Correction TD3 : Helm

Gestionnaire de packages pour Kubernetes

Version pédagogique avec explications détaillées

Maxime Lambert
IUT Grand Ouest Normandie – BUT Informatique S5

Année 2024/2025

Table des matières

1 Introduction

Ce document présente la correction complète du TD3 sur Helm, le gestionnaire de packages pour Kubernetes. Chaque exercice est accompagné d'explications détaillées pour faciliter la compréhension des concepts.

Qu'est-ce que Helm ?

Helm est un gestionnaire de packages pour Kubernetes qui permet de :
- **Packager** des applications Kubernetes en *Charts* (ensemble de fichiers YAML)
- **Versioner** vos déploiements
- **Partager** des configurations réutilisables
- **Gérer** le cycle de vie des applications (installation, mise à jour, rollback)

1.1 Prérequis

- Cluster Kubernetes fonctionnel (Minikube, Kind, ou autre)
- Helm 3 installé (`helm version`)
- kubectl configuré et fonctionnel
- Connaissances des ressources Kubernetes de base (Pod, Service, Deployment, PVC)

1.2 Objectifs pédagogiques

À la fin de ce TD, vous serez capables de :
- Créer un Chart Helm depuis zéro
- Gérer des dépendances entre Charts
- Variabiliser vos configurations avec `values.yaml`
- Utiliser des templates Helm et des helpers
- Appliquer les bonnes pratiques de packaging

2 Exercice 1 : À la découverte de Helm

2.1 Objectif

Créer votre premier Chart Helm pour déployer une application Code Server (VS Code dans le navigateur).

2.2 Concepts clés

Structure d'un Chart Helm

Un Chart Helm est un répertoire contenant :
- `Chart.yaml` : Métadonnées du Chart (nom, version, description)
- `templates/` : Fichiers YAML des ressources Kubernetes
- `values.yaml` : Valeurs par défaut configurables (optionnel)
- `charts/` : Dépendances (sous-Charts)

2.3 Solution pas à pas

2.3.1 Étape 1 : Installation de Helm

Listing 1 – Installation de Helm 3

```
# Sur Linux
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash

# Verification de l'installation
helm version
# Resultat attendu : version.BuildInfo{Version:"v3.x.x", ...}
```

Pourquoi Helm 3 ?

Helm 3 a supprimé le composant serveur "Tiller" de Helm 2, ce qui :
- Simplifie l'architecture (plus de démon côté serveur)
- Améliore la sécurité (utilise directement vos credentials kubectl)
- Facilite l'installation (juste un binaire)

2.3.2 Étape 2 : Création de la structure du Chart

Listing 2 – Création du répertoire et de la structure

```
# Creer le repertoire du projet
mkdir ~/testlab
cd ~/testlab

# Creer la structure du Chart
mkdir -p vs-code-chart/templates
mkdir -p vs-code-chart/charts
```

Bonnes pratiques

- Nommez vos Charts en `kebab-case` (minuscules avec tirets)
- Le nom doit être descriptif de l'application
- Ne pas utiliser d'espaces ou de caractères spéciaux

2.3.3 Étape 3 : Création du fichier Chart.yaml

Listing 3 – vs-code-chart/Chart.yaml

```
apiVersion: v2
name: vs-code-chart
description: A Helm chart for my application
version: 0.1
```

Explication des champs

- `apiVersion: v2` : Version de l'API Helm (v2 pour Helm 3)
- `name` : Nom du Chart (doit correspondre au nom du répertoire)
- `description` : Description courte de l'application
- `version: 0.1` : Version du Chart (suivre le SemVer : MAJOR.MINOR.PATCH)

