TD02 - Kubernetes : Déploiement d'applications

Préparation au TP02

VERSION PROFESSEUR - CORRECTION

Maxime Lambert
BUT Informatique - Semestre 5 - Ressource R5.09

2024/2025

ATTENTION

Ce document contient les corrections des questions du TD02.
Il est destiné uniquement aux enseignants.

Table des matières

1 De Docker à Kubernetes

1.1 Pourquoi Kubernetes ?

Dans le TD01, vous avez appris à déployer des conteneurs Docker individuellement. Cependant, imaginez que vous devez :

— Déployer une application composée de 10 microservices différents

— Assurer la haute disponibilité avec plusieurs replicas de chaque service

— Gérer automatiquement les redémarrages en cas de panne

— Scaler automatiquement selon la charge

— Mettre à jour vos applications sans interruption de service

C'est là que Kubernetes intervient !

2 Introduction à kubectl

2.1 De `docker` à `kubectl`

Vous connaissez déjà les commandes Docker du TD01. Voici leur équivalent en Kubernetes :

Docker (TD01)	**Kubernetes (TD02)**
`docker ps`	`kubectl get pods`
`docker images`	`kubectl get deployments`
`docker run nginx`	`kubectl create deployment nginx -image=nginx`
`docker logs <container>`	`kubectl logs <pod>`
`docker exec -it <container> sh`	`kubectl exec -it <pod> - sh`
`docker rm <container>`	`kubectl delete pod <pod>`
`docker network ls`	`kubectl get services`
`docker volume ls`	`kubectl get pvc`

2.2 Les commandes de base

La commande `kubectl` est l'outil principal pour interagir avec un cluster Kubernetes. Elle utilise une syntaxe de type `kubectl <verbe> <objet>`, similaire à `docker <objet> <commande>` que vous avez vu dans le TD01.

Question 1

Associez chaque verbe kubectl à sa fonction :

a) `get`

b) `describe`

c) `apply`

d) `delete`

e) `logs`

Fonctions :

I. Afficher les détails complets d'une ressource

II. Supprimer une ressource

III. Appliquer une configuration depuis un fichier

IV. Lister les ressources

V. Consulter les journaux d'un conteneur

CORRECTION Question 1

Associations correctes :

— a) `get` → IV. Lister les ressources

— b) `describe` → I. Afficher les détails complets d'une ressource

— c) `apply` → III. Appliquer une configuration depuis un fichier

— d) `delete` → II. Supprimer une ressource

— e) `logs` → V. Consulter les journaux d'un conteneur

Explications complémentaires :

— `get` : Commande la plus utilisée pour lister les ressources (pods, services, deployments, etc.)

— `describe` : Donne des informations détaillées incluant les événements récents

— `apply` : Idempotent, peut être exécuté plusieurs fois (contrairement à `create`)

— `delete` : Suppression d'objets, avec gestion de la finalisation gracieuse

— `logs` : Équivalent de `docker logs`, essentiel pour le debugging

2.3 Les objets Kubernetes

Question 2

Complétez le tableau suivant avec la description de chaque objet Kubernetes :

CORRECTION Question 2

Objet	**Description**
Pod	**Plus petite unité déployable** dans Kubernetes. Contient un ou plusieurs conteneurs qui partagent le même réseau et le même stockage. Les conteneurs dans un Pod sont toujours co-localisés et co-schedulés.
Deployment	**Contrôleur déclaratif** pour les Pods et ReplicaSets. Permet de déclarer le nombre de replicas souhaités, gère les mises à jour progressives (rolling updates) et les rollbacks.
Service	**Abstraction réseau** qui définit un ensemble logique de Pods et une politique d'accès. Fournit une adresse IP stable et un nom DNS pour accéder aux Pods, même quand ils sont recréés.
Ingress	**Gestion du trafic HTTP/HTTPS externe** vers les Services. Fournit un routage basé sur les URL, l'équilibrage de charge, la terminaison SSL/TLS et le virtual hosting.
PersistentVolumeClaim	**Demande de stockage** par un utilisateur. Permet de réclamer un volume persistant avec des caractéristiques spécifiques (taille, mode d'accès, classe de stockage).
Secret	**Objet pour stocker des données sensibles** (mots de passe, tokens, clés). Encodé en base64 (non chiffré !). Peut être monté comme volume ou injecté comme variable d'environnement.

