
AMBIENT·IT

# Table of Contents

# Introduction

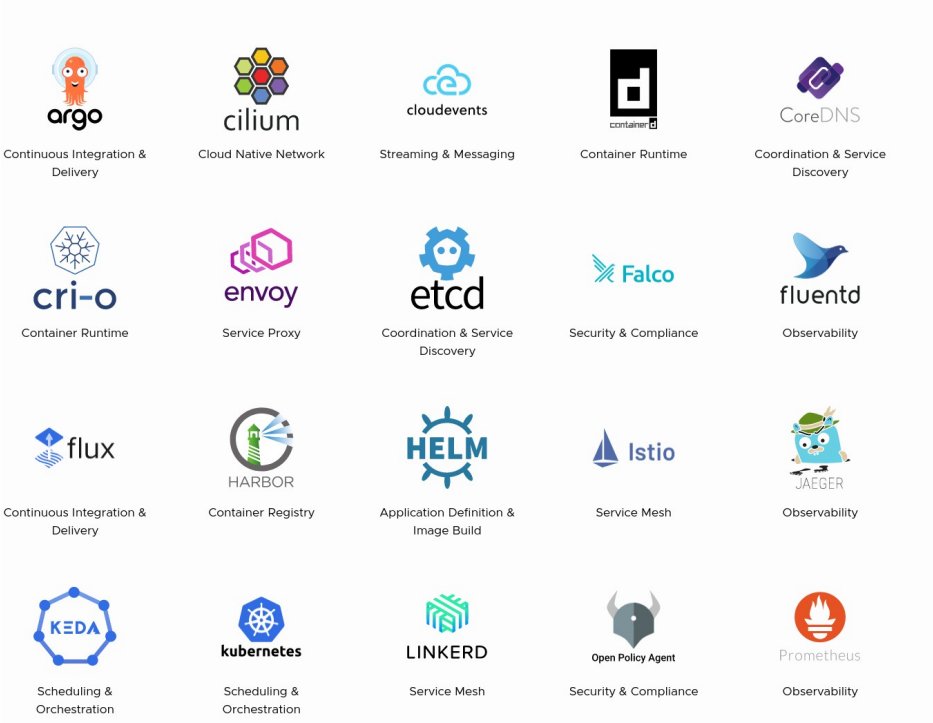
## CNCF

CLOUD NATIVE
COMPUTING FOUNDATION

Les technologies *Cloud Native* permettent aux entreprises de construire et d'exploiter des applications élastiques dans des environnements modernes et dynamiques comme des *clouds* publics, privés ou bien hybrides. Les conteneurs, le maillage de services, les microservices, les infrastructures immuables et les API déclaratives illustrent cette approche.

Ces techniques permettent la mise en œuvre de systèmes faiblement couplés, à la fois résistants, pilotables et observables. Combinés à un robuste système d'automatisation, ils permettent aux ingénieurs de procéder à des modifications impactantes, fréquemment et de façon prévisible avec un minimum de travail.

La *Cloud Native Computing Foundation* cherche à favoriser l'adoption de ce paradigme en encourageant et en soutenant un écosystème de projets *open source* et indépendants. Nous démocratisons l'état de l'art des bonnes pratiques afin de rendre l'innovation accessible à tous.



Communautaire d'autres projets de la CNCF ont rapidement gagné en adoption et obtenu un soutien communautaire diversifié, devenant ainsi certains des projets au l'adoption la plus élevée de l'histoire de l'open source.

## Déployer une application dans Kubernetes

Déployer une application dans Kubernetes demande de créer de nombreux objets (ressources) de type Kubernetes. Typiquement, nous allons créer un objet Deployments, mais également d'autres objets, des configMaps, des secrets, des persistantVolumeClaim, des ingress, etc.

Pour cela nous devons écrire des manifest au format YAML. Ces manifest décrivent notre besoin, notre application et ...

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness
  labels:
   app: demo
  namespace: ludovic
spec:
  containers:
  - image: quenec/liveness:latest
      name:  liveness
      livenessProbe:
        httpGet:
          path: /var/www/html/sonde
          port: 80
        initialDelaySeconds: 5
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 2
      ports:
        - containerPort: 80
          name: http
          protocol: TCP
```

```yaml
io.k8s.api.autoscaling.v1.HorizontalPodAutoscaler (v1@horizon
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-demo-deployment
  namespace: forma-ludo
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

```yaml
apiVersion: gateway.networking.k8s.io/v1beta1
kind: Gateway
metadata:
  name: istio-gateway
spec:
  gatewayClassName: istio
  listeners:
  - name: http
    port: 80
    protocol: HTTP
    allowedRoutes:
      namespaces:
        from: Same
```

Nous pouvons donc déployer notre application

```
[ludo@selfservice ~]$  kubectl apply -f Deployment.yaml pvc.yaml configmap.yaml
```

Cette méthode fonctionne parfaitement. ;) Toutefois, nous voulons parfois personnaliser nos manifest pour nos différentes configurations.

Changer la configuration de notre application, modifier la taille du stockage, modifier l'URL de l'ingress, changer de type de service. Etc.. Cela va nous demander de modifier nos manifest. Ce n'est pas bien compliqué, si nous maîtrisons la création des ressources Kubernetes.

Et si nous possédions un outil qui nous permet de modifier simplement les valeurs des paramètres. Et bien, voila HELM !

## Qu'est-ce que Helm ?

Helm est un projet OpenSource certifié de la Cloud Native Computing Foundation (CNCF). Présenté initialement en 2015 lors de la première édition de la KubeCon, il est devenu un projet à part entière de la CNCF avec Kubernetes.

Helm simplifie le processus en automatisant les opérations de déploiement des appllications. Ce gestionnaire de paquets pour Kubernetes s'utilise comme un système reproductible et partageable pour définir et déployer une application à l'aide de manifestes YAML individuels.

Helm se rapproche d'un outil de création de modèles qui assure la cohérence entre les conteneurs et le respect des exigences spécifiques de l'application. Le même framework de configuration peut être appliqué à plusieurs instances en remplaçant les valeurs, en fonction des priorités de la configuration.

Pour les équipes de développement travaillant avec Kubernetes, les charts Helm simplifient et accélèrent le déploiement d'applications qui pourront être facilement réutilisées ou partagées.

Pour les administrateurs système et les autres professionnels de l'exploitation informatique, Helm offre un outil cohérent pour la mise en œuvre et la rationalisation de l'intégration et du développement continus (CI/CD) dans les pipelines d'applications. Cet outil est autant conçu pour l'agilité que pour la cohérence.

# Prise en main

## Concepts et définitions (Chart, release, template, etc)

Helm est le gestionnaire de paquets pour Kubernetes. Helm propose des **Charts**. Ces charts sont des packages qui vont nous permettre de déployer et surtout personnaliser le déploiement des applications.

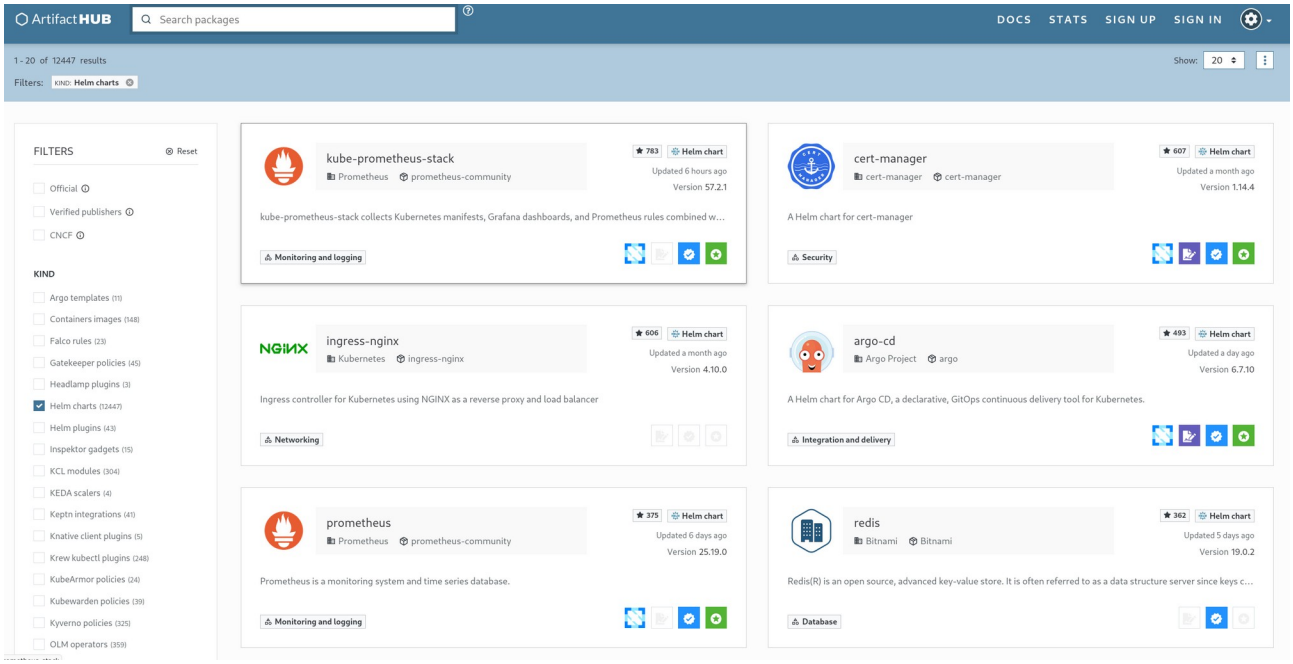
### Ou trouver les Charts ?

Les charts se trouvent sur des dépôts ou repositories applé repo. Nous devons ajouter des dépôts sur notre machine pour rechercher, déployer des Charts sur notre cluster.

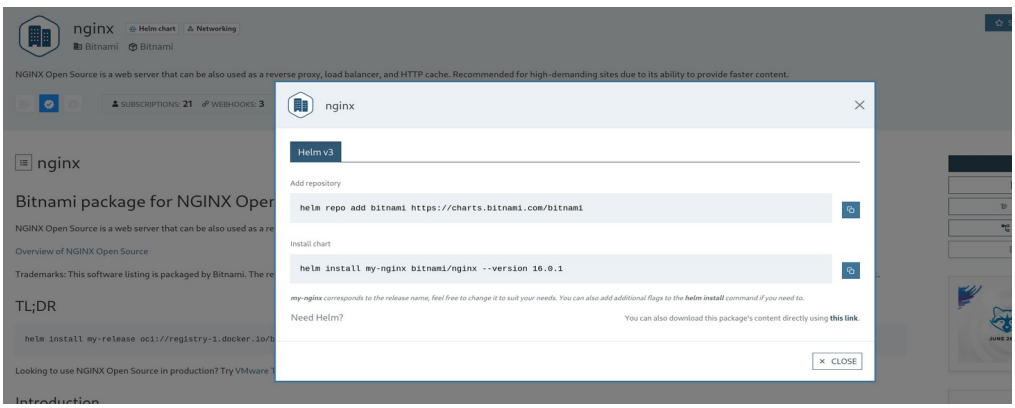
### Ou trouver les repo ?

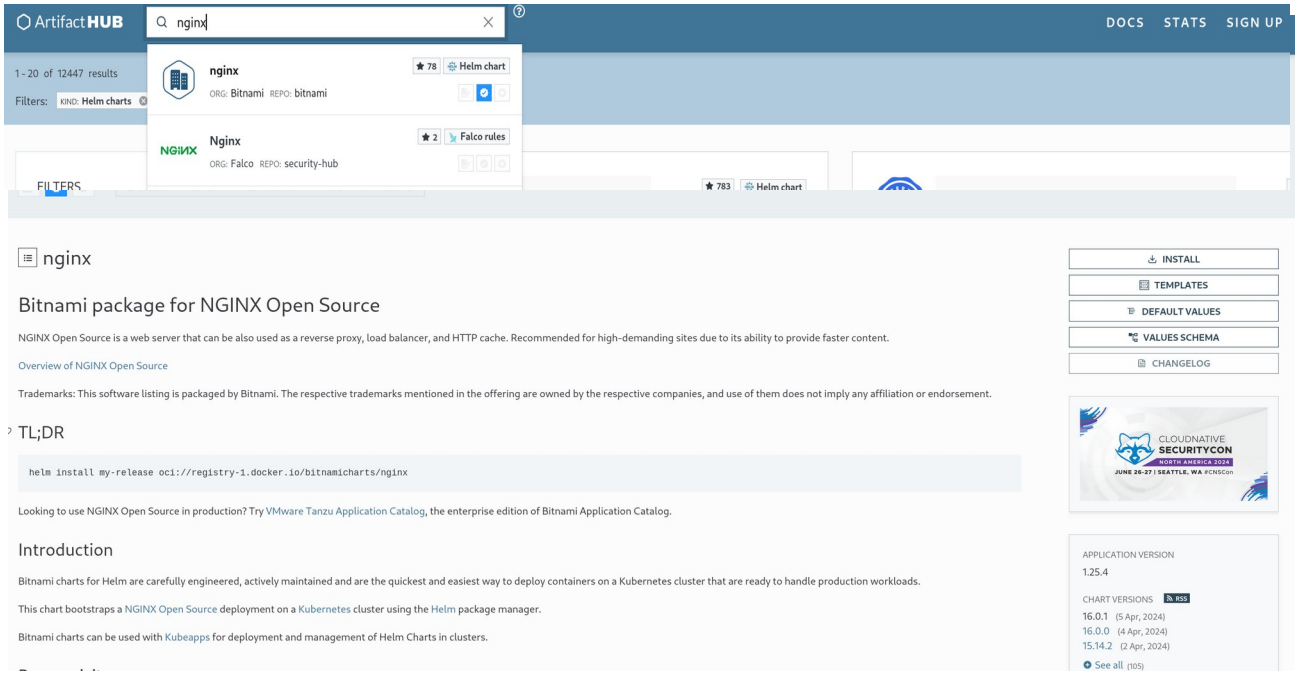
A l'instar du Docker Hub, artifacthub est devenu le hub officiel pour les Charts Helm.

Rendons nous a l'adresse : https://artifacthub.io/




Recherchons un Chart pour le serveur nginx.






## Utiliser des registres basés sur l'OCI

À partir de Helm 3, on peut utiliser des registres de conteneurs avec la prise en charge OCI pour stocker et partager Chart. À partir de Helm v3.8.0, la prise en charge de l'OCI est activée par défaut.

### Prise en charge de l'OCI avant la version 3.8.0

La prise en charge de l'OCI est passée du stade expérimental avec Helm v3.8.0. Dans les versions antérieures de Helm, la prise en charge de l'OCI se comportait différemment. Avant Helm v3.8.0, il est important de comprendre ce qui a changé avec les différentes versions de Helm.

Avant Helm v3.8.0, la prise en charge de l'OCI est expérimentale et doit être activée. Pour activer la prise en charge expérimentale de l'OCI pour les versions de Helm antérieures à la v3.8.0, définir la vairiable HELM_EXPERIMENTAL_OCI dans son environnement.

## TP - Comprendre et gérer les dépôts Helm (recherches, ajout, update)

Rechercher des charts

```
ludo@kubernetes:$ helm search hub nginx
URL                                                 CHART VERSION     APP
VERSION

https://artifacthub.io/packages/helm/cloudnativ...  3.2.0             1.16.0
```

Rechercher des charts

```
ludo@kubernetes:$ helm search repo nginx
```

PAS DE RESULTAT !

Ajout de dépôts

```
ludo@kubernetes:$ helm repo add bitnami https://charts.bitnami.com/bitnami
```

### Lister les dépôts installés

```
ludo@kubernetes:$  helm repo list

NAME                                    URL
bitnami                                 https://charts.bitnami.com/bitnami
elastic                                 https://helm.elastic.co
jenkins                                 https://charts.jenkins.io
prometheus-community                    https://prometheus-community.github.io/helm-
```

### Lister les versions des charts proposé

```
ludo@kubernetes:$  helm search repo --versions nginx
NAME                                            CHART VERSION     APP VERSION
      DESCRIPTION
bitnami/nginx                                   16.0.1            1.25.4
bitnami/nginx                                   16.0.0            1.25.4
bitnami/nginx                                   15.14.2           1.25.4
bitnami/nginx                                   15.14.1           1.25.4 NGINX
```

### Afficher les informations d'une Chart

```
ludo@kubernetes:$ helm show chart bitnami/nginx
ludo@kubernetes:$ helm show readme bitnami/nginx
ludo@kubernetes:$ helm show values bitnami/nginx
```

### Mise à jour des répos

```
ludo@kubernetes:$  helm repo update

Hang tight while we grab the latest from your chart repositories...

...Successfully got an update from the "ingress-nginx" chart repository

...Successfully got an update from the "cilium" chart repository

Update Complete. ▫Happy Helming!▫
```

### Supprimer un dépôt

```
ludo@kubernetes:$  helm repo remove bitnami
NAME
bitnami
```

## TP - Déployer une Chart - Release.

Une release, une version ? La release ou version est une instance d'un chart. Lorsque que l'on installe un chart, nous créons une Release avec sa revision.

Helm install

```
ludo@kubernetes:$  helm install my-nginx bitnami/nginx --version 15.10.0
NAME: my-nginx
LAST DEPLOYED: Sun Apr  7 09:30:17 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: nginx
CHART VERSION: 15.10.0
APP VERSION: 1.25.3
```

Helm list

```
ludo@kubernetes:$ helm list
NAME          NAMESPACE     REVISION      UPDATED    STATUS     CHART           APP VERSION
my-nginx      default       1             2024-04-06 deployed   nginx-15.10.0   1.25.4
```

## TP - Mise a jour de la Release.

Helm upgrade

```
ludo@kubernetes:$ helm upgrade my-nginx bitnami/nginx --version 16.0.0

Release "my-nginx" has been upgraded. Happy Helming!
NAME: my-nginx
LAST DEPLOYED: Sun Apr  7 09:39:14 2024
NAMESPACE: default
STATUS: deployed
REVISION: 2
TEST SUITE: None
NOTES:
CHART NAME: nginx
CHART VERSION: 16.0.0
APP VERSION: 1.25.4
```

## Mise a jour – changement de paramètres

Helm upgrade

```
ludo@kubernetes:$ helm upgrade my-nginx bitnami/nginx \
       --set service.type=nodePort

Error: UPGRADE FAILED: cannot patch "my-nginx" with kind Service: Service "my-
nginx" is invalid: spec.type: Unsupported value: "nodePort": supported values:
"ClusterIP", "ExternalName", "LoadBalancer", "NodePort"
```

Afficher les values

```
ludo@kubernetes:$ helm get values my-nginx --all
....
....
....

ludo@kubernetes:$ helm upgrade my-nginx bitnami/nginx \
     --set service.type=NodePort
```

```
Release "my-nginx" has been upgraded. Happy Helming!
NAME: my-nginx
LAST DEPLOYED: Sun Apr  7 09:43:38 2024
NAMESPACE: default
STATUS: deployed
REVISION: 4
```

Supprimer la revisioin « failed »

```
ludo@kubernetes:$ helm rollback --cleanup-on-fail ...
```

Afficher l'historique

```
ludo@kubernetes:$
REVISION    UPDATED                         STATUS      CHART             APP
VERSION     DESCRIPTION
1           Sun Apr  7 09:30:17 2024      superseded  nginx-15.10.0     1.25.3
      Install complete
2           Sun Apr  7 09:39:14 2024      superseded  nginx-16.0.0      1.25.4
      Upgrade complete
```

Rollback sur une version

```
ludo@kubernetes:$ helm rollback my-nginx 2
Rollback was a success! Happy Helming!
```

```
ludo@kubernetes:$ helm list
NAME        NAMESPACE   REVISION    UPDATED           DESCRIPTION
my-nginx    default     5           2024-04-07        Rollback to 2
```

## Déployer une Chart – fichier de values

Avec le fichier de values, c'est mieux !

