UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Kubernetes avancé

Sommaire

1. **Rappel sur Kubernetes**
   - C'est quoi.
   - Les composantes.
   - Son fonctionnement
2. **Les extensions**
   - CRD
   - Operateur
   - Aggregation Layer
   - Admission Controller
   - Dynamic Admission Controller
   - Scheduler Extender

Sommaire

3. **Helm**
   - C'est quoi.
   - App/lib charts, subcharts, dependencies
   - Pre & post actions/hooks
   - Tester une chart
   - Troubleshooting Helm

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Rappel sur Kubernetes

Rappel sur Kubernetes

C'est quoi ?

Kubernetes est une plateforme open-source de gestion de conteneurs qui automatise le déploiement, la mise à l'échelle et les opérations des applications conteneurisées.

Rappel sur Kubernetes

Vue d'ensemble de Kubernetes

Kubernetes fonctionne en orchestrant un cluster de machines (physiques ou virtuelles) pour exécuter des conteneurs. Chaque cluster Kubernetes comporte les éléments suivants :

- **Master Node** : Gère l'état souhaité du cluster, orchestre les conteneurs et coordonne les nœuds de travail.
- **Worker Nodes** : Exécutent les applications conteneurisées et sont gérés par le nœud maître.

Rappel sur Kubernetes

Composantes principales de Kubernetes

Master Node

1. **API Server**
2. **etcd**
3. **Scheduler**
4. **Controller Manager**

Rappel sur Kubernetes

1. API Server

L'API Server est le **point d'entrée principal** pour toutes les interactions avec Kubernetes. Il reçoit les commandes via des requêtes REST (par exemple, `kubectl`) et communique avec les autres composants comme `etcd`, le Scheduler, et le Controller Manager. Il gère la validation, l'authentification et l'autorisation des requêtes, et expose l'API RESTful de Kubernetes.

Surcharge et Résilience :

- **Évolutivité horizontale** : Vous pouvez exécuter plusieurs instances de l'API Server pour gérer plus de trafic et assurer la haute disponibilité. Ces instances doivent être frontées par un **load balancer** qui répartit les requêtes.
- **Admission Controllers** : Vous pouvez personnaliser la manière dont les objets sont validés avant d'être stockés, en ajoutant des **admission controllers** pour moduler les objets avant leur stockage dans `etcd`.
- **Surveillance et limite de débit** : Configurer des limites de débit (rate limiting) et une surveillance proactive permet d'éviter les surcharges.

Rappel sur Kubernetes

1. API Server

1. Abstraction et Architecture Centralisée :
    - L'API Server agit comme le point central pour tous les composants du cluster Kubernetes. Il expose l'API RESTful que tous les composants du cluster utilisent pour lire et écrire des informations, y compris les contrôleurs (dans le Controller Manager) et le Scheduler. Ce design permet de centraliser les interactions et de fournir une interface cohérente et sécurisée.
2. Sécurité et Contrôle d'Accès :
    - L'API Server gère les autorisations et l'authentification via RBAC (Role-Based Access Control), ce qui garantit que seuls les composants ayant les permissions nécessaires peuvent accéder ou modifier les objets du cluster.
3. Cache et Performance :
    - L'API Server joue aussi un rôle de couche cache pour les interactions avec etcd. Cela permet de réduire la charge sur etcd et d'améliorer les performances globales du cluster. Si le Scheduler ou le Controller Manager communiquait directement avec etcd, cela entraînerait plus de charge sur etcd, ce qui pourrait affecter la performance et la stabilité du cluster.

Rappel sur Kubernetes

2. etcd

`etcd` est la **base de données distribuée** clé-valeur utilisée par Kubernetes pour stocker toutes les données de configuration et l'état du cluster. C'est le **stockage persistant** pour les objets tels que les Pods, Services, ConfigMaps, et plus encore. Toute la gestion de l'état du cluster est stockée dans `etcd`.

Surcharge et Résilience :

- **Haute disponibilité** : Utiliser un cluster `etcd` en configuration HA (3, 5 nœuds) pour assurer la résilience via un quorum. Cela garantit que même en cas de défaillance d'un nœud, les autres nœuds peuvent continuer à fournir les services.
- **Sauvegardes régulières** : Mettre en place des sauvegardes régulières de `etcd` est crucial pour éviter la perte de données. Vous pouvez automatiser ces processus avec des outils comme **Velero**.
- **Chiffrement des données** : Activer le chiffrement des données à la fois en transit et au repos pour protéger les données critiques du cluster.

Rappel sur Kubernetes

3. Scheduler

Le Scheduler est responsable de l'**affectation des Pods aux nœuds**. Il décide où placer un Pod en fonction des demandes de ressources et des contraintes de planification (affinité, anti-affinité, etc.).

Surcharge et Résilience :

- **Custom Scheduler** : Kubernetes permet de définir des **schedulers personnalisés**. Par exemple, vous pourriez vouloir planifier certains types de charges de travail avec un scheduler spécifique qui suit des règles particulières.
- **Priorités de Pods** : Utiliser le concept de **PodPriority** pour s'assurer que les Pods critiques sont traités en premier. Cela renforce la résilience en cas de surcharge.
- **Politique de planification personnalisée** : Vous pouvez ajuster les politiques de planification pour donner la priorité aux ressources spécifiques ou respecter des contraintes géographiques.

Rappel sur Kubernetes

4. Controller Manager

Le **Controller Manager** exécute des contrôleurs spécifiques pour gérer différents objets Kubernetes (comme ReplicaSets, Deployments, etc.). Il s'assure que l'**état actuel** des ressources dans le cluster correspond à l'**état souhaité** défini par les utilisateurs.

Surcharge et Résilience :

- **Leader Election** : Activer le mécanisme d'**élection de leader** pour garantir que même si un contrôleur échoue, un autre pourra le remplacer.
- **Contrôleurs personnalisés** : Vous pouvez créer des **contrôleurs personnalisés** pour gérer des ressources spécifiques à votre environnement. Par exemple, un contrôleur qui surveille des métriques externes et adapte dynamiquement des ressources Kubernetes.
- **Réglage des contrôles** : Ajuster les mécanismes de reprise automatique pour mieux réagir en cas de défaillance d’un Pod ou d'un nœud.

Rappel sur Kubernetes

Workflow Typique (Déploiement d'un Pod) :

1. **Utilisateur → API Server** : L'utilisateur envoie une requête (via `kubectl` ou une API) pour créer un Pod.
2. **API Server → etcd** : L'API Server valide la requête et stocke l'état du Pod dans `etcd`.
3. **etcd → Controller Manager** : Le Controller Manager détecte que le Pod doit être créé et agit pour s'assurer que l'état désiré (par l'utilisateur) est respecté.
4. **API Server → Scheduler** : Le Scheduler détecte les Pods en attente (Pending) et décide sur quel nœud exécuter le Pod.
5. **API Server → Kubelet** : Une fois le nœud choisi, le Kubelet récupère la configuration du Pod via l'API Server et lance le conteneur.
6. **Kubelet → API Server** : Le Kubelet surveille l'état du Pod et renvoie ces informations à l'API Server, qui les stocke dans `etcd`.
7. **Surveillance continue** : Le Controller Manager continue de surveiller l'état du Pod et s'assure qu'il correspond à l'état souhaité.

Rappel sur Kubernetes

La notion de watch dans Kubernetes

Dans Kubernetes, les composants comme le **Scheduler** et le **Controller Manager** n'interagissent pas directement entre eux ou avec `etcd`. Ils utilisent plutôt le **mécanisme de watch** fourni par l'API Server pour suivre les changements d'état des ressources de manière **asynchrone**.

Comment fonctionne le mécanisme de watch :

- **Connexion persistante** : Lorsqu'un composant (comme le Scheduler ou le Controller Manager) s'abonne à un objet Kubernetes (par exemple, un Pod), il utilise l'API Server pour effectuer une requête **watch**. Cette requête maintient une connexion ouverte entre le composant et l'API Server, ce qui permet au composant d'être informé de toute modification en temps réel.
- **Mise à jour en temps réel** : Si un objet suivi (comme un Pod) est créé, modifié, ou supprimé, l'API Server envoie une notification au composant qui l'a surveillé via la requête **watch**. Cela évite de devoir interroger (polling) l'API Server de manière répétée pour détecter des changements, réduisant ainsi la charge globale.
- **Informer et cache local** : Kubernetes utilise également des **Informers** (partie de `client-go`) pour créer un **cache local** des objets Kubernetes. Ce cache est synchronisé avec l'API Server via le mécanisme de watch, et permet aux composants de réagir rapidement aux changements tout en réduisant les appels directs à l'API Server.

Rappel sur Kubernetes

Possibilité de Deux Bases de Données (Events et Data) :

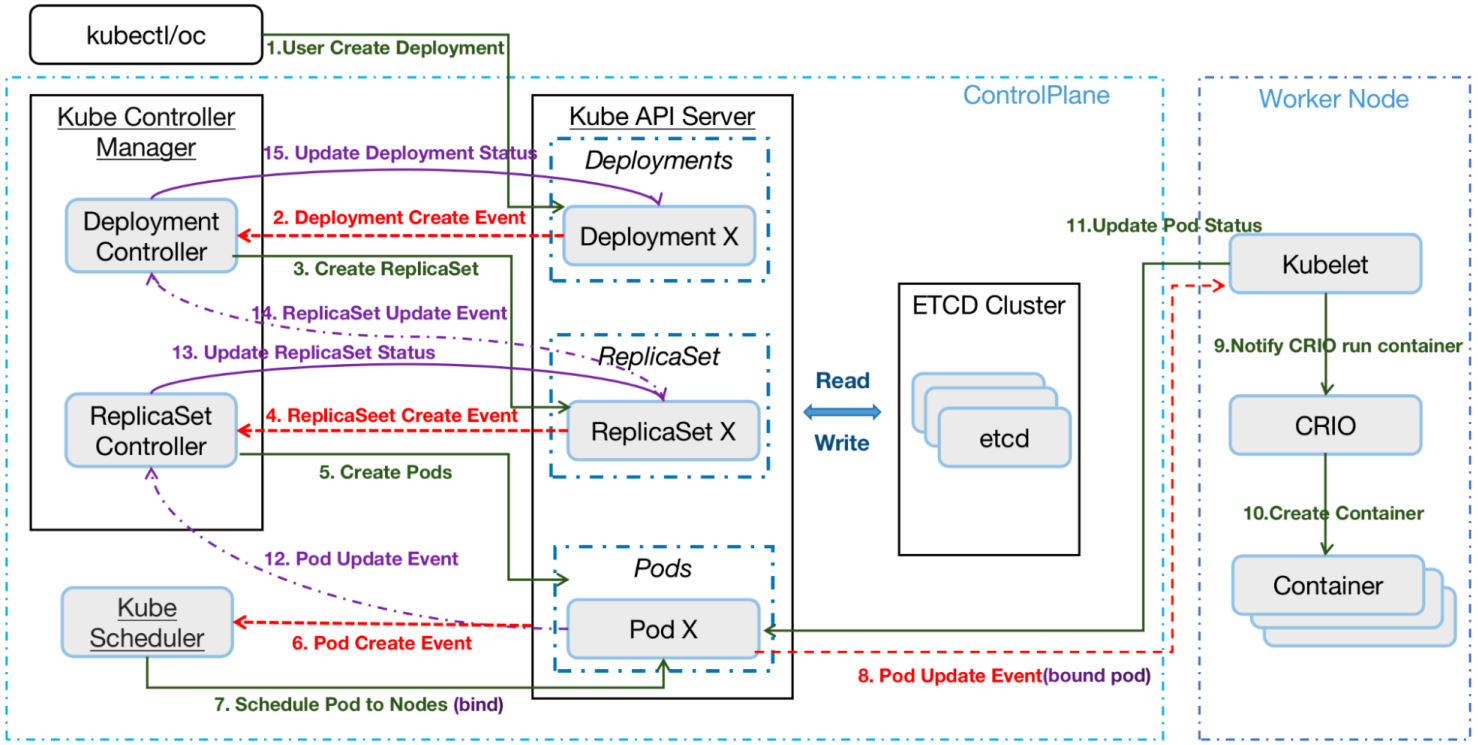
Dans Kubernetes, toutes les données d'état du cluster sont stockées dans `etcd`, y compris les **données de configuration** et les **événements** du cluster. Cependant, il est possible d'**externaliser les logs et les événements** dans une base de données distincte, comme **Elasticsearch**, pour les **événements** (logs), tout en continuant à utiliser `etcd` pour l'état de l'infrastructure.

