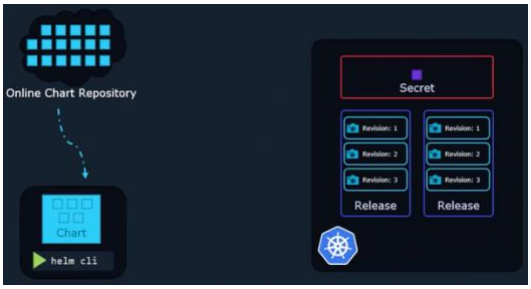
HELM V3

Installation
Se référer à la documentation d'installation se trouvant sur le site officiel

Concepts
Différences fondamentales entre Helm V2 et Helm V3

1) Avec Helm V2 installé en local sur votre poste, vous ne pouvez pas interagir directement avec le cluster, pour cela vous avez besoin de l'agent Tiller qui doit être installé sur le nœud master du cluster, contrairement à Helm V3 qui lui peut interagir directement avec le cluster en utilisant le Kubeconfig file de kubectl.
2) Lorsque l'on applique une configuration sur le cluster à l'aide de Helm, ce dernier crée une nouvelle révision de l'objet modifié afin de permettre un rollback, Helm V2 lors de rollback ne se fie qu'aux différentes révisions, donc ne saura pas réagir si jamais une modification est faite à la main, tandis que Helm V3 en plus de regarder les révisions crées par Helm, prend aussi en compte l'état (Live State) de la ressource et si jamais il se rend compte qu'il y'a des différences entre la dernière révision et le live state, alors il fait néanmoins le rollback

Composants
Les principaux composants de helm sont :



- ***Les charts*** : qui sont des modules ou des rôles(ansible) composé de différents templates variabilisés permettant de créer toutes les ressources kubernetes nécessaires au bon fonctionnement d'une application spécifique
- ***Les releases*** : l'application d'un Chart Helm sur kubernetes crée ce qu'on appelle un release possédant un nom et identifiant unique et permettant ainsi à Helm de pouvoir appliquer une chart Helm à plusieurs reprises sur un cluster (déploiement de la même appli dans divers environnements)
- ***Les révisions*** : qui sont en fait les différents état d'une release fonction des modifications et des application de chart helm faites dans la release (une release peut avoir une ou plusieurs révisions)
- ***Les Metadata*** : ensemble d'informations sur les composants de Helm permettant de les référencer. Ces metadata peuvent être stocké en local sur le poste Helm (Si utilisation unique) ou alors directement sur le cluster sous forme de secrets (utilisation commune ou collaboratif) afin de permettre à tous les utilisateurs de Helm sur ce cluster d'être à jour sur les différentes releases, révisions et autres.

Helm Charts
Dans les Helm charts, on retrouve :

- les templates variabilisées qui permettront de créer les ressources nécessaires (deployment, service…)
- un fichier ***values.yaml*** contenant en effet les valeurs à utiliser pour les différentes variables définies dans les templates.
- Un fichier ***chart.yml*** contenant les informations sur le chart en lui-même, la version de l'API helm qui sera utilisée, la version de l'appli à déployer et d'autres specs liées au chart lui-même.

NB : Il existe des charts open source ou communautaires qui peuvent être utilisés pour déployer certaines apps. Ces charts sont accessibles sur « artifacthub.io »

Structure d'un Chart Helm

Le fichier chart.yml contient en effet plusieurs champs qui sont détaillés ci-dessous :

Helm 2 v1 | Helm 3 v2

- ***ApiVersion*** : version de l'Api Helm à utiliser, champ introduit par Helm3 (Api v1 pour Helm2 et Api v2 pour Helm3)
- ***appVersion*** : version de l'application censée être déployée par ce chart
- ***version*** : version du chart en lui-même, afin de suivre et versionner les modifications
- ***name*** : nom de l'application ou du chart permettant de déployer l'application
- ***type*** : type de chart (Application ou Library), le type par défaut est Library
- ***dependencies*** : dépendances nécessaires au bon fonctionnement du chart avec pour chaque dépendance, la condition nécessaire et la source
- ***keywords*** : mots clés associés à ce chart, facilitant la recherche de ce chart dans l'artifacthub.io
- ***maintainers*** : informations liées au créateur ou mainteneur du chart
- ***home/icon*** : permettant si voulu de partager l'url du repo duc chart.