### Récupérer un Chart

```
ludo@kubernetes:$ helm pull bitnami/nginx

nginx-16.0.1.tgz
```

```
ludo@kubernetes:$ helm pull bitnami/nginx --untar

ludo@kubernetes:$ ls nginx/
Chart.lock  charts  Chart.yaml  README.md  templates  values.schema.json
values.yaml
```

Modifier le type de Service et Déployer avec le fichier values

```
ludo@kubernetes:$ helm install my-nginx --values nginx/values.yaml
bitnami/nginx helm get notes my-nginx
CHART NAME: nginx
CHART VERSION: 16.0.1
APP VERSION: 1.25.4
```

# Créer ses Charts

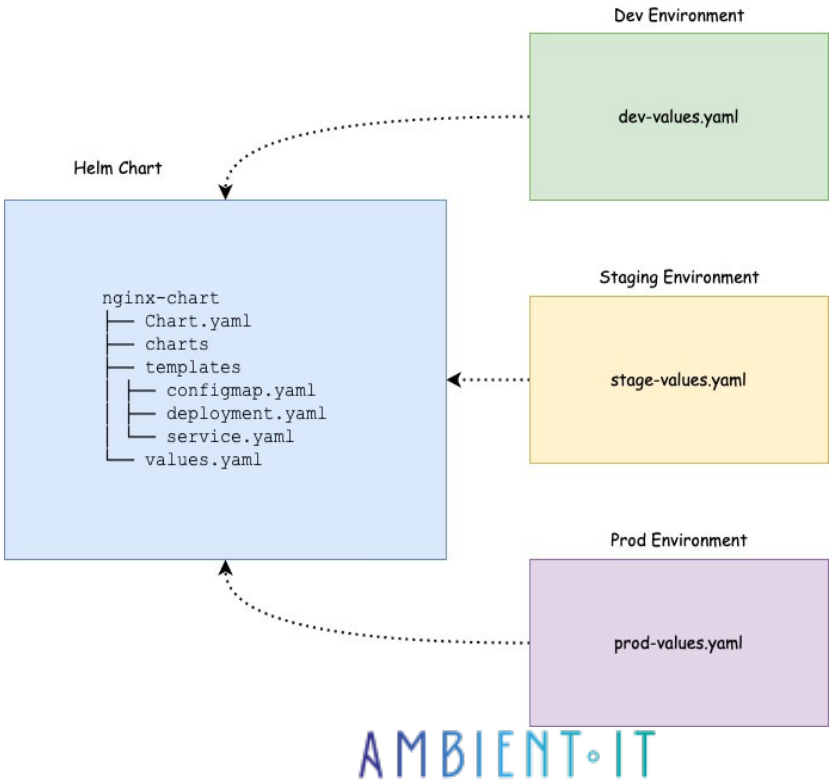
## Introduction

Nous allons prendre un exemple basique de déploiement d'un frontend  web utilisant Nginx sur Kubernetes.

Supposons que vous ayez quatre environnements différents dans votre projet. Dev, QA, Staging et Prod. Chaque environnement aura des paramètres différents pour le déploiement de Nginx. Par exemple :

- Dans les environnements Dev et QA, vous n'aurez peut-être besoin que d'une seule réplique.
- Dans staging et production, vous aurez plus de répliques avec pod autoscaling.
- Les règles de routage d'entrée seront différentes dans chaque environnement.
- La configuration et les secrets seront différents pour chaque environnement.

En raison de la modification des configurations et des paramètres de déploiement pour chaque environnement, vous devez maintenir des fichiers de déploiement de valeurs différents pour chaque environnement. Ou bien vous aurez un seul fichier de déploiement et vous devrez écrire des scripts shell ou pythons personnalisés pour remplacer les valeurs en fonction de l'environnement. Mais ce n’est pas une approche évolutive. C'est ici qu'intervient le Chart (helm chart).



# TP - Création d'un Chart

Exécutez la commande suivante pour créer un modèle de Chart avec les fichiers et dossiers par défaut. Le Chart est nommé nginx-chart

```
ludo@kubernetes:$ helm create nginx-chart
Creating nginx-chart
```

```
ludo@kubernetes:$ cd nginx-chart/

ludo@kubernetes:$ ls -a
charts/  Chart.yaml  .helmignore  templates/  values.yaml

ludo@kubernetes:$ ls templates/
deployment.yaml  _helpers.tpl  hpa.yaml  ingress.yaml  NOTES.txt
serviceaccount.yaml  service.yaml  tests
```

Examinons le contenu d'un Chart helm.

**charts** : Nous pouvons ajouter la structure d'un autre Chart dans ce répertoire si nos Charts principaux dépendent d'autres Charts. Par défaut, ce répertoire est vide.

**.helmignore** : Il est utilisé pour définir tous les fichiers que nous ne voulons pas inclure dans le Chart. Son fonctionnement est similaire à celui du fichier. gitignore.

**Chart.yaml** : Il contient des informations sur le chart comme la version, le nom, la description, etc.

**values.yaml** : Nous le connaissons déjà, nous définissons les valeurs des Templates YAML. Par exemple, le nom de l'image, le nombre de répliques, les valeurs HPA, etc.

**templates** : Ce répertoire contient tous les fichiers manifestes Kubernetes qui forment une application. Ces manifests peuvent être modifier pour accéder aux valeurs du fichier **values.yaml**. Helm crée des modèles par défaut pour les objets Kubernetes comme **deployment.yaml**, **service.yaml** etc, que nous pouvons utiliser directement, modifier, ou surcharger avec nos fichiers.

**templates/NOTES.txt** : Il s'agit d'un fichier en texte clair qui est imprimé une fois que le Chart a été déployé avec succès.

**templates/_helpers.tpl** : Ce fichier contient plusieurs méthodes et sous-modèles. Ces fichiers ne sont pas rendus dans les définitions d'objets Kubernetes mais sont disponibles partout dans les autres modèles de Charts.

**templates/tests/** : Nous pouvons définir des tests dans nos Charts pour valider que votre Chart fonctionne comme prévu lorsqu'il est installé.

**Chart.yaml**

Comme mentionné ci-dessus, nous mettons les détails de notre Chart dans le fichier Chart.yaml.  Remplacez le contenu par défaut de chart.yaml par ce qui suit :

```
apiVersion : v2
name: nginx-chart
description: Mon premier chart
type: application
version: 0.1.0
appVersion:  "1.0.0"
```

**apiVersion** : Ceci indique la version de l'API du Chart. v2 est pour Helm 3 et v1 est pour les versions précédentes.
**name** : Indique le nom du Chart.

**description** : Indique la description de la carte Helm.

**Type** : Le type de chart peut être "application" ou "library". Les charts d'application sont celles que vous déployez sur Kubernetes. Les Charts de bibliothèque sont des Charts réutilisables qui peuvent être utilisés avec d'autres Charts. Il s'agit d'un concept similaire à celui des bibliothèques en programmation.

**Version** : Indique la version du Chart.

**appVersion** : Indique le numéro de version de notre application.

**maintainers** : Informations sur le propriétaire du Chart.

Nous devons incrémenter la version et l'appVersion chaque fois que nous apportons des modifications à l'application. Il existe d'autres champs comme les dépendances, les icônes, etc.

## Modèles - Templates

Il y a plusieurs fichiers dans le répertoire templates créé par helm. Dans notre cas, nous allons travailler sur un simple déploiement Kubernetes Nginx.

Supprimons tous les fichiers par défaut du répertoire templates.

```
ludo@kubernetes:$ rm -rf templates/*
```

```
ludo@kubernetes:$ vim templates/deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
  namespace: forma-ludo
spec:
  selector:
    matchLabels:
      run: php-apache
  template:
    metadata:
      labels:
        run: php-apache
```

```
    spec:
      containers:
      - name: php-apache
        image: registry.k8s.io/hpa-example
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
          requests:
            cpu: 200m
```

Le fichier YAML ci-dessus, les valeurs sont statiques. L'idée d'un Chart des est de « templater » les fichiers YAML afin de pouvoir les réutiliser dans plusieurs environnements en leur attribuant des valeurs de manière dynamique.

Pour modéliser une valeur, il suffit d'ajouter le paramètre de l'objet à l'intérieur d'accolades, comme indiqué ci-dessous. C'est ce qu'on appelle une directive de template et la syntaxe est spécifique au templating Go

***{{.Object.Parameter }}***

Commençons par comprendre ce qu'est un objet. Voici les trois objets que nous allons utiliser dans cet exemple.

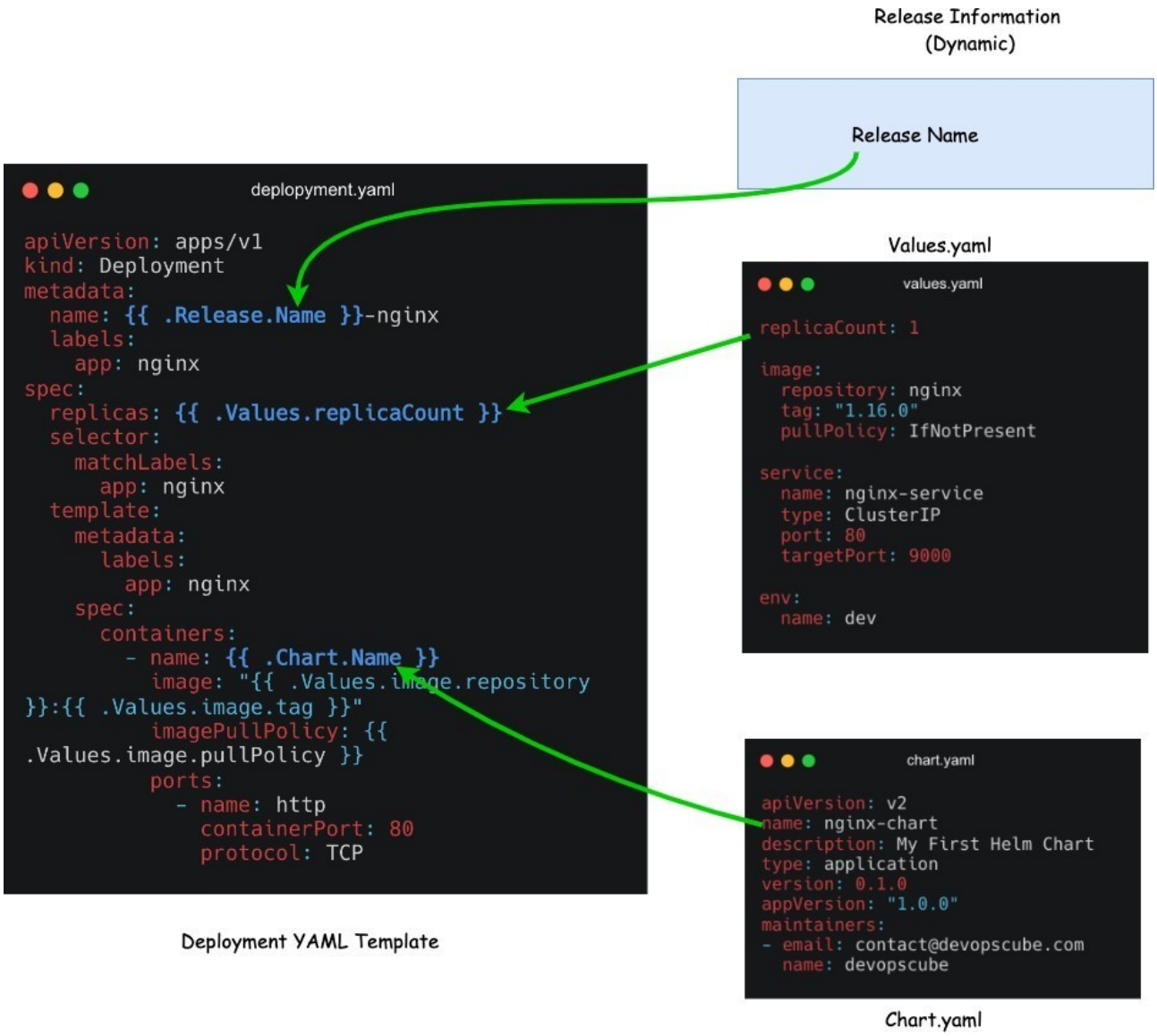
**Release** : Chaque helm chart sera déployé avec un nom de version, une Release. Si vous souhaitez utiliser le nom de la version ou accéder aux valeurs dynamiques liées à la version dans le modèle, vous pouvez utiliser l'objet **release**.

**Chart** : Si vous souhaitez utiliser les valeurs mentionnées dans le fichier chart.yaml, vous pouvez utiliser l'objet **chart**.

**Valeurs** : Tous les paramètres contenus dans le fichier values.yaml sont accessibles à l'aide de l'objet **Values**.

Plus loin dans la formation, nous examinerons en détails les Helm Builtin Object.

L'image suivante montre comment les objets intégrés sont substitués dans un modèle. Flux de travail de la substitution des directives d'un modèle Helm




Tout d'abord, vous devez déterminons quelles valeurs a changer ou ce que vous voulez modéliser. On choisit ***name***, ***replicas***, ***container name***, ***image*** et ***imagePullPolicy*** qui est surligné en gras dans le fichier YAML.

**name** : **name** *: **{{ .Release.Name }}-nginx*** : changeons le nom du déploiement à chaque fois car Helm ne nous permet pas d'installer des ressources avec le même nom. Nous allons donc «  templatiser » le nom du déploiement avec le nom de la release et insérer le chaine -**nginx**. Maintenant, si nous instancions une version en utilisant le nom frontend, le nom du déploiement sera frontend-nginx. De cette façon, nous aurons des noms uniques garantis.

**nom du conteneur** : ***{{ .Chart.Name }}*** : Pour le nom du conteneur, nous allons utiliser l'objet *Chart* et utiliser le nom du Chart dans le chart.yaml comme nom du conteneur.

**Replicas : *{{ .Values.replicaCount }}*** Nous accèderons à la valeur de la réplique à partir du fichier *values.yaml.*

**image** : ***"{{ .Values.image.repository }}:{{ .Values.image.tag }}"*** Ici, nous utilisons plusieurs directives  sur une seule ligne et nous accédons aux informations relatives au référentiel et à la balise sous la clé image à partir du fichier Values.

Voici notre fichier deployment.yaml final après avoir appliqué les modèles. La partie modélisée est mise en évidence en gras. Remplacez le contenu du fichier de déploiement par ce qui suit.

```
ludo@kubernetes:$ vim templates/deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
 name: {{ .Release.Name }}-nginx
 labels:
  app: nginx
spec:
 replicas: {{ .Values.replicaCount }}
 selector:
  matchLabels:
   app: nginx
 template:
  metadata:
   labels:
    app: nginx
  spec:
   containers:
    - name: {{ .Chart.Name }}
      image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
      imagePullPolicy: {{ .Values.image.pullPolicy }}
      ports:
       - name: http
         containerPort: 80
         protocol: TCP
      volumeMounts:
       - name: nginx-index-file
         mountPath: /usr/share/nginx/html/
   volumes:
    - name: nginx-index-file
      configMap:
       name: index-html-configmap
```

Créer le fichier service.yaml et copier le contenu suivant.

```
ludo@kubernetes:$ vim templates/service.yaml
apiVersion: v1
kind: Service
metadata:
 name:  {{ .Release.Name }}-service
spec:
```

```
  selector:
    app.kubernetes.io/instance:  {{ .Release.Name }}
  type:  {{ .Values.service.type }}
  ports:
    - protocol:  {{ .Values.service.protocol | default "TCP" }}
      port:  {{ .Values.service.port }}
      targetPort: {{ .Values.service.targetPort }}
```

Créer le fichier configmap.yaml et copier le contenu suivant.

```
ludo@kubernetes:$ vim templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-index-html-configmap
  namespace: default
data:
  index.html: |
    <html>
    <h1>Welcome</h1>
    </br>
    <h1>Hi! I got deployed in {{ .Values.env.name }} Environment using Helm Chart </h1>
    </html>
```

## values.yaml

Le fichier values.yaml contient toutes les valeurs qui doivent être substituées dans les directives que nous utilisons dans nos templates. Par exemple, le modèle deployment.yaml contient une directive pour obtenir le référentiel d'images, le tag et la politique de téléchargement à partir du fichier values.yaml.

Dans le fichier values.yaml suivant, nous avons des paires clés : valeur.repository, tag et pullPolicy imbriquées sous la clé image. C'est la raison pour laquelle nous avons utilisé ***.Values.image.repository***

Remplacez le contenu du fichier values.yaml par défaut par ce qui suit.

```
ludo@kubernetes:$ vim values.yaml
apiVersion: v2
name: nginx-chart
description: Mon premier Chart Helm
type: application
version: 0.1.0
appVersion: "1.0.0"
maintainers:
- email: contact@ambientit.form
  name: ambientiT
```

Nous avons maintenant le chart de Nginx prêt et la structure finale du répertoire ressemble à ce qui suit.

```
nginx-chart
├── Chart.yaml
├── charts
├── templates
│   ├── configmap.yaml
│   ├── deployment.yaml
│   └── service.yaml
```

## Valider le Chart de Helm

Maintenant, pour s'assurer que notre Chart est valide et que toutes les indentations et Objets sont correctes, nous pouvons exécuter la commande ci-dessous. Assurez-vous d'être dans le répertoire chart.

```
ludo@kubernetes:$ helm lint .

Ou

ludo@kubernetes:$ helm lint /chemin/vers/nginx-chart
```

S'il n'y a pas d'erreur, le résultat sera le suivant Assurez-vous d'être dans le répertoire chart.

```
ludo@kubernetes:$ helm lint .
==> Linting
./nginx [INFO]
Chart.yaml :
```

Pour valider si les valeurs sont substituées dans les templates, vous pouvez afficher les fichiers YAML « templés » en utilisant la commande suivante. Cette commande générera et affichera tous les fichiers manifestes avec les valeurs substituées.

```
ludo@kubernetes:$ helm template .
```

Nous allons utiliser l'option --dry-run pour s'assurer du bon déploiement. Cette commande n'instancie pas le Chart sur Kubernetes et, en cas de problème, elle affichera les erreurs.

```
ludo@kubernetes:$ helm install --dry-run my-nginx nginx-chart
```

## Déployer le Chart Helm

Nous sommes maintenant prêts à déployer. Exécutez la commande suivante où nginx-release est le nom de la version et nginx-chart est le nom du Chart. Cette commande installe nginx-chart dans l'espace de noms par défaut

Nous allons utiliser l'option --dry-run pour vérifier. Cette commande n'instancie pas le Chart sur le cluster et, en cas de problème, affiche les erreurs.

```
ludo@kubernetes:$ helm install my-nginx nginx-chart
```

On affiche notre release

```
ludo@kubernetes:$ helm list
```

On supprime notre instance du chart nginx-chart

```
ludo@kubernetes:$ helm uninstall  my-nginx
```

Créons un nouveau chart et Allons plus loin avec HELM

```
ludo@kubernetes:$ helm helm create mychart
Création de mon Chart
```

Et ce que nous allons faire, c'est... *les supprimer tous !* De cette façon, nous pouvons travailler à partir de zéro.

```
ludo@kubernetes:$ rm -rf mychart/templates/*
```

# Les templates

Le premier modèle que nous allons créer sera un ConfigMap. Dans Kubernetes, un ConfigMap est simplement un objet pour stocker des données de configuration. D'autres objets, comme les pods, peuvent accéder aux données d'un ConfigMap.

