

Introduction à Kubernetes

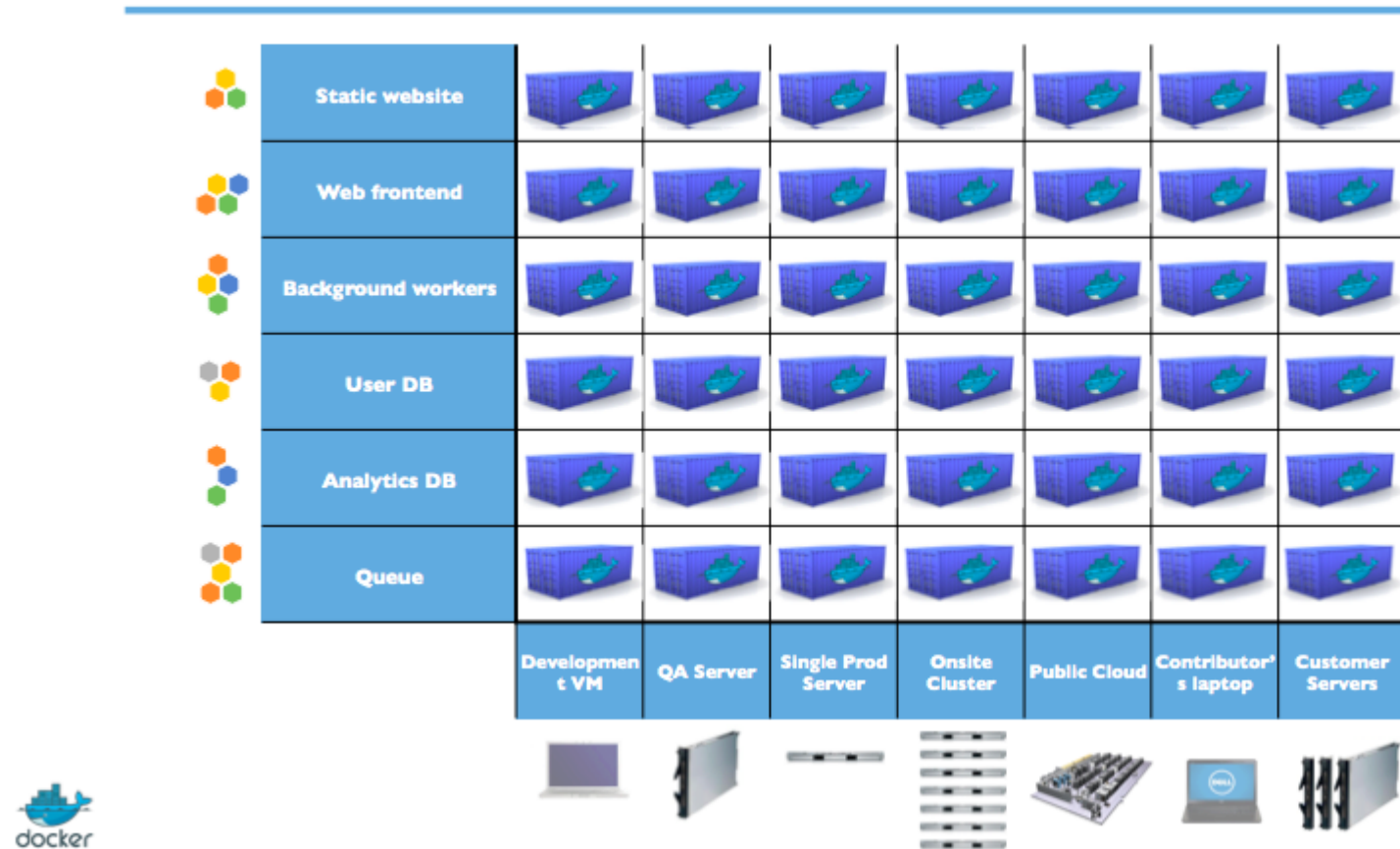
/me

Christophe Furmaniak :

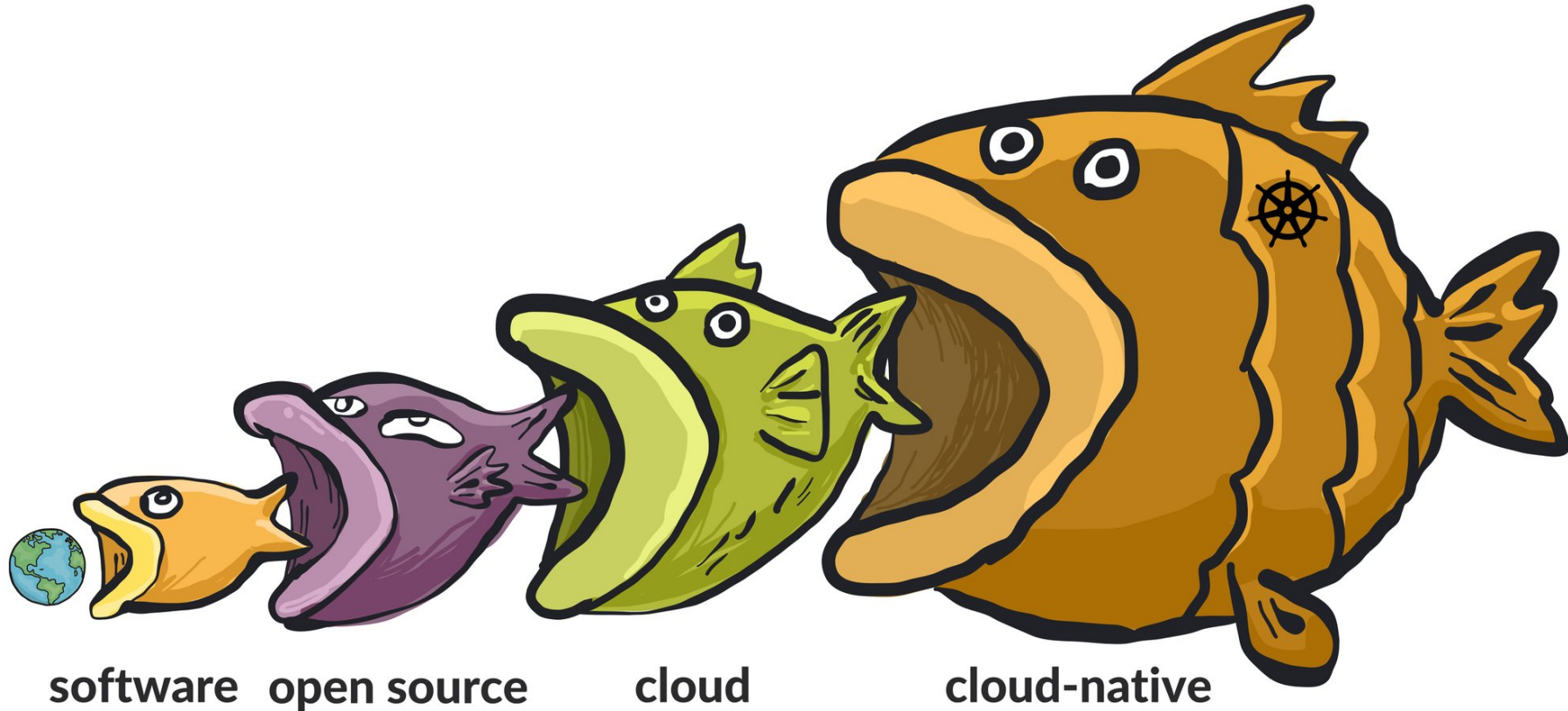
- Twitter : [@cfurmaniak](#)
- Github : [looztra](#)
- Docker hub store : [looztra](#)
- Consultant épanoui chez [Zenika](#)

