it --rm debug --image=alpine --restart=Never -- nslookup redis-0.redis-headless
```

Tester la persistance [🔗](#)

```
# Créer des données  
kubectl exec -it redis-0 -- redis-cli SET persistent "data"  
  
# Supprimer le pod master  
kubectl delete pod redis-0  
  
# Attendre la recréation  
kubectl wait --for=condition=ready pod/redis-0 --timeout=120s  
  
# Vérifier que les données sont toujours là  
kubectl exec -it redis-0 -- redis-cli GET persistent  
# Doit retourner "data"
```

Scaler le StatefulSet [🔗](#)

```
# Augmenter le nombre de replicas  
kubectl scale statefulset redis --replicas=5  
  
# Observer l'ordre de création  
kubectl get pods -l app=redis -w  
  
# Réduire  
kubectl scale statefulset redis --replicas=3  
  
# Observer l'ordre de suppression (inverse)  
kubectl get pods -l app=redis -w
```

Limitations et considérations [🔗](#)

1. **Failover manuel** : Si redis-0 tombe, pas de promotion automatique
2. **Performance** : Les slaves sont en lecture seule

On pourrait installer un opérateur Redis pour avoir une solution plus avancée. Il faudrait également se préoccuper du monitoring et backup de ce Redis

Formation Kubernetes

Les resources kubernetes exotiques plus avancées

Lister toutes les resources de l'API [🔗](#)

Leases [🔗](#)

Admission webhooks [🔗](#)

Pod disruption budget [🔗](#)

Formation Kubernetes

Cours optionnel - Règles de scheduling et capacity planning

Règles de scheduling (planification) [🔗](#)

Le rôle du *scheduler* est de choisir les nœuds sur lesquels exécuter les pods nouvellement créés.

C'est une problématique très compliquée dans le cas général qui dépend complètement des situations et besoins.

Par défaut, le *scheduler* suit les règles :

- Diviser les pods du même replicaset ou statefulset entre les nœuds.
- Planifier les pods sur des nœuds disposant de suffisamment de ressources pour satisfaire les demandes de pods.
- Équilibrer l'utilisation globale des ressources des nœuds.

C'est un comportement par défaut assez bon, mais parfois, vous pouvez vouloir un meilleur contrôle sur le placement spécifique des pods et la gestion des Resources.

Contraintes de placement [🔗](#)

NodeSelector [🔗](#)

Un pod peut spécifier sur quels nœuds il souhaite être planifié dans sa `spec`.

Par exemple, un pod nginx avec un `nodeSelector` spécifiant l'étiquette `kubernetes.io/hostname` du nœud `kcluster-worker2` (nœud à partir de l'installation `kubeadm ansible`) :

```

apiVersion: v1
kind: Pod
metadata:
  name: nginx-scheduling
  labels:
    role: nginx-example
spec:
  nodeSelector:
    kubernetes.io/hostname: kluster-worker2
  containers:
    - name: nginx
      image: nginx:latest

```

Taints et tolerations [🔗](#)

Vous pouvez marquer (`taint` -> c'est différent des `labels` de noeuds) un nœud afin d'empêcher les pods d'être planifiés sur le nœud:

- si vous ne souhaitez pas que des pods soient planifiés sur vos nœuds du control plane.
- si vous avez un type de noeud réservés pour par exemple les workloads gpu
- etc

Les tolérations permettent aux pods de déclarer qu'ils peuvent "tolérer" une `taint`, et alors ces pods peuvent être planifiés sur le nœud marqué.

Un nœud peut avoir plusieurs `taints` et un pod peut avoir plusieurs `tolerations`.

L'effet peut être :

- NoSchedule (aucun pod ne sera planifié sur le nœud à moins qu'ils ne tolèrent la marque)
- PreferNoSchedule (version soft de NoSchedule ; le scheduleur tentera de ne pas planifier de pods qui ne tolèrent pas la marque)
- NoExecute (aucun nouveau pod ne sera planifié, mais aussi les pods existants qui ne tolèrent pas la marque seront évincés)

Tentons de marquer notre nœud `worker2` pour expulser les pods:

```
kubectl taint nodes kluster-worker2 taint-example=true:NoExecute
```

Pour permettre aux pods de tolérer cette taint, ajoutez une toleration à leur `spec`:

```
tolerations:
  - key: "taint-example"
    operator: "Equal"
    value: "true"
    effect: "NoExecute"
```

Critères Node affinity et anti-affinity [🔗](#)

La `node affinity` est proche mais plus élaborée que le concept de `nodeSelector` :

- Critères de sélection plus riches
- Les règles peuvent être souples (s'appliquer si possible)
- Vous pouvez obtenir une `anti-affinity` en utilisant des opérateurs comme `NotIn` et `DoesNotExist`

Si vous spécifiez à la fois `nodeSelector` et `nodeAffinity`, alors le pod sera planifié uniquement sur un nœud qui satisfait **les deux** exigences.

Par exemple, si nous ajoutons la section suivante à notre pod nginx, il ne pourra pas s'exécuter sur aucun nœud car cela entre en conflit avec le `nodeSelector`:

```

affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/hostname
            operator: NotIn
            values:
              - kluster-worker2

```

Critères Pod affinity et anti-affinity [🔗](#)

L'affinité et l'anti-affinité des pods offrent une autre solution pour gérer l'emplacement où s'exécutent vos charges de travail.

Toutes les méthodes que nous avons discutées jusqu'à présent - les sélecteurs de nœuds, les marques/tolérances, l'affinité/anti-affinité des nœuds - concernaient l'attribution de pods à des nœuds.

Mais la `pod affinity` concerne les relations entre différents pods.

la `pod affinity` fonctionne avec d'autres concepts :

- le namespace (puisque les pods sont namespaced)
- la topology zone (nœud, rack, zone du fournisseur cloud, région du fournisseur cloud)
- le poids (pour une planification préférentielle).

Un exemple simple est si vous voulez que hue-reminders soit toujours planifié avec un pod. Voyons comment le définir dans la spécification du modèle de pod du déploiement hue-reminders :

```

affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - weight: 200
        labelSelector:
          matchExpressions:
            - key: role
              operator: In
              values:
                - nginx-example
    topologyKey: topology.kubernetes.io/zone # for clusters on cloud
  providers
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: role
                operator: In
                values:
                  - pod-reposoir
    topologyKey: topology.kubernetes.io/zone

```

La topologyKey est une étiquette de nœud que Kubernetes utilise pour rassembler les pods qui doivent fonctionner ensemble : dans le cas classique d'un cloud provider on utilise `topology.kubernetes.io/zone` lorsque les workloads doivent s'exécuter dans un même datacenter.

Avec le code d'exemple précédent les nouveaux pods avec cette spec seront planifiés sur `kluster-worker2` à côté du pod `nginx-example` et loin du pod-reposoir mais c'est moins prioritaire.

Pod topology spread constraints

L'affinité/anti-affinité de nœud et l'affinité/anti-affinité de pod sont parfois trop strictes et spécifiques. Vous pouvez vouloir répartir vos pods globalement en tolérant une répartition pas totalement équitable.

Les Pod topology spread constraints permettent de spécifier le décalage maximal (max skew), qui représente la distance à laquelle vous pouvez vous trouver

de la répartition optimale, ainsi que le comportement lorsque la contrainte ne peut pas être satisfaite (DoNotSchedule ou ScheduleAnyway).

Voici un exemple sur `monsterstack` avec des topology spread constraints :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:
spec:
  replicas: 7
  selector:
    matchLabels:
      app: monsterstack
      partie: frontend
  template:
    metadata:
      name: frontend
    labels:
      app: monsterstack
      partie: frontend
  spec:
    topologySpreadConstraints:
    - maxSkew: 2
      topologyKey: node.kubernetes.io/instance-type
      whenUnsatisfiable: DoNotSchedule
      labelSelector:
        matchLabels:
          app: monsterstack
          service: frontend
    containers:
    - name: frontend
      image: frontend:latest
      ports:
      - containerPort: 5000
```

Le plugin Descheduler [🔗](#)

Une fois qu'un pod est planifié, Kubernetes ne le déplacera pas vers un autre nœud si les conditions initiales ont changé. ce qui peut poser problème si:

- Certains nœuds sont sous-utilisés ou sur-utilisés.

- La décision initiale de planification n'est plus valide lorsque des taints ou labels ont changé.
- Noeuds défaillants ou nouveau on entraîné des migrations de pods

Le descheduler (<https://github.com/kubernetes-sigs/descheduler>) vérifiera périodiquement le placement actuel des pods et procèdera à l'eviction les pods qui enfreignent certaines contraintes. Les pods seront ensuite reschedule normalement avec les règles a jour.

Resources et capacity planning

Dans un cluster kubernetes on peut ajouter des noeuds facilement donc pas besoin d'anticiper trop à l'avance les besoins en resources mais il faut tout de même étudier ses programmes et arbitrer au fur et a mesure entre:

- utiliser au maximum les resources
- avoir des resources en réserve pour les imprévus

Il est recommandé à la louche de ne pas dépasser 50-60% d'utilisation des ressources ce qui permet d'éviter les réactions en chaîne si un noeud crashe ou si on veut faire un blue-green deploy en prod par exemple.

Étudier ses workloads

- Connaître la "shape" c'est à dire le ratio memoire/CPU de ses workloads et choisir des noeuds adaptés avec un ratio proche
- Avoir une idée dynamique via du monitoring de la consommation des workload en fonction de la charge sur le hardware spécifique de votre cluster (le CPU c'est très variable selon les machines)
- Savoir si les workloads consomment plus au démarrage ou à certains moments pour décloupler les tâches consommatrices et moins consommatrices avec des init containers ou conteneurs annexes : par

exemple on peut mettre des limits/requests haute sur l'init container ce qui évite de devoir donner trop de ressource au workload

Requests et limits [🔗](#)

Voir TP

ResourceQuota [🔗](#)

Limiter les resources pour un namespace : très important en cas de multi-tenancy.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: resource-quota-example
spec:
  hard:
    pods: "10"
    requests.cpu: "4"
    requests.memory: 4Gi
    limits.cpu: "6"
    limits.memory: 6Gi
```

LimitRange [🔗](#)

Spécifier les limites raisonnables pour un conteneur et au total pour un pod mais au niveau du namespace.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-range-example
spec:
  limits:
  - type: Pod
    max:
      cpu: "1"
      memory: 512Mi
    min:
      cpu: "100m"
      memory: 64Mi
  - type: Container
    max:
      cpu: "500m"
      memory: 256Mi
    min:
      cpu: "50m"
      memory: 32Mi
  default:
    cpu: "100m"
    memory: 64Mi
```

Formation Kubernetes

TP optionnel - Écrire un chart pour notre application monsterstack

- Récupérez la correction du dernier TP sur le déploiement de la monsterstack avec la commande: `git clone -b tp_monsterstack_correction_configmap https://github.com/Uptime-Formation/corrections_tp.git tp_monsterchart.`
- Ouvrez le dans VSCode

Pour démarrer le développement d'un chart on peut utiliser une commande helm d'initialisation qui va générer un chart d'exemple :

- aller sur le Bureau: `cd ~/Desktop/tp_monsterchart`
- puis créer le chart: `helm create monsterchart`

Observons un peu le contenu de notre Chart d'exemple :

- Un dossier `templates` avec tous les fichiers resources à trou qui seront templatisés par helm et quelques snippet utilitaires de templating dans `_helpers.tpl`
- Le fichier `values.yaml` avec les valeurs par défaut pour l'installation, qu'on va pouvoir surcharger pour installer le chart
- Un fichier `NOTES.txt` contenant le texte d'aide qui s'affichera pour l'utilisateur à la fin de l'installation
- Un fichier `Chart.yaml` contenant des informations/paramètres caractérisant le chart lui même comme son nom, sa version et la version nde l'application installée correspondante.

Ce chart d'exemple est déjà installable en l'état et créé un déploiement de nginx, un service et un ingress pour pouvoir afficher la page d'accueil de Nginx.

Nous avons un soucis pour étudier ce chart : la syntaxe lourdement chargée de templating est assez illisible en l'état. Étudier le déploiement concret implique de réaliser le templating pour visualiser le résultat final. Deux méthodes plutôt sont possibles :

- `helm template <releasename> --values=valuefile.yaml ./<chart_folder> > result.yaml`
- `helm install <releasename> --dry-run --debug --values=valuefile.yaml ./chart_folder > debug-chart.yaml`

La seconde a l'avantage d'afficher des informations de debug comme les informations de la release et la liste des valeurs calculées pour le templating.

- Utilisez la deuxième commande pour faire une évaluation de ce qui serait installé avec le chart par défaut

Une version statique de notre chart

Nous allons dans un premier temps remplacer les templates (à trou) par les fichiers statiques de notre déploiement du TP précédent, puis nous ajouterons quelques paramètres.

- Dupliquez le dossier `templates` en `templates_backup` puis supprimez les fichiers `template.yaml` et le dossier `test` du dossier `templates`
- Ajoutez vos fichiers d'installation copiés depuis `k8s-deploy-dev`.
- Testez le templating avec la commande utilisée précédemment.

Ce chart statique a plusieurs soucis mais notamment il ne permet pas d'être installé plusieurs fois. Nous allons corriger quelques aspects en le variabilisant.

Utiliser skaffold pour développer [🔗](#)

Skaffold est un outils pour développer de façon dynamique dans un cluster (voir TP afférant avec monsterstack)

- Créez le fichier de config skaffold suivant:

```
apiVersion: skaffold/v2beta5
kind: Config
build:
  artifacts: []
    # - image: docker.io/<votre_login_dockerhub>/frontend
deploy:
  helm:
    releases:
      - name: monsterchart
        chartPath: ./monsterchart
        # artifactOverrides:
        #   imageKey: docker.io/<votre_login_dockerhub>/frontend
        imageStrategy:
          fqn: {}
```

- Assurez vous d'être connecté au docker hub (avec `docker login`).
- Lancez `skaffold dev` va déployer le chart automatiquement à chaque modification des fichiers template (désactiver l'auto-save de vscode peut aider ainsi que SaveAll pour éviter les inconsistance pendant l'édition)

Ajouter des paramètres simples [🔗](#)

Nous allons paramétrier à minima nos templates pour pouvoir modifier:

- les images utilisées pour l'installation des services
- le nom des resources pour éviter les conflits
- la configuration du ingress
- le type de service avec un defaut à `NodePort`

Pour ce faire:

- Déplacez le fichier `values.yaml` existant dans le dossier `templates_backup`.
- Ajoutez à un nouveau fichier `values.yaml` avec le code suivant :

```

frontend:
  image:
    name: frontend
    tag: latest
  service:
    port: 5000
    type: ClusterIP
imagebackend:
  image:
    name: amouat/dnmonster
    tag: "1.0"
redis:
  image:
    name: redis
    tag: latest

```

- Remplacez ensuite dans nos trois fichiers template la valeur `image` du conteneur par ces valeurs dynamiques avec un emplacement de variable de la forme `{{ .Values.imagebackend.image.name }}:{{ .Values.imagebackend.image.tag }}`

Passons maintenant à la gestion dynamique du nom pour pouvoir installer notre chart plusieurs fois. Pour cela nous allons utiliser le helper `monsterchart.fullname` disponible dans le chart d'exemple.

- Observez et commentons le helper qui génère automatiquement un nom compatible pour notre release
- Observez dans le chart d'exemple (`templates_backup`) comment est utilisé ce helper par exemple dans le template du déploiement ?
- Remplacez pour les 3 services (redis, frontend et imagebackend) toutes les occurrences de `<nomservice>` dans le nom des ressources et les labels en ajoutant `-{{ include "monsterchart.fullname" . }}` comme suffixe.
- Testez avec la commande d'installation dry run précédente.

Enfin nous allons ajouter la gestion dynamique du ingress en copiant celle du chart d'exemple.

- Supprimez le template `monsterstack-ingress.yaml`
- Copiez le template `ingress.yaml` du backup
- Copiez la section `ingress` depuis le `values.yaml` déplacé dans `templates_backup` vers le `values.yaml` principal et complétez la avec le nom de domaine de votre choix et activez le ingress avec `enabled: true`:
- Modifiez le template ingress pour qu'il pointe vers la bonne ressource service à savoir `frontend-{{ include "monsterchart.fullname" . }}` (il y a deux valeurs à modifier `service: name:` et `serviceName:` plus bas.)
- Dans le même fichier, modifiez enfin la variable port pour l'adapter à notre cas `{{- $svcPort := .Values.service.port -}} => {{- $svcPort := .Values.frontend.service.port -}}`
- Testez le résultat et s'il vous semble bon, essayez de déployer votre chart pour de vrai (sans dry-run) et avec `tecp1/monstericon` du dockerhub comme image frontend.
- Vous pouvez essayer d'installer une autre release pour confirmer que cela fonctionne.