Pour résumer, la structure d'un chart Helm est la suivante

- ***Les Templates*** : contenant les manifests yaml variabilisés de création des ressources
- ***Values.yaml*** : contenant les valeurs des variables utilisées dans les templates
- ***chart.yaml*** : contenant les infos sur le chart en lui même
- ***LICENSE*** : optionnel contenant les infos sur la License du chart
- ***Readme.md*** :contenant la documentation sur le chart
- ***Charts*** : contenant éventuellement d'autres charts dont le chart principal en serait dépendant.

Les commandes

Après avoir installé Helm, l'exploitation se fait à partir du Helm CLI dont les commandes principales sont les suivantes :

- ***helm --help*** : Affiche l'aide du HELM CLI
- ***helm repo --helm*** : affiche l'aide helm sur les commandes liées au repo
- ***helm repo update --help***
- ***helm search [hub/repo] <chart-name>*** :permet de rechercher le charte en question sur artifacthub.io ou sur un repo.
- ***helm repo add <local-repo-name> <chart repo url>*** : permet de télécharger et d'ajouter le repo de ladite chart en local
- ***helm repo list*** : permet d'afficher l'ensemble des repo charts en local sur notre machine
- ***helm repo update*** : permet de mettre à jour les charts repo en local
- ***helm repo remove <local-repo-name>*** : supprime le chart repo local passé en paramètre
- ***helm install <release-name> <chart-name>*** : permet d'appliquer une chart helm sur un cluster kubernetes en créant la release à partir du nom passé en paramètre
- ***helm install <chart-name>*** : appliquer un chart sans release, dans ce cas on ne peut qu'avoir un seul environnement ou une seule instance de ce chart sur le cluster.
- ***helm list*** : permet d'afficher l'ensemble des releases déployées dans notre environnement Helm
- ***helm uninstall <my-release>*** : permet de supprimer toutes les ressource kubernetes liées à cette release
- ***helm install --set <variable=valeur> <my-release> <chart-name>***: permet d'installer un chart helm sous une release en surchargeant certaines de ses variables à l'aide de variables passée en paramètre
- ***helm install --values <custom-values-file> <my-release> <chart-name>***: permet d'installer un chart helm sous une release en surchargeant certaines de ses variables à l'aide d'un fichier de variables
- ***helm install <my-release> <chart-name> --version x.x.x***: permet d'installer un chart helm sous une release en spécifiant la version du chart que l'on souhaite utiliser
- ***helm pull <chart name>*** : permet de télécharger les fichiers d'un chart sous format zippé
- ***helm pull --untar <hart-name>***:télécharge les fichiers de la chart et les dézippe directement dès lors on peut alors directement modifier le fichier values.yaml
- ***helm install <my-release> <Path-to-edited-chart>***:permet de lancer ses propres charts ou alors une chart aux valeurs customisées.
- ***helm upgrade <my-release> <hart-name>***:met à jour une release à partir du chart (met à jour les images et ressources fonction de la mise à jour faite sur le chart). cela crée également une nouvelle révision pour cette release
- ***helm history <my-release>*** : permet de lister l'ensemble des révisions de ladite release avec les détails sur la version de l'application et du chart utilisé pour chacune des révisions
- ***helm rollback <my-release> 1*** : permet de faire un rollback sur la release passée en paramètres afin de retourner sur la révision 1 (numéro de la révision passée également en paramètres), En effet techniquement Helm crée une autre révision (3) qui est similaire à la révision 1.
- ***helm upgrade dazzling-web bitnami/nginx --version 12*** : mettre à jour une release en utilisant la chart nginx version 12 du repo local bitnami

Création de Charts Helm

pour créer ses propres chart helm, on commence déjà par créer les dossiers et fichiers nécessaires pour respecter la structure des Charts helm vu ci-dessus. Pour cela on peut utiliser la commande : « ***helm create <chart-name>*** »

