UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Kubernetes avancée

Sommaire

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

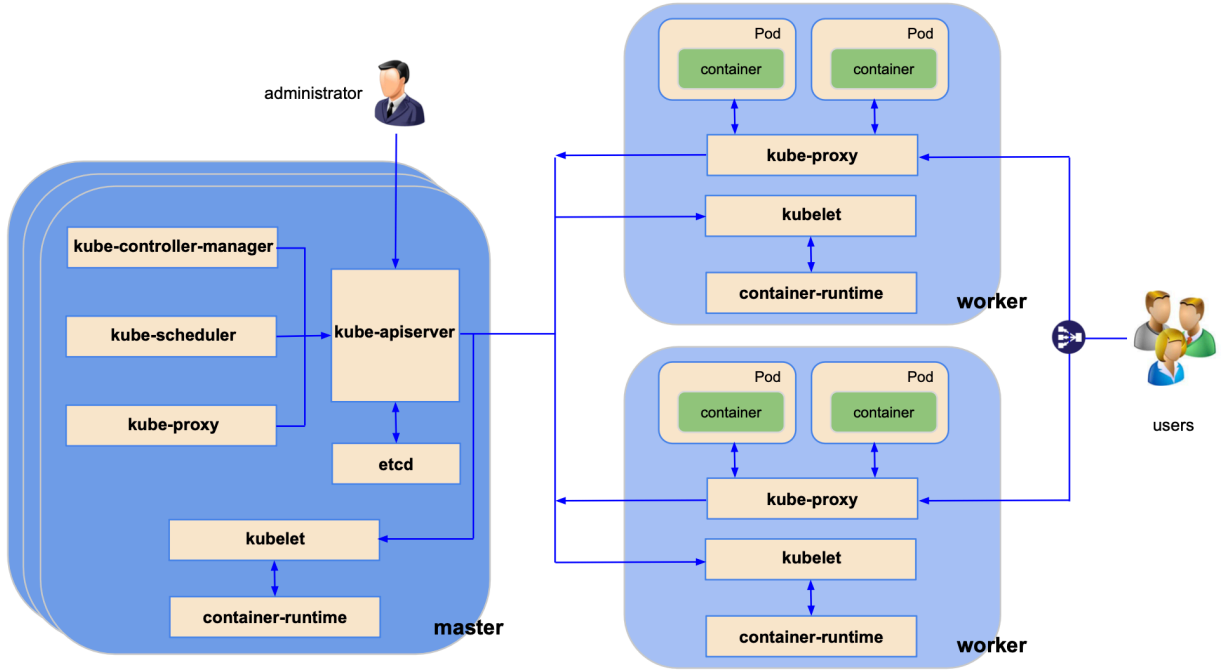
Master Node

1. **API Server** : Point d'entrée principal pour toutes les commandes de l'utilisateur. Il expose l'API Kubernetes.
2. **etcd** : Stockage clé-valeur hautement disponible utilisé comme base de données pour stocker toutes les données de configuration de Kubernetes.
3. **Scheduler** : Attribue les conteneurs aux nœuds de travail en fonction de la disponibilité des ressources et des exigences spécifiées.
4. **Controller Manager** : Exécute les contrôleurs qui surveillent l'état du cluster et apportent des modifications pour atteindre l'état souhaité (ex. : ReplicaSet, Deployment, Job).