Kubernetes?

L'approche containers / docker



Cloud native?



Où utiliser les containers / 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!)

Containers sans orchestration

- Comment démarrer les containers?
- Comment exposer vos services containerisés
- Comment gérer les containers 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?

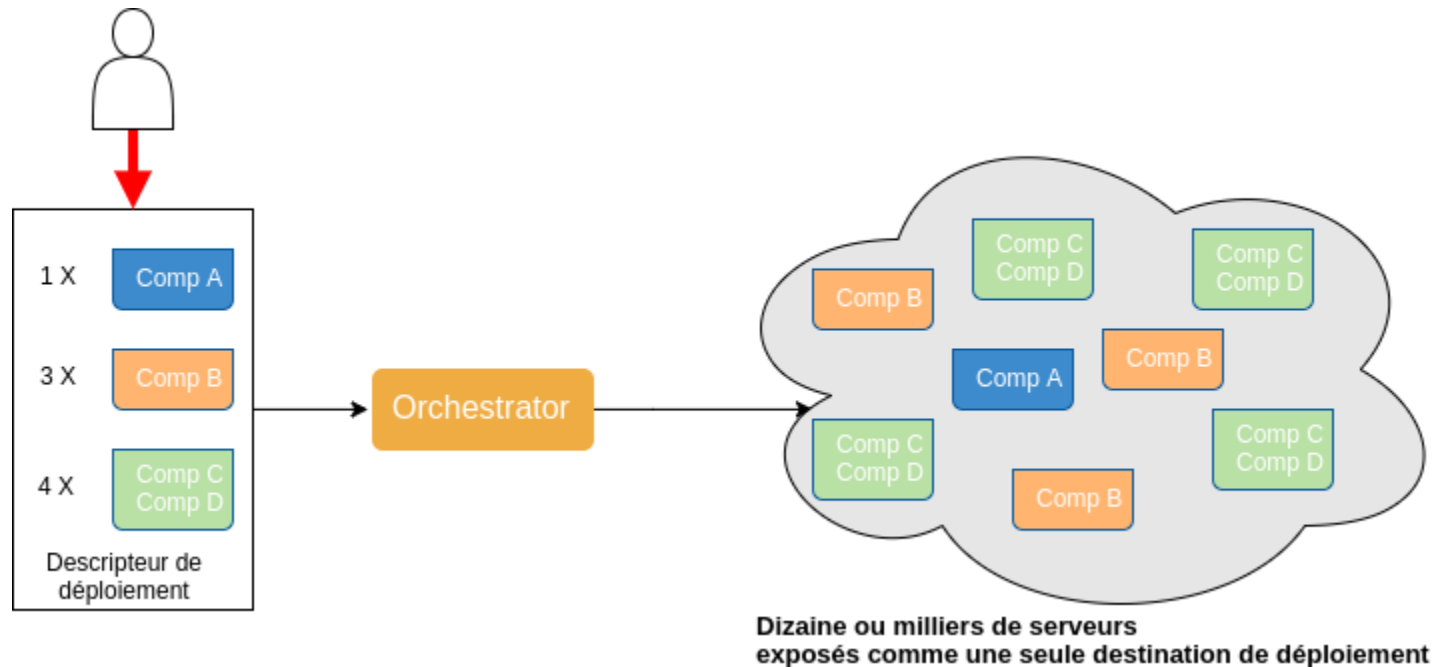
Comment partager vos process de déploiement

 Comment faire pour que les développeurs/testeurs/opérateurs puissent être autonomes?

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 containerisés

L'orchestration vue par l'utilisateur



Placement et gestion du cycle de vie des containers

☐ 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 containers
- 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 d'**Overlay Networks** qui permettent à vos containers 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

Des Orchestrateurs

- Kubernetes
- Docker Swarm
- Mesos + Marathon
- Rancher
- Nomad
- Titus (Mesos + Mantis [scheduling/job mgmt] + Titan)
- Mantl (Mesos + Marathon and Kubernetes)
- Openshift V3 (Kubernetes)
- CloudFoundry
- ...

Kubernetes



Comment ça se prononce?

How Do You Pronounce Kubernetes? 

