Helm


HELM

What is Helm?

Helm is a tool for managing Kubernetes charts.

Charts are packages of pre-configured resources.

Think of it like apt/yum/homebrew for Kubernetes.

Why Helm?

- De-facto standard for templating k8s configs
- Official Kubernetes project maintained by CNCF
- Repeatable application installations
- Painless updates
- Ships with ready to use charts made by the community
- best practices are baked into offical charts

What's wrong with kubectl?

- Resources need to be mutated after a deploy
- You can `kubectl edit <resource>`
- But, need to implement your own updating and rollback or depend on vendor specific tooling for lifecycle management

Helm helps with lifecycle managment while allowing you to define your infrastructure as code

Helm Basics

Architecture

- helm client
- tiller server-side component

Demo: Quickstart

```
# deploy tiller to kube-system namespace
helm init
# Deploy a postgresql instance
helm install stable/postgresql
```

Inspecting helm managed resources

```
# Show deploys
helm list
# Get information on a release
helm status my-release
```

Helm releases

Differentiate multiple deploys of a chart with releases.

Possible values could be.

- `production`, `stage`, `integration`
- `<customer>-production`
- `<app>-postgresql`

This depends on your organizations structure and what you use the cluster for.

What did we just deploy?

- Deployment based on `postgres` image
- NetworkPolicy allowing access.
- PersistentVolumeClaim for storing data
- Secret containing autogenerated postgres-password

- Service to expose database in cluster

Adding metrics

```
helm upgrade my-release stable/postgresql \
    --set 'metrics.enabled=true'
```

We just deployed a postgres_exporter sidecar container exposing metrics for Prometheus!

Rollback

```
# simulate rollback
helm rollback --dry-run my-release old-version
# actual rollback
helm rollback my-release old-version
```

Writing helm charts

```
helm create my-app
```

```
my-app/
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── ingress.yaml
│   ├── NOTES.txt
│   └── service.yaml
└── values.yaml
```

Templating

The Helm template language is implemented in the strongly typed Go language.

Each Helm chart contains a templates directory that contains relevant templates.

Naming Templates

- Use `.yaml` or `.tpl` suffix for files
- Dasherize file names
- Reflect resource kind in names

Good	Bad
`foo-pod.yaml`	`foo-pod.yml`
`my-example-podtemplates.yaml`	`MyExamplePodTemplates.yaml`
`my-example-svc.yaml`	`my-example.yaml`

Built-in Objects

- Everything below `Release`
- `Chart` variable with infos from `Chart.yaml`
- `Values` with data from `values.yaml`, the cli and other sources

A complete list is in the Helm docs

```
# Current release
{{ .Release.Revision }}
# Chart infos
{{ .Chart.version }}
```

Values

The built-in object Values is empty by default. Charts and users can add to it through `values.yaml`, user-supplied files and on the CLI.

Let's see how values get populated with a simple template.

```
echo '{{ .Values.hello.world }}' > templates/hello.tpl
```

Default values are in a the charts values.yaml

```
# values.yaml
hello:
  world: Hello!
```

```
helm template . -x templates/hello.yaml
```

```
---
# Source: my-chart/templates/hello.yaml
Hello!
```

Overriding values on the CLI

```
helm template . --execute templates/hello.yaml \
  --set 'hello.world=Hallo!'
```

```
---
# Source: my-chart/templates/hello.yaml
Hallo!
```

Locally overriding values with a specific YAML file

```
# local.yaml
hello:
  world: Здравствуй!
```

```
helm template . --execute templates/hello.yaml \
    --values local.yaml
```

```
---
# Source: my-chart/templates/hello.yaml
Здравствуй!
```

Functions and Pipelines

Helm has over 60 available functions. Some of them are from Go template, some from Sprig template library.

```
echo  'var: {{ quote .Values.myvar }}' > templates/quote.yaml
```

```
helm template . --execute templates/quote.yaml \
   --set 'myvar=This is a string'
```

```
---
# Source: my-chart/templates/quote.yaml
var: "This is a a string"
```