Rappel sur Kubernetes

Composantes principales de Kubernetes

Les processus : vision d'ensemble



Rappel sur Kubernetes

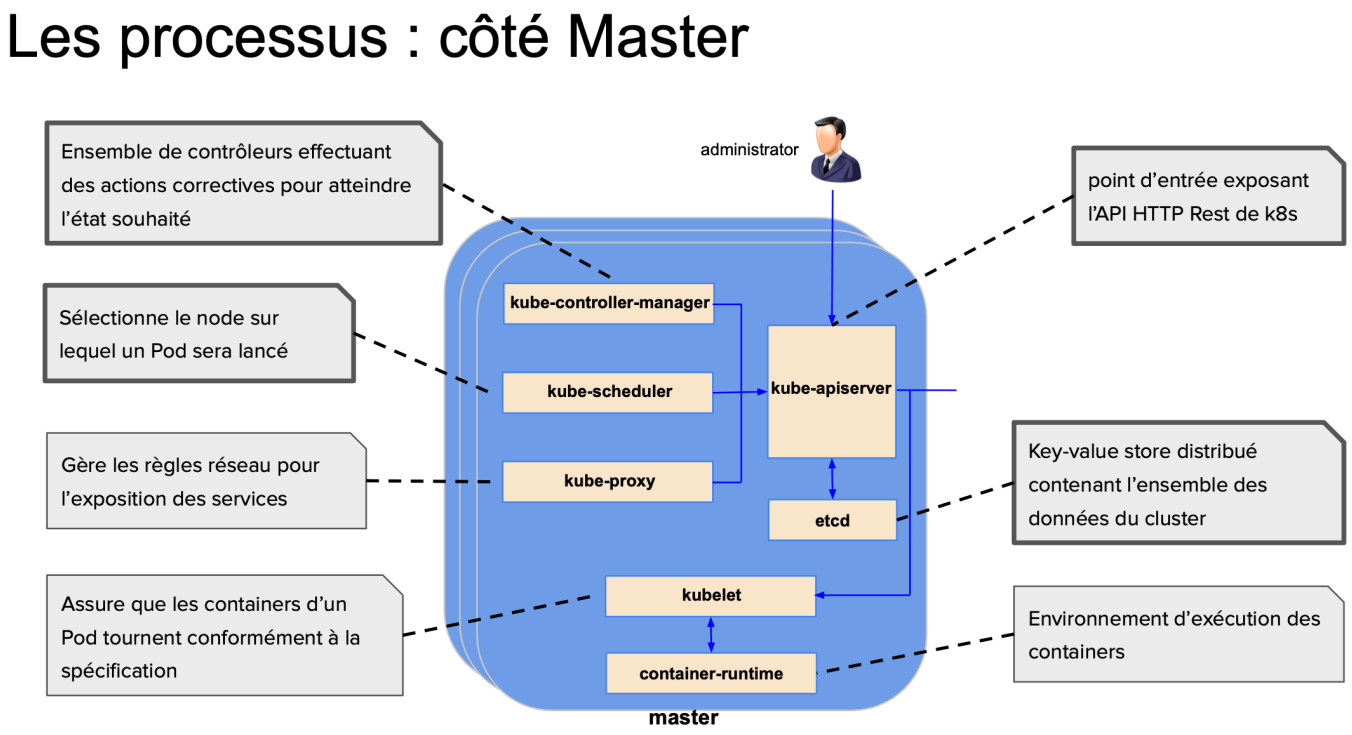
Composantes principales de Kubernetes

Worker Nodes

1. **Kubelet** : Agent qui s'exécute sur chaque nœud de travail et garantit que les conteneurs sont exécutés dans un pod. Il surveille les conteneurs et les maintient en état de marche.
2. **Kube-proxy** : Implémente les règles réseau sur chaque nœud pour permettre la communication entre les services et les pods.
3. **Container Runtime** : Logiciel utilisé pour exécuter les conteneurs (par exemple, Docker, containerd).