Fixer le nom de domaine pour matcher le nom du service [🔗](#)

- Modifier la configmap définissant les deux noms de domaine en incorporant le `fullname: <domain>-{{ include "monsterchart.fullname" . }}`

Utiliser un Chart redis [🔗](#)

- Utiliser pour l'installation de redis une dépendance à un autre chart comme par exemple ce lui de Bitnami (voir sur <https://artifacthub.io>) et la documentation ici :
https://helm.sh/docs/helm/helm_dependency/

- Ajoutez à `chart.yaml`

```
dependencies:
  - name: redis
    condition: redis.enabled
    version: "19.0.1"
    repository: "https://charts.bitnami.com/bitnami"
```

- Ajouter à `values.yaml`:

```
redis:
  enabled: true
  auth:
    enable: false
```

- supprimez les autres paramètres de redis dans `values.yaml` et les fichier de template pour redis
- lancez `helm dependency build` dans le docker `monsterchart`.
- templatez le résultat : comment s'appelle le service redis master ?
- Modifiez la configmap du frontend pour que le domaine rédis soit : "
`{{ .Release.name }}-redis-master"`

Templater notre chart pour l'utiliser en mode GitOps 🔗

Si une instance d'argoCD a été installé dans un TP précédent:

- Utilisez `helm template` pour exporter le chart sous forme d'un seul manifeste yaml
- Uploadez le sur github ou gitlab (en mode public pour simplifier l'exemple)
- Créez un projet `argocd`, idéalement dans votre propre `appProject` pour déployer l'application "en prod"

Solution

Le dépôt Git de la correction de ce TP est accessible ici : `git clone -b tp_monsterchart https://github.com/Uptime-Formation/corrections_tp.git`

Formation Kubernetes

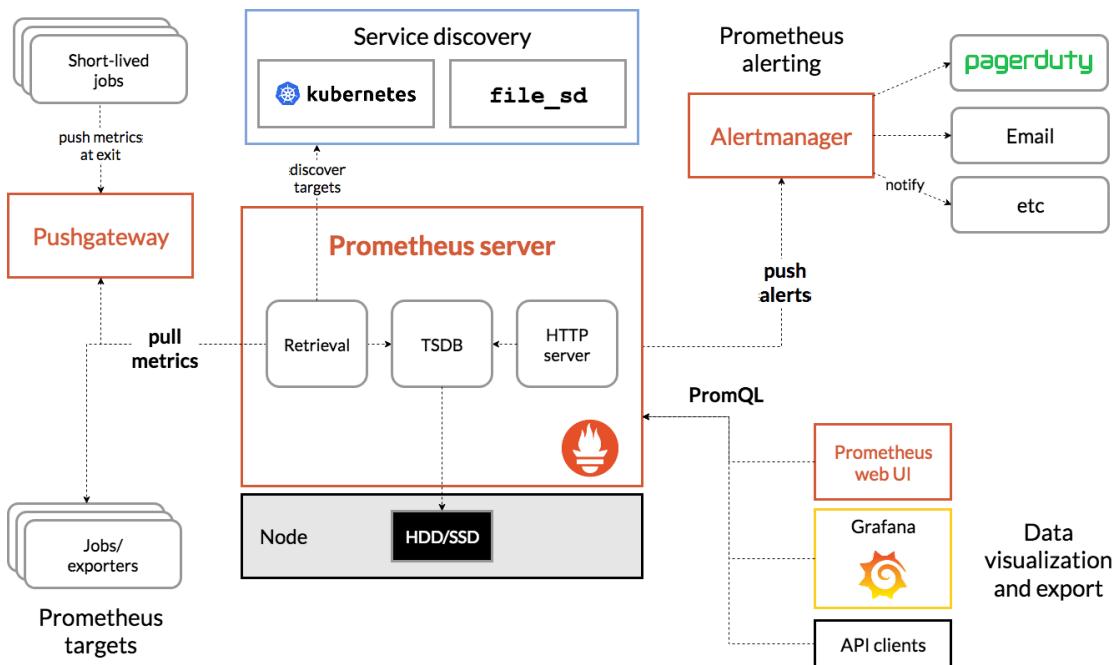
TP optionnel - Monitoring classique d'un cluster avec Prometheus

Prometheus pour monitorer le cluster [🔗](#)

Pour comprendre les stratégies de déploiement et mise à jour d'application dans Kubernetes (deployment and rollout strategies) nous allons installer puis mettre à jour une application d'exemple et observer comment sont gérées les requêtes vers notre application en fonction de la stratégie de déploiement choisie.

Pour cette observation on peut utiliser un outil de monitoring. Nous utiliserons ce TP comme prétexte pour installer une des stack les plus populaires et intégrée avec kubernetes : Prometheus et Grafana. Prometheus est un projet de la Cloud Native Computing Foundation.

Prometheus est un serveur de métriques c'est à dire qu'il enregistre des informations précises (de petite taille) sur différents aspects d'un système informatique et ce de façon périodique en effectuant généralement des requêtes vers les composants du système (metrics scraping).



Une très bonne série d'articles à jour à propos de Prometheus/Grafana et AlertManager dans kubernetes et les concept de l'observability :

<https://www.augmentedmind.de/2021/09/05/observability-prometheus-guide/>

Installer Prometheus et Grafana via kube-prometheus

Actuellement la méthode officielle conseillée pour installer Prometheus et sa webUI grafana est d'employer l'opérateur officiel `Prometheus Operator` packagé dans une stack complète appelée `kube-prometheus`:

<https://github.com/prometheus-operator/kube-prometheus>

Une façon commode de déployer cette stack est d'utiliser le chart officiel :

<https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack>

- Comme précédemment déployez cette stack via ArgoCD avec le manifeste suivant:

```
apiVersion: v1
kind: Namespace
metadata:
  name: kube-prometheus-stack
---
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kube-prometheus-stack
  namespace: argocd
spec:
  destination:
    namespace: kube-prometheus-stack
    server: https://kubernetes.default.svc
  project: default
  source:
    repoURL: https://prometheus-community.github.io/helm-charts
    chart: kube-prometheus-stack
    targetRevision: 43.1.1
  helm:
    values: |
      grafana:
        enabled: true
        ingress:
          enabled: true
          path: /
          hosts:
            - grafana.<stagiaire>.<labdomain>
        tls:
          - hosts:
              - grafana.<stagiaire>.<labdomain>
            secretName: grafana-tls-cert
      annotations:
        kubernetes.io/tls-acme: "true"
        kubernetes.io/ingress.class: nginx
        cert-manager.io/cluster-issuer: letsencrypt-prod
```

Montrer les stratégies de déploiement d'une application 🔗

Voir TP suivant.

Formation Kubernetes

TP optionnel - Stratégies de déploiement

Déployer notre application d'exemple (goprom) et la connecter à prometheus [🔗](#)

Nous allons installer une petite application d'exemple en go.

- Téléchargez le code de l'application et de son déploiement depuis github: `git clone https://github.com/e-lie/k8s-deployment-strategies`

Nous allons d'abord construire l'image docker de l'application à partir des sources et la déployer dans le cluster.

- Allez dans le dossier `goprom_app` et "construisez" l'image docker de l'application avec le tag `<votrelogindockrhub>/goprom`.

➤ Réponse

- Allez dans le dossier de la première stratégie `recreate` et ouvrez le fichier `app-v1.yml`. Notez que `image:` est à `tecp1/goprom` et qu'un paramètre `imagePullPolicy` est défini à `Never`. Changez ces paramètres si nécessaire (oui en général).

- Appliquez ce déploiement kubernetes:

➤ Réponse

Observons notre application et son déploiement kubernetes [🔗](#)

- Explorez le fichier de code go de l'application `main.go` ainsi que le fichier de déploiement `app-v1.yml`. Quelles sont les routes http exposées par l'application ?

> Réponse

- Utilisez le service `NodePort` pour accéder au service `goprom` dans votre navigateur.
- Utilisez le service `NodePort` pour accéder au service `goprom-metrics` dans votre navigateur. Quelles informations récupère-t-on sur cette route ?
- Pour tester le service `prometheus-server` nous avons besoin de le mettre en mode `NodePort` (et non `ClusterIP` par défaut). Modifiez le service dans Lens pour changer son type.
- Exposez le service avec un `NodePort` (n'oubliez pas de préciser le namespace `monitoring`).
- Vérifiez que `prometheus` récupère bien les métriques de l'application avec la requête `PromQL` : `sum(rate(http_requests_total{app="goprom"}[5m])) by (version)`.
- Quelle est la section des fichiers de déploiement qui indique à `prometheus` où récupérer les métriques ?

> Réponse

Créer une dashboard avec un Graphe. Utilisez la requête `prometheus` (champ query suivante):

```
sum(rate(http_requests_total{app="goprom"}[5m])) by (version)
```

Pour avoir un meilleur aperçu de la version de l'application accédée au fur et à mesure du déploiement, ajoutez `{{version}}` dans le champ `legend`.

Observer un basculement de version [🔗](#)

Ce TP est basé sur l'article suivant: <https://blog.container-solutions.com/kubernetes-deployment-strategies>

Maintenant que l'environnement a été configuré :

- Lisez l'article.
- Vous pouvez testez les différentes stratégies de déploiement en lisant leur `README.md`.
- En résumé, pour les plus simple, on peut:
 - appliquer le fichier `app-v1.yml` pour une stratégie.
 - lancer la commande suivante pour effectuer des requêtes régulières sur l'application: `service=$(minikube service goprom --url) ; while sleep 0.1; do curl "$service"; done`
 - Dans un second terminal (pendant que les requêtes tournent) appliquer le fichier `app-v2.yml` correspondant.
 - Observez la réponse aux requêtes dans le terminal ou avec un graphique adapté dans `graphana` (Il faut configurer correctement le graphique pour observer de façon lisible la transition entre v1 et v2). Un aperçu en image des histogrammes du nombre de requêtes en fonction des versions 1 et 2 est disponible dans chaque dossier de stratégie.
 - supprimez le déploiement+service avec `delete -f` ou dans Lens.

Par exemple pour la stratégie `recreate` le graphique donne:

Argo Rollout : un exemple d'opérateur de rollout (controller) [🔗](#)

Facultatif : Installer Istio pour des scénarios plus avancés [🔗](#)

Pour des scénarios plus avancés de déploiement, on a besoin d'utiliser soit un *service mesh* comme Istio (soit un plugin de rollout comme Argo Rollouts mais pas ce que nous proposons ici).

1. Sur k3s, supprimer la release Helm du Ingress Controller Traefik (ou le ingress Nginx) pour le remplacer par l'ingress Istio.
2. Installer Istio, créer du trafic vers l'ingress de l'exemple et afficher le graphe de résultat dans le dashboard Istio :
<https://istio.io/latest/docs/setup/getting-started/>
3. Utiliser ces deux ressources pour appliquer une stratégie de déploiement de type A/B testing poussée :
 - o <https://istio.io/latest/docs/tasks/traffic-management/request-routing/>
 - o <https://github.com/ContainerSolutions/k8s-deployment-strategies/tree/master/ab-testing>

Formation Kubernetes

TP - PromQL - Déetecter des anomalies sur un Cluster Kubernetes

Dans ce TP nous allons creuser plus loin les requêtes PromQL avec des aggregations, fonctions, calculs dans le contexte de l'observation d'un cluster Kubernetes.

Kubernetes est un peu le Linux du cloud : c'est la solution (ou plutôt le "noyau" de solution) open source pour créer des clouds applicatifs (conteneurs applicatifs). Prometheus est la solution de référence pour monitorer les clouds de conteneurs car il est très adapté aux environnements dynamiques ou les "cibles" de la surveillance vont et viennent.

Nous allons d'abord installer un cluster Kubernetes et l'opérateur `kube-prometheus` qui permet de déployer et d'opérer dans le temps un système Prometheus haute disponibilité.

Nous utiliserons ensuite cette configuration pour essayer des requêtes PromQL plus avancées dans un contexte dynamique.

Installation de Kubernetes [🔗](#)

- Installons les dépendances de Kubernetes telles d'indiquées dans le premier TP kubernetes avec un script: `bash /opt/kubernetes.sh`
- Vérifions l'installation avec `kubectl get nodes`
- Lancez `minikube stop`

- Ouvrez OpenLens dans le menu internet pour vérifier son installation

Pour installer k3s (notre kubernetes pour ce TP) lancez dans un terminal la commande suivante: `curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="-- disable=traefik" sh -`

- Pour récupérer la configuration lancez

- `mkdir -p ~/.kube`
- `sudo chmod 744 /etc/rancher/k3s/k3s.yaml cp /etc/rancher/k3s/k3s.yaml ~/.kube/config`
- vérifier la config avec `kubectl get nodes`

Testons notre installation dans OpenLens en visitant le cluster nommé

`default`

Installation de Kube-Prometheus [🔗](#)

Lancez la commande suivante pour récupérer kube-prometheus dans votre dossier personnel: `cd ~ && git clone https://github.com/prometheus-operator/kube-prometheus.git`

- Assurez vous d'être dans le dossier cloné : `cd ~/kube-prometheus`
- Lancez l'installation de l'opérateur de Prometheus avec le bloc de commande suivant:

```
# Create the namespace and CRDs, and then wait for them to be available
before creating the remaining resources
kubectl create -f manifests/setup

# Wait until the "servicemonitors" CRD is created. The message "No resources
found" means success in this context.
until kubectl get servicemonitors --all-namespaces ; do date; sleep 1; echo "";
done

kubectl create -f manifests/
```

- Observons un peu dans Lens (commentaire formateur) et plus d'information dans les supports Kubernetes sur le même site
- Accédons à l'interface de Prometheus avec la commande `kubectl --namespace monitoring port-forward svc/prometheus-k8s 9999:9090` (laissez la tourner dans le terminal) puis en visitant `localhost:9090` dans le navigateur.

Des requêtes PromQL avancées : identifier les problèmes dans notre Cluster [🔗](#)

Pour chacune des requêtes suivantes:

- Comprendre de quoi elle parle (avec l'aide du formateur si Kubernetes ne vous est pas familier)
- créer les resources Kubernetes requises en faisant `+ > create resource` dans OpenLens puis coller le code et create.
- Décomposer la requête et la jouer partie par partie dans PromLens idéalement ou dans Prometheus pour comprendre le sens des différentes sous parties et leur retours

Trouver le nombre de Pods par namespace [🔗](#)

```
sum by (namespace) (kube_pod_info)
```

Trouver les PersistentVolumeClaims dans l'état “Pending” [🔗](#)

Avant d'exécuter la requête, créez un PersistentVolumeClaim avec la spécification suivante :

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-claim-2
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi

```

Cela restera dans l'état "Pending" car nous n'avons pas de storageClass appelé "manual" dans notre cluster.

Maintenant, exécutez la requête suivante :

```
kube_persistentvolumeclaim_status_phase{phase="Pending"}
```

Trouver les Pods Kubernetes dans un état CrashLoop 🔁

Avant d'exécuter la requête, créez un Pod avec la spécification suivante :

```

apiVersion: v1
kind: Pod
metadata:
  name: crashing-pod
spec:
  containers:
    - name: crashing-container
      image: ubuntu
  restartPolicy: Always

```

Ce pod n'a pas de commande qui dure dans le

Maintenant, exécutez la requête suivante :

```
increase(kube_pod_container_status_restarts_total[15m]) > 3
```

Trouver l'état des différents nœuds du cluster [🔗](#)

- `sum(kube_node_status_condition{condition="Ready", status="true"})`
- `sum(kube_node_status_condition{condition="NotReady", status="true"})`
- `sum(kube_node_spec_unschedulable) by (node)`

Localiser les surallocations CPU (CPU overcommit) [🔗](#)

- Avant d'exécuter la requête, créez un Pod avec la spécification suivante :

```
apiVersion: v1
kind: Pod
metadata:
  name: gros-pod
spec:
  containers:
    - name: gros-pod
      image: alpine
      resources:
        limits: # le pod exige au moins 4 coeurs de CPU et 8Go de ram pour fonctionner
          cpu: "4000m"
          memory: "8000M"
        requests:
          cpu: "4000m"
          memory: "8000M"
      # => il ne sera jamais lancé dans notre cluster trop limité
      # dans prometheus on peut détecter que le cluster est trop petit pour les application demandées (resource overcommitment)
```

Maintenant, exécutez la requête suivante :

```
sum(kube_pod_container_resource_limits{resource="cpu"}) -  
sum(kube_node_status_capacity{resource="cpu"})
```

Si cette requête renvoie une valeur positive, le cluster a surallocé le CPU.

Surallocation de mémoire (Memory Overcommit) [🔗](#)

```
sum(kube_pod_container_resource_limits{resource="memory"}) -  
sum(kube_node_status_capacity{resource="memory"})
```

Trouver les Pods Kubernetes “Unhealthy” [🔗](#)

Avant d'exécuter cette requête, créez un Pod avec la spécification suivante :

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: ssd-pod  
spec:  
  containers:  
    - name: ssd-pod  
      image: alpine  
  nodeSelector:  
    disktype: ssd
```

Ce Pod ne pourra pas s'exécuter car nous n'avons pas de nœud avec l'étiquette `disktype: ssd`.

Maintenant, exéutez la requête suivante :

```
min_over_time(sum by (namespace, pod)  
(kube_pod_status_phase{phase=~"Pending|Unknown|Failed"})[15m:1m]) > 0
```

Trouver le nombre de conteneurs sans limites CPU dans chaque espace de noms [🔗](#)

```
count by (namespace)(sum by (namespace, pod, container)
(kube_pod_container_info{container!=""}) unless sum by (namespace, pod,
container)(kube_pod_container_resource_limits{resource="cpu"}))
```

Trouver les nœuds instables [🔗](#)

Dans cette requête, vous trouverez des nœuds qui passent continuellement de l'état "Ready" à l'état "NotReady" de manière intermittente.

```
sum(changes(kube_node_status_condition{status="true", condition="Ready"})
[15m])) by (node) > 2
```

Si les deux nœuds fonctionnent correctement, vous ne devriez pas obtenir de résultat pour cette requête.

Trouver les cœurs CPU inactifs [🔗](#)

```
sum((rate(container_cpu_usage_seconds_total{container!="POD", container!=""})
[30m]) - on (namespace, pod, container) group_left avg by
(namespace, pod, container)
(kube_pod_container_resource_requests{resource="cpu"})) * -1 >0)
```

Trouver la mémoire inutilisée [🔗](#)

```
sum((container_memory_usage_bytes{container!="POD",container!=""} - on  
(namespace,pod,container) avg by (namespace,pod,container)  
(kube_pod_container_resource_requests{resource="memory"})) * -1 >0 ) /  
(1024*1024*1024)
```

Formation Kubernetes

404

Page not found. Check the URL or try using the search bar.