Les ConfigMaps étant des ressources de base, elles constituent pour nous un excellent point de départ.

Commençons par créer un fichier appelé *mychart/templates/configmap.yaml* :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mychart-configmap
data:
  myvalue: "Hello World"
```

**TIP :** Les noms des modèles ne suivent pas un modèle de dénomination rigide. Toutefois, nous recommandons d'utiliser l'extension. yaml pour les fichiers YAML et .tpl pour les aides.

Le fichier YAML ci-dessus est un ConfigMap simple, contenant le minimum de champs nécessaires.  Il est tout à fait possible de mettre un fichier YAML simple comme celui-ci dans le répertoire mychart/templates/. Lorsque Helm lira ce modèle, il l'enverra simplement tel quel à Kubernetes.

Avec ce modèle simple, nous avons maintenant un chart installable. Et nous pouvons l'installer comme suit :

```
$ helm install full-coral  ./mychart
NOM : full-coral
DERNIER DÉPLOIEMENT : Tue Nov 1 17:36:01 2016
NAMESPACE : default
STATUT : DEPLOYED
RÉVISION : 1
SUITE DE TEST : Aucune
```

En utilisant Helm, nous pouvons récupérer la version et voir le modèle qui a été chargé.

```
$ helm get manifest full-coral

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mychart-configmap
data:
  myvalue: "Hello World"
```

La commande *helm get manifest* prend un nom de Release (full-coral) et affiche toutes les ressources Kubernetes déployées. Chaque fichier commence par trois tirets - - - pour indiquer le début d'un document YAML, puis est suivi d'une ligne de commentaire générée automatiquement qui nous indique quel fichier de modèle a généré ce document YAML.

À partir de là, nous pouvons voir que les data correspondent exactement à ce que nous avons mis dans notre fichier configmap.yaml.

Nous pouvons maintenant désinstaller notre release : helm uninstall full-coral.

## Ajout d'un appel de modèle simple

Le fait de coder en dur la clé ***name:*** dans une ressource est généralement considéré comme une mauvaise pratique. Les noms des ressources doivent être propres à une release. Nous pourrions donc vouloir générer un champ ***name:*** en insérant le nom de la release.

Modifions configmap.yaml en conséquence.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
```

Le principal changement concerne la valeur du champ ***name :*** qui est désormais

***{{ .Release.Name }}-configmap.***

Une directive de modèle est entourée de blocs {{ }}.

La directive de modèle ***{{ .Release.Name }}*** injecte le nom de la Release dans le modèle. Les valeurs transmises dans un modèle peuvent être considérées comme des ***objets*** *à espace de noms*, où le point (.) sépare chaque élément à espace de noms.

Le point qui précède Release indique que nous commençons par l'espace de noms le plus élevé pour cette portée (nous parlerons de la portée dans un instant). Nous pourrions donc lire »: ***.Release.Name*** comme "commencer par l'espace de noms supérieur, trouver l'objet *Release*, puis chercher à l'intérieur de celui-ci un objet appelé Name".

L'objet *Release* est l'un des objets intégrés à Helm, et nous l'aborderons plus en détail ultérieurement. Mais pour l'instant, il suffit de dire que cet objet affichera le nom de la *Release* que Heml attribue à notre *Release*.

Lorsque nous installerons notre ressource, nous verrons immédiatement le résultat de l'utilisation de cette directive template :

```
$ helm install clunky-serval ./mychart
NAME: clunky-serval
LAST DEPLOYED: Tue Nov  1 17:45:37 2016
NAMESPACE: default
STATUS: DEPLOYED
REVISION: 1
TEST SUITE: None
```

Vous pouvez éxécutez : *helm get manifest clunky-serval* pour voir l'intégralité du YAML généré.

Notez que le nom de la ConfigMap dans Kubernetes est ***clunky-serval-configmap*** au lieu de full-coral-configmap précédemment.

À ce stade, nous avons vu les modèles dans leur forme la plus basique : des fichiers YAML contenant des directives de modèle intégrées dans un bloc {{ }}. Dans la prochaine partie, nous examinerons les modèles plus en profondeur. Mais avant d'aller plus loin, il y a une petite astuce qui peut rendre la construction de modèles plus rapide : Lorsque vous voulez tester le rendu des modèles, mais sans installer, vous pouvez utiliser ***helm install --debug --dry-run goodly-guppy ./mychart***. Cela affiche les modèles et vous affichera le modèle rendu afin que vous puissiez voir le résultat :

```
$ helm install --debug --dry-run goodly-guppy ./mychart
install.go:149: [debug] Original chart version: ""
install.go:166: [debug] CHART PATH: /Users/ninja/mychart

NAME: goodly-guppy
LAST DEPLOYED: Thu Dec 26 17:24:13 2019
NAMESPACE: default
STATUS: pending-install
REVISION: 1
TEST SUITE: None
USER-SUPPLIED VALUES:
{}

COMPUTED VALUES:
affinity: {}
fullnameOverride: ""
image:
  pullPolicy: IfNotPresent
  repository: nginx
imagePullSecrets: []
```

L'utilisation de ***--dry-run*** facilitera le test de votre code, mais ne garantira pas que Kubernetes lui-même acceptera les modèles que vous générez. Il est préférable de ne pas supposer que votre Chart s'installera simplement parce que ***--dry-run*** fonctionne.

# Objets intégrés

Les objets (**built-in values**) sont transmis à un modèle par le moteur de modèles. Votre code peut également transmettre des objets (nous en verrons des exemples lorsque nous étudierons les instructions **with** et range). Il existe même quelques façons de créer de nouveaux objets dans vos modèles, comme avec la fonction tuple que nous verrons plus tard également.

Les objets peuvent être simples et n'avoir qu'une seule valeur. Ils peuvent également contenir d'autres objets ou fonctions. Par exemple, l'objet Release contient plusieurs objets (comme **Release.Name**) et **l'objet .Files** possède quelques fonctions.

Dans la section précédente, nous avons utilisé **{{ .Release.Name }}** pour insérer le nom d'une Release dans un modèle. La Release est l'un des objets de premier niveau auxquels vous pouvez accéder dans vos modèles.

- **Release** : Cet objet décrit la Release elle-même. Il contient plusieurs objets :
    - **Release.Name**: : Le nom de la  Release
    - **Release.Namespace** : L'espace de noms dans lequel l'objet doit être crée.
    - **Release.IsUpgrade** : La valeur de ce paramètre est vraie si l'opération en cours est une mise à niveau ou un retour en arrière Rollback.
    - **Release.IsInstall**: : Cette valeur vaut true si l'opération en cours est une installation.
    - **Release.Revision** : Le numéro de révision de cette Release. Lors de l'installation, ce numéro est égal à 1. Il est incrémenté à chaque mise à niveau et à chaque retour en arrière.

- **Values** : Valeurs transmises au modèle à partir du fichier values.yaml et de fichiers fournis par l'utilisateur. Par défaut, Values est vide.
- **Chart** : Le contenu du fichier Chart.yaml. Toutes les données contenues dans Chart.yaml seront accessibles ici. Par exemple **{{ .Chart.Name }}-{{ .Chart.Version }}** affichera mychart-0.1.0.
- **Subcharts** : Cela permet d'accéder à la portée (.Values, .Charts, .Releases, etc.) des sous-chart du parent. Par exemple, .Subcharts.mySubChart.myValue pour accéder à myValue dans le Chart mySubChart.
- **Files** : Cette option permet d'accéder à tous les fichiers non spéciaux d'un Chart. Vous ne pouvez pas l'utiliser pour accéder aux modèles, mais vous pouvez l'utiliser pour accéder à d'autres fichiers du Chart.
    - **Files.Get** est une fonction permettant d'obtenir un fichier par son nom (.Files.Get config.ini)

    - **Files.GetBytes** est une fonction qui permet d'obtenir le contenu d'un fichier sous la forme d'un Chart d'octets plutôt que d'une chaîne de caractères. C'est utile pour des choses comme les images.
    - **Files.Glob** est une fonction qui renvoie une liste de fichiers dont les noms correspondent au modèle de glob du shell donné.
    - **Files.Lines** est une fonction qui lit un fichier ligne par ligne. Cette fonction est utile pour parcourir chaque ligne d'un fichier.
    - **Files.AsSecrets** est une fonction qui renvoie les corps de fichiers sous forme de chaînes encodées en base 64.
    - **Files.AsConfig** est une fonction qui renvoie les corps de fichiers sous la forme d'un chart YAML.
- **Capabilities** : Cette section fournit des informations sur les Capabilities prises en charge par le cluster Kubernetes.
    - **Capabilities.APIVersions** est un ensemble de versions.
    - **Capabilities.APIVersions.Has** $version indique si une version (par exemple, batch/v1) ou une ressource (par exemple, apps/v1/Deployment) est disponible sur le cluster.
    - **Capabilities.KubeVersion** et **Capabilities.KubeVersion.Version** est la version de Kubernetes.
    - **Capabilities.KubeVersion.Major** est la version majeure de Kubernetes.
    - **Capabilities.KubeVersion.Minor** est la version mineure de Kubernetes.
    - **Capabilities.HelmVersion** est l'objet contenant les détails de la version de Helm, il s'agit de la même sortie que la version de Helm.
    - **Capabilities.HelmVersion.Version** est la version actuelle de Helm au format semver.
    - **Capabilities.HelmVersion.GitCommit** est le sha1 git de Helm.
    - **Capabilities.HelmVersion.GitTreeState** est l'état de l'arbre git de Helm.
    - **Capabilities.HelmVersion.GoVersion** est la version du compilateur Go utilisée.
- **Template** : Contient des informations sur le modèle en cours d'exécution.
    - **Template.Name** : chemin d'accès au modèle actuel (par exemple, mychart/templates/mytemplate.yaml).
    - **Template.BasePath** : Le chemin d'accès au répertoire des modèles de la chart en cours (par exemple, mychart/templates).

Les Objets intégrées, built-in values commencent toujours par une lettre majuscule. Ceci est conforme à la convention de nommage de Go. Lorsque vous créez vos propres noms, vous êtes libre d'utiliser la convention qui convient à votre équipe. Certaines équipes, comme celles dont vous pouvez voir les Charts sur Artifact Hub,

choisissent de n'utiliser que des lettres minuscules initiales afin de distinguer les noms locaux des noms intégrés.

Dans la section précédente, nous avons examiné les objets intégrés offerts par les modèles Helm. L'un de ces objets intégrés est **Values**. Cet objet permet d'accéder aux valeurs transmises au Chart. Son contenu provient de plusieurs sources :

- Le fichier values.yaml dans le Chart
- S'il s'agit d'un Subchart, le fichier values.yaml d'un Chart parent.
- Un fichier de valeurs s'il est transmis à ***helm install*** ou ***helm upgrade*** avec l'option ***-f*** *(helm install -f myvals.yaml ./mychart)*.
- Paramètres individuels passés avec *--set* (comme ***helm install --set foo=bar ./mychart***)

La liste ci-dessus est traité par ordre de spécificité : values.yaml est la valeur par défaut, qui peut être remplacée par le fichier values.yaml d'un Chart parent, qui peut à son tour être remplacé par un fichier de valeurs fourni par l'utilisateur, qui peut à son tour être remplacé par les paramètres *--set*.

Les fichiers de valeurs sont de simples fichiers YAML. Modifions le fichier mychart/values.yaml, puis le modèle ConfigMap.

En supprimant toutes les valeurs par défaut du fichier values.yaml, nous ne définirons qu'un seul paramètre :

```
favoriteDrink: coffee
```

Nous pouvons maintenant l'utiliser à l'intérieur d'un modèle :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ .Values.favoriteDrink }}
```

Remarquez qu'à la dernière ligne, nous accédons à ***favoriteDrink*** en tant qu'attribut de Values : ***{{ .Values.favoriteDrink }}***.

Voyons ce que cela donne.

```
$ helm install geared-marsupi ./mychart --dry-run --debug
install.go:158 : [debug] Version originale du Chart : ""
install.go:175 : [debug] CHART PATH : /home/bagratte/src/playground/mychart
NAME: geared-marsupi
LAST DEPLOYED: Wed Feb 19 23:21:13 2020
NAMESPACE: default
STATUS: pending-install
REVISION: 1
TEST SUITE: None
USER-SUPPLIED VALUES:
{}
```

```
COMPUTED VALUES:
favoriteDrink: coffee

HOOKS:
MANIFEST:
---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: geared-marsupi-configmap
data:
  myvalue: "Hello World"
  drink: coffee
```

Parce que ***favoriteDrink*** est défini dans le fichier values.yaml à *coffee*, c'est la valeur affichée dans le template. Nous pouvons simplement remplacer (surcharger) cette valeur en ajoutant un flag *--set* dans notre appel à ***helm install*** :

```
$ helm install solid-vulture ./mychart --dry-run --debug --set favoriteDrink=slurm
install.go:158 : [debug] Version originale du Chart : ""
install.go:175 : [debug] CHART PATH : /home/bagratte/src/playground/mychart

NOM : solid-vulture
DERNIER DÉPLOIEMENT : Wed Feb 19 23:25:54 2020
NAMESPACE : default
STATUT : en cours d'installation
RÉVISION : 1
SUITE DE TEST : Aucune
LES VALEURS FOURNIES PAR L'UTILISATEUR :
drink favoriteDrinke : slurm

VALEURS CALCULÉES :
drink favoriteDrinke : slurm

COUCHES :
MANIFESTER :
---
# Source : mychart/templates/configmap.yaml
apiVersion : v1
kind : ConfigMap
metadata :
 nom : solid-vulture-configmap
data :
 myvalue : "Hello World"
 drink : slurm
```

Puisque *--set* a une priorité plus élevée que le fichier values.yaml par défaut, notre modèle génère ***drink: slurm***.

Les fichiers de valeurs peuvent également contenir un contenu plus structuré. Par exemple, nous pourrions créer une section favorite dans notre fichier values.yaml, puis y ajouter plusieurs clés :

```
favorite:
  drink: coffee
  food: pizza
```

Nous devons maintenant modifier légèrement le modèle :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ .Values.favorite.drink }}
  food: {{ .Values.favorite.food }}
```

Bien qu'il soit possible de structurer les données de cette manière, il est recommandé de conserver des arbres de valeurs peu profonds et de privilégier la planéité. Lorsque nous aborderons l'attribution de valeurs aux sous-Charts, nous verrons comment les valeurs sont nommées à l'aide d'une structure arborescente.

## Suppression d'une clé par défaut

Si vous devez supprimer une clé des valeurs par défaut, vous pouvez remplacer la valeur de la clé par null, auquel cas Helm supprimera la clé de la fusion des valeurs remplacées.

Par exemple, la charte Drupal stable permet de configurer la ***livenessProbe***, dans le cas où vous configurez une image personnalisée. Voici les valeurs par défaut :

```
livenessProbe:
  httpGet:
    path: /user/login
    port: http
  initialDelaySeconds: 120
```

Si vous tentez de remplacer le gestionnaire **livenessProbe** par **exec** au lieu de **httpGet** en utilisant **--set livenessProbe.exec.command=[cat,docroot/CHANGELOG.txt]**, Helm fusionnera les clés par défaut et les clés remplacées, ce qui donnera le code YAML suivant :

```
livenessProbe:
  httpGet:
    path: /user/login
    port: http
  exec:
    command:
    - cat
    - docroot/CHANGELOG.txt
  initialDelaySeconds: 120
```

Cependant, Kubernetes échouerait à créer la ressource, car vous ne pouvez pas déclarer plus d'un *livenessProbe* handler. Pour y remédier, vous pouvez demander à Helm de supprimer le *livenessProbe.httpGet* en lui attribuant la valeur *null* :

```
$ helm install stable/drupal --set image=my-registry/drupal:0.1.0 --set
livenessProbe.exec.command=[cat,docroot/CHANGELOG.txt] --set
livenessProbe.httpGet=null
```

À ce stade, nous avons vu plusieurs objets intégrés et les avons utilisés pour injecter des informations dans un modèle. Nous allons maintenant nous pencher sur un autre aspect du moteur de modèles : **les fonctions et les pipelines.**

# Fonctions et pipelines

Jusqu'à présent, nous avons vu comment placer des valeurs dans un modèle. Mais ces valeurs sont placées dans le modèle sans être modifiées (Majuscule, quote. Etc..). Parfois, nous souhaitons transformer les données fournies de manière à ce qu'elles soient plus facilement utilisables.

Commençons par une bonne pratique : Lorsque nous injectons des chaînes de l'objet ***.Values*** dans le modèle, nous devons les mettre entre quote. Nous pouvons le faire en appelant la fonction ***quote*** dans la directive du modèle :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ quote .Values.favorite.drink }}
  food: {{ quote .Values.favorite.food }}
```

Les fonctions de modèle suivent la syntaxe suivante : *function Name arg1 arg2*.... Dans l'extrait ci-dessus, ***quote  .Values.favorite.drink*** appelle la fonction ***quote*** et lui transmet un seul argument.

Helm dispose de plus de 60 fonctions. Certaines d'entre elles sont définies par le langage de template Go lui-même. La plupart des autres font partie de la bibliothèque de modèles Sprig. Nous en verrons beaucoup au fur et à mesure que nous avancerons dans les exemples.

> Bien que nous parlions du "langage de template Helm" comme s'il était spécifique à Helm, il s'agit en fait d'une combinaison du langage de template Go, de quelques fonctions supplémentaires et d'une variété de wrappers pour exposer certains objets aux templates. De nombreuses ressources sur les templates Go peuvent vous être utiles pour vous familiariser avec la création de templates.

## Pipelines

L'une des caractéristiques les plus puissantes du langage de template est son concept de *pipelines*. S'inspirant d'un concept d'UNIX, les pipelines sont un outil permettant d'enchaîner une série de commandes de modèles afin d'exprimer de manière compacte une série de transformations. En d'autres termes, les pipelines sont un moyen efficace de réaliser plusieurs choses en séquence.

Réécrivons l'exemple ci-dessus en utilisant un pipeline.