Rappel sur Kubernetes

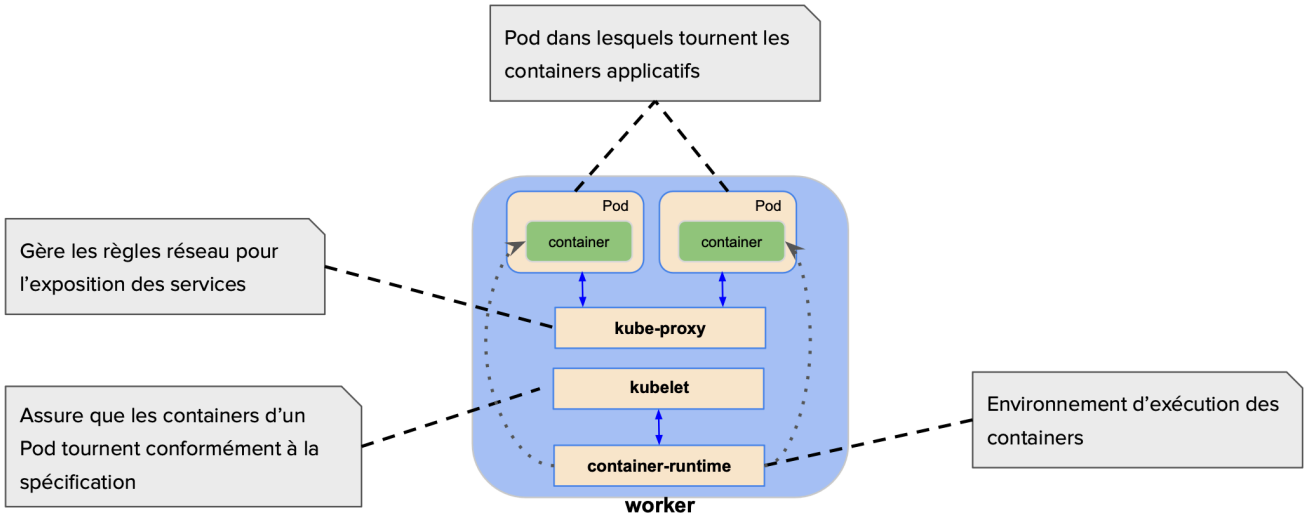
Composantes principales de Kubernetes



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

Fonctionnement global

1. **Déploiement** : Vous définissez l'état souhaité de votre application à l'aide de fichiers de configuration YAML ou JSON. Ces fichiers spécifient les pods, les services et les configurations nécessaires.
2. **API Server** : Le fichier de configuration est envoyé à l'API Server, qui valide et stocke l'état souhaité dans etcd.
3. **Scheduler** : Le Scheduler planifie les pods sur les nœuds de travail en fonction des ressources disponibles et des contraintes définies.
4. **Kubelet** : Sur chaque nœud de travail, Kubelet récupère les configurations de pod et interagit avec le runtime de conteneurs pour lancer les conteneurs spécifiés.
5. **Kube-proxy** : Configure les règles réseau pour permettre la communication entre les pods et les services.
6. **Controller Manager** : Les contrôleurs surveillent l'état du cluster et prennent des mesures correctives si nécessaire pour garantir que l'état actuel correspond à l'état souhaité.

Rappel sur Kubernetes

Concepts avancés

- **Horizontal Pod Autoscaler (HPA)** : Ajuste automatiquement le nombre de pods dans un déploiement en fonction de l'utilisation des ressources.
- **Vertical Pod Autoscaler (VPA)** : Ajuste automatiquement les ressources demandées et limitées par les pods en fonction de leur utilisation.
- **Custom Resource Definitions (CRD)** : Permettent aux utilisateurs de créer leurs propres ressources personnalisées pour étendre les capacités de Kubernetes.

Rappel sur Kubernetes

Sécurité

- **RBAC (Role-Based Access Control)** : Contrôle l'accès aux ressources du cluster en fonction des rôles attribués aux utilisateurs.
- **Network Policies** : Gèrent la sécurité réseau en définissant quelles communications sont autorisées entre les pods.

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

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

3. **CI/CD et gestion des pipelines**

Les CRDs sont utilisées pour définir des pipelines de CI/CD personnalisés. Par exemple, Tekton utilise des CRDs pour définir des tâches, des pipelines, et des ressources, permettant ainsi une orchestration flexible et puissante des workflows de build et de déploiement

4. **Configuration des réseaux et de la sécurité**

Les CRDs peuvent être utilisées pour gérer des configurations réseau avancées et des politiques de sécurité. Par exemple, des opérateurs de réseau comme Calico utilisent des CRDs pour définir des politiques de réseau, des profils de sécurité, et des configurations IP.