Introduction à Kubernetes

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

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 **“ubernete”** par un **“8”**

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

- kubeadm
- Apprenda Kismatic Enterprise Toolkit, CoreOS Tectonic
- GiantSwarm
- Azure Container Service
- Rancher 1.x (orchestration engine option) and Rancher 2.x (native)
- RedHat OpenShift v3 (Origin et Container Platform)
- Docker EE, Docker4mac, Docker4win
- kops (Kubernetes Operations) and kubicorn
- playbooks (kubespray et kubernetes/contrib) et modules Ansible
- templates et provider Terraform
- minikube

Kubernetes et docker

- Docker en version v1.12 est recommandé
- Les versions v1.11, v1.13 and 17.03 sont connues pour fonctionner correctement
- Les version 17.06+ devraient fonctionner mais n'ont pas été testées et vérifiées par l'équipe Kubernetes

Kubernetes sans docker

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

- `rkt` de CoreOS ([Running Kubernetes with rkt](#))
- `CRI-O` est un projet en incubation qui s'appuie sur la Container Runtime Interface de l'Open Container Initiative ([Six reasons why cri-o is the best runtime for k8s](#))

Versions de Kubernetes

- **v1.8.0** le 29/09/17
 - **v1.7.0** le 30/06/17
 - v1.6.0 le 28/03/17
 - v1.5.0 le 13/12/16
 - v1.4.0 le 26/09/16
 - ...
 - v1.0.0 le 13/07/15
-
- **Tous les Changelogs**
 - **Toutes les Releases**

Premiers pas avec Kubernetes

Une instance Kubernetes locale

- Minikube est un outil qui vous permet de lancer facilement une instance Kubernetes locale
- Minikube lance un cluster Kubernetes composé d'un seul noeud
- ... dans une VM sur votre poste
- ... ce qui permet d'utiliser Kubernetes dès l'environnement **Poste du Développeur** (ou du **testeur**...ou de **l'opérateur!**)

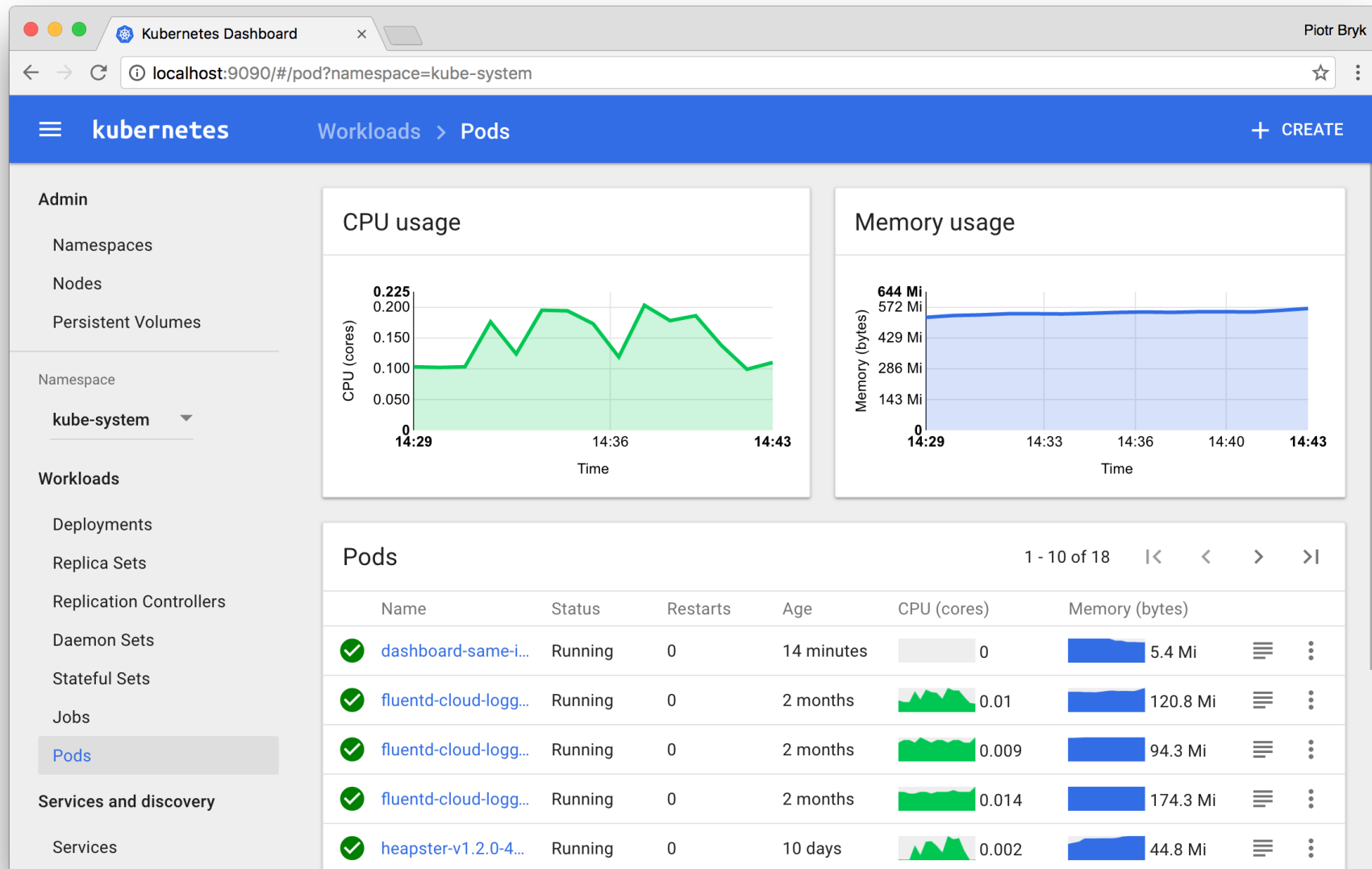
Site de Minikube

DEMO : Minikube


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

Dashboard



Ligne de commande kubectl

- `kubectl` permet d'interagir en ligne de commande avec vos instances *Kubernetes*
- La documentation en ligne est disponible [ici](#) 

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)

