Kubernetes

Labels, Annotations, Ingress et Volumes

Synthèse des concepts fondamentaux

Formation : BUT Informatique - Semestre 5

Module : R5.09 - Virtualisation Avancée

16 novembre 2025

Table des matières

1 Introduction

Ce cours magistral constitue une synthèse des concepts fondamentaux de Kubernetes que vous devez maîtriser parfaitement. Nous allons consolider vos connaissances acquises lors des TD/TP en clarifiant trois axes essentiels :

1. **Labels et Annotations** : Comprendre la différence et leur utilisation stratégique
2. **Ingress, Services et Pods** : Maîtriser le flux complet du trafic réseau
3. **Volumes** : Distinguer persistance et non-persistance des données

Objectifs pédagogiques

À l'issue de ce CM, vous serez capables de :

- Distinguer labels et annotations et utiliser une nomenclature cohérente
- Expliquer le rôle de chaque composant du flux réseau (Ingress Controller, Ingress, Service, Pod)
- Différencier les volumes éphémères des volumes persistants (PV/PVC)
- Implémenter correctement ces concepts dans vos déploiements Kubernetes

2 Labels et Annotations

2.1 Présentation générale

Les labels et annotations sont deux mécanismes de métadonnées dans Kubernetes, mais ils ont des rôles fondamentalement différents.

Point important

Règle d'or :

— **Labels** → Sélection et comportement Kubernetes

— **Annotations** → Métadonnées techniques sans impact sur la sélection

2.2 Les Labels (Étiquettes)

2.2.1 Définition et caractéristiques

Les labels sont des paires clé-valeur attachées aux objets Kubernetes (pods, services, deployments...) qui servent à :

— **Sélectionner** des ensembles d'objets via les selectors

— **Organiser** et regrouper les ressources

— **Influencer** le comportement des contrôleurs Kubernetes

Contraintes techniques :

— Clé : max 63 caractères (préfixe optionnel de 253 caractères)

— Valeur : max 63 caractères

— Format : caractères alphanumériques, tirets, underscores, points

2.2.2 Exemples pratiques

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
  labels:
    app: webapp
    tier: frontend
    environment: production
    version: v1.2.0
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
      tier: frontend
  template:
    metadata:
      labels:
        app: webapp
        tier: frontend
        environment: production
```

Listing 1 – Définition de labels dans un Deployment

Exemple pratique

Utilisation des selectors :
Les Services utilisent les labels pour identifier leurs pods cibles :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  selector:
    app: webapp          # S lectionne tous les pods avec ce label
    tier: frontend       # ET ce label
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Résultat : Le service route le trafic vers tous les pods ayant `app: webapp` **ET** `tier: frontend`.

2.2.3 Nomenclature recommandée

Point d'attention

Une nomenclature incohérente peut entraîner :

- Des **collisions** entre sélecteurs
- Des **règles ambiguës** dans les Network Policies
- Des difficultés de **maintenance** et debugging

Labels recommandés (Kubernetes standard) :

Label	Exemple	Description
`app.kubernetes.io/name`	`mysql`	Nom de l'application
`app.kubernetes.io/instance`	`mysql-prod`	Instance spécifique
`app.kubernetes.io/version`	`5.7.21`	Version de l'application
`app.kubernetes.io/component`	`database`	Rôle dans l'architecture
`app.kubernetes.io/part-of`	`wordpress`	Système parent
`app.kubernetes.io/managed-by`	`helm`	Outil de gestion

TABLE 1 – Labels recommandés par Kubernetes

2.3 Les Annotations

2.3.1 Définition et usage

Les annotations sont des métadonnées qui **ne servent PAS** à la sélection. Elles sont utilisées pour :

— Stocker des informations techniques
— Configurer des outils externes (monitoring, CI/CD)
— Documenter les ressources
— Transmettre des données aux contrôleurs

Différences avec les labels :

— Aucune limite de taille stricte
— Peuvent contenir des données structurées (JSON, YAML)
— **Ne participent PAS** à la sélection d'objets

2.3.2 Exemples d'utilisation

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: webapp-ingress
  annotations:
    # Configuration pour l'Ingress Controller
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/rate-limit: "100"

    # M tadonn es pour les outils externes
    prometheus.io/scrape: "true"
    prometheus.io/port: "9090"

    # Documentation
    description: "Ingress pour l'application web principale"
    maintainer: "equipe-devops@iut.fr"
spec:
  rules:
    - host: webapp.iut.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: webapp-service
                port:
                  number: 80
```

