



KUBERNETES

KUBERNETES, DÉPLOYER ET EXPLOITER VOS APPLICATIONS



**GROUP INFRASTRUCTURE
PLATFORM**



SOMMAIRE (1/2)

- Contexte
- Architecture
- Premiers pas
- Pods
- Controllers
- Services



SOMMAIRE (2/2)

- Volumes
- Configuration et secrets
- Stratégies de déploiement
- Opérateurs
- Fonctionnalités d'entreprise
- Développement d'applications compatibles
- Conclusion

LOGISTIQUE



- Horaires
- Déjeuner & pauses
- Autres questions ?





CONTEXTE

AGENDA DE CE CHAPITRE 'CONTEXTE'



- Rappels sur les conteneurs
- Conteneurs sans orchestration
- Fonctionnalités d'orchestration
- Orchestrateurs du marché
- Kubernetes
- Distributions Kubernetes
- Versions

RAPPELS SUR LES CONTENEURS



- Que vous soyez Ops ou Dev...
- Quel que soit le nombre de composants que vous développez / déployez...
- Un des problèmes auxquels vous aurez à faire face...
- C'est la reproductibilité des environnements

POURQUOI EST-CE SI COMPLIQUÉ ?



- Diversité des applications à gérer
- Nombre d'applications à gérer
- Diversité des environnements (prod, pre-prod, QA, intégration...)
- Contraintes de mise à jour des environnements

CONTRAINTES OPS



- Plusieurs applications différentes sur le même host
- Plusieurs versions des serveurs d'application / runtime
- Plusieurs versions des bibliothèques partagées
- Plusieurs versions des systèmes d'exploitation

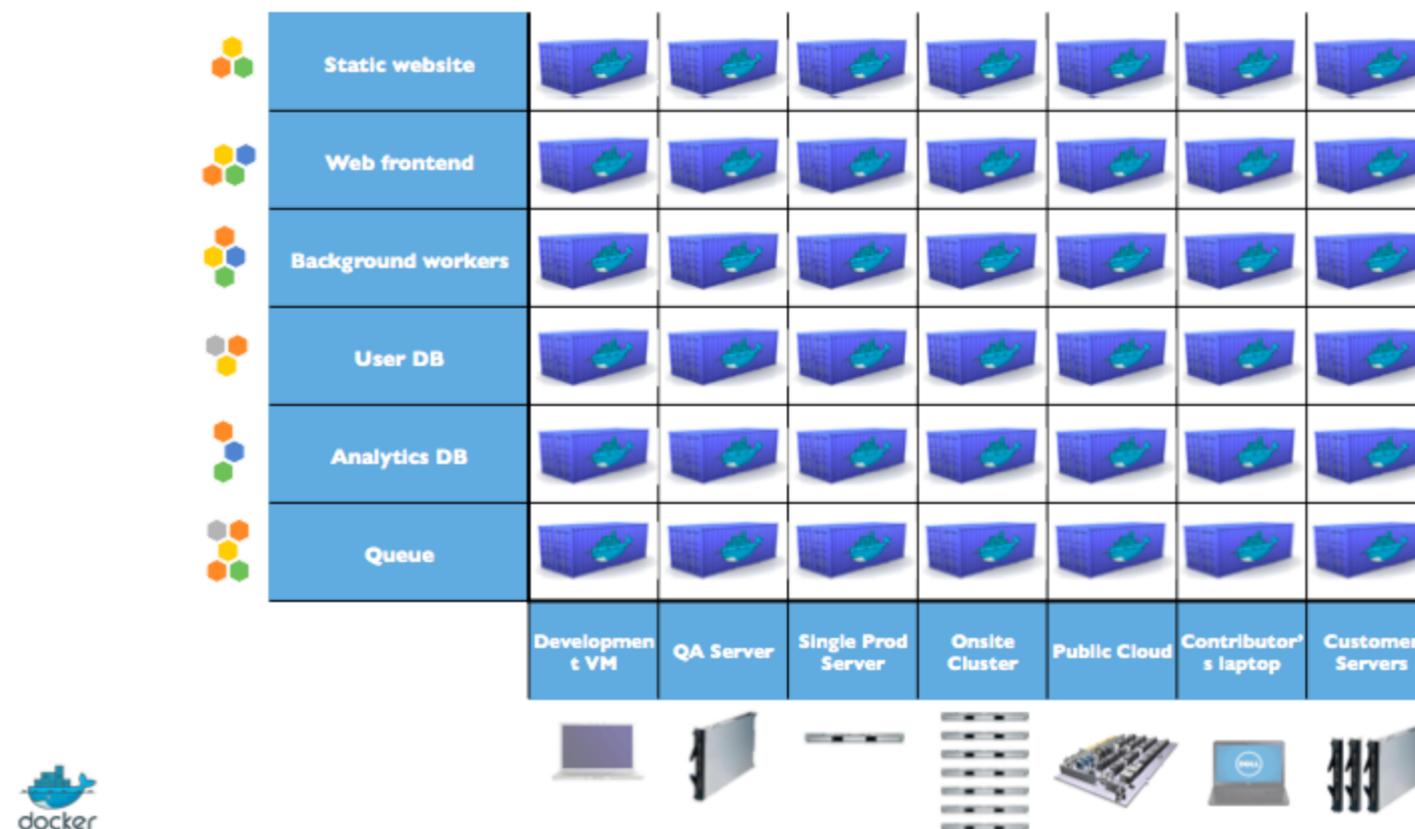


CONTRAINTES DEV

- Pouvoir utiliser/tester rapidement un serveur d'application / runtime
- Pouvoir utiliser les dépendances des composants développés en local (ou au plus près du local)
- Pouvoir reproduire les environnements de production (et tous les autres !)
- Tendance à la multiplication du nombre de composants



L'APPROCHE CONTENEURS / DOCKER



DOCKER, COMMENT ÇA MARCHE



Utilisation du noyau de l'OS 'host' pour avoir plusieurs sous-systèmes isolés :

- Process
- Mémoire
- Réseau
- Stockage
- Périphériques

 Chaque sous-système est appelé un conteneur



RAPPEL : IMAGES (VS CONTENEURS)

Une image :

- Modèle read-only utilisé pour démarrer des conteneurs
- Construite par vous ou par d'autres
- Stockée sur votre host, ou le Docker Hub ou votre registry privée
- Inerte / Immutable

RAPPEL : (IMAGES VS) CONTENEURS



Un conteneur :

- Est construit à partir d'une image
- Contient tout ce dont votre application a besoin pour tourner
- Possède un cycle de vie (create/start/stop/restart)

OÙ UTILISER LES CONTENEURS / DOCKER



- Sur le poste du développeur
- Sur le poste du testeur
- Sur les environnements d'intégration / recette
- En prod

★ Situation idéale : *PARTOUT*(Continuous Delivery FTW !)

CONTENEURS SANS ORCHESTRATION



Comment démarrer les conteneurs ?

- en s'appuyant sur la ligne de commande **docker** ?
- en s'appuyant sur **docker-compose** ?
- en s'appuyant sur des scripts (shell) ?
- en s'appuyant sur *[insérer ici votre outil de gestion de la configuration préféré]* (Ansible, SaltStack, Chef, Puppet, ...)

COMMENT CHOISIR SUR QUELS HOSTS DOIVENT TOURNER VOS CONTENEURS



Le mode *Pet* (animal de compagnie) :

- chaque serveur/host est paramétré pour recevoir un ensemble connu à l'avance d'applications
- chaque serveur possède des caractéristiques correspondant aux applications qu'il héberge
- chaque serveur est taillé pour recevoir un ensemble d'applications et contenir les pics de charge

COMMENT EXPOSER VOS SERVICES CONTENEURISÉS



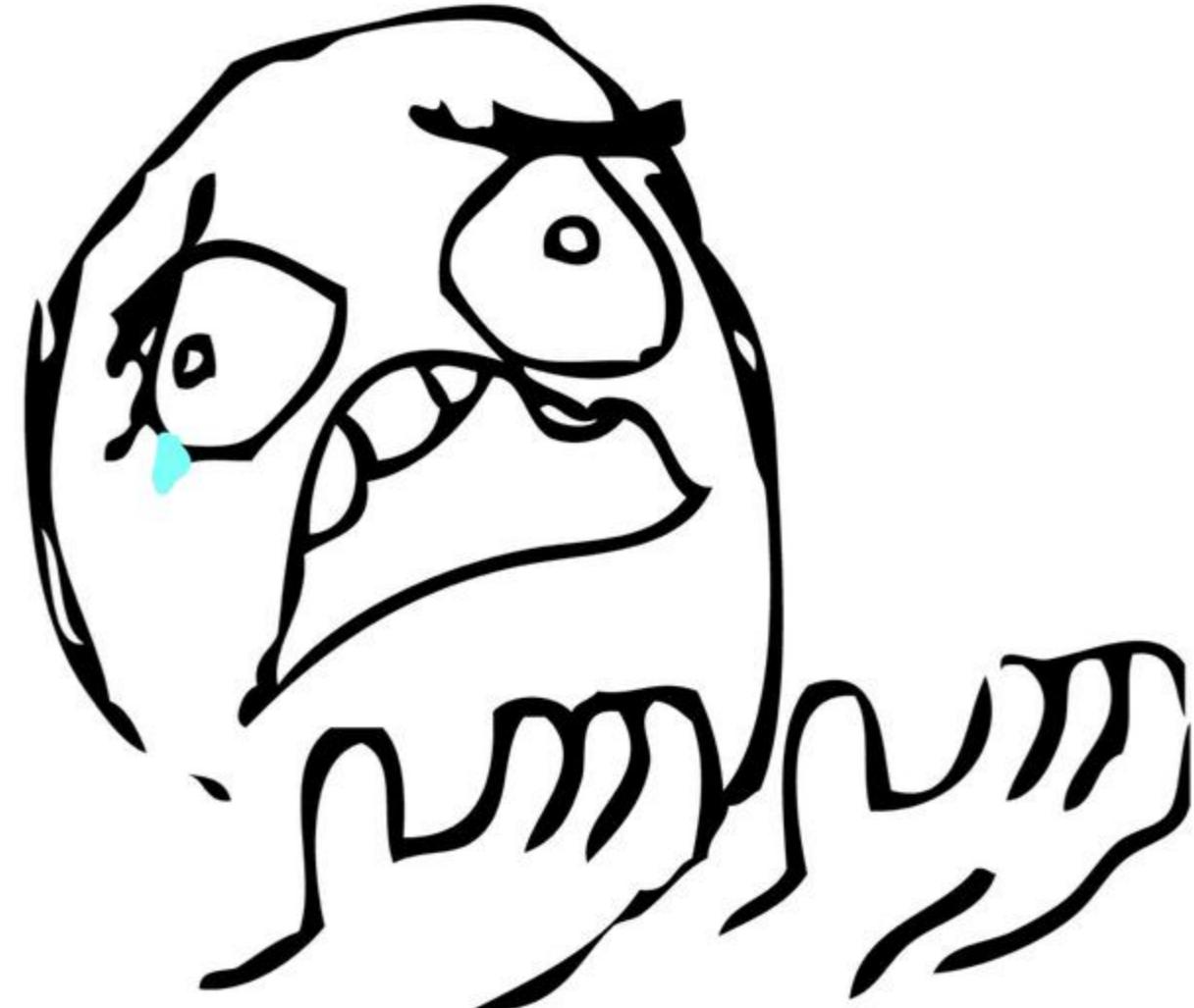
- Host port binding : à vous de maintenir la liste des ports utilisés
 - auto-mapping (`docker container run --publish-all nginx`) : à vous de trouver comment récupérer les ports choisis par Docker
- utiliser des mécanismes de *service discovery* (Consul, Etcd, Zookeeper, ...)

COMMENT...



- Comment gérer les conteneurs arrêtés suite à une erreur ?
- Comment gérer les pannes des hosts ?
- Comment gérer la maintenance de vos hosts ?
- Comment gérer les mises à l'échelle (scale up/down) ?
- Comment gérer la multiplication du nombre de composants à déployer ?
- Comment gérer les mises à jour de vos composants ?
- Comment partager vos process de déploiement ?
- Comment faire pour que les développeurs/testeurs/opérateurs puissent être autonomes ?

THERE'S NOT SUCH THING AS FREELUNCH !



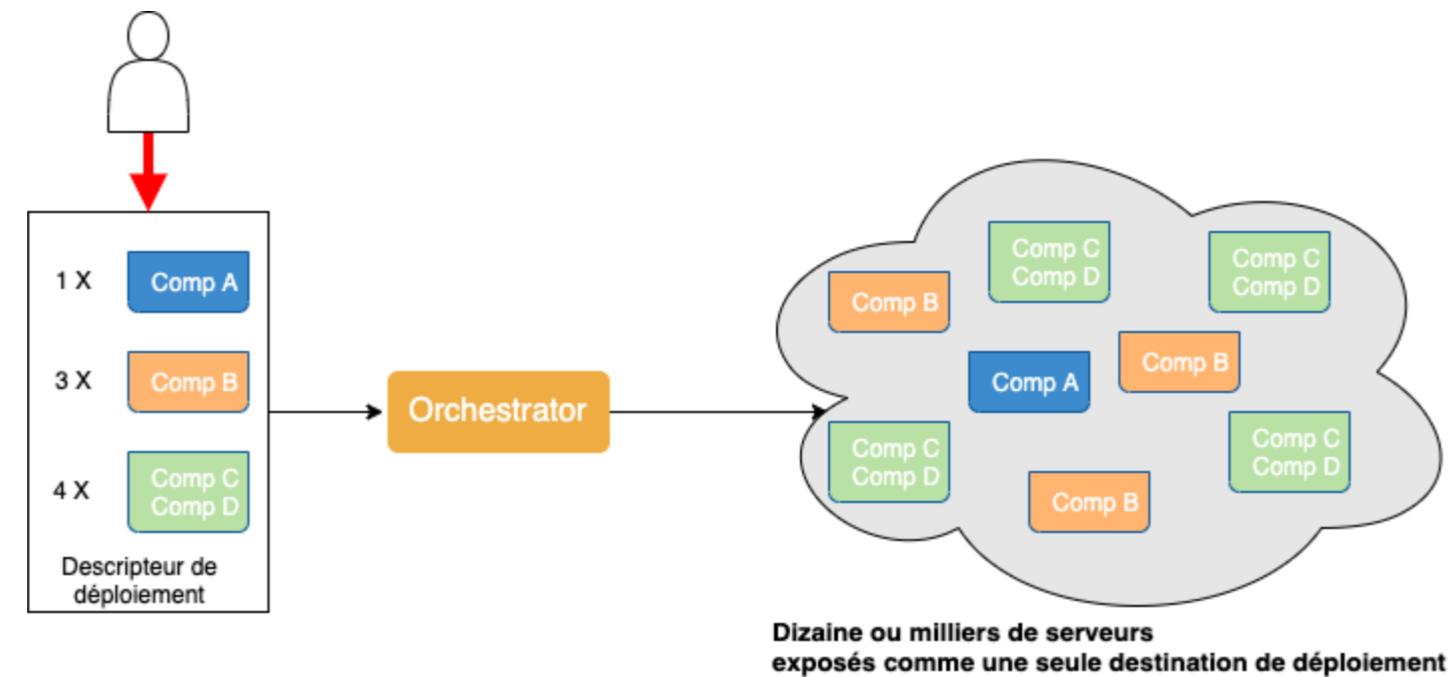


UNE SOLUTION : L'UTILISATION D'UN ORCHESTRATEUR

- Les orchestrateurs répondent à toutes les problématiques citées précédemment
- Les orchestrateurs sont souvent présentés comme des systèmes d'exploitation des datacenters
- Ce sont des outils facilitant la gestion/le pilotage du cycle de vie de vos composants conteneurisés



L'ORCHESTRATION VUE PAR L'UTILISATEUR



PLACEMENT ET GESTION DU CYCLE DE VIE DES CONTENEURS



L'orchestrateur choisit où faire tourner vos composants en fonction des contraintes que vous positionnez :

- contrainte technique : réseau, type de matériel (SSD/HDD), ...
- contrainte de co-localisation
- nombre de replicas
- disponibilités des ressources CPU / mémoire / stockage



GESTION DU FAILOVER

- L'orchestrateur s'occupe de redémarrer les composants stoppés anormalement
- que ce soit des conteneurs
- ou des noeuds / hosts
- L'orchestrateur propose aussi des fonctionnalités de haute-disponibilité des noeuds de management



FONCTIONNALITÉS RÉSEAU

- Load Balancing
- Mécanismes de Service Discovery
- Fourniture de solutions réseau (de type *overlay* la plupart du temps) qui permettent à vos conteneurs de communiquer entre eux sans avoir à passer par des ports des noeuds les hébergeant

FONCTIONNALITÉS D'ORCHESTRATION



- Gestion du Stockage distribué/persistent
- Gestion des Secrets et de Configuration distribuée
- Mise à l'échelle manuelle / automatique
- Fourniture d'une ligne de commande (CLI) et d'une api REST facilitant l'automatisation
- Descripteurs de déploiement sous forme de code
- *Role Based Access Control*: gestion des droits d'accès aux ressources mises à disposition

KUBERNETES N'EST PAS LE SEUL ORCHESTRATEUR DE CONTENEURS



- Docker Swarm de chez Docker Inc.
- Elastic Container Service (ECS) d'Amazon
- Nomad de chez Hashicorp
- Cattle de chez Rancher (obsolète)
- Mesos + Marathon de chez D2iQ (obsolète)
- Et bien d'autres...

KUBERNETES



DÉFINITION DE KUBERNETES



Kubernetes est une plateforme open-source conçue pour automatiser, mettre à l'échelle et opérer des composants applicatifs conteneurisés



HISTORIQUE

- Google a initié le projet Kubernetes en 2014
- Kubernetes s'appuie sur 15 années d'expérience pendant lesquelles Google a fait tourner en production des applications à grande échelle :
 - Borg
 - Omega
- Kubernetes a été enrichi au fil du temps par les idées et pratiques mises en avant par une communauté très active

SIGNIFICATION



- Le nom *Kubernetes* vient du grec, et signifie *Timonier* ou *Pilote*
 - *k8s* est une abbréviation dérivée du remplacement des *8* lettres de “*ubernetes*” par un “*8*”
 - Prononciation koo-ber-nay'-tisse



RESSOURCES



- [Le site officiel](#)
- [La documentation](#) :
 - [Concepts](#)
 - [Tutoriaux](#)
 - [Tâches communes](#)
 - [API et kubectl](#)
- [Le code source](#)
- [Le compte twitter](#)
- [Le canal Slack](#)
- [La section StackOverflow](#)



DISTRIBUTIONS (1/2)

- [kubeadm](#) et [minikube](#)
- [Mirantis Kubernetes Engine](#)
- [Docker Desktop](#)
- [kind](#)
- [GKE, AKS, EKS](#)
- [Rancher 2.x, RKE](#) [Rancher Kubernetes Engine](#)
- RedHat OpenShift v4 ([Kubernetes Distribution](#) et [Container Platform](#))



DISTRIBUTIONS (2/2)

- [PKS /Pivotal Container Service](#)
- [kops](#) (Kubernetes Operations)
- Playbooks ([kubespray](#) et [kubernetes/contrib](#)) et [modules](#) Ansible, [templates](#) et [providers](#) Terraform
- [K3S](#) de Rancher
- [MicroK8s](#) de Canonical
- [Kubernetes Distributions and Platforms](#)

KUBERNETES THE HARD WAY



<https://github.com/kelseyhightower/kubernetes-the-hard-way> par Kelsey Hightower

- Tutoriel d'installation *manuelle* de Kubernetes
- Optimisé pour l'apprentissage :
 - choix de passer par une installation longue et détaillée pour bien comprendre chacune des étapes nécessaires à la création d'un cluster Kubernetes

KUBERNETES ET DOCKER



- Kubernetes va permettre de gérer Docker (mais pas que) sur plusieurs noeuds. Pour ce faire, il y a une compatibilité minimum à respecter entre les deux produits.
- Sur la version **1.20** (12/2020) de Kubernetes, les versions récentes de Docker sont gérées (à partir de la version 1.13.1) mais maintenant utiliser Docker en tant que container-runtime est **déprécié**

KUBERNETES SANS DOCKER



Il est possible d'utiliser d'autres solutions de conteneurisation en lieu et place de Docker :

- [CRI-O](#) est un projet qui s'appuie sur la [Container Runtime Interface](#) de l'[Open Container Initiative](#) ([Six reasons why cri-o is the best runtime for k8s](#))
- [containerd](#), issue de Docker, projet de la [CNCF](#)
- [frakti](#), moteur de conteneurs via hyperviseur



VERSIONS DE KUBERNETES

- [v1.27.0](#) le 2023-04-11
- [v1.26.0](#) le 2022-12-09
- [v1.25.0](#) le 2022-08-23
- [v1.24.0](#) le 2022-05-03
- [v1.23.0](#) le 2021-12-07
- [v1.22.0](#) le 2021-08-04
- ...
- v1.0.0 le 2015-07-13

- [Tous les Changelogs](#)
- [Toutes les Releases](#)

VERSION DE KUBERNETES UTILISÉE POUR LA FORMATION



La formation est écrite par rapport à la version **v1.23**.





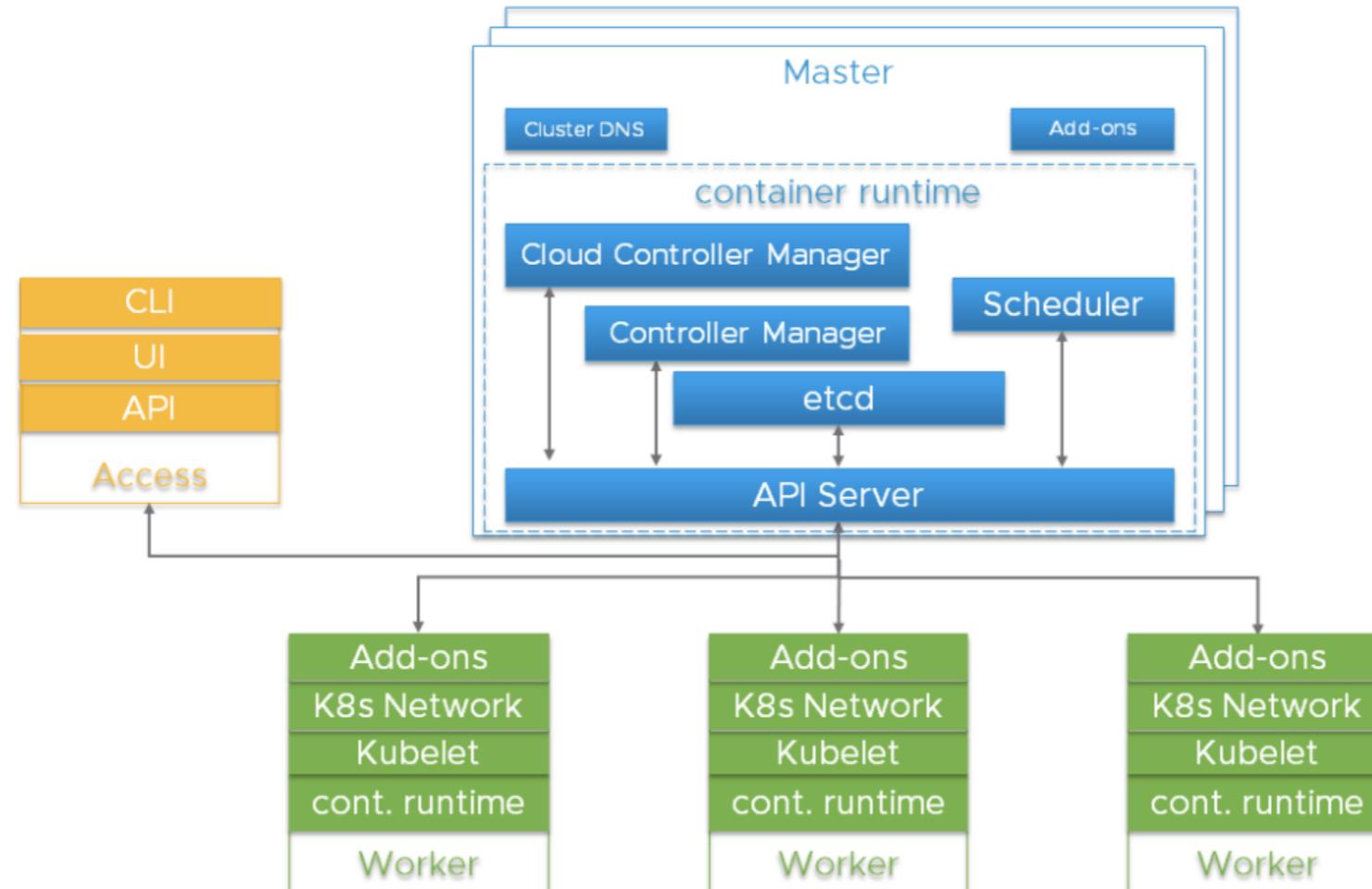
ARCHITECTURE

AGENDA DE CE CHAPITRE 'ARCHITECTURE'



- Architecture
- Control Plane
- Workers

ARCHITECTURE





API SERVER

L'API Server est le point d'entrée du cluster Kubernetes. C'est par lui que passe toutes les requêtes de consultation et de modifications.

Il gère :

- l'authentification
- les droits
- la validation syntaxique et sémantique des requêtes



ETCD

Etcd est la base de données de Kubernetes. Il stocke :

- les descripteurs des applications que vous déployez sur le cluster
- l'état actuel de ce qui est en cours d'exécution sur cluster

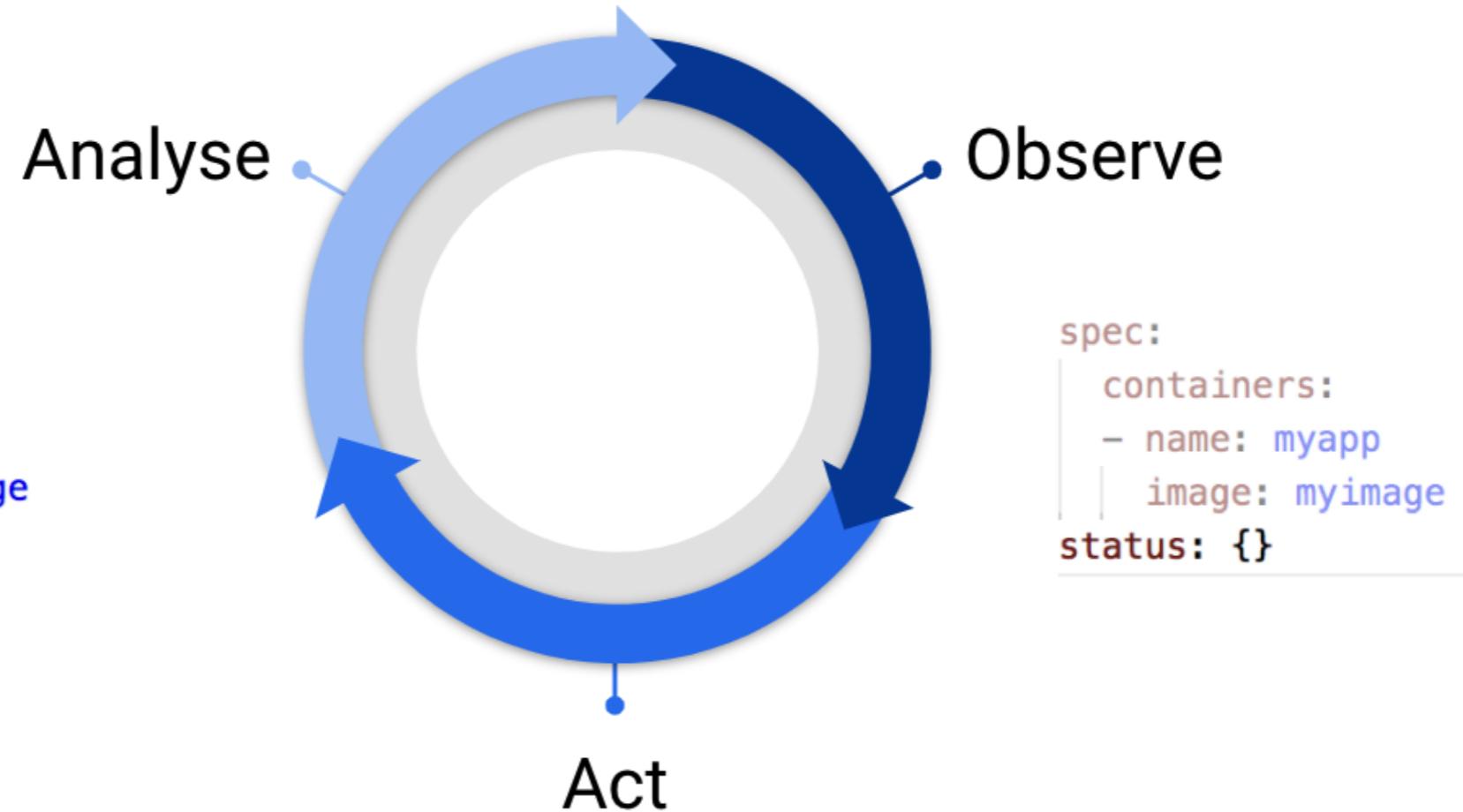
Etcd n'est jamais accédé en direct, mais toujours via l'API Server



CONTROLLER MANAGER

- Implémente la boucle de contrôle pour les charges de travail
(ReplicationController, ReplicaSet, Deployment, ...) et autres ressources
- Fonctionne en mode actif / passif
 - Le contrôleur actif actuel est signalé dans **etcd**

CONTROLLER MANAGER



API KUBERNETES ET VERSIONING D'API



- Afin de faciliter les évolutions telles que les ajouts/suppressions de champs ou les restructurations, Kubernetes supporte plusieurs versions d'API
- Chaque version correspond à un chemin différent, ex: `/api/v1` ou `/apis/autoscaling/v2beta2`
- Différentes versions impliquent différents niveaux de stabilité

[API Overview Reference](#)