```
apiVersion: v1
kind: ConfigMap
```

```
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ .Values.favorite.drink | quote }}
  food: {{ .Values.favorite.food | quote }}
```

Dans cet exemple, au lieu d'appeler ***quote ARGUMENT***, nous avons inversé l'ordre. Nous avons "envoyé" l'argument à la fonction à l'aide d'un pipeline (|) *:* ***.Values.favorite.drink | quote***. Les pipelines permettent d'enchaîner plusieurs fonctions :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ .Values.favorite.drink | quote }}
  food: {{ .Values.favorite.food | upper | quote }}
```

L'inversion de l'ordre est une pratique courante dans les modèles. Vous verrez plus souvent ***.val | quote*** que ***quote .val***. Les deux pratiques sont acceptables.

Lorsqu'il est évalué, ce modèle produit ceci :

```
# Source : mychart/templates/configmap.yaml
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: trendsetting-p-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
```

Notez que notre **pizza** originale a été transformée en "PIZZA".

Lors de l'enchaînement des arguments de cette manière, le résultat de la première évaluation (***.Values.favorite.drink***) est envoyé en tant que *dernier argument à la fonction*. Nous pouvons modifier l'exemple de la ***drink*** ci-dessus pour l'illustrer avec une fonction qui prend deux arguments : *repeat COUNT STRING* :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ .Values.favorite.drink | repeat 5 | quote }}
  food: {{ .Values.favorite.food | upper | quote }}
```

La fonction ***repeat*** (répétition) renvoie la chaîne de caractères donnée le nombre de fois indiqué, ce qui nous permet d'obtenir cette chaîne en sortie :

```
# Source: mychart/templates/configmap.yaml
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: melting-porcup-configmap
data:
  myvalue: "Hello World"
  drink: "coffeecoffeecoffeecoffeecoffee"
  food: "PIZZA"
```

## Utilisation de la fonction « par défaut»

Une fonction fréquemment utilisée dans les modèles est la fonction default : default DEFAULT_VALUE GIVEN_VALUE. Cette fonction vous permet de spécifier une valeur par défaut à l'intérieur du modèle, au cas où la valeur serait omise. Utilisons-la pour modifier l'exemple de ***drink*** ci-dessus :

```
drink: {{ .Values.favorite.drink | default "tea" | quote }}
```

Si nous procédons comme d'habitude, nous aurons notre coffee :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: virtuous-mink-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
```

Nous allons maintenant supprimer le paramètre de la drink ***favoriteDrink*** dans le fichier values.yaml :

```
favorite:
  #drink: coffee
  food: pizza
```

Maintenant, en relançant ***helm install --dry-run --debug fair-worm ./mychart***, on obtiendra ce YAML :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: fair-worm-configmap
data:
  myvalue: "Hello World"
  drink: "tea"
  food: "PIZZA"
```

Dans un Chart réel, toutes les valeurs statiques par défaut devraient se trouver dans le fichier values.yaml et ne devraient pas être répétées à l'aide de la commande default (sinon elles seraient redondantes). Cependant, la commande default est parfaite pour les valeurs calculées, qui ne peuvent pas être déclarées dans

values.yaml. Par exemple, la commande default est parfaite pour les valeurs calculées :

```
drink: {{ .Values.favorite.drink | default (printf "%s-tea" (include "fullname" .)) }}
```

Dans certains cas, une condition **if** peut être mieux adaptée que la valeur par défaut. Nous verrons cela dans la section suivante.

Les fonctions de template et les pipelines sont un moyen puissant de transformer les valeurs pour nos manifest. Mais il est parfois nécessaire d'ajouter une logique de template un peu plus sophistiquée que la simple insertion d'une chaîne de caractères. Dans la section suivante, nous examinerons les structures de contrôle fournies par le langage de template.

## Utilisation de la fonction de recherche

La fonction *lookup* peut être utilisée pour *rechercher des* ressources dans un cluster en cours d'exécution. Le synopsis de la fonction *lookup* est le suivant : *lookup apiVersion, kind, namespace, name -> resource ou resource list*.

| paramètre | type |
|---|---|
| apiVersion | Chaîne de caractères |
| kind | Chaîne de caractères |
| namespace | Chaîne de caractères |
| name | Chaîne de caractères |

Le nom et l'espace de noms sont facultatifs et peuvent être transmis sous la forme d'une chaîne vide ("").

Les combinaisons de paramètres suivantes sont possibles :

| Comportement | Fonction de recherche |
|---|---|
| kubectl get pod mypod -n mynamespace | lookup "v1" "Pod" "mynamespace" "mypod" |
| kubectl get pods -n mynamespace | lookup "v1" "Pod" "mynamespace" "" |
| kubectl get pods --all-namespaces | lookup "v1" "Pod" "" "" |
| kubectl get namespace mynamespace | lookup "v1" "Namespace" "" "mynamespace" |
| kubectl get namespaces | lookup "v1" "namespaces" "" "" |

Lorsque la recherche renvoie un objet, elle renvoie un dictionnaire. Ce dictionnaire peut être parcouru pour extraire des valeurs spécifiques.

L'exemple suivant renvoie les annotations présentes pour l'objet *mynamespace* :

```
(lookup "v1" "Namespace" "" "mynamespace").metadata.annotations
```

Lorsque la recherche renvoie une liste d'objets, il est possible d'accéder à la liste d'objets par l'intermédiaire du champ "*items*" :

```
{{ range $index, $service := (lookup "v1" "Service" "mynamespace" "").items }}
    {{/* do something with each service */}}
{{ end }}
```

Si aucun objet n'est trouvé, une valeur vide est renvoyée. Ceci peut être utilisé pour vérifier l'existence d'un objet.

Gardez à l'esprit que Helm n'est pas censé contacter le serveur API Kubernetes pendant une opération ***helm template|install|upgrade|delete|rollback --dry-run***. Pour tester la recherche sur un cluster en cours d'exécution, *helm template|install|upgrade|delete|rollback* ***--dry-run=server*** doit être utilisé à la place pour autoriser la connexion au cluster.

# Contrôle du flux

Les structures de contrôle (appelées "**actions**" dans le jargon des Template) vous permettent, de contrôler le flux de génération d'un modèle. Le langage de template de Helm fournit les structures de contrôle suivantes :

- **if/else** pour créer des blocs conditionnels
- **with** pour spécifier un champ d'application
- **range** qui fournit une boucle de type "for-each".

En outre, il fournit quelques actions pour déclarer et utiliser des éléments de *modèles nommés* :

- **define** déclare un nouveau modèle nommé à l'intérieur de votre modèle
- **template** importe un modèle nommé
- **block** déclare un type particulier de zone de template remplissable

Dans cette section, nous parlerons de *if, with* et *range*. Les autres actions sont traitées dans la section "*Modèles nommés*" plus loin dans ce guide.

## If/Else

La première structure de contrôle que nous allons examiner permet d'inclure de manière conditionnelle des blocs de texte dans un modèle. Il s'agit du bloc **if/else**.

La structure de base d'une conditionnelle est la suivante :

```
{{ if PIPELINE }}
  # Do something
{{ else if OTHER PIPELINE }}
  # Do something else
{{ else }}
  # Default case
```

```
{{ end }}
```

Remarquez que nous parlons maintenant de *pipelines* et non plus de valeurs. La raison en est qu'il est clair que les structures de contrôle peuvent exécuter un pipeline entier, et pas seulement évaluer une valeur.

Un pipeline est évalué comme *faux* si la valeur est :

- un booléen faux
- un zéro numérique
- une chaîne vide
- un nil (vide ou nul)
- une collection vide (map, slice, tuple, dict, array)

Dans toutes les autres conditions, la condition est vraie.

Ajoutons une simple condition à notre ConfigMap. Nous ajouterons un autre paramètre si ***drink*** est **coffee** :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ .Values.favorite.drink | default "tea" | quote }}
  food: {{ .Values.favorite.food | upper | quote }}
  {{ if eq .Values.favorite.drink "coffee" }}mug: "true"{{ end }}
```

Puisque nous avons commenté ***drink: coffee*** dans notre dernier exemple, la sortie ne devrait pas inclure de ***mug: "true"***. Mais si nous ajoutons cette ligne dans notre fichier values.yaml, la sortie devrait ressembler à ceci :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: eyewitness-elk-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
  mug: "true"
```

## Contrôle des espaces

Nous devrions jeter un coup d'œil sur la façon dont l'espace est contrôlé dans les modèles. Reprenons l'exemple précédent et formatons-le pour qu'il soit plus facile à lire :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
```

```
  myvalue: "Hello World"
  drink: {{ .Values.favorite.drink | default "tea" | quote }}
  food: {{ .Values.favorite.food | upper | quote }}
  {{ if eq .Values.favorite.drink "coffee" }}
    mug: "true"
  {{ end }}
```

Au départ, cela semble correct. Mais si nous le soumettons au moteur de template, nous obtiendrons un résultat malheureux :

```
$ helm install --dry-run --debug ./mychart
SERVER : "localhost:44134"
CHART PATH : /Users/mattbutcher/Code/Go/src/helm.sh/helm/_scratch/mychart
Erreur : Erreur d'analyse YAML sur mychart/templates/configmap.yaml : erreur de conversion de YAML en JSON : yaml : ligne 9 : n'a pas trouvé la clé attendue
```

Qu'est-ce qui s'est passé ? Nous avons généré un YAML incorrect à cause des espaces ci-dessus.

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: eyewitness-elk-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
    mug: "true"
```

***mug*** est incorrectement indenté. Il suffit de mettre cette ligne en retrait et de réexécuter le programme :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ .Values.favorite.drink | default "tea" | quote }}
  food: {{ .Values.favorite.food | upper | quote }}
  {{ if eq .Values.favorite.drink "coffee" }}
  mug: "true"
  {{ end }}
```

Lorsque nous envoyons cela, nous obtenons un YAML qui est valide, mais qui a quand même un aspect un peu bizarre :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: telling-chimp-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"

  mug: "true"
```

Remarquez que nous avons une ligne vide dans notre YAML. Pourquoi ? Lorsque le moteur de template s'exécute, il *supprime* le contenu à l'intérieur de {{ and }}, mais il laisse le reste de l'espace blanc tel quel.

YAML attribue une signification à l'espace, de sorte que la gestion de l'espace devient très importante. Heureusement, les modèles Helm disposent de quelques outils pour nous aider.

Tout d'abord, la syntaxe des accolades des déclarations de modèles peut être modifiée par l'ajout de caractères spéciaux pour indiquer au moteur de modèles qu'il doit supprimer les espaces. *{{-* (avec le tiret et l'espace ajoutés) indique que les espaces doivent être grignotés à gauche, tandis que *-}}* signifie que les espaces à droite doivent être supprimés. *Attention ! Les nouvelles lignes sont des espaces !*

Assurez-vous qu'il y a un espace entre le - et le reste de votre directive. ***{{- 3 }}*** signifie "couper les espaces à gauche et afficher 3", tandis que ***{{-3 }}*** signifie "*afficher -3*".

En utilisant cette syntaxe, nous pouvons modifier notre modèle pour nous débarrasser de ces nouvelles lignes :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ .Values.favorite.drink | default "tea" | quote }}
  food: {{ .Values.favorite.food | upper | quote }}
  {{- if eq .Values.favorite.drink "coffee" }}
  mug: "true"
  {{- end }}
```

Afin de clarifier ce point, ajustons le yaml et remplaçons par un ***** chaque espace qui sera supprimé conformément à cette règle. Un ***** à la fin de la ligne indique un caractère de retour à la ligne qui sera supprimé.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ .Values.favorite.drink | default "tea" | quote }}
  food: {{ .Values.favorite.food | upper | quote }}*
**{{- if eq .Values.favorite.drink "coffee" }}
  mug: "true"*
**{{- end }}
```

En gardant cela à l'esprit, nous pouvons exécuter notre modèle avec Helm et voir le résultat :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: clunky-cat-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
  mug: "true"
```

Soyez prudent avec les modificateurs "chomping". Les modificateurs de chomping définissent si les **sauts de ligne** doivent être conservés ou supprimés lors du rendu des chaînes de caractères dans un fichier YAML. . Attention, Il est facile de faire erreurs accidentellement :

```
  food: {{ .Values.favorite.food | upper | quote }}
  {{- if eq .Values.favorite.drink "coffee" -}}
  mug: "true"
  {{- end -}}
```

Cela produira ***food: "PIZZA "mug: "true"*** parce qu'il a supprimé des nouvelles lignes des deux côtés.

Les modificateurs de chomping sont particulièrement utiles dans les templates Helm lorsque vous souhaitez générer des fichiers YAML complexes où la mise en forme du texte et des chaînes multi-lignes a une importance particulière. Par exemple :

- Créer des valeurs longues de configuration (comme des scripts ou des commandes).
- Gérer les fichiers de configuration d'outils externes avec des formats multi-lignes (comme des configurations Docker, Kubernetes, etc.).

En utilisant ces modificateurs, vous pouvez contrôler précisément comment ces chaînes **sont formatées dans le fichier final, ce qui est essentiel pour le bon fonctionnement de** vos déploiements Kubernetes via Helm.

Enfin, il est parfois plus facile d'indiquer au système de templates comment indenter, plutôt que d'essayer de maîtriser l'espacement des directives de templates. Pour cette raison, vous pouvez parfois trouver utile d'utiliser la fonction ***indent*** .

***{{ indent 2 "mug:true" }}*** *ou* ***{{ .Values.mytext | indent 4 }}***

# Scope, le champ d'application à l'aide de with

Dans Helm, le mot-clé **with** est utilisé pour modifier le **scope** (champ d'application, porté) d'une expression dans un template. En d'autres termes, **with** permet de changer temporairement le contexte dans lequel une expression est évaluée, ce qui peut être utile pour simplifier l'accès aux données dans une structure imbriquée ou pour travailler avec un sous-ensemble de variables.

### Comprendre le concept de "scope"

Le **scope** (la portée) détermine où et comment une variable ou un objet est accessible dans un template Helm. Par défaut, lorsque vous accédez à des variables ou à des données dans un template Helm, vous le faites dans le scope global, c'est-à-dire dans le contexte général du fichier de template.

Cependant, certaines fois, vous souhaitez travailler avec un sous-ensemble des données ou accéder à un champ spécifique sans avoir à répéter son chemin complet à chaque fois. **with** permet de modifier le contexte temporairement pour rendre le code plus lisible et plus efficace.

### Syntaxe de with

```
{{ with .variable}}
  # restricted scope
{{ end }}
```

La syntaxe de with est similaire à celle d'une simple instruction if :

**with** prend une expression, souvent une variable, et établit un nouveau scope pour la section de code qui suit. La variable **.** à l'intérieur de **with** fait référence à l'objet passé à **with**.

**end** : Après le bloc de code, vous devez utiliser **end** pour fermer le bloc **with**.

Lorsque vous utilisez **with**, l'objet référencé devient temporairement le point d'accès principal, et vous n'avez pas besoin de répéter le chemin complet pour accéder à ses propriétés ou sous-propriétés.

Par exemple, nous avons travaillé avec ***.Values.favorite***.

Réécrivons notre ConfigMap pour modifier la portée. Et la faire pointer sur ***.Values.favorite*** :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  {{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
```

```
  food: {{ .food | upper | quote }}
  {{- end }}
```

Notez que nous avons supprimé le conditionnel ***if*** de l'exercice précédent parce qu'il est maintenant inutile - le bloc après ***with*** ne s'exécute que si la valeur de PIPELINE n'est pas vide.

**{{- with .Values.favorite }}** : Ce bloc **with** permet de changer temporairement le contexte pour accéder à .Values.favorite. Si **.Values.favorite** est défini dans le fichier values.yaml, alors vous pouvez accéder aux champs de cet objet directement sans avoir à écrire tout le chemin à chaque fois.

**.Values.favorite** : Cela fait référence à la clé favorite dans le fichier values.yaml. Si la clé favorite existe et contient des sous-champs comme ***drink*** et ***food***, vous pourrez y accéder directement dans le bloc ***with***.

**À l'intérieur du bloc with**

```
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  {{- end }}
```

**drink: {{ .drink | default "tea" | quote }}** :

> **.drink** : Accède à la sous-clé drink dans l'objet favorite (par exemple, si la valeur dans values.yaml est favorite: { drink: "coffee" }, .drink serait coffee).
>
> **default "tea"** : La fonction **default** permet de définir une valeur par défaut si drink n'est pas défini. Si **.drink** est **nil** ou non défini, alors "tea" sera utilisé comme valeur.
>
> **quote** : La fonction **quote** ajoute des guillemets autour de la valeur (cela transformera "tea" en "\"tea\"").

**food: {{ .food | upper | quote }}** :

> **.food** : Accède à la sous-clé food dans l'objet favorite.
>
> **upper** : La fonction **upper** transforme la chaîne en majuscules.
>
> **quote** : Comme pour drink, **quote** ajoutera des guillemets autour de la valeur.

Remarquez que nous pouvons maintenant faire référence à ***.drink*** et ***.food*** sans les parents. C'est parce que l'instruction ***with*** fait pointer **.** sur ***.Values.favorite***. Le **.** est réinitialisé à sa portée précédente après ***{{ end }}***.` Cela signifie que tout ce qui suit après ce bloc reviendra au scope global, et **.** ne fait plus référence à **.Values.favorite**.

Mais voici une mise en garde ! À l'intérieur de la portée restreinte, vous ne pourrez pas accéder aux autres objets de la portée parentale en utilisant ... Ceci, par exemple, échouera :

```
{{- with .Values.favorite }}
drink: {{ .drink | default "tea" | quote }}
food: {{ .food | upper | quote }}
release: {{ .Release.Name }}
{{- end }}
```

Elle produira une erreur car Release.Name n'est pas dans la portée restreinte de ... Cependant, si nous intervertissons les deux dernières lignes, tout fonctionnera comme prévu car la portée est réinitialisée après {{ end }}.

> **with .Values.favorite** change le contexte de **.** à **.Values.favorite**. Cela signifie que **.** (l'objet de travail) représente maintenant **.Values.favorite**.
>
> À l'intérieur de ce bloc **with**, tu peux accéder à **drink**, **food**, ou toute autre clé à l'intérieur de **.Values.favorite**. Cependant, **Release.Name** est dans un autre contexte (le contexte global de Helm), et tu ne peux pas y accéder directement à partir de la portée restreinte définie par **with**.

**L'erreur**

En tentant d'accéder à **Release.Name** dans ce contexte, Helm échouera parce qu'il ne trouve pas cette clé dans la portée actuelle (qui est **.Values.favorite**).

Accéder à des variables de la portée parentale à l'intérieur de **with**, une méthode courante consiste à les transmettre explicitement dans la portée en utilisant une variable intermédiaire. Par exemple, en utilisant la fonction **$** pour capturer la portée parentale :

Nous pouvons donc également utiliser **$** pour accéder à l'objet **Release.Name** à partir du champ d'application parent. $ est mappé à l'étendue racine lorsque l'exécution du modèle commence et il ne change pas pendant l'exécution du modèle. Ce qui suit fonctionnerait également :

```
{{- with .Values.favorite }}
drink: {{ .drink | default "tea" | quote }}
food: {{ .food | upper | quote }}
release: {{ $.Release.Name }}
{{- end }}
```

**$release := .Release.Name** crée une variable **$release** qui contient **.Release.Name** dans la portée globale avant d'entrer dans le bloc **with**.

Ensuite, à l'intérieur du bloc **with**, tu peux utiliser **$release** pour faire référence à **.Release.Name**. Cela fonctionne parce que **$release** est dans la portée globale, alors que **.** à l'intérieur de **with** pointe vers **.Values.favorite**.