Points clés à retenir :

- Pod = unité atomique (comme un conteneur Docker, mais peut en contenir plusieurs)
- Deployment = gestion déclarative des Pods (équivalent de `docker-compose` mais plus puissant)
- Service = point d'accès stable aux Pods (résout le problème des IPs changeantes)
- Ingress = reverse proxy intelligent (comme Nginx/Traefik)
- PVC = abstraction du stockage (découple l'application du fournisseur de stockage)
- Secret = configuration sensible (à sécuriser avec des solutions comme Sealed Secrets ou Vault)

2.4 Namespaces

Question 3

1. Qu'est-ce qu'un namespace dans Kubernetes ?
2. Donnez deux exemples de namespaces système standard.
3. Pourquoi ne devrait-on pas déployer ses applications dans le namespace `default` ?
4. Quelle est la différence entre une ressource *namespaced* et une ressource *cluster-scoped* ?
5. Donnez un exemple de ressource cluster-scoped.

CORRECTION Question 3

1. Qu'est-ce qu'un namespace dans Kubernetes ?
Un namespace est un **mécanisme d'isolation logique** qui permet de diviser les ressources d'un cluster entre plusieurs utilisateurs, équipes ou environnements. C'est comme créer des "espaces de noms" virtuels au sein d'un même cluster physique.

2. Deux exemples de namespaces système standard :

- `kube-system` : Contient les composants système de Kubernetes (kube-dns, kube-proxy, metrics-server, etc.)
- `kube-public` : Namespace public lisible par tous, souvent utilisé pour des ConfigMaps publiques
- `kube-node-lease` : Contient les objets Lease pour améliorer la performance du heartbeat des nodes

3. Pourquoi éviter le namespace `default` ?
Raisons de ne pas utiliser `default` :

- **Mauvaise pratique organisationnelle** : Tout se mélange, difficile de s'y retrouver
- **Pas de séparation des environnements** : dev, staging, prod dans le même namespace
- **Risque de conflits** : Noms de ressources qui se chevauchent
- **Pas de gestion fine des quotas** : Impossible de limiter les ressources par équipe/projet
- **Sécurité réduite** : Pas de RBAC granulaire possible

Bonne pratique : Créer des namespaces par :

- Environnement : `dev`, `staging`, `production`
- Équipe : `team-frontend`, `team-backend`, `team-data`
- Application : `app-webshop`, `app-api`, `app-monitoring`

4. Différence namespaced vs cluster-scoped :

- **Ressource namespaced** : Appartient à un namespace spécifique. Plusieurs ressources du même nom peuvent exister dans différents namespaces. Exemples : Pod, Service, Deployment, ConfigMap, Secret.
- **Ressource cluster-scoped** : Unique au niveau du cluster entier. Pas de notion de namespace. Exemples : Node, PersistentVolume, Namespace, StorageClass, ClusterRole.

5. Exemple de ressource cluster-scoped :
`Node` : Représente une machine physique ou virtuelle dans le cluster. Un nœud ne peut pas appartenir à un namespace car il sert tous les namespaces.
Autres exemples : `PersistentVolume`, `StorageClass`, `ClusterRole`, `ClusterRoleBinding`, `Namespace` lui-même.
Commande pour vérifier :

```
# Lister les ressources namespaced
kubectl api-resources --namespaced=true

# Lister les ressources cluster-scoped
kubectl api-resources --namespaced=false
```

3 Architecture d'une application sur Kubernetes

3.1 Compute : Les Deployments

Question 4

En analysant le manifest ci-dessus :

1. Quelle version de l'API Kubernetes est utilisée ?
2. Combien de replicas (copies) du Pod seront créés ?
3. Quel est le rôle du champ `selector.matchLabels` ?
4. Sur quel port le conteneur écoute-t-il ?
5. Que se passerait-il si vous supprimiez manuellement un des Pods créés par ce Deployment ?

