Qu'est-ce que le Chart Helm ?

Pour les besoins de l'explication, je choisis un exemple très basique de déploiement d'un frontend de site 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,

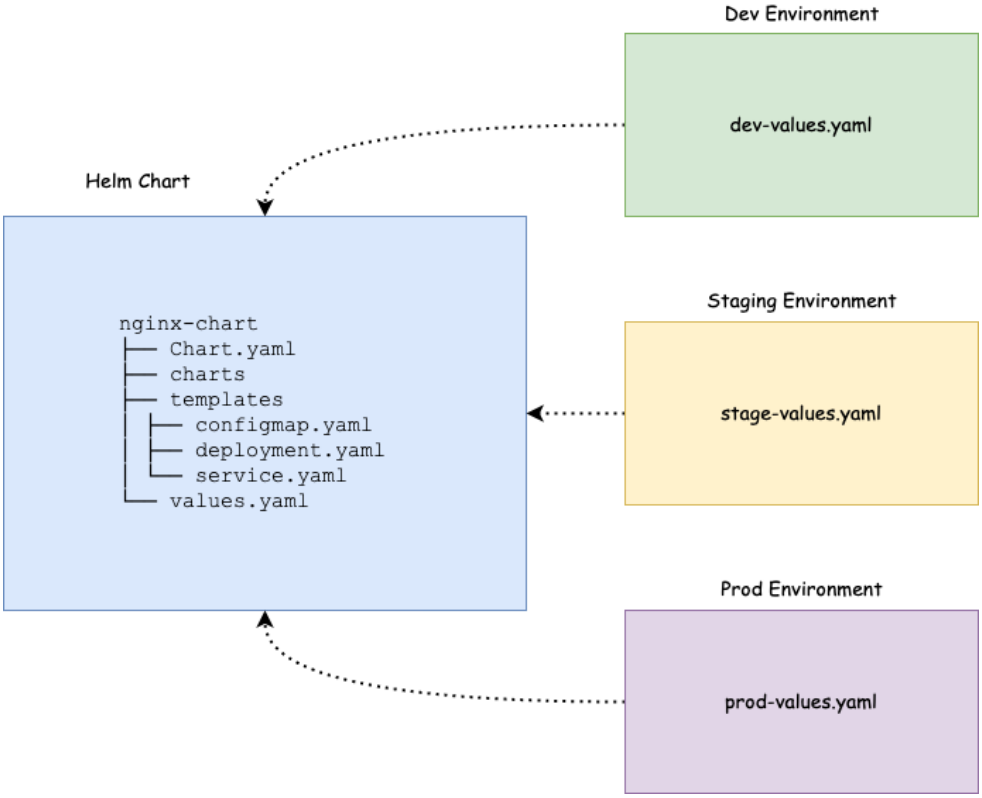
Dans les environnements Dev et QA, vous n'aurez peut-être besoin que d'une seule réplique.

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 Nginx différents pour chaque environnement. Ou bien vous aurez un seul fichier de déploiement et vous devrez écrire des scripts shell ou python 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).



Qu'est-ce qu'un Chart Helm ?

Les Charts Helm sont une combinaison de modèles de manifeste YAML de Kubernetes et de fichiers spécifiques à Helm. Il s'agit en quelque sorte d'un paquet helm. Comme le manifeste YAML de Kubernetes peut être modélisé, vous n'avez pas besoin de maintenir plusieurs Charts helm pour différents environnements. Helm utilise le moteur de templating go pour la fonctionnalité de templating.

Vous pouvez modifier les paramètres de déploiement de chaque environnement en changeant un seul fichier de valeurs. Helm se chargera d'appliquer les valeurs aux modèles.

Les Charts Helm réduisent la complexité et la duplication de chaque environnement (dev, uat, cug, prod) avec un seul modèle.