API Kubernetes et Versioning d'API

- Afin de faciliter les évolutions telles que les ajouts/suppression de champs ou les restructurations, Kubernetes supporte plusieurs versions d'API
- Chaque version correspond à un chemin différent, ex: `/api/v1` ou `/apis/extensions/v1beta1`
- Différentes versions impliquent différents niveaux de stabilité

[API Overview Reference](#) 

Fonctionnalités en version Alpha, Beta, Stable

- **Alpha** :

- 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

- **Beta** :

- 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

- **Stable** :

- Les versions stable des fonctionnalités sont pérennes

Types de ressources valides (1/2)

Ressource	Ressource
<i>all</i>	horizontalpodautoscalers (aka 'hpa')
certificatesigningrequests (aka 'csr')	<i>ingresses</i> (aka ' <i>ing</i> ') <i>jobs</i>
clusterrolebindings	limitranges (aka 'limits')
clusterroles	<i>namespaces</i> (aka ' <i>ns</i> ') networkpolicies (aka 'netpol')
clusters (valid only for federation apiservers)	<i>nodes</i> (aka ' <i>no</i> ') <i>persistentvolumeclaims</i> (aka ' <i>pvc</i> ') componentstatuses (aka 'cs')
<i>configmaps</i> (aka ' <i>cm</i> ') controllerrevisions	

Types de ressources valides (2/2)

Ressource	Ressource
<i>replicasets</i> (aka ' <i>rs</i> ') <i>replicationcontrollers</i> (aka ' <i>rc</i> ') resourcequotas (aka 'quota') rolebindings roles <i>secrets</i> serviceaccounts (aka 'sa') <i>services</i> (aka ' <i>svc</i> ') <i>statefulsets</i> storageclasses	customresourcedefinition (aka 'crd') <i>daemonsets</i> (aka ' <i>ds</i> ') <i>deployments</i> (aka ' <i>deploy</i> ') endpoints (aka 'ep') events (aka 'ev') poddisruptionbudgets (aka 'pdb') podpreset <i>Pods</i> (aka ' <i>po</i> ') podsecuritypolicies (aka 'psp') podtemplates

Pods

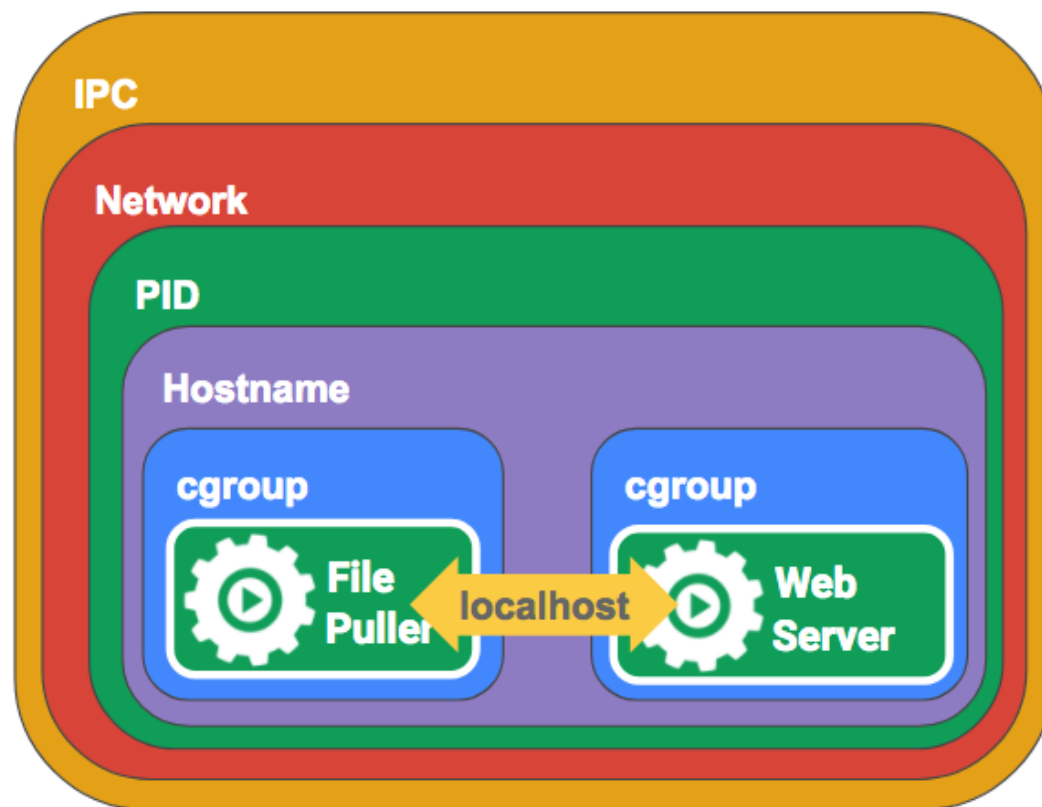
Modèle/Concept du pod

- Un **pod** (une cosse, une gousse, une coque) est un groupe d'un ou plusieurs containers (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