FONCTIONNALITÉS EN VERSION ALPHA



- Le nom de version contient **alpha** (par exemple, **v1alpha1**)
- Activer une fonctionnalité **alpha** peut vous exposer à des bugs
- Ces fonctionnalités peuvent être désactivées par défaut
- Le support pour ces fonctionnalités peut s'arrêter sans avertissement
- L'API pourra changer en apportant des modifications non rétro-compatibles avec une version précédente
- Aucune garantie de support à long terme, a priori, à ne pas utiliser sur des instances de production sauf si vous êtes prêts à accepter les risques associés

FONCTIONNALITÉS EN VERSION BETA (1/2)



- Le nom de version contient **beta** (par exemple, **v2beta3**)
- La fonctionnalité a été *bien* testée, est considérée comme viable et est activée par défaut
- La fonctionnalité restera en place, des détails peuvent cependant changer d'ici le passage en niveau stable
- Des modifications peuvent être apportées dans le schéma ou la sémantique des objets en rapport avec la fonctionnalité dans une version beta ou stable suivante. Cependant, des instructions de migration seront fournies (mais peuvent impliquer un import/export ou une interruption du service utilisant la fonctionnalité)

FONCTIONNALITÉS EN VERSION BETA (2/2)



- La communauté Kubernetes vous invite à tester ces fonctionnalités dès qu'elles sont en beta car il sera difficile de revenir en arrière quand elles seront passées en niveau stable
- A priori, à ne pas utiliser sur des instances de production sauf si vous êtes prêts à accepter les risques associés

FONCTIONNALITÉS EN VERSION STABLE



- Le nom des versions stables est de la forme vX où X est un entier
- Les versions stables des fonctionnalités sont pérennes



GROUPES D'API

- Les **API groups** permettent de séparer la monolithique **API v1** en plusieurs sous-parties qui peuvent être activées/désactivées séparément (**Supporting multiple API Groups**)
- Plusieurs groupes existent :
 - le groupe **core** (aussi appelé **legacy**) est le groupe principal et n'est pas spécifié explicitement dans le chemin REST (**/api/v1**) ou le champ **apiVersion** dans les descripteurs (**apiVersion: v1**)
 - les groupes nommés sont visibles dans les chemins REST (**/apis/GROUP/VERSION**), et dans l'**apiVersion** (**GROUP/VERSION**), par exemple **apiVersion: batch/v1**



EXEMPLES DE GROUPES

batch, extensions, authorization, autoscaling, policy

- [Kubernetes v1.27 API reference](#)
- [Kubernetes v1.26 API reference](#)
- [Kubernetes v1.25 API reference](#)
- [Kubernetes v1.24 API reference](#)
- [Kubernetes v1.23 API reference](#)
- [Kubernetes v1.22 API reference](#)

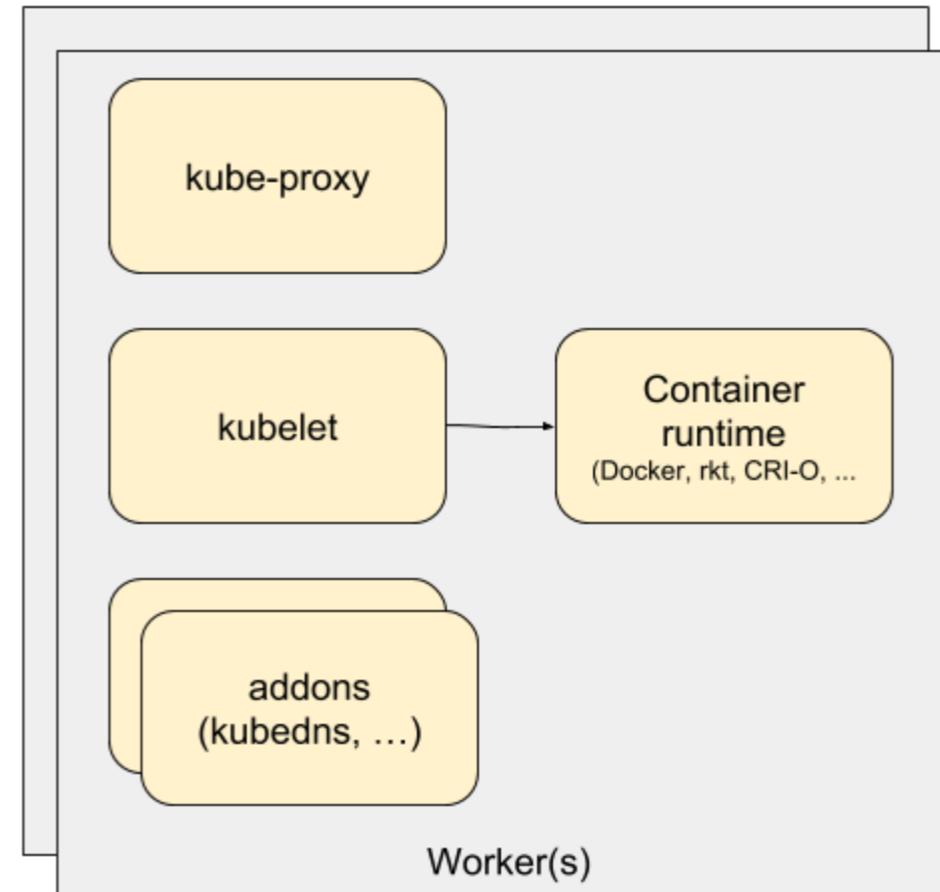
EVOLUTION DE L'API ET DES VERSIONS ASSOCIÉES



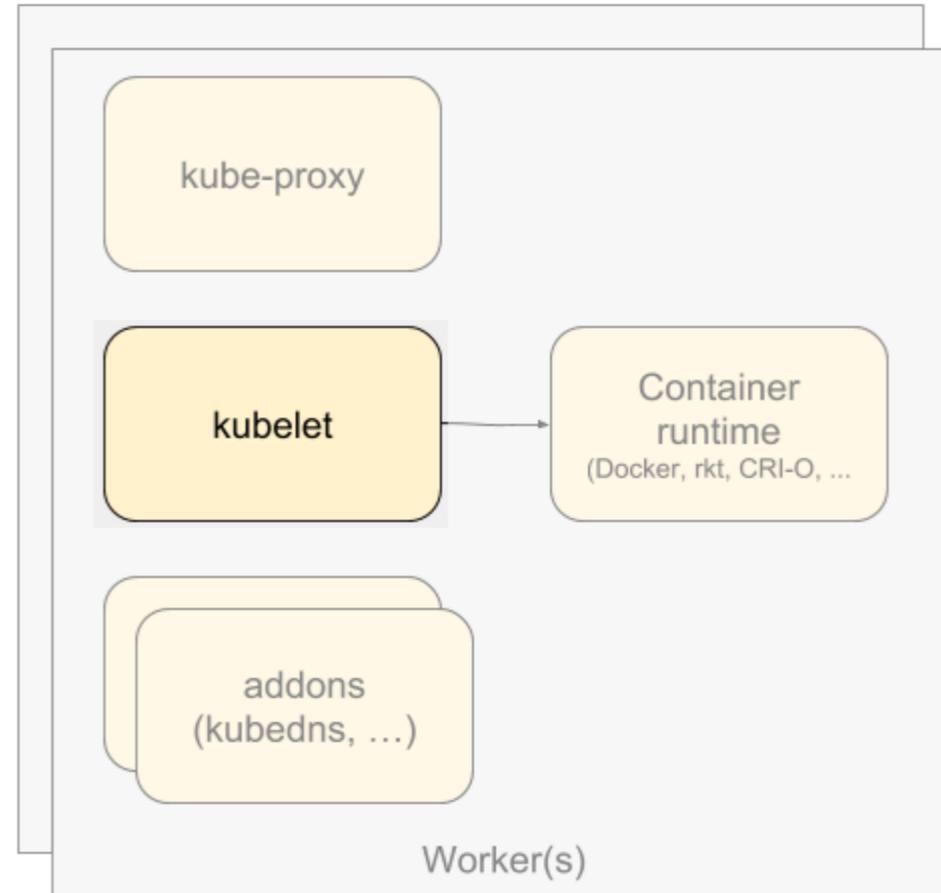
 Si vous êtes intéressés par les stratégies de gestion des versions d'api :

- [Kubernetes Deprecation Policy](#)

WORKERS



KUBELET





KUBELET (1/2)

- C'est le **Node agent**, il s'exécute sur un nœud et assurez-vous que les conteneurs inclus dans les **PodSpecs** affectés sont en cours d'exécution et en bon état.
- Les **PodSpecs** sont fournis au **kubelet** en interrogeant le **API Server**
- Ces communications utilisent un **kubeconfig** comme tout client **API Server**



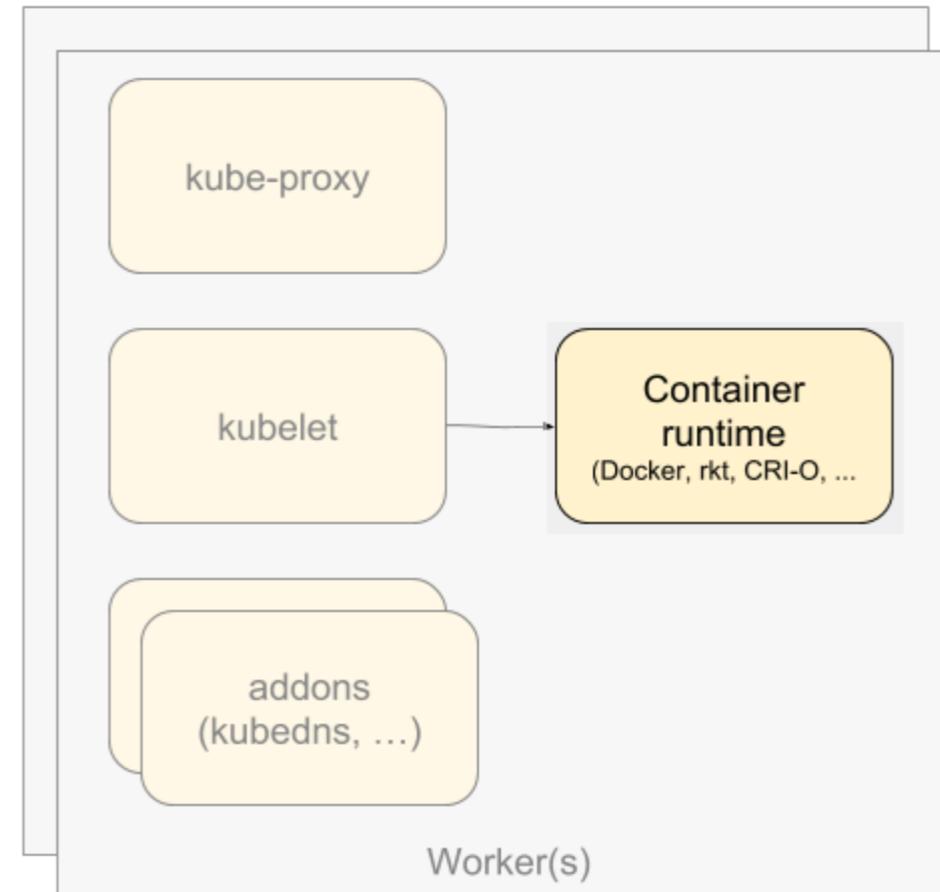
KUBELET (2/2)

Il existe également d'autres moyens de fournir des **PodSpecs** à **kubelet**:

- Fichier: **pod-manifest-path** passé à la ligne de commande est surveillé toutes les 20s par défaut
- HTTP Endpoint: **manifest-url** passé à la ligne de commande et vérifié par le kubelet toutes les 20s par défaut
- Serveur HTTP: le kubelet expose une API utilisée pour interagir avec des **Pods** qui peut être utilisée pour créer de nouveaux **Pods**

⚠ Le serveur HTTP sur le kubelet permet la création de pods *sans authentification*. Il doit être **disabled** ou au moins sécurisé

CONTAINER RUNTIME



CONTAINER RUNTIME



Le **kubelet** gère les runtimes de conteneur qui implémentent le '**Container Runtime Interface (CRI)**'

- containerd
- cri-o
- docker

Il est également possible d'utiliser plusieurs environnements d'exécution de conteneur en parallèle.





PREMIERS PAS AVEC KUBERNETES

AGENDA DE CE CHAPITRE 'PREMIER PAS'



- Installation de Kubernetes
- Dashboard et CLI
- Démarrer un conteneur



KUBEADM (1/2)

- Distribué en un seul binaire
- Maintenu par la communauté Kubernetes
- Depuis **Kubernetes v1.11** peut déployer un cluster **High Availability**
- Suit le cycle de publication de Kubernetes



KUBEADM (2/2)

- kubeadm va:
 - Générer des certificats
 - Déployer kubelet sur le plan de contrôle / nœuds de travail
 - Déployer le cluster etcd et les composants du plan de contrôle en tant que pods statiques
 - Fournir un moyen simple de joindre les nœuds de travail au cluster
- kubeadm ne va pas:
 - Créer des VM
 - Configurer l'infrastructure réseau
 - Installer le runtime du conteneur sur les machines



TP 1.1 : Installation

DASHBOARD ET CLI



Kubernetes Dashboard Piotr Bryk

localhost:9090/#/pod?namespace=kube-system

kubernetes Workloads > Pods + CREATE

Admin

- Namespaces
- Nodes
- Persistent Volumes

Namespace

- kube-system

Workloads

- Deployments
- Replica Sets
- Replication Controllers
- Daemon Sets
- Stateful Sets
- Jobs
- Pods**

CPU usage

Memory usage

Pods

Name	Status	Restarts	Age	CPU (cores)	Memory (bytes)
dashboard-same-i...	Running	0	14 minutes	0	5.4 Mi
fluentd-cloud-log...	Running	0	2 months	0.01	120.8 Mi
fluentd-cloud-log...	Running	0	2 months	0.009	94.3 Mi
fluentd-cloud-log...	Running	0	2 months	0.014	174.3 Mi
heapster-v1.2.0-4...	Running	0	10 days	0.002	44.8 Mi

LE DASHBOARD KUBERNETES



- Le *Dashboard* est une interface web permettant d'interagir avec une instance Kubernetes
- Le *Dashboard* permet de :
 - déployer des applications
 - rechercher des informations suite au comportement anormal d'une application déployée
 - visualiser l'ensemble des applications déployées
 - modifier la configuration des applications déployées et les mettre à jour
- Le *Dashboard* permet aussi de connaître l'état des ressources d'une instance et d'accéder aux logs des composants du cluster ainsi que ceux des applications
- Pour que le *Dashboard* puisse afficher les métriques et les graphiques associés, le metrics-server doit être installé dans le cluster



DASHBOARD VS CLI (KUBECTL)

- Toutes les informations disponibles et les actions réalisables par le biais du *Dashboard* le sont aussi en passant par la ligne de commande `kubectl`
- `kubectl` permet d'interagir en ligne de commande avec vos instances *Kubernetes*
- La documentation en ligne est disponible [ici](#)



KUBECTL : LISTER LES COMMANDES

```
$ kubectl  
kubectl controls the Kubernetes cluster manager.
```

Find more information at: <https://kubernetes.io/docs/reference/kubectl/overview/>

Basic Commands (Beginner):

create	Create a resource from a file or from stdin.
expose	Take a replication controller, service, deployment or pod and expose it as a new Kubernetes Service
run	Run a particular image on the cluster
set	Set specific features on objects

Basic Commands (Intermediate):

explain	Documentation of resources
get	Display one or many resources
edit	Edit a resource on the server
delete	Delete resources by filenames, stdin, resources and names, or by resources and label selector
(...)	

KUBECTL : AIDE EN LIGNE DE COMMANDE



- Lancer `kubectl` pour voir la liste de toutes les commandes disponibles
- Lancer `kubectl <command> --help` pour voir l'aide pour une commande en particulier
- Lancer `kubectl options` pour voir la liste des options globales (qui s'appliquent à toutes les commandes)



TP 1.2 : Kubectl



TP 1.3 : Docker Warmup et Rappels

DÉMARRER SON PREMIER CONTENEUR DANS K8S



- Le meilleur moyen de déployer une application dans *Kubernetes* de manière reproductible est de passer par un fichier descripteur au format `yaml` ou `json`
- Mais il est aussi possible d'utiliser la commande `run` qui permet de se passer d'un tel descripteur



UTILISATION DE 'KUBECTL RUN'

La commande :

```
$ kubectl run whoami --image=traefik/whoami:v1.8 --port=80
```

- démarre un *Pod*(que l'on verra en détail dans le chapitre **Pods**)
- à partir de l'image Docker **traefik/whoami:v1.8**
- et référence le port **80** du conteneur créé
 - "*référence*" dans le sens où c'est une déclaration d'intention permettant d'utiliser par la suite ce port
 - Ne pas spécifier de port n'empêche pas le port d'être accessible



EXEMPLES D'UTILISATION DE 'KUBECTL RUN'

- `kubectl run nginx --image=nginx`

Démarre une instance unique à partir de l'image `nginx`

- `kubectl run hazelcast --image=hazelcast --port=5701`

Démarre une instance `hazelcast` et expose le port `5701` du conteneur

- `kubectl run hazelcast --image=hazelcast --env="DNS_DOMAIN=cluster"`

Démarre une instance `hazelcast` en spécifiant des variables d'environnement

- `kubectl run nginx --image=nginx --dry-run=client --output yaml`

Affiche les descripteurs sans réellement créer les objets



'KUBECTL GET CONTAINERS' ?

- Il n'existe pas de commande `kubectl` permettant de lister les conteneurs lancés
- L'unité la plus petite est le `Pod` (que l'on verra en détail dans le chapitre `Pods`)
- On peut donc lancer `kubectl get pods` (ou `kubectl get po` en utilisant le raccourci de la ressource `Pods`)



LISTER LES PODS

```
$ kubectl get pods  
NAME      READY   STATUS    RESTARTS   AGE  
whoami    1/1     Running   0          2m
```

À noter, par défaut, les informations remontées sont :

- Le nom du Pod
- Le nombre de conteneurs *prêts* (un Pod peut contenir plusieurs conteneurs)
- Le status du Pod (Running, Terminating, CrashLoopBackOff, ...)
- Le nombre de *restarts* éventuels de l'un des conteneurs du Pod
- L'âge du Pod : depuis combien de temps la demande de création du Pod a-t-elle été prise en compte

LISTER PLUS D'INFORMATIONS SUR LES PODS ?



Il est possible d'utiliser l'option **--output** (ou **-o**) pour affiner le format de sortie

- **json** ou **yaml**
- **wide** ou **name**
- **custom-columns=...** ou **custom-columns-file=...** ([Documentation](#))
- **go-template=...** ou **go-template-file=...** ([Documentation](#))
- **jsonpath=...** ou **jsonpath-file=...** ([Documentation](#))



PLUS D'INFOS SUR LES PODS (SUITE)

```
$ kubectl get pods --output wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
whoami	1/1	Running	0	1h	172.17.0.7	minikube

```
$ kubectl get pods -o name
```

pods/whoami

```
$ kubectl get pods \
```

```
  -o=custom-columns=NAME:.metadata.name,RSRC:.metadata.resourceVersion
```

NAME	RSRC
whoami	103751

ACCÉDER À TOUS LES DÉTAILS D'UN POD



La commande `kubectl describe pod <pod-name>` permet d'accéder à des informations détaillées sur un Pod

```
$ kubectl describe pod whoami
Name:           whoami
Namespace:      default
Node:          minikube/192.168.99.100
Start Time:    Sun, 08 Oct 2017 19:02:51 +0200
Labels:         run=whoami
Annotations:   kubernetes.io/created-by=
{"kind": "SerializedReference", "apiVersion": "v1", "reference": {
  "kind": "ReplicaSet", "namespace": "default", "name": "whoami", "uid": "850adefb-ac4a-11e7-b4c7-0800279f22af", "a...
Status:        Running
IP:            172.17.0.7
.
.
.
Node-Selectors: <none>
Tolerations:   <none>
Events:
```



TYPES DE RESSOURCES VALIDES

\$ kubectl api-resources	NAME	SHORTNAMES	APIVERSION	NAMESPACED	KIND
	configmaps	cm	v1	true	ConfigMap
	endpoints	ep	v1	true	Endpoints
	limitranges	limits	v1	true	LimitRange
	namespaces	ns	v1	false	Namespace
	nodes	no	v1	false	Node
	persistentvolumeclaims	pvc	v1	true	PersistentVolumeClaim
	persistentvolumes	pv	v1	false	PersistentVolume
	pods	po	v1	true	Pod
	secrets		v1	true	Secret
	services	svc	v1	true	Service
	daemonsets	ds	apps/v1	true	DaemonSet
	deployments	deploy	apps/v1	true	Deployment
	replicasets	rs	apps/v1	true	ReplicaSet
	statefulsets	sts	apps/v1	true	StatefulSet
	horizontalpodautoscalers	hpa	autoscaling/v1	true	HorizontalPodAutoscaler
	cronjobs	cj	batch/v1	true	CronJob
	jobs		batch/v1	true	Job
	ingresses	ing	networking.k8s.io/v1	true	Ingress
	networkpolicies	netpol	networking.k8s.io/v1	true	NetworkPolicy
	(...)				

EXPOSER L'APPLICATION DÉMARRÉE (CONTEXTE)



- Jusqu'à présent, l'utilisation de `kubectl run` a créé un Pod
- L'application `whoami` n'est pour l'instant accessible que depuis l'intérieur du cluster `Kubernetes`, par le biais de l'adresse IP attribuée au Pod

```
$ kubectl describe pod whoami | grep IP  
IP: 172.17.0.7
```

PROBLÉMATIQUES LIÉES À L'ACCÈS DES PODS



- Comment faire si on veut accéder à l'application *whoami* de manière fiable ?
 - l'ip associée au Pod n'est pas stable dans le temps, si le Pod était redémarré, l'adresse pourrait changer
 - dans le cas d'une mise à l'échelle, nous aurions plusieurs Pods avec plusieurs IP, comment faire pour s'y connecter ?
- ⇒ La solution : passer par un *Service* (les détails seront vus dans le chapitre *Services*).



TP 1.4 : kubectl run





PODS



AGENDA DE CE CHAPITRE 'PODS'

- Modèle/Concept du Pod
- Descripteurs yaml et json
- Organisation des Pods avec les labels, les sélecteurs et les Namespaces
- SecurityContext
- InitContainers
- Cycle de vie des Pods et HealthChecks

MODÈLE/CONCEPT DU POD



- Un *Pod*(une cosse, une gousse, une coque) est un groupe d'un ou plusieurs conteneurs (Docker par ex)
- Le Pod est la brique de base d'un déploiement k8s
- C'est la brique de déploiement la plus petite que l'on puisse créer / déployer

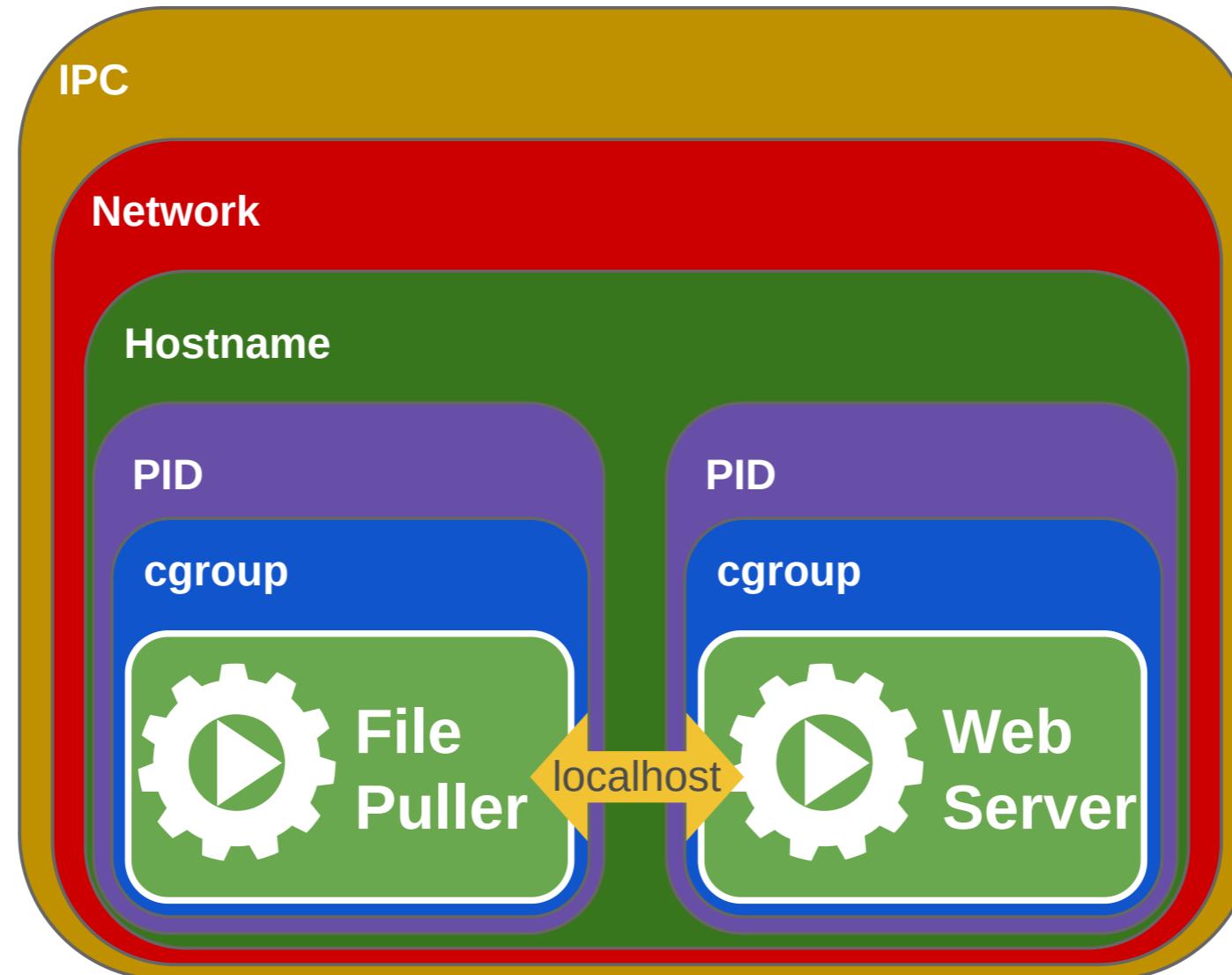
Le concept du Pod permet à des conteneurs de partager des ressources afin de pouvoir collaborer plus facilement.

PARTAGE DE NAMESPACE ENTRE LES CONTENEURS D'UN POD

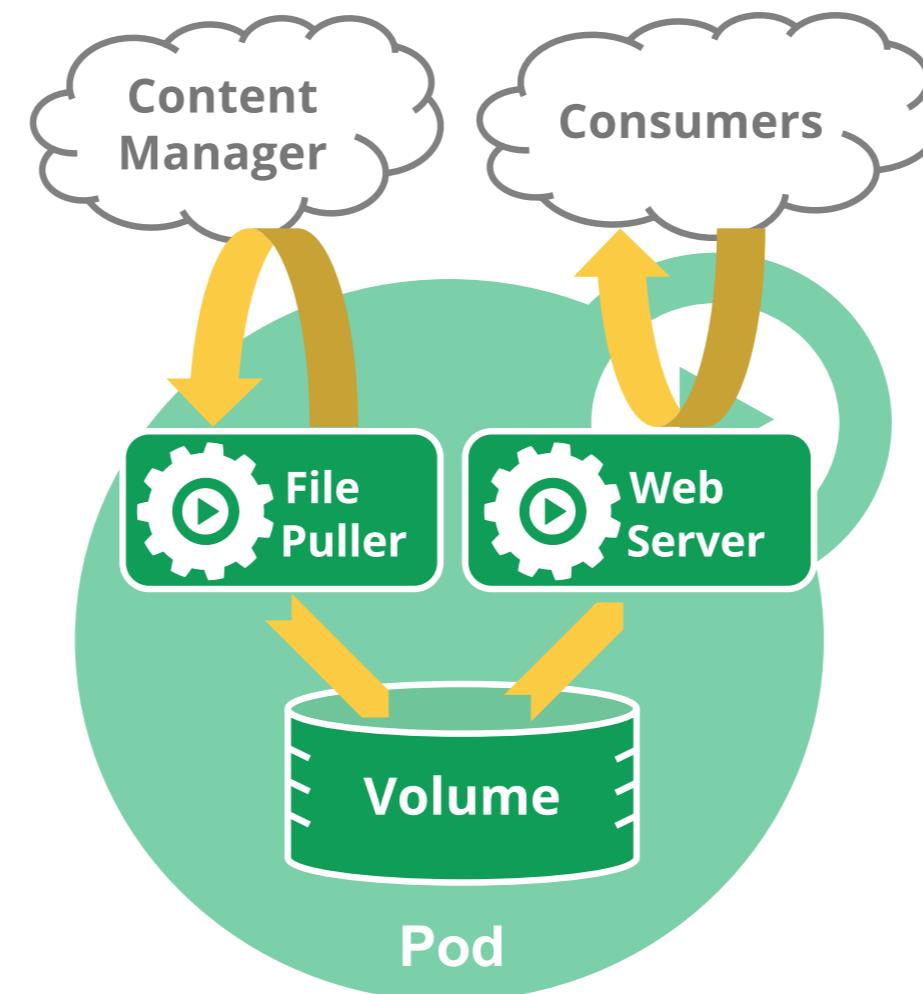


- Les conteneurs au sein d'un Pod partagent le même IPC et peuvent donc communiquer en utilisant les sémaphores *SystemV* ou la mémoire partagée *POSIX*
- Les conteneurs au sein d'un Pod partagent la même adresse IP, le même espace de ports et peuvent se 'trouver' par le biais de *localhost*
- Les conteneurs au sein d'un Pod peuvent accéder aux mêmes volumes
- Les conteneurs d'un Pod sont *toujours* co-localisés et co-gérés

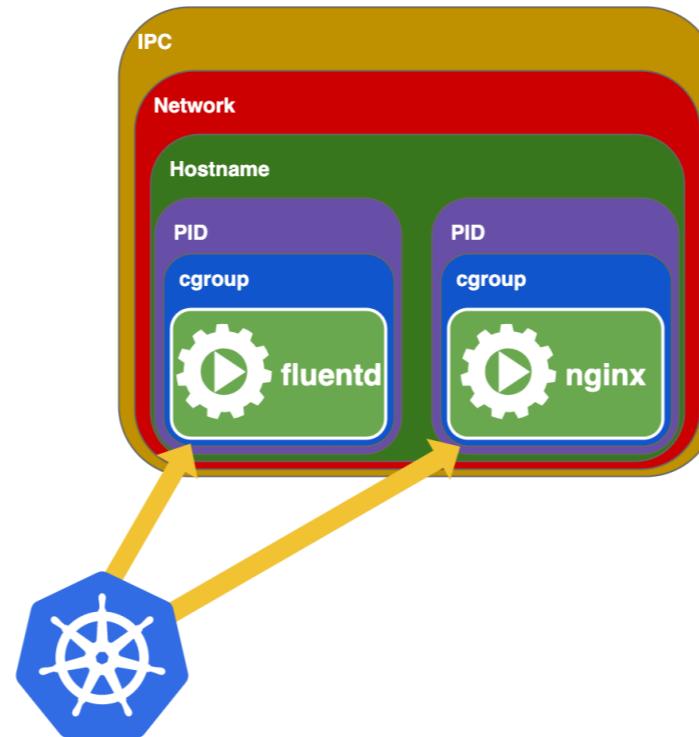
POD MULTI-CONTENEUR PARTAGEANT LES MÊMES RESSOURCES



POD MULTI-CONTENEUR PARTAGEANT UN MÊME VOLUME



CONTENEURS DU POD VUS PAR KUBERNETES



- Grace au modèle des Pods, Kubernetes supervise chacun des process et connaît donc les états de chacun d'entre eux
- De cette manière, il est possible de redémarrer individuellement les process/conteneurs si besoin (ou d'en voir les logs !)



COMMENT CHOISIR ENTRE PLUSIEURS PODS OU PLUSIEURS CONTENEURS DANS UN POD ?

Se poser les questions :

- Les conteneurs doivent-ils tourner au même endroit ou peuvent-ils être répartis sur plusieurs noeuds ?
- Les conteneurs représentent-ils un seul tout ou sont-ils des composants indépendants ?
- Les conteneurs peuvent-ils/doivent-ils être mis à l'échelle ensemble individuellement ?

La plupart du temps, vous devriez séparer vos conteneurs dans des Pods distincts à moins qu'il n'y ait une raison particulière de ne pas le faire !

CAS D'USAGE DE PLUSIEURS CONTENEURS DANS UN POD

- Recharger de la configuration à chaud sur déclenchement d'évènement
- Conteneur web (Nginx, Apache, ...) + conteneur **php-fpm**
- Service routing
- Agrégation de logs
- Télémétrie

Le gros avantage est de ne pas avoir à modifier l'application principale

Dans ce cadre d'utilisation, les conteneurs associés au conteneur principal sont parfois appelés **sidecar containers** ou **sidekick containers**



DESCRIPTEURS YAML ET JSON

- Au lieu d'utiliser `kubectl run xxx ...`
- il est possible de s'appuyer sur des descripteurs au format `yaml` ou `json`



EXEMPLE DE DESCRIPTEUR AU FORMAT JSON