```
{{- $release := .Release.Name }}
{{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  release: {{ $.Release.Name }}
{{- end }}
```

# Boucle avec l'action de range

Dans Helm, vous pouvez utiliser l'action **range** pour itérer sur des éléments dans une liste ou un dictionnaire (map). C'est un peu comme une boucle for dans d'autres langages de programmation. L'action **range** permet de parcourir une collection de données et d'effectuer une action pour chaque élément.

De nombreux langages de programmation permettent d'effectuer des boucles à l'aide de boucles foreach ou d'autres mécanismes similaires. Dans le langage de modèle Helm, la façon d'itérer à travers une collection est d'utiliser l'opérateur range .

### Itération sur une liste

Pour commencer, ajoutons une liste de garnitures de pizza à notre fichier values.yaml. Si vous avez une liste dans votre fichier values.yaml et que vous souhaitez itérer sur chaque élément de cette liste, vous pouvez utiliser range ainsi :

```
favorite:
  drink: coffee
  food: pizza
pizzaToppings:
  - mushrooms
  - cheese
  - peppers
  - onions
```

Nous avons maintenant une liste (appelée slice dans les modèles) de pizzaToppings. Nous pouvons modifier notre modèle pour printer cette liste dans notre ConfigMap :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  {{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  {{- end }}
  toppings: |-
    {{- range .Values.pizzaToppings }}
    - {{ . | title | quote }}
    {{- end }}
```

Nous pouvons utiliser ***$*** pour accéder à la liste ***Values.pizzaToppings*** à partir de la portée parente. ***$*** est mappé à la portée racine lorsque l'exécution du modèle commence et il ne change pas pendant l'exécution du modèle. Ce qui suit fonctionnerait également :

```
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  {{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  toppings: |-
    {{- range $.Values.pizzaToppings }}
    - {{ . | title | quote }}
    {{- end }}
  {{- end }}
```

Examinons de plus près la liste ***toppings :***. La fonction ***range*** va "parcourir" (itérer) la liste ***pizzaToppings***. Mais il se passe maintenant quelque chose d'intéressant. Tout comme avec la portée définit par le . , l'opérateur range fait de même. À chaque fois que l'on parcourt la boucle, . est défini sur la garniture de pizza actuelle. En d'autres termes, la première fois, . est défini sur les champignons. Lors de la deuxième itération, elle est remplacée par du fromage, et ainsi de suite.

Nous pouvons envoyer la valeur de **.** directement dans un pipeline, donc lorsque nous faisons ***{{ . | titre | citation }} ,*** il envoie . à title (title case function) et ensuite à ***quote***. Si nous exécutons ce modèle, la sortie donnera :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: edgy-dragonfly-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
  toppings: |-
    - "Mushrooms"
    - "Cheese"
    - "Peppers"
    - "Onions"
```

Dans cet exemple, la ligne toppings : |- déclare une chaîne de plusieurs lignes. Notre liste de toppings n'est donc pas une liste YAML. C'est une grosse chaîne de caractères. Pourquoi faisons-nous cela ? Parce que les data de ConfigMaps sont composées de paires clé/valeur, où la clé et la valeur sont de simples chaînes de caractères.

Le marqueur |- en YAML prend une chaîne de plusieurs lignes. Cette technique peut s'avérer utile pour intégrer de grands blocs de données dans vos manifestes, comme illustré ici.

Il est parfois utile de pouvoir créer rapidement une liste à l'intérieur de votre modèle, puis d'itérer sur cette dernière. Les modèles Helm disposent d'une fonction qui facilite cette opération : ***tuple***. En informatique, un ***tuple*** est une collection de type liste de taille fixe, mais avec des types de data arbitraires. Ceci traduit à peu près la façon dont un ***tuple*** est utilisé.

```
  sizes: |-
    {{- range tuple "small" "medium" "large" }}
    - {{ . }}
    {{- end }}
```

Le résultat est le suivant :

```
  sizes: |-
    - small
    - medium
    - large
```

Outre les ***listes*** et les ***tuples***, ***range*** peut-être utilisé pour itérer sur des collections ayant une clé et une valeur (comme une map ou un dict). Nous verrons comment procéder dans la section suivante, lorsque nous présenterons les variables modèles.

# Variables

Avec les fonctions, les pipelines, les objets et les structures de contrôle, nous pouvons nous tourner vers l'une des fonctionnalités les plus fondamentales de nombreux langages de programmation : **Les variables**. Dans les modèles, elles sont moins fréquemment utilisées. Mais nous verrons comment les utiliser pour simplifier le code et pour mieux utiliser ***with*** et ***range***.

Dans un exemple précédent, nous avons vu que ce code échouera :

```
  {{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  release: {{ .Release.Name }}
  {{- end }}
```

***Release.Name*** ne fait pas partie de la portée restreinte dans le bloc ***with***. Une façon de contourner les problèmes de scope est d'assigner des objets à des variables auxquelles on peut accéder sans tenir compte de la portée actuelle.

Dans les modèles Helm, une variable est une référence nommée à un autre objet. Elle se présente sous la forme ***$name***. Les variables sont assignées à l'aide d'un opérateur d'assignation spécial ***:=*** . Nous pouvons utiliser une variable pour ***Release.Name***.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  {{- $relname := .Release.Name -}}
  {{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  release: {{ $relname }}
  {{- end }}
```

Remarquez qu'avant de commencer le bloc **with**, nous affectons ***$relname := .Release.Name***. Maintenant, à l'intérieur du bloc ***with***, la variable ***$relname*** pointe toujours vers le nom de la version.

L'exécution de ce programme produira ce résultat :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: viable-badger-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
  release: viable-badger
```

Les variables sont particulièrement utiles dans les boucles range. Elles peuvent être utilisées sur des objets de type liste pour capturer à la fois l'index et la valeur :

```
toppings: |-
    {{- range $index, $topping := .Values.pizzaToppings }}
      {{ $index }}: {{ $topping }}
    {{- end }}
```

Notez que ***range*** vient en premier, puis les variables, puis l'opérateur d'affectation, puis la liste. Ceci affectera l'index (à partir de zéro) à ***$index*** et la valeur à ***$topping***. En l'exécutant, on obtient le résultat suivant

```
  toppings: |-
      0: mushrooms
      1: cheese
      2: peppers
      3: onions
```

Pour les structures de données qui ont à la fois une clé et une valeur, nous pouvons utiliser ***range*** pour obtenir les deux. Par exemple, nous pouvons parcourir en boucle le fichier ***.Values.favorite*** comme suit :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  {{- range $key, $val := .Values.favorite }}
  {{ $key }}: {{ $val | quote }}
  {{- end }}
```

Maintenant, à la première itération, ***$key*** sera ***drink*** et ***$val*** sera ***coffee***, et à la seconde, ***$key*** sera ***food*** et ***$val*** sera ***pizza***. L'exécution de produira ceci :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: eager-rabbit-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"
```

Autre exemple : itérer sur un dictionnaire,

```
favoriteFoods:
  breakfast: pancakes
  lunch: salad
  dinner: pizza
```

```
meals:
  {{- range $key, $value := .Values.favoriteFoods }}
    - {{ $key }}: {{ $value | quote }}
  {{- end }}
```

***range $key, $value := .Values.favoriteFoods*** : Cette ligne parcourt chaque entrée du dictionnaire ***favoriteFoods***. À chaque itération, ***$key*** contiendra la clé (par exemple, breakfast), et ***$value*** contiendra la valeur associée à cette clé (par exemple, pancakes).

**{{ $key }}** : Affiche la clé.

**{{ $value }}** : Affiche la valeur.

**{{- end }}** : Marque la fin de la boucle.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: eager-rabbit-configmap
data:
  meals:
    - breakfast: pancakes
    - lunch: salad
    - dinner: pizza
```

Les variables ne sont normalement pas "globales". Elles sont limitées au bloc dans lequel elles sont déclarées. Plus tôt, nous avons attribué ***$relname*** au scope supérieur du modèle. Cette variable s'appliquera à l'ensemble du modèle. Mais dans notre dernier exemple, ***$key*** et ***$val*** ne seront pris en compte qu'à l'intérieur du bloc ***{{ range... }}{{ end }}***.

Cependant, il existe une variable qui est toujours globale **- *$* -** cette variable pointera toujours vers le contexte racine. Cela peut s'avérer très utile lorsque vous effectuez des boucles dans une range et que vous avez besoin de connaître le nom de l'instance du Chart par exemple.

Un exemple pour illustrer cela :

```yaml
{{- range .Values.tlsSecrets }}
apiVersion: v1
kind: Secret
metadata:
  name: {{ .name }}
  labels:
  # De nombreux modèles  utiliseraient `.` en dessous, mais cela ne fonctionnera pas,
  # cependant `$` fonctionnera ici
  app.kubernetes.io/name: {{ template "fullname" $ }}
  # on ne peut pas faire référence à .Chart.Name, mais on peut faire référence à $.Chart.Name.
  helm.sh/chart: "{{ $.Chart.Name }}-{{ $.Chart.Version }}"
  app.kubernetes.io/instance: "{{ $.Release.Name }}"
  # Valeur de appVersion dans Chart.yaml
  app.kubernetes.io/version: "{{ $.Chart.AppVersion }}"
  app.kubernetes.io/managed-by: "{{ $.Release.Service }}"
type: kubernetes.io/tls
data:
  tls.crt: {{ .certificate }}
  tls.key: {{ .key }}
---
```

```
{{- end }}
```

La directive ***range*** itère sur une liste ou un tableau fourni par ***.Values.tlsSecrets.*** Cette valeur est définie dans le fichier values.yaml. Chaque élément de cette liste sera accessible via **.** à l'intérieur de la boucle.

- **Le -** : Supprime les espaces ou nouvelles lignes inutiles avant cette instruction, ce qui est utile pour générer un YAML propre.

**template "fullname" $** :

- Appelle une fonction personnalisée (probablement définie dans _helpers.tpl) pour générer un nom complet basé sur le contexte global $.
- **Pourquoi $** : Le contexte **.** change à l'intérieur de la boucle. $ reste toujours le contexte global, permettant d'accéder à des valeurs globales.

Accès global aux métadonnées de la charte Helm ($.Chart).

- **$.Chart.Name** : Nom de la charte (défini dans Chart.yaml).
- **$.Chart.Version** : Version de la charte.

**$.Release.Name** : Nom de l'instance Helm, spécifié lors de l'installation (helm install <release-name>).

- **$.Chart.AppVersion** : Version de l'application, définie dans Chart.yaml (souvent différente de $.Chart.Version).

**$.Release.Service** : Helm est identifié comme le gestionnaire de cet objet.

**Pourquoi $ est nécessaire parfois ?**

À l'intérieur d'une boucle, le contexte **.** devient l'élément en cours de traitement. Si vous devez accéder à des valeurs globales comme .Pourquoi **$** est nécessaire parfois ?

- À l'intérieur d'une boucle, le contexte **.** devient l'élément en cours de traitement. Si vous devez accéder à des valeurs globales comme ***.Chart.Name*** ou ***.Release.Name***, vous utilisez ***$***.

Jusqu'à présent, nous avons examiné un seul modèle déclaré dans un seul fichier. Mais l'une des caractéristiques les plus puissantes du langage de template Helm est sa capacité à déclarer plusieurs templates et à les utiliser ensemble. Nous y reviendrons dans la section suivante.

# Modèles nommés

Il est temps d'aller au-delà d'un modèle et de commencer à en créer d'autres. Dans cette section, nous verrons comment définir des ***modèles nommés*** dans un fichier.

Un *modèle nommé* (parfois appelé *modèle partiel - partial* ou *sous-modèle - subtemplate* ) est simplement un modèle défini à l'intérieur d'un fichier externe. Nous verrons plusieurs méthodes différentes de les utiliser.

Dans la section **Contrôle de flux**, nous avons présenté trois actions pour déclarer et gérer les modèles : ***define***, ***template*** et ***block***. Dans cette section, nous aborderons ces trois actions, et nous présenterons également une fonction ***include*** spéciale qui fonctionne de la même manière que l'action template.

Un détail important à garder à l'esprit lorsque vous nommez des modèles : **les noms de modèles sont globaux**. Si vous déclarez deux modèles portant le même nom, c'est celui qui est chargé en dernier qui sera utilisé.

Une convention de nommage courante consiste à préfixer chaque modèle défini par le nom du Chart : ***{{ define "mychart.labels" }}***. En utilisant le nom spécifique du Chart comme préfixe, nous pouvons éviter tout conflit pouvant survenir si deux Charts différents implémentent des modèles portant le même nom.

Ce comportement s'applique également aux différentes versions d'un Chart. Si vous avez la version ***1.0.0*** de ***mychart*** qui définit un modèle d'une certaine manière, et une version ***2.0.0*** de ***mychart*** qui modifie le modèle nommé existant, il utilisera celui qui a été chargé en dernier. Vous pouvez contourner ce problème en ajoutant une version dans le nom du Chart ***: {{ define "mychart.v1.labels" }}*** et ***{{ define "mychart.v2.labels" }}.***

## Partials et helpers

Jusqu'à présent, nous avons utilisé un seul fichier, et ce fichier contenait un seul modèle. Mais le langage des modèles de Helm vous permet de créer des modèles intégrés nommés, auxquels on peut accéder par leur nom ailleurs.

Avant d'aborder les aspects pratiques de l'écriture de ces modèles, une convention de nommage des fichiers mérite d'être mentionnée :

- La plupart des fichiers dans templates/ sont traités comme s'ils contenaient des manifestes Kubernetes
- Le fichier NOTES.txt est une exception
- Mais les fichiers dont le nom commence par un trait de soulignement (_) sont supposés *ne pas* contenir de manifeste. Ces fichiers ne sont pas rendus aux

définitions d'objets Kubernetes, mais sont disponibles partout dans d'autres modèles de Charts pour utilisation.

Ces fichiers sont utilisés pour stocker les ***partials*** et les ***helpers***.

**Partials dans Helm**

**Qu'est-ce qu'un partial ?**

Un **partial** dans Helm est essentiellement un template réutilisable, défini dans un fichier séparé, que vous pouvez appeler dans d'autres templates. Cela permet d'éviter la duplication de code et de garder vos templates organisés.

Un **partial** est défini avec la fonction **define** et est utilisé avec **template**.

**Exemple de partial :**

Supposons que vous ayez un template commun pour définir un label de métadonnées dans plusieurs fichiers.

**Fichier labels.yaml (partial) :**

```
{{- define "myapp.labels" }}
labels:
  app: "{{ .Release.name }}"
  version: "{{ .Chart.version }}"
{{- end }}"
```

Ici, vous avez défini un **partial** appelé **myapp.labels** qui génère des labels à partir de la version de votre chart et du nom de votre release.

**Utilisation du partial dans un autre template :**

Ensuite, vous pouvez utiliser ce partial dans un autre fichier template, par exemple un fichier deployment.yaml :

```
ludo@kubernetes:$ vim templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}
  {{- template "myapp.labels" . }} #appel du partials
  labels:
    app: nginx
spec:
 template:
 metadata:
   name: {{ .Release.Name }}
   {{- template "myapp.labels" . }} #appel du partials
```

Dans cet exemple, le partial **myapp.labels** est utilisé à deux endroits dans le template **deployment.yaml** pour générer les labels.

**Pourquoi utiliser des partials ?**

- **Réutilisation de code** : Vous pouvez définir une logique qui se répète dans plusieurs templates sans avoir à la réécrire.
- **Lisibilité** : Vous améliorez la lisibilité en extrayant les parties de code réutilisables dans des fichiers séparés.
- **Maintenance** : Si vous avez besoin de modifier une partie de votre configuration, vous pouvez le faire dans un seul fichier, ce qui simplifie la maintenance.

**Helpers dans Helm**

**Qu'est-ce qu'un helper ?**

Un **helper** dans Helm est similaire à un **partial**, mais il s'agit d'une fonction que vous pouvez appeler pour effectuer des opérations communes, comme formater des chaînes, manipuler des valeurs ou générer des configurations spécifiques.

Les **helpers** sont souvent utilisés pour centraliser la logique de calcul et de transformation des valeurs. Vous pouvez définir des helpers dans un fichier **_helpers.tpl**, qui est un fichier réservé aux fonctions réutilisables.

**Exemple de helper :**

Voici un exemple de helper qui génère un nom personnalisé en fonction de la release et de l'élément du chart.

**Fichier _helpers.tpl :**

```
{{- define "myapp.fullname" }}
{{ .Release.name }}-{{ .Chart.version }}
{{- end }}"
```

Dans cet exemple, le helper **myapp.fullname** génère un nom en combinant le nom de la release et le nom du chart.