2.3.4 Étape 4 : Création des templates Kubernetes

Listing 4 – vs-code-chart/templates/deployment.yaml

Template du Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: code-server
  labels:
    app: code-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: code-server
  template:
    metadata:
      labels:
        app: code-server
    spec:
      containers:
      - name: code-server
        image: codercom/code-server:latest
        ports:
        - containerPort: 8080
        volumeMounts:
        - name: code-server-storage
          mountPath: /home/coder
      volumes:
      - name: code-server-storage
        persistentVolumeClaim:
          claimName: code-server-pvc
```

Rôle du Deployment

Le Deployment :
- Gère le cycle de vie des Pods
- Assure qu'1 réplica de code-server tourne en permanence
- Monte un volume persistant pour sauvegarder les données utilisateur
- Redémarre automatiquement le pod en cas de crash

Listing 5 – vs-code-chart/templates/service.yaml

Template du Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: code-server
  labels:
    app: code-server
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 8080
    nodePort: 30080
  selector:
    app: code-server
```

Type de Service : NodePort

NodePort expose le service sur un port du nœud Kubernetes :

- `port: 8080` : Port du Service (dans le cluster)
- `targetPort: 8080` : Port du conteneur
- `nodePort: 30080` : Port exposé sur l'hôte (plage 30000-32767)
- Accessible via `<IP-du-noeud>:30080`

Template du PVC Listing 6 – vs-code-chart/templates/pvc.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: code-server-pvc
  labels:
    app: code-server
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

PersistentVolumeClaim (PVC)

Le PVC demande un volume de stockage persistant :
- `ReadWriteOnce` : Le volume peut être monté par un seul nœud
- `5Gi` : Taille du volume demandé
- Le stockage persiste même si le pod est supprimé
- Kubernetes provisionne automatiquement le volume via le StorageClass

2.3.5 Étape 5 : Packaging du Chart

Listing 7 – Création du package .tgz

```bash
# Retour au repertoire parent
cd ~/testlab

# Packager le Chart
helm package vs-code-chart

# Resultat : vs-code-chart-0.1.tgz
ls -lh vs-code-chart-0.1.tgz
```

Qu'est-ce qu'un package Helm ?

`helm package` crée une archive .tgz contenant :
- Tous les fichiers du Chart
- Les métadonnées du Chart.yaml
- Un hash pour vérifier l'intégrité
- Cette archive peut être partagée et installée facilement

2.3.6 Étape 6 : Installation du Chart

Listing 8 – Déploiement du Chart

```bash
# Creer le namespace
kubectl create namespace td3

# Installer le Chart
helm upgrade --install vs-code-release \
  vs-code-chart-0.1.tgz \
  --namespace td3

```

```
# Verifier l'installation
helm list -n td3
kubectl get all,pvc -n td3
```

Commande helm upgrade –install

Cette commande :

- `-install` : Installe si la release n'existe pas
- `-upgrade` : Met à jour si la release existe déjà
- `vs-code-release` : Nom de la release (peut être différent du Chart)
- `-namespace td3` : Namespace où déployer
- Idempotente : peut être relancée sans problème

2.3.7 Étape 7 : Vérification

Listing 9 – Tests de vérification

```
# Verifier l'historique des releases
helm history vs-code-release -n td3

# Tester la connectivite
kubectl run test-curl --rm -it --restart=Never \
  --image=curlimages/curl \
  -- curl -I http://code-server:8080

# Resultat attendu : HTTP/1.1 302 Found (redirection vers /login)
```

Exercice 1 terminé !

Vous avez créé et déployé votre premier Chart Helm contenant :

- 1 Deployment (code-server)
- 1 Service (NodePort sur port 30080)
- 1 PVC (5Gi de stockage)

Version du Chart : **0.1**

3 Exercice 2 : Améliorer la modularité

3.1 Objectif

Séparer la gestion du stockage dans un Chart dédié pour améliorer la réutilisabilité.

3.2 Concepts clés

Pourquoi séparer en sous-Charts ?