Comment ça fonctionne :

- **Events et logs externes** : Vous pouvez configurer Kubernetes pour envoyer les logs et événements vers une **solution externe** comme Elasticsearch, Prometheus ou d'autres systèmes de gestion de logs. Ces événements sont capturés via des outils comme Fluentd ou Logstash.
- **Base de données pour les événements** : En séparant les **logs et événements** de la base de données principale (etcd), vous réduisez la charge sur `etcd`, ce qui améliore la performance globale et la résilience. Cela permet aussi de centraliser la gestion des événements dans une base de données optimisée pour les requêtes de logs.

Rappel sur Kubernetes

Composantes principales de Kubernetes



Rappel sur Kubernetes

Composantes principales de Kubernetes

Worker Nodes

1. **Kubelet**
2. **Kube-proxy**
3. **Container Runtime**

Rappel sur Kubernetes

Composantes principales de Kubernetes

1. Kubelet

Le **Kubelet** est un agent crucial qui s'exécute sur chaque nœud de travail (Worker Node) dans Kubernetes. Son rôle principal est de s'assurer que les conteneurs sont en cours d'exécution dans les **Pods** selon les spécifications fournies dans les fichiers de configuration.

Fonctionnement du Kubelet :

- **Surveillance continue** : Le Kubelet surveille l'état des Pods assignés à son nœud en lisant les configurations de l'API Server. Si un Pod est assigné à un nœud particulier, le Kubelet veille à ce que tous les conteneurs spécifiés dans ce Pod soient démarrés et fonctionnent correctement.
- **Liveness & Readiness Probes** : Il utilise les **probes** configurées (liveness et readiness) pour s'assurer de la santé des conteneurs. Si un conteneur échoue un test de liveness, le Kubelet peut redémarrer automatiquement le conteneur.

Rappel sur Kubernetes

Composantes principales de Kubernetes

- **Communication avec l'API Server** : Le Kubelet ne gère pas directement la planification des Pods (c'est le rôle du Scheduler), mais il reçoit des instructions de l'API Server pour lancer les Pods. Il envoie ensuite des mises à jour à l'API Server concernant l'état des Pods, notamment s'ils sont en cours d'exécution, en panne ou terminés.
- **Container Runtime Interface (CRI)** : Le Kubelet utilise l'interface CRI pour interagir avec le runtime de conteneur (par exemple, **Docker** ou **containerd**) qui gère réellement les conteneurs à bas niveau.

Surcharge et Résilience du Kubelet :

- **Redémarrage automatique** : Si le Kubelet lui-même échoue, Kubernetes s'assure qu'il redémarre automatiquement. Le système d'exploitation du nœud (souvent via un service systemd) veille à la disponibilité constante du Kubelet.
- **Resilience avec les Probes** : Utiliser efficacement les **liveness** et **readiness probes** dans les configurations des Pods pour garantir la disponibilité des applications. Par exemple, un Pod peut être redémarré si une probe de liveness échoue.

Rappel sur Kubernetes

Composantes principales de Kubernetes

2. Kube-proxy

Le **Kube-proxy** est un composant qui s'exécute sur chaque nœud de Kubernetes. Il gère la **couche réseau** pour permettre aux Pods de communiquer entre eux, ainsi que d'exposer les services Kubernetes à l'extérieur du cluster.

Fonctionnement du Kube-proxy :

- **Routage et distribution du trafic** : Kube-proxy implémente les règles réseau qui permettent la communication entre les services et les Pods dans le cluster. Il s'appuie sur des mécanismes tels que **iptables**, **ipvs**, ou des proxys utilisateurs pour configurer les règles de routage réseau.
- **Services et Load Balancing** : Le Kube-proxy est également responsable de la gestion du **load balancing** au niveau du nœud. Lorsque plusieurs répliques d'un Pod fournissent un service, Kube-proxy distribue le trafic entrant vers l'une de ces répliques de manière équitable.
- **Sécurité réseau** : Kube-proxy assure également la sécurité des communications en appliquant des règles réseau spécifiques pour limiter l'accès à certains Pods ou services.

Rappel sur Kubernetes

Composantes principales de Kubernetes

3. Container Runtime

Le **Container Runtime** est le logiciel utilisé par Kubernetes pour démarrer, exécuter et gérer les conteneurs sur chaque nœud. Kubernetes peut fonctionner avec plusieurs runtimes de conteneurs, mais les plus courants sont **Docker** et **containerd**. Le Kubelet utilise l'interface **Container Runtime Interface (CRI)** pour interagir avec ces runtimes.

Fonctionnement du Container Runtime :

- **Exécution des conteneurs** : Le runtime est responsable de la création, de l'exécution, et de la gestion du cycle de vie des conteneurs (lancement, arrêt, suppression). Il gère également les ressources système telles que le CPU, la mémoire et le stockage alloués aux conteneurs.
- **Isolation et sécurité** : Les runtimes de conteneurs utilisent des mécanismes comme les **cgroups** (groupes de contrôle) et les **namespaces** Linux pour isoler les conteneurs des autres processus du système.
- **Gestion des images** : Le Container Runtime télécharge et gère les images de conteneurs à partir des registres d'images (comme Docker Hub, ou des registres privés). Il assure que les bonnes images sont utilisées pour démarrer les Pods définis par Kubernetes.

Rappel sur Kubernetes

Composantes principales de Kubernetes

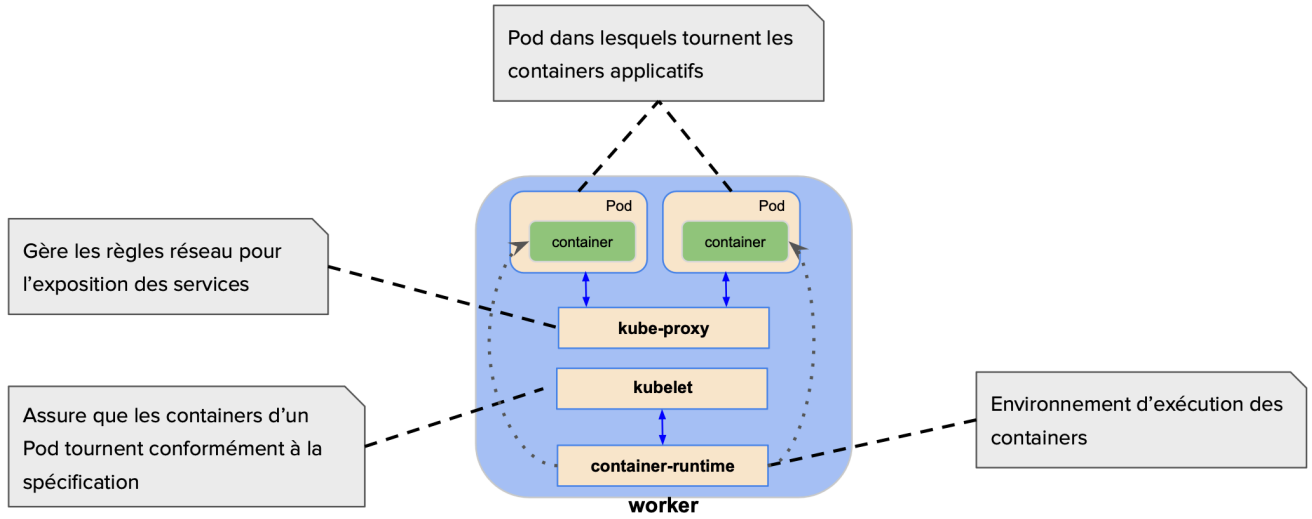
Workflow d'un Pod dans un Worker Node :

1. **API Server → Kubelet** : L'API Server informe le Kubelet du nœud qu'un nouveau Pod doit être démarré. Le Kubelet reçoit la définition du Pod.
2. **Kubelet → Container Runtime** : Le Kubelet utilise l'interface CRI pour demander au Container Runtime (par exemple, containerd) de télécharger l'image du Pod et de démarrer le conteneur.
3. **Container Runtime → Kubelet** : Une fois que le conteneur est en cours d'exécution, le Container Runtime informe le Kubelet que le Pod est actif.
4. **Kube-proxy** : Si le Pod doit exposer un service, le Kube-proxy configure les règles réseau pour permettre la communication avec d'autres services et Pods dans le cluster.
5. **Surveillance continue** : Le Kubelet surveille en permanence l'état du Pod via les probes de santé. Si un conteneur échoue, il demande au Container Runtime de redémarrer le conteneur.
6. **Mise à jour de l'état → API Server** : Le Kubelet communique régulièrement l'état du Pod à l'API Server, qui met à jour l'état dans `etcd` pour refléter les changements.

Rappel sur Kubernetes

Composantes principales de Kubernetes

Les processus : côté Worker



Rappel sur Kubernetes

Composantes avancées de Kubernetes

Pods

- **Pod** : Unité de base de déploiement dans Kubernetes, un pod encapsule un ou plusieurs conteneurs partageant le même réseau et le même espace de stockage.

Services

- **Service** : Abstraction qui définit une politique d'accès pour exposer une application exécutée sur un ensemble de pods. Les services permettent de découvrir et de faire correspondre les pods.

Volumes

- **Volume** : Utilisé pour stocker des données persistantes. Contrairement aux conteneurs éphémères, les volumes peuvent survivre aux redémarrages des pods.

Rappel sur Kubernetes

Composantes avancées de Kubernetes

ConfigMaps et Secrets

- **ConfigMap** : Permet de découpler les configurations environnementales des conteneurs.
- **Secret** : Similaire à ConfigMap, mais destiné à stocker des données sensibles comme des mots de passe, des clés API, etc.

Rappel sur Kubernetes

Composantes avancées de Kubernetes

Controllers

- **ReplicationController** et **ReplicaSet** : Maintiennent un nombre spécifié de répliques de pods en fonctionnement.
- **Deployment** : Fournit des mises à jour déclaratives pour les pods et les ReplicaSets.
- **StatefulSet** : Gère des déploiements d'applications nécessitant des identifiants stables, un stockage persistant ou des commandes de déploiement et de mise à jour ordonnées.
- **DaemonSet** : Assure que tous les nœuds ou certains nœuds exécutent une copie d'un pod spécifié.
- **Job** et **CronJob** : Gèrent l'exécution de tâches de traitement de données ou de tâches périodiques.

