HELM

Helm

Helm is popular tool aims to simplify the **Kubernetes resources management**: it’s a template solution and acts more like a package manager, producing artifacts that are versionable, sharable, or deployable.
In this chapter, we’ll introduce Helm, a package manager for Kubernetes that helps install and manage Kubernetes applications using the Go template language in YAML files.
Helm has the concept of Chart: a packaged artifact that can be shared and contains multiple elements like dependencies on other Charts.

https://helm.sh/docs/intro/install/

Common actions for Helm:

helm **search**: search for charts
helm **pull**: download a chart to your local directory to view
helm **install**: upload the chart to Kubernetes
helm **list**: list releases of charts

Helm commands

BITNAMI https://github.com/bitnami/charts
PROMETHEUS https://github.com/prometheus-community/helm-charts
GRAFANA https://github.com/prometheus-community/helm-charts

1) ADD a CHART provider and update

```
helm repo add questdb https://helm.questdb.io/
helm repo update
```

2) CLONE a CHART

```
helm pull –untar questdb/questdb
```

3) RENDER the resources output

```
helm template ./questdb –values ./questdb/values.yaml
```

4) INSTALL a CHART

```
helm install my-questdb questdb/questdb –values values.yaml -n mynamespace
```

Charts structure

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

The templates/ directory is for **template** files. When Helm evaluates a chart, it will send all of the files in the templates/ directory through the template rendering engine. It then collects the results of those templates and sends them on to Kubernetes.

The values.yaml file is also important to templates. This file contains the **default values** for a chart. These values may be overridden by users during helm install or helm upgrade.

The Chart.yaml file contains a **description** of the chart. You can access it from within a template.

The charts/ directory may contain other charts (which we call **subcharts**). Later in this guide we will see how those work when it comes to template rendering.

Placeholders

```
apiVersion: v1
kind: Service
metadata:
labels:
app.kubernetes.io/name: {{ .Chart.Name }}       ← Use the informations defined in
name: {{ .Chart.Name }}                         Chart.yaml file in placeholders
spec:
ports:
- name: http
port: {{ .Values.image.containerPort }}
targetPort: {{ .Values.image.containerPort }}    ← Externalize in a values file some properties
selector:
app.kubernetes.io/name: {{ .Chart.Name }}
```

Template syntax

Helm lets you create a _**helpers.tpl** file in the templates directory defining statements that can be called in templates to avoid this problem.

1. Define template

```
{{- define "app.selectorLabels" -}}
app.kubernetes.io/name: {{ .Chart.Name}}
app.kubernetes.io/version: {{ .Chart.AppVersion}}
{{- end }}
```

2. Include it

```
{{- include "app.selectorLabels" . | nindent 6 }}
```

Template syntax

helm lint : is your go-to tool for verifying that your chart follows best practices

helm template –debug : will test rendering chart templates locally.

helm install --dry-run –debug : It's a great way to have the server render your templates, then return the resulting manifest file.

helm get manifest : This is a good way to see what templates are installed on the server.

helm get values myrelease: This is a good way to see what values are used by an installed chart.

Template syntax

Built in: {{ .Release.Name }}

Files: This provides access to all non-special files in a chart. While you cannot use it to access templates, you can use it to access other files in the chart.

Functions: with, indent, quote , default, empty, first, concat…
Math functions
Conditionals: and / or /gt /lt ...
Flow control: if/else
Looping: range

Local variables vs global variables ($)

Workshop

1) FOLLOW THE TUTORIAL
https://github.com/ynov-campus-sophia/conteneurisation/tree/master/helm