Custom Resource Definition

Cas d'utilisation typiques

5. **Gestion des clusters Kubernetes**

Des outils comme kubeadm et des solutions de gestion de clusters Kubernetes comme Kubeflow utilisent des CRDs pour orchestrer et gérer des clusters Kubernetes eux-mêmes, permettant une gestion automatisée et déclarative des infrastructures de clusters.

6. **Surveillance et observabilité**

Les CRDs sont utilisées pour intégrer des solutions de surveillance et de journalisation. Par exemple, Prometheus Operator utilise des CRDs pour définir des ressources telles que `ServiceMonitor` et `Prometheus`, facilitant ainsi la configuration et la gestion des capacités de surveillance.

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:
     - `applicationName` (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

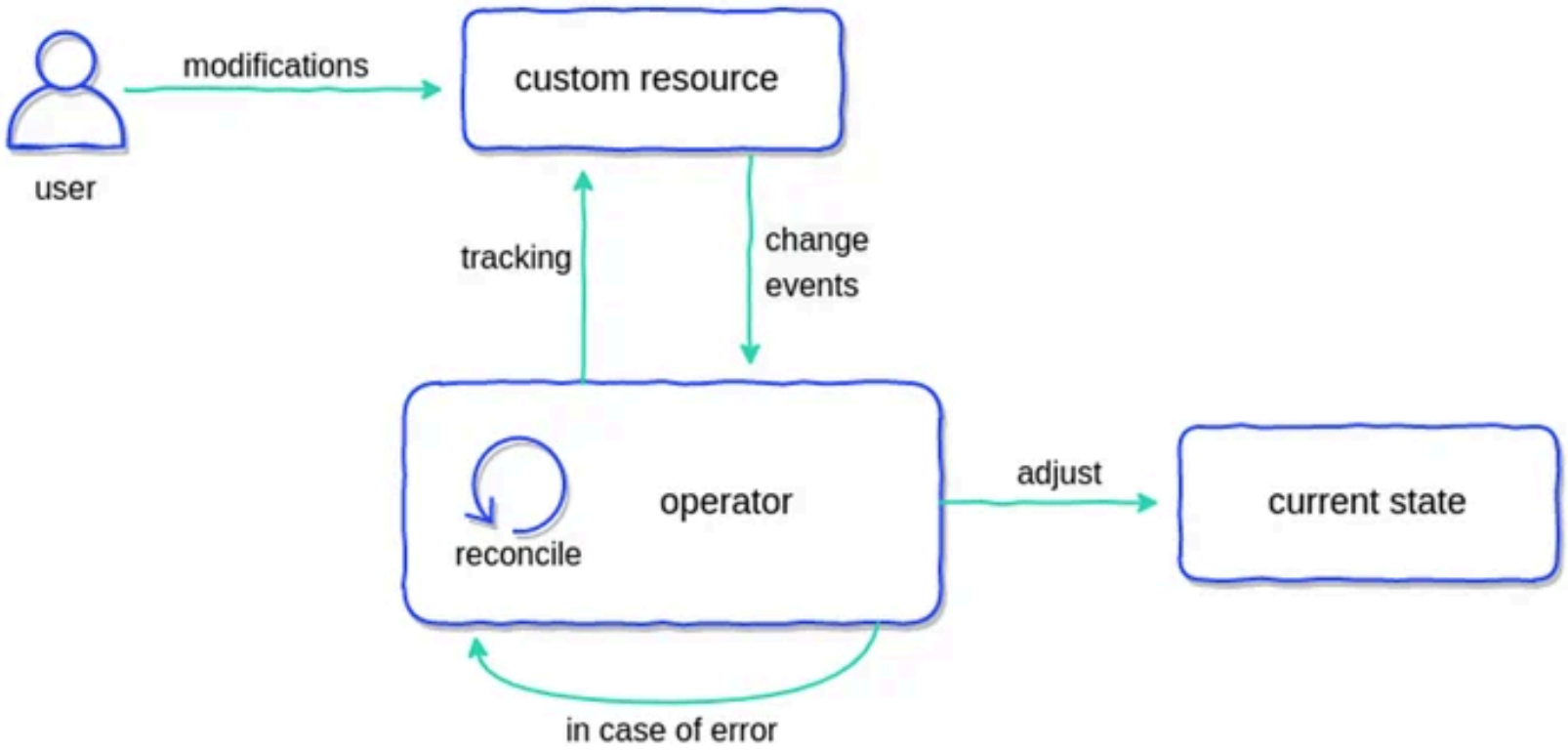
1. **CRD (Custom Resource Definition)** :
   - Les CRD permettent de définir de nouvelles ressources Kubernetes spécifiques à une application. Par exemple, au lieu d'utiliser seulement des ressources comme des Pods, Services, ou Deployments, un opérateur peut définir des ressources personnalisées comme `MySQLCluster`, `RedisInstance`, etc.
2. **Controller** :
   - Le controller est un composant qui surveille les objets définis par les CRD et veille à ce que l'état actuel du cluster corresponde à l'état désiré. Si une divergence est détectée, le controller prend des actions correctives.