Structure du Chart de Helm

Pour comprendre le Chart Helm, prenons un exemple de déploiement de Nginx. Pour déployer Nginx sur Kubernetes, vous auriez typiquement les fichiers YAML suivants.

```
nginx-deployment
    ├── configmap.yaml
    ├── deployment.yaml
    ├── ingress.yaml
    └── service.yaml
```

Maintenant, si nous créons un Chart Helm pour le déploiement de Nginx ci-dessus, il aura la structure de répertoire suivante.

```
nginx-chart/
|-- Chart.yaml
|-- charts
|-- templates
|   |-- NOTES.txt
|   |-- _helpers.tpl
|   |-- deployment.yaml
|   |-- configmap.yaml
|   |-- ingress.yaml
|   |-- service.yaml
|   `-- tests
|       `-- test-connection.yaml
`-- values.yaml
```

Comme vous pouvez le voir, les fichiers YAML de déploiement font partie du répertoire template (surligné en gras) et il y a des fichiers et des dossiers spécifiques à helm. Examinons chaque fichier et répertoire à l'intérieur d'un Chart helm et comprenons leur importance.

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

Chart.yaml : Il contient des informations sur le tableau de bord comme la version, le nom, la description, etc.

values.yaml : Dans ce fichier, nous définissons les valeurs des modèles YAML. Par exemple, le nom de l'image, le nombre de répliques, les valeurs HPA, etc.

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.

templates : Ce répertoire contient tous les fichiers manifestes Kubernetes qui forment une application. Ces fichiers manifestes peuvent être modelés 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.

Helm Chart Tutorial GitHub Repo

L'exemple de chart Helm et les manifestes utilisés dans ce guide sont hébergés sur le repo Github de Helm Chart. Vous pouvez le cloner et l'utiliser pour suivre le guide.

```
git clone https://github.com/techiescamp/helm-tutorial.git
```

Création d'un Chart Helm à partir de zéro

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

```
helm create nginx
```

On se rende dans le répertoire du chart.

```
cd nginx-chart
```

Nous allons éditer les fichiers un par un en fonction de nos besoins de déploiement.

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 Helm
type : application
version : 0.1.0
appVersion : "1.0.0"
mainteneurs :
- email : contact@plb.com
  name : plb
```

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 (Nginx).

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

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/*
```

Nous allons ajouter nos fichiers YAML Nginx et les modifier en fonction du modèle pour une meilleure compréhension.

Créez un fichier deployment.yaml et copiez le contenu suivant.