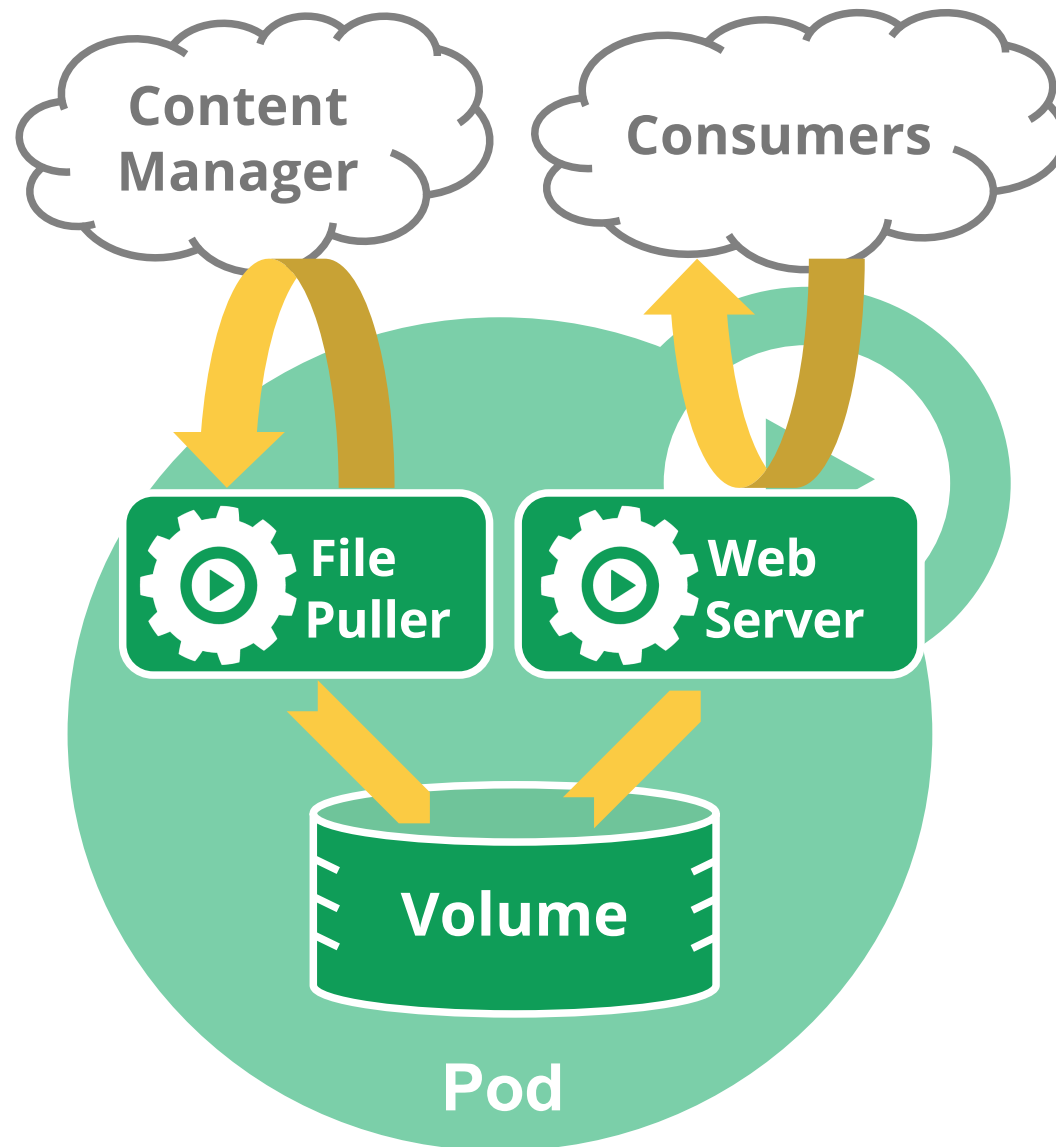
Partage de ressources au sein d'un pod

- Les containers d'un pod partagent les mêmes ressources : espace de pids (* Kubernetes >= 1.7, Docker >=1.13), file system, réseaux, IPC (Inter Process Communication)
- Les containers d'un pod sont toujours co-localisés et co-gérés
- Les containers 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 containers au sein d'un pod peuvent partager le même IPC et peuvent donc communiquer en utilisant les sémaphores **SystemV** ou la mémoire partagée **POSIX**
- Les applications au sein d'un pod peuvent accéder aux mêmes volumes

Pod multi-container partageant les mêmes ressources



Pod multi-container partageant un même volume



Communication réseau entre les pods d'un cluster

- Tous les pods d'un cluster Kubernetes possèdent une IP dans un unique réseau partagé entre tous les noeuds du cluster
- Chaque pod peut donc communiquer avec les autres pods en utilisant son IP

Exemple de descripteur au format json

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "sise",
    "labels": {
      "app": "sise"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "sise",
        "image": "mhausenblas/simpleservice:0.5.0",
        "ports": [
          {
            "containerPort": 9876
          }
        ]
      }
    ],
    "restartPolicy": "Always",
  }
}
```

Exemple de descripteur au format yaml

```
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: sise
    name: sise
spec:
  containers:
  - image: mhausenblas/simple-service:0.5.0
    imagePullPolicy: IfNotPresent
    name: sise
    ports:
    - containerPort: 9876
  restartPolicy: Always
```

Utiliser 'kubectl explain' pour accéder à la documentation des ressources

```
L> 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://](https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-and-status/)

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/>

apiVersion <string>

APIVersion defines the versioned schema of this representation of an object. Servers should convert recognized schemas to the latest internal value, and may reject unrecognized values. More info:

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

Utiliser les paths json pour aller plus loin

```
L> 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)** are expanded using the container's environment. If a variable cannot be resolved, the reference **in** the input string will be unchanged. The **\$(VAR_NAME)** syntax can be escaped with a double \$\$, ie: **\$\$\$(VAR_NAME)**. Escaped references will never be expanded, regardless of whether the variable exists or not. Cannot be updated. More info: <https://kubernetes.io/docs/tasks/inject-data-application/define-command-argument-container/#running-a-command-in-a-shell/>

Créer un pod à partir d'un fichier de description

Pour créer un pod (vrai pour les autres types de ressources), utiliser la commande `kubectl create`

```
L> kubectl create -f pod-from-file.yml  
pod "k8s-rulez" created
```

```
L> kubectl get po k8s-rulez
```

NAME	READY	STATUS	RESTARTS	AGE
k8s-rulez	1/1	Running	0	11s

Plusieurs ressources dans un seul fichier

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
```