Par la suite, comme avec les rôles ansible, on peut modifier le contenu des différents fichiers, soit ajouter des informations dans le fichier chart.yml ; variabiliser notre chart en définissant les variables dans le fichier values.yml ; ensuite il faut mettre les manifest de création de nos ressources kubernetes dans le dossier templates en les variabilisant. le référencement des variables dans les Template se fait de la manière suivante : ***{{ .Values .<var-name> }}***. Il s'agit en effet du langage de Scripting GO. les bonnes pratiques recommandent de renommer toutes les ressources d'un template à l'aide de la variable par défaut « ***{{ .Release.Name }}*** » qui correspond en effet au nom de la release lors du lancement de la commande d'application d'un chart. Il existe plusieurs autres variables spéciales de ce genre : ***Release.Name*** ; ***Release.Namespace*** ; ***Release.IsUpgrade*** ; ***Release.IsInstall*** ; ***Release.Revision*** ; ***Release.Service***

En effet ces variables par défaut sont des variables liées au objet natifs de helm ou de kubernetes (Built-In-Object) et toutes les variables définies par l'utilisateur dans le fichier values.yaml sont référencées dans

Vérifier son chart Helm avant de l'installer

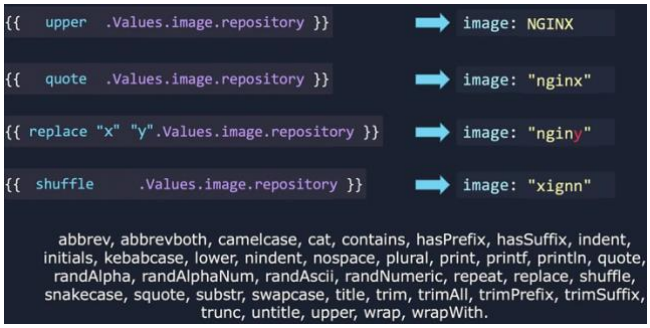
pour cela on peut utiliser des linters Helm qui nous aident à vérifier la syntaxe Yaml et également que notre template fonctionne correctement et pour terminer vérifie en mode dry-run que le chart fonctionne comme souhaité dans notre environnement kubernetes.

Les commandes du Linter sont les suivantes :

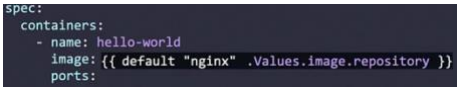
- ***helm lint ./nginx-chart*** : permet de tester la syntaxe yaml d'un chart afin de détecter des erreurs de syntaxe ou d'indentation
- ***helm template ./nginx-chart*** :permet d'exécuter le chart localement et de produire le résultat des différents manifests kubernetes déjà templatisés qui devront être appliqués sur le cluster k8s
- ***helm template <release-name> <chart-name>*** : même commande que celle-ci-dessus mais ici on fournit le nom de la release. si le nom de la release n'est pas fourni, alors Helm utilise sa valeur par défaut « ***RELEASE-NAME*** »
- ***helm template <release-name> <chart-name> --debug*** : permet néanmoins d'afficher les manifests de sortie même en cas d'erreur du template pour des besoins de troubleshooting
- ***helm install <release> <char-path> --dry-run*** : permet de réellement tester les manifests kubernetes sur le cluster kubernetes et retourner les éventuelles erreurs liées à kubernetes.

Les Fonctions dans les Charts Helm

Il existe dans helm des fonctions prédéfinies qui nous permettent de traiter nos inputs(variables) avant d'afficher leur valeur, exemple (mettre en majuscule, supprimer les espaces inutiles, inverser, remplacer une lettre ou un mot et bien d'autres). ces fonctions ci-après sont presque toutes liées à des chaines de caractères, mais il existe de nombreuses fonctions HELM liées à d'autre contexte (math, algorithme, Date, dictionnaires, kubernetes, charts, File Path, network, regEx…)



La fonction qui est très souvent utilisée dans les charts Helm est la fonction « default » qui permet en effet de fournir une valeur par défaut d'une variable dans le template si jamais celle-ci n'est pas définie dans le values.yaml ou lors du lancement de la commande.



