Helm

KEEPCODING
Tech School

Helm

- En esta sección:
  - Introducción a Helm
  - Primeros pasos con Helm
    - Instalación de Helm 3
    - Configuración básica de helm
    - Desplegando paquetes y explorando recursos
  - Opciones de ejecución
  - **Diseño y creación de helm charts**
  - Ejercicios

Helm - Gestor de paquetes en Kubernetes

- Helm se autodenomina "**the package manager for Kubernetes**". Algo así como APT, YUM, APK pero para Kubernetes.
- Helm es un sistema muy potente de plantillas para manifiestos de Kubernetes.
- A los **paquetes de plantillas** los llamamos **Charts** (helm chart = carta de navegación).
- Cada Chart puede tener varios ficheros que son recursos de Kubernetes (a modo de templates).
- Los charts producirán un resultado dependiendo de los valores de entrada (generalmente siempre definen valores por defecto para una instalación estándar).
- Aplicaremos valores de configuración a los Charts durante la instalación mediante el fichero de **values.yaml**.

Helm, ¿Por qué?

- Considera todos los manifiestos y objetos de Kubernetes que vas a necesitar para desplegar **una** aplicación:
  - Deployment APP + ConfigMap / Secrets + servicio + HPA
  - StatefulSet + ConfigMap / Secrets + HeadLess service + Servicio
  - Recurso ingress con FQDN y certificados SSL para acceso desde el exterior.

- Ahora imagina que tienes que desplegar tu aplicación de nuevo, otra vez. ¿Qué opciones tienes? ¿Cuál sería el proceso?

Helm, ¿Por qué?

- Desplegar muchas aplicaciones en Kubernetes significa manejar muchísimos recursos.
- Muchos pueden ser además muy similares.
- Es necesario un sistema para poder agrupar (paquetizar) conjuntos de manifiestos y tratarlos como plantillas sobre las que poder generar los manifiestos finales