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`

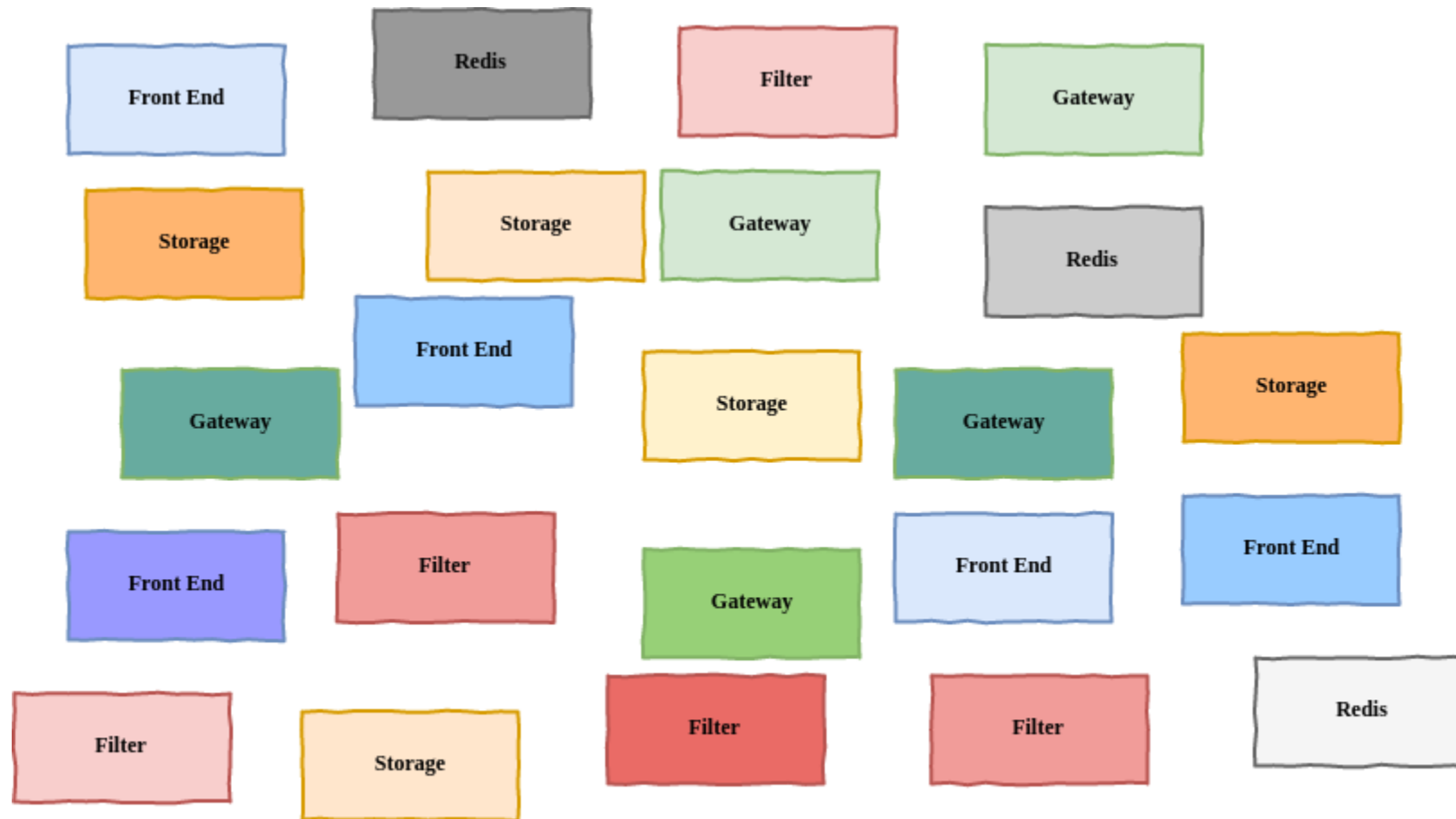
Anatomie d'un fichier descripteur

- Le descripteur peut se décomposer en plusieurs sous-parties:
 - `apiVersion`
 - `kind`
 - `metadata`
 - `spec`
 - `status`
- Ces sous-parties se retrouvent dans tous les types de ressource
- Certaines informations sont en mode `readonly` uniquement (notamment tout ce qui se trouve dans `status`)

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?