Formation Kubernetes

404

Page not found. Check the URL or try using the search bar.

Formation Kubernetes

TP optionnel - Installer ArgoCD

ArgoCD est un opérateur d'application qui implémente la méthode de déploiement moderne de GitOps. Il est plutôt agréable et puissant.

Qu'est-ce que le GitOps:

- Présentation : <https://dev.to/thenjdevopsguy/gitops-with-argocd-getting-started-dg>
- Nouveau projet de standardisation de GitOps par la CNCF :
<https://opengitops.dev/>

Les prérequis pour ce TP sont:

- disposer d'un cluster k3s (ou autre cluster mais étapes à adapter)
- du ingress NGINX
- d'une façon de générer des certificats https avec Certmanager.

L'installation:

- Effectuer l'installation avec la première méthode du getting started :
https://argo-cd.readthedocs.io/en/stable/getting_started/
- Il faut maintenant créer l'ingress (reverse proxy) avec une configuration particulière que nous allons expliquer.

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: argocd-server-ingress
  namespace: argocd
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
    kubernetes.io/ingress.class: nginx
    kubernetes.io/tls-acme: "true"
    nginx.ingress.kubernetes.io/ssl-passthrough: "true"
    # If you encounter a redirect loop or are getting a 307 response code
    # then you need to force the nginx ingress to connect to the backend using
    HTTPS.
    #
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
spec:
  tls:
    - hosts:
        - argocd.<stagiaire>.<labdomain>
      secretName: argocd-secret # do not change, this is provided by Argo CD
  rules:
    - host: argocd.<stagiaire>.<labdomain>
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: argocd-server
                port:
                  number: 443

```

- Créez et appliquez cette ressource Ingress.
- Vérifiez dans Lens que l'ingress a bien généré un certificat (cela peut prendre jusqu'à 2 minutes)
- Chargez la page `argocd.<votre sous domaine>` dans un navigateur. exp
`argocd.stagiaire1.kube.dopl.uk`
- Pour se connecter utilisez le login admin et récupérez le mot de passe admin en allant chercher le secret `argocd-initial-admin-secret` dans Lens (Config > Secrets avec le namespace argocd activé).

Formation Kubernetes

TP optionnel - Configurer un user et un projet dans ArgoCD

ArgoCD est donc un opérateur GitOps ou encore un operator / superviseur d'applications qui peut s'étendre à plusieurs équipes et cluster.

La question des identités et droits est centrale dans un environnement "multi-tenant" comme une cohabitation de plusieurs équipes.

Nous allons définir pour gérer cette situation un **projet** ArgoCD et un **utilisateur** ArgoCD ayant le droit d'intervention sur ce projet. L'entité projet peut être gérée via l'interface, via la CLI, ou mieux via les CRDs installés par l'opérateur.

Créer un utilisateur [🔗](#)

L'utilisateur par contre n'est pas une entité unique dans ArgoCD (donc il n'a pas de CRD). On peut le configurer via diverses integrations/SSO notamment via OIDC ou plus simplement en mode **local user**: Un local user est ajouté en modifiant une ConfigMap.

Créez un user en editant la configmap `argocd-cm` (`kubectl edit cm argocd-cm` ou via lens) pour ajouter:

```
...
data:
...
accounts.<votreusername>: apiKey, login
accounts.<votreusername>.enabled: "false"
```

Puis définir un mot de passe:

- login en tant qu'admin avec la CLI -> utiliser le compte admin avec initial admin passwd: `argocd login --port-forward --port-forward-namespace argocd --plaintext`
- Lister les users: `argocd account list --port-forward --port-forward-namespace argocd --plaintext`
- Définir le mot de passe de votre user: `argocd account update-password --account <votreusername> --current-password <initialadminpassword> --new-password <votrepasswd> --port-forward --port-forward-namespace argocd --plaintext`

Pour aller plus loin:

- <https://argo-cd.readthedocs.io/en/stable/operator-manual/user-management/#create-new-user>
- [https://faun.pub/create-argo-cd-local-users-9e830db3763f*](https://faun.pub/create-argo-cd-local-users-9e830db3763f)
- https://www.reddit.com/r/ArgoCD/comments/15nlbs9/is_it_possible_to_create_argocd_accounts_for_the/

Créer un projet via CRD 🔗

Dans Open-lens, cliquer sur + > créer resource pour définir le project nommé `dev-<votrerenom>` (ou autre) avec le code suivant:

```

apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: dev-<votrerenom>
  namespace: argocd
spec:
  sourceRepos:
    - '*' # tous les dépots
    - '!https://github.com/**' # sauf github (juste pour l'exemple). Nous
utiliserons gitlab par la suite
  destinations:
    - namespace: '<votrenamespace>'
      server: 'https://kubernetes.default.svc' # Dans le cluster par défaut
  clusterResourceWhitelist: # toutes les resources peuvent être créées
  - group: '*'
    kind: '*'
  # policies:
  #   # p, elie, applications, get, my-project/*
  #   # p, <votreusername>, applications, *, dev-<votreusername>/*

```

TODO: fix policies syntax !!

(pour plus d'info la doc: <https://argo-cd.readthedocs.io/en/stable/user-guide/projects/>)

Déployer une application depuis Github 🔗

Créez (via Lens ou un fichier avec `kubectl apply`) la resource suivante pour déployer en GitOps notre projet de code monsterstack:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: monsterstack-<votrenom>
  namespace: argocd
spec:
  project: dev-elie
  source:
    repoURL: https://github.com/<yourusername>/<yourmonsterrepo>
    targetRevision: tp_monsterstack_final ## ou autre branche
    path: k8s-deploy-dev/
    directory:
      recurse: true
  destination:
    server: https://kubernetes.default.svc
    namespace: <yournamespace>
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

Lorsque vous déployer cette application via git, argocd poll/vérifie le dépôt toutes les 3 minutes pour se mettre à jour éventuellement avec les modifications.

Pour éviter le poll inutile et diminuer la réactivité des déploiements quand le code change on configure généralement un webhook sur la forge git (ici github). Vous pouvez suivre cette documentation pour ajouter un webhook pour votre application : <https://argo-cd.readthedocs.io/en/stable/operator-manual/webhook/>

Formation Kubernetes

TP optionnel - Installer le chart Wordpress avec ArgoCD

ArgoCD est un opérateur d'application : il ajoute de nouveaux types d'objets (CRDs) dont le type Application qui permet de décrire des ensembles applicatifs gérées via un mode GitOps par ses soins.

Pour l'exemple nous allons créer une nouvelle application `wordpress-argo` avec le manifeste suivant à compléter:

```

apiVersion: v1
kind: Namespace
metadata:
  name: wordpress-argo
---
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: wordpress-argo
  namespace: argocd
spec:
  destination:
    namespace: wordpress-argo
    server: https://kubernetes.default.svc
    project: default
  source:
    repoURL: https://charts.bitnami.com/bitnami
    chart: wordpress
    targetRevision: 15.2.49
    helm:
      values: |
        wordpressUsername: <stagiaire> # replace
        wordpressPassword: myunsecurepassword
        wordpressBlogName: Kubernetes example blog