Helm Pipelines

En fait il s'agit de la notion de pipe « | » généralement utilisée dans Linux. Helm permet également de l'utiliser notamment lors de l'utilisation des fonctions dans le template.

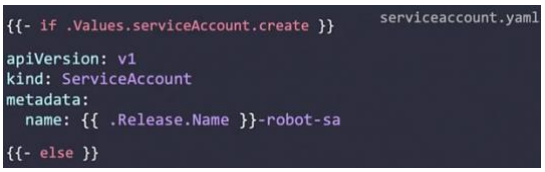
Les expressions conditionnelles

Il est possible d'ajouter des structures conditionnelles dans les templates helm, cela fonctionne similairement aux conditions dans les langages de programmation. par exemple mettre une condition d'écriture d'une ligne si et seulement si une variable est définie.

Dans ce cas les lignes en question sont encadrées dans le bloc conditionnel.

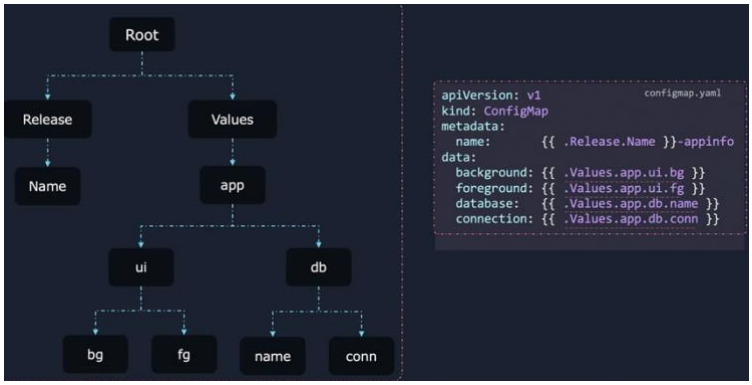
NB : avec cet exemple ci-après, les lignes conditionnelles (if et end) seront considérées bien que vides, ce qui créera des espaces blanc dans notre manifest de sortie. pour corriger cela, il faut ajouter « - »

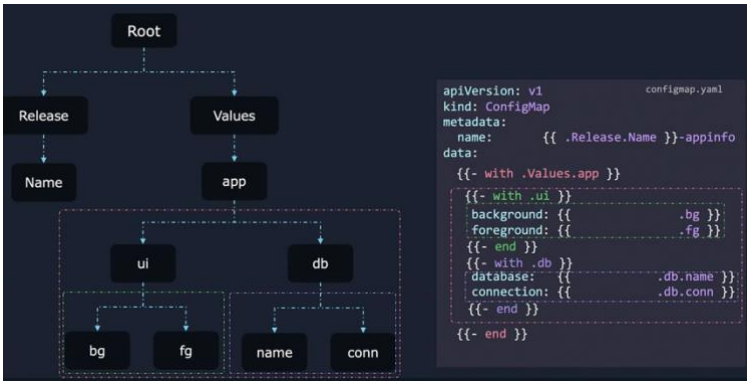
Il existe également un *else if* et *else* dans Helm et cela s'utilise avec des fonctions telles que « *eq* » qui prend en paramètres 2 arguments et retourne true si leurs valeurs sont égales.



Le Scope et les bloc With

En effet lors du référencement des variables dans les templates Helm, par défaut les variables utilisent le scope root qui est le scope de base. Il est cependant possible pour des soucis d'optimisation et de faciliter de créer son propre scope qui s'appliquera à un bloc de lignes de notre template histoire de ne pas répéter une grande partie du code lors du référencement des variables





NB : Dans les blocs à scopes personnalisés ; les variables spéciales ou par défaut de Helm telle que .Release.Name ne sont plus accessibles. Mais ces variables peuvent être récupérés en utilisant le scope par défaut root qui est représenté par un `$` et accessible dans n'importe quel scope personnalisé. `release: {{ $.Release.Name }}`