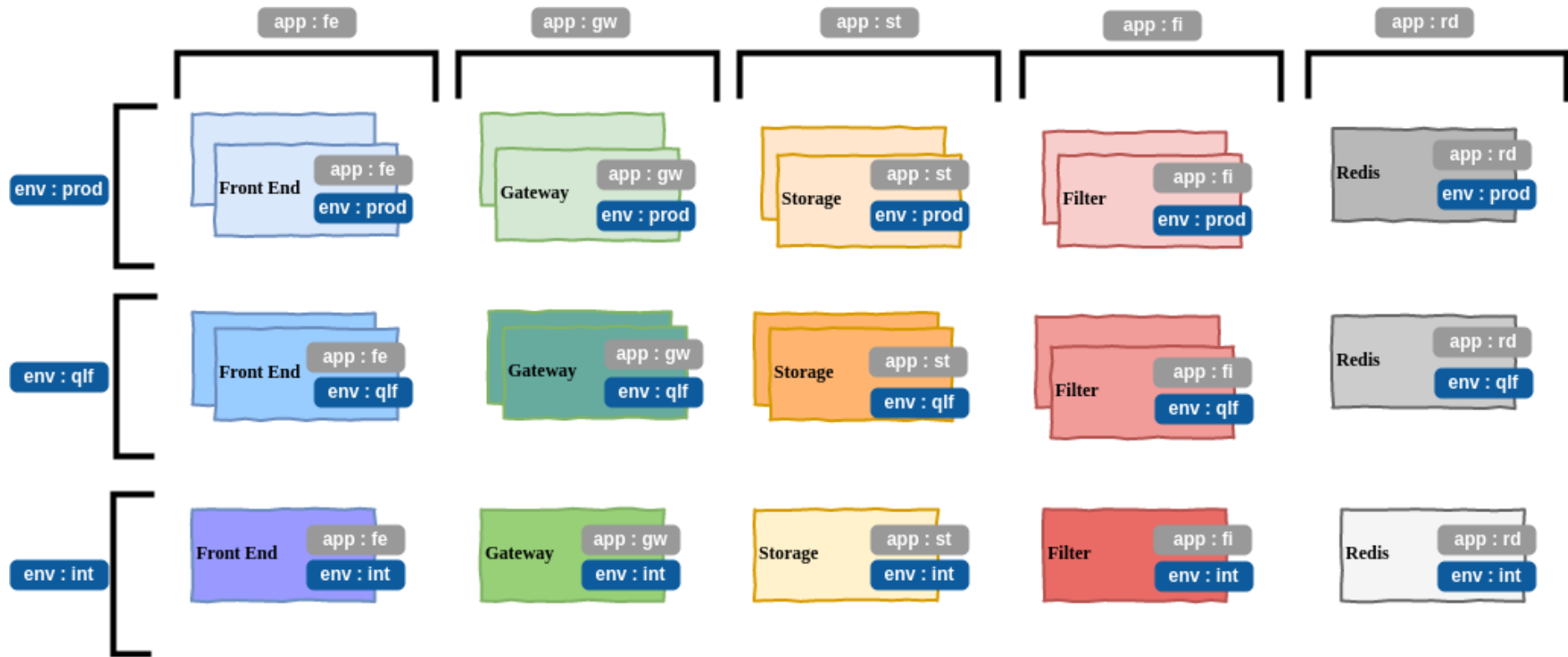


Pods non catégorisés dans une architecture avec beaucoup de (petits) services

Organiser avec des labels

- Les **labels** peuvent être utilisés pour organiser et sélectionner des sous-ensembles d'objets
- 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



Pods catégorisés par composant et environnement

Syntaxe et caractères autorisés pour les labels

- Les **Labels** sont des couples clé/valeur
- Clé valide : 2 sous-parties, 1 préfix 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

Sélectionner les pods en fonction de leurs labels (1/2)

- Lister les pods pour lesquels **env** vaut exactement **int**

```
L> kubectl get po -l env=int
```

NAME	READY	STATUS	RESTARTS	AGE
k8s-rulez	1/1	Running	0	2h

- Lister les pods pour lesquels **env** est différent de **int**

```
L> kubectl get po -l env!=int
```

NAME	READY	STATUS	RESTARTS	AGE
centos-shell	1/1	Running	0	22h
sise-086vd	1/1	Running	0	22h
yaml-pod	2/2	Running	0	6h

Sélectionner les pods en fonction de leurs labels (2/2)

- Lister les pods pour lesquels le label avec comme clé **env** est positionné

```
L> kubectl get po -l env
```

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

```
L> kubectl get po -l '!run'
```

NAME	READY	STATUS	RESTARTS	AGE
k8s-rulez	1/1	Running	0	2h

Annotations

- 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 selections 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

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, release IDs, git branch, numéro de Pull|Merge Request numbers, 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)
- Equipes 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/ressources
- 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**

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**

```
L> kubectl get po --namespace kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
heapster-44042	1/1	Running	0	2d
influxdb-grafana-mp6g0	2/2	Running	0	2d
kube-addon-manager-minikube	1/1	Running	0	2d
kube-dns-1326421443-h9rk4	3/3	Running	0	2d
kubernetes-dashboard-xcw0r	1/1	Running	0	2d

Cycle de vie des pods

- Les pods sont mortels
- Ils sont créés, mais ne sont pas ressuscités quand ils meurent
- (C'est le travail des **ReplicaSets/Replication Controllers** que l'on verra par la suite)
- Par contre, si un des containers gérés par le pod **meurt** il est de la responsabilité du pod de le redémarrer (en fonction de la **restartPolicy** sélectionnée pour le pod)

DEMO : Cycle de vie des pods

Init Containers

- Un pod peut être constitué de plusieurs containers hébergeant des applications
- Mais il peut aussi s'appuyer 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?

- Ils peuvent contenir des outils qu'on ne souhaite pas forcément embarquer dans le container principal du pod pour des raisons de sécurité
- Ils peuvent contenir des outils qu'on ne souhaite pas embarquer dans le container principal pour ne pas allourdir l'image (sed, awk, python, ou dig)
- 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)
- Ils sont lancés avant les autres containers, on peut donc s'en servir pour attendre qu'une dépendance soit prête ou que des préconditions soient remplies

DEMO : InitContainer

Replicaset

HealthChecks

- Un des bénéfices de l'utilisation de Kubernetes (et des autres orchestrateurs) c'est de ne pas avoir à se soucier de où tournent les containers
- Répartir les containers sur les différents noeuds fait partie des responsabilités de l'orchestrateur
- Que se passe-t-il si un des containers pilotés par Kubernetes *meurt*?
- Que se passe-t-il si tous les containers d'un pod *meurent*?

Liveness

- Mais comment détecter que l'application ne fonctionne pas quand le container 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

- Une **sonde** (probe) est un diagnostic effectué par Kubernetes sur un container (par un composant appelé **kubelet**)
- 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 container concerné, un code de retour à **0** équivaut à un **succès**
- Les sondes sont utilisées pour vérifier qu'un container **fonctionne** (**Liveness Probe**) ou pour vérifier qu'un container est **prêt** à recevoir du trafic (**Readiness Probe**)
- Une seule sonde de chaque type peut être définie par container

DEMO : Liveness

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)

Pods vs Controllers

Récapitulons :

- **k8s** s'assure que nos containers 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
- Le **centre de contrôle** de **kubernetes** qui tourne sur le(s) **master(s)** ne joue aucun rôle dans cette partie
- 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 (**ReplicationControllers**,) **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

- 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 **rc** va créer ceux qui manquent
 - s'il y en a trop, il va en supprimer pour avoir le nombre demandé
- 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

DEMO : Replicaset

Services

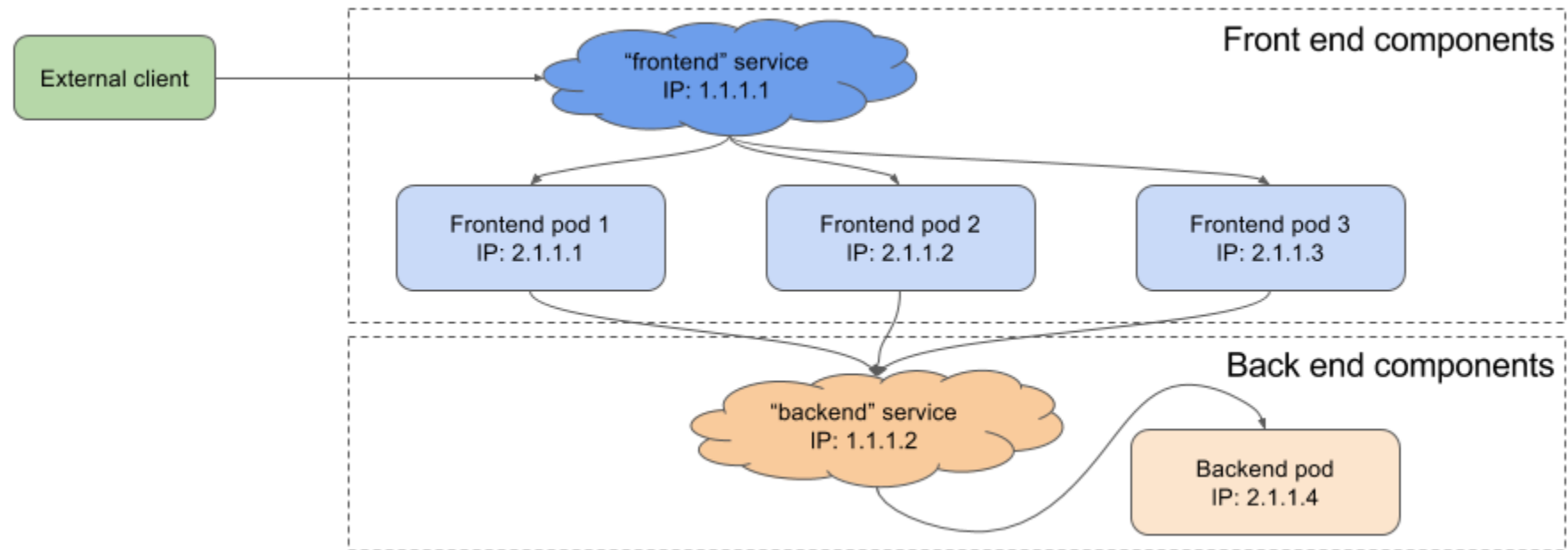
A quoi répond le concept des services

- 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 **ReplicatSet** 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