CORRECTION Question 4

1. Version de l'API utilisée :
`apps/v1` - C'est l'API stable pour les Deployments, introduite dans Kubernetes 1.9.
2. Nombre de replicas :
3 replicas seront créés (défini par `spec.replicas: 3`).
Kubernetes maintiendra toujours 3 Pods en cours d'exécution.
3. Rôle du champ `selector.matchLabels` :
Le `selector.matchLabels` permet au Deployment de **sélectionner quels Pods il doit gérer**.

- Il fait correspondre les labels définis dans `template.metadata.labels`
- Dans cet exemple : `app: mon-app`
- Le Deployment surveillera tous les Pods ayant ce label
- Il s'assurera qu'il y en a toujours exactement 3

Important : Les labels dans `selector.matchLabels` DOIVENT correspondre aux labels dans `template.metadata.labels`, sinon le Deployment sera invalide.
4. Port d'écoute du conteneur :
Le conteneur écoute sur le **port 80** (défini par `containerPort: 80`).
Note : C'est le port INTERNE au conteneur. Il n'est pas automatiquement exposé à l'extérieur du cluster.
5. Suppression manuelle d'un Pod :
Si vous supprimez manuellement un des Pods :

1. Le Deployment détecte immédiatement qu'il manque un Pod (il n'en reste que 2 au lieu de 3)
2. Le **ReplicaSet Controller** (géré par le Deployment) crée automatiquement un nouveau Pod
3. En quelques secondes, le cluster revient à l'état désiré : 3 Pods en cours d'exécution

Démonstration :

```
# Voir les pods
kubectl get pods

# Supprimer un pod
kubectl delete pod mon-app-xxxxx

# Relister imm diatement - un nouveau pod est en cr ation
kubectl get pods
# Vous verrez un nouveau pod avec un nom diff rent
```

C'est la magie de la déclaration d'état désiré ! Vous ne gérez plus des conteneurs individuellement (comme avec Docker), mais vous déclarez un état et Kubernetes le maintient automatiquement.

3.2 Storage : Les PersistentVolumeClaims

Question 5

1. Expliquez la différence entre :
   - PersistentVolume (PV)
   - PersistentVolumeClaim (PVC)
   - StorageClass
2. Pourquoi utilise-t-on un PVC plutôt que de stocker directement les données dans le conteneur ?
3. Que signifie le mode d'accès `ReadWriteOnce` ?
4. Citez deux autres modes d'accès possibles pour un volume.

CORRECTION Question 5

1. Différences entre PV, PVC et StorageClass :

Objet	Description et rôle
PersistentVolume (PV)	**Ressource de stockage physique** provisionnée par un administrateur ou dynamiquement par une StorageClass. Représente un volume réel sur le système de stockage (NFS, iSCSI, cloud provider, etc.). C'est une ressource **cluster-scoped**.
PersistentVolumeClaim (PVC)	**Demande de stockage** faite par un utilisateur/application. Spécifie la taille et le mode d'accès souhaités. Kubernetes cherche un PV correspondant et le "lie" au PVC. C'est une ressource **namespaced**.
StorageClass	**Modèle de provisionnement dynamique**. Définit les classes de stockage disponibles (SSD, HDD, cloud storage) et leur provisionneur. Permet de créer automatiquement des PV quand un PVC est créé. Ressource **cluster-scoped**.

Analogie immobilière :

- **StorageClass** = Type de construction (appartement standard, luxe, maison)
- **PersistentVolume (PV)** = Logement physique disponible
- **PersistentVolumeClaim (PVC)** = Demande de location avec critères spécifiques

Workflow typique :

1. Un développeur crée un PVC demandant 5Gi de stockage
2. Si une StorageClass existe (avec provisionnement dynamique), elle crée automatiquement un PV de 5Gi
3. Le PV est "lié" au PVC
4. Le Pod monte le PVC comme volume

2. Pourquoi utiliser un PVC au lieu du stockage conteneur ?
Rappel du TD01 exercice 8 : les données dans un conteneur sont éphémères !
Raisons d'utiliser un PVC :

- **Persistance** : Les données survivent au redémarrage/suppression du Pod
- **Découplage** : Le stockage est indépendant du cycle de vie du Pod
- **Migration** : Les Pods peuvent être déplacés entre nœuds en conservant leurs données
- **Partage** : Plusieurs Pods peuvent accéder au même volume (selon le mode d'accès)
- **Sauvegarde/Snapshots** : Facilite la mise en place de stratégies de backup
- **Performance** : Possibilité de choisir le type de stockage (SSD, HDD, etc.)