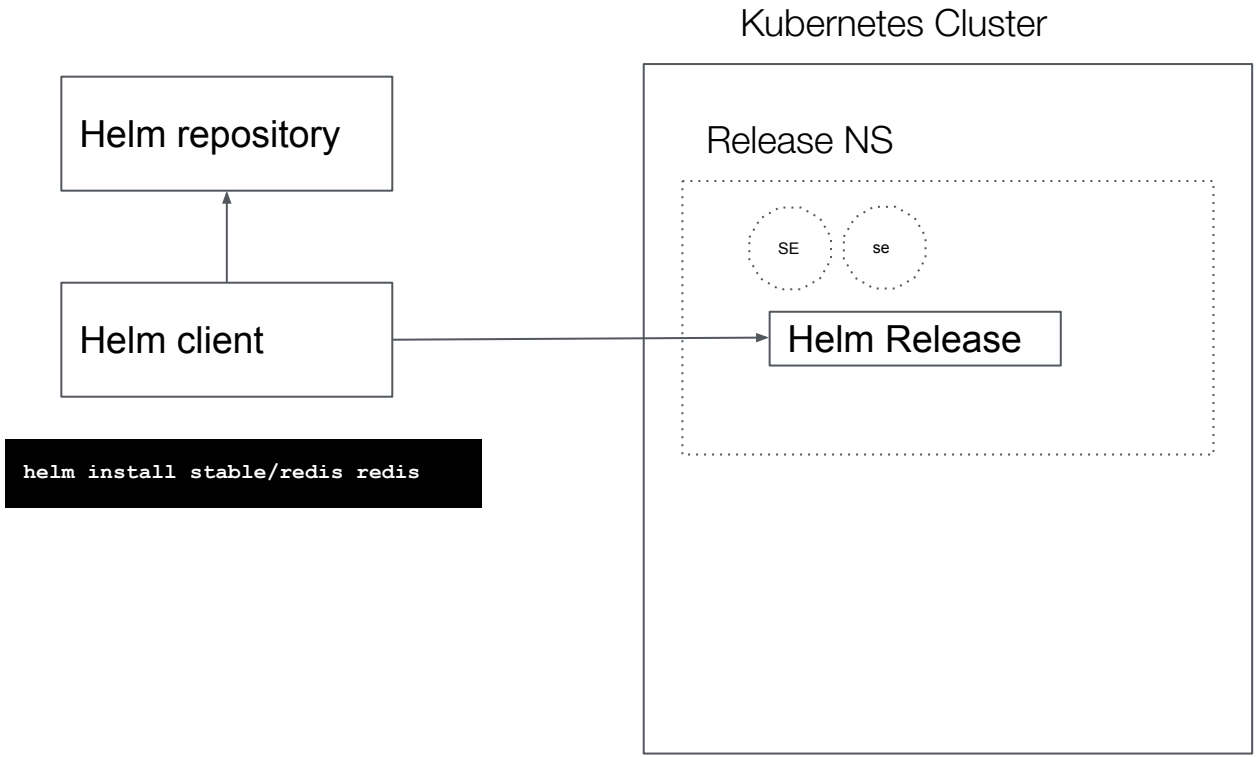
Componentes / Conceptos

- Comando "**helm**": es la herramienta principal capaz de procesar los charts e interactuar con nuestro cluster (internamente vía kubectl).
- **Charts**: son los paquetes en sí sobre los que operaremos (o crearemos)
- **Repositorios**: Al igual que en los sistemas de paquetes tradicionales son servidores que guardan la información de diversos charts.
- **Releases**: Se llama "release" a la instalación de un paquete de helm en un cluster. Una "release" es una instancia del chart ejecutándose en un cluster de Kubernetes.

Helm, casos de uso

- Al igual que con docker tenemos dos tipos de uso completamente diferenciados:
  - El que utiliza charts creados y diseñados por otros
  - El que diseña sus propios charts, ya sea para sus aplicaciones o aplicaciones de terceros.

Arquitectura de Helm 3



Glosario de términos

- **Chart**: Es el paquete de helm que contiene los recursos de Kubernetes a instalar.
- **Chart dependency**: Un chart puede depender de otros charts, al igual que en los sistemas de paquetes tradicionales.
- **Chart.yaml**: La meta información acerca del Chart se guarda en este fichero.
- **Release**: Cuando instalamos un chart se crea una release.
- **Release number**: Una misma release puede ser actualizada múltiples veces, el número de release es secuencial y aumenta en cada actualización.
- **Rollback**: Dado que se guarda el historial de releases podemos volver a una versión anterior indicando el número de release.
- **Repository**: Los charts pueden ser almacenados dentro de un repositorio de Charts. La comunicación se realiza por HTTP.
- **Values**: Son una forma de especificar parámetros con valores que no sean los de por defecto en nuestros Charts a la hora de hacer una instalación

Instalación y primeros pasos

KEEPCODING
Tech School

Instalación de Helm 3

- Install: https://helm.sh/docs/intro/install/
  - https://github.com/helm/helm/releases
- `helm --help`

```
$ wget
https://github.com/helm/helm/releases/download/v3.7.2/helm-v3.7.2-darwin-amd64.tar.gz.asc

$ tar xzvf helm-v3.7.2-darwin-amd64.tar.gz

$ sudo mv darwin-amd64/helm /usr/local/bin/

$ helm version
version.BuildInfo{Version:"v3.7.2", GitCommit:"663a896f4a815053445eec4153677ddc24a0a361",
GitTreeState:"clean", GoVersion:"go1.16.10"}
```

Primeros pasos con helm

- Quickstart: https://helm.sh/docs/intro/quickstart
  - Añade un repositorio
  - Busca paquetes (charts)
  - Instala un paquete

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
"bitnami" has been added to your repositories

$ helm repo update              # Make sure we get the latest list of charts

$ helm search repo wordpress
NAME                    CHART VERSION    APP VERSION     DESCRIPTION
bitnami/wordpress       12.3.3           5.8.3           Web publishing platform for building...