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

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: "nginx:1.16.0"
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
```

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. 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.

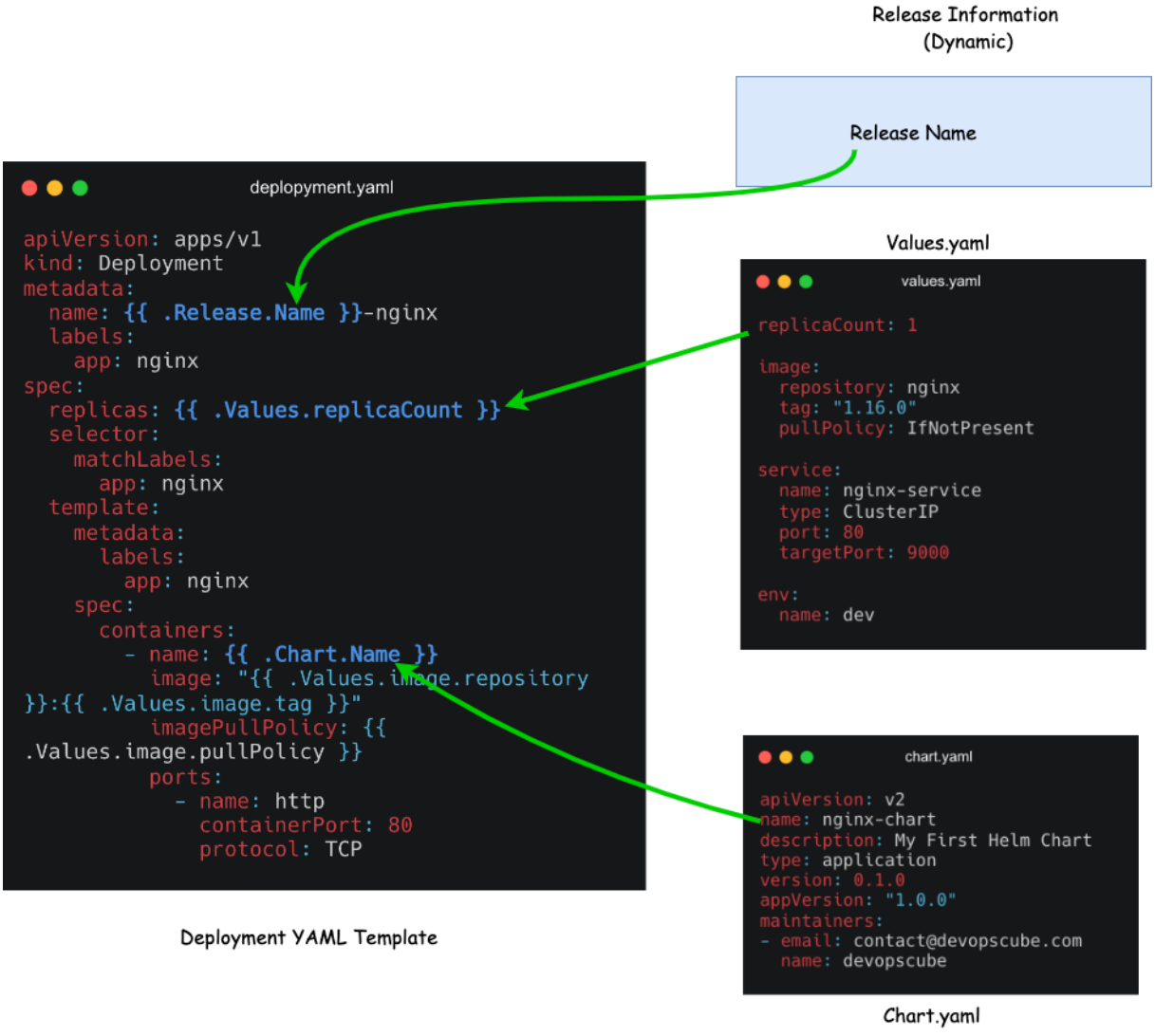
Chart : Si vous souhaitez utiliser les valeurs mentionnées dans le chart.yaml, vous pouvez utiliser l'objet chart.

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

Pour en savoir plus sur les objets pris en charge, consultez le document 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 déterminer quelles valeurs pourraient 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 : Nous devons changer le nom du déploiement à chaque fois car Helm ne nous permet pas d'installer des releases avec le même nom. Nous allons donc templatiser le nom du déploiement avec le nom de la release et interpoler -nginx avec lui. Maintenant, si nous créons 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 de gabarit 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 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
```

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

```
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 }}
```

Dans la directive du modèle de protocole, vous pouvez voir un Pipe (|) . Il est utilisé pour définir la valeur par défaut du protocole comme étant TCP. Cela signifie que nous ne définirons pas la valeur du protocole dans le fichier values.yaml ou, s'il est vide, qu'il prendra TCP comme valeur pour le protocole.

Créez un fichier configmap.yaml et ajoutez-y le contenu suivant. Ici, nous remplaçons la page index.html par défaut de Nginx par une page HTML personnalisée. Nous avons également ajouté une directive template pour remplacer le nom de l'environnement en HTML.

```
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 de modèle que nous avons utilisées dans les modèles. Par exemple, le modèle deployment.yaml contient une directive de modèle pour obtenir le référentiel d'images, le tag et la politique d'extraction à partir du fichier values.yaml. Si vous vérifiez le fichier values.yaml suivant, nous avons des paires clé-valeur

repository, tag et pullPolicy imbriquées sous la clé image. C'est la raison pour laquelle nous avons utilisé Values.image.repository

Maintenant, remplacez le contenu du fichier values.yaml par défaut par ce qui suit.