    replicaCount: 1

  service:
    type: ClusterIP

  ingress:
    enabled: true
    hostname: wordpressargo.<stagiaire>.<labdomain> # replace with your
    hostname pointing on the cluster ingress loadbalancer IP
    tls: true
    certManager: true
    annotations:
      cert-manager.io/cluster-issuer: letsencrypt-prod
      kubernetes.io/ingress.class: nginx

```

- Vous pouvez l'ajouter avec `kubectl` ou même directement en cliquant sur + dans OpenLens.
- Visitez l'interface de ArgoCD et cliquer sur sync pour déployer effectivement l'application.

Dans une gestion GitOps de l'infrastructure ce code d'application sera **ajouté à un dépôt git et la synchronisation mise en mode automatique**

à chaque nouveau commit sur la branche adéquate du dépôt.

Formation Kubernetes

TP optionnel - CI/CD GitOps avec ArgoCD

Refactoring in progress... à prendre comme des pistes de travail.

Déploiement de l'application dans argoCD

Expliquons un peu le reste du projet.

- Créez un token de déploiement dans `Gitlab > Settings > Repository > Deploy Tokens`. Ce token va nous permettre de donner l'autorisation à ArgoCD de lire le dépôt gitlab (facultatif si le dépôt est public cela ne devrait pas être nécessaire). Complétez ensuite **2 fois** le token dans le fichier `k8s/argocd-apps.yaml` comme suit : `https://<nom_token>:<motdepasse_token>@gitlab.com/<votre depo>.git` dans les deux sections `repoURL` des deux applications.
- Créer les deux applications `monstericon-dev` et `monstericon-prod` dans `argocd` avec `kubectl apply -f k8s/argocd-apps.yaml`.
- Allons voir dans l'interface d'ArgoCD pour vérifier que les applications se déploient bien sauf le conteneur `monstericon` dont l'image n'a pas encore été buildée avec le bon tag. Pour cela il va falloir que notre pipeline s'execute complètement.

Les deux étapes de déploiement (dev et prod) du pipeline nécessitent de pousser automatiquement le code du projet à nouveau pour déclencher le

redéploiement automatique dans ArgoCD (en mode pull depuis gitlab). Pour cela nous avons besoin de créer également un token utilisateur:

- Allez dans Gitlab > User Settings (en haut à droite dans votre profil) > Access Tokens et créer un token avec `read_repository write_repository read_registry write_registry` activés. Sauvegardez le token dans un fichier.
- Allez dans Gitlab > Settings > CI/CD > Variables pour créer deux variables de pipelines: `CI_USERNAME` contenant votre nom d'utilisateur gitlab et `CI_PUSH_TOKEN` contenant le token précédent. Ces variables de pipelines nous permettent de garder le token secret dans gitlab et de l'ajouter automatiquement aux pipeline pour pouvoir autoriser la connexion au dépôt depuis le pipeline (git push).
- Nous allons maintenant tester si le pipeline s'exécute correctement en commitant et poussant à nouveau le code avec `git push gitlab`.
- Debuggons les pipelines s'ils sont en échec.
- Allons voir dans ArgoCD pour voir si l'application dev a été déployée correctement. Regardez la section `events` et `logs` des pods si nécessaire.
- Une fois l'application dev complètement healthy (des coeurs verts partout). On peut visiter l'application en mode dev à l'adresse `https://monster-dev.<votre_sous_domaine>`.
- On peut ensuite déclencher le stage `deploy-prod` manuellement dans le pipeline, vérifier que l'application est healthy dans ArgoCD (debugger sinon) puis visiter `https://monster.<votre_sous_domaine>`.

Idées d'amélioration

- Déplacer le code de déploiement dans un autre dépôt que le code d'infrastructure. Le pipeline de devra cloner le dépôt d'infrastructure, templater avec kustomize la bonne version de l'image dans le bon environnement. Pousser le code d'infrastructure sur le dépôt

d'infrastructure. Corriger l'application ArgoCD pour monitoner le dépôt d'infrastructure.

- Mutualiser le code de déploiement k8s avec des overlays kustomize
- Utiliser une stragégie de blue/green ou A/B déploiement avec Argo Rollouts ou Istio avec vérification de réussite du déploiement et rollback en cas d'échec.
- Ajouter plus d'étapes réalistes de CI/CD en se basant par exemple sur le livre GitOps suivant.
- Gérer la création des ressources gitlab automatiquement avec Terraform et gérer les secrets (tokens gitlab) consciencieusement.

Bibliographie

- 2021 - GitOps and Kubernetes Continuous Deployment with Argo CD, Jenkins X, and Flux
- Billy Yuen, Alexander Matyushentsev, Todd Ekenstam, Jesse Suen (z-lib.org)

Formation Kubernetes

Introduction - Les différents types de cluster - avancé

Kubernetes

- Kubernetes est une solution d'orchestration de conteneurs extrêmement populaire.
- Le projet est très ambitieux : une façon de considérer son ampleur est de voir Kubernetes comme un système d'exploitation (et un standard ouvert) pour les applications distribuées et le cloud.
- Le projet est développé en Open Source au sein de la Cloud Native Computing Foundation.

Architecture de Kubernetes

Kubernetes a une architecture master/worker ce qui signifie que certains certains noeuds du cluster contrôlent l'exécution par les autres noeuds de logiciels ou jobs.

Les noeuds Kubernetes

Les nœuds d'un cluster sont les machines (serveurs physiques, machines virtuelles, etc. généralement Linux mais plus nécessairement aujourd'hui) qui exécutent vos applications et vos workflows. Tous les noeuds d'un cluster qui ont besoin de faire tourner des tâches (workloads) kubernetes utilisent trois services de base:

- Comme les workloads sont généralement conteneurisés chaque noeud doit avoir une *runtime de conteneur* compatible avec la norme Container Runtime Interface (CRI) : containerd,cri-o OU Docker n'est pas la plus recommandée bien que la plus connue. Il peut également s'agir d'une runtime utilisant de la virtualisation.
- Le kubelet composant (binaire en go, le seul composant jamais conteneurisé) qui contrôle la création et l'état des pods/conteneur sur son noeud.
- D'autres composants et drivers pour fournir fonctionnalités réseau (Container Network Interface - CNI et kube-proxy) ainsi que fonctionnalités de stockage (Drivers Container Storage Interface (CSI))

Pour utiliser Kubernetes, on définit un état souhaité en créant des ressources (pods/conteneurs, volumes, permissions etc). Cet état souhaité et son application est géré par le control plane composé des noeuds master.

Les noeuds master kubernetes forment le control plane du Cluster 🔒

Le control plane est responsable du maintien de l'état souhaité des différents éléments de votre cluster. Lorsque vous interagissez avec Kubernetes, par exemple en utilisant l'interface en ligne de commande `kubectl`, vous communiquez avec les noeuds master de votre cluster (plus précisément l'`API Server`).

Le control plane conserve un enregistrement de tous les objets Kubernetes du système. À tout moment, des boucles de contrôle s'efforcent de faire converger l'état réel de tous les objets du système pour correspondre à l'état souhaité que vous avez fourni. Pour gérer l'état réel de ces objets sous forme de conteneurs (toujours) avec leur configuration le control plane envoie des instructions aux différents kubelets des noeuds.

Donc concrètement les noeuds du control plane Kubernetes font tourner, en plus de `kubelet` et `kube-proxy`, un ensemble de services de contrôle:

- `kube-apiserver`: expose l'API (rest) kubernetes, point d'entrée central pour la communication interne (intercomposants) et externe (kubectl ou autre) au cluster.
- `kube-controller-manager`: contrôle en permanence l'état des resources et essaie de le corriger s'il n'est plus conforme.
- `kube-scheduler`: surveille et cartographie les resources matérielles et les contraintes de placement des pods sur les différents noeuds pour décider ou doivent être créés ou supprimés les conteneurs/pods.
- `cloud-controller-manager`: Composant *facultatif* qui gère l'intégration avec le fournisseur de cloud comme par exemple la création automatique de loadbalancers pour exposer les applications kubernetes à l'extérieur du cluster.

L'ensemble de la configuration kubernetes est stockée de façon résiliante (consistance + haute disponibilité) dans un gestionnaire configuration distribué qui est généralement `etcd`.

`etcd` peut être installé de façon redondante sur les noeuds du control plane ou configuré comme un système externe sur un autre ensemble de serveurs.

Lien vers la documentation pour plus de détails sur les composants :

<https://kubernetes.io/docs/concepts/overview/components/>

Formation Kubernetes

Cours optionnel - Principes de conception et architecture détaillée de Kubernetes

Le concept de configuration déclarative est l'un des principaux fondements du développement de Kubernetes : un utilisateur déclare un état souhaité du système pour produire un résultat.

Exemple : "Je veux qu'il y ait 3 instances de mon serveur en fonctionnement à tout moment". Kubernetes, prend cette information déclarative et se charge de la rendre effective.

La puissance de cette approche déclarative (qui est plus complexe au départ que l'approche impérative) est qu'en donnant au système une déclaration de l'état que vous souhaitez plutôt qu'une série d'instructions de bas niveau vous lui permettez de prendre des mesures autonomes. Il peut mettre en œuvre des comportements automatiques pour se réparer lui-même sans vous réveiller au milieu de la nuit.

Et ce cadre déclaratif de haut niveau peut servir de base à une automatisation extrêmement dynamique de nombreuses tâches opérationnelles grâce aux opérateurs.

Contrôleurs et boucles de réconciliation 🔗

Kubernetes est basé sur une approche décentralisée (plutôt que monolithique) : Au lieu d'un unique contrôleur monolithique, il est composé d'un grand

nombre de contrôleurs, chacun effectuant sa propre boucle de réconciliation indépendante pour autocorriger l'état du système.

- Chaque boucle individuelle n'est responsable que d'une petite partie du système (par exemple, la mise à jour de la liste des endpoints pour une ressource Service/Endpoint particulière)
- Chaque contrôleur ignore l'état global du système ce qui rend l'ensemble beaucoup plus stable : chaque contrôleur n'est pas affecté par des problèmes ou des changements sans rapport direct avec lui.
- Un inconvénient de cette approche distribuée est que le comportement global peut être plus difficile à comprendre, car il n'y a pas d'endroit unique où chercher une explication : il est nécessaire d'examiner le fonctionnement coordonné d'un grand nombre de processus indépendants.

Une boucle de réconciliation répète sans arrêt les étapes suivante :

1. Obtenir l'état souhaité.
2. Observer le système.
3. Trouver les différences entre l'observation et l'état souhaité.
4. Prendre des mesures pour que le système corresponde à l'état souhaité.

Concrètement : "Je veux trois instances de ce serveur " => Le contrôleur de réPLICATION (des replicatsets) prend cet état souhaité observe ensuite le système => il y a actuellement trois instances du conteneur de serveur => Le contrôleur constate la différence entre l'état actuel et l'état souhaité (un serveur manquant) => il prend des mesures pour que l'état actuel corresponde en créant un quatrième conteneur de serveur

Regroupement dynamique par labels

Le défi dans cette configuration de contrôleurs dynamiques et distribués est de déterminer l'ensemble des ressources auxquelles la boucle de réconciliation doit

prêter attention. Lorsqu'il s'agit de regrouper des éléments dans un ensemble deux approches possibles : le regroupement explicite/statique ou implicite/dynamique.

Avec le regroupement explicite chaque groupe est défini par une liste concrète "Les membres du groupe sont A, B, et C". Mais cette méthode ne peut pas répondre facilement à un système qui change de façon dynamique.

Avec les groupes implicites, au lieu d'une liste de membres, le groupe est défini par une déclaration telle que "Les membres du groupe sont les personnes qui portent des vêtements en coton". Ils sont implicites récupérés en évaluant la définition du groupe par rapport à un ensemble de personnes présentes.

Comme l'ensemble des objets du groupe peut toujours changer, la composition du groupe est dynamique et changeante. Bien que cela puisse introduire de la complexité de devoir évaluer à chaque fois les membres, cette méthode est beaucoup plus flexible et stable dans un environnement changeant sans nécessiter d'ajustements constants.

Dans Kubernetes chaque objet d'API peut avoir un nombre arbitraire de "labels" clé/valeur qui sont associés aux objets d'API. Vous pouvez ensuite utiliser une requête ou un sélecteur de labels pour identifier un ensemble logique d'objets correspondant à cette requête.

- `kubectl get service --selector='app.kubernetes.io/name=monapp'`
- `kubectl get node --selector='!node-role.kubernetes.io/master'`

La documentation suggère aussi un modèle de labels standard pour vos applications pour favoriser l'interopérabilité avec plusieurs outils:

<https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/>

Pour écrire des chart Helm il est également important (bien que facultatif) de respecter son standard d'étiquettes.

Retour détaillé sur les composants du Control Plane



Kubernetes adhère à la philosophie (Unix) de modularité de petits composants qui font bien leur travail. Kubernetes n'est pas une application d'un seul tenant qui implémente les diverses fonctionnalités du système dans un seul binaire. Il s'agit plutôt d'une collection de différentes applications qui travaillent toutes ensemble, en grande partie en s'ignorant les unes les autres, pour mettre en œuvre le système global.

Même lorsqu'il existe un binaire (par exemple, le `controller manager`) qui regroupe un grand nombre de fonctions différentes, ces fonctions sont maintenues indépendantes les unes des autres dans ce binaire. Elles sont compilées ensemble pour faciliter la tâche de déploiement et de gestion réseaux de Kubernetes.

etcd Link

Le composant etcd est au cœur de tout cluster Kubernetes. Il s'agit d'un `key/value store` où tous les objets d'un cluster Kubernetes sont conservés.

Les serveurs etcd utilisent un algorithme de `consensus distribué Raft` qui garantit que même si un des serveurs tombe en panne, la réPLICATION est suffisante pour maintenir les données stockées et pour récupérer automatiquement les données lorsqu'un serveur etcd redevient sain et se réintègre au cluster.

Les serveurs etcd fournissent également deux autres fonctionnalités importantes dont Kubernetes fait un usage intensif:

- La `concurrence optimiste`. Chaque valeur stockée dans etcd a une version de ressource correspondante. Lorsqu'une paire clé-valeur est écrite sur un serveur etcd, elle peut être conditionnée à une version de ressource particulière. Cela signifie qu'en utilisant etcd, vous pouvez

implémenter le principe de `compare and swap`, qui est au cœur de tout système de concurrence. La comparaison et l'échange permettent à un utilisateur de lire une valeur et de la mettre à jour en sachant qu'aucun autre composant du système n'a également mis à jour cette valeur. Ces assurances permettent au système d'avoir en toute sécurité plusieurs threads manipulant des données dans etcd sans avoir besoin de recourir à des `pessimistic locks`, ce qui peut réduire de manière significative le débit du serveur. L'idée qui sous-tend la concurrence optimiste est la capacité d'effectuer la plupart des opérations sans utiliser de verrous (concurrence pessimiste) et, au lieu de cela, de détecter l'apparition d'une écriture concurrente et de rejeter la dernière des deux écritures concurrentes.

- Le protocole `watch`. L'intérêt de `watch` est qu'il permet aux clients de surveiller efficacement les changements dans le `key/value store` pour un grand ensemble de valeurs. Par exemple, tous les objets objets d'un `Namespace` sont stockés dans un répertoire dans etcd. L'utilisation d'une veille permet à un client d'attendre et de réagir efficacement aux changements dans ce `Namespace` sans avoir à interroger en permanence le serveur etcd.

L'API comme gateway pour toutes les interactions

Un autre concept structurel central de Kubernetes est que toutes les interactions entre les composants sont pilotées à travers une API centralisée : l'API que les composants utilisent est exactement la même API que celle utilisée par tous les autres utilisateurs du cluster:

- Donc aucune partie du système n'est plus privilégiée et aucun composant n'a accès à directement aux données internes.
- Et aussi chaque composant peut être remplacé pour une implémentation alternative sans avoir à reconstruire les composants

de base. Même le `Scheduler` peut être échangé ou simplement augmenté par des implémentations alternatives.

Le serveur API implémente une API RESTful en HTTP, exécute toutes les opérations d'API et est responsable du stockage des objets d'API dans un backend de stockage persistant, généralement etcd.

Il sert de médiateur pour toutes les interactions entre les clients et les objets API stockés dans etcd. Par conséquent, il est le point de rencontre central de tous les différents composants.

Kubernetes API et extension par APIgroups [🔗](#)

Tous les types de resources Kubernetes correspondent à un morceau (un sous arbre) d'API REST de Kubernetes. Ces chemins d'API pour chaque ressources sont classés par groupe qu'on appelle des `apiGroups`:

- On peut lister les resources et leur groupes d'API avec la commande `kubectl api-resources -o wide`.
- Ces groups correspondent aux préfixes indiqué dans la section `apiVersion` des descriptions de ressources.
- Ces groupes d'API sont versionnés sémantiquement et classés en `alpha` `beta` et `stable`. `beta` indique déjà un bon niveau de stabilité et d'utilisabilité et beaucoup de ressources officielles de kubernetes ne sont pas encore en `api stable`. Exemple: les `CronJobs` viennent de se stabiliser au début 2021.
- N'importe qui peut développer ses propres types de resources appelées `CustomResourceDefinition` (voir ci dessous) et créer un `apiGroup` pour les ranger.

Documentation: <https://kubernetes.io/docs/reference/using-api/>

Les routes web de et la découverte de l'API [🔗](#)

L'API a des routes systématiques pour les ressources :

Voici les composants des deux chemins différents pour les types de ressources namespaced :

- /api/v1/namespaces/<namespace-name>/<resource-type-name>/<resource-name>.
- /apis/<api-group>/<api-version>/namespaces/<namespace-name>/<resource-type-name>/<resource-name>.

Voici les composants des deux chemins différents pour les types de ressources sans namespace :

- /api/v1/<nom-de-la-ressource>/<nom-de-la-ressource>
- /apis/<api-group>/<api-version>/<resource-type-name>/<resource-name>

Maintenant, nous aimerais explorer l'API et voir comment elle donne des informations pour sa propre exploration. Pour explorer une API REST, l'outil classique `curl` peut suffire.

Pour faciliter l'exploration du serveur API, exécutez l'outil `kubectl` en mode proxy pour exposer un serveur API non authentifié sur `localhost:8001` en utilisant la commande suivante : `kubectl proxy`

Ensuite pour voir comment l'API vous donne de nombreuses informations comme toutes les ressources dans chaque `api-group` vous pouvez lancer : `curl localhost:8001/api` OU `curl localhost:8001/api/v1`

Pour accéder à la page web pour l'ensemble des spécifications de l'api (comme tout point de terminaison swagger/openAPI), ouvrez dans un navigateur : `localhost:8001/openapi/v2`

Vie d'une requête [🔗](#)

Pour mieux comprendre ce que fait le serveur d'API pour chacune de ces différentes demandes, nous allons décomposer et décrire le traitement d'une seule demande au serveur d'API.

Authentification [🔗](#)

La première étape du traitement des demandes est l'authentification, qui établit l'identité associée à la demande. Le serveur API prend en charge plusieurs modes d'authentification : certificat x509, Token ou backend d'identité principalement. voir cours sécurité.

RBAC/Autorisation [🔗](#)

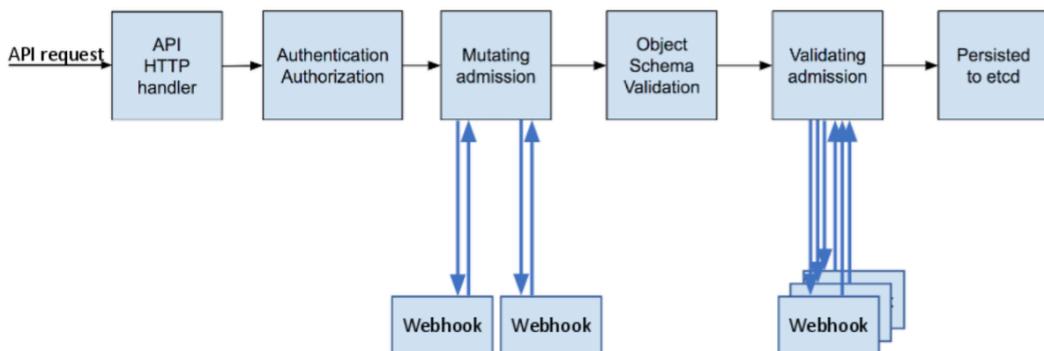
Après que le serveur API ait déterminé l'identité d'une requête, il passe à l'autorisation de celle-ci avec RBAC

Contrôle d'admission [🔗](#)

Une fois qu'une demande a été authentifiée et autorisée, elle passe au contrôle d'admission. Le contrôle d'admission détermine si la demande est bien formée (syntaxe) et applique éventuellement des modifications à la demande avant qu'elle ne soit traitée. On peut ajouter des plugin pour l'admission

Si un contrôleur d'admission trouve une erreur, la requête est rejetée. Si la demande est acceptée, la demande transformée est utilisée à la place de la demande initiale. Les contrôleurs d'admission sont appelés en série, chacun recevant le résultat du précédent.

C'est un mécanisme très général utilisé par exemple pour ajouter des valeurs par défaut aux objets, appliquer une pod security policy etc.



- <https://kubernetes.io/blog/2019/03/21/a-guide-to-kubernetes-admission-controllers/>

Requêtes spécialisées [🔗](#)

En plus des requêtes RESTful standard, le serveur API dispose d'un certain nombre de modèles de requête spécialisés qui fournissent des fonctionnalités étendues aux clients :

`/proxy, /exec, /attach, /logs.`

La première catégorie importante d'opérations est constituée par les connexions ouvertes, de longue durée, au serveur d'API. Ces requêtes fournissent des données en continu plutôt que des réponses immédiates.

`_logs` est la demande de streaming la plus facile à comprendre car elle laisse simplement la demande ouverte et fournit plus de données en continu. Les autres opérations tirent parti du protocole WebSocket pour la diffusion bidirectionnelle des données.

En plus de ces flux, le serveur API Kubernetes introduit en fait un protocole de flux multiplexé supplémentaire. La raison en est que, pour de nombreux cas d'utilisation, il est très utile que le serveur API soit capable de gérer plusieurs flux d'octets indépendants. Prenons, par exemple, l'exécution d'une commande dans un conteneur. Dans ce cas, il y a en fait trois flux qui doivent être gérés (`stdin`, `stderr` et `stdout`).

Opérations Watch [🔗](#)

En plus des données en continu, le serveur API prend en charge une API pour surveiller les ressources. Ainsi, au lieu d'interroger l'API à intervalle régulier pour détecter d'éventuelles mises à jour, ce qui introduit soit une charge/latence supplémentaire, le mode `watch` permet à un utilisateur d'obtenir des mises à jour à faible latence avec une seule connexion en ajoutant le paramètre de requête `?watch=true`.

Boucle de contrôle de l'API pour les CRD [🔗](#)

Les définitions de ressources personnalisées (CRD) sont des objets API dynamiques qui peuvent être ajoutés à un serveur API en cours

d'exécution. Étant donné que l'acte de création d'une CRD crée de nouveaux chemins HTTP que le serveur d'API doit savoir comment ajouter ces chemins et implémente pour cela un contrôleur avec un boucle de contrôle.

Journaux d'audit [🔗](#)

`/var/log/kubernetes/kube-apiserver.log` traditionnellement.

Les journaux d'audit sont destinés à permettre à un administrateur de serveur de récupérer "forensically" l'état du serveur et la série d'interactions avec les clients qui ont abouti à l'état actuel des données dans l'API Kubernetes. Par exemple, il permet à un utilisateur de répondre à des questions telles que : "Pourquoi ce ReplicaSet a-t-il été mis à l'échelle à 100 ? ", " Qui a supprimé ce Pod ? ", entre autres. Les journaux d'audit peuvent être poussés dans un backend spécifique (SIEM ?) pour stockage et analyse

En utilisant `--v` sur le serveur API, vous pouvez ajuster le niveau de verbosité de la journalisation (defaut `--v=2`).

En plus du débogage du serveur d'API via les journaux, il est également possible de déboguer les interactions avec le serveur d'API, via l'outil de ligne de commande `kubectl`. Comme le serveur d'API, l'outil de ligne de commande `kubectl` enregistre les journaux via le paquet github.com/golang/glog et prend en charge l'indicateur de verbosité `--v`. Définir la verbosité au niveau 10 (`--v=10`)

Scheduler : assigner des pods aux noeuds [🔗](#)

Avec etcd et le serveur API fonctionnant correctement, un cluster Kubernetes est, d'une certaine manière, fonctionnellement complet. Avec seulement ces deux composants, vous pouvez créer tous les différents objets de l'API, tels que Deployments et Pods. Cependant, en testant cela on peut se rendre compte que les pods ne sont jamais exécutés.

En effet, trouver un emplacement pour l'exécution d'un Pod est le travail du scheduler de Kubernetes. Il scanne le serveur d'API à la recherche de Pods non programmés et détermine ensuite les meilleurs nœuds sur lesquels les exécuter.

Le scheduler va pour cela cartographier en temps réel les ressources encore disponible sur les noeuds, les resources demandées par les pods, les étiquettes de caractéristiques des noeuds appelées `Taints`, et autres contraintes géographiques des pods comme les `Affinity`, `Anti-affinity` ainsi que contraintes de volume etc.

`Controller Manager` : les boucles de réconciliations

Une fois qu'`etcd`, le serveur API et le `scheduler` sont opérationnels, vous pouvez créer des pods et les voir positionnés et exécutés sur les nœuds, mais vous constaterez que les `ReplicaSets`, les `Deployment` et `Services` ne fonctionnent pas . Cela est dû au fait que toutes les boucles de contrôle de réconciliation nécessaires à leur mise en œuvre ne sont pas actuellement en cours d'exécution. C'est le travail du `Controller Manager`. Il regroupe de nombreuses boucles de contrôle/réconciliation différentes nécessaires au fonctionnement de la plupart des objects auto-piloté/auto-réparant (`selfhealing`) ou simplement dynamiques du cluster.

Composants des nœuds `worker`

Kubelet

Le `kubelet` est l'agent de base pour tous les serveur d'un cluster Kubernetes qui doivent exécuter des conteneurs. Il est nécessaire installé directement sous forme de binaire car il est lui-même en charge des conteneurs. Il est l'élément qui relie le CPU, le disque et la mémoire disponibles d'un nœud

au cluster : il communique avec le serveur API pour définir et démarrer les conteneurs qui à exécuter sur son nœud.

Le kubelet communique également l'état de ces conteneurs au serveur API afin que d'autres boucles de contrôle de réconciliation puissent observer leur état à tout moment. Il est également en charge de la vérification de l'état de santé et du redémarrage des conteneurs.

Comme il est assez inefficace d'envoyer toutes les informations sur l'état de santé vers le serveur API, le kubelet court-circuite cette interaction API et exécute lui-même la boucle de réconciliation. Ainsi, si un conteneur exécuté par le kubelet s'arrête ou échoue à son `healthcheck`, le kubelet le redémarre tout en communiquant cet état de santé (et le redémarrage) à l'API.

kube-proxy et KubeDNS [🔗](#)

Voir la partie réseau avancé.

Kubernetes Metric Server remplacement de heapster [🔗](#)

Il s'agit d'un autre composant nécessairement binaire (non conteneurisé) qui est chargé de collecter des mesures telles que l'utilisation du CPU, du réseau et du disque de tous les conteneurs s'exécutant dans le cluster Kubernetes. Son objectif est d'être plus réactif que le monitoring classique en mode pull et de fournir les données de base pour gérer la mise à l'échelle automatique des Pods (pour alimenter la boucle de réconciliation de l'autoscaler `HorizontalPodAutoscaler`). Cet objet peut, par exemple, automatiquement augmenter la taille d'un `Deployment` lorsque l'utilisation du CPU des containers du déploiement dépasse 80 %.

Formation Kubernetes

TP optionnel - Ajouter une identité via certificat et configurer le RBAC

Authentification et authorization [🔗](#)

La gestion des identités est très flexible et configurable dans kubernetes comprendre: il n'y a pas de façon standard et automatique de gérer des utilisateurs mais de multiples mécanismes d'authentification. Il revient à chaque équipe/solution basée sur kubernetes d'implémenter quelque chose de satisfaisant.

- doc officielle: <https://kubernetes.io/docs/reference/access-authn-authz/authentication>

La façon manuelle la plus simple et directe pour créer des identités est de générer des certificats client avec la fonctionnalité d'approbation de certificat des cluster k8s. Pour cela comme dans toute PKI il va nous falloir:

1. générer une clé privée et une CertificateSigningRequest puis
2. demander au cluster de l'approuver et générer le certificat correspondant
3. le télécharger
4. ensuite nous pourrons ajouter ce nouveau certificat client à notre kubeconfig: voilà notre identité. mais elle n'a aucun droit donc..
5. il faut créer une matrice de droit (Role)
6. Associer cette matrice de droits a notre "User" i.e. le Common Name de notre certificat client
7. vérifier avec `kubectl can-i` ou autre que nous avons bien les droits requis

Générer la clé privée et la CSR

- Choisir un nom à utiliser pour le Common Name de la CSR et pour les fichiers/champs user: par exemple `votreprenomnom`
- créer un dossier `tp_auth` et aller dedans.
- créer la clé privée: `openssl genrsa -out votreprenomnom.key 4096`
- créer la CSR `openssl req -new -key votreprenomnom.key -out votreprenomnom.csr` : la seule question importante ici est de bien mettre `votreprenomnom` à la question Common Name.

Créer la resource Kubernetes CSR

- Convertir la CSR en base64 (bin to txt) pour k8s : `cat votreprenomnom.csr | base64 | tr -d "\n"`
- Créer la ressource suivante en remplaçant avec la valeur base64.

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: votreprenomnom
spec:
  groups:
    - system:authenticated
  signerName: kubernetes.io/kube-apiserver-client
  usages:
    - client auth
  request: <crs_base64 from previous cmd>
```