Exemple concret : Base de données PostgreSQL

- SANS PVC : Si le Pod crash, toutes les données sont perdues !
- AVEC PVC : Le nouveau Pod se reconnecte au même volume, les données sont intactes

3. Signification de `ReadWriteOnce` :
`ReadWriteOnce (RWO)` signifie :

- Le volume peut être monté en **lecture-écriture**
- Par **UN SEUL nœud à la fois**
- Plusieurs Pods sur le MÊME nœud peuvent l'utiliser simultanément
- Mais pas par des Pods sur des nœuds différents

Cas d'usage : Bases de données traditionnelles (MySQL, PostgreSQL) qui ne supportent

3.3 Networking : Services et Ingress

Question 6

1. Expliquez pourquoi on a besoin d'un Service pour accéder à des Pods.
2. Quels sont les trois principaux types de Services dans Kubernetes ?
3. Quelle est la différence entre un Service de type `ClusterIP` et `NodePort` ?
4. À quoi sert un objet Ingress ?
5. Qu'est-ce qu'un Ingress Controller ? Donnez un exemple.

CORRECTION Question 6

1. Pourquoi a-t-on besoin d'un Service ?
Problème : Les Pods dans Kubernetes sont **éphémères** :

- Ils ont des adresses IP qui changent à chaque redémarrage
- Ils peuvent être créés/détruits automatiquement par le Deployment
- Un client ne peut pas savoir quelle IP utiliser

Solution : Le Service

- Fournit une **adresse IP stable** (ClusterIP)
- Fournit un **nom DNS** stable (ex : `mon-app.default.svc.cluster.local`)
- Fait du **load balancing** automatique entre tous les Pods correspondant au selector
- Découvre automatiquement les nouveaux Pods et retire les Pods supprimés

Analogie : Un Service c'est comme le numéro de téléphone d'une entreprise :

- Vous appelez toujours le même numéro (IP stable)
- En interne, l'appel est routé vers un employé disponible (load balancing)
- Si un employé part ou arrive, le numéro ne change pas (découverte automatique)

2. Les trois principaux types de Services :

1. **ClusterIP** (par défaut) :
   - Expose le Service sur une IP interne au cluster
   - Accessible uniquement depuis l'intérieur du cluster
   - Cas d'usage : Communication inter-services (backend ↔ database)
2. **NodePort** :
   - Expose le Service sur un port de chaque nœud (30000-32767)
   - Accessible depuis l'extérieur via `<NodeIP>:<NodePort>`
   - Crée aussi un ClusterIP automatiquement
   - Cas d'usage : Accès externe simple, environnements de développement
3. **LoadBalancer** :
   - Provisionne un load balancer externe (cloud provider)
   - Obtient une IP publique externe
   - Crée aussi un NodePort et un ClusterIP automatiquement
   - Cas d'usage : Exposition de services en production sur le cloud

Bonus : ExternalName

- Mappe un Service à un nom DNS externe
- Pas de proxy, juste un alias CNAME
- Cas d'usage : Accéder à des services externes (API, bases de données hébergées)

3. Différence ClusterIP vs NodePort :

Aspect	**ClusterIP**	**NodePort**
Portée	Interne au cluster uniquement	Accessible depuis l'extérieur
IP	IP virtuelle interne	IP de chaque nœud + port
Port	Port standard (ex : 80, 3000)	Port haute numérotation (30000-32767)
Accès	`http://svc-name:80`	`http://<node-ip>:30123`
Sécurité	Plus sécurisé (pas exposé)	Moins sécurisé (exposé au monde)
Cas d'usage	Services internes (DB, cache)	Développement, tests

Exemple visuel :

4 Les variables d'environnement et les Secrets

Question 7

1. Quelle est la différence entre définir une valeur directement avec `value` et utiliser `valueFrom` ?
2. Dans quel cas devriez-vous utiliser un Secret plutôt qu'une valeur en clair ?
3. Le Secret Kubernetes utilise l'encodage base64. Est-ce un chiffrement sécurisé ? Pourquoi ?
4. Proposez une solution pour améliorer la sécurité des Secrets dans Kubernetes.