```
{  
  "apiVersion": "v1",  
  "kind": "Pod",  
  "metadata": {  
    "name": "whoami",  
    "labels": { "app": "whoami" }  
  },  
  "spec": {  
    "containers": [  
      {  
        "name": "whoami",  
        "image": "containous/whoami:latest",  
        "imagePullPolicy": "Always",  
        "ports": [  
          { "containerPort": 80 },  
          { "containerPort": 443 }  
        ]  
      }  
    ],  
    "restartPolicy": "Always"  
  }  
}
```

EXEMPLE DE DESCRIPTEUR AU FORMAT YAML



```
---
apiVersion: v1
kind: Pod
metadata:
  name: whoami
  labels:
    app: whoami
spec:
  containers:
    - name: whoami
      image: containous/whoami:latest
      imagePullPolicy: Always
      ports:
        - containerPort: 80
        - containerPort: 443
  restartPolicy: Always
```

RAPPEL : OBTENIR UN TEMPLATE DE DESCRIPTEUR



```
kubectl run whoami \
--image=containous/whoami:latest \
--port=80 \
--dry-run=client \
--output yaml
```

ANATOMIE D'UN FICHIER DESCRIPTEUR



- Le descripteur `yaml` peut se décomposer en plusieurs sous-parties :
 - `apiVersion`
 - `kind`
 - `metadata`
 - `spec, data, rules` ou `stringData`
- Ces sous-parties se retrouvent dans tous les types de ressource

UTILISER 'KUBECTL EXPLAIN' POUR ACCÉDER À LA DOCUMENTATION DES RESSOURCES



```
$ kubectl explain pods
```

DESCRIPTION:

Pod is a collection of containers that can run on a host. This resource is created by clients and scheduled onto hosts.

FIELDS:

metadata <Object>

Standard object's metadata. More info:

<https://git.k8s.io/community/contributors/devel/api-conventions.md#metadata>

spec <Object>

Specification of the desired behavior of the Pod. More info: <https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-and-status/>

status <Object>

Most recently observed status of the Pod. This data may not be up to date.

Populated by the system. Read-only. More info: <https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-and-status/>

(...)

UTILISER LES PATHS JSON POUR ALLER PLUS LOIN



```
$ kubectl explain pods.spec.containers
```

RESOURCE: containers <[]Object>

DESCRIPTION:

List of containers belonging to the Pod. Containers cannot currently be added or removed. There must be at least one container in a Pod. Cannot be updated.

A single application container that you want to run within a Pod.

FIELDS:

workingDir <string>

Container's working directory. If not specified, the container runtime's default will be used, which might be configured in the container image. Cannot be updated.

command <[]string>

Entrypoint array. Not executed within a shell. The docker image's ENTRYPOINT is used if this is not provided. Variable references \$(VAR_NAME) (...)

CONTRAINTES SUR CERTAINES VALEURS DANS LES DESCRIPTEURS



```
$ kubectl explain pods.spec.containers
```

```
...  
FIELDS:
```

```
...  
imagePullPolicy <string>  
  Image pull policy. One of Always, Never, IfNotPresent. Defaults to Always  
  if :latest tag is specified, or IfNotPresent otherwise. Cannot be updated.  
  More info:  
  https://kubernetes.io/docs/concepts/containers/images#updating-images
```

```
name <string> -required-  
  Name of the container specified as a DNS_LABEL. Each container in a Pod  
  must have a unique name (DNS_LABEL). Cannot be updated.
```

```
...
```



DNS_LABEL

La contrainte **specified as a DNS_LABEL** fait référence à la [rfc1123](#)

a DNS-1123 label must consist of lower case alphanumeric characters or '-', and must start and end with an alphanumeric character (e.g. `my-name`, or `123-abc`, regex used for validation is `[a-z0-9] ([-a-z0-9]*[a-z0-9])?`)

Attention : Kubernetes impose d'utiliser des valeurs **lower case** (voir la [PR 39675](#))



IMAGEPULLPOLICY

- **imagePullPolicy** permet de spécifier à k8s sa politique pour aller chercher les images
- Elle vaut par défaut :
 - **Always** si le tag de l'image docker est **latest**
 - **IfNotPresent** si tag pas **latest**

Ex :

- **image: nginx** ⇒ imagePullPolicy par défaut = "Always"
- **image: mon-service-springboot:1.2.1-SNAPSHOT** ⇒ imagePullPolicy par défaut = "IfNotPresent"

→ Il est fortement conseillé d'être explicite et de toujours spécifier la valeur du **imagePullPolicy**



CRÉER UN POD À PARTIR D'UN FICHIER DE DESCRIPTION

Pour créer un Pod de manière *déclarative* (vrai pour les autres types de ressources), utiliser la commande `kubectl apply` avec l'option `--filename` (ou `-f`)

```
$ kubectl apply --filename pod-from-file.yml
pod "k8s-rulez" created

$ kubectl get pod k8s-rulez
NAME      READY   STATUS    RESTARTS   AGE
k8s-rulez  1/1     Running   0          11s
```

MISE À JOUR D'UNE RESSOURCE EXISTANTE



- Il est possible de mettre à jour certains paramètres d'une ressource existante
- Les paramètres qui ne peuvent pas être mis à jour sont indiqués dans la documentation (`kubectl explain pods.spec.containers` par exemple)
- Pour mettre à jour une ressource : `kubectl apply -f <descripteur>.yaml|json`



SUPPRESSION D'UNE RESOURCE

La suppression d'une resource s'effectue avec la commande **kubectl delete**.

- **kubectl delete <type> <name>** : suppression de la resource **<name>** de type **<type>**
- **kubectl delete --filename <descripteur>** : suppression de toutes les resources déclarées dans le descripteur

```
$ kubectl delete pod k8s-rulez
pod "k8s-rulez" deleted

$ kubectl delete -f two-pods-in-one-file.yml
pod "first-of-two" deleted
pod "second-of-two" deleted
```



SUPPRESSION D'UN POD

La suppression d'un Pod va provoquer la suppression de ses conteneurs de la manière suivante :

1. Envoi du signal **SIGTERM (15)**
2. Après une période de grâce (30 secondes par défaut), envoi du signal **SIGKILL (9)** si le conteneur n'est pas encore terminé

Certains processus peuvent avoir besoin de plus de temps pour s'arrêter lors du **SIGTERM**. Ou l'on peut au contraire souhaiter stopper immédiatement les conteneurs.
Il est alors possible de :

- définir **terminationGracePeriodSeconds** au niveau des conteneurs pour définir une durée adaptée
- utiliser l'option **--grace-period** pour surcharger la période lors de la suppression
- utiliser l'option **--now** pour définir la période à **1**, signifiant l'envoi direct de **SIGKILL**



TP 2.1 : Création de Pods avec descripteurs

PLUSIEURS RESSOURCES DANS UN SEUL FICHIER (1/2)



Il est tout à fait possible de décrire plusieurs ressources (de type différent ou pas) dans un unique fichier

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: first-of-two  
spec:  
  containers:  
    - image: nginx:alpine  
...  
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: second-of-two  
spec:  
  containers:  
    - image: redis:alpine  
...
```

PLUSIEURS RESSOURCES DANS UN SEUL FICHIER (2/2)



```
$ kubectl apply -f two-pods-in-one-file.yml
pod "first-of-two" created
pod "second-of-two" created
```

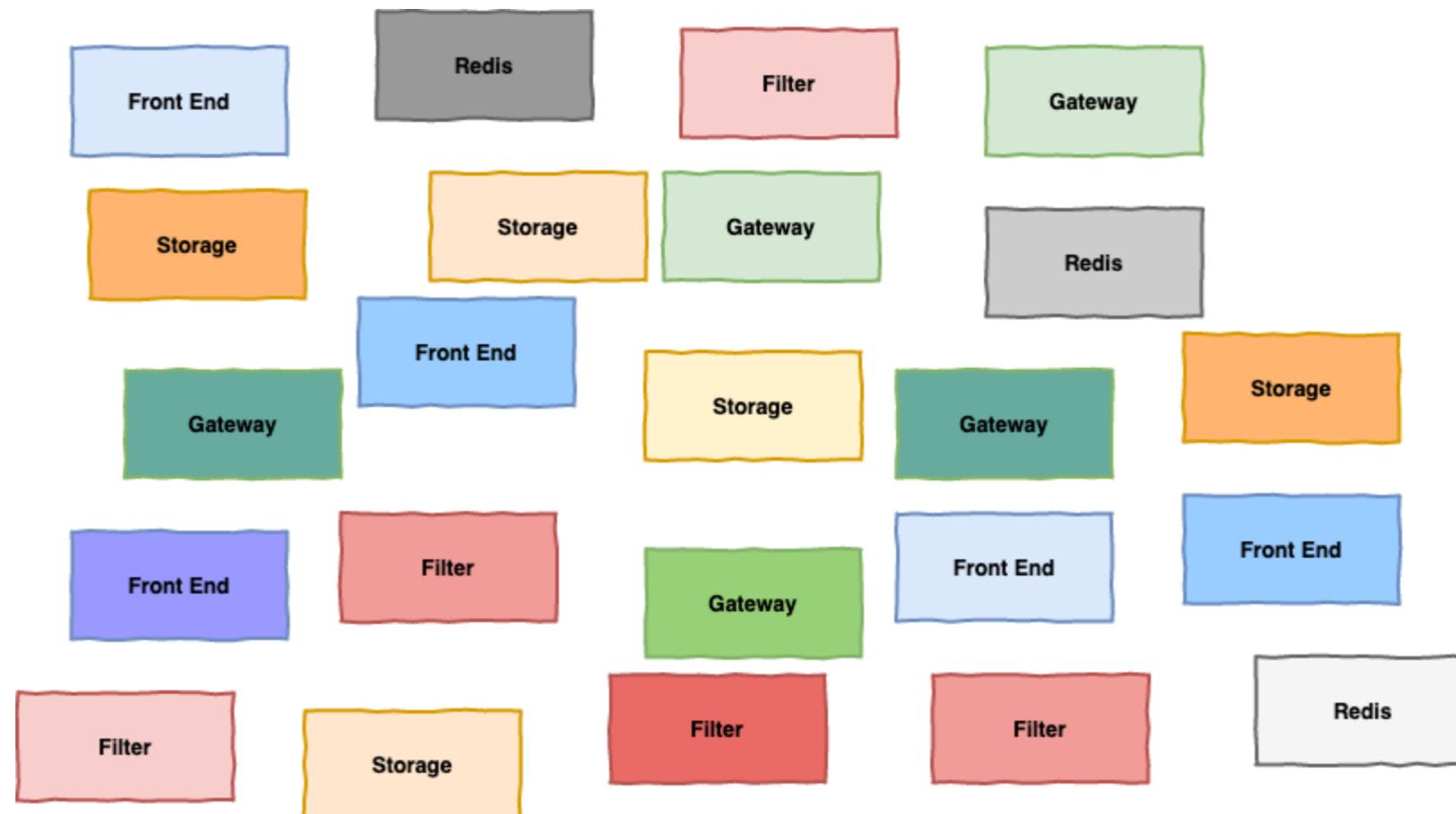
ORGANISATION DES PODS AVEC LES LABELS, LES SÉLECTEURS ET LES NAMESPACES



- Pour l'instant, notre instance Kubernetes ne contient que quelques Pods
- Mais dans la **vraie vie**, le nombre de Pods va se multiplier :
 - plusieurs versions d'un même composant peuvent cohabiter (sur des environnements différents, ou pas)
 - chaque composant peut potentiellement être répliqué
- Comment s'y retrouver (que l'on soit dev ou ops) ?



BESOIN DE RANGER ?



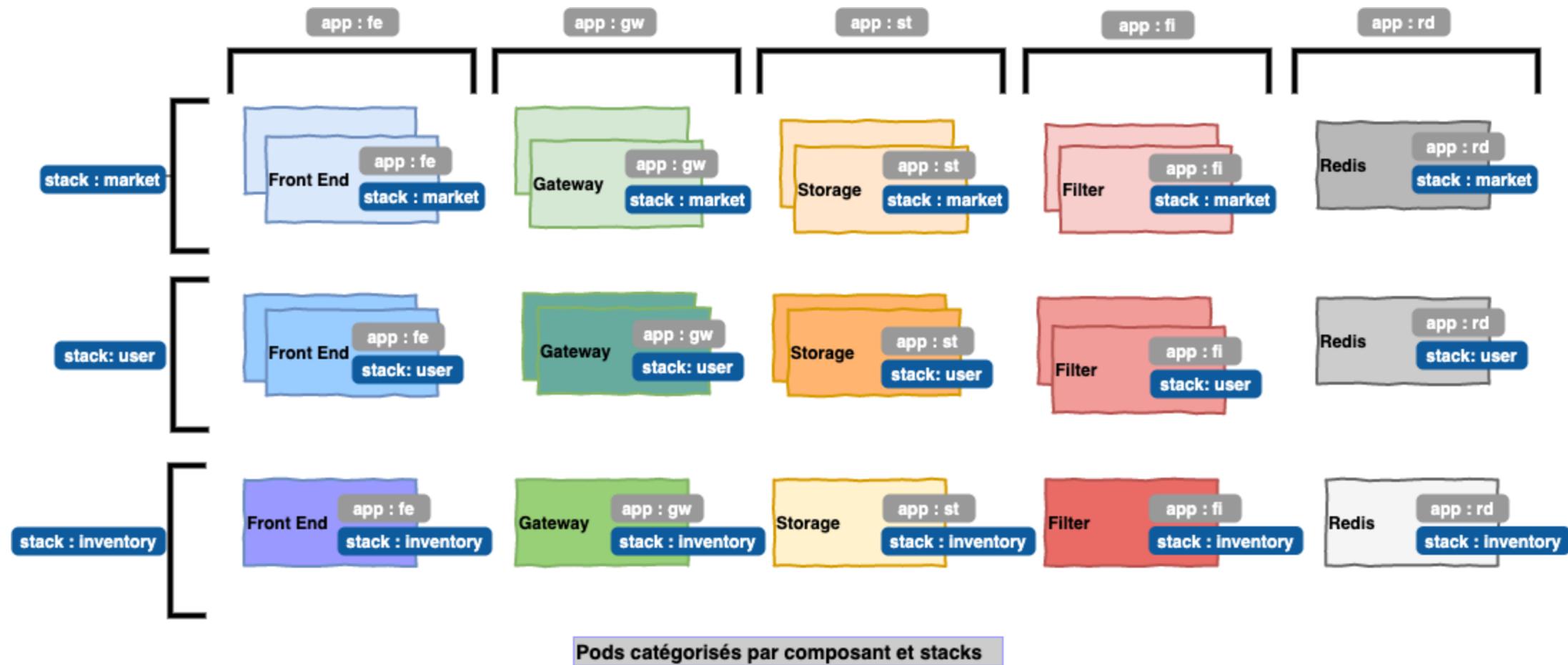
ORGANISER AVEC DES LABELS



- Les *labels* peuvent être utilisés pour organiser et sélectionner des sous-ensembles d'objets
- Le but des *labels* est de pouvoir positionner des attributs sur les objets Kubernetes
- Ces attributs sont utiles pour les utilisateurs de *k8s* mais n'ont pas de valeur pour *k8s* lui-même
- Les *labels* peuvent être attachés aux objets lors de leur création ou être ajoutés/supprimés par la suite
- Chaque objet peut avoir un ensemble de *labels* sous la forme *clé/valeur*
- Chaque clé doit être unique pour un objet donné



EXEMPLE D'ORGANISATION AVEC DES LABELS



SYNTAXE ET CARACTÈRES AUTORISÉS POUR LES LABELS



- Les *labels* sont des couples clé/valeur
- Clé valide : 2 sous-parties, 1 préfixe optionnel et un nom, séparés par un slash (/)
 - Le nom est obligatoire et doit contenir moins de 63 caractères, commencer et finir par un alphanumérique et peut contenir des tirets (-), des underscores (_), des points (.), et d'autres alphanumériques
 - Le préfixe est optionnel
 - S'il est spécifié, le préfixe doit être sous la forme d'un domaine *DNS*: une série de *labels DNS*, séparés par des points et dont la longueur totale ne doit pas dépasser 253 caractères
- Valeur valide : moins de 63 caractères, doit commencer et finir par un alphanumérique et peut contenir des tirets (-), des underscores (_), des points (.), et d'autres alphanumériques



SYNTAXE ET CARACTÈRES AUTORISÉS POUR LES LABELS (EXEMPLES)

Label	Valeur	Validité
my-label	zenika	OK
nodevops.io/my-label	zenika	OK
orchestrator	I_love_Kubernetes	OK
/my-label	whatever	KO
label-with-large-value	invalid-very-very-large-value-label-that-has-more-than-63-characters	KO
value-does-not-start-with-alphanum	_anything	KO



LABELS RECOMMANDÉS

Beaucoup d'outils permettent de visualiser les ressources Kubernetes. Il existe des **labels recommandés** qui permettent d'assurer une sorte d'interopérabilité entre tous ces outils pour décrire d'une manière commune les ressources Kubernetes.

Label	Description	Exemple
app.kubernetes.io/name	Le nom de l'application	mysql
app.kubernetes.io/instance	Un nom unique pour identifier l'instance de l'application	mysql-int
app.kubernetes.io/version	La version de l'application	5.7.21
app.kubernetes.io/component	Le type de composant dans l'architecture	database
app.kubernetes.io/part-of	Le nom de la stack à laquelle l'application appartient	wordpress
app.kubernetes.io/managed-by	L'outil utilisé pour gérer l'application	helm

SPÉCIFIER DES LABELS LORS DE LA CRÉATION



```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: k8s-rulez  
  labels:  
    app.kubernetes.io/component: labs  
    nodevops.io/my-label: zenika-rulez  
    app: shell-gateway  
spec:  
  containers:  
  - name: shelly  
    image: debian:10-slim  
    imagePullPolicy: Always  
    command:  
      - "bash"  
      - "-c"  
      - "sleep infinity"
```

SPÉCIFIER DES LABELS A POSTERIORI



- Utiliser la commande `kubectl label <resourceType> <resourceName> key=value` pour ajouter des labels à une ressource existante :
 - ex : `kubectl label pod k8s-rulez release=stable` pour ajouter le label `release=stable`
- Pour surcharger une clé déjà existante, utiliser l'option `--overwrite`
 - ex : `kubectl label pod k8s-rulez release=unstable --overwrite` pour écraser la précédente valeur du label avec comme clé `release`
- Pour supprimer une clé, utiliser la syntax `key-` :
 - ex : `kubectl label pod k8s-rulez release-`



AFFICHER LES LABELS DES PODS

- Les labels sont visibles par le biais de la commande `kubectl describe`
- Ou en ajoutant l'option `--show-labels` à `kubectl get <resourceType> [<resourceName>]`
- L'option `--label-columns <clé1>[,<clé2>,...]` (ou `-L`) permet d'ajouter une colonne dédiée pour chaque clé spécifiée

```
$ kubectl get pods --label-columns app,run
NAME      READY   STATUS    RESTARTS   AGE     APP           RUN
training-shell 1/1     Running   0          22h
k8s-rulez    1/1     Running   0          1h      shell-gateway
whoami-086vd  1/1     Running   0          21h
whoami-5c05r  1/1     Running   0          1d
whoami-p83md  1/1     Running   1          21h
yaml-pod     2/2     Running   0          6h
```



SÉLECTIONNER LES PODS EN FONCTION DE LEURS LABELS (1/3)

- Lister les Pods pour lesquels **app** vaut exactement **shell-gateway**

```
$ kubectl get pods --selector app=shell-gateway
NAME        READY   STATUS    RESTARTS   AGE
k8s-rulez   1/1     Running   0          2h
```

- Lister les Pods pour lesquels **app** est différent de **shell-gateway**

```
$ kubectl get pods -l 'app!=shell-gateway'
NAME        READY   STATUS    RESTARTS   AGE
training-shell 1/1     Running   0          22h
whoami-086vd   1/1     Running   0          22h
yaml-pod      2/2     Running   0          6h
```



SÉLECTIONNER LES PODS EN FONCTION DE LEURS LABELS (2/3)

- Lister les Pods pour lesquels le label avec comme clé `app` est positionné

```
$ kubectl get pods --selector app
NAME        READY   STATUS    RESTARTS   AGE
k8s-rulez   1/1     Running   0          2h
```

- Lister les Pods pour lesquels le label avec comme clé `run` n'est pas positionné

```
$ kubectl get pods --selector '!run'
NAME        READY   STATUS    RESTARTS   AGE
k8s-rulez   1/1     Running   0          2h
```



SÉLECTIONNER LES PODS EN FONCTION DE LEURS LABELS (3/3)

- Lister les Pods pour lesquels **run** possède les valeurs **whoami** ou **yaml-pod**

```
$ kubectl get pods --selector 'run in (whoami,yaml-pod)'  
NAME        READY   STATUS    RESTARTS   AGE  
whoami-086vd 1/1     Running  0          22h  
whoami-5c05r 1/1     Running  0          1d  
whoami-p83md 1/1     Running  1          22h  
yaml-pod     2/2     Running  0          6h
```

- Lister les Pods pour lesquels **run** ne possède pas les valeurs **whoami** ou **yaml-pod**

```
$ kubectl get pods --selector 'run notin (whoami,yaml-pod)'  
NAME        READY   STATUS    RESTARTS   AGE  
training-shell 1/1     Running  0          23h  
k8s-rulez    1/1     Running  0          2h  
testdnslabel 1/1     Running  0          2h
```



COMBINER LES SÉLECTEURS

- Lister les Pods pour lesquels app vaut **shell-gateway** et app.kubernetes.io/component vaut **labs**

```
$ kubectl get pods --selector app=shell-gateway,app.kubernetes.io/component= labs
NAME      READY   STATUS    RESTARTS   AGE
k8s-rulez  1/1     Running   0          2h
```



POSITIONNER DES LABELS SUR UN NOEUD

- Les labels sont des attributs que l'on peut positionner sur tous les types de ressources Kubernetes
- ... y compris les noeuds
- Pour lister les noeuds, utiliser la commande **kubectl get nodes**

```
$ kubectl get nodes --show-labels
NAME     STATUS   ROLES      AGE    VERSION   LABELS
minikube  Ready    control-plane  2m5s   v1.23.7
                                                     kubernetes.io/arch=amd64,
                                                     kubernetes.io/hostname=minikube,
                                                     kubernetes.io/os=linux,
                                                     node-role.kubernetes.io/control-plane=
```



UTILISATION DES LABELS SUR LES NOEUDS

- Positionner un label sur un noeud permet de le typer
- Par exemple, on peut ajouter un label pour préciser que le noeud utilise des disques SSD, ou qu'il a accès à des ressources GPU

```
$ kubectl get nodes --label-columns ssd,GPU
NAME      STATUS    ROLES      AGE       VERSION   SSD   GPU
minikube  Ready     control-plane  5m50s    v1.23.7  true  false
```



UTILISER LES LABELS POUR CONTRAINdre LE SCHEDULING (1/2)

- Il est possible de contraindre le placement des Pods sur des noeuds possédant des labels particuliers
- Pour cela, utiliser la directive `nodeSelector` dans la section `spec` de la définition du Pod



UTILISER LES LABELS POUR CONTRAINdre LE SCHEDULING (2/2)

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: elastic-on-ssd  
  labels:  
    app: search-engine  
    stack: elastic  
spec:  
  nodeSelector:  
    ssd: "true"  
  containers:  
  - name: elastic  
    image: elasticsearch:5
```



TP 2.2 : labels



ANNOTATIONS (1/2)

- En plus des labels, il est possible d'attacher des *annotations* aux objets Kubernetes
- A la différence des labels, les *annotations* ne sont pas faites pour identifier les objets, i.e. il n'est pas possible d'effectuer des sélections par rapport à celles-ci
- Par contre, les annotations peuvent contenir des informations dont la taille est plus grande que celles des valeurs de label (rappel : 63 caractères) et des caractères interdits dans les valeurs de label



ANNOTATIONS (2/2)

- Les annotations ne sont visibles par le biais de la commande `kubectl get` qu'en utilisant l'option `--output=yaml|json`
- Les annotations sont aussi visibles en passant commande `kubectl describe` (mais leur valeur peut être tronquée si elle est trop grande, ce qui n'est pas le cas en passant par `kubectl get pod <podName> --output=yaml|json`)



AJOUTER UNE ANNOTATION

- Il est possible d'ajouter une annotation à une ressource existante en passant par la commande **kubectl annotate**

```
$ kubectl annotate pod light-sleeper k8s.zenika.com/created-by=cfurmaniak  
pod "light-sleeper" annotated
```

```
$ kubectl describe pod light-sleeper | grep Annotations  
Annotations: k8s.zenika.com/created-by=cfurmaniak
```



EXEMPLES D'INFORMATIONS QUE L'ON PEUT STOCKER DANS LES ANNOTATIONS

- Numéro de build dans l'outil de CI/CD, date et IDs de release, branche git, numéro de Pull | Merge Request, hash des images utilisées, adresse de registries privées
- Liste des dépendances (services)
- Pointeurs vers les outils de centralisation des logs et des métriques, des outils de monitoring ou d'audit
- Source (qui a généré le descripteur par exemple, et quand)
- Équipes responsables de l'application, adresses de contacts (tel, mel, channels slack, ...)

NAMESPACES



- Kubernetes supporte plusieurs clusters virtuels hébergés sur le même cluster physique.
- Ces clusters virtuels sont appelés **Namespaces**
- Les **Namespaces** permettent de séparer les objets en groupes disjoints (ce que ne peut pas assurer l'utilisation des labels)
- Les **Namespaces** fournissent une portée pour les noms des objets
- Les noms des ressources sont uniques au sein d'un même **Namespace**
- Mais des objets portant le même nom peuvent exister dans 2 **Namespaces** distincts
- Les administrateurs du cluster peuvent allouer des quotas de ressources par **Namespace**



LISTER LES NAMESPACES

- Pour lister les Namespaces existants, utiliser `kubectl get namespace` (ou `kubectl get ns`)

```
$ kubectl get namespaces
NAME        STATUS  AGE
default     Active  2d
kube-public Active  2d
kube-system Active  2d
```



SPÉCIFIER UN NAMESPACE

- Quand aucun Namespace n'est spécifié, c'est le Namespace **default** qui est utilisé (il est possible de modifier le Namespace par défaut)
- Pour spécifier le Namespace dans lequel un objet doit être créé ou requêté, spécifier l'option **--namespace=<namespace>** (ou **-n <namespace>**)
- Pour afficher toutes les ressources, tous Namespaces confondus, utiliser l'option **--all-namespaces**

```
$ kubectl get pods --namespace kube-system
NAME                      READY   STATUS    RESTARTS   AGE
coredns-5644d7b6d9-nn5tl   1/1     Running   0          17m
etcd-minikube              1/1     Running   0          16m
kube-addon-manager-minikube 1/1     Running   0          16m
kube-apiserver-minikube    1/1     Running   0          15m
kube-controller-manager-minikube 1/1     Running   0          16m
kube-proxy-64kv8            1/1     Running   0          17m
kube-scheduler-minikube    1/1     Running   0          16m
metrics-server-6754dbc9df-crdh7 1/1     Running   0          17m
```

MODIFIER LE NAMESPACE PAR DÉFAUT



- `kubectl` peut se connecter à plusieurs clusters, le cluster courant est visible par la commande `kubectl config current-context`
- Le Namespace par défaut est visualisable par la commande `kubectl config get-contexts <currentContextName>`
- Pour changer le Namespace par défaut du contexte (cluster) en cours : `kubectl config set-context <currentContextName> --namespace=<insert-namespace-name-here>`
- ... ou passer par `kubectx` et `kubens`

REQUÊTER LES NAMESPACES



- Lister les Namespaces : `kubectl get namespaces`
- Voir les informations détaillées sur un Namespace : `kubectl describe ns <namespace>`
- Attention, les Namespaces ne peuvent pas contenir de point (`.`)
- Créer un Namespace : `kubectl create ns <namespace>`
- Créer un Namespace à partir d'un descripteur : `kubectl apply -f my-ns.yml`

```
---  
apiVersion: v1  
kind: Namespace  
metadata:  
  name: sandbox  
  labels:  
    type: sandbox
```

CRÉER DES OBJETS DANS UN NAMESPACE



- Pour créer un objet à partir d'un descripteur vierge de tout Namespace : `kubectl apply -f fichier.yaml -n <insert-namespace-name-here>`
- Il est possible de spécifier le Namespace cible dans le descripteur directement : `kubectl apply -f descripteur-avec-ns.yaml`

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: k2000  
  namespace: zenika  
  labels:  
    app: knight-rider  
    nickname: kitt  
spec:  
  ...
```



SUPPRESSION D'UN NAMESPACE

- ⚠ Supprimer un *Namespace* supprime tous les objets associés au Namespace...
- ... il n'est pas possible de supprimer les Namespaces **kube-system**, **default** et **kube-public**



TP 2.3 : Annotations et Namespaces

VISUALISATION DES LOGS



- Pour visualiser les logs d'un Pod, utiliser `kubectl logs`
- Pour visualiser les logs du conteneur `whoami` dans le Pod `json-pod` : `kubectl logs json-pod --container whoami`
- Pour visualiser les logs de tous les conteneurs du Pod `json-pod` : `kubectl logs json-pod --all-containers --prefix`
- Pour visualiser les logs de tous les conteneurs de tous les Pods avec le label `app=whoami` : `kubectl logs --selector app=whoami --all-containers --prefix`
- Utiliser `--tail=10` pour ne voir que les 10 dernières lignes
- Utiliser `--follow` (ou `-f`) pour *suivre* les logs
- Utiliser `--since=5s | 2m | 3h` pour ne voir que les logs plus récents que la durée passée en paramètre
- ... ou passer par `stern`

LOGS (CAGIP)



- Les projets doivent sortir les logs sur les sorties standards.
- Les logs sont consultables via la commande `kubectl logs`.
- Ils sont également exportés via des conteneurs Filebeat déployés en DaemonSet avec l'utilisation de l'autodiscover kubernetes.