La modularité apporte plusieurs avantages :
- **Réutilisabilité** : Le storage-chart peut être utilisé par d'autres applications
- **Séparation des responsabilités** : Stockage vs Application
- **Versioning indépendant** : Le stockage évolue indépendamment de l'app
- **Maintenance facilitée** : Modifications isolées

3.3 Solution pas à pas

3.3.1 Étape 1 : Création du Chart de stockage

Listing 10 – Structure du storage-chart

```
# Creer la structure
mkdir -p storage-chart/templates

# Creer le Chart.yaml
cat > storage-chart/Chart.yaml <<EOF
apiVersion: v2
name: storage-chart
description: Storage management for applications
version: 0.1
EOF
```

3.3.2 Étape 2 : Déplacer le PVC

Listing 11 – storage-chart/templates/pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: code-server-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Attention au nom du PVC

Le nom `code-server-pvc` doit rester identique pour que le Deployment puisse le référencer via `claimName: code-server-pvc`.

3.3.3 Étape 3 : Packaging du storage-chart

Listing 12 – Création du package storage

```
cd ~/testlab
helm package storage-chart

# Resultat : storage-chart-0.1.tgz
```

3.3.4 Étape 4 : Déclaration de la dépendance

Listing 13 – vs-code-chart/Chart.yaml (version 0.2)

```
apiVersion: v2
name: vs-code-chart
description: A Helm chart for my application
version: 0.2
dependencies:
  - name: storage-chart
    version: 0.1
    repository: "file://../storage-chart-0.1.tgz"
```

Section dependencies

Chaque dépendance spécifie :
- `name` : Nom du Chart dépendant
- `version` : Version exacte requise
- `repository` : Où trouver le Chart
  - `file://` : Fichier local
  - `https://` : Repository Helm distant

3.3.5 Étape 5 : Suppression du PVC du Chart principal

Listing 14 – Nettoyage

```
# Supprimer le PVC du Chart principal
rm vs-code-chart/templates/pvc.yaml

# Le PVC sera maintenant gere par storage-chart
```

3.3.6 Étape 6 : Ajout de la dépendance au Chart

Listing 15 – Installation de la dépendance

```
# Copier le package dans le repertoire charts/
cp storage-chart-0.1.tgz vs-code-chart/charts/

# Packager la nouvelle version
helm package vs-code-chart

# Resultat : vs-code-chart-0.2.tgz
```

Répertoire charts/

Le répertoire `charts/` contient les Charts dépendants :
- Peuvent être des packages .tgz
- Ou des sous-répertoires de Charts
- Helm les installe automatiquement avec le Chart parent

3.3.7 Étape 7 : Déploiement de la version 0.2

Listing 16 – Mise à jour du déploiement

```
# Desinstaller l'ancienne version proprement
helm uninstall vs-code-release -n td3
kubectl delete pvc code-server-pvc -n td3

# Installer la nouvelle version
helm install vs-code-release vs-code-chart-0.2.tgz \
  --namespace td3

# Verifier
helm list -n td3
kubectl get all,pvc -n td3
```

Exercice 2 terminé !

Vous avez modularisé votre Chart :

- **storage-chart v0.1** : Gère le PVC
- **vs-code-chart v0.2** : Dépend de storage-chart
- Architecture plus maintenable et réutilisable

4 Exercice 3 : Variabilisation avec values.yaml

4.1 Objectif

Rendre le Chart configurable en externalisant les valeurs dans `values.yaml`.

4.2 Concepts clés

Pourquoi variabiliser ?

La variabilisation permet :
- **Configuration sans modifier les templates**
- **Environnements multiples** : dev, staging, prod
- **Personnalisation** : Chaque utilisateur peut adapter
- **Surcharge** : `-set` ou fichiers values distincts

4.3 Solution pas à pas

4.3.1 Étape 1 : Création du fichier values.yaml

Listing 17 – vs-code-chart/values.yaml

```yaml
# Configuration de l'application
app:
  name: code-server
  image:
    repository: codercom/code-server
    tag: latest
    pullPolicy: IfNotPresent

  replicas: 1

  resources:
    requests:
      cpu: 100m
      memory: 256Mi
    limits:
      cpu: 500m
      memory: 512Mi

  service:
    type: NodePort
    port: 8080
    nodePort: 30080

  storage:
    size: 5Gi
    accessModes:
      - ReadWriteOnce
```