Rappel sur Kubernetes

Composantes avancées de Kubernetes

Network Policies

- **Network Policy** : Permet de contrôler le trafic réseau entrant et sortant des pods en définissant des règles de réseau basées sur les étiquettes des pods.

Ingress

- **Ingress** : Gère l'accès externe aux services dans un cluster, généralement via HTTP ou HTTPS.

Helm

- **Helm** : Gestionnaire de packages pour Kubernetes, qui simplifie le déploiement d'applications et de services sur Kubernetes en utilisant des chartes.

Rappel sur Kubernetes

Concepts avancés

- Horizontal Pod Autoscaler (HPA)
- Vertical Pod Autoscaler (VPA)
- RBAC (Role-Based Access Control)

Rappel sur Kubernetes

Concepts avancés

1. Horizontal Pod Autoscaler (HPA)

L'**Horizontal Pod Autoscaler (HPA)** ajuste automatiquement le nombre de Pods dans un déploiement ou un ReplicaSet en fonction de l'utilisation des ressources du cluster, telles que l'**utilisation du CPU** ou de la **mémoire**. L'objectif principal du HPA est de maintenir la performance et l'efficacité des applications en ajustant dynamiquement la capacité en fonction de la charge de travail.

Fonctionnement :

- **Surveillance des métriques** : Le HPA surveille les métriques des Pods (comme l'utilisation du CPU ou de la mémoire) via **Metrics Server**, un composant de Kubernetes qui collecte les métriques du cluster.
- **Algorithme de scaling** : En fonction des valeurs de seuil définies (par exemple, si l'utilisation du CPU dépasse 70 %), le HPA augmente ou réduit le nombre de répliques d'un Pod.

Rappel sur Kubernetes

Concepts avancés

2. Vertical Pod Autoscaler (VPA)

Le **Vertical Pod Autoscaler (VPA)** ajuste automatiquement les **ressources demandées** (CPU et mémoire) des Pods en fonction de leur utilisation réelle. Contrairement à HPA qui agit sur le nombre de répliques, le VPA ajuste les ressources d'un Pod spécifique pour mieux adapter la demande en fonction des besoins réels.

Fonctionnement :

- **Recommandation dynamique** : Le VPA analyse les besoins en ressources des Pods et propose des ajustements des limites de CPU et de mémoire.
- **Mise à jour des ressources** : En fonction de ces recommandations, Kubernetes peut redémarrer les Pods pour appliquer les nouveaux paramètres (ce qui peut entraîner une interruption temporaire). Le VPA offre trois modes : "recommandation", "auto" et "effacement".
- **Applications avec des charges variables** : Le VPA est utile pour des applications qui ont des variations importantes d'utilisation des ressources sur une seule réplique.

Rappel sur Kubernetes

Concepts avancés

3. RBAC (Role-Based Access Control)

Le **Role-Based Access Control (RBAC)** est un mécanisme de sécurité dans Kubernetes qui contrôle l'accès aux ressources du cluster en fonction des **rôles** et des **permissions** attribuées aux utilisateurs et aux applications.

Fonctionnement :

- **Rôles et ClusterRoles** : Un **Rôle** définit un ensemble de permissions sur les ressources d'un **namespace** spécifique, tandis qu'un **ClusterRole** s'applique à l'ensemble du cluster.
- **Bindings** : Les rôles sont appliqués aux utilisateurs ou aux groupes via des **RoleBindings** ou des **ClusterRoleBindings**. Cela permet de limiter qui peut exécuter certaines actions comme lire ou écrire sur des objets spécifiques (Pods, Services, etc.).

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les Extensions (CRDs, Aggregation Layer, Admission Controllers...)

Custom Resource Definition

Definition

- Une **Custom Resource Definition (CRD)** est une fonctionnalité de Kubernetes qui permet aux utilisateurs de créer leurs propres types de ressources personnalisées.
- Les **CRDs** permettent d'étendre l'API Kubernetes pour inclure des ressources que Kubernetes ne prend pas en charge de manière native.
- Cela permet aux utilisateurs de Kubernetes de définir, gérer et manipuler de nouvelles ressources au sein de leurs clusters Kubernetes, comme s'il s'agissait de ressources Kubernetes standard.

Custom Resource Definition

Pourquoi utiliser les CRDs?

Les CRDs sont utiles pour les cas suivants :

- **Extensibilité** : Elles permettent aux utilisateurs de Kubernetes d'étendre les fonctionnalités de Kubernetes sans avoir à modifier le code source de Kubernetes.
- **Gestion des applications** : Elles facilitent la gestion des applications complexes en permettant de définir des API spécifiques à ces applications.
- **Automatisation** : Elles permettent l'automatisation de tâches spécifiques grâce à des opérateurs qui utilisent ces CRDs pour gérer l'état des applications.

Custom Resource Definition

Composantes d'un CRD

1. **`apiVersion: apiextensions.k8s.io/v1`** : Spécifie que cette ressource utilise la version `v1` de l'API `apiextensions.k8s.io`.
2. **`kind: CustomResourceDefinition`** : Indique que cette ressource est une CRD.
3. **`metadata`** : Contient les métadonnées, y compris le nom de la CRD.
4. **`spec`** : La spécification de la CRD, contenant les sous-sections importantes comme `group`, `versions`, `scope`, et `names`.
5. **`group: example.com`** : Définis le groupe d'API pour la ressource personnalisée.
6. **`versions`** : Liste des versions de la ressource. Chaque version contient des informations sur le schéma de la ressource.
7. **`scope: Namespaced`** : Indique que la ressource est limitée à un espace de noms.
8. **`names`** : Définit les noms utilisés pour accéder à la ressource, y compris les noms pluriels, singuliers, le type (`kind`), et les noms abrégés (`shortNames`).

Custom Resource Definition

Création d'une CRD

- Pour créer une CRD, vous devez définir un manifeste YAML qui décrit la nouvelle ressource personnalisée.
- Voici un exemple simple d'une CRD pour une ressource personnalisée appelée `MyResource` :

```yaml
  apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
name: myresources.example.com
spec:
group: example.com
versions:
  - name: v1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              field1:
                type: string
              field2:
                type: integer
```

Custom Resource Definition

Création et utilisation d'une CRD

1. **Déployer la CRD** : Vous pouvez créer la CRD dans le cluster Kubernetes en utilisant la commande `kubectl apply` :

```
kubectl apply -f myresource-crd.yaml
```

Custom Resource Definition

Création et utilisation d'une CRD

2. **Créer des instances de la ressource personnalisée** : Une fois la CRD déployée, vous pouvez créer des instances de la nouvelle ressource personnalisée. Voici un exemple de fichier YAML pour créer une instance de `MyResource` :

```yaml
apiVersion: example.com/v1
kind: MyResource
metadata:
  name: myresource-sample
spec:
  field1: "value1"
  field2: 42
```

Vous pouvez créer cette instance en utilisant la commande `kubectl apply` :

```
kubectl apply -f myresource-instance.yaml
```

Custom Resource Definition

Création et utilisation d'une CRD

3. **Gérer les ressources personnalisées** : Les ressources personnalisées peuvent être gérées comme n'importe quelle autre ressource Kubernetes en utilisant les commandes `kubectl` habituelles, telles que `kubectl get`, `kubectl describe`, et `kubectl delete`.

Custom Resource Definition

Création et utilisation d'une CRD

1. CRD et Namespaces :

Les CRD peuvent être soit **namespaced**, soit **non-namespaced**, selon la définition que vous choisissez au moment de la création. Cela détermine si les ressources personnalisées doivent être **confinées à un namespace** ou si elles sont **globales** à l'ensemble du cluster.

- **CRD Namespaced** : Ces ressources sont associées à un namespace spécifique, ce qui signifie qu'elles ne peuvent exister que dans un ou plusieurs namespaces désignés. Par exemple, si vous créez une CRD pour un objet `Backup`, vous pouvez restreindre son utilisation à des namespaces spécifiques.
- **CRD Non-Namespaced** : Ces ressources sont globales et ne sont pas associées à un namespace particulier. Elles sont accessibles à tout le cluster, comme les objets `Node` ou `ClusterRole`.

Custom Resource Definition

Création et utilisation d'une CRD

Exemple :

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: backups.example.com
spec:
  scope: Namespaced  # ou "Cluster" pour une CRD non-namespaced
  group: example.com
  versions:
  - name: v1
    served: true
    storage: true
  names:
    plural: backups
    singular: backup
    kind: Backup
```

Custom Resource Definition

Création et utilisation d'une CRD

2. Gestion des Versions Multiples de CRD :

Kubernetes prend en charge la gestion des **versions multiples** de CRD, permettant à une API de CRD d'évoluer tout en maintenant la compatibilité avec les versions antérieures. Cela est particulièrement utile lors des mises à jour progressives où les utilisateurs ou les systèmes peuvent utiliser différentes versions en parallèle.

Fonctionnement :

- **Serveur de plusieurs versions** : Une CRD peut exposer plusieurs versions d'une ressource, mais **une seule version** est désignée comme version de **stockage** dans `etcd`, tandis que les autres versions sont simplement "servies" pour l'API.
- **Conversion automatique des versions** : Kubernetes gère automatiquement la conversion des objets entre les versions servies via des mécanismes de conversion, qui peuvent être configurés pour des conversions simples ou complexes.

Custom Resource Definition

Création et utilisation d'une CRD

Exemple de CRD avec plusieurs versions :

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: backups.example.com
spec:
  group: example.com
  names:
    plural: backups
    singular: backup
    kind: Backup
  versions:
  - name: v1
    served: true
    storage: false  # Cette version est exposée mais pas stockée
  - name: v2beta1
    served: true
    storage: true   # Cette version est la version de stockage
  - name: v1alpha1
    served: false   # Cette version n'est plus servie, mais elle peut encore être stockée
```

Custom Resource Definition

Création et utilisation d'une CRD

3. Invoquer Plusieurs Versions de CRD :

Lorsqu'une CRD expose plusieurs versions, vous pouvez choisir laquelle invoquer dans vos appels API en spécifiant la version désirée dans l'URL de la requête API. Kubernetes gère ensuite la conversion et l'interopérabilité entre les versions.

Exemple de requêtes API :

- Pour interagir avec la version `v1` :

```
kubectl get backups.example.com/v1
```

- Pour interagir avec la version `v2beta1` :

```
kubectl get backups.example.com/v2beta1
```

En fonction de la version invoquée, Kubernetes récupérera et convertira automatiquement l'objet au bon format.

Custom Resource Definition

Création et utilisation d'une CRD

4. Stratégies de Conversion des CRD :

Les conversions de versions peuvent être gérées de deux façons :

- **Conversion par défaut** : Kubernetes applique une conversion simple qui adapte automatiquement les champs présents dans les différentes versions sans intervention de l'utilisateur.
- **Conversion via webhook** : Pour des conversions plus complexes (comme changer la structure des champs ou les types de données), un **webhook de conversion** peut être défini. Ce webhook est un service externe auquel Kubernetes envoie les objets lorsqu'ils doivent être convertis d'une version à une autre.