LOGS (CAGIP)



- La fonctionnalité « hint » est activée ce qui permet au projet de paramétriser du parsing de log directement au niveau de leur manifest de déploiement via des annotations elastic :
<https://www.elastic.co/guide/en/beats/filebeat/master/configuration-autodiscover-hints.html>.
- Ces logs sont envoyés dans un topic Kafka.
- Des logstash récupèrent les logs dans Kafka et les envoient dans une solution elasticsearch/kibana par entité.

```
selector:
  matchLabels:
    app: hello-java-app
  replicas: 1
  template:
    metadata:
      annotations:
        co.elastic.logs/multiline.pattern: '^-[0-9]{4}-[0-9]{2}-[0-9]{2}'
        co.elastic.logs/multiline.negate: "true"
        co.elastic.logs/multiline.match: after
      labels:
        app: hello-java-app
    spec:
      containers:
        - name: hello-java-app
          image: danroscigno/hello-java:v1
```

SECURITYCONTEXT



Pour lancer un conteneur avec un utilisateur et/ou groupe différent, il est possible d'utiliser le **securityContext**.

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  containers:
    - name: sec-ctx-demo
      image: busybox
      command: [ "id" ]
    - name: root
      image: busybox
      command: [ "id" ]
      securityContext:
        runAsUser: 0
```

UTILISATEUR



Le **securityContext** permet de définir l'utilisateur du processus lancé :

- dans tous les conteneurs du **Pod**, lorsqu'il est positionné dans le **Pod**
- d'un conteneur particulier, lorsqu'il est placé au niveau du conteneur



EXEMPLE

```
$ kubectl apply -f security-context-demo.yaml  
pod/security-context-demo created
```

```
$ kubectl logs security-context-demo -c sec-ctx-demo  
uid=1000 gid=3000 groups=2000
```

```
$ kubectl logs security-context-demo -c root  
uid=0(root) gid=3000 groups=2000
```



INIT CONTAINERS

- Un Pod peut être constitué de plusieurs conteneurs hébergeant des applications
- Mais il peut aussi s'appuyer sur un ou plusieurs **Init Containers**
- Les **Init Containers** sont comme les **containers** sauf que :
 - ils ont une durée de vie limitée
 - chacun des **Init Containers** n'est lancé que si le précédent s'est terminé sans erreur
- Si un **Init Container** est en échec, k8s redémarre le Pod jusqu'à ce que l'**Init Container** se termine avec le status *succès* (sauf si la **restartPolicy** du Pod vaut **Never**)

QUAND UTILISER UN INIT CONTAINER ?



- Les Init Containers partagent les mêmes volumes que les Containers, on peut donc s'en servir pour générer la configuration de l'application principale en **one-shot** (avec confd par exemple)
- On peut utiliser un Init Container pour réaliser des opérations nécessitant des privilèges et abaisser le niveau de privilège requis dans le conteneur applicatif
- On peut aussi mettre dans un Init Container toute la logique de migration d'une base de données
- Ils sont lancés avant les autres conteneurs, on peut donc s'en servir pour attendre qu'une dépendance soit prête ou que des préconditions soient remplies



EXEMPLE D'INIT CONTAINER

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: myapp-pod  
  labels:  
    app: myapp  
spec:  
  initContainers:  
    - name: init-myservice  
      image: busybox  
      command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2;  
done;']  
    - name: init-mydb  
      image: busybox  
      command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']  
  containers:  
    - name: myapp-container  
      image: debian:10-slim  
      command: ['bash', '-c', 'echo The app is running! && sleep 3600']
```



TP 2.4 : Logs, TP 2.5 Init container



CYCLE DE VIE DES PODS

- Les Pods sont mortels (i.e., ils peuvent être supprimés)
- Ils sont créés, mais ne sont pas recréés automatiquement
- (C'est le travail des *ReplicaSets* que l'on verra par la suite)
- Par contre, si un des conteneurs gérés par le Pod *meurt* il est de la responsabilité du Pod de le redémarrer (en fonction de la *restartPolicy* sélectionnée pour le Pod)

LIVENESS



- Mais comment détecter que l'application ne fonctionne pas quand le conteneur associé ne *meurt* pas ?
 - Une application Java avec un problème mémoire peut envoyer des **OutOfMemoryErrors**, mais la JVM n'est pas terminée pour autant !
- Comment faire pour indiquer à Kubernetes que l'application ne fonctionne plus correctement ?
- Vous pourriez *intercepter* ce type d'erreurs et terminer l'application, mais ce serait fastidieux de gérer tous les cas
- ... et comment faire si votre application se trouve dans une *boucle infinie* ou une situation de *deadlock* ?

 Kubernetes propose la notion de *Liveness Probe* ("sonde de vie") pour répondre à cette problématique



CONTAINER PROBES (1/2)

- Une *sonde* (probe) est un diagnostic effectué par Kubernetes (par un composant appelé *kubelet* que nous verrons plus tard) sur un conteneur
- Il existe 3 types de sondes :
 - *httpGet*: un appel *GET* est effectué, un *status code* ≥ 200 et < 400 est considéré comme un *succès*
 - *tcpSocket*: si le port concerné est ouvert, la sonde est en *succès*
 - *exec*: une commande est effectuée dans le conteneur concerné, un code de retour à *0* équivaut à un *succès*



CONTAINER PROBES (2/2)

- Les sondes sont utilisées pour vérifier qu'un conteneur est *démarré* (*Startup Probe*), *fonctionne* (*Liveness Probe*) ou est *prêt* à recevoir du trafic (*Readiness Probe*)
 - Nous aborderons les *Readiness Probe* dans le chapitre suivant, *Services*
- Une seule sonde de chaque type (httpGet, tcpSocket, exec) peut être définie pour chaque catégorie (Startup/Liveness/Readiness) *par conteneur*



CONFIGURATION D'UNE SONDE

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  labels:  
    test: liveness  
  name: liveness-http  
spec:  
  containers:  
    - name: liveness  
      image: k8s.gcr.io/e2e-test-images/agnhost:2.21  
      args:  
        - liveness  
    livenessProbe:  
      httpGet:  
        path: /healthz  
        port: 8080  
      initialDelaySeconds: 15  
      timeoutSeconds: 1
```

PARAMÈTRES ADDITIONNELS D'UNE SONDE



- Parce que l'application dans le conteneur peut ne pas être utilisable de suite, **initialDelaySeconds** permet de régler le délai avant de commencer à lancer les premiers diagnostics (défaut: 0, minimum: 0)
- **timeoutSeconds** permet de spécifier la durée max autorisée pour recevoir la réponse. Si la réponse n'est pas récupérée dans le temps imparti, la sonde est en échec (défaut: 1, minimum: 1)
- **periodSeconds** permet de spécifier la période entre les exécutions du diagnostic (défaut: 10, minimum: 1)
- **successThreshold** permet de spécifier le nombre minimum de succès pour que la sonde soit considérée en état **succès** (défaut: 1, minimum: 1, doit être 1 pour *Startup Probe* et *Liveness Probe*)
- **failureThreshold** permet de spécifier le nombre minimum d'échecs pour que la sonde soit considérée en état **échec** (défaut: 3, minimum: 1)

DÉLAI DE DÉMARRAGE DE CONTENEUR



- Il peut arriver qu'un conteneur mette du temps à démarrer (si par exemple il y a très peu de CPU pour le Pod) et il faudrait alors mettre un long `initialDelaySeconds`
- Mais cela retarde d'autant le temps de détection d'un conteneur fonctionnel
- Depuis la version 1.16, il existe une *Startup Probe* pour détecter que le conteneur n'est pas encore prêt

```
startupProbe:  
  httpGet:  
    path: /healthz  
    port: 8080  
  failureThreshold: 30  
  periodSeconds: 10  
  initialDelaySeconds: 15
```

- Cela va permettre d'attendre jusqu'à `initialDelaySeconds + (failureThreshold * periodSeconds)` que le conteneur démarre et passer à la *Liveness Probe* pour ensuite vérifier son état



VISUALISATION DE L'ÉTAT DES SONDES (1/2)

La colonne *RESTARTS* dans la sortie de la commande `kubectl get pod` indique le nombre de restarts du/des conteneurs du Pod (mais ces restarts ne sont pas forcément dus à une sonde en échec)

```
$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
rp-check   1/1     Running   2          1m
```

VISUALISATION DE L'ÉTAT DES SONDES (2/2)



La commande `kubectl describe pod` retourne des informations sur les potentielles sondes en échec (et les restarts associés)

Events:					
Type	Reason	Age	From	Message	
---	-----	-----	-----	-----	
Warning	Unhealthy	56s (x33 over 34m)	kubelet, minikube	Liveness probe failed: HTTP probe failed with statuscode: 500	
Normal	Killing	56s (x16 over 33m)	kubelet, minikube	Killing container with id docker://liveness:pod "liveness-http_default(b87d60ab-ad9d-11e7-b4c7-0800279f22af)" container "liveness" is unhealthy, it will be killed and re-created.	
Warning	BackOff	7s (x128 over 31m)	kubelet, minikube	Back-off restarting failed container	
...					



TP 2.6 : Cycle de vie, TP 2.7 Liveness Probes

ANATOMIE D'UNE BONNE LIVENESS PROBE



- Il est fortement recommandé de positionner des *Liveness Probes* pour vos Pods en production car si ce n'est pas le cas, *k8s* n'a aucun moyen de savoir si vos applications fonctionnent comme attendu
- Une sonde (http) devrait correspondre à un chemin dédié dans votre application et correspondre à un diagnostic interne de fonctionnement
- Une sonde ne doit pas être affectée par les dépendances de votre application :
 - ce sont vos dépendances qui sont en erreur, pas votre application !
 - à vous de bien gérer les dépendances en erreur dans votre application (utiliser des *circuit breaker* par exemple)
- Gardez vos sondes *légères* : elles doivent répondre rapidement et sans consommer trop de *cpu* et de *mémoire* pour ne pas perturber le fonctionnement de votre application





CONTROLLERS

AGENDA DE CE CHAPITRE 'CONTROLLERS'



- ReplicaSets
- DaemonSets
- Jobs

INTRODUCTION AUX CONTROLLERS (1/2)



Récapitulons :

- *k8s* s'assure que nos conteneurs fonctionnent et les redémarre si le process principal *meurt* ou si la *liveness probe* échoue
- Cette tâche incombe au composant *kubelet* qui tourne sur chaque noeud worker
- Le *centre de contrôle* de *Kubernetes* qui tourne sur des noeuds dédiés ne joue aucun rôle dans cette partie

INTRODUCTION AUX CONTROLLERS (2/2)



- Que se passe-t-il si le noeud qui héberge le Pod venait à tomber ?
- Pour s'assurer que notre application/Pod puisse être redémarré sur un autre noeud il faut passer par un mécanisme de plus haut niveau que le *Pod*
- Ces mécanismes, appelés *Controllers*, sont les *ReplicaSets*, *DaemonSets*, *Jobs* et *CronJobs*

REPLICASET



- Un *ReplicaSet* est une ressource Kubernetes dont le rôle est de s'assurer qu'un Pod est lancé et fonctionne correctement
- Si un Pod disparaît/meurt pour quelque raison que ce soit :
 - un noeud qui tombe
 - le Pod a été retiré d'un noeud lors d'une maintenance
- ... le *ReplicaSet* va détecter l'absence du Pod et faire en sorte qu'un nouveau soit créé



REPLICAS (1/2)

- un *ReplicaSet* peut gérer plusieurs copies (appelées *replicas*) d'un même Pod et s'assurer que le nombre de replicas en fonctionnement correspond à celui attendu :
 - s'il n'y pas assez de replicas par rapport à la cible, le *rs* va créer ceux qui manquent
 - s'il y en a trop, il va en supprimer pour avoir le nombre demandé



REPLICAS (2/2)

- un *ReplicaSet* s'appuie sur un *selecteur* de labels pour identifier les Pods qu'il doit gérer
- un *ReplicaSet* est similaire à un superviseur de process, mais au lieu de superviser des process sur un seul noeud, il peut superviser plusieurs Pods sur plusieurs noeuds
- il est fortement recommandé d'utiliser un *ReplicaSet* même si vous ne devez gérer qu'un Pod

EXEMPLE DE DÉFINITION D'UN REPLICASET



```
---  
apiVersion: apps/v1  
kind: ReplicaSet  
metadata:  
  name: rs-nginx  
spec:  
  replicas: 3  
  selector:  
    matchLabels:  
      app: nginx  
  template:  
    metadata:  
      name: pod-nginx  
      labels:  
        app: nginx  
    spec:  
      containers:  
        - name: container-nginx  
          image: nginx  
          ports:  
            - containerPort: 80
```

CONTENU D'UN DESCRIPTEUR DE REPLICASET



- les sections `apiVersion`, `kind` et `metadata` sont semblables à ce qu'on trouve pour toutes les ressources k8s
- la section `specs` contient :
 - le nombre de `replicas` attendu
 - le `selecteur` utilisé pour `identifier` les Pods associés
 - le `modèle` à utiliser pour la création des Pods



POD SELECTOR ET REPLICASET (1/2)

- Le champ `.spec.selector` est un sélecteur de label
- Le `ReplicaSet` va gérer tous les Pods qui correspondent à ce sélecteur
- ⚠️ Le `ReplicaSet` ne fait pas la différence entre les Pods qu'il a créés/supprimés et ceux qu'un autre process a créés/supprimés (s'il ne sont pas dans le giron d'un ReplicaSet [voir TP])



POD SELECTOR ET REPLICASET (2/2)

- Le champ `.spec.selector` doit être obligatoirement spécifié, il doit correspondre aux labels du template de Pod (`.spec.template.metadata.labels`)
- Si le champ `.spec.selector` est spécifié et ne correspond pas aux labels du template de Pod (`.spec.template.metadata.labels`), la création sera refusée

```
$ kubectl apply -f Exercises/workspaces/Lab3/rs--nginx--cannot--create--3.1.yml
The ReplicaSet "frontend" is invalid: spec.template.metadata.labels:
Invalid value: map[string]string{"app":"frontend", "level":"advanced"}: `selector` does not match template `labels`
```



SELECTOR DE REPLICASET

- Le selector de **ReplicaSet** accepte 2 types de sélection :
 - **matchLabels** : simple sélection **clé=valeur**
 - **matchExpressions** : des selections avancées :
 - **level** avec comme valeur **novice** ou **intermediate**
 - clé **level** définie, quelque soit la valeur

EXEMPLE DE SELECTOR D'UN REPLICASET



```
---  
apiVersion: apps/v1  
kind: ReplicaSet  
metadata:  
  name: frontend  
  labels:  
    app: frontend  
spec:  
  replicas: 3  
  selector:  
    matchLabels:  
      app: frontend  
    matchExpressions:  
    - key: level  
      operator: In  
      values:  
      - novice  
      - intermediate  
  template:  
  ...
```

OPERATORS, MATCHLABELS ET MATCHEXPRESSIONS



- Il y a 4 opérateurs disponibles :
 - **In** (**values** doit aussi être spécifié)
 - **NotIn** (**values** doit aussi être spécifié)
 - **Exists** (**values** ne doit pas être spécifié)
 - **DoesNotExist** (**values** ne doit pas être spécifié)
- Si **matchLabels** et **matchExpressions** sont spécifiés tous les deux, les labels doivent correspondre aux 2 contraintes



MODIFICATIONS DES LABELS DES PODS OU DU TEMPLATE

- Si vous modifiez les labels d'un Pod piloté par un **RS** il sortira du giron du **RS**
- Le **ReplicaSet** ne contrôle pas le contenu d'un Pod, si vous modifiez le modèle des Pods d'un **RS**, seuls les nouveaux Pods créés seront affectés
 - Il existe cependant un moyen d'obliger les Pods à se mettre à jour, nous verrons cela dans le chapitre **Stratégies de déploiement**



KUBECTL ET LES REPLICASETS

- `kubectl get replicsets` ou (plus court !) `kubectl get rs`
- `kubectl describe rs <rsName>` pour accéder aux infos détaillées
- `kubectl scale rs <rsName> --replicas=N` pour mettre un **RS** à l'échelle de manière impérative
- `kubectl delete rs <rsName>` pour supprimer un **RS** : attention, cela va aussi supprimer les Pods associés
- si vous souhaitez supprimer le **RS** sans supprimer les Pods associés, utilisez `kubectl delete rs <rsName> --cascade=orphan`

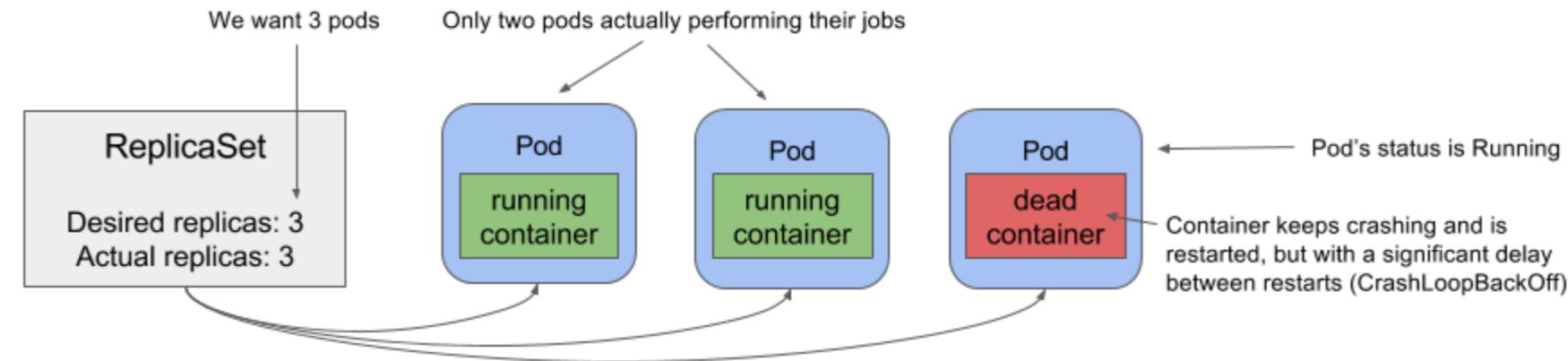


TP 3.1 : ReplicaSets

REPLICASET ET PODS QUI CRASHENT EN BOUCLE



- Attention, le ReplicaSet est responsable d'avoir un nombre de Pods up
- Mais si un Pod crash toutes les 5 minutes, ce n'est pas son problème !



DAEMONSETS



- Les **ReplicaSets** sont utilisés pour faire tourner un nombre défini de Pod n'importe où sur le cluster
- Un autre cas d'utilisation existe : faire tourner un et un seul exemplaire d'un Pod sur chaque noeud (ou sur un sous-ensemble défini de noeuds)
- Pour répondre à cette problématique, utilisons un **DaemonSet**

CAS D'USAGE DES DAEMONSETS



- Faire tourner un daemon de stockage sur chaque noeud (glusterd, ceph, rook, ...)
- Faire tourner un agent de collecte des logs (fluentd, logstash, ...)
- Faire tourner un agent de collecte de métriques (Prometheus, collectd, datadog, ganglia, ...)

DIFFÉRENCE REPLICASET VS DAEMONSET



- Un **ReplicaSet** s'assure qu'un nombre désiré de replicas d'un Pod tourne sur le cluster :
 - Si un noeud est ajouté au cluster, rien ne se passe
 - Si un noeud est retiré du cluster, les Pods éventuellement schéduités sur ce noeud seront réaffectés sur d'autres noeuds
- Un **DaemonSet** s'assure qu'un et un seul exemplaire de Pod tourne sur chaque noeud (ou sous-ensemble de noeud)
 - Si un noeud est ajouté au cluster, un nouveau Pod sera lancé sur le nouveau noeud
 - un **DaemonSet** sera déployé même sur les noeuds marqués comme non-utilisables (par les administrateurs)
 - Si un noeud est retiré du cluster, aucun nouveau Pod ne sera lancé



DESCRIPTEUR DE DAEMONSET

```
---  
apiVersion: apps/v1  
kind: DaemonSet  
metadata:  
  name: filebeat  
  namespace: kube-system  
spec:  
  selector:  
    matchLabels:  
      k8s-app: filebeat  
  template:  
    metadata:  
      labels:  
        k8s-app: filebeat  
  spec:  
    containers:  
    - name: filebeat  
      image: docker.elastic.co/beats/filebeat:7.11.12
```



OBJECTIF DES JOBS

- Nous n'avons pour l'instant vu que des Pods qui doivent tourner en continu
- Mais on doit parfois lancer une tâche qui se termine quand le travail qu'elle représente est achevé
- Les **ReplicaSets** et **DaemonSets** font tourner des tâches qui ne sont jamais considérées comme étant achevées
- Les process dans les Pods associés aux **RS** et **DS** sont redémarrés quand ils se terminent
- Dans le cas des tâches qui peuvent s'achever légitimement, il ne faut pas les redémarrer

Les **Jobs** Kubernetes correspondent à ce genre de tâches

JOBS



- Un cas simple consiste à faire tourner un **Job** pour lancer un Pod qui s'achèvera avec succès quand le process associé sera terminé avec un code de retour en *succès*
- Le Job pourra cependant aussi redémarrer le Pod si celui-ci n'a pu terminer sa tâche associée correctement
- Un Job peut aussi lancer plusieurs Pods en parallèle



DESCRIPTEUR DE JOBS

```
---  
apiVersion: batch/v1  
kind: Job  
metadata:  
  name: batch-job  
spec:  
  template:  
    metadata:  
      labels:  
        app: batch-job  
    spec:  
      restartPolicy: OnFailure  
      containers:  
        - name: main  
          image: zenika/sample-batch
```

👉 Noter le **restartPolicy: OnFailure** ↔ le Pod n'est pas redémarré quand le conteneur se termine normalement, il faut le préciser même pour un Job

PARAMÈTRES COMPLÉMENTAIRES DES JOBS



- Pour spécifier qu'un **Job** doit être exécuté plusieurs fois, utiliser le paramètre **.spec.completions**
- Pour spécifier que plusieurs **Pods** d'un même **Job** peuvent s'exécuter en parallèle, utiliser le paramètre **.spec.parallelism**
- Pour spécifier qu'un **Job** ne doit pas durer plus qu'un temps défini, utiliser **.spec.activeDeadlineSeconds**
- Pour spécifier qu'un **Job** ne doit pas être redémarré plus qu'un certain nombre de fois, utiliser **.spec.backoffLimit**



JOBS ET RESTARTPOLICY

- L'utilisation d'un **Job** n'est appropriée que si **RestartPolicy** vaut **OnFailure** ou **Never**
- Rappel : si aucune valeur n'est spécifiée, la valeur par défaut pour un Pod est **Always**
- Il est donc obligatoire de spécifier la valeur de **RestartPolicy** dans le cas d'un **Job** !

CRONJOBS



- Il est aussi possible de lancer des **Jobs** à fréquence fixe ou à une date dans le futur
- C'est la ressource de type **CronJob**



DESCRIPTEUR DE CRONJOBS

```
---  
apiVersion: batch/v1  
kind: CronJob  
metadata:  
  name: hello  
spec:  
  schedule: "*/1 * * * *"  
  jobTemplate:  
    spec:  
      template:  
        spec:  
          containers:  
            - name: hello  
              image: debian:10-slim  
              args:  
                - bash  
                - -c  
                - date; echo Hello from the Kubernetes cluster  
  restartPolicy: OnFailure
```

FUSEAU HORAIRE ET CRONJOBS



- Avec la version actuelle de la ressource CronJob, on ne peut pas préciser le fuseau horaire pour l'expression `cron` présente dans le champ `schedule`.
- Le fuseau horaire pris en compte par Kubernetes est celui du composant `controller manager`
- C'est donc soit le fuseau horaire du serveur où est démarré le composant, soit le fuseau horaire du conteneur où tourne le composant.

FIN DES JOBS ET NETTOYAGE



- Quand un **Job** est terminé, il n'est pas supprimé automatiquement, de même pour les **Pods** associés
- Ceci afin de pouvoir accéder aux logs des Pods concernés
- C'est donc à l'utilisateur de supprimer manuellement les **Jobs**
- Il est aussi possible de spécifier la taille maximum des historiques :
 • **.spec.successfulJobsHistoryLimit** et **.spec.failedJobsHistoryLimit**

POINTS D'ATTENTIONS JOBS/CRONJOBS (1/2)



- Une ressource de type **Job** sera créée à partir d'une ressource **CronJob** approximativement à la date schédule
- Puis le **Job** crée un **Pod**
- Il est possible que le **Job** et le **Pod** soient créés en retard par rapport à la cible
- Si la date cible est un critère fort pour vous, utilisez `.spec.startingDeadlineSeconds` (si la deadline est dépassée, le **Job** ne sera pas lancé)
- Dans des circonstances normales, le **CronJob** ne va créer qu'un seul **Job** pour une date d'exécution
- Mais il arrive que 2 **Jobs** soient créés, ou aucun

POINTS D'ATTENTIONS JOBS/CRONJOBS (2/2)



- Il faut rendre vos Jobs **idempotents** :
 - pour prendre en compte le cas où un noeud tombe pendant que le Job a commencé, car celui-ci sera alors relancé sur un autre noeud
 - pour prendre en compte les CronJobs qui se lancent 2 fois (c'est un cas qui peut arriver)
 - pour prendre en compte les CronJobs qui ne se lancent pas du tout, le Job doit potentiellement pouvoir "rattraper" la non exécution du job précédent



TP 3.2 : Jobs et CronJobs





SERVICES

AGENDA DE CE CHAPITRE 'SERVICES'



- À quoi répond le concept des Services
- Exposer en interne du cluster
- Exposer un service externe
- Exposer vers l'extérieur : Ingress vs LoadBalancer
- Sonde Readiness
- Services headless
- Port forward
- Sécuriser les Services au sein du cluster

À QUOI RÉPOND LE CONCEPT DES SERVICES



- Les **Pods** sont éphémères (ils peuvent être supprimés ou déplacés)
- Kubernetes assigne une adresse IP au **Pod** juste avant sa création, il n'est donc pas possible de la connaître à l'avance
- La mise à l'échelle d'un **ReplicaSet** implique que plusieurs **Pods** hébergent la même application et fournissent donc le même service
- Les clients de ces applications ne devraient pas avoir à connaître les IPs des différents **Pods** qu'ils consomment
- Pour résoudre ces problématiques, Kubernetes fournit une ressource appelée **Service** que nous allons explorer dans ce chapitre

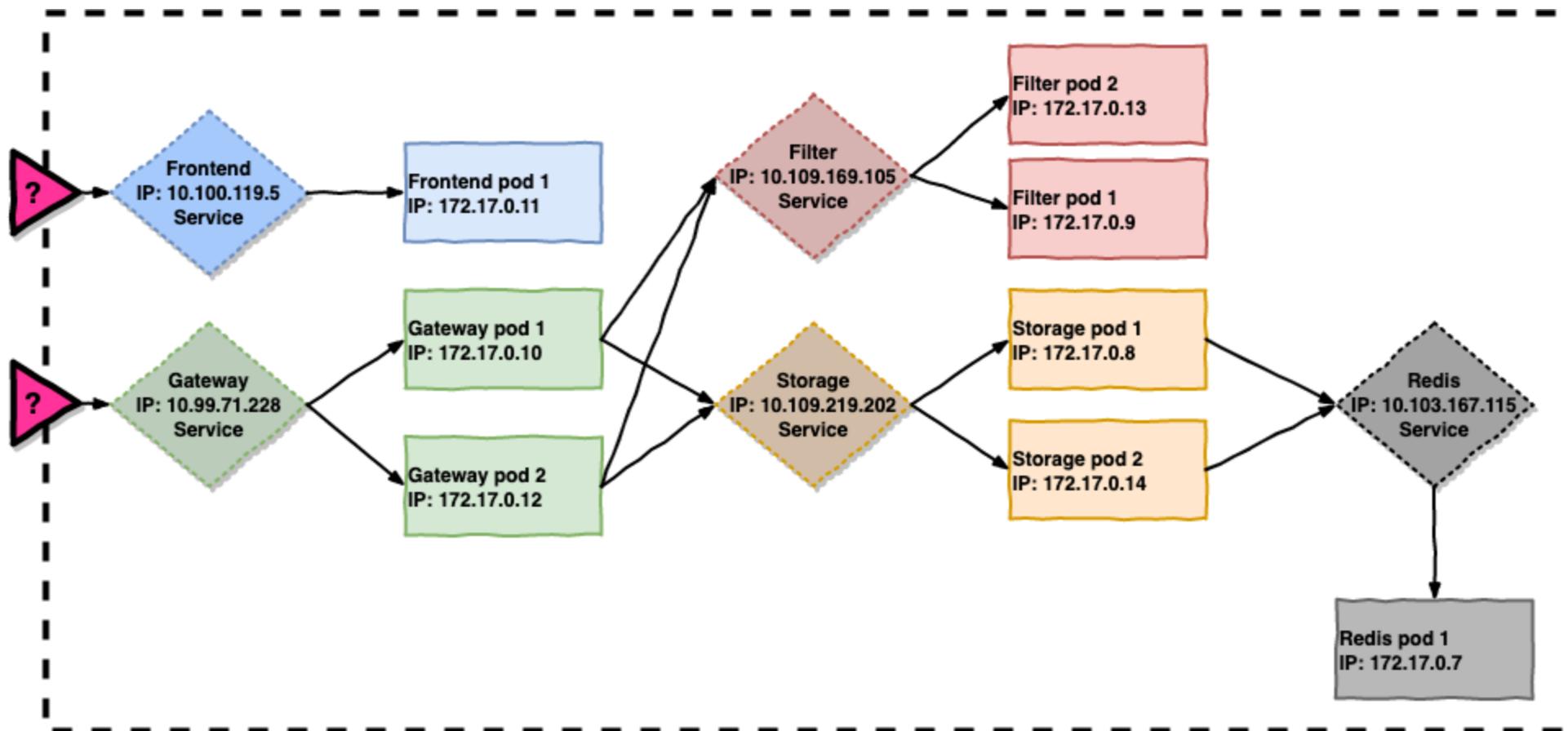
INTRODUCTION AUX SERVICES



- Un *Service* Kubernetes est une ressource que vous créez pour fournir un unique et constant point d'entrée pour un groupe de *Pods* qui hébergent la même application
- Chaque *Service* a une adresse IP (et un port associé) qui ne change pas tant que le *Service* existe
- Les clients peuvent ouvrir des connexions vers ce couple *IP:port* et ces connexions seront redirigées vers les *Pods* qui composent l'application
- De cette façon, les clients n'ont pas besoin de connaître les IPs des *Pods* qui composent l'application, ce qui permet à ces *Pods* de pouvoir être déplacés/supprimés sans impacter les clients



EXEMPLE D'UTILISATION DES SERVICES





DESCRIPTEUR DE SERVICE