Structure du values.yaml

Organisation hiérarchique recommandée :
- Regrouper par composant (`app`, `database`, etc.)
- Sous-groupes logiques (`image`, `resources`, etc.)
- Noms explicites et cohérents
- Commentaires pour documenter les valeurs

4.3.2 Étape 2 : Modification du template Deployment

Listing 18 – templates/deployment.yaml variabilisé

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.app.name }}
  labels:
    app: {{ .Values.app.name }}
spec:
  replicas: {{ .Values.app.replicas }}
  selector:
    matchLabels:
      app: {{ .Values.app.name }}
  template:
    metadata:
      labels:
        app: {{ .Values.app.name }}
    spec:
      containers:
      - name: {{ .Values.app.name }}
        image: "{{ .Values.app.image.repository }}:{{ .Values.app.image.tag }}"
        imagePullPolicy: {{ .Values.app.image.pullPolicy }}
        ports:
        - containerPort: 8080
        resources:
          requests:
            cpu: {{ .Values.app.resources.requests.cpu }}
            memory: {{ .Values.app.resources.requests.memory }}
          limits:
            cpu: {{ .Values.app.resources.limits.cpu }}
            memory: {{ .Values.app.resources.limits.memory }}
        volumeMounts:
        - name: code-server-storage
          mountPath: /home/coder
      volumes:
      - name: code-server-storage
        persistentVolumeClaim:
          claimName: {{ .Values.app.name }}-pvc
```

Syntaxe des templates Helm

Helm utilise le langage de template Go :

- `{{ .Values.xxx }}` : Accès aux valeurs du values.yaml
- `{{ .Release.Name }}` : Nom de la release
- `{{ .Chart.Version }}` : Version du Chart
- `"..."` : Guillemets pour forcer le type string

4.3.3 Étape 3 : Modification du template Service

Listing 19 – templates/service.yaml variabilisé

```yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.app.name }}
  labels:
    app: {{ .Values.app.name }}
spec:
```

```
  type: {{ .Values.app.service.type }}
  ports:
  - port: {{ .Values.app.service.port }}
    targetPort: 8080
    nodePort: {{ .Values.app.service.nodePort }}
  selector:
    app: {{ .Values.app.name }}
```

4.3.4 Étape 4 : Packaging et déploiement

Listing 20 – Mise à jour vers v0.3

```
# Mettre a jour le Chart.yaml
cat > vs-code-chart/Chart.yaml <<EOF
apiVersion: v2
name: vs-code-chart
description: A Helm chart for my application
version: 0.3
dependencies:
  - name: storage-chart
    version: 0.1
    repository: "file://../storage-chart-0.1.tgz"
EOF

# Packager
helm package vs-code-chart

# Deployer
helm upgrade vs-code-release vs-code-chart-0.3.tgz \
  --namespace td3
```

4.3.5 Étape 5 : Tester la variabilisation

Listing 21 – Exemples de surcharge de valeurs

```
# Changer le nombre de replicas
helm upgrade vs-code-release vs-code-chart-0.3.tgz \
  --namespace td3 \
  --set app.replicas=2

# Changer la version de l'image
helm upgrade vs-code-release vs-code-chart-0.3.tgz \
  --namespace td3 \
  --set app.image.tag=4.10.0

# Utiliser un fichier values personnalise
cat > values-prod.yaml <<EOF
app:
  replicas: 3
  resources:
    requests:
      cpu: 200m
      memory: 512Mi
    limits:
      cpu: 1000m
      memory: 1Gi
EOF