$ helm install mywp bitnami/wordpress

# And test access with kubectl port-forward for example :)
# what is the user and password to access the wp-admin site?
```

Analizando qué se ha desplegado

- Usa `helm list` para ver la lista de paquetes instalados (releases)
- `helm get` te ayuda a obtener detalles sobre las releases instaladas
  - manifest, hooks, notes, values, all

```
$ helm list
NAME  NAMESPACE  REVISION   UPDATED                                 STATUS     CHART
      APP VERSION
mywp  default    1          2022-01-18 21:02:13.133988 +0100 CET   deployed   wordpress-12.3.3
      5.8.3

$ kubectl get --all
$ helm get manifest mywp > wordpress_helm_installed.yaml
```

Analizando paquetes sin instalar

- Usa `helm show [chart|all]` para mostrar detalles sobre el paquete
- Usa `helm pull` para descargarte un chart.

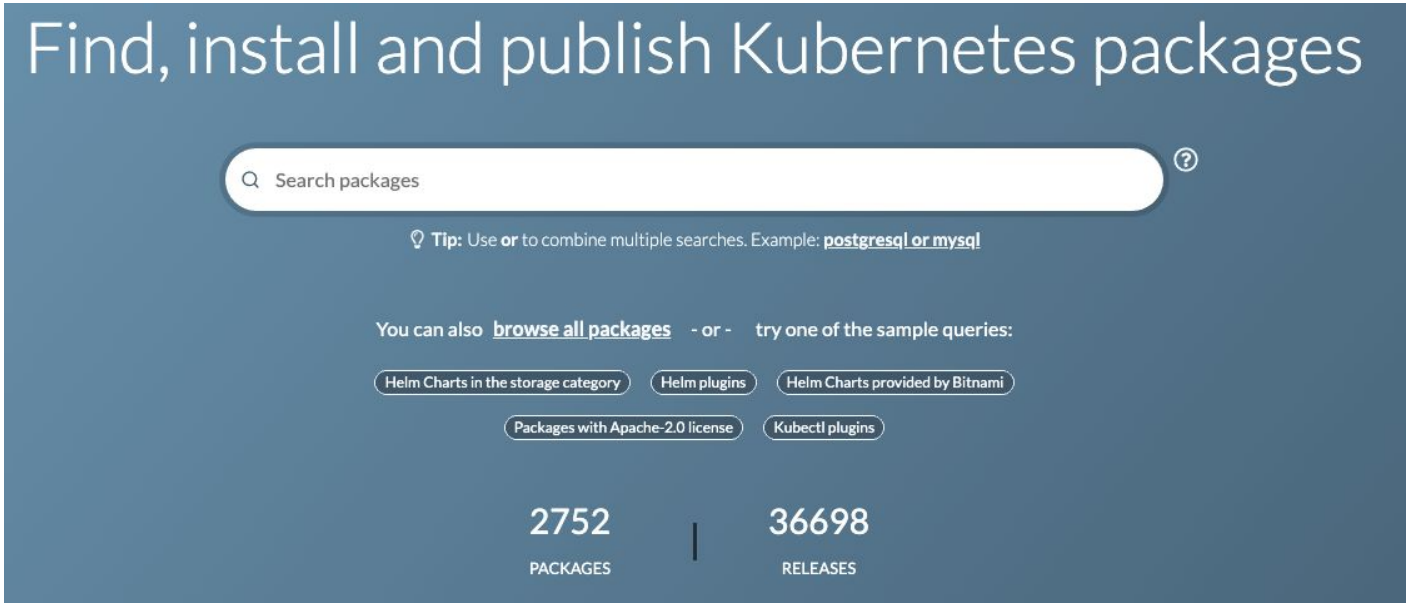
```
$ helm show chart bitnami/mysql