Custom Resource Definition

Création et utilisation d'une CRD

Exemple de webhook pour conversion :

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: backups.example.com
spec:
  conversion:
    strategy: Webhook
    webhook:
      clientConfig:
        service:
          name: my-conversion-service
          namespace: default
          path: "/convert"
```

Ce webhook de conversion recevra les requêtes de conversion des objets CRD entre les différentes versions et renverra l'objet converti à Kubernetes.

Custom Resource Definition

Création et utilisation d'une CRD

5. Mise à jour et évolution des CRD :

- **Ajout de versions** : Vous pouvez ajouter de nouvelles versions à une CRD existante en mettant à jour sa définition. Les objets stockés dans d'anciennes versions seront automatiquement convertis lors de la lecture.
- **Dépréciation des versions** : Si une version doit être retirée, elle peut être marquée comme `served: false`, ce qui signifie que l'API ne la servira plus, mais les objets existants peuvent encore exister dans `etcd`.

6. Contrôleurs personnalisés avec CRD :

Bien que les CRD seules permettent de définir de nouveaux types d'objets, leur véritable puissance réside dans l'implémentation de **contrôleurs personnalisés** qui gèrent ces objets. Un contrôleur personnalisé peut surveiller des objets CRD via le mécanisme **Informer** de Kubernetes et exécuter des actions automatisées pour gérer leur état.

Custom Resource Definition

Utilisation des opérateurs avec les CRDs

- Les **CRDs** sont souvent utilisées en combinaison avec des opérateurs Kubernetes.
- Un opérateur est un contrôleur personnalisé qui utilise les **CRDs** pour gérer les ressources et les applications complexes de manière automatisée.
- Les **opérateurs** surveillent les événements sur les ressources personnalisées et appliquent les modifications nécessaires pour maintenir l'état souhaité.

Custom Resource Definition

Cas d'utilisation typiques

Les **Custom Resource Definitions (CRDs)** de Kubernetes permettent d'étendre l'API Kubernetes en créant de nouvelles ressources personnalisées.

1. Opérateurs Kubernetes (Kubernetes Operators)

Les **opérateurs Kubernetes** utilisent des CRDs pour gérer des applications complexes en automatisant les tâches d'administration, comme la mise à jour, la sauvegarde, et le déploiement d'applications. Un opérateur encapsule la logique de gestion d'une application dans un contrôleur qui utilise des CRDs pour surveiller et gérer l'état de l'application.

Exemple :

- **PostgreSQL Operator** : Un opérateur PostgreSQL pourrait utiliser des CRDs pour créer et gérer des bases de données PostgreSQL, y compris la mise à jour des instances, la gestion des répliques et les sauvegardes.

Custom Resource Definition

Cas d'utilisation typiques

2. Gestion d'infrastructure

Les CRDs sont souvent utilisés pour gérer l'infrastructure en tant que code, où vous pouvez définir des objets qui représentent des éléments d'infrastructure tels que des volumes de stockage, des réseaux, ou des machines virtuelles. Cela permet d'étendre Kubernetes au-delà des ressources de conteneurs.

Exemple :

- **ClusterAPI** : Utilise des CRDs pour gérer les clusters Kubernetes eux-mêmes. Vous pouvez définir des CRDs représentant des **clusters**, des **machines**, et d'autres éléments d'infrastructure nécessaires pour exécuter Kubernetes sur un cloud ou des environnements bare-metal.

Custom Resource Definition

Cas d'utilisation typiques

3. Automatisation des processus métier

Les entreprises peuvent créer des CRDs pour gérer des processus métiers spécifiques. Cela inclut des workflows, des tâches planifiées, ou des pipelines CI/CD entièrement automatisés.

Exemple :

- **Jenkins X** : Utilise des CRDs pour déployer des pipelines CI/CD Kubernetes natifs. Des objets CRD représentent les builds, les versions et les pipelines, automatisant ainsi le déploiement des applications.

Custom Resource Definition

Cas d'utilisation typiques

4. Gestion de la configuration

Les CRDs sont également utilisés pour gérer la configuration complexe d'applications déployées dans Kubernetes. Cela permet de centraliser et de versionner les configurations des applications.

Exemple :

- **KubeDB** : Un projet qui utilise des CRDs pour gérer les bases de données, permettant de créer, configurer, sauvegarder et restaurer des bases de données telles que MySQL, MongoDB ou PostgreSQL dans Kubernetes.

5. Services managés personnalisés

Les CRDs permettent de définir des services managés internes dans un cluster Kubernetes. Cela peut inclure des bases de données managées, des services de cache ou d'autres services spécifiques aux besoins de l'entreprise.

Exemple :

- **Etcd Operator** : Utilise des CRDs pour créer et gérer des clusters Etcd managés. Il gère la création, la sauvegarde, et la récupération des clusters `etcd` à l'aide d'une ressource CRD `EtcdCluster`.

Custom Resource Definition

Cas d'utilisation typiques

6. Applications multi-tenant

Les CRDs sont utiles dans les environnements multi-tenant où des équipes distinctes ou des applications partagent les mêmes ressources Kubernetes mais doivent être isolées au niveau des ressources et de l'administration.

Exemple :

- **Cert-Manager** : Utilise des CRDs pour gérer les certificats SSL/TLS dans un cluster multi-tenant, automatisant le processus d'émission et de renouvellement des certificats pour chaque équipe ou application.

7. Observabilité et Monitoring

Les CRDs sont utilisés pour créer des ressources qui aident à surveiller et à observer l'état du cluster et des applications. Cela inclut la définition de CRDs pour des alertes, des règles de supervision, ou des rapports de diagnostic.

Exemple :

- **Prometheus Operator** : Utilise des CRDs pour gérer les règles d'alertes, les cibles de scrape, et les configurations de Prometheus. Les CRDs permettent de déployer et de gérer Prometheus dans Kubernetes de manière native.

Custom Resource Definition

Commandes pour créer et gérer la CRD