Listing 2 – Annotations dans un Ingress

Point important

Annotations critiques pour Ingress-nginx :

— `nginx.ingress.kubernetes.io/rewrite-target` : Réécriture d'URL
— `nginx.ingress.kubernetes.io/ssl-redirect` : Redirection HTTPS
— `nginx.ingress.kubernetes.io/auth-type` : Authentification basique
— `nginx.ingress.kubernetes.io/rate-limit` : Limitation de débit

2.4 Tableau récapitulatif

Critère	**Labels**	**Annotations**
Usage principal	Sélection et organisation	Métadonnées techniques
Sélection possible ?	**OUI** (via selectors)	**NON**
Impact sur Kubernetes	Influence le comportement	Aucun impact direct
Taille	Limitée (63 caractères)	Flexible
Exemple	`app: frontend`	`description: "Service web"`

TABLE 2 – Labels vs Annotations

3 Ingress, Services et Pods - Synthèse critique

3.1 Vue d'ensemble du flux réseau

Point important

Point clé à retenir : Le flux de trafic dans Kubernetes suit toujours cette chaîne :

$$Client externe \rightarrow IngressController \rightarrow Ingress(règles) \rightarrow Service \rightarrow Pods$$

3.2 Distinction fondamentale : Ingress vs Ingress Controller

3.2.1 Objet Ingress (règles de routage)

L'Ingress est une **ressource Kubernetes** qui décrit les règles de routage HTTP/HTTPS :

— Définit les **règles** (host, path, backend)

— Est un simple fichier YAML déclaratif

— **Ne fait rien seul** sans Ingress Controller

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
    - host: app.iut.local
      http:
        paths:
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: api-service
                port:
                  number: 8080
          - path: /web
            pathType: Prefix
            backend:
              service:
                name: web-service
                port:
                  number: 80
```

Listing 3 – Exemple d'objet Ingress

3.2.2 Ingress Controller (implémentation)

L'Ingress Controller est un **composant logiciel** (pod) qui implémente réellement les règles :

— C'est un **reverse proxy** (nginx, Traefik, HAProxy, etc.)

— Surveille les objets Ingress et configure dynamiquement le proxy

— Gère le trafic entrant et applique les règles de routage

Contrôleurs populaires :

— `ingress-nginx` : Basé sur Nginx (le plus utilisé)

— `Traefik` : Moderne, avec tableau de bord

— `HAProxy Ingress` : Performant pour la charge élevée

— `Contour` : Basé sur Envoy proxy

Point d'attention

Erreur fréquente des étudiants :
Créer un objet Ingress sans avoir déployé d'Ingress Controller ne sert à rien ! Les règles existent mais ne sont pas appliquées.
Vérification :

```
kubectl get pods -n ingress-nginx
# Doit afficher un pod ingress-nginx-controller
```

3.3 Le rôle des Services

Les Services sont l'abstraction qui permet de cibler un ensemble de pods via des labels :

1. **ClusterIP** (par défaut) : IP interne au cluster
2. **NodePort** : Expose un port sur chaque nœud
3. **LoadBalancer** : Provisionne un load balancer externe (cloud)