CORRECTION Question 7

1. Différence entre `value` et `valueFrom` :
Avec `value` (valeur directe) :

```
env:
- name: DATABASE_URL
  value: "postgres://db:5432/mydb"
```

- La valeur est **écrite en dur** dans le manifest
- Visible dans le YAML du Deployment
- **Problème** : Pas de séparation configuration/code
- **Problème** : Difficile de changer sans redéployer
- **Problème** : Risque de commit de secrets dans Git !

Avec `valueFrom` (référence externe) :

```
env:
- name: PASSWORD
  valueFrom:
    secretKeyRef:
      name: db-secret
      key: password
```

- La valeur provient d'une **source externe** (Secret ou ConfigMap)
- Le Deployment ne contient qu'une **référence**
- **Avantage** : Séparation configuration/déploiement
- **Avantage** : Peut changer le Secret sans modifier le Deployment
- **Avantage** : Le secret n'est jamais dans le manifeste

Autres sources possibles avec `valueFrom` :

- `configMapKeyRef` : Depuis un ConfigMap
- `fieldRef` : Depuis les métadonnées du Pod (nom, namespace, IP)
- `resourceFieldRef` : Depuis les limites de ressources

2. Quand utiliser un Secret ?
Utilisez un Secret pour :

- **Mots de passe** de bases de données
- **Tokens API** et clés d'accès
- **Certificats TLS/SSL**
- **Clés SSH**
- **Credentials OAuth**
- **Registry credentials** (pour pull des images privées)

Utilisez un ConfigMap pour :

- Configuration applicative non sensible
- Fichiers de configuration (nginx.conf, etc.)
- Variables d'environnement publiques
- Feature flags

Règle d'or : Si vous ne voudriez pas que cette valeur soit visible dans les logs ou dans Git, utilisez un Secret !

3. Base64 est-il un chiffrement sécurisé ?
NON ! Base64 n'est PAS un chiffrement !
Base64 est seulement un ENCODAGE

- Convertit des données binaires en texte ASCII
- **Facilement décodable** : `echo "cGFzc3dvcmQ=" | base64 -d` → `password`

5 Volumes et montages

Question 8

1. Expliquez le rôle de la section `volumes`.
2. Expliquez le rôle de la section `volumeMounts`.
3. Que se passe-t-il si le PVC `my-pvc` n'existe pas ?
4. Pourquoi les données stockées dans `/data` persistent-elles alors que celles stockées ailleurs dans le conteneur sont perdues ?

CORRECTION Question 8

1. Rôle de la section `volumes` :
La section `volumes` **déclare les sources de stockage** disponibles pour le Pod.

- Définit **QUOI** monter (le volume source)
- Peut référencer différents types : PVC, ConfigMap, Secret, emptyDir, hostPath, etc.
- Donne un **nom local** au volume dans le Pod
- N'effectue pas encore le montage (c'est juste une déclaration)

Types de volumes possibles :

```
volumes:
# Depuis un PVC
- name: data
  persistentVolumeClaim:
    claimName: my-pvc

# Depuis un ConfigMap
- name: config
  configMap:
    name: my-config

# Depuis un Secret
- name: credentials
  secret:
    secretName: my-secret

# Volume temporaire (vide au d marrage)
- name: cache
  emptyDir: {}

# Dossier de l'h te (dangereux !)
- name: host-data
  hostPath:
    path: /data
    type: Directory
```

2. Rôle de la section `volumeMounts` :
La section `volumeMounts` **monte effectivement les volumes dans le conteneur**.

- Définit **OÙ** monter dans le système de fichiers du conteneur
- Référence un volume déclaré dans `volumes` par son nom
- Spécifie le chemin de montage (`mountPath`)
- Peut définir des options (lecture seule, sous-chemin, etc.)