```
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: whoami  
spec:  
  selector:  
    app: whoami  
    stack: market  
  ports:  
    - protocol: TCP  
      port: 8080  
      targetPort: 80
```

- Ce descripteur crée un Service qui écoute sur le port **8080** et va rediriger le trafic vers tous les Pods qui correspondent à **app=whoami, stack=market** sur le port **80**.
- Le Service se verra assigner une IP interne appelée **ClusterIP**



EXPOSER EN INTERNE DU CLUSTER

- Par défaut, si la valeur `.spec.type` d'un Service n'est pas spécifiée, le type sera `ClusterIP`
- Choisir cette valeur vous permet de vous assurer que le Service ne sera pas accessible depuis l'extérieur du cluster Kubernetes

⚠ Il est impossible de "pinger" un Service car :

- ping utilise le protocole `ICMP` qui n'est pas supporté par les Services Kubernetes (seul les protocoles `TCP`, `UDP` et `SCTP` sont supportés)
- il n'est pas possible de préciser un port de destination avec la commande `ping`, or le Service n'est accessible que sur les ports déclarés

En revanche, en utilisant l'outil `httping`, il est possible d'avoir un comportement similaire : il effectue une requête HTTP avec le verbe `HEAD` sur le nom d'hôte + port donné

EXPOSER PLUSIEURS PORTS POUR LE MÊME SERVICE



- Les Services peuvent exposer plusieurs ports (qui doivent alors être nommés)
- Par exemple, une application peut exposer le port principal et le port d'administration
- Lorsque les ports sont nommés, on pourra faire référence au nom plutôt qu'à sa valeur

```
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: simple-springboot  
spec:  
  selector:  
    app: simple-springboot  
  ports:  
    - name: main  
      port: 80  
      targetPort: 8080  
    - name: admin  
      port: 8000  
      targetPort: 8081
```



SERVICE DISCOVERY

- Devoir utiliser la **ClusterIP** pour communiquer avec le Service n'est pas pratique
- Utiliser l'adresse DNS du Service :
 - chaque Service est disponible par le **FQDN** : **<nom-du-service>. <namespace>.svc.cluster.local**
 - au sein d'un même namespace, chaque Service est même directement adressable simplement par son nom : **<nom-du-service>**



TP 4.1 : Services et Service Discovery

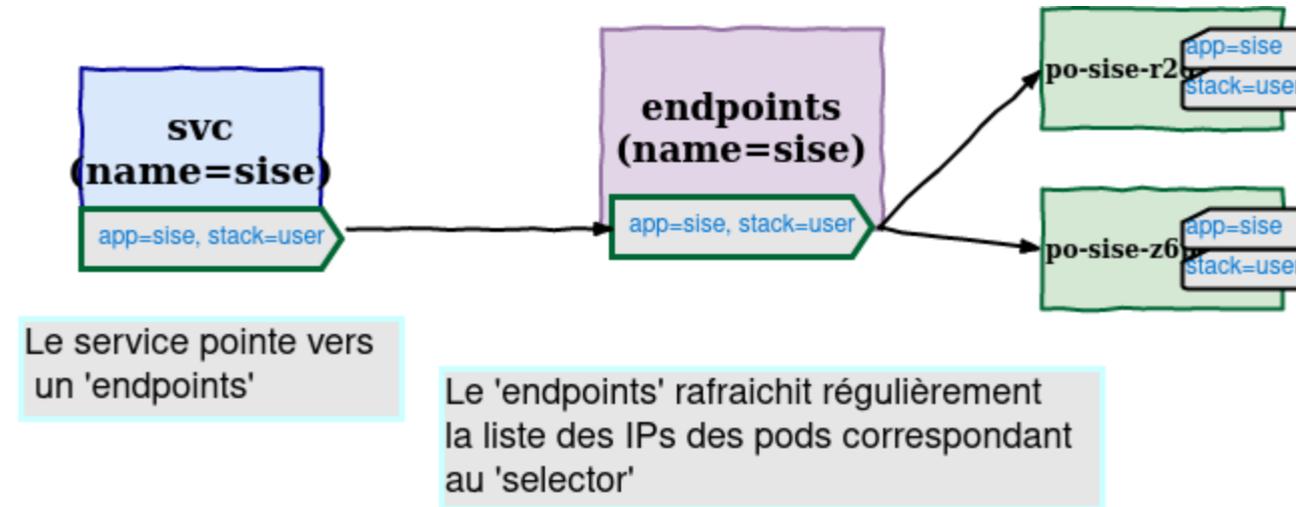
DIS, COMMENT ÇA MARCHE UN SERVICE EN FAIT ?



```
$ kubectl describe service whoami
Name:           whoami
Namespace:      default
Labels:         app=whoami
                stack=market
Selector:       app=whoami,stack=market
Type:           ClusterIP
IP:             10.0.0.72
Port:           <unset>  8080/TCP
TargetPort:     80/TCP
Endpoints:      172.17.0.8:80,172.17.0.9:80
Session Affinity: None
```



ENDPOINTS



```
$ kubectl get endpoints whoami
```

NAME	ENDPOINTS	AGE
whoami	172.17.0.8:80,172.17.0.9:80	3m

EXPOSER UN SERVICE EXTERNE



- Il est possible de référencer un service externe (i.e. pas hébergé par k8s) comme un Service de k8s
- i.e : le Service sera défini dans le cluster mais pointera vers un/des IPs externe/s au cluster
- Mode opératoire :
 - Faire pointer vos applications (hébergées dans k8s) vers le Service
 - Si la/les IP/s de votre service externe change(nt), mettre à jour uniquement la définition du service externe
 - Vos applications ne sont pas impactées



SERVICE EXTERNE ET ENDPOINTS (1/2)

- Pour cela :
 1. Créer un Service *sans* sélecteur
 2. Créer la ressource **Endpoints** manuellement (avec le même nom que le Service)
- Si par la suite vous voulez héberger le service externe dans k8s, il suffit d'ajouter un sélecteur sur le Service (et la liste des IP/Ports sera managée par k8s)
- À l'inverse, vous pouvez supprimer un sélecteur d'un Service et mettre à jour manuellement la liste des IP/port



SERVICE EXTERNE ET ENDPOINTS (2/2)

Service externe avec endpoints :

```
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: external-with-port  
spec:  
  ports:  
    - port: 80  
      targetPort: 8080
```

```
---  
apiVersion: v1  
kind: Endpoints  
metadata:  
  name: external-with-port  
subsets:  
  - addresses:  
    - ip: 1.2.3.4  
    - ip: 5.6.7.8  
  ports:  
    - port: 8080
```

EXPOSER UN SERVICE À DES CLIENTS EXTERNES



- Pour l'instant nous n'avons parlé que de consommation de Service par des Pods du cluster lui-même
- Il est bien sûr possible d'exposer les Services pour qu'ils soient accessibles par des clients externes au cluster



COMMENT FAIRE ?

- Passer par le mécanisme des **HostPort** (⚠ ce n'est pas un **type** de Service)
- Utiliser un Service de type **NodePort**
- Utiliser un Service de type **LoadBalancer**
- Créer une ressource de type **Ingress**

MÉCANISME HOSTPORT



```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: nginx-squat  
spec:  
  containers:  
    - name: port-squatter  
      image: nginx:alpine  
      ports:  
        - name: exposed  
          hostPort: 80  
          containerPort: 80
```

- k8s va réserver le **.spec.containers.ports.hostPort** sur le noeud du cluster où va tourner le Pod pour le Service
- Il faut donc savoir sur quel noeud tourne le Pod pour pouvoir y accéder
- Réservez l'utilisation des **hostPorts** à des cas particuliers (voir **Ingress** plus tard)



SERVICE DE TYPE NODEPORT (1/3)

```
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: whoami-nodeport  
spec:  
  type: NodePort  
  selector:  
    app: whoami  
  ports:  
    - nodePort: 30123  
      port: 8080  
      targetPort: 80
```

- k8s va réserver un port sur tous ses noeuds (le même sur tous les noeuds) et rediriger le trafic qui arrive sur ce port vers le Service concerné
- À noter : une **ClusterIP** sera aussi créée pour la communication interne



SERVICE DE TYPE NODEPORT (2/3)

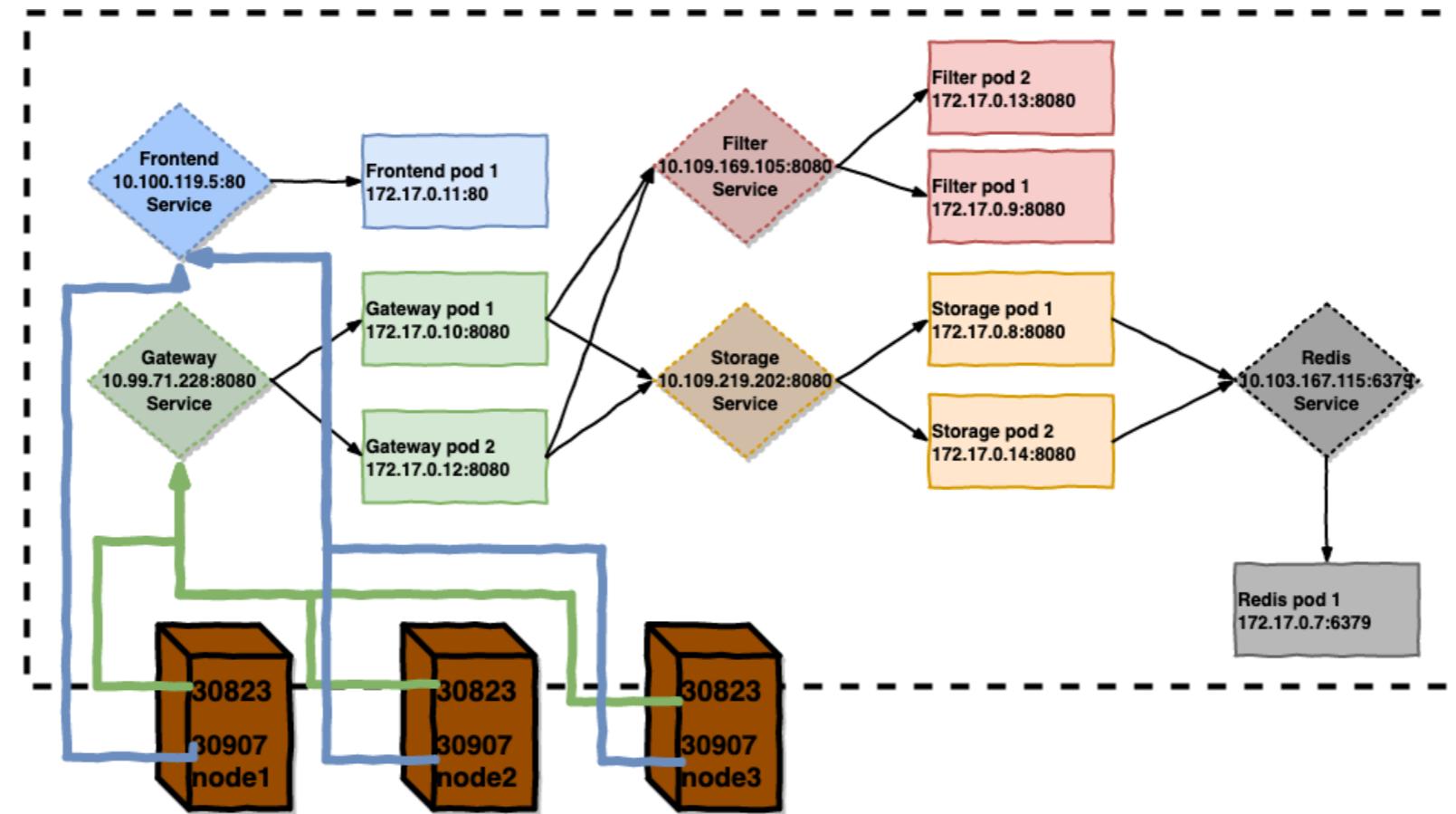
```
$ kubectl get svc whoami
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
whoami	NodePort	10.0.0.72	<none>	8080:30123/TCP	52m

- Le range des **NodePorts** disponible est prédéfini au niveau du cluster (par défaut 30000-32767)
- Il est possible de ne pas préciser la valeur du **.spec.ports.nodePort**, Kubernetes en attribuera automatiquement un de libre



SERVICE DE TYPE NODEPORT (3/3)



```
$ kubectl get services
svc/frontend      NodePort   10.100.119.5    <none>        80:30907/TCP
svc/gateway       NodePort   10.99.71.228   <none>        8080:30823/TCP
```

SERVICE DE TYPE LOADBALANCER



- Certains fournisseurs de services **Cloud** proposent une intégration de Kubernetes avec leurs fonctionnalités internes de LoadBalancing (AWS, Azure, GCP)
- En positionnant le **.spec.type** à **LoadBalancer**, Kubernetes va interagir avec l'api du **Cloud Provider** et provisionner/configurer automatiquement un loadbalancer associé au Service
- Cette configuration/provision est asynchrone
- L'information sur la configuration du loadbalancer n'apparaît que lorsque la configuration est terminée
- Techniquement, le LoadBalancer créé redirige le trafic vers le Service en passant par un **NodePort** qui est associé au Service (que vous n'avez pas à créer vous-même)



TP 4.2 : NodePort et LoadBalancer



POURQUOI A-T-ON BESOIN D'UN MÉCANISME SUPPLÉMENTAIRE ?

- Tous les clusters Kubernetes ne sont pas configurés pour créer dynamiquement des **LoadBalancers**
- Exposer ses Services par le biais de **NodePort** n'est pas très élégant
- Vous pouvez aussi éventuellement configurer un LoadBalancer non piloté par Kubernetes (HAProxy, F5, ...) ... si vous y avez accès
- Les Services fonctionnent au niveau de la couche **TCP** et ne permettent pas des pratiques telles que l'affinité par cookie ou d'autres configurations faites au niveau **HTTP**
- Les **Ingress**, oui 😊



AUTRES AVANTAGES DES INGRESS

- Chaque Service de type **LoadBalancer** nécessite son propre loadbalancer (au sens infra cloud) et sa propre IP
- Un **Ingress** ne nécessite qu'une seule IP, qui peut/sera partagée par plusieurs Services

INGRESS CONTROLLER



- L'utilisation des *Ingress* nécessite la mise en place d'un *Ingress Controller* (en général, ce sont les **ops** du cluster qui s'en occupent)
- Il existe plusieurs *Ingress Controllers* :
 - nginx
 - haproxy
 - traefik
 - contour (qui s'appuie sur envoy)
 - voyager
 - kanali
 - ...

CRÉER UNE RESSOURCE INGRESS



```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: whoami
spec:
  rules:
    - host: whoami.mycompany.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: whoami
                port:
                  number: 8080
```

- Toutes les requêtes (**Path** vaut `/`) pour le **Host `whoami.mycompany.com`** seront redirigées vers le port `8080` du Service **whoami**

DES CHEMINS DIFFÉRENTS VERS DES SERVICES DISTINCTS



```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: jj
spec:
  rules:
    - host: abrams.mycompany.com
      http:
        paths:
          - path: /alias
            pathType: Prefix
            backend:
              service:
                name: forty-seven
                port:
                  number: 80
          - path: /ze-others
            pathType: Prefix
            backend:
              service: (...)
```

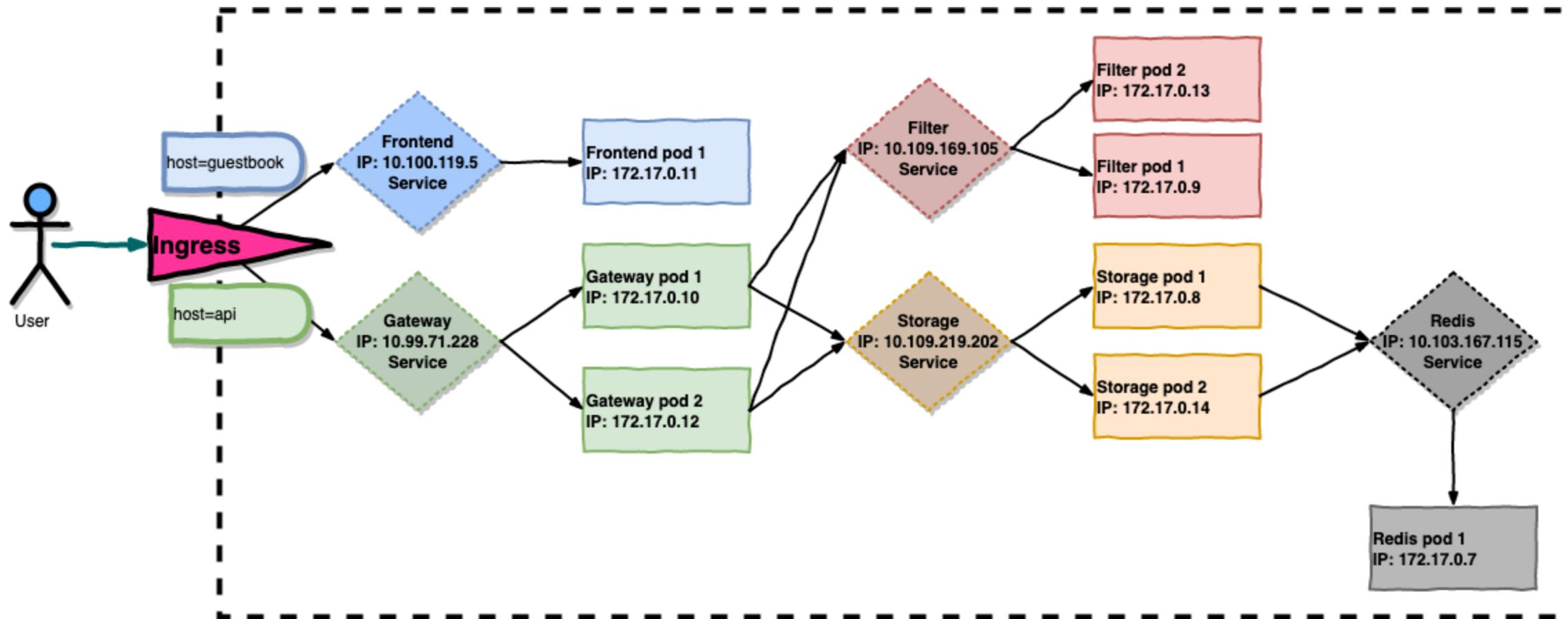


PLUSIEURS HOSTS

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: whoami
spec:
  rules:
    - host: fringe.mycompany.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: bishop
                port:
                  number: 80
    - host: lost.mycompany.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service: (...)
```



INGRESS FTW !



```
$ kubectl get ingresses
ing/frontend    guestbook.192.168.99.100.nip.io      80
ing/gateway     guestbook-api.192.168.99.100.nip.io    80
```



TP 4.3 : Ingress



SONDE READINESS

- En parallèle des sondes **Liveness** vues précédemment, existent les sondes de type **Readiness**
- Les sondes **Liveness** permettent à Kubernetes de savoir si votre application fonctionne correctement
- Les sondes **Readiness** permettent à Kubernetes de savoir si votre application est prête à recevoir du flux
 - si la sonde **Readiness** est en échec, et même si la sonde **Liveness** est **OK**, alors le Pod ne sera pas mis (ou sera exclu) du flux du Service associé. Il ne sera pas listé dans le **Endpoints** associé au Service
- Principe de *circuit-breaker*

LIVENESS VS READINESS



- Dans quel cas avoir un endpoint différent entre la sonde de **Liveness** et **Readiness** ?
 - pour rappel, la sonde **Liveness** provoque un redémarrage du conteneur.
- vérifications opérées par la sonde **Liveness** : problème mémoire, dead lock...
- vérifications opérées par la sonde **Readiness** : vérification de la bonne connexion à la base de données ou services externes

DÉFINITION D'UNE SONDE READINESS



```
----  
spec:  
  containers:  
    - name: whoami  
      image: containous/whoami:latest  
      readinessProbe:  
        httpGet:  
          path: /health  
          port: 80  
        initialDelaySeconds: 15  
        timeoutSeconds: 1
```



TP 4.4 : Sonde Readiness



SERVICES HEADLESS

- Un **Service** permet de donner un point d'entrée unique vers un ensemble de **Pods**
- Mais il arrive que le client a besoin de connaître les IPs des Pods plutôt que de passer par l'IP du Service (ex: Netflix Ribbon)
- Il est possible de créer un **Service headless**
- Pour cela, spécifier **clusterIP: None**, aucune **ClusterIP** ne sera créée
- Dans ce cas, la résolution **DNS** du Service retournera toutes les IPs des **Pods** au lieu de retourner une **ClusterIP**

ACCÉDER À UN POD SANS PASSER PAR UN SERVICE



- Il est aussi possible d'accéder à un Pod sans passer par un Service de manière temporaire, pour du debug notamment.
- La commande `kubectl port-forward` permet de forwarder temporairement un port local (du host d'où est lancée la commande `kubectl`) vers le port d'un Pod.

```
$ kubectl port-forward whoami 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Handling connection for 8080
Handling connection for 8080
```



TP 4.5 : Services headless, TP 4.6 : kubectl port-forward

SÉCURISER LES SERVICES AU SEIN DU CLUSTER



- Par défaut, tous les **Pods** acceptent du trafic de toutes les origines.
- Il serait utile de restreindre l'accès depuis/vers certains **Pods**
- Pour faire ça, on peut utiliser une **NetworkPolicy**
 - Les Pods seront isolés dès qu'ils correspondent au podSelector d'une **NetworkPolicy**
 - L'administrateur du cluster doit avoir installé un plugin réseau qui supporte les **NetworkPolicy**

Deux types de policy peuvent être définies :

- **egress**: trafic qui **sort** des **Pods** sélectionnés.
- **ingress**: trafic qui **rentre** vers les **Pods** sélectionnés.



EXEMPLE DE NETWORKPOLICY

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-allow
spec:
  podSelector:
    matchLabels:
      app: review
      role: api
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: review
```

Cette policy autorise uniquement l'accès aux **Pods** avec les labels **app=review** et **role=api** depuis les **Pods** avec le label **app=review**

ISOLER DES NAMESPACES ENTRE EUX



```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: my-project
  name: deny-from-other-namespaces
spec:
  podSelector:
    matchLabels:
  ingress:
    - from:
      - podSelector: {}
```

- Cette policy est déployée dans le Namespace **my-project**
- **podSelector.matchLabels** est vide, donc elle s'applique à tous les **Pods** du Namespace.
- **ingress.from.podSelector** est vide, donc elle **autorise** du trafic depuis tous les **Pods** du Namespace.