helm upgrade vs-code-release vs-code-chart-0.3.tgz \
  --namespace td3 \
  --values values-prod.yaml
```

Ordre de priorité des valeurs

Helm fusionne les valeurs dans cet ordre (du moins au plus prioritaire) :

1. Valeurs par défaut du values.yaml du Chart
2. Valeurs des Charts parents
3. Fichier values fourni avec `-values`
4. Valeurs individuelles avec `-set`

Exercice 3 terminé !

Votre Chart est maintenant entièrement configurable :
- Fichier `values.yaml` avec toutes les valeurs
- Templates variabilisés avec `{{ .Values.* }}`
- Possibilité de surcharger avec `-set` ou `-values`

Version du Chart : **0.3**

5 Exercice 4 : Chart commun avec helpers

5.1 Objectif

Créer un Chart contenant des labels standardisés réutilisables via des helpers.

5.2 Concepts clés

Qu'est-ce qu'un helper ?

Un helper est un template nommé réutilisable :
- Défini dans `_helpers.tpl`
- Peut être appelé depuis n'importe quel template
- Évite la duplication de code
- Pratique pour labels, annotations, sélecteurs

5.3 Solution pas à pas

5.3.1 Étape 1 : Création du common-chart

Listing 22 – Structure du Chart commun

```
# Creer la structure
mkdir -p common-chart/templates

# Creer le Chart.yaml
cat > common-chart/Chart.yaml <<EOF
apiVersion: v2
name: common-chart
description: Common labels and helpers
version: 0.1
EOF
```

5.3.2 Étape 2 : Création du fichier de helpers

Listing 23 – common-chart/templates/_helpers.tpl

```
{{- define "common-chart.labels" -}}
orga: "IUT-C3"
res: "R5-09"
app: {{ .Values.app.name | default "myapp" }}
version: {{ .Release.Name }}
managed-by: {{ .Release.Service }}
{{- end }}
```

Syntaxe des helpers

- `{{- define "nom" -}}` : Définit un helper nommé
- `{{- end }}` : Fin de la définition
- Le `-` supprime les espaces/retours à la ligne
- `| default "valeur"` : Valeur par défaut si vide
- `.Release.Service` : Toujours "Helm" (identifie Helm)

5.3.3 Étape 3 : Packaging du common-chart

Listing 24 – Création du package

```
cd ~/testlab
helm package common-chart

# Resultat : common-chart-0.1.tgz
```

5.3.4 Étape 4 : Ajout comme dépendance

Listing 25 – vs-code-chart/Chart.yaml (version 0.4)

```
apiVersion: v2
name: vs-code-chart
description: A Helm chart for my application
version: 0.4
dependencies:
  - name: storage-chart
    version: 0.1
    repository: "file://../storage-chart-0.1.tgz"
  - name: common-chart
    version: 0.1
    repository: "file://../common-chart-0.1.tgz"
```

Listing 26 – Ajout du package

```
# Copier le package dans charts/
cp common-chart-0.1.tgz vs-code-chart/charts/
```

5.3.5 Étape 5 : Utilisation dans les templates

Listing 27 – templates/deployment.yaml avec labels communs

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.app.name }}
  labels:
{{ include "common-chart.labels" . | indent 4 }}
spec:
  replicas: {{ .Values.app.replicas }}
  selector:
    matchLabels:
      app: {{ .Values.app.name }}
  template:
    metadata:
      labels:
        app: {{ .Values.app.name }}
    spec:
      containers:
      - name: {{ .Values.app.name }}
        image: "{{ .Values.app.image.repository }}:{{ .Values.app.image.tag }}"
        imagePullPolicy: {{ .Values.app.image.pullPolicy }}
        ports:
        - containerPort: 8080
        resources:
          requests:
            cpu: {{ .Values.app.resources.requests.cpu }}
```

```
            memory: {{ .Values.app.resources.requests.memory }}
          limits:
            cpu: {{ .Values.app.resources.limits.cpu }}
            memory: {{ .Values.app.resources.limits.memory }}
        volumeMounts:
        - name: code-server-storage
          mountPath: /home/coder
      volumes:
      - name: code-server-storage
        persistentVolumeClaim:
          claimName: {{ .Values.app.name }}-pvc
```