Exemple complet :

```
spec:
  containers:
  - name: app
    image: nginx
    volumeMounts:
    - name: storage-volume      # R f rence le volume "storage-volume"
      mountPath: /data           # Monte dans /data du conteneur
      readOnly: false            # Lecture- criture
    - name: config-volume
      mountPath: /etc/nginx/nginx.conf
      subPath: nginx.conf        # Monte seulement un fichier
      readOnly: true

  volumes:
  - name: storage-volume         # D claration du volume
    persistentVolumeClaim:
      claimName: my-pvc
  - name: config-volume
    configMap:
      name: nginx-config
```

6 Labels et Selectors

Question 9

Considérez le Deployment et le Service. Questions :

1. Le Service pourra-t-il router le trafic vers les Pods créés par ce Deployment ? Pourquoi ?
2. Que se passerait-il si le Service utilisait le selector `tier: web` au lieu de `app: frontend` ?
3. Proposez une commande kubectl pour lister uniquement les Pods ayant le label `app=frontend`.

CORRECTION Question 9

1. Le Service peut-il router vers les Pods ?
OUI, le Service pourra router le trafic vers les Pods créés par ce Deployment.
Explication :
Le Service utilise le selector :

```
selector:
  app: frontend
```

Les Pods créés par le Deployment ont les labels :

```
labels:
  app: frontend
  tier: web
```

Règle de matching :

- Un Pod correspond au selector si **tous les labels du selector** sont présents sur le Pod
- Le Pod peut avoir **plus de labels** que demandé par le selector
- Ici : Le Pod a `app: frontend` donc il match !

Fonctionnement :

1. Le Service cherche tous les Pods ayant `app: frontend`
2. Il trouve les 3 Pods du Deployment (ils ont tous ce label)
3. Il crée des endpoints vers ces Pods
4. Il fait du load balancing entre ces 3 Pods

Vérification :

```
# Voir les endpoints du Service
kubectl get endpoints frontend-service

# R sultat attendu : 3 adresses IP (une par Pod)
NAME               ENDPOINTS
frontend-service   10.244.0.5:80,10.244.0.6:80,10.244.0.7:80
```

2. Que se passerait-il avec selector `tier: web` ?
Si le Service utilisait :

```
selector:
  tier: web
```

Résultat : Le Service fonctionnerait AUSSI !
Pourquoi ?

- Les Pods ont le label `tier: web`
- Le selector cherche `tier: web`
- Les Pods correspondent donc au critère

MAIS... Différence importante :
Avec `app: frontend` (spécifique) :

- Sélectionne UNIQUEMENT les Pods de cette application
- Précis et sûr
- Best practice

Avec `tier: web` (générique) :

- Sélectionne **TOUS** les Pods web du namespace
- Si vous avez d'autres apps avec `tier: web`, elles seront aussi sélectionnées !
- Peut causer du trafic vers les mauvais Pods
- **Dangereux en production**

Exemple de problème :

```
# Deployment 1: Frontend
```

7 Exercice de synthèse - CORRECTION

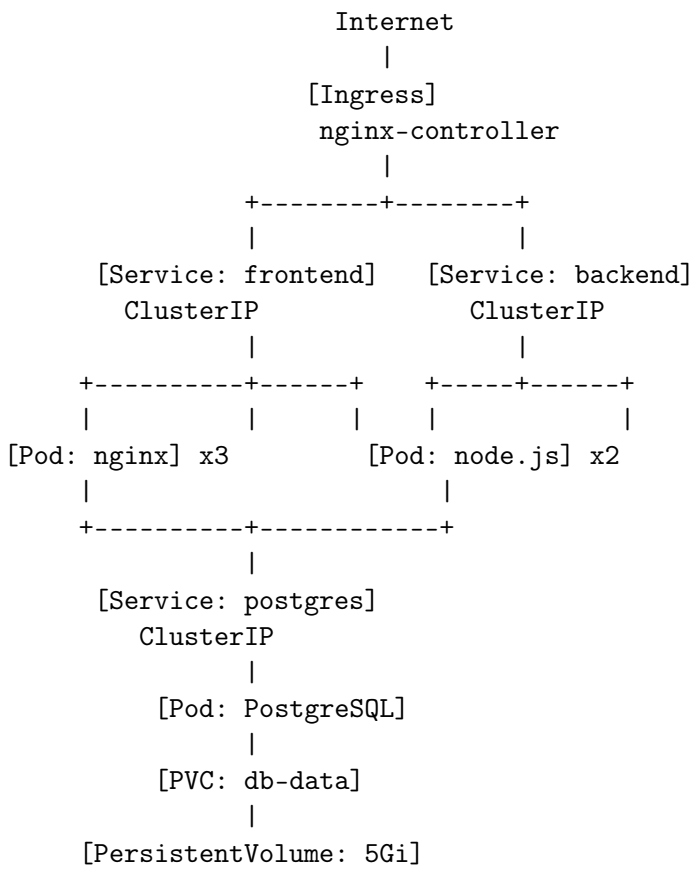
Exercice pratique sur papier

Vous devez déployer une application web simple composée de :