- Approuver la CSR pour créer le certificat client : `kubectl certificate approve votreprenomnom`
- Télécharger le certificat présent dans la resource CSR : `kubectl get csr votreprenomnom -o jsonpath='{.status.certificate}' | base64 --decode > votreprenomnom-client-certificate.crt`

Compléter le fichier de connexion kubeconfig [🔗](#)

- affichez la version base64 du client-certificate-data : `cat votreprenomnom.key | base64 | tr -d "\n"`
- afficher la version base64 du client-key-data: `cat votreprenomnom-client-certificate.crt | base64 | tr -d "\n"`
- Ajoutez un item dans la section `users` du kubeconfig i.e. : les 4 lignes suivantes en complétant par les valeurs précédentes:

```
- name: votreprenomnom
  user:
    client-certificate-data: <CLIENT-CRT-DATA>
    client-key-data: <CLIENT-KEY-DATA>
```

- Ajoutez un contexte de connexion dans la section `contexts`:

```
- context:
  cluster: <CLUSTER-NAME>
  user: votreprenomnom
  name: votreprenomnom@<CLUSTER-NAME> # ou whatever
```

Configuration RBAC [🔗](#)

Basé sur cette identité, nous allons définir un role correspondant à une fonction : imaginons que notre identité soit celle d'un.e développeur qui aurait le droit de tout faire dans le namespace dev mais seulement le droit de consulter la configuration de l'application et les logs sur la prod.

- `kubectl create namespace dev`
- `kubectl create namespace prod`

Créons les roles (chacun dans leur namespace):

- `kubectl create role developer --verb="" --resource="" -n dev`
- `kubectl create role prod-observer --verb="get,list" --resource="pod,pods/logs,deploy,svc,ing,sts" -n prod`

Assignons ces rôle à notre identité (dans le bon namespace...)

- `kubectl create rolebinding developer --role=developer --user=votreprenomnom -n dev`
- `kubectl create rolebinding prod-observer --role=prod-observer --user=votreprenomnom -n prod`

Vérifions nos droits

La sous commande `auth can-i` permet sans rien appliquer de vérifier qu'on a le droit d'effectuer un type de requête spécifique sur l'API. On peut aussi activer un impersonification (une sorte de sudo et ce pour toutes les commandes) avec `--as=nom` :

- `kubectl auth can-i get pods/logs --as=votreprenom -n prod`
- changeons le contexte de kubectl:
- `kubectx OU ...`
- `kubectl config get-contexts puis kubectl config use-context lecontexte`

Testons avec de vrais commandes:

- créer un deployment dans `dev`: `kubectl create deploy nginx --image=nginx -n dev`
- créer un deployment dans `prod`: `kubectl create deploy nginx --image=nginx -n prod`

Aller plus loin

- <https://johnharris.io/2019/08/least-privilege-in-kubernetes-using-impersonation/>

Formation Kubernetes

Cours optionnel - Réseau Kubernetes avancé

Le réseau Pod-to-Pod et les CNI plugins [🔗](#)

La base du réseau Kubernetes consiste dans les communication entre les pods car tout est pod dans un cluster (même les éléments réseau comme les ingress et souvent le kube-proxy)

Sous Linux les conteneurs (quelle que soit la runtime choisie, Docker, containerd etc) tirent partie d'une fonctionnalité du noyau appelée Network namespace c'est à dire des espaces réseau isolés du réseau principal du serveur grâce à une forme de virtualisation. On peut s'en rendre compte en créant soit même des interfaces virtuelle dans un namespace (cf <https://linuxhint.com/use-linux-network-namespace/>).

Les pods sont composés de plusieurs conteneurs qui partagent le même namespace réseau. Au moment de la création d'un pod, le premier conteneur créé est appelé pause, il est vide mais sert de support à la création de l'interface réseau virtuelle pour le pod. Les conteneurs démarrés ensuite récupèrent cette interface commune. On peut constater la création de ce conteneur en utilisant la cli docker ou ctr.

Dans Kubernetes cette interface virtuelle de chaque pod est créée et routée automatiquement par le plugin CNI (Container Network Interface, une norme standard qui permet à toutes les runtime de conteneurs et tous les plugins d'être compatibles.).

Ce plugin doit être présent dans le cluster pour pouvoir automatiser la communication entre les pods. À l'installation (manuelle) du cluster on configure donc un CIDR pour les pods généralement un /16, et un plugin CNI.

Il existe de nombreux plugin CNI du plus simple au plus sophistiqué pour pouvoir répondre à des edge cases (notamment la communication interne/externe du cluster) des problématiques de performance, sécurité, observabilité. Voir plus loin.

On peut même implémenter un plugin simple en bash a des fins pédagogiques : <https://www.altoros.com/blog/kubernetes-networking-writing-your-own-simple-cni-plug-in-with-bash/>.

Une fois cette configuration mise en place les pods sont capables de communiquer avec tous les autres pods du cluster sans NAT (à leur niveau), c'est à dire sans modification de l'adresse IP au cours de la communication.

On donc peut tester la base du fonctionnement d'un plugin CNI installé en essayant de pinguer depuis un pod l'IP d'un autre pod sur un autre noeud.

<https://projectcalico.docs.tigera.io/about/about-k8s-networking>

Une chaîne vidéo géniale pour aller en profondeur dans le réseau k8s

Pleins de schémas. Des principes fondamentaux et une comparaison des différents plugin CNI

- <https://www.youtube.com/@TheLearningChannel-Tech/videos>

Communication inter-services avec le kube-proxy

Une fois que l'on a mis en place un moyen de communication inter-pod la communication est-ouest de notre cluster n'est pas encore opérationnelle. En effet Kubernetes étant un environnement dynamique avec des pods créés et détruit automatiquement par des Deployment ou autre controller, gérer les IP des pods manuellement est en réalité impraticable. Pour cela nous avons besoin de la gestion automatique d'une ip virtuelle associée fournissant le loadbalancing proposé par les resources de type Service.

Cette gestion d'IP virtuelle est réalisée traditionnellement par le composant Kubernetes appelé `kube-proxy`. Le `kube-proxy` est généralement installé dans le cluster en temps que pod privilégié sur chaque noeud (mais peut également être un service `systemd`) et manipule `iptables`. On peut s'en rendre compte par exemple avec la commande

```
sudo iptables-save | grep KUBE | grep "kubernetes-dashboard" # ou autre nom de service.
```

Ainsi, `kube-proxy` créé dynamiquement des règles `iptables` qui indiquent que tout paquet venant du CIDR des pods et à destination de l'IP virtuelle du service (en réalité l'IP de l'objet endpoint) doit être redirigé aléatoirement vers l'un des pods de backend d'un groupe désignés comme nous l'avons vu grâce à un selecteur de labels.

`kube-proxy` peut également être configuré pour utiliser à la place de `iptables` la fonctionnalité d'IP virtuelle du noyau Linux appelé `IPVS`. Cela amène surtout de meilleures performances dans le cas d'un grand nombre de services (<500).

Enfin depuis quelques temps la gestion du traffic inter-node avec `kube-proxy` peut être remplacée par une implémentation eBPF dans le noyau linux par exemple grâce à Calico ou Cilium.

Video: [Kubernetes Services networking](#)

Découverte de service avec kube-DNS

Les services viennent avec un nom de domaine local au namespace et local grâce à au composant `kube-dns`. Ce composant peut depuis longtemps être remplacé par coreDNS plus performant et modulaire (recommandé).

On peut tester le bon fonctionnement du composant DNS avec `nslookup` depuis par exemple un pods du conteneur `dnsutils`.

Deux types de services moins connus :

- service de type `headless` avec `clusterIP=None` -> une requête vers le nom de domaine du service renvoie la liste des IP des pods backend. On peut ensuite implémenter soit même d'une manière ou d'une autre la connexion/loadbalancing vers ces backend.
- service `ExternalName`: utilise CoreDNS pour mapper le nom de domaine du service à un enregistrement `CNAME` renvoyant vers un autre service par exemple à l'extérieur du Cluster. Aucun proxy d'aucune sorte n'est utilisé.

Fournir des services LoadBalancer on premise avec [MetalLB](#)

Dans un cluster managé provenant d'un fournisseur de cloud, la création d'un objet Service Loadbalancer entraîne le provisionnement d'une nouvelle machine de loadbalancing à l'extérieur du cluster avec une IPv4 publique grâce à l'offre d'IaaS du fournisseur (impliquant des frais supplémentaires).

Cette intégration n'existe pas par défaut dans les clusters de dev comme minikube ou les cluster on premise (le service restera pending et fonctionnera comme un NodePort). Le projet [MetalLB](#) cherche à y remédier en vous permettant d'installer un loadbalancer directement dans votre cluster en utilisant une connexion IP classique ou BGP pour la haute disponibilité.

Le mesh networking et les *service meshes*

Un **service mesh** est un type d'outil réseau pour connecter un ensemble de pods, généralement les parties d'une application microservices de façon encore plus intégrée que ne le permet Kubernetes.

En effet opérer une application composée de nombreux services fortement couplés discutant sur le réseau implique des besoins particuliers en terme de routage des requêtes, sécurité et monitoring qui nécessite l'installation d'outils fortement dynamique autour des nos conteneurs.

Un exemple de service mesh est <https://istio.io> qui, en ajoutant dynamiquement un conteneur "sidecar" à chacun des pods à supervisés, ajoute à notre application microservice un ensemble de fonctionnalités d'intégration très puissant.

Un autre service mesh populaire et plus simple/léger qu'Istio, Linkerd :
<https://linkerd.io/>

Les network policies : des firewalls dans le cluster

Voir cours sur la sécurité

Comparaison des plugins CNI

<https://platform9.com/blog/the-ultimate-guide-to-using-calico-flannel-weave-and-cilium>

Flannel

Flannel est le plugin le plus ancien et simple pour kubernetes. Il implémente la communication Pod-to-Pod créant un réseau overlay par

dessus un backend linux standard comme VXLAN. Il ne s'occupe pas de la communication inter-service.

Son principal avantage est d'être plus simple à implémenter et maintenir que Calico ou Cilium et plus compatible (pas besoin d'un noyau eBPF par exemple). Idéal pour les petits clusters peu critiques.

Il ne dispose pas d'implémentation des Network Policies à moins d'être complémenté par Calico (Cannal)

Calico

Calico développé par l'entreprise Tigera est le plugin CNI le plus populaire pour les clusters à grande échelle avec des configurations avancées et des performances exemplaires.

Il peut implémenter la communication inter-Pod avec un réseau overlay ou en routant le traffic entre les noeuds à l'aide d'un routeur virtuel utilisant BGP (protocole inter-routeur structurant internet) :

<https://projectcalico.docs.tigera.io/networking/determine-best-networking>

Calico est assez modulaire et très configurable. Il peut:

- venir compléter flannel avec des network policies (Canal) :
<https://ubuntu.com/kubernetes/docs/cni-canal>
- s'intégrer avec Istio pour proposer de network policies au niveau application (HTTP ou gRPC etc)
<https://projectcalico.docs.tigera.io/security/tutorials/app-layer-policy/enforce-policy-istio>
- S'intégrer avec metallb pour fournir des LoadBalancer in cluster résilient avec BGP
- Ou encore depuis peu fournir un dataplane complet basé sur eBPF (des hooks sur programmables pour étendre le noyau le noyau linux et en particulier sa gestion des paquets réseau):
<https://www.tigera.io/blog/introducing-the-calico-ebpf-dataplane/>.

Dans cette configuration il prend complètement en charge la communication inter-Services.

Calico est extrêmement fin sur les network policies mais peut aussi gérer le traffic en dehors de kubernetes et notamment s'occuper de règles de firewall pour des VM. Toutes ces règles sont alors configurées par exemple avec des CRD kubernetes ou via la dashboard Tigera (offre SaaS) :

<https://www.tigera.io/features/microsegmentation/>

Calico dispose de plus de nombreuses offres managées de différents vendeurs. Il s'intègre en particulier avec l'offre de Tigera et stack complète d'observabilité et sécurité réseau. Cf cours sur la sécurité.

Cilium

Solution plus récente mais très en vogue, Cilium implémente depuis le départ un dataplane eBPF qui lui permet d'associer une configuration simple avec des fonctionnalités puissantes qui concurrencent Calico:

- remplace kube-proxy avec sa communication inter-services eBPF ()
- fournit une observabilité réseau puissante via une récolte de métrique et son dashboard Hubble (Open source ce qui est un avantage par rapport à Tigera)
- fournit des Network Policies standard et avancées au niveau transport (OSI 3-4) et application (OSI 7) qui sont DNS-aware
- fournit en option un chiffrement du traffic
- Permet de créer facilement un multi-cluster

Comparaison Cilium et Calico

Cilium et Calico ont à peu près le même périmètre de fonctionnalités puissantes et performances exemplaires mais :

- Calico est un peu plus modulaire et peut s'installer de plusieurs façons... ce qui peut être un avantage mais peut aussi amener de la

complexité.

- Cilium est plus open source côté dashboard et intégration SIEM car non adossé à une offre SaaS
- Calico est capable d'étendre sa gestion de sécurité réseau au delà du cluster à d'autres machines avec l'agent calico installé ce qui peut être très puissant.

Ressources sur le réseau

- Documentation officielle :
<https://kubernetes.io/fr/docs/concepts/services-networking/service/>
- [An introduction to service meshes - DigitalOcean](#)
- [Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?](#)
- [Determine best networking option - Project Calico](#)
- [Doc officielle sur les solutions de networking](#)

Formation Kubernetes

404

Page not found. Check the URL or try using the search bar.

Formation Kubernetes

Cours - L'opérateur CertManager

Introduction

cert-manager est un contrôleur Kubernetes qui automatise la gestion et le renouvellement des certificats TLS. Il transforme la gestion des certificats en ressources Kubernetes déclaratives, éliminant ainsi les processus manuels fastidieux.

cert-manager fonctionne selon un modèle de réconciliation continue typique de Kubernetes. Il surveille les ressources qu'il gère et s'assure que l'état réel correspond à l'état désiré.

Les Custom Resource Definitions (CRDs)

cert-manager introduit plusieurs CRDs qui constituent son API. Voici les principales ressources :

1. Issuer et ClusterIssuer

Les Issuers représentent les autorités de certification qui peuvent émettre des certificats.

Issuer : Limité à un namespace spécifique

```

apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: letsencrypt-staging
  namespace: default
spec:
  acme:
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    email: admin@example.com
    privateKeySecretRef:
      name: letsencrypt-staging-key
    solvers:
      - http01:
          ingress:
            class: nginx

```

ClusterIssuer : Disponible dans tous les namespaces

```

apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: admin@example.com
    privateKeySecretRef:
      name: letsencrypt-prod-key
    solvers:
      - http01:
          ingress:
            class: nginx

```

Types d'Issuers disponibles (voir section plus loin):

- **ACME** (Let's Encrypt) : Certificats gratuits avec validation automatique
- **CA** : Utilise une paire de clés CA existante
- **Vault** : HashiCorp Vault PKI
- **Venafi** : Solutions enterprise
- **SelfSigned** : Auto-signé (pour développement/test)
- **External** : Issuers tiers (AWS PCA, Google CA, etc.)