$ helm show all bitnami/mysql

$ helm pull bitnami/mysql
```

Consejos Helm

- Estudia cada paquete desde su documentación oficial antes de instalarlo para saber opciones de configuración, etc.
- `helm pull` por si quieres descargarte un CHART completo en local
- `helm get manifest` → Obtiene todos los manifiestos sobre un paquete ya instalado.
- `helm install RELEASE CHART --dry-run --debug` o `helm template --debug CHART` → Opciones similares para paquetes aún no instalados
- Prepara siempre que necesites un `values.yaml` con la configuración (los charts suelen ofrecer siempre un values.yaml por defecto)

Helm public repositories

https://artifacthub.io



Instalación de un Chart

- Podemos hacerlo teniendo el Chart en local o mediante un repositorio de Charts.
- Los charts se configuran mediante ficheros de "valores", que contienen parámetros de configuración y sus valores asociados.
- Hemos de documentarnos sobre las opciones de configuración / parámetros que soporta cada paquete.

```
$ helm install [RELEASE_NAME] [CHART] [flags]
$ helm install -f myvalues.yaml ./redis --name redis
$ helm install -f myvalues.yaml stable/redis --name redis
$ helm install --set port=6379 stable/redis --name redis
```

Listar releases (instalaciones realizadas)

- Usaremos `helm list`

```
$ helm list
$ helm list --namespace NAMESPACE
```

Actualizar una release

- Similar a instalar una release
- Podemos especificar nuevos values

```
$ helm upgrade RELEASE CHART
$ helm upgrade redis stable/redis -f myvalues.yaml

# Lo instala si no existe
$ helm upgrade --install redis stable/redis -f myvalues.yaml
```

Ver historial de una release

```
$ helm history RELEASE
```

Rollback de una release

- Primero debemos conocer a qué versión debemos hacer rollback con “helm history”

```
$ helm rollback RELEASE RELEASE_VERSION_NUMBER

$ helm rollback redis 4
```

Desinstalar una release

- Usaremos `helm uninstall release_name`

```
$ helm uninstall my-hello-world

release "my-hello-world" uninstalled
```

Creación de Charts

KEEPCODING
Tech School

Diseño y creación de charts

Estructura de un Chart:

- El directorio templates guarda los ficheros de templates.
- `values.yaml` tiene todos los "parámetros" / "opciones de configuración" posibles definidos, con sus valores por defecto.
- el fichero `Chart.yaml` define el chart y algunos parámetros.
- el directorio charts puede contener otros charts que pueden ser dependencias

```
mychart/
  Chart.yaml
  values.yaml
  charts/
  templates/
  ...
```

Mi primer chart

```
$ helm create mychart
```

- Nos creará un directorio `mychart`
- `NOTES.txt`, es un fichero que renderizará helm por pantalla cuando instalemos el chart
- `deployment.yaml`, ejemplo de template de deployment
- `service.yaml`, ejemplo de template de service
- `ingress.yaml`, ejemplo de template de ingress
- `_helpers.tpl`, es un fichero dónde podemos poner nuestras definiciones y funciones.
- `tests`, ejecuta tests para comprobar que funciona correctamente

Built-in objects

- **.Release**
  - .Relase.Name → **Nombre que le demos durante la instalación**
  - .Release.Time
  - .Release.Namespace
  - .Release.Revision
  - .Release.IsUpgrade
  - .Release.IsInstall
- **.Values**: Valores del fichero `values.yaml`
- **.Chart**: Todas las propiedades que declaremos en Chart.yaml son accesibles desde este objeto. También podemos definir **dependencias**
- Files: Nos permite acceder a recursos de tipo ficheros.

Built-in objects

- Template → **`helpers`**:
  - Define funciones auxiliares (devuelven YAML) que podemos usar:
    - '**chart-name.fullname**' → suele incluir el Release.Name con el nombre del chart y además control de tipo de caracteres y truncado.
  - Utilizaremos los helpers que vienen por defecto ya que siguen las best practices

- Para usar el 'fullname' en nuestros manifiestos tendremos que poner:

```
{{ include "mychart.fullname" . }}
```

- Si nuestro chart se llamara "flaskapp" y la instalación / release "prueba" el full name sería "flaskapp-prueba" (**muy útil para evitar conflictos de nombres**)

Built-in objects

- **`helpers`** creados por defecto (todos se referencian utilizando el nombre del chart).
  - `name` → como .Chart.Name pero verificado
  - `fullname` → release name + chart name verificados
  - `chart` → chart name + versión
  - `labels` → labels por defecto
  - `selectorLabels` → labels para selector de pods y pods

NOTES.txt

- NOTES.txt Se procesa tras la instalación para mostrar información al usuario. Es otro template, no es simple texto. Ejemplo:

```
Gracias por instalar {{ .Chart.Name }}.
La release se ha llamado {{ .Release.Name }}.
Para más detalles puedes ejecutar:
  $ helm status {{ .Release.Name }}
  $ helm get all {{ .Release.Name }}
  $ kubectl get pod
```

Fichero de values

- Debemos definir valores para usar dentro de los templates mediante el fichero de `values.yaml`
- Todo `'Value'` debe de estar definido previamente en el fichero de `values.yaml`
- Se pueden sobrescribir los valores por defecto en tiempo de instalación del chart mediante el parámetro “--set” o “-f” proporcionando un fichero de values adicional.
- Se pueden proporcionar tantos valores como se quieran, donde siempre prevalece el último.

Probando / compilando el Chart

- Para comprobar qué generará el chart:

```
$ helm template --debug CHART/

$ helm install XXX --dry-run --debug CHART/
```

Pruebas tras la instalación

- Los charts permiten la definición de jobs en el directorio "tests" de los templates. Para lanzarlos, una vez instalado el release:

```
$ helm test release-name
```

Instalando el Chart

- Instalación

```
$ helm install -f my-values.yaml nombre-release CHART/
```

Helm - Consejos prácticos

- Utiliza fullname (release name + chart name) donde sea más útil.
- Utiliza el helpers de labels y selectorLabes para los labels.
- Si tienes varios componentes (Deployments, StatefulSets, DaemonSets) que generan pods dentro de tu chart tendrás que añadir labels extra a mano además de usar los helpers de labels. Por supuesto controla los nombres de los recursos para que no solapen.
- Si utilizas dependencias tendrás que conocer y aprender bien el chart dependiente para poder integrarnos con él: utilizar su servicio, acceder a sus secrets, etc. Consejos dependencias
- Intenta hacer todo lo que puedas opcional y configurable, por ejemplo: autoescalado, reglas de afinidad, creación de Ingress, tipo de servicio, …

Helm - Consejos prácticos

- Un chart de helm debería poder ser instalado múltiples veces en el mismo namespace.
- No incluyas la información del "namespace" en tus manifiestos, deja que los usuarios instalen el chart donde consideren.

Helm: Funciones y Pipelines

KEEPCODING
Tech School

Funciones y Pipelines

En helm hay más de 60 funciones disponibles, podemos hacer uso de aquellas disponibles en Go template.

- https://helm.sh/docs/chart_template_guide/functions_and_pipelines/
- https://helm.sh/docs/chart_template_guide/function_list/
- https://banzaicloud.com/blog/creating-helm-charts-part-2/ → Buen tutorial.

Los pipes se pueden utilizar al estilo de Linux.

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

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: trendsetting-p-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
```

La función default

Nos permite declarar **valores por defecto**.
No abusar (los valores por defecto deberían estar en el values.yaml original del chart.
Útil para valores que necesiten de algún tipo de cálculo (por ejemplo un valor por defecto en función del nombre del release), ya que no se podría declarar en el values.yaml.

```
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 }}
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: virtuous-mink-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
```

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

Operadores y funciones

Los operadores son implementados a modo funcional y devuelven un valor booleano.

Para usar **“eq”, “ne”, “lt”, “gt”, “and” “or”, “not”**, etc, coloca un operador delante de la sentencia seguido de sus parámetros tal como lo harías en una función.