```
# Créer la CRD
kubectl apply -f myresource-crd.yaml

# Vérifier la création de la CRD
kubectl get crds

# Créer une instance de la ressource personnalisée
kubectl apply -f myresource-instance.yaml

# Lister les instances de la ressource personnalisée
kubectl get myresources

# Obtenir des détails sur une instance spécifique
kubectl describe myresource myresource-sample

# Supprimer une instance de la ressource personnalisée
kubectl delete myresource myresource-sample
```

TP CRDs et WebHook

L'objectif de ce TP est de créer un CRD appelé `AppConfig` avec deux versions (`v1` et `v2`), de mettre en place un webhook de conversion pour gérer les transitions entre ces versions, et de déployer ce webhook dans un cluster Kubernetes.

- **Création du CRD**

1. **Définir un CRD `AppConfig`** avec deux versions (`v1` et `v2`). Les spécifications pour chaque version sont les suivantes:
   - Version 1 (`v1`) doit contenir les champs:
     - `appName` (type `string`)
     - `config` (type `object` avec des propriétés additionnelles de type `string`)
   - Version 2 (`v2`) doit contenir les champs:
     - `appName` (type `string`)
     - `configuration` (type `object` qui contient un sous-objet `settings` de type `object` avec des propriétés additionnelles de type `string` et un champ `version` de type `string`)
2. **Configurer la version `v2` comme la version de stockage** et assurez-vous que la version `v1` est toujours servie.

TP CRDs et WebHook

- **Implémentation du Webhook de Conversion**

1. **Développer une application Flask** qui sert de webhook de conversion. Ce webhook doit:
   - Recevoir des demandes de conversion pour le CRD `AppConfig`.
   - Convertir les objets entre les versions `v1` et `v2` selon la `desiredAPIVersion` spécifiée dans la demande.
   - Retourner les objets convertis avec le statut approprié.
2. **Gérer les certificats TLS** pour sécuriser les communications entre l'API server de Kubernetes et le webhook.

- **Déploiement du Webhook**

1. **Créer un Dockerfile** pour votre application Flask.
2. **Construire une image Docker** de votre application.
3. **Déployer cette image dans un cluster Kubernetes**:
   - Configurer un service et un déploiement pour le webhook.
   - Utiliser des Secrets Kubernetes pour gérer les certificats TLS.
4. **Configurer le CRD pour utiliser ce webhook** de conversion.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les opérateurs Kubernetes

Opérateur Kubernetes

Definition

- Un **opérateur Kubernetes** (ou Kubernetes Operator) est une méthode d'automatisation avancée des opérations dans un cluster Kubernetes.
- Il est conçu pour étendre les capacités de Kubernetes en utilisant les ressources natives de Kubernetes et en ajoutant une logique d'application spécifique.
- Les **opérateurs** sont écrits en utilisant l'API Kubernetes et sont utilisés pour gérer des applications complexes et des services au-delà des capacités des contrôleurs Kubernetes standards.

Concepts des opérateurs Kubernetes

Concepts clés

1. Le Controller : Implémentation et Communication

Le **controller** est un composant clé dans Kubernetes qui suit une boucle de surveillance et d'action. Il fonctionne en collaboration avec l'**API Server** de Kubernetes, qui est le point d'entrée principal pour interagir avec le cluster.

1. **Interaction avec l'API Server** :
   - Le controller communique en permanence avec l'API Server de Kubernetes pour observer les objets qui l'intéressent. Il envoie des **requêtes GET** pour lire l'état des objets, ou des **requêtes WATCH** pour recevoir des notifications chaque fois qu'un changement se produit sur ces objets.
   - Chaque fois qu'un nouvel objet (par exemple un objet défini par un CRD comme `MySQLCluster`) est créé ou mis à jour, l'API Server le notifie via le mécanisme de `watch`.
2. **Décision et action** :
   - Le controller analyse ces événements (modifications des objets) et vérifie si l'état réel correspond à l'état souhaité. Si ce n'est pas le cas, il prend des mesures correctives. Ces actions peuvent inclure la création de nouveaux Pods, le redémarrage d'un service, ou d'autres opérations selon la logique définie.

Concepts des opérateurs Kubernetes

Concepts clés

2. La boucle de réconciliation (Reconcilier loop)

La **boucle de réconciliation** est un mécanisme central du controller. Elle permet de s'assurer que chaque ressource dans le cluster atteint son état désiré.

1. **Détection des différences** :
   - La boucle de réconciliation est déclenchée lorsqu'un événement pertinent est reçu (création, mise à jour, suppression d'un objet surveillé). Elle compare l'**état actuel** d'une ressource avec son **état désiré**, défini dans le champ `Spec`.
   - L'état actuel provient de l'API Server, qui expose l'état de tous les objets Kubernetes.
2. **Action corrective** :
   - Si la boucle détecte une différence entre l'état actuel et l'état désiré, elle déclenche une **action corrective**. Cela peut être la création de nouveaux Pods, la suppression de ressources excédentaires, ou la mise à jour d'une configuration.
3. **Itération continue** :
   - La boucle de réconciliation est continue, même après avoir atteint l'état souhaité. Elle fonctionne en

Concepts des opérateurs Kubernetes

Concepts clés

3. `Spec` et `Status` : Définition et Interaction

`Spec` : État désiré

Dans Kubernetes, chaque ressource (que ce soit un Pod, un Service, ou un objet personnalisé) a un champ `Spec`. Ce champ décrit l'état **désiré** de la ressource, c'est-à-dire comment elle devrait être configurée ou fonctionner.

`Status` : État actuel

Le champ `Status` reflète l'état **actuel** de la ressource. Ce champ est automatiquement mis à jour par Kubernetes (souvent par le controller lui-même) pour indiquer l'état dans lequel se trouve la ressource.

Concepts des opérateurs Kubernetes

Concepts clés

Interaction entre `Spec` et `Status` :

Le controller utilise ces deux champs (`Spec` et `Status`) pour décider quelles actions prendre. Il compare constamment le **`Spec`** (ce que vous voulez) avec le **`Status`** (ce que vous avez actuellement).

1. Si les deux correspondent (par exemple, 3 répliques sont demandées et 3 sont en cours d'exécution), alors le controller ne fait rien.
2. Si le `Status` montre que quelque chose ne va pas (par exemple, seulement 2 répliques sont en cours d'exécution alors que `Spec` en demande 3), alors le controller prend des mesures pour corriger cela (il créera un troisième Pod).

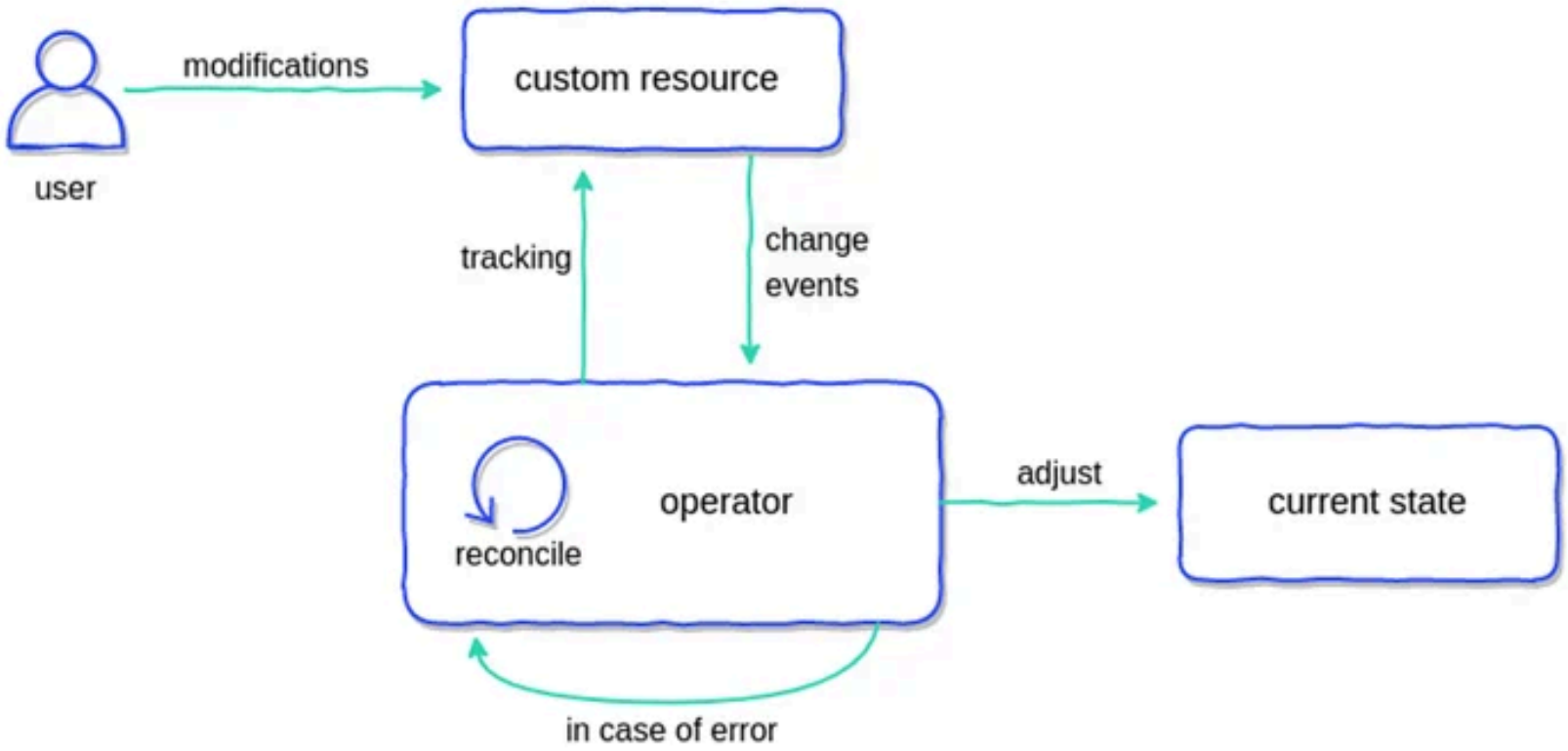
Concepts des opérateurs Kubernetes

Concepts clés

Implémentation dans Kubernetes :

1. **Communication avec l'API Server** :
   - Toutes les actions du controller passent par l'API Server de Kubernetes. Que ce soit pour lire l'état actuel des ressources (via des appels GET ou WATCH) ou pour appliquer des changements (via des appels POST, PUT ou PATCH).
   - L'API Server est donc un point central de communication entre le controller et les ressources du cluster.
2. **Boucle infinie** :
   - Le controller fonctionne dans une boucle infinie, interrogeant constamment l'API Server pour s'assurer que tout est en ordre. S'il détecte une divergence, il réagit, puis continue de surveiller.
3. **Langage et cadre de développement** :
   - La majorité des controllers Kubernetes sont écrits en **Go**, en utilisant le framework **Kubebuilder** ou le **Kubernetes Controller Runtime**. Ces frameworks fournissent des abstractions pour interagir facilement avec l'API Server et mettre en place la boucle de réconciliation.

Opérateurs Kubernetes



Opérateurs Kubernetes

Avantages des opérateurs Kubernetes

1. **Automatisation de la gestion des applications** :
   - Les opérateurs automatisent des tâches complexes comme les déploiements, les sauvegardes, les mises à jour et les restaurations, réduisant ainsi la charge de travail des administrateurs système.
2. **Consistance et répétabilité** :
   - Les opérateurs assurent que les applications sont déployées de manière cohérente et répétable à travers différents environnements, éliminant les erreurs humaines et les variations.
3. **Surveillance et réparation automatique** :
   - En surveillant constamment l'état des applications et en prenant des mesures correctives en cas de besoin, les opérateurs augmentent la résilience et la disponibilité des applications.

Opérateurs Kubernetes

Avantages des opérateurs Kubernetes

4. **Expertise encapsulée** :
   - Les opérateurs codifient les meilleures pratiques et l'expertise nécessaires pour gérer des applications spécifiques, facilitant leur adoption et leur gestion même par des équipes ayant moins d'expérience.
5. **Flexibilité et extensibilité** :
   - Les opérateurs peuvent être étendus pour supporter de nouvelles fonctionnalités ou adapter les comportements en fonction des besoins spécifiques de l'organisation.
6. **Intégration continue et déploiement continu (CI/CD)** :
   - En facilitant l'intégration et le déploiement continus, les opérateurs aident à maintenir les applications à jour avec les dernières versions et correctifs.

Opérateurs Kubernetes

Les grandes étapes de création d'un opérateur Kubernetes

1. **Préparer l'Environnement de Développement**
2. **Initialiser le Projet d'Opérateur**
3. **Créer une API et un Contrôleur**
4. **Définir les Ressources Personnalisées (CRD)**
5. **Implémenter la Logique du Contrôleur**
6. **Construire et Pousser l'Image Docker**
7. **Créer et Gérer les Custom Resources**
8. **Déployer l'Opérateur sur Kubernetes**
9. **Superviser et Maintenir l'Opérateur**

Opérateurs Kubernetes

Les options de création d'un operateur

Créer un opérateur Kubernetes peut être accompli de différentes manières en fonction des outils et frameworks utilisés.

1. Operator SDK

L'**Operator SDK** est un outil populaire qui facilite la création d'opérateurs Kubernetes. Il prend en charge plusieurs langages de programmation et niveaux de complexité :

- **Go** : Pour les développeurs qui préfèrent Go, l'Operator SDK offre un cadre robuste pour écrire des opérateurs.
- **Ansible** : Pour ceux qui préfèrent les scripts d'automatisation, l'Operator SDK permet d'écrire des opérateurs en utilisant Ansible.
- **Helm** : Si vous utilisez déjà des charts Helm, vous pouvez utiliser l'Operator SDK pour gérer vos déploiements Helm avec un opérateur.

Opérateurs Kubernetes

Les options de création d'un operateur

2. Kubebuilder

- Kubebuilder est un autre framework puissant pour développer des opérateurs Kubernetes, en particulier pour les développeurs Go.
- Il utilise les mêmes outils que l'Operator SDK et est basé sur les outils de contrôle de Kubernetes.

Opérateurs Kubernetes

Les options de création d'un operateur

3. Metacontroller

Metacontroller est un framework léger pour écrire des contrôleurs Kubernetes personnalisés en utilisant des scripts simples ou des configurations JSON/YAML.

- **CompositeController** : Pour créer des ressources complexes en réponse à des événements.
- **DecoratorController** : Pour ajouter ou modifier des ressources existantes.

Opérateurs Kubernetes

Les options de création d'un operateur

4. Kopf (Kubernetes Operator Python Framework)

- **Kopf** permet d'écrire des opérateurs Kubernetes en utilisant Python.
- C'est une excellente option pour ceux qui préfèrent écrire des scripts en Python.

Opérateurs Kubernetes

Les options de création d'un operateur

5. Custom Solutions

Il est également possible de développer des opérateurs personnalisés en utilisant directement les bibliothèques client Kubernetes disponibles pour différents langages de programmation (par exemple, client-go pour Go, client-python pour Python).

Opérateurs Kubernetes

Focus : Kubebuilder

- **Kubebuilder** est un framework robuste et extensible pour la création d'opérateurs Kubernetes.
- Développé par l'équipe SIG API Machinery de Kubernetes, **Kubebuilder** est conçu pour simplifier le processus de développement d'opérateurs en fournissant des outils et des abstractions qui automatisent de nombreuses tâches courantes.

Opérateurs Kubernetes

Focus : Kubebuilder

Principales caractéristiques de Kubebuilder

Scaffolding de projets :
Kubebuilder génère le squelette d'un projet opérateur, incluant la structure des répertoires, les dépendances, et les fichiers de configuration nécessaires.

Génération de code :
Kubebuilder génère automatiquement du code pour les CRDs (Custom Resource Definitions), les contrôleurs et les webhooks, réduisant ainsi le besoin de code boilerplate.

Intégration avec Controller Runtime :
Kubebuilder utilise la bibliothèque Controller Runtime, qui fournit des abstractions haut niveau pour écrire des contrôleurs Kubernetes.

Opérateurs Kubernetes

Focus : Kubebuilder

Principales caractéristiques de Kubebuilder

Validation et conversion des CRDs :
Kubebuilder supporte la validation et la conversion des versions des CRDs, facilitant la gestion des évolutions des API des opérateurs.

Support pour les webhooks :
Kubebuilder permet de créer des webhooks d'admission mutante et validante pour ajouter une logique de validation ou de modification des ressources.

Tests et débogage:
Kubebuilder fournit des outils pour écrire des tests unitaires et des tests d'intégration, facilitant le développement et le débogage des opérateurs.

Opérateurs Kubernetes

Focus : Kubebuilder

Étape 1 : Installer les outils nécessaires

1. **Installer Go** :
   - Suivez les instructions d'installation de Go sur le site officiel : https://golang.org/doc/install
2. **Installer Kubebuilder** :
   - Télécharger Kubebuilder depuis le dépôt GitHub :