2. Certificate

La ressource Certificate décrit un certificat désiré. C'est la ressource principale avec laquelle les utilisateurs interagissent.

```

apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: example-com-cert
  namespace: default
spec:
  # Nom du Secret où stocker le certificat
  secretName: example-com-tls

  # Durée de validité du certificat
  duration: 2160h  # 90 jours

  # Renouveler 15 jours avant expiration
  renewBefore: 360h

  # Référence vers l'Issuer à utiliser
  issuerRef:
    name: letsencrypt-prod
    kind: ClusterIssuer

  # Informations du certificat
  commonName: example.com
  dnsNames:
    - example.com
    - www.example.com

  # Subject Alternative Names (optionnel)
  ipAddresses:
    - 192.168.1.1
  uris:
    - spiffe://cluster.local/ns/default/sa/myapp

  # Usages du certificat
  usages:
    - digital signature
    - key encipherment
    - server auth
    - client auth

  # Clé privée
  privateKey:
    algorithm: RSA
    encoding: PKCS1
    size: 2048
    rotationPolicy: Always

```

3. CertificateRequest [🔗](#)

Ressource de bas niveau créée automatiquement par cert-manager lors d'une demande de Certificate. Elle contient le CSR brut.

```
apiVersion: cert-manager.io/v1
kind: CertificateRequest
metadata:
  name: example-com-cert-abcd123
  namespace: default
spec:
  request: LS0tLS1CRUDJTi... # CSR encodé en base64
  duration: 2160h
  issuerRef:
    name: letsencrypt-prod
    kind: ClusterIssuer
  usages:
    - digital signature
    - key encipherment
```

Note : Les utilisateurs n'ont généralement pas besoin de créer cette ressource manuellement.

4. Order et Challenge (pour ACME) [🔗](#)

Ces ressources sont créées automatiquement lors de l'utilisation d'un Issuer ACME (Let's Encrypt).

Order : Représente une commande de certificat ACME

```
apiVersion: acme.cert-manager.io/v1
kind: Order
metadata:
  name: example-com-cert-order
  namespace: default
spec:
  request: LS0tLS1CRUDJTi...
  issuerRef:
    name: letsencrypt-prod
    kind: ClusterIssuer
  dnsNames:
    - example.com
```

Challenge : Représente un challenge de validation ACME (HTTP-01 ou DNS-01)

```
apiVersion: acme.cert-manager.io/v1
kind: Challenge
metadata:
  name: example-com-cert-challenge
  namespace: default
spec:
  type: HTTP-01
  dnsName: example.com
  token: abcd1234
  key: xyz789
  solver:
    http01:
      ingress:
        class: nginx
```

Voici le processus complet de création d'un certificat :

1. **Création de Certificate** : L'utilisateur crée une ressource Certificate
2. **Génération de clé privée** : cert-manager génère une clé privée
3. **Création de CertificateRequest** : Un CSR est généré et encapsulé
4. **Soumission à l'Issuer** : Le CSR est envoyé à l'autorité de certification
5. **Validation** (si ACME) : Création d'Order et Challenges pour prouver la propriété du domaine
6. **Signature** : L'autorité signe le certificat
7. **Stockage** : Le certificat et la clé sont stockés dans un Secret Kubernetes
8. **Surveillance** : cert-manager surveille l'expiration et renouvelle automatiquement

Types de validation ACME [🔗](#)

Pour Let's Encrypt et autres ACME, il existe deux méthodes de validation :

HTTP-01 Challenge [🔗](#)

Prouve que vous contrôlez le domaine en servant un fichier spécifique via HTTP.

```
solvers:
- http01:
  ingress:
    class: nginx
```

Avantages :

- Simple à configurer
- Fonctionne avec n'importe quel Ingress Controller

Limitations :

- Ne supporte pas les wildcards (*.example.com)
- Nécessite que le port 80 soit accessible publiquement

DNS-01 Challenge [🔗](#)

Prouve la propriété en créant un enregistrement DNS TXT.

```
solvers:
- dns01:
  cloudflare:
    email: admin@example.com
    apiTokenSecretRef:
      name: cloudflare-api-token
      key: api-token
```

Avantages :

- Support des certificats wildcard
- Pas besoin d'exposition HTTP publique

Limitations :

- Configuration plus complexe

- Nécessite l'accès à l'API DNS

Intégration avec les ressources Kubernetes [🔗](#)

Avec Ingress [🔗](#)

cert-manager s'intègre nativement avec les ressources Ingress via des annotations :

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  tls:
  - hosts:
    - example.com
    secretName: example-com-tls # cert-manager créera ce Secret
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: web
            port:
              number: 80
```

Avec Gateway API [🔗](#)

Support natif pour la nouvelle Gateway API de Kubernetes :

```

apiVersion: gateway.networking.k8s.io/v1beta1
kind: Gateway
metadata:
  name: example-gateway
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
spec:
  listeners:
  - name: https
    protocol: HTTPS
    port: 443
    tls:
      certificateRefs:
      - name: example-gateway-tls

```

Certificats mTLS (Mutual TLS) [🔗](#)

Le mTLS (Mutual TLS) est une méthode d'authentification bidirectionnelle où non seulement le client vérifie l'identité du serveur (comme dans TLS standard), mais le serveur vérifie également l'identité du client. C'est essentiel pour sécuriser les communications entre microservices dans une architecture moderne.

On peut configurer les objets certificats pour les lier à certains objets services Kubernetes

Emetteurs de certificats pour CertManager [🔗](#)

Issuers Intégrés (In-tree) [🔗](#)

cert-manager peut obtenir des certificats de diverses autorités de certification, notamment Let's Encrypt, HashiCorp Vault, et des PKI privées.

1. ACME / Let's Encrypt [🔗](#)

- Certificats gratuits et automatisés

- Idéal pour les workloads publics
- Renouvellement automatique tous les 90 jours

2. HashiCorp Vault

Vault est un magasin de secrets multi-usage qui peut signer des certificats pour votre infrastructure PKI. Configuration via plusieurs méthodes d'authentification :

- **AppRole** : authentification basée sur des rôles
- **Kubernetes Auth** : utilisation des Service Accounts Kubernetes
- **JWT/OIDC** : pour les clusters avec OIDC activé
- **Token** : authentification par token direct

3. Venafi

- Solutions enterprise (TPP / TLS Protect Cloud)
- Gestion centralisée des certificats
- Conformité et audit avancés

Issuers Externes (External/Out-of-tree)

Les issuers externes sont installés en déployant un pod supplémentaire dans votre cluster qui surveille les ressources CertificateRequest.

4. AWS Private CA (AWS PCA)

AWS Private CA Issuer est un projet open source qui fait le pont entre AWS Private CA et cert-manager, permettant à cert-manager de signer des demandes de certificats via AWS PCA.

5. Google Cloud CA Service

- Intégration native avec Google Cloud
- Gestion centralisée pour GKE

7. Cloudflare Origin CA

origin-ca-issuer est utilisé pour demander des certificats signés par Cloudflare Origin CA afin d'activer TLS entre le edge Cloudflare et vos workloads Kubernetes

- Idéal si vous utilisez Cloudflare comme CDN
- Certificats gratuits valides 15 ans

et d'autres...

Formation Kubernetes

TP - Installer l'opérateur Certmanager pour les certificats HTTPS

Nous aimerais héberger des sites proprement en HTTPS, mais pour cela nous avons besoin de pouvoir:

- Exposer nos applications en HTTP sur des noms de domaines adéquats comme dans le TP monsterstack.
- Configurer HTTPS pour authentifier et chiffrer la connection à nos applications. Pour cette partie nous avons besoin de générer des certificats d'authentification délivrés par une autorité dont la plus pratique pour nous est **letsencrypt** qui permet de réclamer un certificat via un challenge automatique. Cette configuration est souvent réalisé dans kubernetes via l'opérateur Certmanager

A noter que vos serveurs VNC qui sont aussi désormais des clusters k8s ont déjà plusieurs sous-domaines configurés: <votrelogin>.<sousdomaine>.dopl.uk et *.<votrelogin>.<sousdomaine>.dopl.uk. Le sous domaine argocd.<login>.<sousdomaine>.dopl.uk pointe donc déjà sur le serveur (Wildcard DNS).

Ce nom de domaine va nous permettre de générer un certificat HTTPS pour notre application web argoCD grâce à un ingress nginx, le cert-manager de k8s et letsencrypt (challenge HTTP101).

Si nécessaire : installer le ingress NGINX 

Les TPs de ce supports utilisant cert-manager supposent que vous avez l'ingress nginx installé. Ils peuvent cependant fonctionner avec d'autres ingress directement ou avec de petites modifications.

Si ce n'est pas encore fait vous pouvez installer le ingress nginx selon votre plateforme

- |> **Dans minikube**
- |> **Dans kind**
- |> **Dans le lab kube_tofu / hobby-kube**
- |> **Dans k3s**

Installer Cert-manager [🔗](#)

- Pour installer cert-manager lancez : `kubectl apply -f https://github.com/jetstack/cert-manager/releases/download/v1.16.1/cert-manager.yaml`
- Il faut maintenant créer une ressource de type `clusterIssuer` pour pourvoir émettre (to issue) des certificats.
- Créez une ressource comme suit (soit dans Lens avec + soit dans un fichier à appliquer ensuite avec `kubectl apply -f`):

```

apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    # You must replace this email address with your own.
    # Let's Encrypt will use this to contact you about expiring
    # certificates, and issues related to your account.
    email: cto@nomail.fr
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      # Secret resource that will be used to store the account's private key.
      name: letsencrypt-prod-account-key
    # Add a single challenge solver, HTTP01 using nginx
    solvers:
    - http01:
        ingress:
          class: nginx

```

D'autres Issuers [🔗](#)

Pour utiliser l'issuer précédent (challenge HTTP01 de letsencrypt) il faut que le cluster soit joignable sur une IP publique et que l'ingress nginx soit bien configuré.

Pour générer des certificats si ces requirements ne sont pas vérifiés on peut utiliser des **certificats autosignés** ou un **challenge DNS** letsencrypt.

Self signed [🔗](#)

Certificats autosignés créez la resource suivante:

```

apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: selfsigned
spec:
  selfSigned: {}

```

Challenge DNS avec l'api DigitalOcean [🔗](#)

► Détails

Formation Kubernetes

Cours optionnel - Problématiques de sécurité Kubernetes

Modèles de menace et l'environnement dynamique Kubernetes

La première étape pour pouvoir choisir et configurer la sécurité d'un cluster est d'élaborer un modèle de menace en répondant à généralement à la question : quelles sont les menaces qui me concernent et les vecteurs d'attaque probables pour ces menaces ?

Grossièrement les risques : "un attaquant compromet la sécurité d'une application exposée publiquement" et "une partie utilisatrice d'un cluster multitenant occupe toutes les ressources du cluster suite à une fuite mémoire / a accès à des information confidentielles ne la concernant pas" n'appellent pas les mêmes réponses en sécurité.

Notamment, la question de la taille du cluster et de ses usagers compte beaucoup. Il est parfois plus simple d'utiliser de multiples clusters pour mitiger un risque que de vouloir mutualiser dans un seul cluster, ce qui amène souvent le besoin d'une politique plus ou moins proche du "zero trust".

Mais généralement et sans devenir paranoïaque il est important de réaliser à quel point un cluster Kubernetes est un environnement dynamique qui implique de nombreux vecteurs d'attaque potentiels : si un attaquant peut passer d'un conteneur à l'autre par des mouvement latéraux sur le réseau (ce qui est possible par défaut) du cluster il pourra plus facilement trouver un conteneur suffisamment unsecure pour

éventuellement prendre le contrôle du noeud ou du cluster, ce qui est dramatique à grande échelle.

Face à ce type de menace l'idée de sécuriser simplement l'accès au cluster de l'extérieur n'est pas suffisant. Il faut considérer également la sécurité dynamique interne au cluster.

Quelques vecteurs d'attaque classiques [🔗](#)

Il existe plusieurs types classiques de vecteurs d'attaque qui peuvent être utilisés pour compromettre un cluster Kubernetes :

- Misconfigured authentication and authorization - Si les mécanismes d'authentification et d'autorisation dans Kubernetes ne sont pas correctement configurés, un attaquant pourrait potentiellement accéder au cluster en exploitant des identifiants faibles ou compromis.
- Vulnerable or malicious container images - Si une image de conteneur en cours d'exécution dans un cluster Kubernetes contient une vulnérabilité ou un code malveillant, un attaquant pourrait potentiellement l'utiliser pour prendre le contrôle de l'hôte ou du réseau sous-jacent.
- Unsecured API server - Le serveur API Kubernetes est un composant critique du cluster, et s'il n'est pas correctement sécurisé, un attaquant pourrait potentiellement accéder à des informations sensibles ou même prendre le contrôle de l'ensemble du cluster.
- Insecure network traffic - Si le trafic réseau entre les composants ou les nœuds Kubernetes n'est pas correctement chiffré ou authentifié, un attaquant pourrait potentiellement intercepter ou modifier le trafic pour accéder à des informations sensibles ou exécuter une attaque Man-in-the-Middle (MitM).

- Excessive privileges - Si un utilisateur ou un compte de service dans Kubernetes dispose de privilèges excessifs ou est en mesure d'escalader ses privilèges, un attaquant pourrait potentiellement utiliser cet accès pour prendre le contrôle du cluster.
- Malicious or misconfigured plugins - Si un plugin utilisé par Kubernetes pour le stockage, le réseau ou d'autres services est malveillant ou mal configuré, un attaquant pourrait potentiellement l'utiliser pour accéder au cluster ou exécuter du code malveillant.
- Social engineering - Les attaquants peuvent tenter d'utiliser des tactiques d'ingénierie sociale, telles que le phishing ou le spear-phishing, pour accéder à des identifiants ou des informations sensibles qui peuvent être utilisés pour compromettre le cluster.

Il est important d'être conscient de ces vecteurs d'attaque et de prendre des mesures pour les atténuer :

- mise en place d'authentification et d'autorisation solides
- utilisation d'images de conteneurs sécurisées
- chiffrement du trafic réseau
- configuration minutieuse des plugins et des privilèges. Des audits de sécurité réguliers et une surveillance peuvent également aider à identifier les vulnérabilités potentielles et à atténuer les risques avant qu'ils ne puissent être exploités par des attaquants.

Sécuriser l'API avec le Role-Based Access Control (RBAC)

L'API est le point d'accès universel au Cluster. Les composants de base du cluster, comme les utilisateurs et même tous les pods du Cluster y ont accès par défaut et peuvent donc contrôler potentiellement n'importe quoi dans le cluster si on en donne le droit. Il est donc impératif de bien limiter l'accès à l'API au cas par cas, pour les utilisateurs humains du cluster mais aussi les pods/composants.

Kubernetes intègre un système de permissions fines pour chaque action sur les ressources et les namespaces. Il fonctionne en liant des ensembles de permissions appelées `Role`s à des identités. Ces identités peuvent être celles d'humains appelés `User/Group` ou des comptes de service/automatisation pour vos programmes appelés `ServiceAccount`.

L'authentification des `User`

On peut authentifier un utilisateur avec notamment:

- de façon statique à l'aide d'un `fichier token` ou d'un `certificat x509` a créer par l'administrateur.
- à l'aide d'une intégration avec l'une ou l'autre solution de gestion d'identité (compatible OpenID, fournie par un cloudprovider comme IAM d'AWS, Active Directory, Keycloak etc). Pour une solution keycloak voir fin du TP : Bootstrapper un cluster multi-noeud avec Ansible.

Exemple de comment générer un certificat à créer un nouvel utilisateur dans minikube: <https://docs.bitnami.com/tutorials/configure-rbac-in-your-kubernetes-cluster/>

Doc officielle: <https://kubernetes.io/docs/reference/access-authn-authz/authentication/>

Le `ServiceAccount` d'un pod

Chaque pod dans le Cluster est lié à sa création avec un `ServiceAccount` soit implicitement au service account par défaut (`default`) du namespace soit explicitement à un autre `ServiceAccount` adapté. Il peut ensuite utiliser une authentification à l'API grâce au token associé.

Tutoriel pour jouer avec le RBAC et les `ServiceAccounts` :

<https://dzone.com/articles/using-rbac-with-service-accounts-in-kubernetes>

Roles et ClusterRoles

Une `role` est un objet qui décrit un ensemble d'actions permises sur certaines ressources et s'applique sur **un seul namespace**. Pour prendre un exemple concret, voici la description d'un roles qui autorise la lecture, création et modification de `pods` et de `services` dans le namespace par défaut:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-and-services
rules:
- apiGroups: []
  resources: ["pods", "services"]
  verbs: ["create", "delete", "get", "list", "patch", "update", "watch", "proxy"]
```

- Un role est une liste de règles `rules`

Les rules sont décrites à l'aide de verbes qui décrivent des type d'action sur les objects. Les verbes de base :

HTTP verb	request verb
POST	create
GET, HEAD	get (for individual resources), list (for collections, including full object content), watch (for watching an individual resource or collection of resources)
PUT	update
PATCH	patch
DELETE	delete (for individual resources), deletecollection (for collections)

Il y a aussi le verbe `impersonate` qui permet d'agir en tant qu'autre utilisateur/identité avec la syntaxe `--as=myuser`. Exemple:

<https://johnharris.io/2019/08/least-privilege-in-kubernetes-using-impersonation/>

- Classiquement on crée des `Roles` comme `admin` OU `monitoring` qui désignent un ensemble de permission consistante pour une tâche donnée.
- Notre role exemple est limité au namespace `default`. Pour créer des permissions valable pour tout le cluster on utilise à la place un objet

appelé un `clusterRole` qui fonctionne de la même façon mais indépendamment des namespace.