VOLUMES

AGENDA DE CE CHAPITRE 'VOLUMES'



- Présentation des Volumes
- Partage simple de données entre 2 containers d'un même Pod
- Accéder au fs d'un noeud du cluster
- PersistentVolumes et PersistentVolumeClaims

PRÉSENTATION DES VOLUMES



- Les fichiers créés dans un conteneur sont éphémères
- si un **Pod** est redémarré, les conteneurs créés pour la nouvelle instance du **Pod** n'auront pas accès aux fichiers du conteneur de l'instance précédente
- Un conteneur d'un **Pod** doit parfois partager des fichiers avec les autres conteneurs du même **Pod** (dans le cas d'un **Pod** multi-conteneurs)
- Les *Volumes* Kubernetes permettent de répondre à ces problématiques



TYPES DE VOLUMES (1/2)

- emptyDir : un répertoire initialement vide
- hostPath : un répertoire *monté* depuis le filesystem du noeud du cluster
- nfs : un partage NFS exposé comme un volume
- **gcePersistentDisk** (Google Compute Engine Persistent Disk), **awsElasticBlockStore** (Amazon Web Services Elastic Block Store Volume), **azureDisk** (Microsoft Azure Disk Volume) : des volumes exposés par les fournisseurs d'infrastructure Cloud



TYPES DE VOLUMES (2/2)

- cinder, cephfs, iscsi, flocker, glusterfs, quobyte, rbd, flexVolume, vsphereVolume, photonPersistentDisk, scaleIO : des volumes partagés dits **réseau (network storage)**
- **ConfigMaps**, **Secrets**, **downwardAPI** : des volumes spécialisés pour exposer certaines ressources Kubernetes
- persistentVolumeClaim : une manière d'allouer dynamiquement des volumes



VOLUME DE TYPE EMPTYDIR

- Un volume de type **emptyDir** fournit un volume ... vide au départ
- L'application qui tourne dans le **Pod** qui monte ce volume peut alors y écrire des fichiers
- Attention, le cycle de vie du volume **emptyDir** est lié à celui du **Pod**
- Si le **Pod** est détruit, le volume aussi
- Ce type de volume est parfaitement adapté dans le cas de conteneurs d'un même **Pod** qui doivent collaborer sur des fichiers communs



PARTAGE SIMPLE DE DONNÉES ENTRE 2 CONTENEURS D'UN MÊME POD

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: shared-vol  
spec:  
  volumes:  
    - name: var-log-nginx  
      emptyDir: {}  
  containers:  
    - name: log2fs  
      image: zenika/k8s-training-nginx-log2fs:1.19-alpine-v1  
      volumeMounts:  
        - name: var-log-nginx  
          mountPath: /var/log/nginx  
    - name: shell  
      image: debian:10-slim  
      command: ["bash", "-c", "sleep infinity"]  
      volumeMounts:  
        - name: var-log-nginx  
          mountPath: /data
```

ACCÉDER AU FS D'UN NOEUD DU CLUSTER



- La plupart des **Pods** devraient être agnostiques par rapport au noeud sur lequel ils tournent
- Mais certains **Pods** d'administration doivent pouvoir potentiellement accéder au filesystem du noeud sur lequel ils tournent :
 - par exemple, un **Pod** créé par un **DaemonSet** d'archivage des logs
- les volumes de type **hostPath** permettent d'accéder au filesystem des noeuds

PRÉCAUTIONS D'UTILISATION DES VOLUMES "HOSTPATH"



- Les données stockées dans les volumes **hostPath** sont persistantes (contrairement aux **emptyDir**)
- Mais il ne faut pas trop compter sur ce point car rien ne vous garantit que vos **Pods** seront relancés sur le même noeud



EXEMPLE DE VOLUME "HOSTPATH"

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: hostpath-vol  
spec:  
  volumes:  
    - name: var-log-nginx  
      hostPath:  
        path: /data/var-log-nginx  
  containers:  
    - name: log2fs  
      image: zenika/k8s-training-nginx-log2fs:1.19-alpine-v1  
      volumeMounts:  
        - name: var-log-nginx  
          mountPath: /var/log/nginx
```

PERSISTENT VOLUMES ET PERSISTENT VOLUMES CLAIMS



- Gérer les problématiques d'espace disque demande une approche dédiée
- Le principe des **PersistentVolumes** permet aux administrateurs de cluster de fournir une couche d'abstraction de consommation et de fourniture de **Volumes**
- Cette couche d'abstraction s'appuie sur les **PersistentVolumes** et les **PersistentVolumeClaims**
- Un **PersistentVolume** est un espace de stockage qui a été réservé par les admins du cluster
- Une **PersistentVolumeClaim** (littéralement **Demande de PersistentVolume**) est une requête d'espace de stockage faite par les utilisateurs du cluster



PERSISTENTVOLUMECLAIM VS PODS

- Les **PVC** sont semblables aux **Pods** :
 - Les **Pods** consomment des ressources (CPU & Mémoire) des noeuds du cluster
 - Les **PVC** consomment des ressources de stockage du cluster
- Les **PVC** fournissent des **PersistentVolumes** répondant à des critères :
 - notamment de taille et de type d'accès (ReadOnly/ReadWrite/...)
- ... tout en n'ayant pas à exposer aux utilisateurs du cluster la façon dont les volumes sont fournis
- Les **StorageClass** sont le moyen donné aux admins d'exposer les différents types de volumes disponibles

CYCLE DE VIE DES VOLUMES ET DES DEMANDES (CLAIMS)



- Les **PersistentVolumes** peuvent être provisionnés de manière **Statique** ou **Dynamique**
- Les **PV Statiques** sont des **PV** pré-provisionnés par les admins
- Les **PV Dynamiques** sont des **PV** provisionnés à la volée par le cluster en s'appuyant sur une **StorageClass**
- Une **PersistentVolumeClaim** doit spécifier le type de **StorageClass** souhaité
- Une **StorageClass** vide ("") équivaut à demander un **PV** statique

L'ÉTAPE D'ASSOCIATION ("BINDING")



- Un utilisateur crée une **PersistentVolumeClaim** en précisant la taille de l'espace de stockage et le mode d'accès souhaités
- Le cluster examine les **PersistentVolumes** disponibles et correspondant **a minima** à la demande
- L'utilisateur aura toujours **au moins** ce qu'il a demandé, mais peut avoir plus, et l'association entre la **PVC** et le **PV** est effectuée
- Si aucun **PV** ne correspond à la demande, elle reste en attente indéfiniment (ou jusqu'à sa suppression)

L'ÉTAPE D'UTILISATION ("USING")



- Le **Pod** utilise les demandes comme des volumes
- Le cluster inspecte la **PVC** pour retrouver le **PV** associé et monte le volume dans le **Pod**
- Une fois que l'utilisateur a créé une demande (**PVC**) et que cette demande est associée (à un **PV**), le **PV** associé appartient à l'utilisateur tant qu'il en a besoin

L'ÉTAPE DE LIBÉRATION ("RECLAIMING")



- Quand l'utilisateur n'a plus besoin du volume, il peut supprimer la **PVC** associée
- *Attention*, supprimer un **Pod** ne supprime pas les **PersistenceVolumeClaims** associées!
- La **ReclaimPolicy** associée au **PersistentVolume** détermine ce qui arrive au **PersistentVolume** une fois qu'il a été libéré :
- Les **PV** libérés peuvent être :
 - retenus
 - recyclés
 - supprimés



RETAINED/RECYCLED/DELETED

- retenu (**Retained**) signifie que le **PV** n'est plus utilisé mais ne peut pas être associé à une autre **PersistentVolumeClaim** tant qu'un admin ne le libère pas définitivement
- recyclé (**Recycled**) : le **PersistentVolume** est nettoyé, c.a.d les données qui y sont stockées sont supprimées. Une fois que c'est fait il peut être de nouveau associé à une **PVC**
- supprimé (**Deleted**) : le **PersistentVolume** est supprimé (ex AzureDisk, GC Disk, AWS EBS...)



ACCESS MODES

- **ReadWriteOnce** (RWO) : le volume peut être monté en read-write par un seul noeud
- **ReadOnlyMany** (ROX) : le volume peut être monté en read-only par plusieurs noeuds
- **ReadWriteMany** (RWX) : le volume peut être monté en read-write par plusieurs noeuds

⚠️ Un volume ne peut être monté que suivant un seul mode à la fois, même s'il supporte plusieurs modes



EXEMPLE DE PERSISTENTVOLUME

```
---  
apiVersion: v1  
kind: PersistentVolume  
metadata:  
  name: pv0003  
spec:  
  capacity:  
    storage: 5Gi  
  accessModes:  
    - ReadWriteOnce  
  persistentVolumeReclaimPolicy: Recycle  
  storageClassName: slow  
  nfs:  
    path: /tmp  
    server: 172.17.0.2  
  mountOptions:  
    - hard  
    - nfsvers=4.1
```

EXEMPLE DE PERSISTENTVOLUMECLAIM



```
---  
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
  name: myclaim  
spec:  
  accessModes:  
    - ReadWriteOnce  
  resources:  
    requests:  
      storage: 8Gi  
  storageClassName: slow  
  selector:  
    matchLabels:  
      release: "stable"  
    matchExpressions:  
      - {key: environment, operator: In, values: [dev]}
```



UTILISATION DE LA PVC

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: mypod  
spec:  
  volumes:  
    - name: mypd  
      persistentVolumeClaim:  
        claimName: myclaim  
  containers:  
    - name: myfrontend  
      image: nginx  
      volumeMounts:  
        - name: mypd  
          mountPath: /var/www/html
```



EXEMPLE DE STORAGECLASS

```
---  
apiVersion: storage.k8s.io/v1  
kind: StorageClass  
metadata:  
  name: standard  
provisioner: kubernetes.io/aws-ebs  
parameters:  
  type: gp2  
reclaimPolicy: Retain  
mountOptions:  
  - debug
```





CONFIGURATION ET SECRETS

AGENDA DE CE CHAPITRE 'CONFIGMAP ET SECRETS'



- Variables d'environnements
- ConfigMaps
- Secrets

VARIABLES D'ENVIRONNEMENT



- Une des promesses de Docker est de pouvoir utiliser la même image sur tous les environnements
- Mais chaque environnement a ses caractéristiques
- Il est commun d'adapter le comportement d'un conteneur en passant par des variables d'environnement
- Par exemple :
 - une url JDBC
 - le nom d'un service externe utilisé pour l'authentification
 - un profil Spring

DÉCLARATION D'UNE VARIABLE D'ENVIRONNEMENT



```
---  
apiVersion: apps/v1  
kind: ReplicaSet  
metadata:  
  name: docker-registry  
spec:  
  replicas: 1  
  template:  
    metadata:  
      labels:  
        app: registry  
    spec:  
      containers:  
      - name: registry  
        image: registry:2  
        env:  
        - name: REGISTRY_HTTP_DEBUG_ADDR  
          value: "localhost:9090"  
        - name: REGISTRY_HEALTH_HTTP_THRESHOLD  
          value: "5"
```



VARIABLES COMPOSÉES

- Il est aussi possible de faire référence à d'autres variables d'environnement

```
---  
...  
env:  
  - name: ENV_COLOR  
    value: "blue"  
  - name: COLORED_HOSTNAME  
    value: "$(ENV_COLOR)_$(HOSTNAME)"
```

CONFIGMAPS



- 🤢 L'inconvénient de passer par des variables d'environnement c'est que vous devrez construire des descripteurs pour chaque environnement
- Tout l'intérêt de la configuration c'est de pouvoir séparer les valeurs dépendantes de l'environnement du code source de l'application
- ... et vos descripteurs devraient faire partie du code source de l'application (Everything as a code)
- Kubernetes permet de séparer complètement la configuration d'une application par le biais de la ressource de type *ConfigMap*



EXEMPLE DE CONFIGMAP

```
---  
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: ma-premiere-config-map  
data:  
  ma.property.1: hello  
  ma.property.2: world  
  mon-fichier-de-properties: |-  
    property.1=value-1  
    property.2=value-2  
    property.3=value-3
```

CRÉATION DE CONFIGMAP



- Il est possible de créer une **ConfigMap** par le biais d'un descripteur
- Il est possible de créer une **ConfigMap** à partir de valeurs littérales
- Il est aussi possible d'importer un fichier ou un répertoire
- Dans le cas d'un répertoire, une clé sera créée pour chaque fichier



VALEURS LITTÉRALES

```
$ kubectl create configmap special-config \
--from-literal=special.how=very \
--from-literal=special.type=charm
```

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
data:
  special.how: very
  special.type: charm
```

CONFIGMAP À PARTIR D'UN RÉPERTOIRE



```
$ kubectl create configmap game-config --from-file= configs/
```

```
configs
  |- game.properties
    \- ui.properties
```

```
// game.properties
enemies=aliens
lives=3
enemies.cheat=true
secret.code.allowed=true
secret.code.lives=30
```

```
// ui.properties
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

RÉSULTAT DE L'IMPORT DE RÉPERTOIRE



```
$ kubectl get configmaps game-config -o yaml
```

```
---  
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: game-config  
data:  
  game.properties: |  
    enemies=aliens  
    lives=3  
    enemies.cheat=true  
    secret.code.allowed=true  
    secret.code.lives=30  
  ui.properties: |  
    color.good=purple  
    color.bad=yellow  
    allow.textmode=true  
    how.nice.to.look=fairlyNice
```

UTILISATION DES CONFIGMAPS : VAR D'ENV



```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: dapi-test-pod  
spec:  
  containers:  
    - name: test-container  
      image: debian:10-slim  
      command: [ "bash", "-c", "env" ]  
      env:  
        - name: SPECIAL_LEVEL_KEY  
          valueFrom:  
            configMapKeyRef:  
              name: special-config  
              key: special.how
```

RÉFÉRENCER TOUTES LES CLÉS D'UNE CONFIGMAP



```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: dapi-test-pod  
spec:  
  containers:  
    - name: test-container  
      image: debian:10-slim  
      command: [ "bash", "-c", "env" ]  
      envFrom:  
        - configMapRef:  
            name: special-config
```

CRÉER UN VOLUME À PARTIR D'UNE CONFIGMAP



```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: dapi-test-pod  
spec:  
  volumes:  
    - name: config-volume  
      configMap:  
        name: game-config  
  containers:  
    - name: test-container  
      image: debian:10-slim  
      command: [ "bash", "-c", "ls /etc/config/" ]  
      volumeMounts:  
        - name: config-volume  
          mountPath: /etc/config
```

- Autant de fichiers que de clés dans la ConfigMap seront créés
- Chaque fichier contiendra le contenu de chaque clé

CONTRAINTES D'UTILISATION DES CONFIGMAPS



- Vous devez créer une ConfigMap **AVANT** de la référencer dans un Pod
- Si vous référez une ConfigMap qui n'existe pas dans une définition de Pod, le Pod pourra être créé, mais les conteneurs seront en erreur
- De même si vous référez une clé qui n'existe pas
- Une **ConfigMap** n'est valide/utilisable que dans le **Namespace** où elle a été créée



TP 5.1 : ConfigMaps



SECRETS

- Les objets de type *Secret* sont faits pour stocker des informations sensibles telles que des mots de passe, des tokens OAuth, des certificats ou des clés SSH
- Stocker ces informations dans un *Secret* est plus sûr et flexible que de les stocker en dur dans un *Pod* ou même dans une *ConfigMap* (même si ce n'est pas encore idéal)
- En fait, les *Secrets* sont des sortes de *ConfigMap* personnalisées



STOCKER DES SECRETS (1/2)

- Soit vous encodez les valeurs que vous voulez stocker en **Base64**

```
$ echo -n "admin" | base64  
YWRtaW4=  
$ echo -n "1f2d1e2e67df" | base64  
MjYyZDFlMmU2N2Rm
```

```
---  
apiVersion: v1  
kind: Secret  
metadata:  
  name: mysecret  
type: Opaque  
data:  
  username: YWRtaW4=  
  password: MjYyZDFlMmU2N2Rm
```



STOCKER DES SECRETS (1/2)

- Soit vous laissez faire Kubernetes

```
---  
apiVersion: v1  
kind: Secret  
metadata:  
  name: mysecret  
type: Opaque  
stringData:  
  username: admin  
  password: 1f2d1e2e67df
```



VISUALISER UN SECRET

```
$ kubectl get secret mysecret -o yaml
```

```
---  
apiVersion: v1  
data:  
  username: YWRtaW4=  
  password: MWYyZDFlMmU2N2Rm  
kind: Secret  
type: Opaque
```

UTILISER UN SECRET POUR CRÉER DES FICHIERS



```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: mypod  
spec:  
  volumes:  
    - name: foo  
      secret:  
        secretName: mysecret  
  containers:  
    - name: mypod  
      image: redis  
      volumeMounts:  
        - name: foo  
          mountPath: "/etc/foo"  
          readOnly: true
```

Chaque clé dans le Secret devient un fichier



UTILISER UN SECRET POUR RENSEIGNER UNE VARIABLE D'ENV

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: secret-env-pod  
spec:  
  containers:  
    - name: mycontainer  
      image: redis  
      env:  
        - name: SECRET_USERNAME  
          valueFrom:  
            secretKeyRef:  
              name: mysecret  
              key: username  
        - name: SECRET_PASSWORD  
          valueFrom:  
            secretKeyRef:  
              name: mysecret  
              key: password
```



TP 5.2 : Secrets

UTILISER UN SECRET POUR RÉCUPÉRER UNE IMAGE SUR UN REGISTRY PRIVÉ



```
apiVersion: v1
kind: Secret
metadata:
  name: my-docker-creds-secret
data:
  .dockerconfigjson: eyJodHRwczovL2luZG54L ... J0QUL6RTIifX0=
type: kubernetes.io/dockerconfigjson
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-private-image
spec:
  containers:
    - name: mycontainer
      image: registry.internal.company.com/my-secret-image
  imagePullSecrets:
    - name: my-docker-creds-secret
```

UTILISER UN SECRET POUR EXPOSER UN INGRESS EN HTTPS

```
apiVersion: v1
kind: Secret
metadata:
  name: testsecret-tls
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls
```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tls-example-ingress
spec:
  tls:
    - hosts:
        - https-example.foo.com
      secretName: testsecret-tls
  rules:
    - host: https-example.foo.com
      ...

```



UTILISER UN VAULTSECRET (CAGIP)



- Opérateur créé un vault path lorsqu'un nouveau projet (CRD projet) est créé.
- Configuration d'un appRole stocké sous forme d'un secret dans le namespace associé
 - Credentials pour lecture du path vault

CRÉATION D'UNE CRD VAULTSECRET.



- Enregistrer ses secrets projets dans `secret/{{entité}}/{{projet}}/k8s/{{environnement}}/`
 - `environnement` est `{{clusterName}}-{{maturité}}` (maturité : development, integration, uat, preproduction, production)

```
apiVersion: ca-gip.github.com/v1
kind: VaultSecret
metadata:
  name: my-vault-secret
spec:
  secretName: my-secret # nom du Secret K8S souhaité
  secretPath: secret/data/{{ chemin du catalogue dans vault }}
  version: "1" # Optionnel, représente la version du secret dans le store KV Vault (la dernière
est chargée si absent)
  base64: true # Optionnel, ajouter cette option si votre secret est encodé en base64 dans le
catalogue Vault
```

Le secret créé portera le nom défini dans `spec.secretName` (ici `my-secret`)



UTILISATION DU SECRET

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: vault-secret-env-pod  
spec:  
  containers:  
    - name: mycontainer  
      image: myimage:v1.0  
      env:  
        - name: SECRET_PASSWORD  
          valueFrom:  
            secretKeyRef:  
              name: my-secret # nom du Secret K8S créé par le VaultSecret  
              key: password # clé d'une des paires KV définies dans le catalogue Vault
```

Attention: lors de la mise à jour du secret dans Vault, la nouvelle valeur du secret n'est *pas propagée* dans le cluster Kubernetes, *il faut redéployer* le VaultSecret avec une **version** incrémentée et update le service pour utiliser le nouveau VaultSecret.





STRATÉGIES DE DÉPLOIEMENT

AGENDA DE CE CHAPITRE 'STRATÉGIES DE DÉPLOIEMENT'



- Mise à disposition d'une nouvelle version d'un Pod
- Rolling Update

MISE À DISPOSITION D'UNE NOUVELLE VERSION D'UN

POD

- Il existe plusieurs stratégies de mise à jour d'un ensemble de Pods

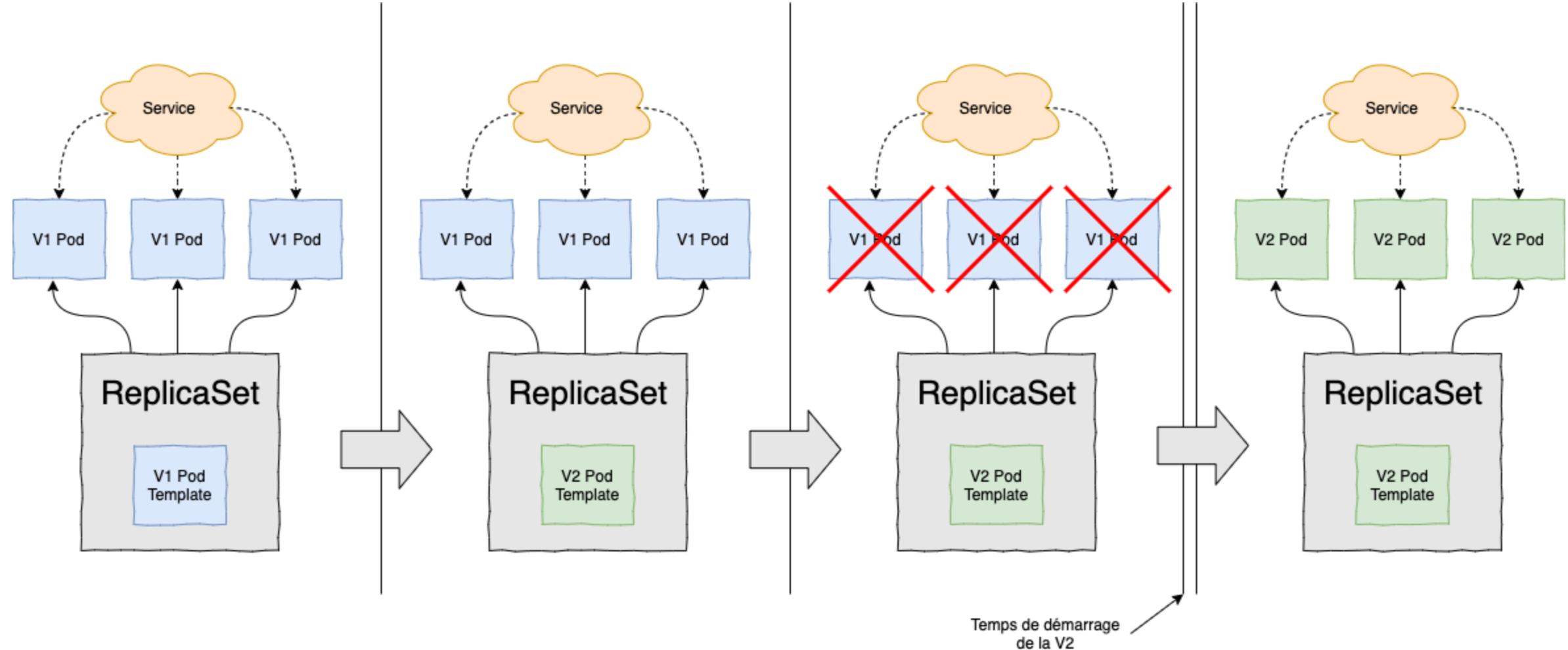


REPLICASET FTW (1/2)

- Étape 1 : mettre à jour le template
- Étape 2 : supprimer les Pods utilisant l'ancienne version
- Étape 3 : attendre



REPLICASET FTW (2/2)



L'inconvénient majeur est qu'il faut accepter une interruption de service le temps que le **RS** crée les Pods avec le nouveau template

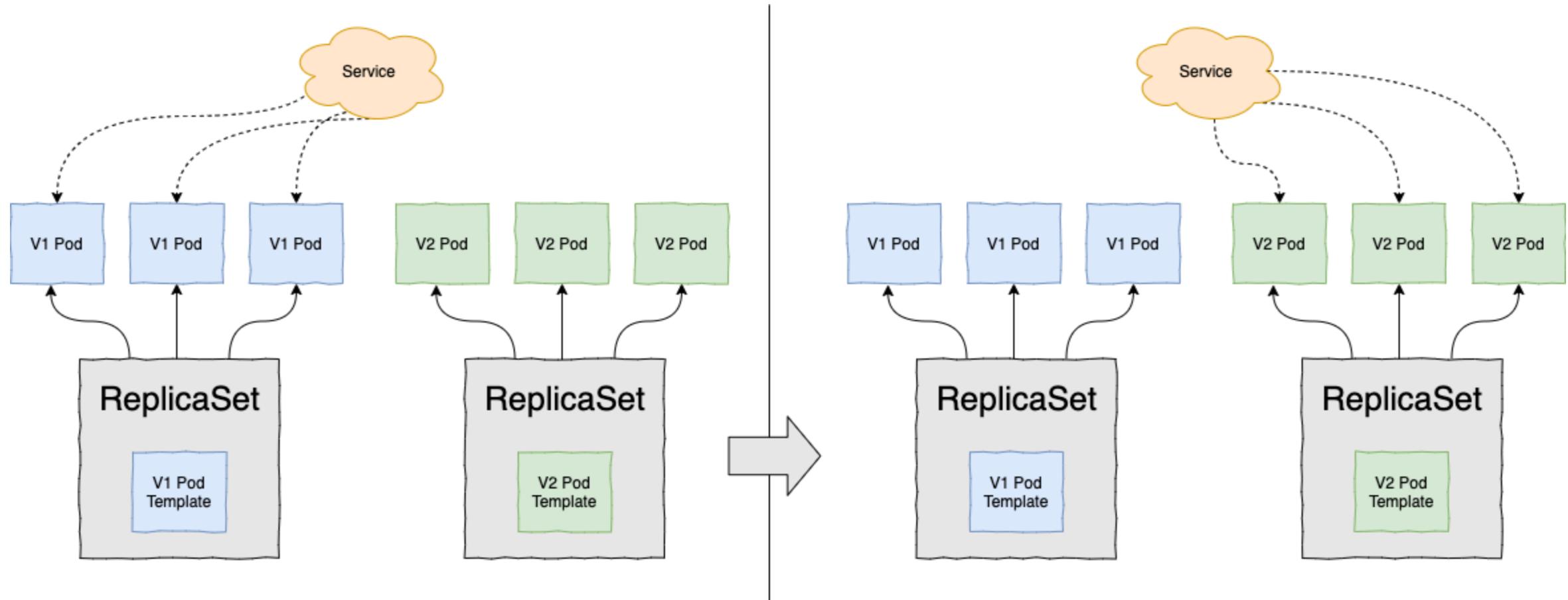


REPLICASET + SERVICE (1/2)

- Étape 1 : créer un nouveau ReplicatSet dédié à la nouvelle version
- Étape 2 : quand tous les Pods sont READY, changer le sélecteur du Service
- Étape 3 : supprimer le ReplicaSet associé à l'ancienne version



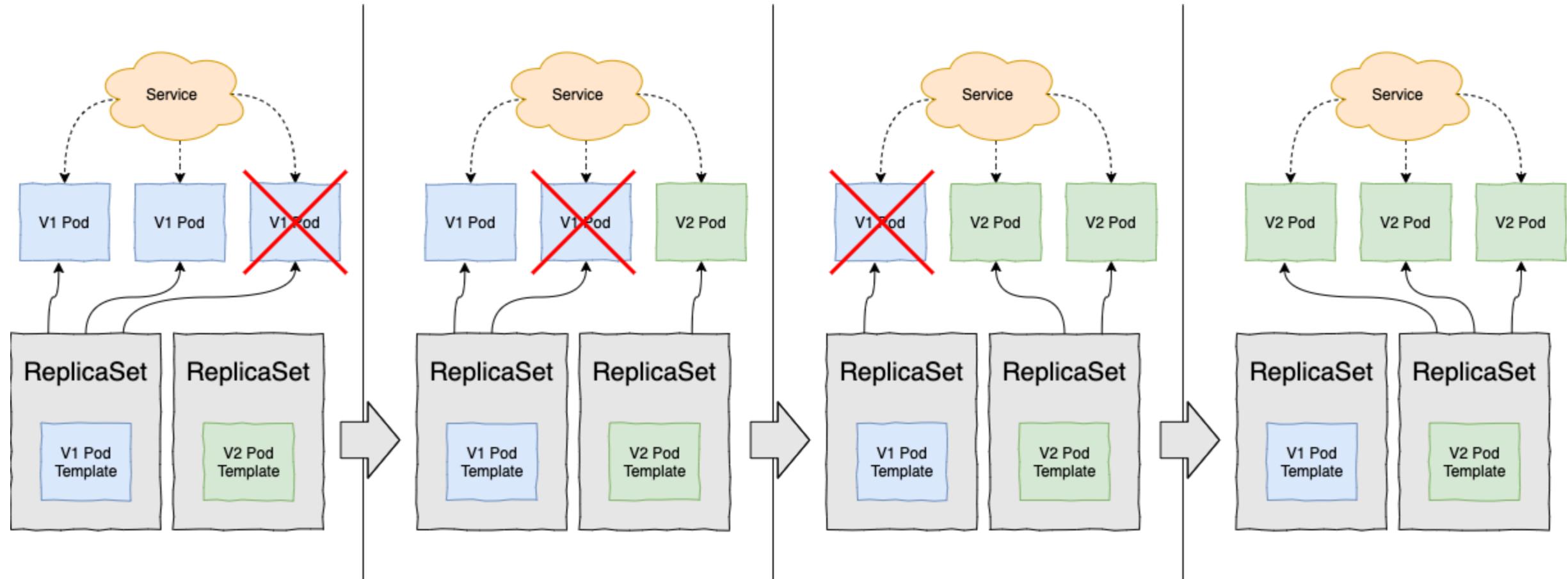
REPLICASET + SERVICE (2/2)



Inconvénient : nécessite à un moment d'avoir 2 versions de l'application (et donc 2 fois plus de ressources consommées)

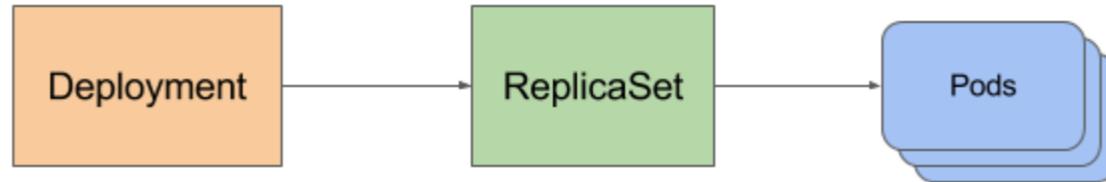


ROLLING UPDATE MANUEL



Inconvénient : nécessite beaucoup d'opérations manuelles...donc source d'erreurs !

DEPLOYMENT FTW !