**Utilisation du helper dans un autre template :**

Ensuite, vous pouvez utiliser ce helper dans d'autres templates Helm, comme suit :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ template "myapp.fullname" . }} #appel du helper
  labels:
    app: nginx
spec:
 template:
```

```
metadata:
  name: {{ template "myapp.fullname" . }} #appel du helper
```

Dans cet exemple, le helper **myapp.fullname** est appelé pour générer le nom du déploiement en utilisant le nom de la release et du chart.

**Pourquoi utiliser des helpers ?**

- **Centralisation de la logique** : Vous pouvez centraliser des calculs ou des transformations dans des helpers plutôt que de les répéter dans chaque fichier template.
- **Lisibilité** : Un helper permet de rendre vos templates plus lisibles et moins encombrés.
- **Facilité de maintenance** : En modifiant un helper, vous appliquez immédiatement les changements dans tous les templates qui l'utilisent.

---

**Différences entre Partial et Helper**

| Critère | Partial | Helper |
|---|---|---|
| **Définition** | Un bloc de template réutilisable dans plusieurs fichiers | Une fonction réutilisable qui effectue une opération |
| **Utilisation** | Utilisé pour insérer des morceaux de code dans des templates | Utilisé pour effectuer des calculs ou manipulations |
| **Syntaxe** | **define** pour définir, **template** pour appeler | **define** pour définir, **template** pour appeler |
| **Exemples** | Génération de labels, configurations répétées | Formatage de noms, transformations de données |

# Déclarer et utiliser des modèles avec define et template

L'action **define** nous permet de créer un modèle nommé à l'intérieur d'un fichier de modèle. Sa syntaxe est la suivante

```
{{- define "MY.NAME" }}
  # body of template here
{{- end }}
```

Par exemple, nous pouvons définir un modèle pour encapsuler un bloc d'étiquettes Kubernetes :

```
{{- define "mychart.labels" }}
```

```
  labels:
    generator: helm
    date: {{ now | htmlDate }}
{{- end }}
```

Nous pouvons maintenant intégrer ce modèle dans notre ConfigMap existant, puis l'inclure dans l'action :

```
{{- define "mychart.labels" }}
  labels:
    generator: helm
    date: {{ now | htmlDate }}
{{- end }}
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  {{- template "mychart.labels" }}
data:
  myvalue: "Hello World"
  {{- range $key, $val := .Values.favorite }}
  {{ $key }}: {{ $val | quote }}
  {{- end }}
```

Lorsque le moteur lit ce modèle, il stocke la référence à ***mychart.labels*** jusqu'à ce que le modèle "***mychart.labels***" soit appelé.

Le résultat ressemblera donc à ceci :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: running-panda-configmap
  labels:
    generator: helm
    date: 2016-11-02
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"
```

Remarque : un ***define*** ne produit pas de sortie à moins qu'il ne soit appelé avec un modèle, comme dans cet exemple.

Par convention, les fonctions ***define*** doivent comporter un bloc de documentation simple ({{/* ... */}}) décrivant ce qu'elles font.

```
{{/* Generate basic labels */}}
{{- define "mychart.labels" }}
  labels:
    generator: helm
    date: {{ now | htmlDate }}
{{- end }}
```

Même si cette définition se trouve dans _helpers.tpl, elle est toujours accessible dans configmap.yaml :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  {{- template "mychart.labels" }}
data:
  myvalue: "Hello World"
  {{- range $key, $val := .Values.favorite }}
  {{ $key }}: {{ $val | quote }}
  {{- end }}
```

Comme indiqué ci-dessus, les **noms de modèles sont globaux**. Par conséquent, si deux modèles sont déclarés avec le même nom, c'est le dernier qui sera utilisé. Étant donné que les modèles des sous-Charts sont compilés avec les modèles de niveau supérieur, il est préférable de donner à vos modèles des *noms spécifiques aux Charts*. Une convention de nommage populaire consiste à préfixer chaque modèle défini par le nom du Chart : **{{ define "mychart.labels" }}**.

## Définir la portée d'un modèle

Dans le modèle que nous avons défini ci-dessus, nous n'avons pas utilisé d'objets. Nous avons simplement utilisé des fonctions. Modifions le modèle que nous avons défini pour y inclure le nom et la version du Chart :

```
{{/* Generate basic labels */}}
{{- define "mychart.labels" }}
  labels:
    generator: helm
    date: {{ now | htmlDate }}
    chart: {{ .Chart.Name }}
    version: {{ .Chart.Version }}
{{- end }}
```

Si nous rendons cela, nous obtiendrons une erreur comme celle-ci :

```
$ helm install --dry-run moldy-jaguar ./mychart
Erreur : Impossible de construire des objets kubernetes à partir du manifeste de la version : erreur de validation "" : erreur de validation des data : [unknown object kind "nil" in ConfigMap.metadata.labels.chart, unknown object kind "nil" in ConfigMap.metadata.labels.version]
```

Pour voir ce qui s'est passé, réexécutez avec **--disable-openapi-validation** : **helm install --dry-run --disable-openapi-validation moldy-jaguar ./mychart**. Le résultat ne sera pas celui que nous attendons :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: moldy-jaguar-configmap
  labels:
    generator: helm
    date: 2021-03-06
    chart:
    version:
```

Qu'est-il advenu du nom et de la version ? Ils n'étaient pas dans la portée de notre modèle défini. Lorsqu'un modèle nommé (créé avec ***define***) est rendu, il reçoit la portée transmise par l'appel au modèle. Dans notre exemple, nous avons inclus le modèle comme suit :

```
{{- template "mychart.labels" }}
```

Aucune portée n'a été passée, donc dans le modèle nous ne pouvons pas accéder à quoi que ce soit dans ... C'est assez facile à corriger, cependant. Il suffit de passer une portée au modèle :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  {{- template "mychart.labels" . }}
```

Notez que nous passons un . à la fin de l'appel au modèle. Nous pourrions tout aussi bien passer ***.Values*** ou ***.Values.favorite*** ou n'importe quel scope. Mais ce que nous voulons, c'est la portée de premier niveau.

Maintenant, lorsque nous exécutons ce modèle ***avec helm install --dry-run --debug plinking-anaco ./mychart***, nous obtenons ceci :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: plinking-anaco-configmap
  labels:
    generator: helm
    date: 2021-03-06
    chart: mychart
    version: 0.1.0
```

Maintenant ***{{ .Chart.Name }}*** se résout en ***mychart***, et ***{{ .Chart.Version }}*** se résout en ***0.1.0.***

# La fonction include

Supposons que nous ayons défini un modèle simple qui ressemble à ceci :

```
{{- define "mychart.app" -}}
app_name: {{ .Chart.Name }}
app_version: "{{ .Chart.Version }}"
{{- end -}}
```

Supposons maintenant que l'on veuille insérer ceci à la fois dans la section labels : du modèle, et dans la section ***data***: :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  labels:
```

```
    {{ template "mychart.app" . }}
data:
  myvalue: "Hello World"
  {{- range $key, $val := .Values.favorite }}
  {{ $key }}: {{ $val | quote }}
  {{- end }}
{{ template "mychart.app" . }}
```

Si nous rendons cela, nous obtiendrons une erreur comme celle-ci :

```
$ helm install --dry-run measly-whippet ./mychart
Erreur : Impossible de construire des objets kubernetes à partir de la version manifeste : erreur de validation de "" : erreur de validation des data : [ValidationError(ConfigMap) : unknown field "app_name" in io.k8s.api.core.v1.ConfigMap, ValidationError(ConfigMap) : unknown field "app_version" in io.k8s.api.core.v1.ConfigMap]
```

Pour voir ce qui s'est passé, réexécutez avec ***--disable-openapi-validation : helm install --dry-run --disable-openapi-validation measly-whippet ./mychart***. Le résultat ne sera pas celui que nous attendons :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: measly-whippet-configmap
  labels:
    app_name: mychart
app_version: "0.1.0"
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"
app_name: mychart
app_version: "0.1.0"
```

Notez que l'indentation de app_version est erronée aux deux endroits. Pourquoi ? Parce que le texte du modèle qui est substitué est aligné à gauche. Le modèle étant une action et non une fonction, il n'existe aucun moyen de transmettre la sortie d'un appel de modèle à d'autres fonctions ; les data sont simplement insérées en ligne.

Pour contourner ce problème, Helm fournit une alternative au modèle qui importera le contenu d'un modèle dans le pipeline actuel où il pourra être transmis à d'autres fonctions du pipeline.

Voici l'exemple ci-dessus, corrigé pour utiliser indent afin d'indenter correctement le modèle ***mychart***.app :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  labels:
{{ include "mychart.app" . | indent 4 }}
data:
  myvalue: "Hello World"
  {{- range $key, $val := .Values.favorite }}
  {{ $key }}: {{ $val | quote }}
  {{- end }}
```

```
{{ include "mychart.app" . | indent 2 }}
```

Le YAML produit est maintenant correctement indenté pour chaque section :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: edgy-mole-configmap
  labels:
    app_name: mychart
    app_version: "0.1.0"
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"
  app_name: mychart
  app_version: "0.1.0"
```

Il est préférable d'utiliser include plutôt que template dans les modèles Helm, simplement parce que le formatage de sortie peut être mieux géré pour les documents YAML.

Il arrive que l'on veuille importer du contenu, mais pas sous forme de modèles. En d'autres termes, nous voulons importer des fichiers mot pour mot. Nous pouvons y parvenir en accédant aux fichiers par l'intermédiaire de l'objet .Files décrit dans la section suivante.

# .Files : Les fichiers dans les modèles

Dans la section précédente, nous avons examiné plusieurs façons de créer des modèles nommés et d'y accéder. Cela permet d'importer facilement un modèle à partir d'un autre modèle. Mais il est parfois souhaitable d'importer un fichier qui n'est pas un modèle *et d'injecter son contenu sans le faire passer par le moteur de rendu du modèle.*

Helm permet d'accéder aux fichiers par l'intermédiaire de l'objet ***.Files***. Avant d'aborder les exemples de modèles, il convient de noter quelques points sur le fonctionnement de cet objet :

Vous pouvez ajouter des fichiers à votre chart Helm.
Attention cependant.

- Les Charts doivent être inférieurs à 1Mo en raison des limites de stockage des objets Kubernetes.
- Certains fichiers ne sont pas accessibles via l'objet ***.Files***, généralement pour des raisons de sécurité.
- Il n'est pas possible d'accéder aux fichiers dans templates/.
- Les fichiers exclus à l'aide de ***.helmignore*** ne sont pas accessibles.
- Il n'est pas possible d'accéder aux fichiers situés en dehors d'un sous-groupe de l'application Helm, y compris ceux du parent.
- Les charts ne préservent pas les informations relatives au mode UNIX, de sorte que les autorisations au niveau du fichier n'ont aucun impact sur la disponibilité d'un fichier lorsqu'il s'agit de l'objet ***.Files***.

## .files.Get

Ces mises en garde étant faites, écrivons un modèle qui affiche le contenu de trois fichiers dans notre ConfigMap. Pour commencer, nous allons ajouter trois fichiers au Chart, en les plaçant tous les trois directement dans le répertoire ***mychart/***.

config1.toml :

```
message = Hello from config 1
```

config2.toml :

```
message = Ceci est la configuration 2
```

config3.toml :

```
message = Au revoir de la config 3
```

Chacun de ces fichiers est un simple fichier TOML (pensez aux anciens fichiers INI de Windows). Nous connaissons les noms de ces fichiers, nous pouvons donc utiliser une fonction ***range*** pour les parcourir en boucle et injecter leur contenu dans notre ConfigMap.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  {{- $files := .Files }}
  {{- range tuple "config1.toml" "config2.toml" "config3.toml" }}
  {{ . }}: |-
        {{ $files.Get . }}
  {{- end }}
```

Ce ConfigMap utilise plusieurs des techniques abordées dans les sections précédentes. Par exemple, nous créons une variable ***$files*** pour contenir une référence à l'objet ***.Files***. Nous utilisons également la fonction ***tuple*** pour créer une liste de fichiers que nous parcourons en boucle. Nous printons ensuite le nom de chaque fichier ***{{ . }} :*** **|-** suivi du contenu du fichier ***{{ $files.Get . }}.***

**{{- $files := .Files }}**

- **Déclaration de variable** : On crée une variable locale $files qui pointe vers .Files. Cela permet d'accéder plus facilement aux fichiers sans répéter *.Files* à chaque utilisation.

**{{- range tuple "config1.toml" "config2.toml" "config3.toml" }}**

- **Boucle range** : On itère sur une liste de fichiers (tuple) contenant "config1.toml", "config2.toml", et "config3.toml".
- **tuple** : Crée une liste dans Helm.

**Corps de la boucle :**

**{{ . }}: |-**

**{{ $files.Get . }}**

- **{{ . }}** : Représente l'élément actuel de l'itération (par exemple, "config1.toml" lors de la première itération).
- **Syntaxe YAML** : La clé est le nom du fichier (config1.toml, config2.toml, etc.).
- **|-** : Indique un bloc YAML "littéral" pour inclure du contenu multi-lignes.
- **{{ $files.Get . }}** : Récupère et insère le contenu du fichier actuel.

**Fermeture de la boucle :**

La boucle se termine et le contenu des trois fichiers est ajouté successivement au rendu.

L'exécution de ce modèle produira un ConfigMap unique avec le contenu des trois fichiers

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: quieting-giraf-configmap
data:
  config1.toml: |-
        message = Hello from config 1

  config2.toml: |-
        message = This is config 2

  config3.toml: |-
        message = Goodbye from config 3
```

### Cas d'utilisation typique

Cette approche est utile pour inclure plusieurs fichiers de configuration dans un seul objet Kubernetes, par exemple :

- Une **ConfigMap** contenant des fichiers de configuration pour une application.
- Un **Secret** si les fichiers incluent des données sensibles.

## Files.Glob

***Files.Glob*** est une fonctionnalité très utile dans Helm lorsqu'on travaille avec des fichiers inclus dans un chart.
C'est particulièrement utile lorsque vous avez plusieurs fichiers et que vous souhaitez en traiter seulement certains en fonction de leur nom ou de leur emplacement.

### Syntaxe de base

```
{{- $matchedFiles := .Files.Glob "pattern" }}
```

**pattern** : Un motif utilisé pour rechercher les fichiers. Par exemple :

- "*.txt" : Tous les fichiers avec l'extension .txt.
- "config/*.yaml" : Tous les fichiers YAML dans le sous-répertoire files/config/.
- "**/*.json" : Tous les fichiers JSON, quel que soit le sous-répertoire.

Au fur et à mesure que votre chart se développe, il se peut que vous ayez besoin d'organiser davantage vos fichiers.

***.Glob*** renvoie un type Fichiers, vous pouvez donc appeler n'importe quelle méthode ***Files.Glob*** Fichiers sur l'objet renvoyé.

Imaginez, par exemple, la structure d'un répertoire :

```
foo/:
  foo.txt foo.yaml

bar/:
  bar.go bar.conf baz.yaml
```

Vous disposez de plusieurs options avec les Globs :

```
{{ $currentScope := .}}
{{ range $path, $_ :=  .Files.Glob  "**.yaml" }}
    {{- with $currentScope}}
        {{ .Files.Get $path }}
    {{- end }}
{{ end }}
```

**{{ $currentScope := . }}**

- Cette ligne assigne l'objet actuel (.) à une variable appelée $currentScope.
- Cela permet de conserver l'état du contexte actuel de manière à pouvoir y faire référence plus tard dans le bloc range, même si le contexte change au sein du bloc.

**{{ range $path, $_ := .Files.Glob "**.yaml" }}**

- Cette ligne lance un **boucle** range sur les fichiers qui correspondent au motif **.yaml dans le répertoire files/ du chart Helm.
- ***.Files.Glob "**.yaml"*** récupère tous les fichiers YAML dans le répertoire files/ et ses sous-répertoires. Cela renverra une liste de fichiers correspondants, et range les itère un par un.

  $path : représente le chemin relatif du fichier YAML actuel dans files/.

  $_ : c'est une variable inutilisée (elle est souvent utilisée pour ignorer une valeur), mais ici, elle est utilisée pour la syntaxe du range car Glob retourne une map avec la clé étant le chemin et la valeur étant l'objet fichier (mais cette valeur n'est pas utilisée ici).

**{{- with $currentScope }}**

- **with** est l'instruction de contrôle qui change le contexte dans un bloc de code. Dans ce cas, elle permet de revenir au contexte initial ($currentScope), c'est-à-dire l'objet d'origine avant que la boucle range n'ait modifié le contexte.
- Le tiret {{- avant ***with*** supprime les espaces inutiles qui précèdent cette ligne dans le rendu.

**{{ .Files.Get $path }}**

- **.Files.Get $path** permet de récupérer le contenu du fichier spécifié par $path dans le répertoire files/.

- Ici, $path est le chemin relatif du fichier YAML que l'on souhaite récupérer dans le répertoire files/. Cela permet de charger le contenu de chaque fichier YAML pendant l'itération de la boucle range.

**5. {{- end }}**

- Ceci marque la fin du bloc with, et donc du retour au contexte d'origine ($currentScope).

**6. {{ end }}**

- Cela marque la fin de la boucle range.

Le code parcourt tous les fichiers YAML dans le répertoire files/ du chart Helm et ses sous-répertoires grâce à .Files.Glob "**.yaml".

Pour chaque fichier YAML, le contenu du fichier est récupéré via .Files.Get $path, où $path est le chemin du fichier.

Le contexte est maintenu intact grâce à l'utilisation de la variable $currentScope, qui permet d'éviter que le contexte de la boucle range ne perturbe les autres parties du template.

Ce code permet de récupérer le contenu de chaque fichier YAML dans files/ tout en préservant le contexte du template.

Ou

```
{{ range $path, $_ :=  .Files.Glob  "**.yaml" }}
      {{ $.Files.Get $path }}
{{ end }}
```

**$ (dans $.Files.Get $path)** :

- L'utilisation de $.Files.Get $path permet de revenir au contexte global, c'est-à-dire au contexte **d'origine**, qui est celui qui a été défini en dehors de la boucle range.
- $.Files.Get indique que vous voulez accéder à Files.Get à partir du **contexte global**, et pas du contexte local de la boucle range.

**Contexte local (dans l'exemple précédent avec with $currentScope)** :

- Dans l'exemple précédent, vous utilisez {{- with $currentScope }} pour garantir que vous revenez au contexte d'origine (celui de la première ligne avec {{ $currentScope := . }}), mais vous auriez aussi pu utiliser {{ .Files.Get $path }} directement si vous vouliez utiliser le contexte local de la boucle range.

**Code plus simple** : Utilise $.Files.Get pour s'assurer que vous accédez à Files.Get dans le contexte global.
**Code original avec with $currentScope** : Gère explicitement le contexte et assure que vous travaillez avec un contexte particulier (en l'occurrence, le contexte global avant la boucle).

Les deux approches donneront le même résultat à la fin, c'est juste une question de gestion explicite du contexte. Si vous êtes certain de vouloir utiliser le contexte global à l'intérieur de la boucle range, le code plus simple avec $.Files.Get est suffisant.

## Fonctions utiles ConfigMap et Secrets

Il est très courant de vouloir placer le contenu des fichiers dans les ConfigMaps et les Secrets, pour les monter dans vos pods au moment de l'exécution. Pour cela, Helm fournit quelques méthodes utiles sur le type Files.

Pour une meilleure organisation, il est particulièrement utile d'utiliser ces méthodes en conjonction avec la méthode Glob.