```
curl -L https://github.com/kubernetes-sigs/kubebuilder/releases/download/vX.X.X/kubebuilder_$(go env GOOS)_$(go env GOARCH) -o /usr/local/bin/kubebuilder
chmod +x /usr/local/bin/kubebuilder
```

3. **Installer Kubernetes et Minikube** pour tester localement.

Opérateurs Kubernetes

Focus : Kubebuilder

Étape 2 : Initialiser le projet d'opérateur

1. **Créer un répertoire de projet** :
   - Créez un dossier pour votre opérateur :

```
mkdir my-operator
cd my-operator
```

2. **Initialiser le projet Kubebuilder** :
   - Utilisez la commande Kubebuilder pour initialiser un nouveau projet :

```
kubebuilder init --domain example.com --repo my-operator
```

- Cela génère la structure de base du projet, avec les fichiers nécessaires (les fichiers Go, les Makefiles, etc.).

Opérateurs Kubernetes

Focus : Kubebuilder

3. **Créer une API et un controller** :
    - Créez les définitions pour une nouvelle API et un controller :

```
kubebuilder create api --group mygroup --version v1 --kind MyResource
```

- Cela va :
    - Générer un CRD `MyResource`.
    - Créer un fichier `controller_myresource.go` qui contiendra la logique de gestion de votre ressource personnalisée.

Opérateurs Kubernetes

Focus : Kubebuilder

Étape 3 : Définir le `Spec` et le `Status`

Maintenant que votre API est créée, vous devez définir la ressource personnalisée (CRD) en remplissant les champs `Spec` et `Status` dans les fichiers Go générés :

1. **Modifier le fichier de type** :
   - Le fichier `api/v1/myresource_types.go` contient la définition des champs `Spec` et `Status`.
   - Exemple :

```go
type MyResourceSpec struct {
    Replicas int `json:"replicas,omitempty"`
}

type MyResourceStatus struct {
    AvailableReplicas int `json:"availableReplicas,omitempty"`
}
```

- Le `Spec` définit les valeurs souhaitées (par exemple, le nombre de répliques), tandis que le `Status` reflète l'état actuel.

Opérateurs Kubernetes

Focus : Kubebuilder

2. **Générer les CRD** :
    - Utilisez `kubebuilder` pour générer les fichiers CRD YAML à partir des types Go :

```
make manifests
```

3. **Appliquer la CRD dans votre cluster** :
    - Déployez la définition CRD dans votre cluster Kubernetes :

```
make install
```

Cela installera la CRD `MyResource` dans Kubernetes, rendant ainsi cette ressource disponible dans le cluster.

Opérateurs Kubernetes

Focus : Kubebuilder

Étape 4 : Implémenter la logique du controller

Maintenant que vous avez défini la CRD, il faut implémenter la logique qui surveille cette ressource et agit en conséquence (via la boucle de réconciliation).

1. **Modifier le fichier `controller` :**
   - Ouvrez `controllers/myresource_controller.go`. Ce fichier contient la logique de réconciliation.

```go
func (r *MyResourceReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    // Fetch the MyResource instance
    myresource := &mygroupv1.MyResource{}
    if err := r.Get(ctx, req.NamespacedName, myresource); err != nil {
        if errors.IsNotFound(err) {
            // MyResource not found, ignore it.
            return ctrl.Result{}, nil
        }
        return ctrl.Result{}, err
    }

    // Compare the desired state (`Spec`) with the actual state (`Status`)
    if myresource.Spec.Replicas != myresource.Status.AvailableReplicas {
        // Implement logic to handle the discrepancy (e.g., create/delete Pods)
    }

    // Update the status
    myresource.Status.AvailableReplicas = actualReplicas
    if err := r.Status().Update(ctx, myresource); err != nil {
        return ctrl.Result{}, err
    }

    return ctrl.Result{}, nil
}
```

Opérateurs Kubernetes

2. **Registrez le controller** :

- Assurez-vous que le controller est bien enregistré dans `main.go` :

```
if err = (&controllers.MyResourceReconciler{
    Client: mgr.GetClient(),
    Scheme: mgr.GetScheme(),
}).SetupWithManager(mgr); err != nil {
    setupLog.Error(err, "unable to create controller", "controller", "MyResource")
    os.Exit(1)
}
```

Opérateurs Kubernetes

Étape 5 : Tester l'opérateur localement

1. **Compiler et déployer l'opérateur** :
   - Utilisez `make` pour compiler et déployer votre opérateur dans Kubernetes :

```
make docker-build docker-push
```

2. **Lancer l'opérateur dans le cluster** :
   - Déployez votre opérateur dans le cluster :

```
make deploy
```

Opérateurs Kubernetes

3. **Créer une instance de la ressource personnalisée** :
    - Créez une instance de votre ressource `MyResource` dans Kubernetes via un fichier YAML :

```yaml
apiVersion: mygroup.example.com/v1
kind: MyResource
metadata:
  name: my-example
spec:
  replicas: 3
```

- Appliquez-le dans le cluster :

```
kubectl apply -f config/samples/mygroup_v1_myresource.yaml
```

Opérateurs Kubernetes

Étape 6 : Superviser et déboguer

1. **Vérifiez que l'opérateur fonctionne** :
   - Observez les logs de votre opérateur pour vous assurer qu'il fonctionne correctement :

```
kubectl logs -f deployment/my-operator-controller-manager
```

2. **Vérifiez les ressources** :
   - Vérifiez que les ressources définies dans votre opérateur (`MyResource`) fonctionnent correctement en utilisant `kubectl get` :

```
kubectl get myresource
```

Opérateurs Kubernetes

La **concurrence d'accès** dans Kubernetes, lorsque vous avez plusieurs instances de votre controller (ce qui peut se produire pour assurer la haute disponibilité ou une gestion distribuée), est gérée via un mécanisme qui évite que deux instances ne tentent d'agir simultanément sur les mêmes ressources.

1. Le mécanisme de "Leader Election"

Kubernetes utilise un mécanisme appelé **élection de leader** (Leader Election) pour résoudre le problème de concurrence entre plusieurs instances d'un même controller. Ce mécanisme garantit qu'à tout moment, une seule instance d'un controller est active et responsable de la gestion des ressources, tandis que les autres restent en veille et n'interviennent pas directement.

- Lorsqu'un controller est déployé avec plusieurs répliques (ou instances), une **élection de leader** est effectuée.
- L'instance élue en tant que leader est la seule qui exécute la logique de réconciliation et prend des décisions pour ajuster les ressources.
- Les autres instances du controller sont passives et attendent qu'un nouvel événement d'élection de leader se produise (par exemple, si l'instance leader échoue ou est arrêtée).
- Ce mécanisme est basé sur la création d'un objet `ConfigMap` ou d'un **lock** dans le cluster Kubernetes, qui est utilisé pour indiquer quelle instance est le leader.

Opérateurs Kubernetes

Exemple d'implémentation avec Kubebuilder :

Avec **Kubebuilder**, le support pour l'élection de leader est souvent activé par défaut lorsque vous déployez plusieurs instances de votre controller. Dans le fichier `main.go`, Kubebuilder configure généralement l'élection de leader automatiquement :

```
mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
    Scheme:                 scheme,
    MetricsBindAddress:     metricsAddr,
    LeaderElection:         enableLeaderElection,
    LeaderElectionID:       "my-operator-leader-election",
})
```

- **LeaderElectionID** est un identifiant unique utilisé pour identifier quel controller est le leader.
- **LeaderElection** est une option activée pour dire au manager Kubernetes qu'il doit gérer l'élection de leader.

Opérateurs Kubernetes

2. La gestion de la concurrence dans la boucle de réconciliation

Même avec une élection de leader, il est important que la boucle de réconciliation soit capable de gérer les conflits potentiels. Voici comment cela se passe :

1. **Locking au niveau des objets** :
   - Le controller peut implémenter une stratégie de **locking optimiste** en vérifiant les versions (`resourceVersion`) des objets Kubernetes.
   - Chaque fois qu'un objet dans Kubernetes (comme un Pod ou un CRD) est modifié, son champ `resourceVersion` est mis à jour. Avant d'appliquer une modification, le controller peut vérifier si la `resourceVersion` n'a pas changé depuis la dernière fois qu'il l'a consultée.
   - Si une autre instance du controller ou un autre composant a modifié l'objet, le controller détectera que la `resourceVersion` a changé et pourra recharger l'état de l'objet avant d'essayer à nouveau.

Opérateurs Kubernetes

2. **Rejet et reprise automatique** :
   - Si un controller tente de mettre à jour un objet et qu'un conflit de version est détecté (l'objet a été modifié par une autre instance du controller ou un autre processus), Kubernetes retourne une erreur de conflit (**409 Conflict**).
   - Dans ce cas, la logique de la boucle de réconciliation est configurée pour **réessayer automatiquement** après un délai. Cela permet d'éviter les conflits tout en garantissant que l'état désiré sera finalement atteint.

Exemple de gestion de conflit dans un controller :