Listing 28 – templates/service.yaml avec labels communs

```yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.app.name }}
  labels:
{{ include "common-chart.labels" . | indent 4 }}
spec:
  type: {{ .Values.app.service.type }}
  ports:
  - port: {{ .Values.app.service.port }}
    targetPort: 8080
    nodePort: {{ .Values.app.service.nodePort }}
  selector:
    app: {{ .Values.app.name }}
```

Fonctions de template

- `include "nom" .` : Appelle le helper et passe le contexte
- `| indent 4` : Indente le résultat de 4 espaces
- Le point `.` passe tout le contexte au helper
- Sans le point, le helper n'aurait pas accès à `.Values`, `.Release`, etc.

5.3.6 Étape 6 : Déploiement de la version finale

Listing 29 – Packaging et déploiement

```bash
# Packager la version finale
helm package vs-code-chart

# Resultat : vs-code-chart-0.4.tgz

# Mettre a jour le deploiement
helm upgrade vs-code-release vs-code-chart-0.4.tgz \
  --namespace td3
```

5.3.7 Étape 7 : Vérification des labels

Listing 30 – Tests de vérification

```bash
# Verifier les labels du Deployment
kubectl get deployment code-server -n td3 \
  -o yaml | grep -A 10 "labels:"

# Resultat attendu :
# labels:
#   app: code-server
#   managed-by: Helm
#   orga: IUT-C3
#   res: R5-09
#   version: vs-code-release

# Verifier les labels du Service
kubectl get service code-server -n td3 \
  -o yaml | grep -A 10 "labels:"

# Filtrer les ressources par label
kubectl get all -n td3 -l orga=IUT-C3
```

```
19 kubectl get all -n td3 -l res=R5-09
```

Exercice 4 terminé !

Vous avez créé un système de labels réutilisable :

- **common-chart v0.1** : Helper avec labels standardisés
- **vs-code-chart v0.4** : Utilise les labels communs
- Labels appliqués : orga, res, app, version, managed-by
- Réutilisable pour tous vos projets R5.09

6 Tests et validation

6.1 Commandes de vérification

Listing 31 – Tests complets du TD3

```
# 1. Verifier les releases Helm
helm list -n td3
helm history vs-code-release -n td3

# 2. Verifier les ressources Kubernetes
kubectl get all,pvc -n td3

# 3. Verifier les labels
kubectl get deployment code-server -n td3 \
  -o jsonpath='{.metadata.labels}' | jq

# 4. Tester la connectivite
kubectl run test-curl --rm -it --restart=Never \
  --image=curlimages/curl \
  -- curl -I http://code-server:8080

# 5. Acceder depuis l'exterieur (port-forward)
kubectl port-forward -n td3 svc/code-server 8080:8080 --address=0.0.0.0

# Puis dans un navigateur : http://<IP-VM>:8080
```

6.2 Résultats attendus

État final du déploiement

- **Namespace** : td3
- **Release** : vs-code-release (revision 3)
- **Chart** : vs-code-chart-0.4
- **Pod** : code-server (Running)
- **Service** : NodePort 30080
- **PVC** : 5Gi (Bound)
- **Labels** : orga=IUT-C3, res=R5-09

7 Commandes Helm essentielles

7.1 Gestion des Charts

Listing 32 – Commandes Chart

```
# Creer un Chart depuis un template
helm create mon-chart

# Valider la syntaxe d'un Chart
helm lint mon-chart

# Afficher les templates sans deployer (dry-run)
helm template mon-release mon-chart

# Packager un Chart
helm package mon-chart

# Afficher les valeurs par defaut
helm show values mon-chart
```

7.2 Gestion des releases

Listing 33 – Commandes Release

```
# Installer une release
helm install mon-release mon-chart -n namespace

# Mettre a jour une release
helm upgrade mon-release mon-chart -n namespace

# Installer ou mettre a jour (idempotent)
helm upgrade --install mon-release mon-chart -n namespace

# Lister les releases
helm list -n namespace
helm list --all-namespaces

# Voir l'historique
helm history mon-release -n namespace

# Rollback vers une revision precedente
helm rollback mon-release 2 -n namespace

# Desinstaller une release
helm uninstall mon-release -n namespace
```