Flow Control

- `if` / `else` for creating conditional blocks
- `with` to specify a scope
- `range`, which provides a "for each"-style loop


```
{{- if .Values.ingress.enabled -}}
apiVersion: extensions/v1beta1
kind: Ingress
# ...
{{- end -}}
```

range

```
# ...
spec:
  rules:
  {{- range .Values.server.ingress.hosts }}
    - host: {{ . }}
  {{- end -}}
# ...
```

Debugging Templates

```
# verify that chart follows best practices
helm lint
# let server render templates and return resulting manifest
helm install --dry-run --debug .
# See what templates are installed on the server
helm get manifest my-release
```

Named Templates

- `define` declares a new named template inside of your template
- `template` imports a named template
- `block` declares a special kind of fillable template area

Templates are usually defined in `templates/_helpers.tpl`

```
{{/*
Expand the name of the chart.
*/}}
{{- define "my-chart.name" -}}
{{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-" -}}
{{- end -}}
```

```
{{/*
Create a default fully qualified app name.
We truncate at 63 chars because some Kubernetes name fields are limited to this (by the DNS naming spec).
*/}}
{{- define "my-chart.fullname" -}}
{{- $name := default .Chart.Name .Values.nameOverride -}}
{{- printf "%s-%s" .Release.Name $name | trunc 63 | trimSuffix "-" -}}
{{- end -}}
```

```
metadata:
  name: {{ template "my-chart.fullname" . }}
  labels:
    app: {{ template "my-chart.name" . }}
```

```
metadata:
  name: my-release-my-chart
  labels:
    app: mychart
```

Demo: Let's write some infra

In this example we will create a chart for glances.

Glances can be scheduled on plain docker as follows:

```
docker run \
  --rm \
  --detach \
  --publish 61208-61209:61208-61209 \
  --env GLANCES_OPT="-w" \
  --volume /var/run/docker.sock:/var/run/docker.sock:ro \
  --pid host \
  docker.io/nicolargo/glances
```

Let's look at it on localhost to see what is does.

```
helm create glances
cd glances
```

For this example we make some changes to the default chart generated by helm.

values.yaml

```
image:
  # path to docker hub container
  repository: nicolargo/glances
service:
  # match ports to EXPOSE from image
  externalPort: 61208
  internalPort: 61208
```

templates/deployment.yaml

```
spec:
  template:
    spec:
      containers:
        - name: {{ .Chart.Name }}
          # add environment to container
          env:
            - name: GLANCES_OPT
              value: -w
```

template/NOTES.tpl

```
# change source port for port-forward example
echo "Visit http://127.0.0.1:{{ .Values.service.internalPort }} to use your application"
kubectl port-forward $POD_NAME {{ .Values.service.internalPort }}:{{ .Values.service.internalPort }}
```

deploy to k8s

```
helm install . --name glances-test
```

Best Practices

- Use SemVer 2 to represent version number.
- Indent yaml with 2 spaces (and never tabs).
- Specify a tillerVersion SemVer contraint in you chart.

```
tillerVersion: ">=2.4.0"
```

- use labels so k8s can identify ressources and to expose operators for the purpose of querying

The offical best practices guide has more pointers you should follow

Advanced Helming

Composing systems with subcharts

- Charts can depend on other charts.
- Dependencies are described in `requirements.yaml`.
- `helm dep build` creates `requirements.lock`.

```
# download dependencies
helm dep build stable/redmine

# install redmine with postgresql
helm install stable/redmine \
    --set databaseType.mariadb=false,databaseType.postgresql=true
```

Subcharts and Values

- subcharts cannot depend on their parent charts values.
- parent charts can override values for subcharts.
- global values can be accessed from any chart.

```
# override postgresql values
postgresql:
  postgresPassword: muchsecretverysecure

# define some global variables
global:
  myVariable: myValue
```

Please plan ahead when using globals or don't use them at all. Official charts rarely use them.

Managing vendor specific resources

Helm can be used to manage any resources that are available through a k8s style API endpoint.

```
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
  name: {{ .Values.name | quote }}
  annotations:
    description: Defines how to deploy the application server
    template.alpha.openshift.io/wait-for-ready: 'true'
spec:
# ...
```

More infos are on the OpenShift blog.