```yaml
replicaCount: 2

image:
  repository: nginx
  tag: "1.16.0"
  pullPolicy: IfNotPresent

service:
  name: nginx-service
  type: ClusterIP
  port: 80
  targetPort: 9000

env:
  name: dev
```

Nous avons maintenant le tableau de bord 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
└── values.yaml
```

Valider le Chart de Helm

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

```
helm lint .
```

Si vous l'exécutez depuis l'extérieur du répertoire nginx-chart, indiquez le chemin complet de nginx-chart

```
helm lint /chemin/vers/nginx-chart
```

S'il n'y a pas d'erreur ou de problème, le résultat sera le suivant

```
==> Linting ./nginx
[INFO] Chart.yaml :

1 chart(s) linted, 0 chart(s) failed
```

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.

```
helm template .
```

Nous pouvons également utiliser la commande --dry-run pour vérifier. Elle fera semblant d'installer le Chart sur le cluster et, en cas de problème, elle affichera l'erreur.

```
helm install --dry-run my-release nginx-chart
```

Si tout est bon, vous verrez le manifeste qui sera déployé dans le cluster.

Déployer le Chart Helm

Lorsque vous déployez le chart, Helm lit les valeurs de la carte et de la configuration dans le fichier values.yaml et génère les fichiers manifestes. Il envoie ensuite ces fichiers au serveur API de Kubernetes, et Kubernetes crée les ressources demandées dans le cluster.

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

```
helm install frontend nginx-chart
```

Vous verrez la sortie comme indiqué ci-dessous.

```
NOM : frontend
LAST DEPLOYED : Tue Dec 13 10:15:56 2022
NAMESPACE : default
STATUT : deployed
REVISION : 1
SUITE DE TEST : Aucune
```

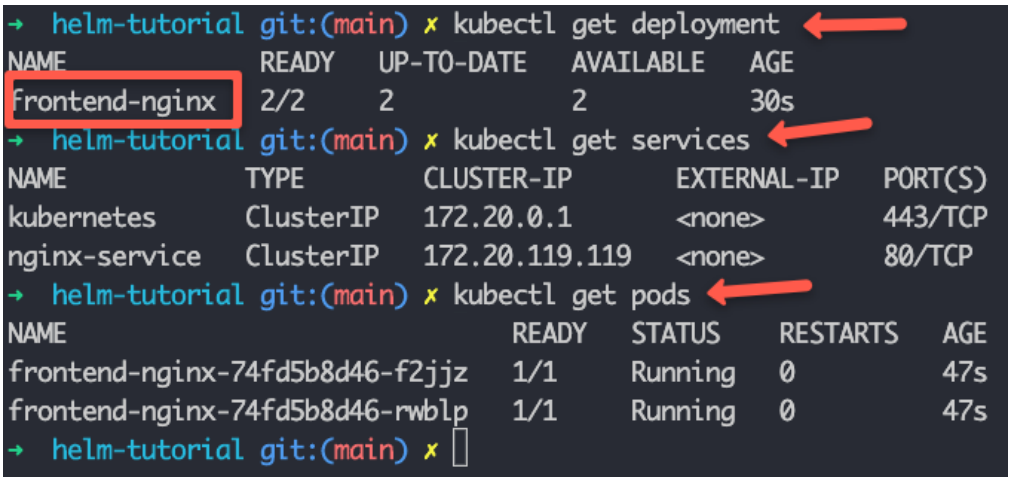
Vous pouvez maintenant vérifier la liste des versions en utilisant cette commande :

```
helm list
```

Exécutez les commandes kubectl pour vérifier le déploiement, les services et les pods.

```
kubectl get deployment
kubectl get services
kubectl get configmap
kubectl get pods
```

Nous pouvons voir que le déploiement frontend-nginx, nginx-service et les pods sont opérationnels comme indiqué ci-dessous.



Nous avons vu comment un Chart peut être utilisé pour plusieurs environnements à l'aide de différents fichiers values.yaml. Pour installer un Chart helm avec un fichier values.yaml externe, vous pouvez utiliser la commande suivante avec l'option --values et le chemin du fichier values.

```
helm install frontend nginx-chart --values env/prod-values.yaml
```

Lorsque Helm fait partie de votre pipeline CI/CD, vous pouvez écrire une logique personnalisée pour passer le fichier de valeurs requis en fonction de l'environnement.