Puedes encadenar varias operaciones seguidas separando cada unas de las funciones individuales usando paréntesis.

```
{{/* include the body of this if statement when the variable .Values.fooString exists and is set to "foo"
*/}}
{{ if and .Values.fooString (eq .Values.fooString "foo") }}
    {{ ... }}
{{ end }}


{{/* include the body of this statement when the variable .Values.anUnsetVariable is set or
.values.aSetVariable is not set */}}
{{ if or .Values.anUnsetVariable (not .Values.aSetVariable) }}
   {{ ... }}
{{ end }}
```

Control de flujo de ejecución

Podemos usar estructuras de control para manejar el flujo de la generación de un template.

En helm podemos hacer uso de las siguientes estructuras:

- **if / else** para crear bloques condicionales
- **with**, para especificar un scope
- **range**, para iterar valores

Además podemos hacer uso de:

- **define**, que declara un nuevo named-template dentro de nuestro template
- **template (o include)**, permite importar un named template
- **block**, declara un área especial que permite ser rellenada.

if / else

La primera estructura de control que veremos es el típico if/else.
La estructura básica tiene la siguiente forma:

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

El pipeline será evaluado como false en los siguientes casos:

- un booleano falso
- un numérico cero
- una cadena vacía
- nil
- Una colección vacía (map, slice, tuple, dict, array)

La indentación importa

Hay que tener cuidado con los espacios en blanco y la indentación. Por ejemplo:

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

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

    mug: true
```

```
$ helm install --dry-run --debug ./mychart
Error: YAML parse error on mychart/templates/configmap.yaml: error converting YAML to JSON: yaml: line 9: did not find expected
key
```

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

¿Qué pasa con las líneas en blanco?

Espacios en blanco…

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

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: eyewitness-elk-configmap
data:
  food: "PIZZA"

   mug: true

  foo: bar
```

Solución OK:

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

# Results in:
food: "PIZZA"
mug: "true"
```

Solución NO-OK:

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

# Results in:
food: "PIZZA"mug: "true"
```

Modificación de scope con WITH

Cada vez que usemos una estructura de control entraremos en un scope restringido.

```
{{ with PIPELINE }}
  # restricted scope
{{ end }}
```

Podemos hacer:

```
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 }}
```

Cuidado con el scope restringido

No podremos acceder directamente a objetos que estén fuera del scope restringido.

Fallará:

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

OK:

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

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

Loops con RANGE

Teniendo el siguiente listado en nuestro values.yaml podemos:

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

```
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 }}
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: edgy-dragonfly-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
  toppings: |-
    - "Mushrooms"
    - "Cheese"
    - "Peppers"
    - "Onions"
```

Variables

Podemos definir variables, las variables podemos usarlas incluso dentro de scopes restringidos.

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

```
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 }}
```

Variables + range

Podemos definir variables a la hora de hacer uso de range

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

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

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

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

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: eager-rabbit-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"
```

Extras

Comentarios: {{/* ... */}}

Definición de named templates:

- **define** → define un nuevo template que no produce salida hasta que no se llame (necesitará scope si accede a objetos)
- template / **include** → llamada a un template, le pasamos un scope (por ejemplo ".")
    - include es una mejora de template, ya que permite integración con pipelines

Y mucho más…

- Acceso a ficheros
- Subcharts y global values

Helm - Práctica 2

- Analizaremos ahora el chart por defecto para comprobar si se entiende todo el código.

Ejercicios Helm

KEEPCODING
Tech School

Helm - 1

Diseña un chart con tu implementación anterior del stack de Wordpress para Kubernetes.
Intenta dotarlo de flexibilidad y configurabilidad en la medida de lo posible, por ejemplo:

- Tamaño de los PVCs configurable
- Storageclass configurable
- Tipo de Base de datos → MariaDB vs MySQL

Helm - 2

1. Instalar Wordpress mediante el chart de Bitnami
2. Explora los diferentes valores de configuración en su página oficial.
3. Actualiza la versión cambiando algún valor.
4. Haz un rollback a una versión anterior.
5. Analiza las diferencias entre el Wordpress de Bitnami y el tuyo :)

Helm - 3

1. Empaqueta en un Chart de helm la aplicación de flask-counter.

KEEPCODING

Tech School

Madrid | Barcelona | Bogotá