- Un frontend Nginx (3 replicas)
- Un backend Node.js (2 replicas)
- Une base de données PostgreSQL (1 replica avec stockage persistant)

CORRECTION Exercice de synthèse

1. Schéma de l'architecture :



2. Liste complète des objets Kubernetes nécessaires :

N°	Type	Nom
1	Namespace	web-app
2	Deployment	frontend-deployment
3	Service	frontend-service
4	Deployment	backend-deployment
5	Service	backend-service
6	StatefulSet	postgres-statefulset
7	Service	postgres-service
8	PersistentVolumeClaim	postgres-pvc
9	Secret	postgres-credentials
10	ConfigMap	backend-config
11	Ingress	web-app-ingress

3. Détail de chaque objet :

Objet 1 : Namespace

- Type : Namespace
- Nom : `web-app`
- Rôle : Isoler l'application, gérer les ressources et les permissions
- Interactions : Contient tous les autres objets

Objet 2 : Frontend Deployment

- Type : Deployment
- Nom : `frontend-deployment`
- Rôle : Gérer 3 replicas de Nginx, assurer la haute disponibilité
- Interactions :

8 Questions de réflexion pour le TP - CORRECTIONS

CORRECTIONS - Préparation au TP02

1. Comment vérifier que tous les Pods d'un Deployment sont en état Running ?

```
# M thode 1 : V rifier le statut du Deployment
kubectl get deployment <nom-deployment>
# Observer: READY 3/3 signifie 3 Pods sur 3 sont pr ts

# M thode 2 : Lister les Pods du Deployment
kubectl get pods -l app=<nom-app>

# M thode 3 : Describe le Deployment pour voir les d tails
kubectl describe deployment <nom-deployment>

# M thode 4 : Watch en temps r el
kubectl get pods -l app=<nom-app> -w
```

2. Quelle commande pour consulter les logs d'un Pod qui redémarre en boucle ?

```
# Voir les logs de l'instance actuelle
kubectl logs <nom-pod>

# IMPORTANT : Voir les logs de l'instance PRECEDENTE (crash loop)
kubectl logs <nom-pod> --previous

# Suivre les logs en temps r el
kubectl logs <nom-pod> -f

# Si plusieurs conteneurs dans le Pod
kubectl logs <nom-pod> -c <nom-conteneur> --previous
```

3. Comment accéder à une application via Ingress avec hostname mon-app.local ?

```
# 1. Modifier le fichier /etc/hosts
sudo nano /etc/hosts

# Ajouter cette ligne (remplacer IP par celle de votre cluster)
127.0.0.1  mon-app.local
# ou
<IP-NODE>  mon-app.local

# 2. Acc der via navigateur
http://mon-app.local

# 3. Ou via curl
curl http://mon-app.local
```

4. Si vous supprimez un Pod, que devient son PersistentVolume ?
Réponse : Le PersistentVolume **reste intact** et les données sont préservées.

- Le Pod est supprimé
- Le PVC reste (pas supprimé automatiquement)
- Le PV reste lié au PVC
- Les données sur le disque sont intactes
- Quand un nouveau Pod est créé, il se reconnecte au même PVC/PV
- Les données sont récupérées

5. Comment modifier le nombre de replicas d'un Deployment ?

```
# M thode 1 : kubectl scale (imp ratif)
kubectl scale deployment <nom-deployment> --replicas=5

# M thode 2 : kubectl edit (modifier le YAML)
kubectl edit deployment <nom-deployment>
# Changer spec.replicas: 5
```

9 Questions supplémentaires sur l'Ingress

Question 6 bis - Analyse d'Ingress

En analysant le manifest Ingress :

1. Quelle est la version de l'API utilisée pour l'Ingress ?
2. Quel hostname est configuré pour accéder à l'application ?
3. Vers quel Service le trafic sera-t-il routé ?
4. Que signifie `pathType: Prefix` ?
5. Comment pourriez-vous ajouter le support HTTPS à cet Ingress ?