```
err := r.Client.Update(ctx, myResource)
if apierrors.IsConflict(err) {
    // Recharger la ressource et réessayer
    return ctrl.Result{Requeue: true}, nil
}
```

Opérateurs Kubernetes

3. Stratégies de gestion de la concurrence :

Voici quelques techniques supplémentaires pour gérer la concurrence dans les opérateurs :

a. Requêtes Idempotentes

- Les opérations exécutées par le controller dans la boucle de réconciliation doivent être **idempotentes**. Cela signifie que si la même action est exécutée plusieurs fois (par exemple, créer un Pod, configurer une ressource), cela ne doit pas entraîner d'effets indésirables ou incohérents.
- L'idempotence garantit que même en cas de ré-exécution de la même opération due à des conflits ou des erreurs, l'état final des objets restera correct.

b. Rate Limiting (Limitation de débit)

- Pour éviter que plusieurs instances d'un controller ne surcharge le cluster ou n'entrent en conflit trop rapidement, Kubernetes permet de mettre en place des **limiteurs de débit** dans la boucle de réconciliation.
- Cela limite le nombre de tentatives de réconciliation par unité de temps, offrant une approche plus progressive et réduisant les risques de conflits d'accès.

Opérateurs Kubernetes

c. Utilisation des Finalizers

- Les **finalizers** sont des mécanismes que les controllers peuvent ajouter aux objets Kubernetes. Ils garantissent qu'un objet ne peut pas être supprimé tant qu'une certaine logique n'a pas été exécutée.
- Dans un contexte de concurrence, les finalizers permettent de s'assurer qu'une ressource n'est pas supprimée par une autre instance ou un autre processus tant que le controller actuel n'a pas terminé son travail.

4. Redémarrage ou basculement en cas de panne

Lorsque le leader échoue ou est redémarré, une nouvelle élection de leader a lieu, et l'une des instances passives devient le nouveau leader. Cette **tolérance aux pannes** permet d'assurer que même si une instance du controller est indisponible, une autre instance prendra le relais et continuera à gérer les ressources sans interruption.

TP Opérateur de Gestion d'Applications Web Scalables

Créer un opérateur Kubernetes pour gérer le déploiement, la configuration et la mise à l'échelle automatique d'une application web composée d'un frontend et d'une base de données backend. L'opérateur doit automatiquement configurer une politique de mise à l'échelle basée sur le trafic HTTP.

1. **Description de la CRD `WebApp`**

- **Champs spécifiques :**
  - `appName` (String): Nom de l'application.
  - `image` (String): Image Docker de l'application frontend.
  - `dbImage` (String): Image Docker pour la base de données.
  - `replicas` (Int): Nombre initial de pods pour le frontend.
  - `dbSize` (String): Taille de l'allocation de stockage pour la base de données.
  - `autoScaleEnabled` (Boolean): Indique si la mise à l'échelle automatique est activée.
  - `trafficThreshold` (Int): Seuil de trafic HTTP pour déclencher la mise à l'échelle.

2. **Initialisation et Création de la CRD**
   - Utilisez Kubebuilder pour initialiser un projet et créer une API pour la CRD `WebApp`.
   - Définissez les champs dans `api/v1/webapp_types.go`.

TP Opérateur de Gestion d'Applications Web Scalables

3. **Définition des Manifestes de la CRD**
   - Générez et modifiez les manifestes pour inclure les spécifications de la CRD, incluant les validations.
4. **Développement du Contrôleur**
   - Implémentez la logique dans `controllers/webapp_controller.go` pour :
     - Déployer l'application et la base de données en utilisant les images spécifiées.
     - Configurer un service et un volume persistant pour la base de données.
     - Activer la mise à l'échelle automatique en utilisant les métriques de trafic HTTP si `autoScaleEnabled` est `true`.
5. **Tests Locaux**
   - Testez l'opérateur localement ou dans un cluster de développement.
   - Vérifiez que l'opérateur réagit correctement aux changements de la CRD, ajuste les déploiements et configure la mise à l'échelle.

Opérateurs Kubernetes

Focus : Operateur SDK

- Operator SDK est un outil de développement qui simplifie la création d'opérateurs Kubernetes.
- Il offre des abstractions et des outils pour automatiser les tâches courantes dans le cycle de vie des applications Kubernetes.
- Développé par la CNCF (Cloud Native Computing Foundation), l'Operator SDK supporte plusieurs langages et frameworks, notamment Go, Ansible et Helm.

Opérateurs Kubernetes

Focus : Operateur SDK

Principales caractéristiques de l'Operator SDK

Scaffolding de projets :

- Génère la structure de base des projets opérateurs, incluant les définitions de types, les contrôleurs et les configurations nécessaires.

Support Multi-langages :

- Permet de développer des opérateurs en utilisant Go, Ansible ou Helm, en fonction des besoins et des préférences des développeurs.

Génération de code :

- Automatise la génération du code nécessaire pour les CRDs et les contrôleurs, réduisant ainsi la quantité de code boilerplate.

Opérateurs Kubernetes

Focus : Operateur SDK

Principales caractéristiques de l'Operator SDK

Validation et conversion des CRDs :

- Facilite la gestion des versions des API et la validation des schémas des CRDs.

Outils de test :

- Fournit des outils pour écrire des tests unitaires et d'intégration, facilitant le développement et le débogage des opérateurs.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Admission Controller

Admission Controller

Definition

- Les **Admissions Controllers** sont des plugins au sein de Kubernetes qui interceptent les requêtes au serveur d'API après leur authentification et autorisation, mais avant qu'elles ne soient persistées dans etcd (la base de données de Kubernetes).
- Ces plugins peuvent modifier ou rejeter des requêtes.
- Ils jouent un rôle crucial pour appliquer des politiques de sécurité, valider des configurations et implémenter des contraintes spécifiques.

Admission Controller

Fonctionnement des admission Controllers

1. **Authentification** : La requête est authentifiée pour vérifier l'identité de l'utilisateur ou du service.
2. **Autorisation** : La requête est autorisée pour s'assurer que l'utilisateur ou le service a les permissions nécessaires pour effectuer l'action demandée.
3. **Admission Control** : Les Admission Controllers interceptent la requête. Ils peuvent la modifier ou la rejeter en fonction des politiques définies.

Admission Controller

Fonctionnement des admission Controllers

Il existe plusieurs types principaux d'Admission Controllers :

1. **NamespaceLifecycle** : Gère la création et la suppression des namespaces, en empêchant la création de ressources dans des namespaces en cours de suppression.
2. **ResourceQuota** : Assure que les namespaces ne dépassent pas les quotas de ressources alloués (CPU, mémoire, etc.).
3. **LimitRanger** : Applique des limites et des demandes par défaut pour les ressources si elles ne sont pas spécifiées par l'utilisateur.
4. **ServiceAccount** : Assure que les pods ont un compte de service associé, permettant ainsi une gestion fine des permissions et de la sécurité.
5. **NodeRestriction** : Restreint les modifications que les kubelets (agents de nœuds) peuvent apporter aux ressources des nœuds et des pods.

Admission Controller

Fonctionnement des admission Controllers (dynamiques)

Il existe plusieurs types principaux d'Admission Controllers :

1. **Mutating Admission Controllers** : Ils peuvent modifier les objets avant qu'ils ne soient persistés.
2. **Validating Admission Controllers** : Ils valident les objets et peuvent rejeter les requêtes non conformes.
3. **Webhook Admission Controller** : Utilise des webhooks pour déléguer la logique de mutation ou de validation à des services externes. Ceci permet de centraliser la logique d'admission dans des services indépendants.
4. **Initializers** : Un type d'admission controller qui permet d'ajouter des initialisateurs à des objets avant leur création complète. Les initializers sont progressivement dépréciés au profit des webhooks mutating.

Admission Controller

Fonctionnement général

1. **Intercept Request** : Les contrôleurs d'admission interceptent les requêtes qui arrivent au serveur d'API avant qu'elles ne soient persistées dans etcd (la base de données clé-valeur de Kubernetes).
2. **Evaluate Request** : Ils évaluent la requête en fonction des règles et politiques définies. Cela peut inclure la vérification des quotas, la validation de la sécurité, et l'application de valeurs par défaut.
3. **Modify or Reject Request** : En fonction de l'évaluation, le contrôleur peut modifier la requête (mutation) ou la rejeter (validation).
4. **Pass to Next Stage** : Si la requête est acceptée, elle est passée aux étapes suivantes pour le traitement par le serveur d'API et, éventuellement, persistée dans etcd.

Admission Controller

Mutating Admission Controllers

- Les **Mutating admission controllers** sont responsables de la modification des objets envoyés au serveur API avant qu'ils ne soient persistés dans etcd.
- Ces contrôleurs peuvent ajouter, modifier ou supprimer des champs dans les objets.

Admission Controller

Mutating Admission Controllers

Fonctionnalités et utilisations :

- **Ajout de labels ou d'annotations** : Ils peuvent ajouter automatiquement des labels ou des annotations à des ressources nouvellement créées.
- **Définition de valeurs par défaut** : Ils peuvent définir des valeurs par défaut pour les champs non spécifiés par l'utilisateur.
- **Injection de sidecars** : Ils peuvent ajouter automatiquement des conteneurs sidecar dans les pods (par exemple, pour la gestion des logs ou la sécurité).

Admission Controller

Exemple : Mutating Admission Controllers

Un exemple courant de Mutating Admission Controller est l'injection automatique de sidecars, comme les conteneurs Envoy pour les proxys de service mesh.

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  name: example-mutating-webhook
webhooks:
  - name: example.mutating.webhook.com
    clientConfig:
      service:
        name: example-service
        namespace: default
        path: "/mutate"
      caBundle: <base64-encoded-ca-cert>
    rules:
      - operations: ["CREATE", "UPDATE"]
        apiGroups: [""]
        apiVersions: ["v1"]
        resources: ["pods"]
    admissionReviewVersions: ["v1", "v1beta1"]
    sideEffects: None
```

Admission controller

Validation admission controllers

Fonctionnalités et utilisations :

- **Contrôle de conformité** : Ils s'assurent que les objets créés respectent les politiques de sécurité et les contraintes de l'organisation.
- **Validation des champs** : Ils vérifient que les champs des objets contiennent des valeurs valides et appropriées.
- **Enforcement des politiques** : Ils peuvent imposer des règles spécifiques, comme l'utilisation de certaines images de conteneurs ou des configurations réseau.

Admission Controller

Exemple : validation admission controllers

Un exemple de Validating Admission Controller est la vérification des spécifications de déploiement pour s'assurer qu'elles respectent les politiques de l'organisation.

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: example-validating-webhook
webhooks:
  - name: example.validating.webhook.com
    clientConfig:
      service:
        name: example-service
        namespace: default
        path: "/validate"
      caBundle: <base64-encoded-ca-cert>
    rules:
      - operations: ["CREATE", "UPDATE"]
        apiGroups: [""]
        apiVersions: ["v1"]
        resources: ["pods"]
    admissionReviewVersions: ["v1", "v1beta1"]
    sideEffects: None
```

TP

L'objectif du TP est de créer deux webhooks pour un cluster Kubernetes:

1. **Mutating Webhook**: Ajoute automatiquement un label `departement` aux Pods qui n'en ont pas, basé sur le namespace dans lequel le Pod est déployé.
2. **Validating Webhook**: Vérifie que tous les Pods ont une annotation `owner` avant leur création. Si l'annotation `owner` est manquante, le Pod ne doit pas être créé.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

API Aggregation

API Aggregation

L'**API Aggregation** dans Kubernetes est un mécanisme qui permet d'étendre l'API native de Kubernetes en ajoutant des serveurs API supplémentaires, appelés **aggregated API servers**, tout en maintenant le kube-apiserver principal inchangé. Cela permet d'introduire des fonctionnalités supplémentaires, des API personnalisées, ou des services spécifiques directement au sein du cluster Kubernetes sans affecter le cœur du système.

API Aggregation

Fonctionnement détaillé de l'API Aggregation

1. Principe de base :

L'API Aggregation repose sur l'idée d'un **proxy intégré** dans le kube-apiserver, qui permet de rediriger des requêtes destinées à des APIs non natives (agrégées) vers des serveurs API supplémentaires déployés au sein du cluster. Cette approche permet de créer des extensions modulaires à l'API Kubernetes sans toucher au serveur API principal.

- **Déploiement des aggregated API servers** : Les nouveaux serveurs API sont déployés comme des pods dans le cluster Kubernetes. Ils sont totalement indépendants du kube-apiserver et fonctionnent comme des services Kubernetes standards.
- **Enregistrement des API personnalisées** : Ces serveurs API doivent s'enregistrer auprès du kube-apiserver en utilisant des objets **APIService**. Ce processus d'enregistrement permet au kube-apiserver de reconnaître qu'il existe une nouvelle API (par exemple, `/apis/monitoring/v1/`) et de savoir où rediriger les requêtes correspondantes.
- **Proxying des requêtes** : Le kube-apiserver, en tant que proxy, reçoit toutes les requêtes destinées à des API agrégées et les redirige vers le bon serveur API en fonction de l'API demandée. Une fois que la requête est redirigée, le serveur API agrégé répond comme s'il faisait partie intégrante du système Kubernetes.

API Aggregation

2. Architecture de l'API Aggregation

a) kube-apiserver :

Le kube-apiserver est le serveur central qui gère toutes les requêtes API dans Kubernetes. Dans le cadre de l'API Aggregation, il joue également le rôle de **proxy** en dirigeant les requêtes vers les aggregated API servers enregistrés.

- **Extension via APIService** : Le kube-apiserver utilise l'objet **APIService** pour enregistrer les nouvelles API. Cet objet spécifie le groupe d'API, les versions disponibles, et l'adresse du serveur API agrégé à qui rediriger les requêtes.

b) Aggregated API Server :

Ce sont des serveurs API supplémentaires déployés dans Kubernetes. Ils fonctionnent indépendamment du kube-apiserver et peuvent offrir des fonctionnalités personnalisées. Ces serveurs API doivent respecter les conventions des API Kubernetes pour s'intégrer de manière fluide dans le cluster.

- **Exemple de serveur API agrégé** : Un serveur API agrégé pourrait fournir une API pour gérer les certificats SSL dans Kubernetes. Les utilisateurs pourraient interagir avec cette API via des commandes `kubectl` comme s'il s'agissait d'une API native, sans savoir que c'est un serveur distinct derrière le kube-apiserver.

API Aggregation

c) APIService Object :

C'est un objet Kubernetes essentiel qui agit comme un registre pour les API agrégées. Il est utilisé pour enregistrer les nouveaux serveurs API et indiquer au kube-apiserver vers quels serveurs rediriger les requêtes pour ces API.

Exemple d'un objet APIService :