7.3 Debugging

Listing 34 – Commandes Debug

```
# Voir les valeurs utilisees
helm get values mon-release -n namespace

# Voir les manifests deployes
helm get manifest mon-release -n namespace

# Voir toutes les infos de la release
helm get all mon-release -n namespace
```

```
9
10 # Debug avec mode dry-run et debug
11 helm install mon-release mon-chart --dry-run --debug
```

8 Bonnes pratiques

8.1 Versioning

Semantic Versioning (SemVer)

Format : `MAJOR.MINOR.PATCH`

- **MAJOR** : Changements incompatibles (breaking changes)
- **MINOR** : Nouvelles fonctionnalités rétro-compatibles
- **PATCH** : Corrections de bugs rétro-compatibles

Exemples :

- 0.1 → 0.2 : Nouvelle fonctionnalité
- 0.2 → 0.2.1 : Correction de bug
- 0.9 → 1.0 : Version stable
- 1.0 → 2.0 : Breaking change

8.2 Organisation des Charts

Structure recommandée

```
mon-chart/
 Chart.yaml              # Métadonnées
 values.yaml             # Valeurs par défaut
 charts/                 # Dépendances
 templates/              # Templates Kubernetes
    _helpers.tpl        # Helpers réutilisables
    deployment.yaml
    service.yaml
    ingress.yaml
    NOTES.txt           # Message post-install
 .helmignore             # Fichiers à ignorer
 README.md               # Documentation
```

8.3 Nomenclature

- **Noms de Charts** : kebab-case (my-awesome-chart)
- **Noms de releases** : kebab-case (my-release-name)
- **Clés dans values.yaml** : camelCase (imagePullPolicy)
- **Labels Kubernetes** : kebab-case (app.kubernetes.io/name)

8.4 Sécurité

Points de vigilance

- Ne jamais inclure de secrets en dur dans les Charts
- Utiliser des Secrets Kubernetes ou des outils dédiés (Sealed Secrets, Vault)
- Définir des limites de ressources (limits/requests)
- Utiliser des tags d'image précis (pas `latest`)
- Valider les Charts avec `helm lint`

9 Aller plus loin

9.1 Fonctionnalités avancées

9.1.1 Hooks Helm

Les hooks permettent d'exécuter des actions à des moments précis :

Listing 35 – Exemple de hook pre-install

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pre-install-job
  annotations:
    "helm.sh/hook": pre-install
    "helm.sh/hook-weight": "-5"
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  template:
    spec:
      containers:
      - name: pre-install
        image: busybox
        command: ['sh', '-c', 'echo "Pre-install hook"']
      restartPolicy: Never
```

Hooks disponibles : pre-install, post-install, pre-upgrade, post-upgrade, pre-delete, post-delete

9.1.2 Tests Helm

Listing 36 – templates/tests/test-connection.yaml

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

Exécuter les tests : `helm test mon-release -n namespace`

9.1.3 Conditions et boucles

Listing 37 – Exemple de condition

```yaml
{{- if .Values.ingress.enabled }}
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ .Values.app.name }}
spec:
  rules:
  - host: {{ .Values.ingress.host }}
    http:
```

```
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: {{ .Values.app.name }}
            port:
              number: {{ .Values.app.service.port }}
{{- end }}
```

9.2 Repositories Helm

9.2.1 Utiliser un repository public

Listing 38 – Ajout de repositories

```
# Ajouter un repository
helm repo add bitnami https://charts.bitnami.com/bitnami

# Lister les repositories
helm repo list

# Mettre a jour les repositories
helm repo update

# Chercher un Chart
helm search repo nginx

# Installer depuis un repository
helm install mon-nginx bitnami/nginx -n namespace
```

9.2.2 Créer son propre repository

Listing 39 – Repository GitHub Pages