CORRECTION Question 6 bis

1. Version de l'API utilisée :
`networking.k8s.io/v1` - C'est l'API stable pour les Ingress depuis Kubernetes 1.19.
Historique :

— Avant K8s 1.19 : `extensions/v1beta1` (obsolète)

— K8s 1.19+ : `networking.k8s.io/v1` (actuelle)

2. Hostname configuré :
`mon-app.local` - C'est le nom de domaine pour accéder à l'application.
Pour tester localement :

```
# Ajouter dans /etc/hosts
echo "127.0.0.1  mon-app.local" | sudo tee -a /etc/hosts

# Puis acc der via navigateur
curl http://mon-app.local
```

3. Routage du trafic :
Le trafic sera routé vers le Service `frontend-service` sur le port `80`.
Flux complet :

```
Utilisateur (http://mon-app.local)
    |
    v
Ingress Controller (Nginx)
    |
    v
Service: frontend-service:80
    |
    v
Pods du Deployment frontend
```

4. Signification de `pathType: Prefix` :
`pathType: Prefix` signifie que l'Ingress matchera toutes les URLs qui **commencent** par le chemin spécifié.
Types de pathType disponibles :

— **Prefix** : Correspond à tout chemin commençant par le préfixe

```
path: /api
pathType: Prefix
Matches:
    /api
    /api/users
    /api/v1/data
    /application

```

— **Exact** : Correspond exactement au chemin spécifié

```
path: /api
pathType: Exact
Matches:
    /api
    /api/            (diff rent !)
    /api/users

```

— **ImplementationSpecific** : Dépend de l'Ingress Controller

Exemple avec path : /

```
- path: /
  pathType: Prefix
  # Matche TOUTES les URLs (/, /home, /api, /anything)
```

5. Ajouter le support HTTPS :

Question 6 ter - Routage

1. Un utilisateur accède à `http://myapp.example.com/api/users`. Vers quel Service sera-t-il routé ?
2. Que se passe-t-il si l'utilisateur accède à `http://myapp.example.com/home` ?
3. Proposez un cas d'usage réel pour ce type de configuration multi-chemins.

CORRECTION Question 6 ter

1. Routage de `/api/users` :
Le trafic sera routé vers `backend-service:3000`.
Explication :

- L'URL `/api/users` commence par `/api`
- La règle `path: /api` avec `pathType: Prefix` matche
- Ingress route vers `backend-service` port 3000

Note importante : L'ordre des paths compte !
Les paths sont évalués du plus spécifique au plus général :

```
paths:
# 1. Plus sp cifique en premier
- path: /api/v2
  backend:
    service:
      name: backend-v2-service
- path: /api
  backend:
    service:
      name: backend-service
# 2. Plus g n ral en dernier
- path: /
  backend:
    service:
      name: frontend-service
```

2. Routage de `/home` :
Le trafic sera routé vers `frontend-service:80`.
Explication :

- L'URL `/home` ne commence ni par `/api` ni par `/admin`
- Elle matche la règle `path: /` (catch-all)
- Ingress route vers `frontend-service` port 80

Tableau récapitulatif :

URL	Path matché	Service
/api/users	/api	backend-service :3000
/api	/api	backend-service :3000
/admin/settings	/admin	admin-service :8080
/home	/	frontend-service :80
/	/	frontend-service :80
/about	/	frontend-service :80

3. Cas d'usage réel :
Exemple : Application e-commerce

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ecommerce-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: shop.example.com
    http:
      paths:
      # Frontend SPA (React/Vue)
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-service
```

10 Pour aller plus loin

Ressources complémentaires

- Documentation officielle Kubernetes : `https://kubernetes.io/fr/docs/home/`
- Interactive tutorial : `https://kubernetes.io/docs/tutorials/`
- Kubernetes CheatSheet : `https://kubernetes.io/docs/reference/kubectl/cheatsheet/`
- Play with Kubernetes : `https://labs.play-with-k8s.com/`
- Kubernetes patterns : `https://k8spatterns.io/`

Fin du corrigé

Ce document est strictement confidentiel et réservé aux enseignants.