```
apiVersion: v1
kind: Service
metadata:
  name: api-service
spec:
  type: ClusterIP
  selector:
    app: api
    tier: backend
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 3000
```

Listing 4 – Service ciblant des pods via labels

Fonctionnement :

— Le Service sélectionne tous les pods avec `app: api` et `tier: backend`

— Kubernetes crée des **Endpoints** (liste d'IPs de pods)

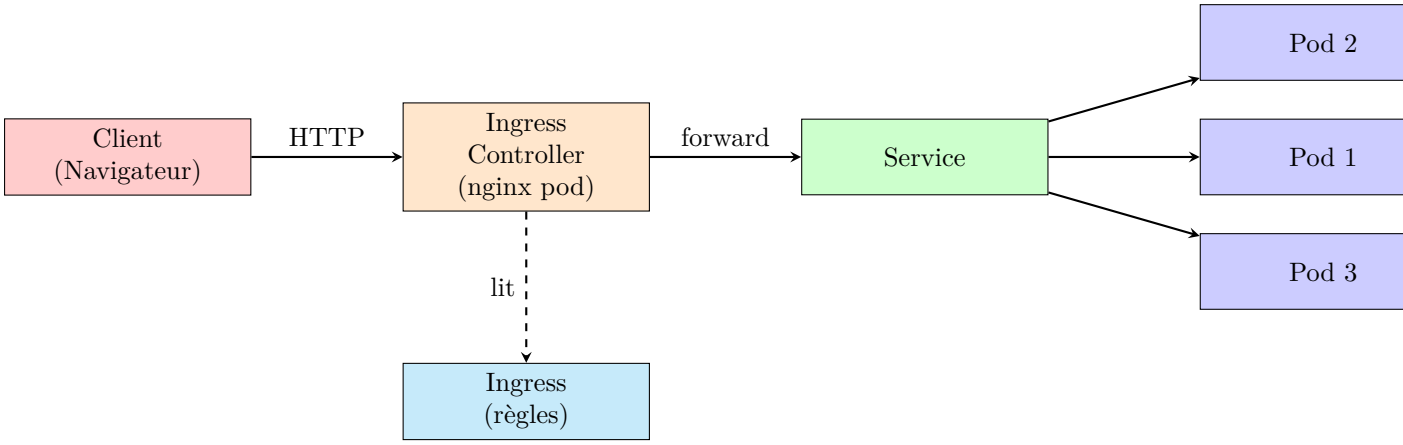
— Le trafic est réparti entre les pods via load balancing (round-robin)

3.4 Flux complet illustré

Démonstration

Scénario : Un utilisateur accède à `http://app.iut.local/api/users`

1. **Requête DNS** : L'utilisateur résout `app.iut.local` vers l'IP du cluster
2. **Ingress Controller** : Reçoit la requête HTTP
3. **Ingress (règles)** : Le contrôleur consulte l'objet Ingress et trouve la règle :
   - Host : `app.iut.local`
   - Path : `/api`
   - Backend : `api-service:8080`
4. **Service** : Le contrôleur forward la requête au Service `api-service`
5. **Endpoints** : Le Service consulte ses Endpoints (liste des IPs de pods)
6. **Pod** : Le trafic est routé vers un pod de l'API (load balancing)
7. **Réponse** : Le pod traite la requête et renvoie la réponse via le même chemin



3.5 Gestion du trafic sortant (Egress)

3.5.1 NetworkPolicy pour Egress

Les **NetworkPolicy** permettent de contrôler le trafic sortant (Egress) des pods :

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-api-egress
spec:
  podSelector:
    matchLabels:
      app: frontend
  policyTypes:
    - Egress
  egress:
    # Autoriser uniquement le trafic vers le backend
    - to:
```

```
    - podSelector:
        matchLabels:
          app: backend
    ports:
      - protocol: TCP
        port: 8080
  # Autoriser DNS
  - to:
    - namespaceSelector:
        matchLabels:
          name: kube-system
    ports:
      - protocol: UDP
        port: 53
```

Listing 5 – NetworkPolicy avec règles Egress

Cas d'usage :

— Créer une **whitelist** interne (seuls certains services peuvent être contactés)

— Bloquer l'accès à Internet depuis les pods

— Implémenter une micro-segmentation réseau

Point d'attention

Les NetworkPolicy nécessitent un **CNI compatible** (Calico, Cilium, Weave Net). Le CNI par défaut de Kind/Minikube ne supporte pas toujours les NetworkPolicy.

3.6 Solutions avancées (hors scope BUT)

3.6.1 Service Mesh (Istio, Linkerd)

Les Service Mesh ajoutent des fonctionnalités avancées :

— Chiffrement mTLS automatique entre services

— Telemetry et tracing distribué

— Traffic management (retry, timeout, circuit breaker)

— Canary deployments et A/B testing

Complexité : Ces solutions dépassent le niveau BUT mais sont utilisées en production dans les grandes entreprises.

4 Volumes - Persistance des Données

4.1 Introduction

Les volumes dans Kubernetes résolvent le problème de la **persistance des données** au-delà du cycle de vie d'un pod.

Point important

Concept fondamental :
Les conteneurs sont **éphémères** : quand un pod est détruit, toutes ses données sont perdues. Les volumes permettent de conserver les données.