Compte tenu de la structure des répertoires de la base de data Glob ci-dessus :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: conf
data:
{{ (.Files.Glob "foo/*").AsConfig | indent 2 }}
---
apiVersion: v1
kind: Secret
metadata:
  name: very-secret
type: Opaque
data:
{{ (.Files.Glob "bar/*").AsSecrets | indent 2 }}
```

**Code 1 : {{ (.Files.Glob "foo/*").AsConfig | indent 2 }}**

**(.Files.Glob "foo/*")** :

- .Files.Glob est utilisé pour récupérer les fichiers dans le chart Helm.
- Le motif "foo/*" indique que vous voulez récupérer tous les fichiers présents dans le répertoire foo/ du chart Helm. Ce motif permet de capturer tous les fichiers (de n'importe quelle extension) dans ce sous-répertoire.

**.AsConfig** :

Cette fonction est utilisée pour transformer les fichiers récupérés avec .Files.Glob en une **configmap** Kubernetes.

- Une **configmap** est une ressource Kubernetes qui permet de stocker des données de configuration sous forme clé-valeur. Dans ce cas, chaque fichier dans le répertoire foo/ sera transformé en une entrée dans une configmap, où chaque fichier est stocké sous forme de clé, et son contenu sous forme de valeur.
- Par exemple, si foo/file1.yaml existe, son contenu sera utilisé comme valeur de la clé file1.yaml dans la configmap.

**| indent 2** :

- indent 2 est un **filtre** qui permet d'ajouter une indentation de 2 espaces à tout le texte généré par le bloc précédent.
- Cela est souvent utilisé pour formater le résultat dans un fichier YAML afin de le rendre plus lisible et respecter la structure d'indentation de Kubernetes.
    - o

**{{ (.Files.Glob "bar/*").AsSecrets | indent 2 }}**

Ce code suit la même logique que le premier, mais avec une différence clé :
**(.Files.Glob "bar/*")** :

- Cette partie récupère tous les fichiers dans le répertoire bar/ du chart Helm, de la même manière que pour foo/.

**.AsSecrets** :

- Au lieu de transformer les fichiers en une **configmap**, cette fonction les transforme en **secrets** Kubernetes.
- Un **secret** est une ressource Kubernetes utilisée pour stocker des informations sensibles, telles que des mots de passe, des clés API ou des certificats. Chaque fichier sera traité comme une valeur dans un secret, où le nom du fichier sera la clé, et son contenu sera la valeur.
- Par exemple, si bar/secret1.txt existe, son contenu sera utilisé comme valeur de la clé secret1.txt dans le secret.

**| indent 2** :

- Comme dans le premier code, cela ajoute une indentation de 2 espaces pour formater correctement le fichier YAML généré.

**Résumé des différences entre les deux codes :**

- **.AsConfig** transforme les fichiers en une **configmap** Kubernetes.
- **.AsSecrets** transforme les fichiers en un **secret** Kubernetes.
- Les deux utilisent le filtre indent 2 pour bien formater la sortie dans le YAML.

Si vous avez un fichier dans foo/ appelé file1.yaml et un fichier dans bar/ appelé secret1.txt, voici ce que cela pourrait produire :
Cette méthode Helm est utilisé pour récupérer des fichiers depuis des répertoires spécifiques et les formater soit en configmaps (pour des données de configuration), soit en secrets (pour des informations sensibles), tout en appliquant une indentation pour garantir un bon formatage YAML.

## Encodage

Vous pouvez importer un fichier et demander au modèle de l'encoder en base-64 pour garantir une transmission réussie :

```
apiVersion: v1
kind: Secret
metadata:
  name: {{ .Release.Name }}-secret
type: Opaque
data:
  token: |-
        {{ .Files.Get "config1.toml" | b64enc }}
```

Prend le même fichier config1.toml que nous avons utilisé précédemment et l'encode :

```
# Source: mychart/templates/secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: lucky-turkey-secret
type: Opaque
```

```
data:
  token: |-
        bWVzc2FnZSA9IEhlbGxvIGZyb20gY29uZmlnIDEK
```

## Lines

Il est parfois souhaitable d'accéder à chaque ligne d'un fichier dans votre modèle. Vous pouvez parcourir les lignes en boucle à l'aide d'une fonction d'intervalle :

```
data:
  some-file.txt: {{ range .Files.Lines "foo/bar.txt" }}
    {{ . }}{{ end }}
```

Il n'existe aucun moyen de transmettre des fichiers externes au chart lors de l'installation de helm. Par conséquent, si vous demandez aux utilisateurs de fournir des data, celles-ci doivent être chargées à l'aide de ***helm install -f*** ou ***helm install --set***.

Cette partie conclut notre plongée dans les outils et les techniques d'écriture des modèles Helm. Dans la prochaine section, nous verrons comment vous pouvez utiliser un fichier spécial, templates/NOTES.txt, pour envoyer des instructions post-installation aux utilisateurs de votre Chart.

# Création d'un fichier NOTES.txt

Dans cette section, nous allons fournir des instructions aux utilisateurs de vos charts. A la fin d'une installation ou d'une mise à jour de Helm, Helm peut imprimer un bloc d'informations utiles pour les utilisateurs. Ces informations sont hautement personnalisables à l'aide de modèles.

Pour ajouter des notes d'installation à votre Chart, il suffit de créer un fichier templates/NOTES.txt. Ce fichier est du texte brut, mais il est traité comme un modèle et dispose donc de toutes les fonctions et de tous les objets des modèles.

Créons un simple fichier NOTES.txt :

```
Thank you for installing {{ .Chart.Name }}.

Your release is named {{ .Release.Name }}.

To learn more about the release, try:

  $ helm status {{ .Release.Name }}
  $ helm get all {{ .Release.Name }}
```

Maintenant, si nous éxecutons ***helm install rude-cardinal ./mychart***, nous verrons ce message :

```
RESOURCES:
==> v1/Secret
NAME                   TYPE      DATA      AGE
rude-cardinal-secret   Opaque    1         0s

==> v1/ConfigMap
NAME                       DATA      AGE
rude-cardinal-configmap    3         0s


NOTES:
Thank you for installing mychart.

Your release is named rude-cardinal.

To learn more about the release, try:

  $ helm status rude-cardinal
  $ helm get all rude-cardinal
```

L'utilisation de NOTES.txt de cette manière est un excellent moyen de donner à vos utilisateurs des informations détaillées sur la manière d'utiliser le Chart nouvellement installé. La création d'un fichier NOTES.txt est fortement recommandée, bien qu'elle ne soit pas obligatoire.

# TP – Réalisation de chart helm

Nous avons pu découvrir de multiples fonctionnalités afin de construire ses charts. Nous allons partir de manifest Yamls existant pour construire une application de paye en php.

Nous allons construire deux charts à partir de manifest yamls exitant et fonctionnel.

Dans votre répertoire $HOME/app-php. Vous trouverez les manifest destinés a la création de Charts Helm.

Nous devons transformer ces derniers en Chart helm.  Deux Charts. Un pour le frontend, un autre pour la base de données.

Le frontend sera accessible via un ingress à l'URL : quelquechose.ludovic.io

Vous devez créer un chart pour le frontend web-php Les fichiers deployment, pvc, service, ingress, sont à transformer en Template helm.

Vous devez créer un chart pour une base de données, accessibles par son DNS mysql, Les fichiers deployment, pvc, service, ingress, sont à transformer en Template helm.

Pensez au range, ***with***, Modelés nommées ***_helpers.tpl*** , Au ***objet built-in***,

Faire un tour sur la documentation, les fonctions.

Attention à l'indentation.

# Subcharts et valeurs globales

Jusqu'à présent, nous n'avons travaillé qu'avec un seul Chart. Mais les Charts peuvent avoir des dépendances, appelées *sous-Charts*, qui ont également leurs propres valeurs et modèles. Dans cette section, nous allons créer un Subchart et voir les différentes façons d'accéder aux valeurs à partir des modèles.

Avant de nous plonger dans le code, il y a quelques détails importants à connaître sur les Subcharts d'application.

1. Un Chart secondaire est considéré comme "autonome", ce qui signifie qu'il ne peut jamais dépendre explicitement de son Chart parent.
2. C'est pourquoi un sous-Chart ne peut pas accéder aux valeurs de son parent.
3. Un Chart parent peut remplacer les valeurs des Charts secondaires.
4. Helm dispose d'un concept de *valeurs globales accessibles à* toutes les charts.

   Ces limitations ne s'appliquent pas nécessairement aux charts de bibliothèque, qui sont conçus pour fournir une fonctionnalité d'aide standardisée.

Au fil des exemples présentés dans cette section, bon nombre de ces concepts deviendront plus clairs.

## Création d'un Subchart

Pour ces TPs, nous partirons du Chart mychart/ que nous avons créé au début, et nous ajouterons un nouveau Chart à l'intérieur de celui-ci.

```
$ cd mychart/charts
$ helm create mysubchart
Creating mysubchart
$ rm -rf mysubchart/templates/*
```

Remarquez que, comme précédemment, nous avons supprimé tous les modèles de base afin de pouvoir repartir de zéro. Dans ce guide, nous nous concentrons sur le fonctionnement des modèles et non sur la gestion des dépendances.

## Ajout de valeurs et d'un template au Subchart

Ensuite, créons un modèle simple et un fichier de valeurs pour notre Chart mysubchart. Il devrait déjà y avoir un values.yaml dans mychart/charts/mysubchart. Nous allons le configurer comme suit :

```
dessert: cake
```

Ensuite, nous allons créer un nouveau modèle ConfigMap dans ***mychart/charts/mysubchart/templates/configmap.yaml*** :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-cfgmap2
data:
  dessert: {{ .Values.dessert }}
```

Comme chaque sous-Chart est un *Chart autonome*, nous pouvons tester mysubchart seul :

```
$ helm install --generate-name --dry-run --debug mychart/charts/mysubchart
SERVER : "localhost:44134"
CHART PATH : /Users/mattbutcher/Code/Go/src/helm.sh/helm/_scratch/mychart/charts/mysubchart
NOM : newbie-elk
TARGET NAMESPACE : default
Chart : mysubchart 0.1.0
MANIFESTER :
---
# Source : mysubchart/templates/configmap.yaml
apiVersion : v1
kind : ConfigMap
metadata :
 nom : newbie-elk-cfgmap2
data :
 dessert : gâteau
```

## Remplacer les valeurs d'un Chart parent

Notre Chart original, mychart, est maintenant le Chart *parent de* mysubchart. Cette relation est entièrement basée sur le fait que mysubchart se trouve dans mychart/charts.

Parce que mychart est un parent, nous pouvons spécifier la configuration dans mychart et faire en sorte que cette configuration soit poussée dans mysubchart. Par exemple, nous pouvons modifier ***mychart/values.yaml*** comme suit :

```
favorite:
  drink: coffee
  food: pizza
pizzaToppings:
  - mushrooms
  - cheese
  - peppers
  - onions

mysubchart:
  dessert: ice cream
```

Notez les deux dernières lignes. Toutes les directives à l'intérieur de la section mysubchart seront envoyées au Chart mysubchart. Ainsi, si nous lançons helm install --dry-run --debug mychart, l'une des choses que nous verrons est le ConfigMap de mysubchart :

```
# Source: mychart/charts/mysubchart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: unhinged-bee-cfgmap2
data:
  dessert: ice cream
```

La valeur du niveau supérieur a maintenant remplacé la valeur du Subchart.

Il y a un détail important à noter ici. Nous n'avons pas modifié le modèle de mychart/charts/mysubchart/templates/configmap.yaml pour qu'il pointe sur .Values.mysubchart.dessert. Du point de vue de ce modèle, la valeur est toujours située dans .Values.dessert. Au fur et à mesure que le moteur de template transmet des valeurs, il définit la portée. Ainsi, pour les modèles mysubchart, seules les valeurs spécifiques à mysubchart seront disponibles dans .Values.

Parfois, cependant, vous souhaitez que certaines valeurs soient disponibles pour tous les modèles. Pour ce faire, vous pouvez utiliser les valeurs globales du Chart.

## Valeurs globales du Chart

Les valeurs globales sont des valeurs accessibles à partir de n'importe quel Chart ou sous Chart portant exactement le même nom. Les valeurs globales doivent être déclarées explicitement. Vous ne pouvez pas utiliser une valeur non globale existante comme s'il s'agissait d'une valeur globale.

Le type de data Values possède une section réservée appelée Values.global dans laquelle des valeurs globales peuvent être définies. Définissons-en une dans notre fichier mychart/values.yaml.

```
favorite:
  drink: coffee
  food: pizza
pizzaToppings:
  - mushrooms
  - cheese
  - peppers
  - onions

mysubchart:
  dessert: ice cream

global:
  salad: caesar
```

En raison du fonctionnement des globaux, ***mychart/templates/configmap.yaml*** et mysubchart/templates/configmap.yaml devraient pouvoir accéder à cette valeur sous la forme ***{{ .Values.global.salad }}.***

mychart/templates/configmap.yaml :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  salad: {{ .Values.global.salad }}
```

mysubchart/templates/configmap.yaml :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-cfgmap2
data:
  dessert: {{ .Values.dessert }}
  salad: {{ .Values.global.salad }}
```

Maintenant, si nous effectuons une installation à blanc, nous verrons la même valeur dans les deux sorties :

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: silly-snake-configmap
data:
  salad: caesar

---
# Source: mychart/charts/mysubchart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: silly-snake-cfgmap2
data:
  dessert: ice cream
  salad: caesar
```

Les globaux sont utiles pour transmettre des informations de ce type, bien qu'il faille s'assurer que les bons modèles sont configurés pour utiliser les globaux.

# Partage de modèles avec des Subcharts

Les Charts parents et les Charts secondaires peuvent partager des modèles. Tout bloc défini dans un Chart est disponible pour les autres Charts.

Par exemple, nous pouvons définir un modèle simple comme suit :

```
{{- define "labels" }}from: mychart{{ end }}
```

Rappelez-vous que les étiquettes des modèles sont *partagées globalement*. Ainsi, le Chart des étiquettes peut être inclus dans n'importe quel autre Chart.

Bien que les développeurs de Charts aient le choix entre include et template, l'un des avantages de l'utilisation de include est qu'il peut référencer dynamiquement des templates :