Opérateurs Kubernetes

Concept clés

3. **Reconcilier boucle** :
    - La boucle de réconciliation est une partie du controller qui vérifie continuellement l'état des ressources et prend des mesures pour atteindre l'état souhaité, comme spécifié par les CRD.
4. **Spec et status** :
    - `Spec` définit l'état désiré de la ressource, tandis que `Status` reflète l'état actuel. L'opérateur utilise ces informations pour prendre des décisions de gestion.

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. **Déployer l'Opérateur sur Kubernetes**
8. **Créer et Gérer les Custom Resources**
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.

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. **PodSecurityPolicy** : Gère la politique de sécurité des pods, contrôlant ce qu'un pod peut et ne peut pas faire, par exemple, l'utilisation de volumes hostPath ou de privilèges escaladés.
6. **NodeRestriction** : Restreint les modifications que les kubelets (agents de nœuds) peuvent apporter aux ressources des nœuds et des pods.

TP Admission Controle

En executant le contenu pod-security-policy.yaml, il faudra faire en sorte de deployer l'application vote avec les manifests qui vous ont été transmis.

Il faudra probablement effectuer des modifications pour que le tout puisse fonctionner.

N'hésitez pas à utiliser les outils de debug pour vous aider à comprendre les points bloquants.

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 `department` 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

Definition

- **L'Aggregation layer** est une méthode permettant d'étendre l'API Kubernetes en ajoutant des API servers supplémentaires pour fournir des fonctionnalités supplémentaires ou des API personnalisées sans modifier le code de l'API server principal de Kubernetes.
- Ces **API servers** supplémentaires sont appelés des "aggregated API servers" et sont intégrés au cluster via un proxy intégré dans le kube-apiserver.

API Aggregation

Avantages de l'aggregation layer

1. **Extensibilité :** Permet l'ajout de nouvelles fonctionnalités ou d'API personnalisées sans modifier le kube-apiserver principal.
2. **Modularité :** Les nouvelles fonctionnalités peuvent être développées, testées et déployées indépendamment du cycle de vie du kube-apiserver principal.
3. **Isolation :** Les aggregated API servers peuvent fonctionner indépendamment, limitant les impacts potentiels sur l'API server principal en cas de problème.

API Aggregation

Fonctionnement de l'aggregation layer

1. **Deployment des Aggregated API Servers :** Ces API servers sont déployés comme des pods normaux dans le cluster Kubernetes.
2. **Enregistrement des API :** Les aggregated API servers enregistrent leurs API auprès du kube-apiserver en utilisant des ressources APIService.
3. **Proxying des requêtes :** Le kube-apiserver agit comme un proxy pour les requêtes destinées aux aggregated API servers. Lorsqu'une requête pour une API étendue arrive, le kube-apiserver la redirige vers l'API server approprié.

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.