```
---
kind: Service
apiVersion: v1
metadata:
  name: sise-int
spec:
  selector:
    app: sise
    env: int
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9876
```

- Ce descripteur crée un service qui écoute sur le port **80** et va rediriger le trafic vers tous les pods qui correspondent à **app=sise, env=int** sur le port **9876**.
- Le service se verra assigner une IP interne (**ClusterIP**)

Service Discovery

- Devoir utiliser la **ClusterIP** pour communiquer avec le service n'est pas pratique
- Comment faire pour que les clients puissent s'en passer?
 - Par le biais de variables d'environnement : quand un **Pod** est démarré, Kubernetes initialise un ensemble de variables d'environnement pour chaque service qui existe au moment de la création du pod
 - Par le biais d'adresse DNS :
 - 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>**

DEMO : Services

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 sur possible d'exposer les services pour qu'ils soient accessibles par des clients externes au cluster

Comment faire?

- Utiliser un pod avec un **hostPort**
- Utiliser un service de type **NodePort**
- Utiliser un service de type **LoadBalancer**
- Créer une ressource de type **Ingress**

Service de type NodePort (1/2)

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

- 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
- A noter : une **ClusterIP** sera aussi créée pour la communication interne

Service de type NodePort (2/2)

```
L> kubectl get svc sise-int
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
sise-int	NodePort	10.0.0.72	<none>	80:30123/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 LoadBalancer

- Certains fournisseurs de services **Cloud** proposent une intégration de Kubernetes avec leurs fonctionnalités internes de LoadBalancing (AWS, Azure, GCE)
- En positionnant le `.spec.serviceType` 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)

DEMO : NodePort

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 😊

Ingress?

Ingress (noun) – the act of going in or entering; the right to enter; a means or place of entering; entryway.

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
 - envoy
 - voyager
 - kanali
 - ...

Créer une ressource Ingress

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: sise
spec:
  rules:
  - host: sise.mycompany.com
    http:
      paths:
      - path: /
        backend:
          serviceName: sise
          servicePort: 80
```

- Toutes les requêtes (**Path** vaut **/**) pour le **Host** **sise.mycompany.com** seront redirigées vers le port **80** du service **sise**

Des chemins différents vers des services distincts

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: jj
spec:
  rules:
    - host: abrams.mycompany.com
      http:
        paths:
          - path: /alias
            backend:
              serviceName: forty-seven
              servicePort: 80
          - path: /ze-others
            backend:
              serviceName: lost
              servicePort: 80
```

Plusieurs Hosts

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: size
spec:
  rules:
    - host: fringe.mycompany.com
      http:
        paths:
          - path: /
            backend:
              serviceName: bishop
              servicePort: 80
    - host: lost.mycompany.com
      http:
        paths:
          - path: /
            backend:
              serviceName: shephard
              servicePort: 80
```

DEMO : 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**, le pod ne sera pas mis dans le flux du service associé (autrement dit, il ne sera pas listé dans le **Endpoints** associé au service)

Définition d'une sonde Readiness

```
---
spec:
  containers:
  - name: buddy
    image: looztra/guestbook-storage:0.5.2-aio
    readinessProbe:
      httpGet:
        path: /info
        port: 8080
      initialDelaySeconds: 15
      timeoutSeconds: 1
```

Stratégies de Déploiement

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)

- Etape 1 : mettre à jour le template
- Etape 2 : supprimer les pods utilisant l'ancienne version
- Etape 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

Deployment FTW!



Descripteur de Deployment

```
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: kubia
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: kubia
    spec:
      containers:
      - name: nodejs
        image: luksa/kubia:v1
```


DEMO : Deployment

Code!

<https://github.com/looztra/guestbook-api-server>