Bindings

- Le rôle **ne fait rien par lui-même** : il doit être appliqué à une identité ie un `User`, `Group` ou `ServiceAccount` à l'aide respectivement de `RoleBinding` et `clusterRoleBinding` comme l'exemple suivant:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: default
  name: pods-and-services
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: alice
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: mydevs
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: pod-and-services
```

En plus des rôles que vous pouvez créer pour les utilisateur·ices et processus de votre cluster, il existe déjà dans kubernetes un ensemble de `Roles` et `ClusterRoles` prédéfinis qui sont affichables avec :

```
kubectl get clusterroles
```

- Le rôle `cluster-admin` fournit un accès complet à l'ensemble du cluster.
- Le rôle `admin` fournit un accès complet à un espace de noms précis.
- Le rôle `edit` permet à un·e utilisateur·ice de modifier des choses dans un espace de noms.
- Le rôle `view` permet l'accès en lecture seule à un espace de noms.

Il y a plein d'autre `roles` et `clusterroles` destinés à différents composants du cluster qu'on peut étudier.

La commande `kubectl auth can-i <verb> <type_de_resource>` permet de déterminer selon le profil utilisé (défini dans votre `kubeconfig`) les permissions actuelles de l'user sur les objets Kubernetes. On peut en plus utiliser l'impersonnaison avec `can-i` (et toutes les commandes `kubectl`) pour tester les limites d'une d'action avec identité. Exemple:

```
...  
kubectl auth can-i delete pod --as=my-limited-serviceaccount  
no
```

Auditer le RBAC... [🔗](#)

... Pour trouver les pods avec trop de droits ou qui ont réussi à en avoir plus. Il existe plusieurs solutions la plus populaire est Kubiscan:

<https://github.com/cyberark/KubiScan>

Sécurité des images et de la runtime de conteneur. [🔗](#)

Un vaste sujet : Cf livre O'Reilly Container Security.

- Utiliser une runtime de conteneur à jour et qui exécute les conteneurs en userspace idéalement.
- Containerd et CRI-O n'ont pas besoin de démon privilégié comme Docker pour fonctionner et sont plus simples. Elles sont donc à conseiller d'un point de vue sécurité.
- Éviter les conteneurs privilégiés sauf cas exceptionnel (exemple `kube-proxy` est privilégié pour éditer les règles iptables) et les isoler le cas échéant.

Auditer les images [🔗](#)

Avoir une analyse statique des images incluses dans son registry (harbor ou quay.io, ou le registry d'un provider par exemple) et qui bloque les déploiements au niveau de la CI/CD si trop de failles critiques.

Pod Security Admission [🔗](#)

Cependant il n'est pas toujours suffisant ou possible d'auditer systématiquement les images en amonts

- a la phase admission de la création d'un pod on peut vérifier que le pod respecte une classe de `PodSecurityStandard` ou autre spécification grâce à un plugin.

=> les `PodSecurityPolicies` sont dépréciées et vont bientôt sortir du core de Kubernetes. Elles sont remplacées par <https://kubernetes.io/docs/concepts/security/pod-security-admission/> qui fournit les mêmes fonctionnalités en tant qu'extension de Kubernetes.

<https://kubernetes.io/docs/concepts/security/pod-security-standards/>

Privileged : Unrestricted policy, providing the widest possible level of permissions. This policy allows for known privilege escalations. Baseline : Minimally restrictive policy which prevents known privilege escalations. Allows the default (minimally specified) Pod configuration. Restricted : Heavily restricted policy, following current Pod hardening best practices.

Zero Trust network pour le cluster [🔗](#)

Comme évoqué précédemment les mouvements latéraux sur le réseau d'un cloud sont un vecteur privilégié d'attaque ou d'amplification d'une attaque. Pour réduire la surface offerte il est possible notamment de chiffrer les connexions entre les pods.

<https://projectcalico.docs.tigera.io/security/adopt-zero-trust>

NetworkPolicies

Crédits [Ahmet Alp Balkan](#)

<https://ahmet.im/blog/kubernetes-network-policy/>

Par défaut, les pods ne sont pas isolés au niveau réseau : ils acceptent le trafic de n'importe quelle source.

Les pods sont isolés dès qu'une NetworkPolicy les sélectionne. Une fois qu'une NetworkPolicy (dans un certain namespace) inclut un pod particulier, ce pod rejette toutes les connexions qui ne sont pas autorisées par cette NetworkPolicy. Il faut ensuite whitelister les connexions pertinentes.

- Des exemples de Network Policies : [Kubernetes Network Policy Recipes](#)

Chiffrement TLS des communications inter-pod

En complément des NetworkPolicies, il est possible de systématiser le chiffrement des communications intra cluster:

- Avec un Service Mesh comme `Istio` OU `Linkerd...`
- ... Combiné ou non avec un plugin de CNI comme `Calico` OU `Cilium`

Sécurité et Observabilité

Il est important d'avoir une visualisation dynamique adéquate du réseau et des pods/objets d'un cluster pour identifier les dépendances et élaborer une politique de sécurité. Il existe pour cela en plus du monitoring (Prometheus/Grafana, ou service type DataDog etc), des dashboards comme celle de tigera (Calico), hubble (Cillium), Istio ou linkerd qui permette une observabilité avancée du réseau.

De plus, un cluster est un espace dynamique où le nombre d'événements est trop important pour être traité facilement par un humain encore moins pour une évaluation continue des menaces. A grande échelle, il faut automatiser la détection d'intrusion et les mitigations associées.

Pour cela il est nécessaire de collecter des informations de façon centralisée et paramétrier des patterns pour les traiter massivement et identifier les comportements suspicieux.

- Les audit logs du cluster pour détecter les appels étranges à l'API
- Les communications interpods
- provenance des requêtes externes (à passer à la moulinette des sources suspectes)
- Voir faire du Deep Packet Inspection sur le traffic
- ...

Un dispositif de SIEM (Security Information and Event Management) permet ce type d'analyse massive et mitigation automatique. Plusieurs solutions on premise ou en SaaS sont possibles : Installation de EFK en mode SIEM, offre du provider comme Azure etc., Offre enterprise de Tigera/Calico etc.

Overview

Bibliographie

- Kubernetes Security and Observability, chez O'Reilly (sponsorisé par Calico donc un peu sur mesure pour leur solution)

Formation Kubernetes

TP optionnel - Kubernetes the (not so) hard way avec Ansible

Kubernetes the hard way avec Ansible

La version la plus manuelle de l'installation de Kubernetes a été documentée à des fins d'apprentissage par Kelsey Hightower qui l'a nommée `Kubernetes the hard way`. On peut la retrouver à l'adresse <https://github.com/kelseyhightower/kubernetes-the-hard-way/tree/master/docs>.

La principale limite de cette méthode d'installation est le nombre très important de manipulations sur plusieurs serveurs et donc le temps d'installation conséquent, peu reproductible et qui favorise les erreurs. Pour remédier à cela, l'installation manuelle a été notamment reprise par `githubixx/RW` de [tauceti.blog](#) et intégré dans une série de tutoriels adossés à des roles Ansible qui documentent cette installation manuelle et sont d'après l'auteur utilisable en production à une échelle moyenne : <https://www.tauceti.blog/posts/kubernetes-the-not-so-hard-way-with-ansible-the-basics/>

Nous allons suivre étape par étapes cette installation manuelle Ansible pour observer et commenter concrètement les différents composants et étapes d'installation de Kubernetes.

- Commencez par cloner le projet de base avec `git clone -b k8s_hard_way_ansible_correction https://github.com/elie/k8s_notsohardway_correction.git`

- Installer Terraform et Ansible avec `bash /opt/terraform.sh` et `sudo apt remove ansible && bash /opt/ansible.sh`

Nous allons maintenant ouvrir le projet et suivre le README pour créer l'infrastructure dans le cloud avec terraform puis exécuter les différents playbooks pour installer étape par étape cluster.

Chaque étape sera l'occasion de commenter le code Ansible et explorer notre cluster au cours de son installation.

Créer les serveurs en IaC avec Terraform [🔗](#)

- compléter le subdomain dans `terraform/variables.tf`
- compléter les tokens infra et DNS en copiant `terraform/secrets.auto.tfvars.dist` sans le `.dist` puis en complétant avec les tokens formateur.

=> pour éviter les conflits, besoin de faire autant de projet hcloud que de stagiaires et envoyer un fichier avec un token cloud par personne par exemple dans le dépôt git

Observons et expliquons ensemble le code.

- `./cloud_init setup_terraform`

Setup Ansible [🔗](#)

L'inventaire Ansible Terraform

- `source env_file`
- `ssh-add ~/.ssh/id_stagiaire`
- `ansible all -K -m ping`
- `ansible-inventory --host controller-0`

Installation d'un réseau privé [wireguard](#) [🔗](#)

Chapitre du tutorial : <https://www.tauceti.blog/posts/kubernetes-the-not-so-hard-way-with-ansible-wireguard/>

- `role: githubixx.ansible_role_wireguard` à appliquer avec `ansible-playbook -K --tags=role-wireguard k8s.yaml`
- variables de configuration dans `terraform/ansible_hosts` et `group_vars/vpn.yml`

Setup PKI infrastructure

Chapitre du tutorial : <https://www.tauceti.blog/posts/kubernetes-the-not-so-hard-way-with-ansible-certificate-authority/>

- `role: githubixx.cfssl`, `ansible-playbook -K --tags=role-cfssl k8s.yaml`
- `role: githubixx.kubernetes-ca`, `ansible-playbook -K --tags=role-kubernetes-ca k8s.yaml`
- variables de configuration dans `group_vars/k8s_ca.yml`
- Génération des kubeconfigs des composants : `ansible-playbook -K playbooks/all_kubeconfs.yml` variables dans `group_vars/all.yml`

Installation de `etcd` sur les controllers

Chapitre : <https://www.tauceti.blog/posts/kubernetes-the-not-so-hard-way-with-ansible-etcd/>

- variables dans `group_var/all.yml`
- `role: githubixx.etcd`, `ansible-playbook --tags=role-etcd k8s.yaml`

test de `etcd` avec

```
● ● ●

ansible -m shell -e "etcd_conf_dir=/etc/etcd" \
-a 'ETCDCTL_API=3 etcdctl endpoint health \
--endpoints=https://{{ ansible_wgk8slaab.ipv4.address }}:2379 \
--cacert={{ etcd_conf_dir }}/ca-etcd.pem \
--cert={{ etcd_conf_dir }}/cert-etcd-server.pem \
--key={{ etcd_conf_dir }}/cert-etcd-server-key.pem' \
k8s_etcd
```

Installation des composants du control plane sur les controllers [🔗](#)

Chapitre: <https://www.tauceti.blog/posts/kubernetes-the-not-so-hard-way-with-ansible-control-plane/>

- role `githubixx.kubernetes-controller` appliquer avec `ansible-playbook --tags=role-kubernetes-controller k8s.yml`
- variables dans `all.yml`

test des composants avec :

```
● ● ●

kubectl cluster-info
echo "test scheduler"
curl -k https://10.8.0.101:10257/healthz
echo "\ntest controller manager"
curl -k https://10.8.0.102:10259/healthz
```

Installation de `containerd`, `kubelet` et `kube-proxy` sur les workers [🔗](#)

Chapitre : <https://www.tauceti.blog/posts/kubernetes-the-not-so-hard-way-with-ansible-worker-2020/> alternative plus ancienne avec Docker et Flannel

CNI : <https://www.tauceti.blog/posts/kubernetes-the-not-so-hard-way-with-ansible-worker/>

variables dans k8s_worker

role : `githubixx.containererd` appliquer avec `ansible-playbook --tags=role-containererd k8s.yml`

Puis role `githubixx.kubernetes-worker` appliquer avec `ansible-playbook --tags=role-kubernetes-worker k8s.yml`

Tester avec `kubectl get nodes` les nodes sont notready car il manque le plugin CNI

Installation du CNI

Même chapitre

role : `githubixx.cilium_kubernetes` appliquer avec `ansible-playbook --tags=role-cilium-kubernetes -K -e cilium_install=true k8s.yml`

Installation de CoreDNS

Même chapitre

playbook : `githubixx_playbooks/coredns.yml` appliquer avec `ansible-playbook -K`

Faire un déploiement de test `kubectl -n default apply -f`

<https://k8s.io/examples/application/deployment.yaml>

- `kubectl -n default get all -o wide`
- `ansible -m get_url -a "url=http://10.200.1.23 dest=/tmp/test.html" k8s_worker`

Mise à jour de l'infra

La mise à jour des différents composants est discutée dans les posts de blogs tauceti mais pour une vue générale on peut se référer à la documentation officielle : <https://kubernetes.io/docs/tasks/administer-cluster/cluster-upgrade/>

Correction par un script [🔗](#)

Pour installer toute l'infrastructure en une seule commande : `bash`

```
deploy_all.sh
```

Détruire l'infra [🔗](#)

- `./cloud_init destroy_infra`

Cluster de 4 noeuds terraform/kubeadm avec metallb, rook, argoCD [🔗](#)

- `git clone -b kubadm_tf_prod_cluster https://github.com/e-lie/provisioning.git`
- compléter le subdomain avec votre prenom ou autre dans `variables.tf`
- compléter les tokens infra et DNS dans en copiant `env_secrets.dist` en `env_secret` et complétant avec les token formateur.
- `./cloud_init setup_terraform`. Si il y a une errur concernant le `remote exec` reexécutez `ssh-add ~/.ssh/id_stagiaire` et relancez l'installation.
- Modifiez la ligne `export KUBE1_DOMAIN=kube1.k8slab.dopl.uk` du fichier `get_k8s_admin_config.sh` en remplaçant `k8slab` par votre sous domaine et exécutez ce script avec `bash`.
- Testez la bonne installation du cluster avec `kubectl cluster-info` et `kubectl get nodes`. Vous pouvez également ajouter la `kubeconfig hobby-kube-connection.yaml` à Lens.

Installer un storage provisionner (CSI plugin) [🔗](#)

Deux options dans ce TP: rook ceph ou le plugin local storage de rancher

Option rook

- Utilisez le quickstart (<https://rook.io/docs/rook/v1.9/quickstart.html>) et les manifestes présents dans le dossier `k8s-bootstrap/rook1_9_2`.

On peut ensuite debugger avec un pod rook toolbox.

Option localstorage

```
kubectl apply -f https://raw.githubusercontent.com/rancher/local-path-provisioner/v0.0.22/deploy/local-path-storage.yaml
```

Installer metallb

Autre élément indispensable d'un cluster on premise, être capable de faire rentrer le traffic depuis l'extérieur. Par défaut les services de type `LoadBalancer` ne fonctionneront pas et resteront des `NodePort`. Il est alors possible de provisionner manuellement des loadbalancer externes vers le bon nodeport. Mais cette méthode est peu efficace et provoque vite des erreurs liées à des conflits de ports et problèmes de mise à jour manuelle.

La solution adaptée est probablement d'installer la solution générique `metallb` qui peut fournir des loadbalancer internes au cluster.

- Compléter `k8s-bootstrap/metallb-values.yaml` avec les listes des ips des noeuds récupérées avec `ping kube1-3.<subdomain>.dopl.uk`
- Installer `metallb` avec le chart helm grâce à la commande :

```
helm upgrade --install metallb metallb \
--repo https://metallb.github.io/metallb \
--namespace metallb-system --create-namespace \
--version 0.12.1 --values=k8s-bootstrap/metallb-values.yaml
```

- Par défaut nous l'avons installé en mode IP : les agents speakers vont répondre aux requêtes ARP pour assigner les IP que nous avons

fournies aux noeuds et rediriger le traffic vers le bon service endpoint.

Installer le Ingress Nginx [🔗](#)

Installons le Ingress Nginx pour exposer des services HTTP et immédiatement vérifier que les services `LoadBalancer` fonctionnent:

```
helm upgrade --install ingress-nginx ingress-nginx \
--repo https://kubernetes.github.io/ingress-nginx \
--namespace ingress-nginx --create-namespace \
--version 4.1.0
```

- Si metallb est bien configuré, le service qui expose le ingress controller devrait se voir attribuer une IP externe. On peut le vérifier avec la commande: `kubectl get svc -n ingress-nginx -o wide`.

Installer l'opérateur CertManager [🔗](#)

voir le début du TP CI/CD avec Gitlab et ArgoCD

Installer ArgoCD pour superviser les applications [🔗](#)

voir le début du TP CI/CD avec Gitlab et ArgoCD

Installer du monitoring [🔗](#)

voir TP monitoring et série de tutorial dans ce TP pour plus avancé

Installer le gestionnaire d'identité keycloak et la connection openID à Kubernetes [🔗](#)

- <https://github.com/int128/kubelogin>

- <https://www.keycloak.org/getting-started/getting-started-kube>
- <https://www.talkingquickly.co.uk/installing-keycloak-kubernetes-helm>
- <https://www.talkingquickly.co.uk/setting-up-oidc-login-kubernetes-kubectl-with-keycloak>

Installer Rancher ? ↗

Formation Kubernetes

404

Page not found. Check the URL or try using the search bar.

Formation Kubernetes

404

Page not found. Check the URL or try using the search bar.

Formation Kubernetes

404

Page not found. Check the URL or try using the search bar.

Formation Kubernetes

Cours - Service mesh et Istio

Le mesh networking et les *service meshes*

Un **service mesh** est un type d'outil réseau pour connecter un ensemble de pods, généralement les parties d'une application microservices de façon encore plus intégrée que ne le permet Kubernetes, mais également plus sécurisé et contrôlable.

En effet opérer une application composée de nombreux services fortement couplés discutant sur le réseau implique des besoins particuliers en terme de routage des requêtes, sécurité et monitoring qui nécessite l'installation d'outils fortement dynamique autour des nos conteneurs.

Un exemple de service mesh est <https://istio.io> qui, en ajoutant dynamiquement un conteneur "sidecar" à chacun des pods à supervisés, ajoute à notre application microservice un ensemble de fonctionnalités d'intégration très puissant.

Un autre service mesh populaire et "plus simple" qu'Istio, Linkerd :
<https://linkerd.io/>

Cilium, le plugin réseau CNI peut aussi opérer comme un service mesh

- <https://www.youtube.com/watch?v=16fgzkIcF7Y>

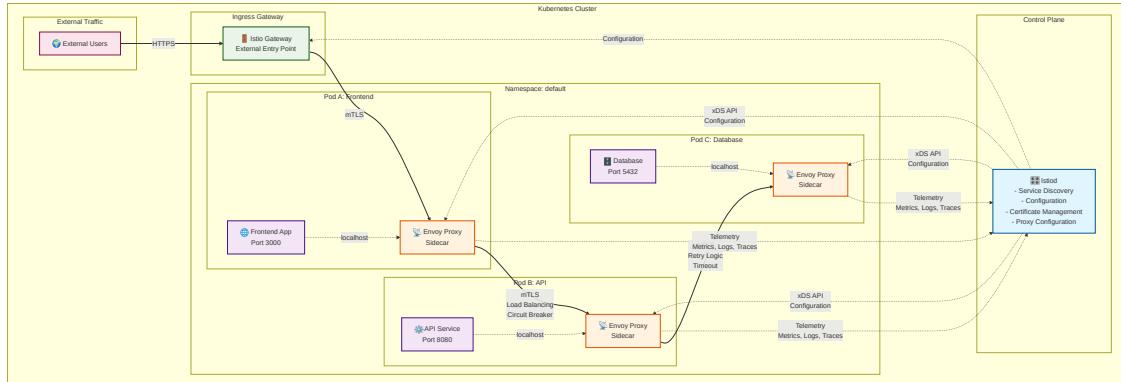
Istio

Istio est un **service mesh** open source qui peut se superposer de manière transparente aux applications distribuées existantes.

Il offrent un moyen uniforme et efficace de sécuriser, connecter et surveiller les services. Il ajoute au cluster, avec peu ou pas de modifications du code des services.

- des fonctionnalités plus flexibles et puissantes que le cœur de Kubernetes pour le LoadBalancing entre les services. Notamment un loadbalancing L7 (la couche réseau application) du trafic HTTP, gRPC, WebSocket et TCP
- une authentification entre les services
- une autorisation fine des communications basée sur cette authentification
- une communication chiffrée avec TLS mutuel
- la surveillance du trafic dans le mesh en temps réel
- Un contrôle granulaire du trafic avec des règles de routage riches, des retry automatiques pour les requêtes, des failovers en cas de panne d'un service et un mécanisme de **fault injection**.
- Une couche de politique de sécurité modulaire. l'API de configuration permet d'exprimer des contrôles d'accès, des ratelimits et des quotas pour le traffic
- Istio expose aussi des métriques, des journaux et des traces (chaîne d'appels réseau) pour tout le trafic au sein d'un cluster, y compris concernant le traffic entrant et sortant (ingress et egress) du cluster.
- Istio est conçu pour être extensible et gère une grande variété de besoins de déploiement.

Architecture de Istio - Mode Sidecar (Classique) [🔗](#)



Composants principaux :

- **Istioid (Control Plane)** : Contrôleur centralisé qui gère la configuration, la découverte de services, et la distribution des certificats
- **Envoy Proxies (Data Plane)** : Proxies sidecar injectés dans chaque pod pour intercepter et gérer le trafic
- **Ingress/Egress Gateways** : Points d'entrée et de sortie du mesh pour le trafic externe

Fonctionnement :

1. **Istioid** configure les proxies via l'API xDS (configuration dynamique)
2. Chaque **sidecar Envoy** intercepte tout le trafic entrant/sortant de son pod
3. Communication sécurisée avec **mTLS automatique** entre les services
4. **Télémétrie** centralisée (métriques, logs, traces) remontée vers Istiod

Istio par l'exemple et autres resources [🔗](#)

- <https://istiobyexample.dev/>
- <https://www.istioworkshop.io/09-traffic-management/06-circuit-breaker/>

CRDs de Istio

Istio utilise plusieurs Custom Resource Definitions (CRDs) pour étendre les fonctionnalités de Kubernetes et permettre la configuration de ses différents composants:

- **VirtualService** : Le CRD `VirtualService` permet de définir les règles de trafic pour contrôler le routage des requêtes HTTP et TCP vers différentes destinations dans le maillage Istio. Il est utilisé pour configurer des fonctionnalités telles que le routage basé sur des en-têtes, des poids du trafic, des redirections et des destinations de service.
- **DestinationRule** : Le CRD `DestinationRule` est utilisé pour définir les règles de trafic spécifiques à une destination, telles que la configuration des politiques de répartition de charge, des répliques et des stratégies de rééquilibrage de charge.
- **Gateway** : Le CRD `Gateway` est utilisé pour configurer les points d'entrée de trafic externes dans le maillage Istio. Il permet de définir des règles pour l'exposition de services Istio à l'extérieur du maillage, comme l'exposition de services HTTP, HTTPS et TCP.
- **ServiceEntry** : Le CRD `ServiceEntry` est utilisé pour déclarer des services externes (non-Istio) au sein du maillage Istio. Il permet à Istio de gérer la communication avec des services situés en dehors du maillage et d'appliquer des politiques de sécurité, de trafic et de résilience à ces services.
- **Sidecar** : Le CRD `Sidecar` est utilisé pour configurer les sidecars Envoy dans les pods de votre application. Il permet de définir des options de configuration spécifiques pour chaque sidecar, telles que les politiques de trafic, les filtres réseau et les règles de sécurité.
- **AuthorizationPolicy** : Le CRD `AuthorizationPolicy` est utilisé pour définir des politiques de contrôle d'accès basées sur les rôles et les permissions pour les services dans le maillage Istio. Il permet de

spécifier des règles d'autorisation pour limiter l'accès aux services en fonction des identités et des attributs de requête.

Istio et API Gateway [🔗](#)

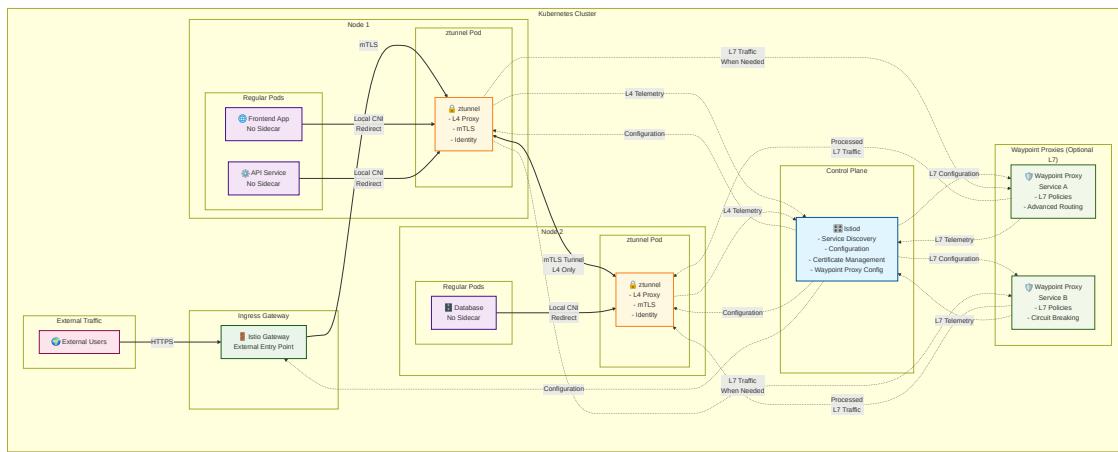
Istio soutient la standardisation via la norme API Gateway de Kubernetes. Il est donc possible d'utiliser les CRDs de l'API Gateway pour configurer le routage du trafic et les points d'entrée du mesh plutôt que les CRDs de Istio.

Correspondance entre les CRDs Istio et Gateway API :

Istio CRD	Gateway API CRD	Utilisation
Gateway	Gateway	Points d'entrée du mesh
VirtualService	HTTPRoute, TLSRoute, TCPRoute	Règles de routage L7/L4
DestinationRule	BackendPolicy (expérimental)	Politiques de trafic (load balancing, circuit breaker)
ServiceEntry	Pas d'équivalent direct	Services externes au cluster
AuthorizationPolicy	SecurityPolicy (en développement)	Politiques d'autorisation

Note : L'API Gateway ne supporte pas encore tous les cas d'usage d'Istio. Pour une standardisation à long terme, elle peut être intéressante, mais la documentation reste moins fournie. Les CRDs de l'API Gateway doivent être installés indépendamment d'Istio.

Architecture Ambient Mesh (Nouvelle approche) [🔗](#)



L'architecture classique de Istio avec des proxies sidecar pour chaque pod peut être lourde à mettre en œuvre et consommatrice en ressources. Istio développe une nouvelle architecture appelée **Ambient Mesh** qui simplifie le déploiement.

Composants de l'Ambient Mesh :

- **ztunnel** : Proxy L4 déployé sur chaque nœud (DaemonSet) qui gère :
 - La sécurité mTLS entre les nœuds
 - L'identité des workloads
 - La redirection du trafic via eBPF/CNI
 - La télémétrie de niveau 4
- **Waypoint Proxies** (optionnels) : Proxies L7 déployés uniquement quand nécessaire pour :
 - Les politiques avancées (autorisation fine, rate limiting)
 - Le routage complexe et circuit breakers
 - Les fonctionnalités L7 (HTTP headers, retries)

Avantages de l'Ambient Mesh :

1. **Simplicité** : Pas de modification des pods applicatifs (pas de sidecar)
2. **Performance** : Moins de ressources consommées, communication directe L4
3. **Adoption progressive** : Activation du mesh par namespace sans redéploiement

4. **Flexibilité** : L7 uniquement quand nécessaire via les **Waypoint Proxies**

Mode de fonctionnement :

1. Le trafic entre pods est automatiquement intercepté par **ztunnel** via CNI
2. **ztunnel** assure la sécurité mTLS et l'observabilité L4
3. Si des politiques L7 sont requises, le trafic passe par un **Waypoint Proxy**
4. La configuration reste centralisée via **Istiod**

Formation Kubernetes

TP optionnel - Stratégies de déploiement

Déployer notre application d'exemple (goprom) et la connecter à prometheus [🔗](#)

Nous allons installer une petite application d'exemple en go.

- Téléchargez le code de l'application et de son déploiement depuis github: `git clone https://github.com/e-lie/k8s-deployment-strategies`

Nous allons d'abord construire l'image docker de l'application à partir des sources et la déployer dans le cluster.

- Allez dans le dossier `goprom_app` et "construisez" l'image docker de l'application avec le tag `<votrelogindockrhub>/goprom`.

› Réponse

- Allez dans le dossier de la première stratégie `recreate` et ouvrez le fichier `app-v1.yml`. Notez que `image:` est à `tecp1/goprom` et qu'un paramètre `imagePullPolicy` est défini à `Never`. Changez ces paramètres si nécessaire (oui en général).

- Appliquez ce déploiement kubernetes:

› Réponse

Observons notre application et son déploiement kubernetes [🔗](#)

- Explorez le fichier de code go de l'application `main.go` ainsi que le fichier de déploiement `app-v1.yaml`. Quelles sont les routes http exposées par l'application ?

> Réponse

- Utilisez le service `NodePort` pour accéder au service `goprom` dans votre navigateur.
- Utilisez le service `NodePort` pour accéder au service `goprom-metrics` dans votre navigateur. Quelles informations récupère-t-on sur cette route ?
- Pour tester le service `prometheus-server` nous avons besoin de le mettre en mode `NodePort` (et non `ClusterIP` par défaut). Modifiez le service dans Lens pour changer son type.
- Exposez le service avec un `NodePort` (n'oubliez pas de préciser le namespace `monitoring`).
- Vérifiez que `prometheus` récupère bien les métriques de l'application avec la requête `PromQL` : `sum(rate(http_requests_total{app="goprom"}[5m])) by (version)`.
- Quelle est la section des fichiers de déploiement qui indique à `prometheus` où récupérer les métriques ?

> Réponse

Créer une dashboard avec un Graphe. Utilisez la requête `prometheus` (champ query suivante):

```
sum(rate(http_requests_total{app="goprom"}[5m])) by (version)
```

Pour avoir un meilleur aperçu de la version de l'application accédée au fur et à mesure du déploiement, ajoutez `{{version}}` dans le champ `legend`.

Observer un basculement de version [🔗](#)

Ce TP est basé sur l'article suivant: <https://blog.container-solutions.com/kubernetes-deployment-strategies>

Maintenant que l'environnement a été configuré :

- Lisez l'article.
- Vous pouvez testez les différentes stratégies de déploiement en lisant leur `README.md`.
- En résumé, pour les plus simple, on peut:
 - appliquer le fichier `app-v1.yml` pour une stratégie.
 - lancer la commande suivante pour effectuer des requêtes régulières sur l'application: `service=$(minikube service goprom --url) ; while sleep 0.1; do curl "$service"; done`
 - Dans un second terminal (pendant que les requêtes tournent) appliquer le fichier `app-v2.yml` correspondant.
 - Observez la réponse aux requêtes dans le terminal ou avec un graphique adapté dans `graphana` (Il faut configurer correctement le graphique pour observer de façon lisible la transition entre v1 et v2). Un aperçu en image des histogrammes du nombre de requêtes en fonction des versions 1 et 2 est disponible dans chaque dossier de stratégie.
 - supprimez le déploiement+service avec `delete -f` ou dans Lens.

Par exemple pour la stratégie `recreate` le graphique donne:

Argo Rollout : un exemple d'opérateur de rollout (controller) [🔗](#)

Stratégies de déploiement avancées [🔗](#)

Canary Deployment [🔗](#)

Avec la stratégie canary, le rollout peut scale un ReplicaSet avec la nouvelle version pour recevoir un pourcentage spécifié de trafic, attendre un temps spécifié, puis revenir au pourcentage 0, et enfin déployer tout le trafic une fois que l'utilisateur est satisfait.

```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: myapp
spec:
  replicas: 5
  strategy:
    canary:
      steps:
        - setWeight: 20      # 20% du trafic vers canary
        - pause: {duration: 1m}
        - setWeight: 40
        - pause: {duration: 1m}
        - setWeight: 60
        - pause: {duration: 1m}
        - setWeight: 80
        - pause: {duration: 1m}
```

Blue-Green Deployment ↗

Avec la stratégie BlueGreen, Argo Rollouts permet aux utilisateurs de spécifier un service preview et un service active. Le Rollout configurera le service preview pour envoyer du trafic vers la nouvelle version tandis que le service active continue à recevoir le trafic de production.

```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: myapp
spec:
  replicas: 3
  strategy:
    blueGreen:
      activeService: myapp-active
      previewService: myapp-preview
      autoPromotionEnabled: false
      scaleDownDelaySeconds: 30
```

Analyse automatique (Canary Analysis) [🔗](#)

Intégration avec des fournisseurs de métriques pour automatiser les décisions :

```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: myapp
spec:
  strategy:
    canary:
      analysis:
        templates:
          - templateName: success-rate
        startingStep: 2
        args:
          - name: service-name
            value: myapp
```

```
apiVersion: argoproj.io/v1alpha1
kind: AnalysisTemplate
metadata:
  name: success-rate
spec:
  args:
    - name: service-name
  metrics:
    - name: success-rate
      interval: 5m
      successCondition: result[0] >= 0.95
      failureLimit: 3
      provider:
        prometheus:
          address: http://prometheus:9090
          query: |
            sum(rate(
              http_requests_total{service="{{args.service-name}}",status=~"2.."})
            [5m])
            /
            sum(rate(
              http_requests_total{service="{{args.service-name}}"}[5m]
            ))
```

Intégrations avec Traffic Management [🔗](#)

Argo Rollouts s'intègre (optionnellement) avec les ingress controllers et les service meshes, exploitant leurs capacités de traffic shaping pour progressivement déplacer le trafic vers la nouvelle version pendant une mise à jour.

Providers supportés :

- **Service Meshes** : Istio, Linkerd, AWS App Mesh, SMI
- **Ingress Controllers** : NGINX, ALB (AWS), Traefik, Ambassador, Apache APISIX
- **Gateway API** : Support natif pour la nouvelle Gateway API Kubernetes

```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: myapp
spec:
  strategy:
    canary:
      trafficRouting:
        istio:
          virtualService:
            name: myapp-vs
            routes:
              - primary
    steps:
      - setWeight: 10
      - pause: {duration: 5m}
```

Rollback automatique [🔗](#)

En cas de problème détecté par l'analyse :

```

spec:
  strategy:
    canary:
      analysis:
        templates:
          - templateName: error-rate
        startingStep: 1
    abortScaleDownDelaySeconds: 30 # Garde les pods pour debug
  
```

Alternatives [🔗](#)

- Flagger
- ?

Facultatif : Installer Istio pour des scénarios plus avancés [🔗](#)

Pour des scénarios plus avancés de déploiement, on a besoin d'utiliser soit un *service mesh* comme Istio (soit un plugin de rollout comme Argo Rollouts mais pas ce que nous proposons ici).

1. Sur k3s, supprimer la release Helm du Ingress Controller Traefik (ou le ingress Nginx) pour le remplacer par l'ingress Istio.
2. Installer Istio, créer du trafic vers l'ingress de l'exemple et afficher le graphe de résultat dans le dashboard Istio :
<https://istio.io/latest/docs/setup/getting-started/>
3. Utiliser ces deux ressources pour appliquer une stratégie de déploiement de type A/B testing poussée :
 - <https://istio.io/latest/docs/tasks/traffic-management/request-routing/>
 - <https://github.com/ContainerSolutions/k8s-deployment-strategies/tree/master/ab-testing>

Formation Kubernetes

404

Page not found. Check the URL or try using the search bar.

Formation Kubernetes

404

Page not found. Check the URL or try using the search bar.