DESCRIPTEUR DE DEPLOYMENT



```
---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: zenika  
spec:  
  replicas: 3  
  selector:  
    matchLabels:  
      app: zenika  
  revisionHistoryLimit: 5  
  progressDeadlineSeconds: 60  
  template:  
    metadata:  
      labels:  
        app: zenika  
    spec:  
      containers:  
        - name: app  
          image: zenika/k8s-training-deploy:v1
```



DEPLOYMENT STATUS

- Comment savoir si le déploiement s'est bien passé ?

```
$ kubectl apply -f zenika-deployment-v1.yml
```

```
$ kubectl rollout status deployment zenika
deployment zenika successfully rolled out
```

```
$ kubectl get deploy -l app=zenika
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
zenika    3          3          3           3           16m
```

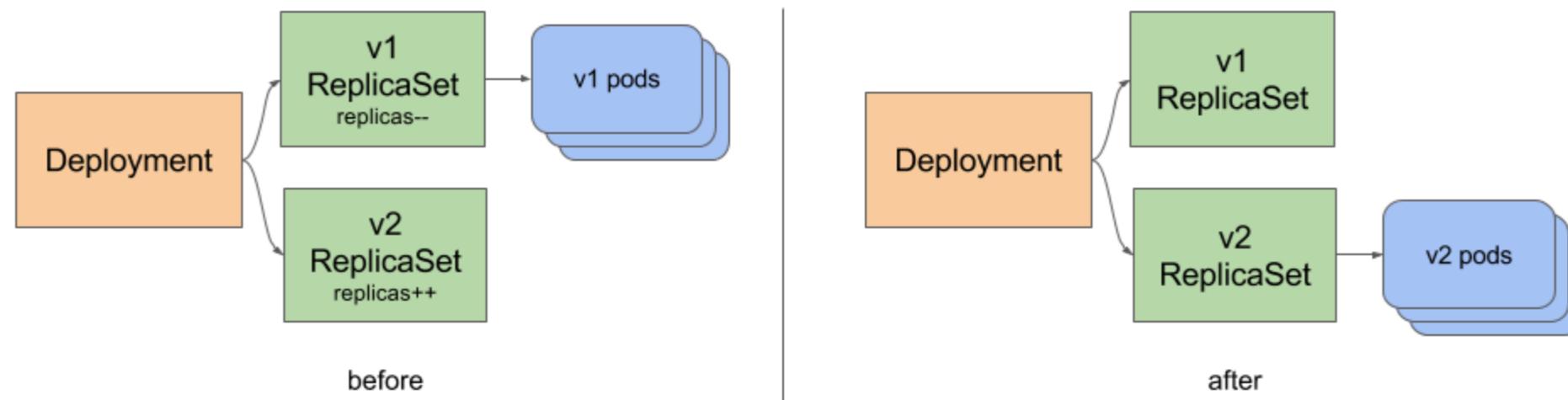
```
$ kubectl get rs -l app=zenika
NAME      DESIRED   CURRENT   READY      AGE
zenika-3842228658 3          3          3          24s
```

```
$ kubectl get po -l app=zenika
NAME                  READY   STATUS    RESTARTS   AGE
zenika-3842228658-l0jwc 1/1     Running   0          54s
zenika-3842228658-pd3s9  1/1     Running   0          54s
zenika-3842228658-s5ww2  1/1     Running   0          54s
```

MISE À JOUR



```
$ kubectl get rs -l app=zenika
NAME          DESIRED   CURRENT   READY   AGE
zenika-3842228658  0         0         0      2m
zenika-43373277    3         3         3      52s
```





AUTRES COMMANDES

```
$ kubectl rollout history deploy frontend  
deployments "frontend"  
REVISION  CHANGE-CAUSE  
17        kubectl apply --filename=frontend-b1386-d1387.yml  
18        kubectl apply --filename=frontend-b1391-d1392.yml  
19        kubectl apply --filename=frontend-b1405-d1407.yml  
20        kubectl apply --filename=frontend-b1409-d1410.yml  
21        kubectl apply --filename=frontend-b1414-d1415.yml  
  
$ kubectl rollout undo deployment frontend --to-revision=20
```

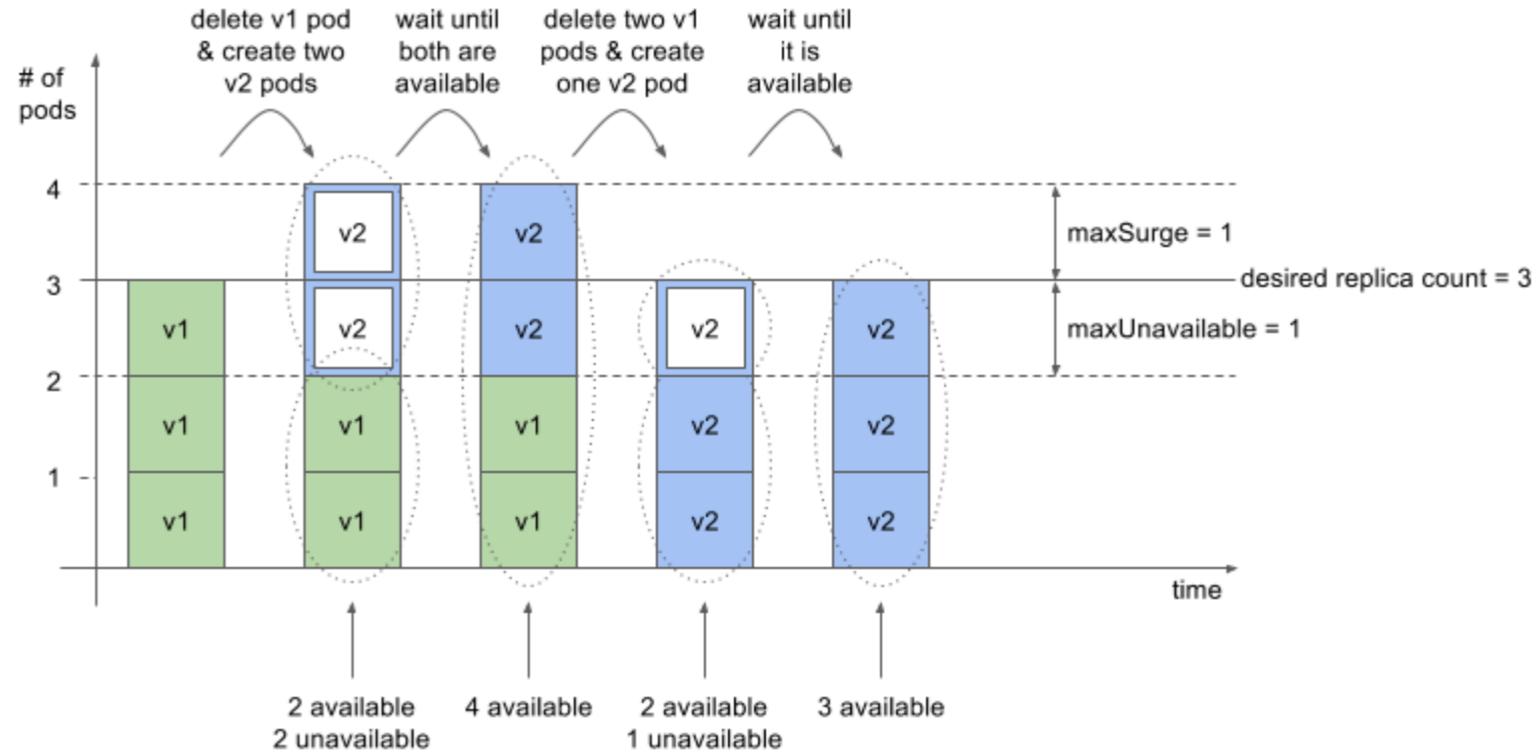


CONTROLLER LE "ROLLOUT"

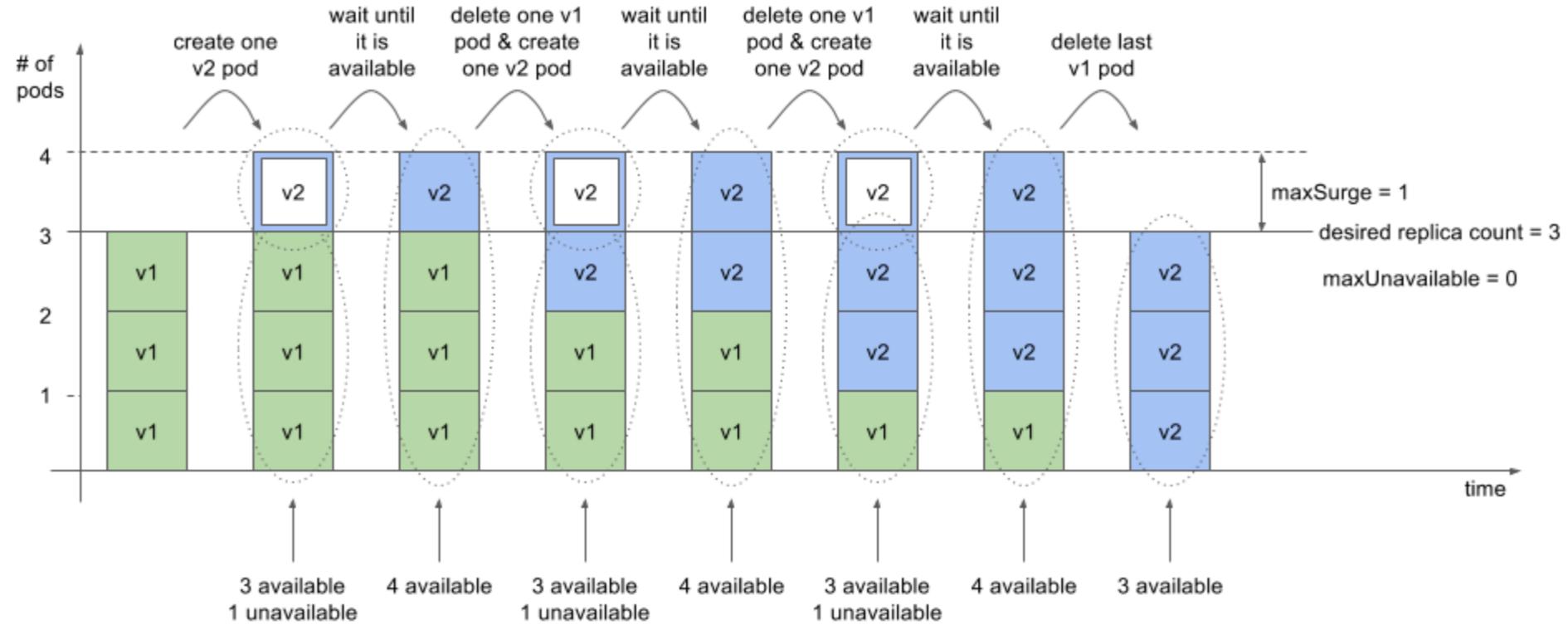
- **maxSurge** (défaut 25%) : combien d'instances de Pod au dessus du nombre de replicas cible on autorise, la valeur absolue est arrondie à la valeur entière supérieure.
 - si nb_replicas=4, il n'y aura jamais plus que 5 Pods "ready"
- **maxUnavailable** (défaut 25%) : combien d'instances de Pods peuvent être indisponibles par rapport au nombre de replicas cible, la valeur absolue est arrondie à la valeur entière inférieure.
 - si nb_replicas=4, il n'y aura jamais moins de 3 Pods "ready"

```
spec:  
  strategy:  
    type: RollingUpdate  
    rollingUpdate:  
      maxSurge: 30%  
      maxUnavailable: 0
```

REPLICAS=3, MAXSURGE=1, MAXUNAVAILABLE=1



REPLICAS=3, MAXSURGE=1, MAXUNAVAILABLE=0



STRATÉGIE DE RECRÉATION



Si vous ne pouvez avoir 2 versions de votre application déployées en même temps, il existe une autre stratégie : **Recreate**

1. Supprimer tous les Pods existants
2. Créer les nouveaux Pods

```
spec:  
  strategy:  
    type: Recreate
```

⚠ Cette stratégie implique une interruption de service.

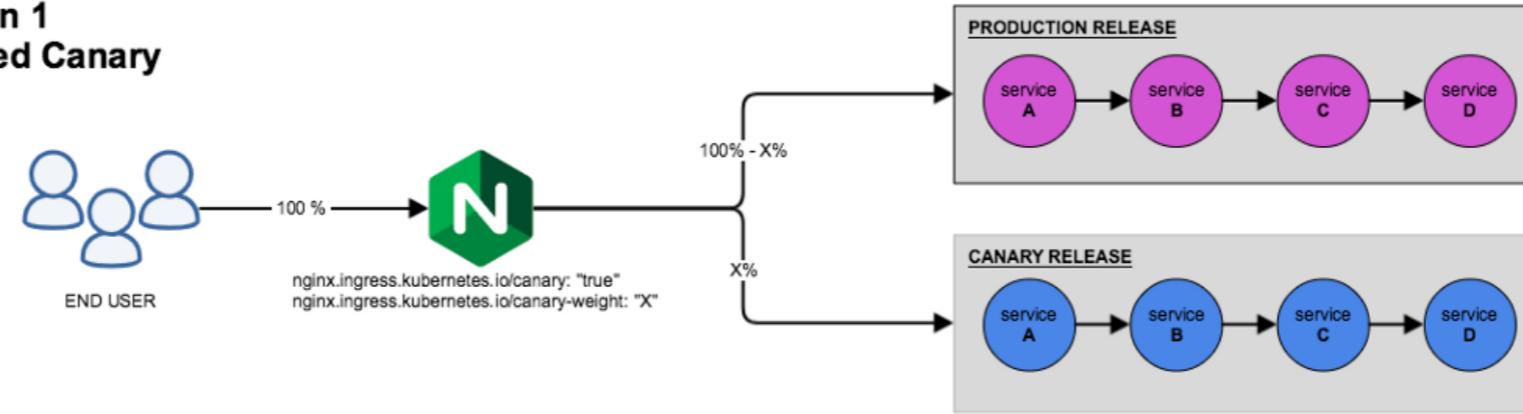


TP 6.1 : Deployments

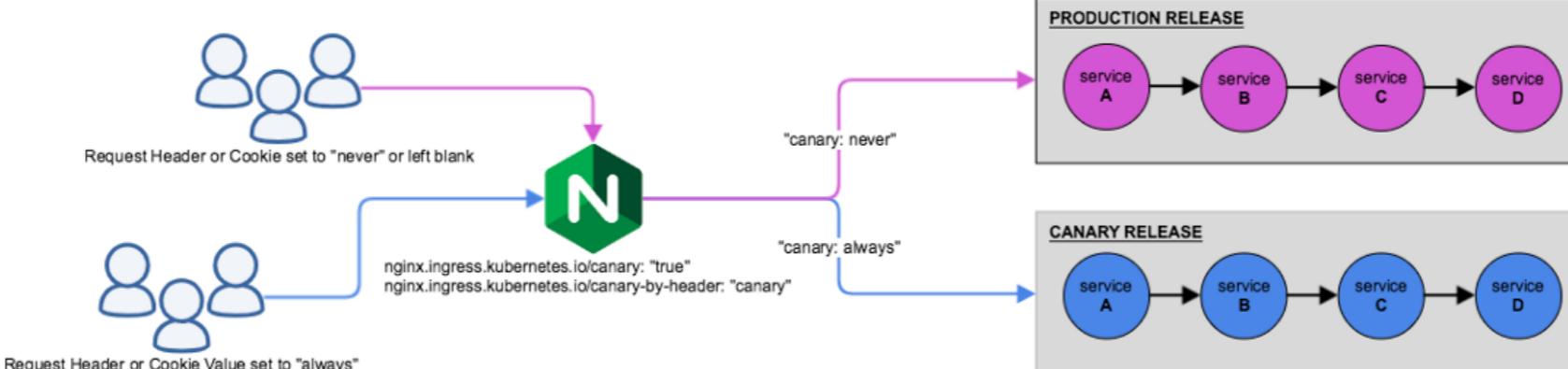
CANARY : INGRESS + ANNOTATION



Option 1 Weight-Based Canary



Option 2 User-Based Canary



CANARY : INGRESS + ANNOTATION ✨



Utilisation d'annotations pour piloter la configuration du traffic.

- `nginx.ingress.kubernetes.io/canary`: "true" pour activer la fonctionnalité
- `nginx.ingress.kubernetes.io/canary-weight` : % de request
- `nginx.ingress.kubernetes.io/canary-by-header` : Présence d'un header http
- `nginx.ingress.kubernetes.io/canary-by-cookie` : Présence d'un cookie http
- `nginx.ingress.kubernetes.io/canary-by-header-value ...`
- `nginx.ingress.kubernetes.io/canary-by-header-pattern ...`



EXEMPLE DE CANARY

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: whoami
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-weight: "10"
spec:
  rules:
  - host: whoamiFIXME.nip.io
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: whoami-int
            port:
              number: 8080
```



TP 6.2 : Canary





OPÉRATEURS

SCENARIOS



Imaginez que vous vouliez:

- Créer un objet Postgresql pour faciliter la mise en place de base de données dans votre cluster.

CONTROLLERS



Dans Kubernetes, un **controller** est une boucle de contrôle qui va observer les changements d'états du cluster à travers l'apiserver. Si l'état désiré et l'état actuel du cluster ne sont pas au même niveau, le controller va créer les tâches nécessaires pour réconcilier les deux états.

- Généralement écrit en Golang
- Il est recommandé de limiter les droits d'un controller en utilisant un service account dédié
- Autre recommandation : KISS (Keep it Stupid Simple)



CUSTOM RESOURCE DEFINITION (1/2)

La ressource **CustomResourceDefinition**(CRD) permet de définir une ressource qui n'est pas de base dans Kubernetes. Définir une CRD permet d'avoir un type de ressource avec le nom et le schéma que l'on veut. L'API de Kubernetes va gérer le stockage et requêtage de cette nouvelle ressource.

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
```

```
$ kubectl get crontab
```

NAME	AGE
my-new-cron-object	6s



CUSTOM RESOURCE DEFINITION (2/2)

Exemple de définition:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
      - ct
```



OPERATORS



Kubernetes Operators est un concept qui a été créé par [CoreOS](#)

La fonction principale d'un Operator est de :

- Lire depuis un objet custom qui représente votre ressource
- Faire les changements nécessaire pour atteindre l'état désiré

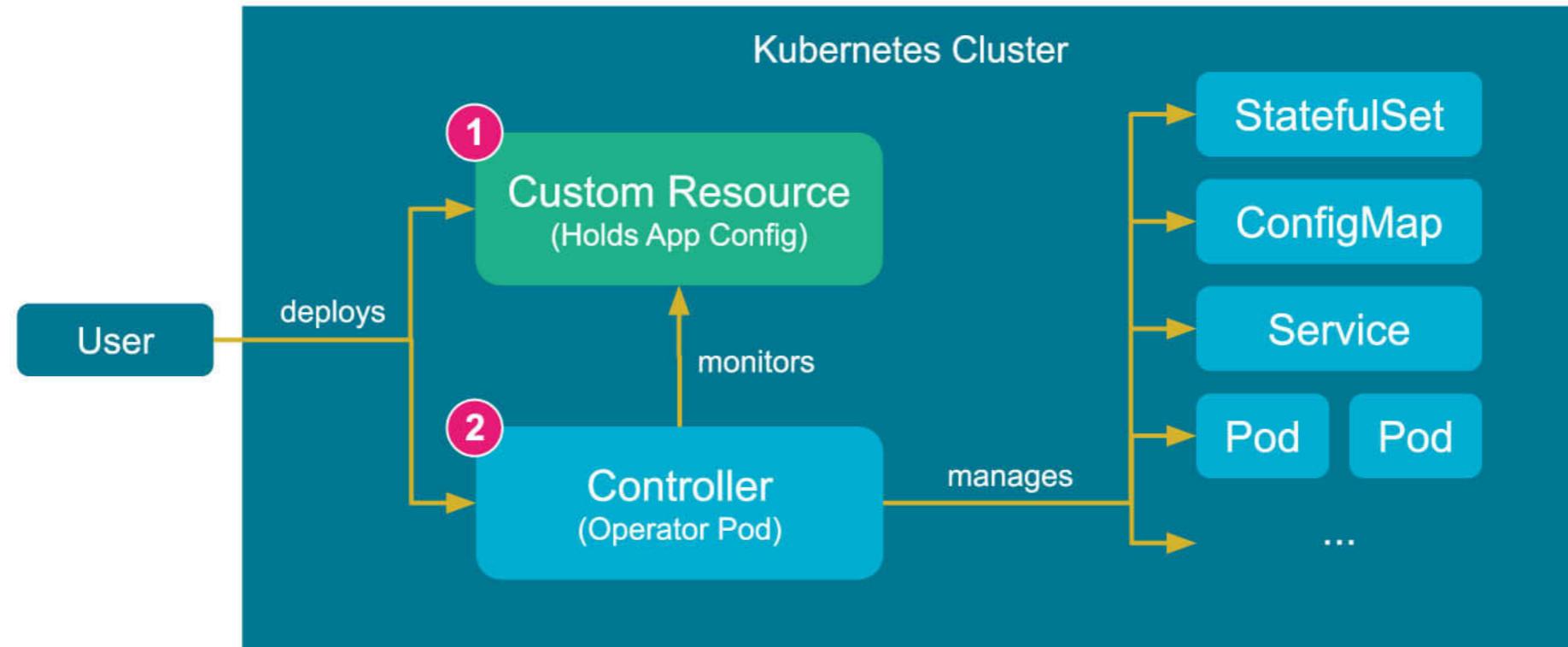
Exemple:

```
apiVersion: "vault.security.coreos.com/v1alpha1"
kind: "VaultService"
metadata:
  name: "example"
spec:
  nodes: 2
  version: "0.9.1-0"
```

OPERATORS



Operator = controller + API extension + single-app focus



OPERATORS



<https://operatorhub.io/>



TP 7.1 : Opérateur prometheus





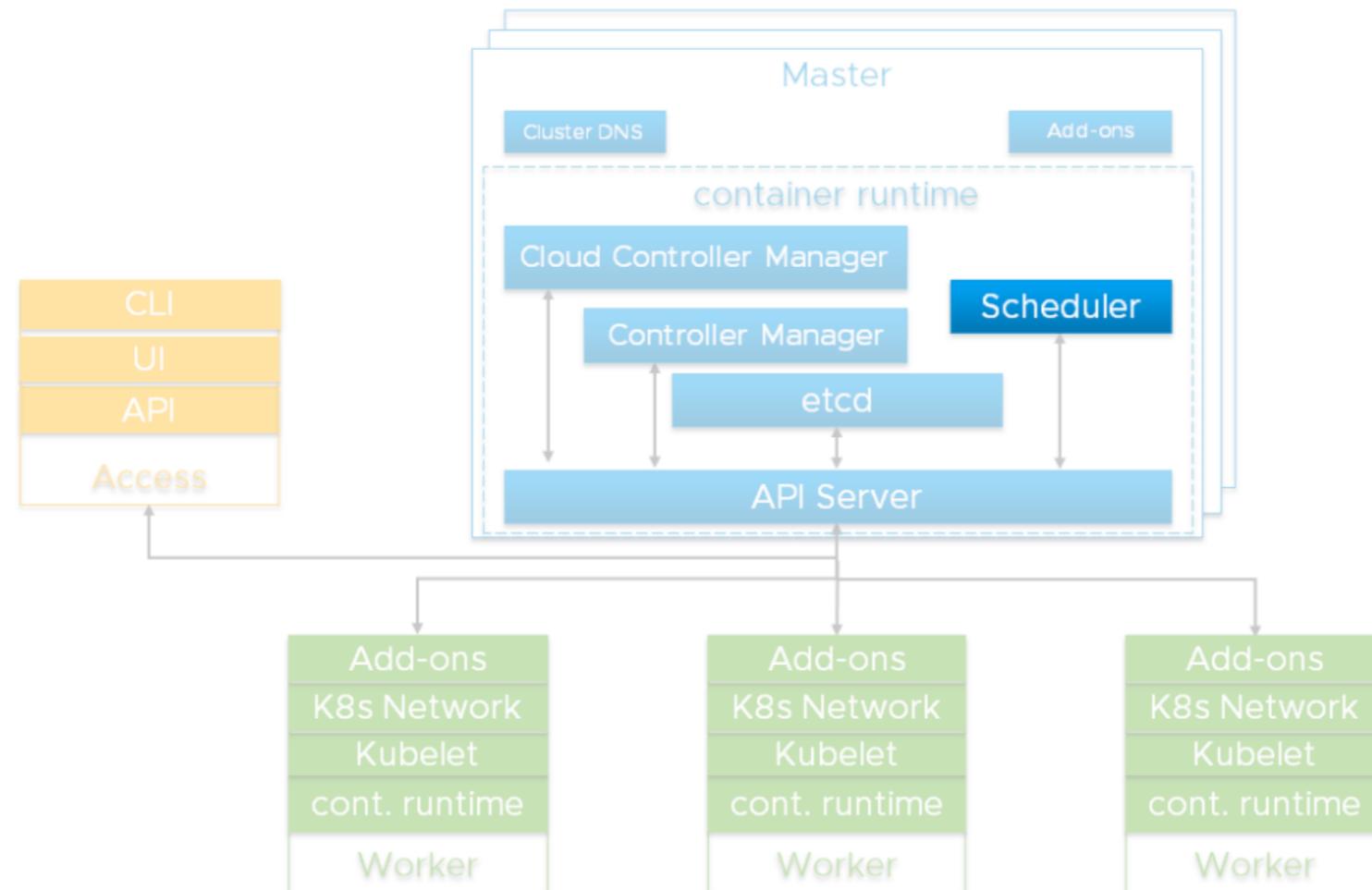
FONCTIONNALITÉS D'ENTREPRISE

AGENDA



- Scheduling
- gestion des ressources
- auto-scaling

SCHEDULER



SCHEDULER



- Définit où exécuter les charges de travail en fonction des ressources disponibles
- Fonctionne en mode actif / passif
 - Le planificateur actif actuel est marqué dans **etcd**
- Deux étapes
 - Filtrage par prédictats
 - Notation par priorités
 - Si deux nœuds obtiennent le même score, choix aléatoire

SCHEDULER - FILTERING



Liste des prédictats de filtre disponible :

- PodFitsResources
- CheckNodeMemoryPressure
- CheckNodeDiskPressure
- PodFitsHostPorts
- HostName
- MatchNodeSelector
- NoDiskConflict
- NoVolumeZoneConflict
- MaxEBSVolumeCount
- MaxGCEPDVolumeCount



SCHEDULER - RANKING (1/2)

Par exemple, supposons qu'il existe deux fonctions de priorité, `priorityFunc1` et `priorityFunc2` avec respectivement des facteurs de pondération `weight1` et `weight2`, le score final de certains `NodeA` est:

```
finalScoreNodeA = (poids1 \ * priorityFunc1) + (poids2 \ * priorityFunc2)
```

Une fois les scores de tous les nœuds calculés, le nœud avec le score le plus élevé est choisi comme hôte du `Pod`. S'il y a plus d'un nœud avec les scores les plus élevés égaux, un aléatoire parmi eux est choisi.



SCHEDULER - RANKING (2/2)

- Stratégie par défaut de tri :
 - `SelectorSpreadPriority`
 - `InterPodAffinityPriority`
 - `LeastRequestedPriority`
 - `BalancedResourceAllocation`
 - `NodePreferAvoidPodsPriority`
 - `NodeAffinityPriority`
 - `TaintTolerationPriority`
 - `ImageLocalityPriority`

GESTION DU SCHEDULING



Le scheduler Kubernetes peut connaître la topologie du cluster et répartir les charges de travail afin d'augmenter la résilience.

Ce mécanisme utilise:

- La fonction de priorité **SelectorSpreadPriority**
- Les labels **topology.kubernetes.io/zone** et **topology.kubernetes.io/region** sur les nœuds de cluster
- Sur un Kubernetes géré, les labels sont définis par votre fournisseur
- Lorsque vous utilisez votre propre cluster, vous devez vous assurer de les définir.

voir: <https://kubernetes.io/docs/reference/kubernetes-api/labels-annotations-taints/#topologykubernetesioregion>

SCHEDULING



Il existe plusieurs façons de contrôler la façon dont le scheduler attribuera des **Pods** au **Node**:

- **NodeSelector**
 - **Pod** sera assigné à **Nodes** avec un sélecteur de label
 - sera obsolète au profit de **NodeAffinity**
- **NodeAffinity**: peut exprimer l'exigence et / ou la préférence
- **PodAffinity**: pour colocaliser des **Pods**
- **PodAntiAffinity**: pour s'assurer que deux **Pods** ne seront pas sur le même **Node**

NODE AFFINITY



Il existe deux types d'affinités:

- preferredDuringSchedulingIgnoredDuringExecution
- requiredDuringSchedulingIgnoredDuringExecution

Un troisième requiredDuringSchedulingRequiredDuringExecution est prévu, il ajoutera l'éviction si le Node ne correspond plus au sélecteur.



NODE AFFINITY OPERATORS

Exemple de sélecteur de nœud pour l'affinité

```
nodeSelectorTerms:  
  - matchExpressions:  
    - key: kubernetes.io/env  
      operator: In  
      values:  
        - qa
```

In, NotIn, Exists, DoesNotExist, Gt, Lt



NODE AFFINITY EXAMPLE

Exemple de pod avec affinité de nœud

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/e2e-az-name
                operator: In
                values:
                  - e2e-az1
  containers:
    - name: with-node-affinity
      image: k8s.gcr.io/pause:2.0
```



POD AFFINITY/ANTI-AFFINITY

Le mécanisme d'affinité / anti-affinité **Pod** est utilisé pour s'assurer que les **Pods**:

- Seront exécuter sur un **Node** différent ou un ensemble de **Nodes**
- Seront exécuter sur le même **Node** ou ensemble de **Nodes**

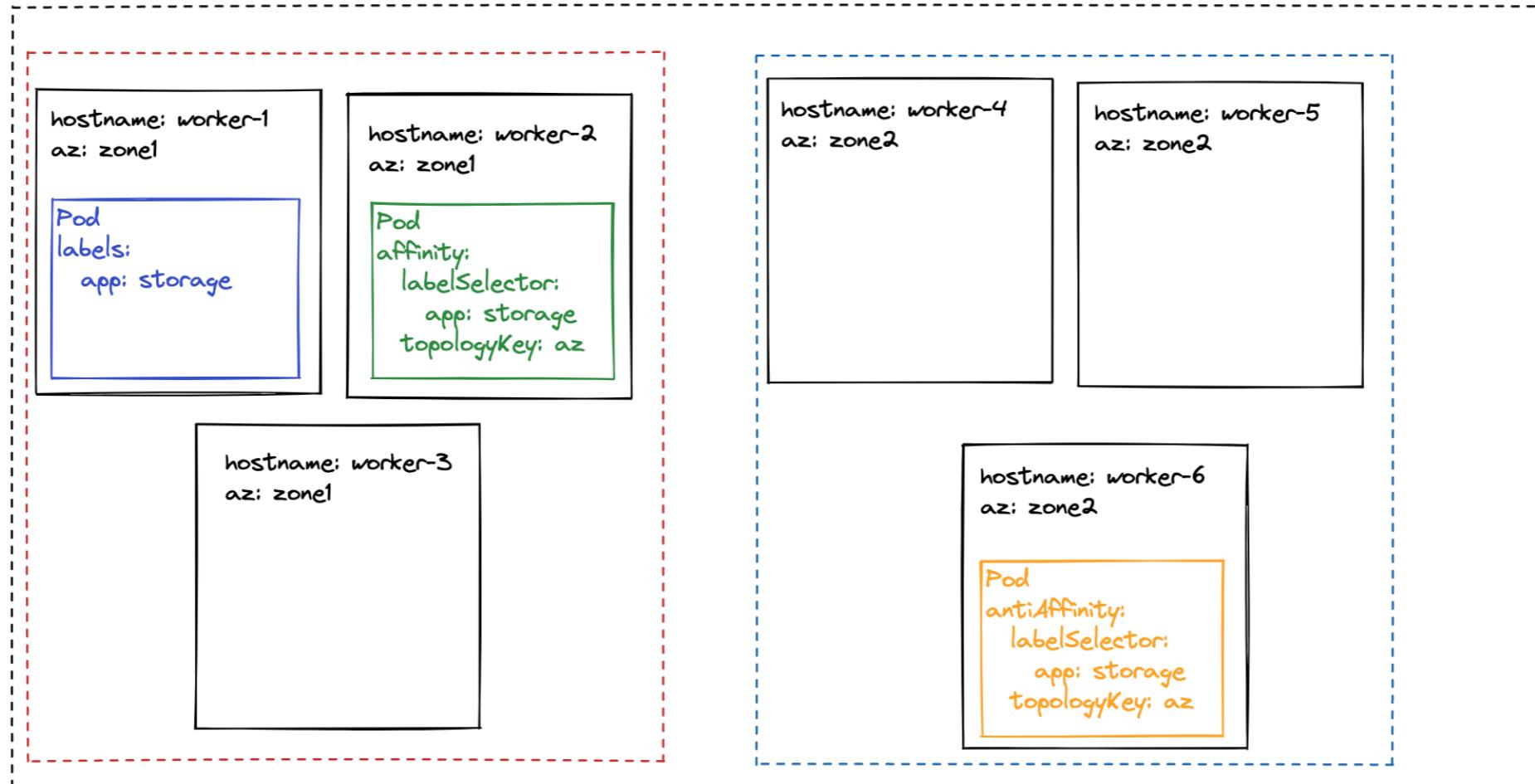
Il peut être utilisé pour:

- Répartir les pods d'un **ReplicaSet** ou **StatefulSet** sur différents **Nodes** et améliorez la résilience
- Colocaliser certains **Pods** sur le même **Node** ou zone de disponibilité pour réduire la latence

 More examples [here](#)



POD AFFINITY/ANTI-AFFINITY





POD AFFINITY EXAMPLE

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: app
                operator: In
                values:
                  - storage
      topologyKey: az
  containers:
    - name: with-pod-affinity
      image: k8s.gcr.io/pause:2.0
```

POD ANTI-AFFINITY EXAMPLE



```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-anti-affinity
spec:
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchExpressions:
                - key: app
                  operator: In
                  values:
                    - storage
            topologyKey: az
  containers:
    - name: with-pod-anti-affinity
      image: k8s.gcr.io/pause:2.0
```





TP 8.1 : Affinity



TAINTS AND TOLERATIONS

Les **Taints** sont utilisées pour marquer certains **Nodes** comme non utilisables par des **Pods** sans la **Toleration** correspondante.

Ce mecanisme peut être utilisé pour:

- Réserver certains nœuds à certains utilisateurs ou usages
- Réserver certains nœuds avec du matériel spécial



TAINTS EXAMPLE

```
$ kubectl taint nodes node1 key=value:NoSchedule
```

La taint a la clé **key**, valeur **value**, et l'effet **NoSchedule**.

Il y a 3 effets possibles pour **Taints**:

- **NoExecute**: les pods sans tolérance seront expulsés et ne seront pas programmés sur ce nœud
- **NoSchedule**: les pods sans tolérance ne seront pas programmés sur ce nœud
- **PreferNoSchedule**: le scheduler essaiera de ne pas planifier les pods sans tolérance sur ce nœud



TOLERATION EXAMPLE

Les tolérances correspondantes peuvent ensuite être ajoutées aux pods:

```
tolerations:  
  - key: "key"  
    operator: "Equal"  
    value: "value"  
    effect: "NoExecute"  
    tolerationSeconds: 3600
```

Le "TolerationSeconds" supplémentaire est utilisé pour retarder l'expulsion du Pod



TP 8.2 : Taints & Toleration

GESTION DES RESSOURCES



- Quand vous créez un Pod vous pouvez positionner :
 - La quantité de CPU et de mémoire dont chacun de ses conteneurs a besoin (**Requests**)
 - Une limite max sur ces 2 mêmes compteurs qu'il lui sera impossible de dépasser (**Limits**)
- Les ressources consommées par un **Pod** sont les sommes des ressources consommées par ses conteneurs



EXEMPLE

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: frontend  
spec:  
  containers:  
    - name: app  
      image: super.mycompany.com/app:v4  
      resources:  
        requests:  
          memory: "256Mi"  
          cpu: "500m"  
        limits:  
          memory: "2Gi"  
          cpu: "2"
```

SCHEDULE ET REQUESTS



- Quand vous créez un **Pod**, le **scheduler** sélectionne un noeud pour l'héberger
- Chaque noeud a une capacité max (CPU et Mémoire)
- Le **scheduler** s'assure que la somme des **Requests** des Pods sur un noeud ne dépasse pas la capacité du noeud
- Note : ce n'est donc pas la consommation réelle en CPU/Mémoire du noeud au moment du scheduling qui fait foi mais bien la somme des **Requests** des Pods du noeud

GESTION DES LIMITS POUR LA MÉMOIRE



- Un conteneur peut consommer plus de mémoire que la **request** effectuée si le noeud qui l'héberge a de la mémoire disponible
- Un conteneur ne peut pas consommer plus que la **limit** paramétrée
- Si un conteneur alloue plus de mémoire que la **limit** le conteneur va être marqué comme un candidat pour être terminé. S'il continue à consommer plus de mémoire que la valeur du **limit** alors il sera **terminé**.
 - S'il est **redémarrable**, le **kubelet** essaiera de le redémarrer (c'est un cas d'erreur comme les autres)

Voir l'exemple **workspace/others/pod--try-to-allocate-too-much-memory.yml**



GESTION DES LIMITS POUR LE CPU

- Un conteneur peut consommer plus de cpu que la **request** effectuée si le noeud qui l'héberge a des ressources cpu disponibles
- Un conteneur ne peut pas consommer plus que la **limit** positionnée

GESTION DES LIMITS POUR LE CPU : EXEMPLE



```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: cpu-demo  
spec:  
  containers:  
    - name: cpu-demo-ctr  
      image: zenika/k8s-training-stress:v2  
      resources:  
        limits:  
          cpu: "1"  
        requests:  
          cpu: "0.5"  
      args:  
        - --vm  
        - "2"
```



UNITÉ DE CPU (1/2)

- La ressource CPU est mesurée en **cpu units**
- 1 unité de cpu vaut :
 - 1 AWS vCPU
 - 1 GCP Core
 - 1 Azure vCore
 - 1 Hyperthread sur un processeur Intel d'un serveur **bare-metal** avec Hyperthreading



UNITÉ DE CPU (2/2)

- Un conteneur qui demande 0.5 cpu est garanti d'avoir la moitié de cpu d'un conteneur qui demande 1 cpu
- Il est possible d'utiliser le suffixe **m** qui signifie **milli**
 - 100m cpu, 100 millicpu, et 0.1 cpu représentent la même valeur
- La valeur d'une unité CPU est une valeur absolue : 0.1 correspond à la même quantité de CPU sur un **single-core**, **dual-core**, ou une machine **48-core**



CONSOMMATION PAR NOEUD (1/3)

```
$ kubectl describe nodes e2e-test-minion-group-4lw4
Name:           e2e-test-minion-group-4lw4
Capacity:
  cpu:            2
  memory:        7679792Ki
  pods:          110
Allocatable:
  cpu:           1800m
  memory:       7474992Ki
  pods:          110
...

```



CONSOMMATION PAR NOEUD (2/3)

...

Non-terminated Pods:

Namespace	Name	(5 in total)	CPU Requests	CPU Limits	Memory Requests	Memory Limits
kube-system	fluentd-g...	100m (5%)	0 (0%)	200Mi (2%)	200Mi (2%)	
kube-system	kube-dns-...	260m (13%)	0 (0%)	100Mi (1%)	170Mi (2%)	
kube-system	kube-prox...	100m (5%)	0 (0%)	0 (0%)	0 (0%)	
kube-system	monitorin...	200m (10%)	200m (10%)	600Mi (8%)	600Mi (8%)	
kube-system	node-prob...	20m (1%)	200m (10%)	20Mi (0%)	100Mi (1%)	

...



CONSOMMATION PAR NOEUD (3/3)

Allocated resources:

(Total limits may be over 100 percent, i.e., overcommitted.)			
CPU Requests	CPU Limits	Memory Requests	Memory Limits
680m (34%)	400m (20%)	920Mi (12%)	1070Mi (14%)



LIMIT RANGES (1/2)

- Il est possible de configurer par **namespace** :
 - des valeurs par défaut pour les **requests** et les **limits**
 - des valeurs max pour les **requests** et les **limits**



LIMIT RANGES (2/2)

```
---  
apiVersion: v1  
kind: LimitRange  
metadata:  
  name: mem-limit-range  
spec:  
  limits:  
    - default:  
        memory: 512Mi  
    defaultRequest:  
        memory: 256Mi  
    max:  
        memory: 1Gi  
    min:  
        memory: 500Mi  
  type: Container
```

AUTO-SCALING



- L'auto-scaling horizontal de Pod est la mise à l'échelle automatique du nombre de replicas d'un controller
- Il est effectué par un controller appelé **autoscaler** qui est piloté par une ressource de type **HorizontalPodAutoscaler**
- Le controller vérifie régulièrement (**f=30s** par défaut) les métriques des Pods concernés et calcule le nombre de replicas par rapport aux critères d'auto-scaling paramétrés pour la ressource cible (Deployment, ReplicaSet)
- Les métriques sont soit :
 - les ressources cpu/mémoire
 - des métriques personnalisées



CONFIGURATION

```
$ kubectl autoscale deployment autosc --cpu-percent=30 --min=1 --max=5  
deployment "autosc" autoscaled
```

```
$ kubectl get hpa  
NAME      REFERENCE   TARGETS   MINPODS   MAXPODS   REPLICAS   AGE  
autosc    Deployment/autosc <unknown> / 30%     1          5          0          28s
```



DESCRIPTEUR AUTOSCALING/V1

En version **autoscaling/v1**, support uniquement du cpu :

```
---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: autosc
spec:
  maxReplicas: 5
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: autosc
  targetCPUUtilizationPercentage: 30
```



DESCRIPTEUR

En version **autoscaling/v2beta1** (k8s >= 1.6), support cpu+mémoire et des métriques personnalisées :

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef: { apiVersion: apps/v1, kind: Deployment, name: php-apache, minReplicas: 1,
maxReplicas: 10 }
  metrics:
    - type: Resource
      resource: { name: cpu, targetAverageUtilization: 50 }
    - type: Pods
      pods: { metricName: packets-per-second, targetAverageValue: 1k }
    - type: Object
      object:
        metricName: requests-per-second
        target: { apiVersion: networking.k8s.io/v1, kind: Ingress, name: main-route }
        targetValue: 10k
```





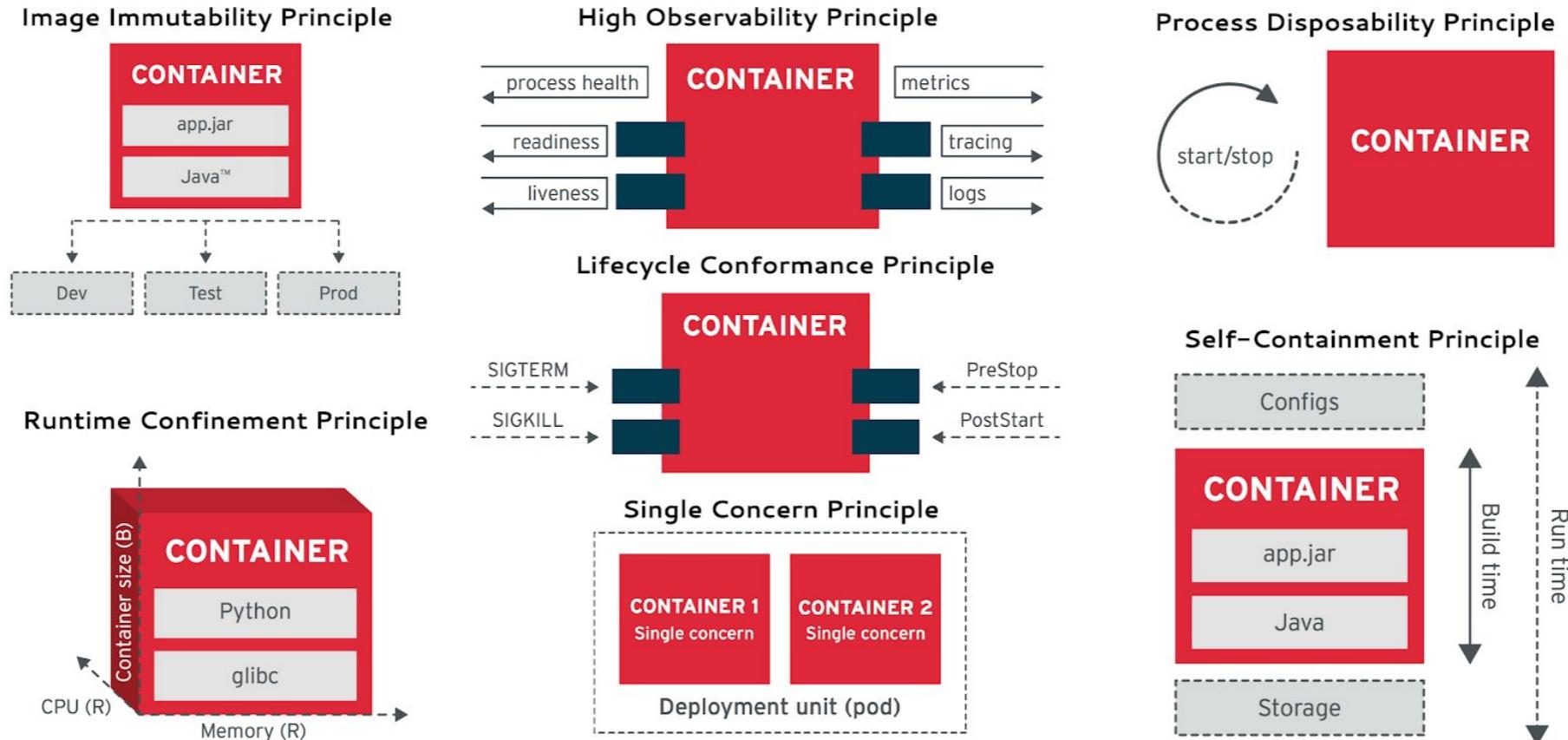
DÉVELOPPEMENT D'APPLICATIONS COMPATIBLES

AGENDA



- Applications Compatibles Kubernetes
- Helm pour simplifier vos déploiements
- Kustomize pour gérer vos environnements

APPLICATIONS COMPATIBLES KUBERNETES



<https://medium.com/@bibryam/cloud-native-container-design-principles-144ceaa98dba>

CAGIP GOLDEN RULES



- | | | | |
|---|---|----|---|
| 1 | La plateforme est conçue pour accueillir des applications stateless uniquement . Le non respect de cette règle casse l'agilité de la plateforme | 6 | Faire une seule et unique image pour tous les environnements |
| 2 | Pour le cas d'une application statefull, si un système de réPLICATION des données est nativement mis en œuvre une exception dans cet unique cas sera tolérée | 7 | Les configurations liées à l'environnement doivent se faire soit par variable d'environnement , soit à l'aide d'un key value store , soit par secret |
| 3 | Les déploiements doivent se faire à minima sous forme de service avec au minimum 2 répliques par site | 8 | La chaîne applicative doit embarquer son propre reverse-proxy |
| 4 | Chaque conteneur doit être "cappé" en cpu et ram et doit résERVER une partie de la mémoire. | 9 | Les logs doivent être redirigés sur la sortie standard (accessible via docker logs) ou vers une plateforme externe |
| 5 | Les images déployées en production ne doivent pas utiliser le tag "latest" , idéalement un numéro de version x.y.z suivi de l'identifiant git (ex: 1.0.1-dbc2d5f) | 10 | Pas de conteneur en mode privilégié |

1

bullet point vert = contrôle automatique de la règle pouvant bloquer le déploiement

GESTION DES DESCRIPTEURS DE DÉPLOIEMENT EN ÉQUIPE



- Installer des ressources Kubernetes devrait être aussi facile que d'utiliser `yum` ou `apt`
- Des équipes différentes devraient pouvoir facilement collaborer
- Les `Releases` devraient être reproductibles
- Les `packages` devraient être facilement partageables
- Plusieurs outils vont faciliter cette approche : Helm, Kustomize...



- **Helm** est un gestionnaire de packages Kubernetes
- **Helm** package plusieurs ressources Kubernetes dans un seul objet de déploiement logique appelé **chart**
- **Helm** propose une gestion de dépendances
- **Helm** simplifie la gestion des ressources Kubernetes en s'appuyant sur un moteur de template
- ... et en proposant un mécanisme de gestion de version au niveau de l'application multi-composants

👍 **Helm**

CHART HELM



- Un **Chart** est un package, un ensemble de ressources Kubernetes (Deployments, Configmaps, Services, Ingress, ...)
- Une **Release** est une instance de **Chart** qui a été déployée dans un cluster Kubernetes
- Un **Dépôt** est une collection de **Charts**
- Un **Template** est un modèle de ressource Kubernetes s'appuyant sur le moteur de template **Golang** et la bibliothèque **Sprig**



ARBORESCENCE D'UN CHART

```
|- Chart.yaml
|- values.yaml
|- charts
|   '-- redis-10.2.1.tgz
`-- templates
    |-- NOTES.txt
    |-- _helpers.tpl
    |-- hasher.yaml
    |-- rng.yaml
    |-- serviceaccount.yaml
    |-- webui.yaml
    '-- worker.yaml
```



CHART.YAML

```
apiVersion: v2
name: dockercoins
description: A Helm chart for dockercoins application
home: https://github.com/jpetazzo/orchestration-workshop/tree/master/dockercoins
# A chart can be either an 'application' or a 'library' chart.
type: application
# This is the chart version. This version number should be incremented each time you make
changes
# to the chart and its templates, including the app version.
version: 0.1.0
# This is the version number of the application being deployed. This version number should be
# incremented each time you make changes to the application.
appVersion: 1.0.0
keywords:
- dockercoins
maintainers:
- name: Zenika
  url: https://www.zenika.com
dependencies:
- name: redis
  version: ~10.2.0
  repository: https://kubernetes-charts.storage.googleapis.com
```



VALUES.YAML

```
# Default values for dockercoins.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.
replicaCount:
  hasher: 1
  worker: 1
image:
  default:
    version: "1.0"
  rng:
    repository: training/dockercoins-rng
  hasher:
    repository: training/dockercoins-hasher
  worker:
    repository: training/dockercoins-worker
  webui:
    repository: training/dockercoins-webui
  pullPolicy: IfNotPresent
ingress:
  webui:
    host: dockercoins.127.0.0.1.nip.io
```



EXEMPLE DE TEMPLATE

```
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: {{ .Values.application.name }}  
  labels:  
    {{- include "dockerooms.labels" . | nindent 4 }}  
    component: rng  
spec:  
  type: {{ default .Values.service.default.type .Values.service.rng.type }}  
  ports:  
  - port: {{ default .Values.service.default.port .Values.service.rng.port }}  
    targetPort: http  
    protocol: TCP  
    name: http  
  selector:  
    {{- include "dockerooms.selectorLabels" . | nindent 4 }}  
    component: rng
```

ARTIFACTHUB



Annuaire de charts de différents dépôts <https://artifacthub.io/>

Artifact HUB BETA Search packages ?

1 - 20 of 2672 results Show: 20 ▾

SIGN UP SIGN IN ⚙

FILTERS

- Official repositories ⓘ
- Verified publishers ⓘ

KIND

- Helm charts (2206)
- OLM operators (166)
- Krew kubectl plugins (137)
- Tekton tasks (110)
- Helm plugins (27)
- Falco rules (23)
- Tinkerbell actions (2)
- OPA policies (1)

CATEGORY

- Database
- Integration and Delivery
- Logging and Tracing
- Machine learning
- Monitoring
- Networking
- Security
- Storage
- Streaming and Messaging
- Web applications

PUBLISHER

No publisher selected

REPOSITORY

cube-prometheus-stack

ORG: Prometheus REPO: prometheus-community
VERSION: 13.7.2 APP VERSION: 0.45.0

kube-prometheus-stack collects Kubernetes manifests, Grafana dashboards, and Prometheus rules combined...

Official Verified Publisher

prometheus

ORG: Prometheus REPO: prometheus-community
VERSION: 13.3.2 APP VERSION: 2.24.0

Prometheus is a monitoring system and time series database.

Official Verified Publisher

artifact-hub

ORG: Artifact Hub REPO: Artifact Hub
VERSION: 0.14.0 APP VERSION: 0.14.0 LICENSE: Apache-2.0 ⓘ

Artifact Hub is a web-based application that enables finding, installing, and publishing Kubernetes packages.

Official Verified Publisher Images Security Rating

datadog

ORG: Datadog REPO: Datadog
VERSION: 2.8.3 APP VERSION: 7

Datadog Agent

Official Verified Publisher

sumologic

ORG: Sumo Logic REPO: sumologic
VERSION: 2.0.1 APP VERSION: 2.0.1

A Helm chart for collecting Kubernetes logs, metrics, traces and events into Sumo Logic.

Official Verified Publisher

kubeview

USER: benc-uk REPO: Kubeview (Official)
VERSION: 0.1.20 APP VERSION: 0.1.20

Kubernetes cluster visualiser and visual explorer

Official Verified Publisher

renovate

ORG: Renovate Bot REPO: Helm Charts
VERSION: 24.44.2 APP VERSION: 24.44.2 LICENSE: AGPL-3.0-only

Universal dependency update tool that fits into your workflows.

Official Verified Publisher Images Security Rating

prometheus-adapter

ORG: Prometheus REPO: prometheus-community
VERSION: 2.12.0 APP VERSION: v0.8.3

A Helm chart for k8s prometheus adapter

Official Verified Publisher



KUSTOMIZE POUR GÉRER VOS ENVIRONNEMENTS



Lorsque vous devez déployer et gérer une application sur plusieurs environnements, il y aura souvent des spécificités qui ne peuvent être gérées par les **ConfigMap** seules :

- les **Host** pour les **Ingress**
- les ressources matérielles affectées aux **Pods**
- des métadonnées (labels, annotations, ...) communes à vos ressources

KUSTOMIZE ?



Le projet **Kustomize** permet de faire de la composition entre vos descripteurs.

- Chargement de descripteurs classiques
- Crédation de resources **ConfigMap** et **Secret**
- Application de transformations
- Variations (surcharge uniquement) depuis une (ou plusieurs) base via des overlays

Concepts fondamentaux :

- Pas de templatisation
- Pas de variabilisation
- Définition statique de la configuration : ce qui est déployé est le reflet exact de ce qui est décrit

GÉNÉRATEURS KUSTOMIZE



Kustomize propose par défaut 2 générateurs :

- `configMapGenerator`
- `secretGenerator`

Permet de créer des `ConfigMap` et `Secret` :

- depuis des entrés clé=valeur
- depuis des fichiers (clé = nom, valeur = contenu)
- depuis le contenu d'un fichier de variables d'environnement (une ligne = un couple clé=valeur)

TRANSFORMATEURS KUSTOMIZE



Kustomize propose plusieurs transformateurs :

- `commonAnnotations`, `commonLabels` : définition globale des annotations et labels (et sélecteurs)
- `namespace` : définition globale du namespace
- `namePrefix`, `nameSuffix` : modification des noms des resources
- `images` : remplacement de la référence d'une image
- `patches` : applique des modifications issues de "patch" (format `JSON 6902` ou merge de fichier)

Il est possible d'étendre Kustomize pour ajouter ses propres transformateurs.



STRUCTURE DE KUSTOMIZATION

```
~/someApp
└── base
    ├── deployment.yaml
    ├── kustomization.yaml
    └── service.yaml
└── overlays
    ├── development
    │   └── kustomization.yaml
    │   └── ingress.yaml
    └── production
        ├── kustomization.yaml
        └── memory-limit.yaml
        └── ingress.yaml
```

EXEMPLES DE KUSTOMIZATION



base/kustomization.yaml

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

nameSuffix: -v1

resources:
  - service.yaml
  - deployment.yaml
```



EXEMPLES D'OVERLAY

overlays/production/kustomization.yaml

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

namespace: prod

resources:
  - ../../base
  - ingress.yaml

patches:
  - path: memory-limit.yaml

configMapGenerator:
  - name: my-config
    literals:
      - ENVIRONMENT=prod
```



EXEMPLES DE PATCH DE MERGE

overlays/production/memory-limit.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
        - name: my-app
          resources:
            limits:
              memory: 256Mi
```

CONTRAINTES ET PRATIQUES LIÉES À KUSTOMIZE



De par les contraintes de non templatisation et non variabilisation, Kustomize encourage certaines pratiques :

- Déclaration à l'avance des spécificités d'un environnement dans un overlay spécifique (idéalement versionné)
- Composition entre bases et overlays potentiellement à plusieurs niveaux
- Définition de **components** pour plus librement composer entre plusieurs configurations



MODIFICATIONS LOCALES

Kustomize offre la possibilité de modifier localement le fichier **kustomization.yaml**, ceci afin d'adapter certains éléments au contexte de déploiement, par exemple dans le cadre du processus d'intégration continue.

Exemples :

- Modification d'une image

```
$ kustomize edit set image \
my.registry.com/myimage=my.registry.com/myimage:${TAG_VERSION}
```

- Ajout d'une annotation

```
$ kustomize edit add annotation deploy-build-id:${PIPELINE_ID}
```



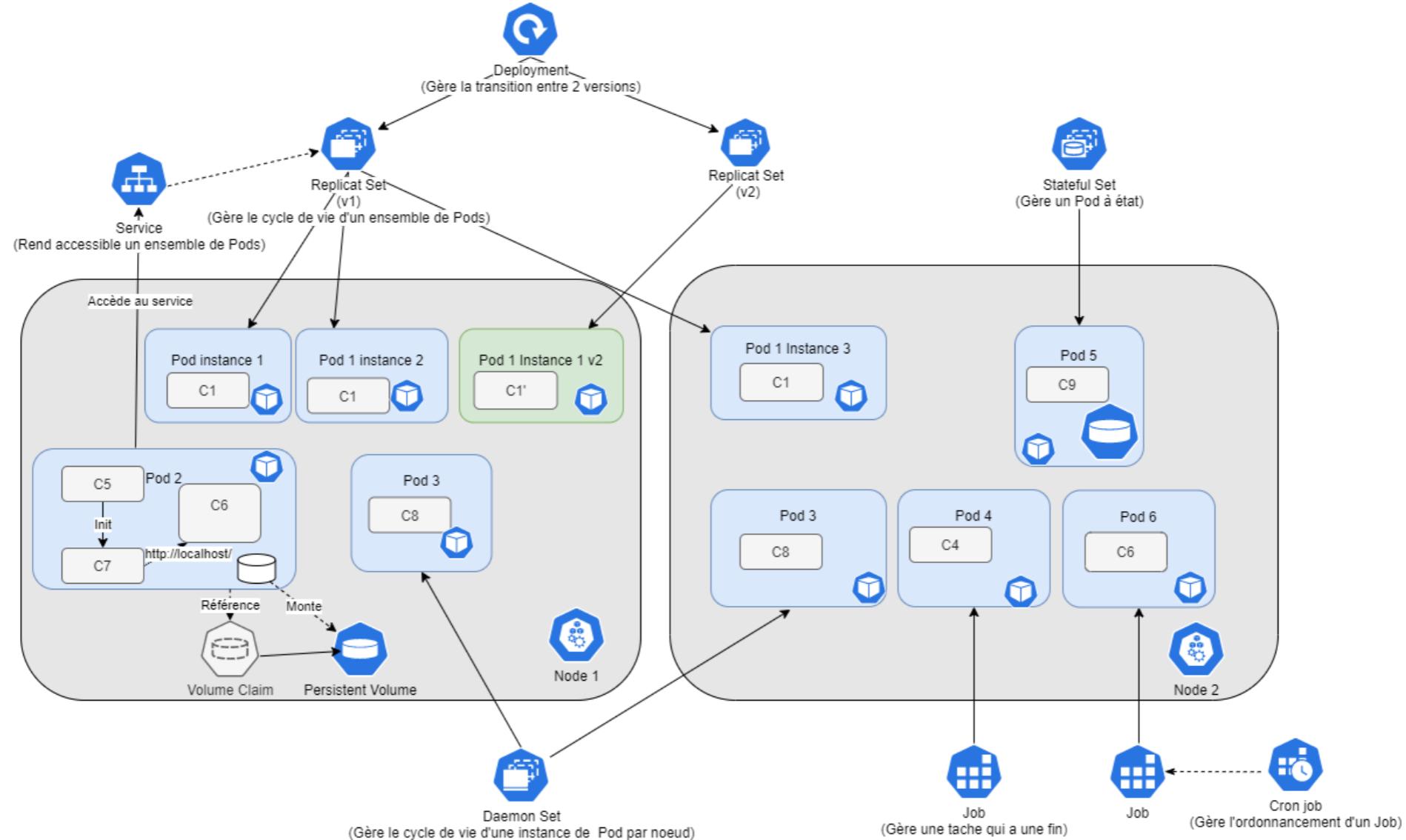
TP 9 : Kustomize





CONCLUSION

RÉSUMÉ DES RESSOURCES ABORDÉES



BONNES PRATIQUES POUR VOS DESCRIPTEURS



- Utilisez des **Deployments**
- Configurez les sondes **Startup**, **Readiness** et **Liveness**
- Utilisez des **Init Containers**
- Réfléchissez 2 fois avant de faire des **Pods** multi-conteneurs
- Précisez le **ImagePullPolicy**
- Spécifiez les ressources ! (**Requests** et **Limits**)
- Utilisez les **labels** et les **annotations**
- Utilisez Kubernetes le plus tôt possible !
- Vos descripteurs sont aussi du code, historisez/versionnez les ! (mode déclaratif)

KUBERNETES, ET APRÈS ?



Kubernetes seul ne répond pas à tous les besoins...

- Un ServiceMesh pour gérer le traffic shaping ([Linkerd](#), [Istio](#), ...)
- [CertManager](#) pour gérer vos certificats HTTPS
- [Vault](#) pour chiffrer vos secrets
- [ArgoCD](#), [Flux](#) pour gérer votre workflow GitOps
- ...

POUR ALLER PLUS LOIN...



Formation Kubernetes Admin

- Comprendre l'architecture et le fonctionnement interne d'une plateforme Kubernetes
- Savoir installer et opérer un cluster Kubernetes
- Avoir toutes les clés pour bien choisir les différents outils liés à la maintenance d'un cluster (réseau, logs, métriques)
- Savoir configurer et faire le suivi opérationnel d'un cluster Kubernetes