4.2 Déclaration vs Montage

4.2.1 Deux étapes distinctes

1. **Déclaration du volume** dans `spec.volumes` (au niveau du pod)
2. **Montage du volume** dans `volumeMounts` (au niveau du conteneur)

Exemple pratique

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: webapp-pod
spec:
  # 1. DÉCLARATION du volume au niveau du pod
  volumes:
    - name: config-volume
      configMap:
        name: app-config
    - name: data-volume
      emptyDir: {}

  containers:
    - name: webapp
      image: nginx:latest
      # 2. MONTAGE du volume dans le conteneur
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
          readOnly: true
        - name: data-volume
          mountPath: /var/data
```

Listing 6 – Déclaration et montage d'un volume

4.3 Volumes non persistants (éphémères)

4.3.1 EmptyDir

Caractéristiques :

— Créé lorsque le pod est assigné à un nœud
— Supprimé lorsque le pod est détruit
— Partagé entre tous les conteneurs du pod

Cas d'usage :

— Cache temporaire
— Fichiers de travail entre conteneurs
— Checkpoint pour longues computations

```yaml
spec:
  volumes:
    - name: cache-volume
      emptyDir:
        sizeLimit: 1Gi
  containers:
    - name: app
      volumeMounts:
        - name: cache-volume
          mountPath: /app/cache
```

Listing 7 – EmptyDir pour cache partagé

4.3.2 ConfigMap et Secret (lecture seule)

Rôle :

— **ConfigMap** : Configuration non sensible (fichiers de config, variables)
— **Secret** : Données sensibles (mots de passe, certificats, tokens)

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  nginx.conf: |
    server {
        listen 80;
        server_name localhost;
        location / {
            root /usr/share/nginx/html;
        }
    }
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  volumes:
    - name: config
      configMap:
        name: nginx-config
  containers:
    - name: nginx
```

```
      image: nginx:latest
      volumeMounts:
        - name: config
          mountPath: /etc/nginx/nginx.conf
          subPath: nginx.conf
          readOnly: true
```

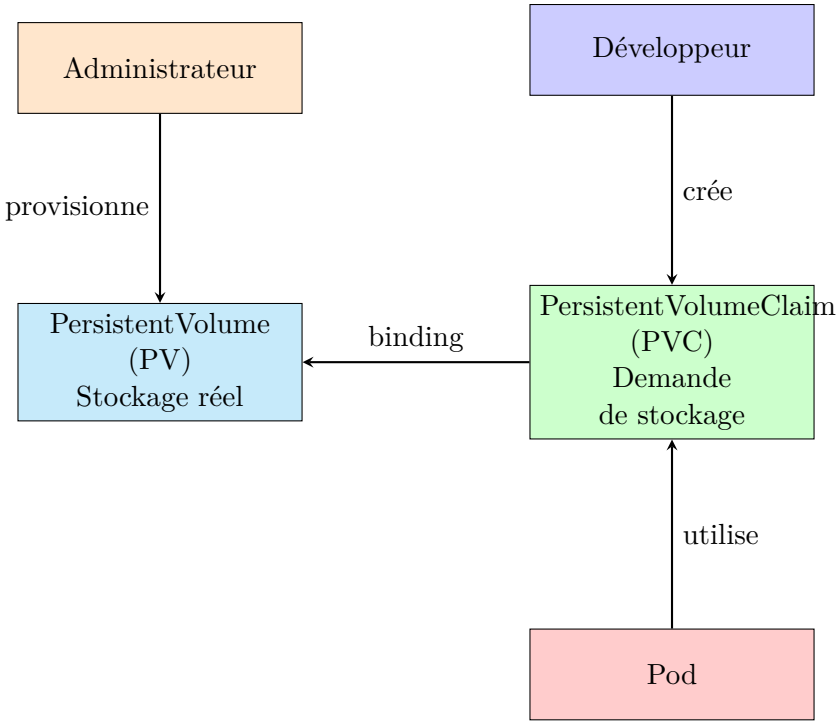
Listing 8 – ConfigMap monté en volume

Point important

Les ConfigMap et Secret sont montés en **lecture seule** par défaut. Ils sont liés au cycle de vie du pod (éphémères).