```
{{ include $mytemplate }}
```

déréférencera $mytemplate. La fonction template, en revanche, n'accepte qu'une chaîne de caractères.

# Éviter l'utilisation de blocs

Le langage de template Go fournit un mot-clé block qui permet aux développeurs de fournir une implémentation par défaut qui sera remplacée ultérieurement. Dans les charts Helm, les blocs ne

sont pas le meilleur outil pour la surcharge, car si plusieurs implémentations du même bloc sont fournies, celle qui est sélectionnée est imprévisible.

Il est suggéré d'utiliser plutôt include.

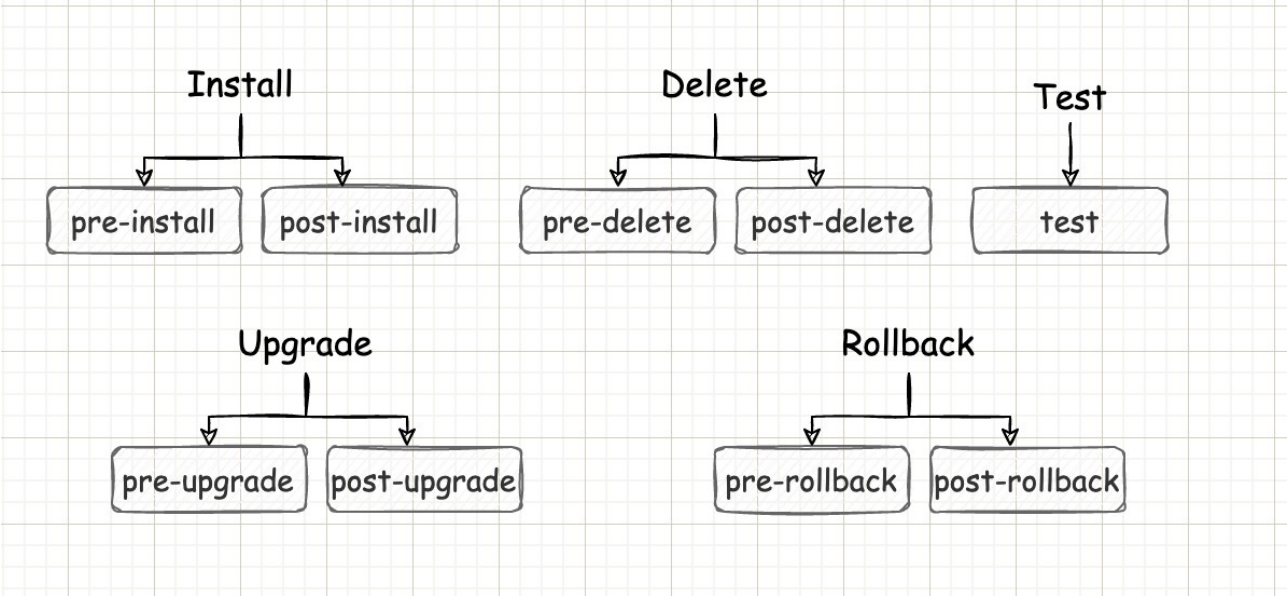
# Chart Hooks

Helm propose un mécanisme de hook permettant aux développeurs de charts d'intervenir à certains moments du cycle de vie d'une version.

Par exemple, vous pouvez utiliser des hooks pour :

- Charger un ConfigMap ou un Secret pendant l'installation avant que d'autres charts ne soient chargés.
- Exécuter une tâche pour sauvegarder une base de données avant d'installer un nouveau chart , puis exécuter une deuxième tâche après la mise à niveau afin de restaurer les données.
- Exécuter une tâche avant la suppression d'une version pour retirer  un service de la rotation avant de le supprimer.

Les hooks fonctionnent comme des modèles ordinaires, mais ils ont des annotations spéciales qui font que Helm les utilise différemment. Dans cette section, nous couvrons le modèle d'utilisation de base des hooks.




## Les hooks disponibles

Les hooks suivants sont définis :

- **pre-install** S'exécute après le traitement des Templates, mais avant que les ressources ne soient créées dans Kubernetes.
- **post-install** Exécute après que toutes les ressources ont été chargées dans Kubernetes.
- **pre-delete** S'exécute lors d'une demande de suppression avant que toute ressource ne soit supprimée de Kubernetes.
- **post-delete** Exécute une demande de suppression après que toutes les ressources de la version ont été supprimées.
- **pre-upgrade** Exécute une demande de mise à niveau après le traitement des Templates, mais avant la mise à jour des ressources.
- **post-upgrade** Exécute une demande de mise à niveau après que toutes les ressources ont été mises à jour.

- **pre-rollback** Exécute une demande de rollback après le traitement des Templates, mais avant que toutes les ressources ne soient rollbackées.
- **post-rollback** Exécute une demande de rollback après que toutes les ressources ont été modifiées.

## Les hooks et le cycle de vie de la version

Les hooks vous permettent, en tant que développeur de charts, d'effectuer des opérations à des moments stratégiques du cycle de vie d'une version. Par exemple, considérons le cycle de vie d'une installation de Helm.

Par défaut, le cycle de vie ressemble à ceci :

- L'utilisateur exécute `helm install foo`
- L'API d'installation de la bibliothèque Helm est appelée
- Après quelques vérifications, la bibliothèque interprète les templates foo
- La bibliothèque charge les ressources résultantes dans Kubernetes.
- La bibliothèque renvoie l'objet release (et d'autres données) au client.
- Le client quitte l'application.

Si le développeur de la chart  foo implémente deux hooks, le cycle de vie est modifié comme suit :

- L'utilisateur exécute `helm install foo`
- L'API d'installation de la bibliothèque Helm est appelée
- Après quelques vérifications, la bibliothèque interprète les templates foo.
- La bibliothèque se prépare à exécuter les hooks de pre-installation (chargement des ressources des hooks dans Kubernetes).
- La bibliothèque trie les hooks par poids (en attribuant un poids de 0 par défaut), par type de ressource et enfin par nom dans l'ordre croissant.
- La bibliothèque charge ensuite le hook ayant le poids le plus faible en premier (du négatif au positif).
- La bibliothèque attend que le hook soit "prêt".
- La bibliothèque charge les ressources résultantes dans Kubernetes.
- La bibliothèque exécute le hook post-installation (chargement des ressources du hook)
- La bibliothèque attend que le hook soit "prêt".
- La bibliothèque renvoie l'objet de validation (et d'autres données) au client.
- Le client quitte le système.

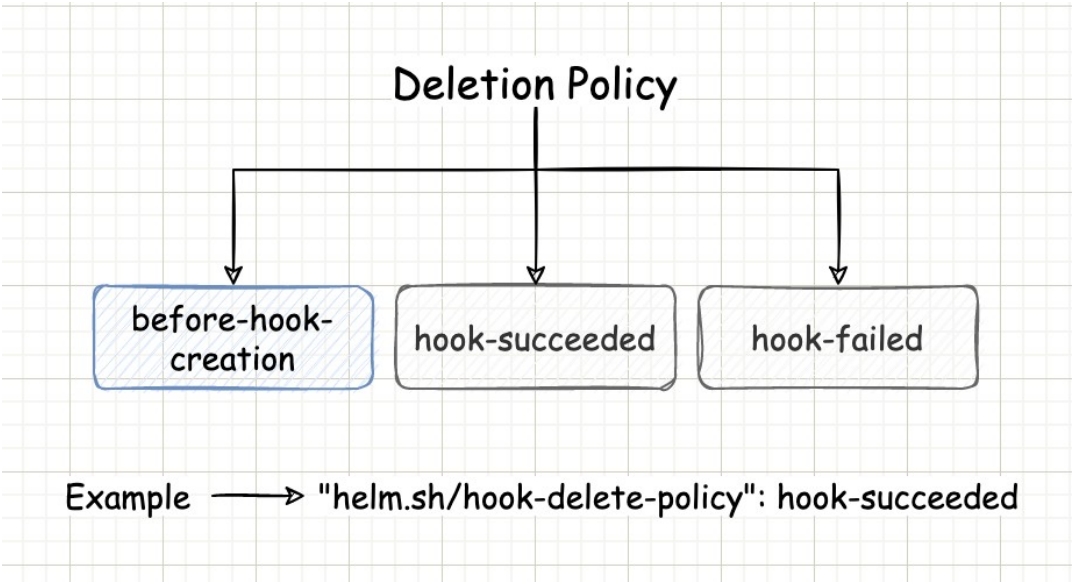
Que signifie attendre qu'un hook soit prêt ? Cela dépend de la ressource déclarée dans le hook. Si la ressource est de type Job ou Pod, Helm attendra jusqu'à ce qu'elle s'exécute avec succès jusqu'à la fin. Et si le hook échoue, la release échouera également. C'est une opération bloquante, donc Helm fera une pause pendant que le Job est exécuté.

Pour tous les autres types, dès que Kubernetes marque la ressource comme chargée (ajoutée ou mise à jour), la ressource est considérée comme "prête". Lorsque plusieurs ressources sont déclarées dans un hook, les ressources sont exécutées en série. Si elles ont des poids de hook (voir ci-dessus), elles sont exécutées dans l'ordre pondéré. À partir de Helm 3.2.0, les ressources de hook ayant le même poids sont installées dans le même ordre que les ressources normales sans hook. Dans le cas contraire, l'ordre n'est pas garanti. Il est considéré comme une bonne pratique d'ajouter un poids au hook, et de le mettre à 0 si le poids n'est pas important.

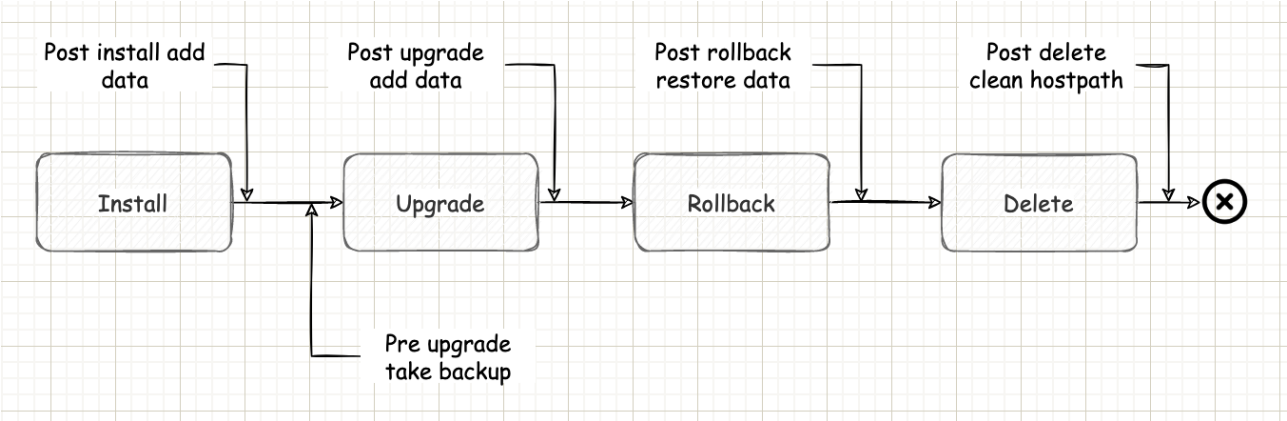
**Les ressources des hooks ne sont pas gérées avec les versions correspondantes**

Les ressources créées par un hook ne sont actuellement pas suivies ou gérées comme faisant partie de la Release. Une fois que Helm a vérifié que le hook a atteint son état prêt, il laissera la ressource du hook tranquille. La collecte des déchets des ressources de hook lorsque la version correspondante est supprimée pourrait être ajoutée à Helm 3 dans le futur, donc toutes les ressources de hook qui ne doivent jamais être supprimées devraient être annotées avec `helm.sh/resource-policy : keep`.

En pratique, cela signifie que si vous créez des ressources dans un hook, vous ne pouvez pas compter sur `helm uninstall` pour les supprimer. Pour détruire de telles ressources, vous devez soit ajouter une annotation personnalisée `helm.sh/hook-delete-policy` au fichier modèle du hook, soit définir le champ time to live (TTL) d'une ressource Job.



Les hooks sont des objets Kubernetes autres que des Pods et des Jobs, alors les hooks sont prêts dès que ces objets sont chargés ou mis à jour.




Helm nous fournit un mécanisme qui nous permet d'intervenir à certains moments du cycle de vie d'une version. Par exemple, nous pouvons utiliser des crochets pour :

- Prioriser la tâche pendant le processus d'installation. Comme, par exemple, charger une ConfigMap ou un Secret avant que tout autre composant des chartes ne soit chargé.

- Sauvegarder une base de données avant d'installer/mettre à niveau un nouveau graphique ou restaurer la base de données après la mise à niveau.

Le fichier hook ressemble aux templates qui nous permet de générer des manifestes kubernetes. Supposons que nous voulions charger une ConfigMap kubernetes avant toute autre installation. Un template avec des hooks ressemblera à ceci

```
kind: ConfigMap
metadata:
  name: "{{ .Release.Name }}"
  annotations:
    "helm.sh/hook": pre-install
    "helm.sh/hook-weight": "5"
    "helm.sh/hook-delete-policy": before-hook-creation
data:
 name: "{{ .Release.Name }}"
```

## Écrire un hook

Les hooks sont simplement des templates qui nous permet de créer des fichiers manifestes Kubernetes. À l'exception de quelques annotations spéciales dans la section des métadonnées.

Comme il s'agit de fichiers modèles, vous pouvez utiliser toutes les fonctionnalités normales des modèles, y compris la lecture des Objets `.Values, .Release` et `.Template.`

Par exemple, ce modèle, stocké dans templates/post-install-job.yaml, déclare un job à exécuter lors de la post-installation :

```
apiVersion: batch/v1
kind: Job
metadata:
```

```
  name: "{{ .Release.Name }}"
  labels:
    app.kubernetes.io/managed-by: {{ .Release.Service | quote }}
    app.kubernetes.io/instance: {{ .Release.Name | quote }}
    app.kubernetes.io/version: {{ .Chart.AppVersion }}
    helm.sh/chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
  annotations:
    # This is what defines this resource as a hook. Without this line, the
    # job is considered part of the release.
    "helm.sh/hook": post-install
    "helm.sh/hook-weight": "-5"
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  template:
    metadata:
      name: "{{ .Release.Name }}"
    spec:
      restartPolicy: Never
      containers:
      - name: post-install-job
        image: "alpine:3.3"
        command: ["/bin/sleep","{{ default "10" .Values.sleepyTime }}"]
```

Ce qui fait de ce modèle un hook est l'annotation :

```
annotations:
  "helm.sh/hook": post-install
```

Une ressource peut implémenter plusieurs hooks :

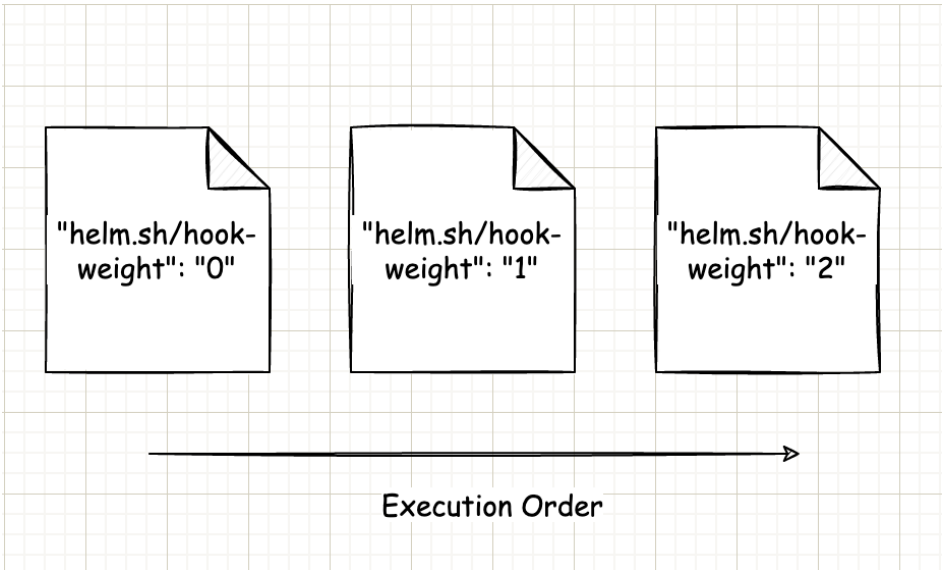
```
annotations:
  "helm.sh/hook": post-install,post-upgrade
```

De même, il n'y a pas de limite au nombre de ressources différentes qui peuvent implémenter un hook donné. Par exemple, on peut déclarer à la fois un secret et une Configmap comme hook de pré-installation.

```
kind: ConfigMap
metadata:
  name: "{{ .Release.Name }}"
  annotations:
    "helm.sh/hook": pre-install
    "helm.sh/hook-weight": "5"
    "helm.sh/hook-delete-policy": before-hook-creation
data:
 name: "{{ .Release.Name }}"
```

Lorsque des sous-charts déclarent des hooks, ceux-ci sont également évalués. Il n'existe aucun moyen pour un chart parent de désactiver les hooks déclarés par les sous-chartes.

**Ordre d'exécution des hooks**

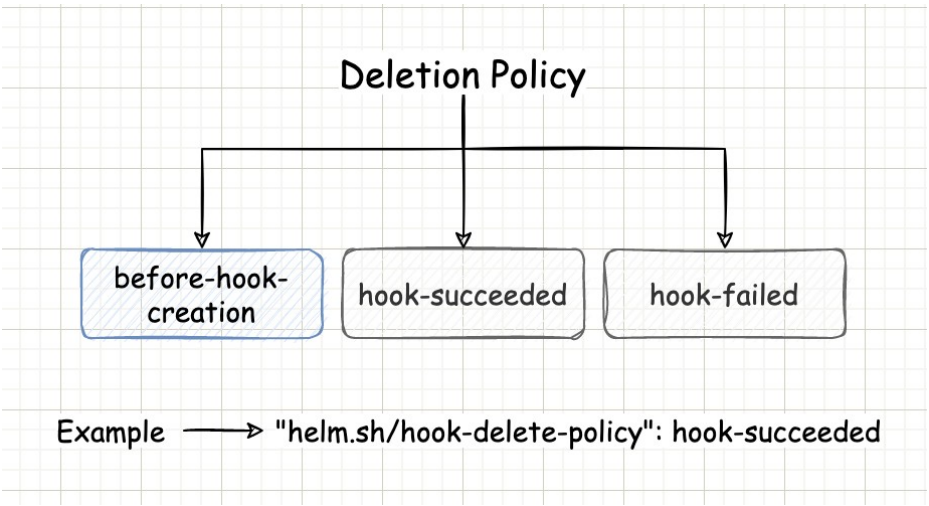



Il est possible de définir un poids pour un hook qui aidera à construire un ordre d'exécution déterministe. Les poids sont définis à l'aide de l'annotation suivante :

```
annotations:
  "helm.sh/hook-weight": "5"
```

Les poids des hooks peuvent être des nombres positifs ou négatifs, mais doivent être représentés sous forme de chaînes de caractères. Lorsque Helm démarre le cycle d'exécution des hooks d'un Kind particulier, il triera ces hooks dans l'ordre croissant.

**Politiques de suppression des hooks**




Il est possible de définir des politiques qui déterminent quand supprimer les ressources de hooks correspondantes. Les politiques de suppression de hooks sont définies à l'aide de l'annotation suivante :

```
annotations:
  "helm.sh/hook-delete-policy": before-hook-creation,hook-succeeded
```

Vous pouvez choisir une ou plusieurs valeurs d'annotation définies :

- **before-hook-creation** Supprimer la ressource précédente avant qu'un nouveau hook ne soit lancé (par défaut)
- **hook-succeeded** Supprime la ressource après l'exécution réussie du hook

Si aucune annotation relative à la politique de suppression des hooks n'est spécifiée, le comportement "avant la création du hook" (**before-hook-creation**) s'applique par défaut.

# TP – Sauvegarder de notre DB avec un hook

Dans le TP précédant, nous avons produit deux charts.

Un frontend Web et une base de données.

Les données de cette derniere sont persistante via un PVC dont le nom est {{ .Values.volumes.claimName }} Pour ma configuration

Ajouter un hook qui créer un PersistanteVolumeClaim et sauvegarde les données créer par notre application paye, dans ce stockage.

# Chart Dependencies

Helm propose une gestion des dépendances des charts. Nous avons parfois besoin d'executer un chart avant un autre chart. Typiquement, la création d'une base de données avant la partie de la création de l'applicatif.

## dépendances

Gère les dépendances d'un chart.

Les charts Helm stockent leurs dépendances dans le dossier `charts/`. Pour les développeurs de chart, Il est souvent plus facile de gérer les dépendances dans le fichier `Chart.yaml` qui déclare toutes les dépendances.

Les commandes de dépendance fonctionnent a partir de ce fichier, ce qui facilite la synchronisation entre les dépendances souhaitées et les dépendances réelles stockées dans le répertoire `charts/`.

Par exemple, ce fichier Chart.yaml déclare deux dépendances :

```
# Chart.yaml
dependencies:
- name: nginx
  version: "1.2.3"
  repository: "https://example.com/charts"
- name: memcached
  version: "3.2.1"
  repository: "https://another.example.com/charts"
```

Le 'name' doit être le nom d'un chart, où ce nom doit correspondre au nom dans le fichier `Chart.yaml` de ce chart.

Le champ 'version' doit contenir une version sémantique ou une plage de version.

L'URL du 'repository' doit pointer vers un dépôt de Chart. Helm s'attend à ce qu'en ajoutant `/index.yaml` à l'URL, il puisse récupérer l'index du dépôt de chart. @.

À partir de la version 2.2.0, le dépôt peut être défini comme le chemin d'accès au répertoire des charts dépendants stockés localement. Le chemin doit commencer par le préfixe `file://`. Par exemple :

```
# Chart.yaml
dependencies:
- name: nginx
  version: "1.2.3"
  repository: "file://../dependency_chart/nginx"

- name: mariadb
  version: "0.1.0"
  repository: "file://../mariadb/"
  condition: db.enabled
```

Si le chart dépendant est récupéré localement, il n'est pas nécessaire d'ajouter le dépôt à Helm avec la commande `helm add repo`. La correspondance des versions est également prise en charge pour ce cas.

lister les dépendances pour un chart donné

La commande `helm dependency list`, liste toutes les dépendances déclarées dans un chart.

Cela prend en entrée les archives de chart et le répertoire des charts. Cela ne modifiera pas le contenu d'un chart. Cela produira une erreur si le chart ne peut pas être chargé.

```
helm dependency list CHART
```

Chargement des dépendances

`helm dependency update <chart>`

met à jour le dossier charts/ basé sur le contenu du fichier Chart.yaml

Cette commande vérifie que les charts requis, tel qu'exprimés dans le fichier `Chart.yaml`, soient présent dans le dossier `charts/` et sont dans une version acceptable. Elle récupéra les derniers charts qui satisfont les dépendances et nettoiera les anciennes dépendances.

En cas de mise à jour réussie, cela générera une fichier lock qui pourra être utilisé pour reconstruire les dépendances vers une version spécifique.

Il n'est pas nécessaire que les dépendances soient présentes dans le fichier `Chart.yaml`. Pour cette raison, une commande de mise à jour ne supprimera pas les charts à moins qu'ils ne soient présents dans le fichier `Chart.yaml` mais dans une version incorrecte.

**helm dependency build**

reconstruire le répertoire charts/ à partir du fichier Chart.lock

Reconstruire le répertoire charts/ à partir du fichier Chart.lock.

Build est utilisé pour reconstruire les dépendances d'un chart selon l'état spécifié dans le fichier lock. Cela ne renégociera pas les dépendances, comme le fait la commande `helm dependency update`.

Si aucun fichier lock n'est trouvé, la commande `helm dependency build` aura le même comportement que la commande `helm dependency update`.

# TP – Gérer la dépendances de nos charts.

Notre chart Frontend dépends de notre chart DB.

Créer une dépendances de facon que notre DB démarre avant notre Front web.