```
# Creer un repository Git
git init my-helm-repo
cd my-helm-repo

# Copier les Charts
cp ../mon-chart-1.0.0.tgz .

# Generer l'index
helm repo index . --url https://example.com/helm-charts

# Pousser sur GitHub
git add .
git commit -m "Add chart"
git push origin main

# Configurer GitHub Pages sur la branche main

# Utiliser le repository
helm repo add my-repo https://example.com/helm-charts
helm repo update
helm install my-release my-repo/mon-chart
```

10 Troubleshooting

10.1 Problèmes courants

10.1.1 Erreur : Release already exists

```
# Symptome
Error: cannot re-use a name that is still in use

# Solution : Utiliser upgrade --install au lieu de install
helm upgrade --install mon-release mon-chart -n namespace
```

10.1.2 Erreur : Template rendering

```
# Symptome
Error: template: ... executing ... at <.Values.xxx>:
nil pointer evaluating interface {}.xxx

# Solution : Verifier que la valeur existe dans values.yaml
# Ou utiliser un default
{{ .Values.xxx | default "valeur-par-defaut" }}
```

10.1.3 Erreur : YAML indentation

```
# Symptome
Error: YAML parse error ... mapping values are not allowed

# Solution : Verifier l'indentation
# Utiliser 'helm lint' pour detecter les erreurs
helm lint mon-chart

# Tester le rendu avec dry-run
helm template mon-release mon-chart --debug
```

10.1.4 Dépendances non trouvées

```
# Symptome
Error: found in Chart.yaml, but missing in charts/ directory

# Solution : Copier le package dans charts/
cp mon-dependency-chart-0.1.tgz mon-chart/charts/

# Ou utiliser helm dependency update (pour repositories distants)
helm dependency update mon-chart
```

11 Ressources complémentaires

11.1 Documentation officielle

— **Helm** : `https://helm.sh/docs/`
— **Chart Template Guide** : `https://helm.sh/docs/chart_template_guide/`
— **Best Practices** : `https://helm.sh/docs/chart_best_practices/`
— **Artifact Hub** : `https://artifacthub.io/` (recherche de Charts)

11.2 Charts populaires

— **Bitnami** : nginx, postgresql, redis, wordpress, etc.
— **Prometheus Stack** : monitoring complet
— **Ingress NGINX** : contrôleur Ingress
— **Cert-Manager** : gestion des certificats TLS
— **ArgoCD** : GitOps pour Kubernetes

11.3 Outils complémentaires

— **Helmfile** : Déclarer plusieurs releases Helm en YAML
— **Helm Diff** : Voir les différences avant upgrade
— **Helm Secrets** : Chiffrer les values.yaml avec SOPS
— **Chart Testing (ct)** : Tester et valider les Charts

12 Conclusion

12.1 Récapitulatif

Vous avez appris à :
— Créer un Chart Helm depuis zéro
— Packager et déployer des applications Kubernetes
— Gérer des dépendances entre Charts
— Variabiliser les configurations avec values.yaml
— Créer des templates réutilisables avec helpers
— Appliquer des labels standardisés
— Tester et valider vos déploiements

12.2 Architecture finale

TD3 complété !

Votre architecture modulaire :
- **vs-code-chart v0.4** : Application principale
  - Deployment variabilisé
  - Service NodePort configurable
  - Resources limits/requests
  - Labels standardisés
- **storage-chart v0.1** : Gestion du stockage
  - PVC réutilisable
  - Taille configurable
- **common-chart v0.1** : Labels communs
  - Helper pour labels standardisés
  - Réutilisable dans tous vos projets

12.3 Compétences acquises

Ces compétences sont essentielles pour :
— Déployer des applications en production
— Travailler dans une équipe DevOps
— Maintenir des infrastructures Kubernetes
— Créer des Charts réutilisables pour votre entreprise
— Contribuer à des projets open-source

Félicitations !

Vous maîtrisez maintenant les fondamentaux de Helm
et êtes prêts pour des déploiements Kubernetes professionnels.