4.4 Volumes persistants (PV/PVC)

4.4.1 Architecture PV/PVC



Séparation des responsabilités :

- **Administrateur** : Provisionne les PV (stockage réel)
- **Développeur** : Crée des PVC (demandes de stockage)
- Kubernetes fait le **binding** automatique entre PVC et PV

4.4.2 PersistentVolume (PV)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-database
```

```
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: manual
  hostPath:
    path: /mnt/data
```

Listing 9 – Exemple de PersistentVolume

Access Modes :

- `ReadWriteOnce (RWO)` : Lecture/écriture par un seul nœud
- `ReadOnlyMany (ROX)` : Lecture seule par plusieurs nœuds
- `ReadWriteMany (RWX)` : Lecture/écriture par plusieurs nœuds

Reclaim Policy :

- `Retain` : Les données sont conservées après suppression du PVC
- `Delete` : Le volume est supprimé automatiquement
- `Recycle` : Les données sont effacées mais le volume est recyclé (déprécié)

4.4.3 PersistentVolumeClaim (PVC)

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-database
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: manual
```

Listing 10 – PersistentVolumeClaim

4.4.4 Utilisation dans un Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: postgres-pod
spec:
  volumes:
    - name: postgres-storage
      persistentVolumeClaim:
        claimName: pvc-database
  containers:
    - name: postgres
      image: postgres:14
      volumeMounts:
```

```
14          - name: postgres-storage
15            mountPath: /var/lib/postgresql/data
16        env:
17          - name: POSTGRES_PASSWORD
18            value: "secret"
```

Listing 11 – Pod utilisant un PVC

Point important

Persistance garantie :
Même si le pod `postgres-pod` est détruit et recréé, les données dans `/var/lib/postgresql/data` sont conservées car elles résident sur le PV.

4.5 Tableau comparatif

Type	Cycle de vie	Cas d'usage	Persistance
`emptyDir`	Lié au pod	Cache, fichiers temporaires	Non
`configMap`	Lié au ConfigMap	Configuration applicative	Non
`secret`	Lié au Secret	Credentials, certificats	Non
`hostPath`	Lié au nœud	Tests locaux (Kind/Minikube)	Oui (sur le nœud)
`PV/PVC`	Découplé du pod	Bases de données, fichiers	**Oui**

TABLE 3 – Types de volumes et persistance

4.6 Scénarios stateful vs stateless

Exemple pratique

Application stateless (frontend web) :

— Utilise `ConfigMap` pour la configuration Nginx
— Utilise `emptyDir` pour des logs temporaires
— Aucun PVC nécessaire

Application stateful (base de données) :

— Utilise un `PVC` pour les données (`/var/lib/postgresql/data`)
— Utilise `Secret` pour le mot de passe root
— StatefulSet pour garantir l'identité stable

5 Démonstration pratique

5.1 Application front + back pour illustration

Dans le chapitre 3 du dépôt GitLab que vous avez, vous trouverez une application complète **frontend + backend** qui illustre :

1. **Load balancing** : Plusieurs réplicas du backend distribuent la charge
2. **High Availability (HA)** : Si un pod tombe, le Service redirige le trafic
3. **Health checks** : Liveness et Readiness probes pour la résilience
4. **Ingress** : Routage basé sur le path (`/api` → backend, `/` → frontend)

Démonstration

Étapes de la démo live :

1. Déployer le backend avec 3 réplicas
2. Déployer le frontend avec 2 réplicas
3. Créer les Services ClusterIP pour chaque composant
4. Configurer un Ingress pour router le trafic
5. Tester le load balancing en tuant des pods
6. Observer la récupération automatique via les ReplicaSets

Commandes de test :

```
# Observer les pods et leur distribution
kubectl get pods -o wide -w

# Tuer un pod pour voir la recreation
kubectl delete pod <pod-name>

# Tester le load balancing
while true; do curl http://app.iut.local/api/health; sleep 1; done

# Observer les endpoints du Service
kubectl get endpoints backend-service
```

5.2 Concepts illustrés

5.2.1 Load Balancing

Le Service distribue les requêtes entre les pods selon un algorithme round-robin :

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - port: 8080
```

```
10      targetPort: 3000
```

Listing 12 – Service avec sélection de plusieurs pods

Résultat : Les requêtes sont distribuées équitablement entre les 3 pods backend.