```yaml
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  name: v1.monitoring.example.com
spec:
  service:
    name: monitoring-service       # Le service Kubernetes qui expose l'aggregated API server
    namespace: monitoring-namespace
  group: monitoring.example.com    # Le groupe d'API
  version: v1                      # La version de l'API
  insecureSkipTLSVerify: true      # Pour éviter la vérification TLS dans cet exemple
  groupPriorityMinimum: 1000       # Priorité du groupe d'API
  versionPriority: 10              # Priorité de la version de l'API
```

API Aggregation

3. Avantages de l'API Aggregation

a) Extensibilité :

L'API Aggregation permet d'ajouter de nouvelles fonctionnalités au cluster Kubernetes sans avoir besoin de modifier le kube-apiserver principal. Cela facilite l'ajout d'extensions sans risquer d'affecter la stabilité ou la sécurité du serveur API principal.

- **Exemple d'extension** : Supposons qu'on développe un système pour gérer des pipelines CI/CD complexes. Plutôt que d'intégrer ces fonctionnalités directement dans Kubernetes, on peut créer une API agrégée dédiée, déployée comme un serveur API distinct.

b) Modularité :

Les aggregated API servers sont déployés et maintenus séparément du kube-apiserver. Chaque serveur API peut être développé, mis à jour et déployé indépendamment, ce qui permet de maintenir une approche modulaire.

- **Exemple de modularité** : Si on veut ajouter une nouvelle API pour gérer des secrets ou des certificats, on peut le faire via un serveur API agrégé sans affecter le cycle de vie ou les versions du kube-apiserver.

API Aggregation

c) Isolation :

Les aggregated API servers fonctionnent indépendamment du kube-apiserver. Cela signifie que si un serveur API agrégé rencontre un problème ou échoue, cela n'affecte pas le kube-apiserver principal, assurant ainsi une meilleure résilience.

- **Exemple d'isolation** : Si un serveur API agrégé pour la gestion des métriques tombe en panne, cela n'impacte pas les autres services ni le kube-apiserver. Le reste du cluster continue de fonctionner normalement.

API Aggregation

5. Cas d'utilisation avancés de l'API Aggregation

a) Systèmes de Monitoring avancés :

Avec l'API Aggregation, on peut développer une API complète pour gérer et exposer des métriques de monitoring avancées.

- **Exemple** : Un serveur API agrégé dédié au monitoring pourrait fournir des endpoints pour récupérer des métriques spécifiques comme le nombre de requêtes, le temps de réponse moyen, ou les niveaux d'utilisation des ressources. Par exemple, `/apis/monitoring/v1/metrics`.

b) Gestion des secrets et des certificats :

Les systèmes qui gèrent des données sensibles, comme les secrets ou les certificats, peuvent bénéficier de l'API Aggregation pour offrir une API sécurisée et isolée pour la gestion de ces ressources.

- **Exemple** : Un serveur API agrégé pourrait offrir une API pour la gestion de certificats SSL avec des endpoints pour créer, renouveler ou révoquer des certificats.

API Aggregation

c) Orchestration de Workflows CI/CD :

Pour des workflows complexes dans des pipelines CI/CD, l'API Aggregation permet de créer des APIs spécifiques pour gérer les étapes d'intégration continue, les tests et les déploiements.

- **Exemple** : Un serveur API agrégé pourrait offrir une API pour orchestrer des pipelines CI/CD, gérer des jobs spécifiques (`/apis/cicd/v1/`) et offrir des fonctionnalités pour planifier ou surveiller chaque étape du pipeline.

API Aggregation

6. Sécurité et gestion des accès avec RBAC (Role-Based Access Control)

Lorsque on utilise l'API Aggregation, la sécurité reste un aspect crucial. Kubernetes utilise **RBAC** (Role-Based Access Control) pour gérer les autorisations et définir qui peut accéder à quelles ressources et actions dans l'API agrégée.

Exemple de gestion RBAC :

- On peut créer un **ClusterRole** pour définir quelles actions (lecture, écriture, suppression, etc.) sont autorisées pour un certain groupe d'utilisateurs sur les nouvelles APIs qu'on expose.
- Un **ClusterRoleBinding** permet de lier ces permissions à un utilisateur ou un groupe spécifique.

Exemple : Si on veut permettre à un utilisateur de lire les métriques d'un serveur API agrégé de monitoring, on peut créer un **ClusterRole** pour donner les permissions de `get` et `list` sur les métriques, puis lier ce rôle à l'utilisateur via un **ClusterRoleBinding**.

Scheduler Extender

Un Scheduler Extender dans Kubernetes est un mécanisme permettant d'étendre les fonctionnalités de l'ordonnanceur par défaut de Kubernetes, aussi appelé kube-scheduler. L'ordonnanceur est responsable de la sélection des nœuds sur lesquels les pods doivent être déployés en fonction de diverses contraintes et stratégies. Cependant, les besoins spécifiques d'un cluster peuvent nécessiter des règles d'ordonnancement personnalisées que le kube-scheduler par défaut ne prend pas en charge. C'est là qu'un Scheduler Extender entre en jeu.

1. **Filtrage personnalisé**:
   - Permet d'ajouter des règles de filtrage supplémentaires pour déterminer quels nœuds sont éligibles pour héberger un pod.
   - Par exemple, vous pouvez filtrer les nœuds en fonction de critères spécifiques comme la latence réseau, les politiques de sécurité, ou des contraintes métiers particulières.
2. **Priorisation personnalisée**:
   - Permet de définir des règles de priorisation supplémentaires pour classer les nœuds éligibles en fonction de leur adéquation pour héberger un pod.
   - Par exemple, vous pouvez prioriser les nœuds en fonction de la proximité géographique, des coûts énergétiques, ou des préférences de l'application.

Architecture d'un Scheduler Extender

Un Scheduler Extender est typiquement un service HTTP qui expose des API spécifiques que le kube-scheduler peut appeler. Ces API sont principalement `filter` et `prioritize`.

1. **API de filtrage (`filter`)**:
   - Cette API est appelée par le kube-scheduler pour filtrer la liste des nœuds candidats.
   - Le Scheduler Extender reçoit une liste de nœuds candidats et renvoie une liste réduite de nœuds qui répondent aux critères de filtrage personnalisés.
2. **API de priorisation (`prioritize`)**:
   - Cette API est appelée par le kube-scheduler pour classer les nœuds candidats restants après le filtrage.
   - Le Scheduler Extender reçoit une liste de nœuds et attribue un score à chaque nœud, indiquant leur adéquation relative pour exécuter le pod.

Helm

Helm est un gestionnaire de paquets pour Kubernetes qui permet de définir, installer et mettre à jour des applications Kubernetes à l'aide de "charts". Un chart est un package préconfiguré contenant les configurations nécessaires pour déployer une application.

Fonctionnalités principales :

- **Gestion des versions** : Facilite les mises à jour et les retours en arrière des déploiements.
- **Modularité et réutilisabilité** : Les charts peuvent être réutilisés et partagés.
- **Automatisation** : Simplifie le déploiement et la mise à jour des applications Kubernetes.

Helm

Installation de Helm et création d'un Chart

Installation de Helm :

```
curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash
helm version
```

Création d'un nouveau chart :

```
helm create mychart
```

Exploration du contenu du chart :

```
cd mychart
tree
```

App/lib charts, subcharts, dependencies

App Charts

Un chart d'application contient les fichiers nécessaires pour déployer une application sur Kubernetes.

```
mychart/
  Chart.yaml
  values.yaml
  charts/
  templates/
    deployment.yaml
    service.yaml
    _helpers.tpl
```

App/lib charts, subcharts, dependencies

App Charts

Lib Charts: Les charts de bibliothèque sont utilisés comme dépendances pour d'autres charts. Ils ne contiennent pas de ressources Kubernetes directes mais fournissent des templates partagés.
Subcharts et Dependencies: Les subcharts et les dépendances sont définis dans le fichier `Chart.yaml` d'un chart principal.
Définir la dépendance dans `Chart.yaml` :

```
dependencies:
  - name: mysql
    version: "1.6.7"
    repository: "https://charts.helm.sh/stable"
```

Mettre à jour les dépendances :

```
helm dependency update mychart
```

Vérification des dépendances téléchargées :

```
ls charts/
```

Pre & post actions/hooks

Les hooks Helm permettent d'exécuter des actions spécifiques avant ou après certains événements du cycle de vie des déploiements.

Exemple de hook `pre-install` :

Ajoutez un fichier `pre-install-job.yaml` dans le répertoire `templates` :

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: "{{ .Release.Name }}-pre-install"
  annotations:
    "helm.sh/hook": pre-install
spec:
  template:
    spec:
      containers:
      - name: pre-install-job
        image: busybox
        command: ['sh', '-c', 'echo Hello, World!']
      restartPolicy: Never
```

Déployer le chart avec le hook :

```
helm install my-release mychart
```

Tester une chart

Helm permet de définir des tests pour vérifier que le déploiement s'est effectué correctement.

Ajouter un fichier de test :

Créez un fichier `test-connection.yaml` dans le répertoire `templates` :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: "{{ .Release.Name }}-test-connection"
  annotations:
    "helm.sh/hook": test
spec:
  containers:
  - name: curl
    image: curlimages/curl
    command: ['curl']
    args: ['{{ .Release.Name }}:{{ .Values.service.port }}']
  restartPolicy: Never
```

Exécuter le test :

```
helm test my-release
```

Troubleshooting Helm

- Utiliser `--debug` pour le dépannage :

```
helm install my-release mychart --debug --dry-run
```

- Inspecter les ressources Kubernetes :

```
kubectl get all -l release=my-release
kubectl describe pod <pod-name>
kubectl logs <pod-name>
```

- Consulter l'historique et effectuer un rollback :

```
helm history my-release
helm rollback my-release <revision>
```

- Utiliser `helm template` pour examiner les templates rendus :

```
helm template mychart
```

TP Helm

Sujet de TP : Déploiement d'un Admission Controller avec Helm

Objectif :

L'objectif de ce TP est de créer un Admission Controller personnalisé pour Kubernetes et de le déployer à l'aide d'un chart Helm. Un Admission Controller est un composant de Kubernetes qui intercepte les requêtes API vers le serveur API avant que les objets ne soient persistés dans etcd.

Étapes à suivre :

1. **Création d'un Admission Controller personnalisé**
2. **Création d'un chart Helm**
3. **Déploiement sur un cluster Kubernetes**
4. **Tests et Validation :**