5.2.2 High Availability

Si un pod tombe (crash, nœud défaillant), le Deployment garantit que le nombre de réplicas est maintenu :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: api
          image: backend:v1.0.0
```

Listing 13 – Deployment avec 3 réplicas

5.2.3 Health Checks

Les probes permettent à Kubernetes de détecter les pods défaillants :

```
containers:
  - name: api
    image: backend:v1.0.0
    livenessProbe:
      httpGet:
        path: /health
        port: 3000
      initialDelaySeconds: 10
      periodSeconds: 5
    readinessProbe:
      httpGet:
        path: /ready
        port: 3000
      initialDelaySeconds: 5
      periodSeconds: 3
```

Listing 14 – Liveness et Readiness Probes

Différence :

— **Liveness** : Si échec, Kubernetes redémarre le pod

— **Readiness** : Si échec, Kubernetes retire le pod des endpoints (plus de trafic)

6 Synthèse et Points Clés

6.1 Récapitulatif Labels vs Annotations

— Les **labels** sont utilisés pour la sélection (selectors) → influencent le comportement
— Les **annotations** ne servent PAS à la sélection → métadonnées techniques
— Utilisez une **nomenclature cohérente** pour éviter les collisions

6.2 Récapitulatif Ingress / Services / Pods

— **Ingress (objet)** : Décrit les règles de routage
— **Ingress Controller** : Implémente réellement ces règles (nginx, traefik...)
— **Service** : Abstraction qui cible des pods via labels
— **Pods** : Instances de l'application
— **NetworkPolicy** : Contrôle du trafic Egress (sortant) pour micro-segmentation

6.3 Récapitulatif Volumes

— **Déclaration** dans `spec.volumes` (niveau pod)
— **Montage** dans `volumeMounts` (niveau conteneur)
— **Non persistants** : emptyDir, ConfigMap, Secret (cycle de vie lié au pod)
— **Persistants** : PV/PVC (découplés du pod, conservent les données)
— **Stateless** : Pas besoin de PVC
— **Stateful** : Nécessite PVC pour les données (BDD, fichiers)

6.4 Questions pour auto-évaluation

1. Quelle est la différence entre un label et une annotation ?
2. Que se passe-t-il si vous créez un Ingress sans déployer d'Ingress Controller ?
3. Expliquez le flux complet d'une requête HTTP de l'utilisateur jusqu'au pod.
4. Quelle est la différence entre `spec.volumes` et `volumeMounts` ?
5. Pourquoi utiliser un PVC au lieu d'un emptyDir pour une base de données ?
6. Qu'est-ce qu'une NetworkPolicy Egress et à quoi sert-elle ?
7. Comment un Service sait quels pods cibler ?

7 Pour aller plus loin

7.1 Service Mesh (Istio)

Les Service Mesh ajoutent une couche de gestion avancée du trafic :

— **mTLS automatique** : Chiffrement entre services
— **Observabilité** : Métriques, traces, logs distribués
— **Traffic management** : Retry, timeout, circuit breaker
— **Déploiements avancés** : Canary, blue/green

7.2 Gateway API (successeur d'Ingress)

La nouvelle Gateway API remplace progressivement Ingress avec plus de fonctionnalités :

— Support de TCP/UDP (pas seulement HTTP)
— Séparation des rôles (admin vs développeur)
— Routage plus avancé

7.3 StorageClass et provisionnement dynamique

Les StorageClass permettent le provisionnement automatique de volumes :

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  replication-type: regional-pd
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: database-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 50Gi
  storageClassName: fast-ssd
```

Listing 15 – StorageClass pour provisionnement dynamique

Le PV est créé automatiquement par le provisioner cloud (GCE, AWS, Azure).

7.4 Ressources complémentaires

— Documentation officielle Kubernetes : https://kubernetes.io/docs/
— Kubernetes Patterns (livre) : O'Reilly
— Tutoriels interactifs : https://www.katacoda.com/courses/kubernetes
— Certifications : CKA (Certified Kubernetes Administrator), CKAD (Developer)

Conclusion

Ce cours magistral a consolidé trois piliers essentiels de Kubernetes :

1. **Labels et Annotations** : Comprendre leur rôle distinct et adopter une nomenclature cohérente
2. **Ingress / Services / Pods** : Maîtriser le flux complet du trafic réseau et la distinction Ingress objet vs Controller
3. **Volumes** : Différencier volumes éphémères et persistants pour architecturer correctement vos applications

Ces concepts sont FONDAMENTAUX et seront évalués.

Assurez-vous de les maîtriser avant le contrôle continu !

Prochaines étapes :

— Pratiquer avec les exemples du dépôt GitLab (chapitre 3)
— Expérimenter avec Kind/Minikube sur vos machines
— Poser vos questions lors des TD/TP