The missing CI/CD Kubernetes component: Helm package manager

The past two months I have been actively migrating all my services from dedicated servers to Kubernetes cluster. I have done the containerisation of the existing applications and have written the Kubernetes manifests. However, I was missing a way to configure, release, version, rollback and inspect the deployments. This is when I have discovered **Helm**.

Lets keep things simple.

Key definitions:

- **Helm** is a *chart* manager.
- **Charts** are packages of pre-configured Kubernetes resources.
- **Release** is a collection of Kubernetes resources deployed to the cluster using Helm.

Helm is used to:

- Make configurable releases
- Upgrade, delete, inspect releases made using Helm

Helm is made of two components:

- helm client. Used to create, fetch, search and validate charts and to instruct *tiller*.
- *tiller* server. Runs inside the Kubernetes cluster and manages the releases.

Install Helm

macOS users can install Helm client using Homebrew.

brew install kubernetes-helm

Use helm client to deploy tiller to the Kubernetes cluster:
helm init *uses the* ~/.kube/config *configuration to connect to the Kubernetes cluster. Ensure that your configuration is referencing a cluster that is safe to make test deployments.*

helm init

Instructions for other platforms, refer to

https://github.com/sunilake/helm.git

Define a Chart

Helm documentation includes a succinct guide to defining Helm charts. I recommend reading the guide in full. However, I will guide you through a "Hello, World!" example.

A chart is organized as a collection of files inside of a directory. The directory name is the name of the chart.

```
$ mkdir ./hello-world
$ cd ./hello-world
```

A chart must include a chart definition file, `Chart.yaml`. Chart definition file must define two properties: `name` and `version` (Semantic Versioning 2).

```
$ cat <<'EOF' > ./Chart.yaml
name: hello-world
version: 1.0.0
EOF
```

A chart must define templates used to generate Kubernetes manifests, e.g.

```
$ mkdir ./templates
$ cat <<'EOF' > ./templates/deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-world
```

```
spec:
  replicas: 1
  template:
   metadata:
     labels:
       app: hello-world
    spec:
     containers:
       - name: hello-world
         image: gcr.io/google-samples/node-hello:1.0
         ports:
           - containerPort: 8080
             protocol: TCP
EOF
```

```
$ cat <<'EOF' > ./templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-world
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 8080
    protocol: TCP
  selector:
    app: hello-world
EOF
```

That is all thats required to make a release.

Release, list, inspect, delete, rollback, purge

Use helm install RELATIVE_PATH_TO_CHART to make a release.

$ helm install . (NOTE There is a G+DOT at the end)

NAME: cautious-shrimp
LAST DEPLOYED: Thu Jan 5 11:32:04 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
hello-world 10.0.0.175 <nodes> 8080:31419/TCP 0s

==> extensions/Deployment
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
hello-world 1 1 1 0 0s

helm install . used Kubernetes manifests in ./templates directory to create a deployment and a service:

$ helm ls
NAME REVISION UPDATED STATUS CHART

cautious-shrimp 1 Thu Jan 5 11:32:04 2017 DEPLOYED hello -world-1.0.0

Use helm status RELEASE_NAME to inspect a particular release:

$ helm status cautious-shrimp
LAST DEPLOYED: Thu Jan 5 11:32:04 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
hello-world 10.0.0.175 <nodes> 8080:31419/TCP 6m

==> extensions/Deployment
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
hello-world 1 1 1 1 6m

Use helm delete RELEASE_NAME to remove all Kubernetes resources associated with the release:

$ helm delete cautious-shrimp

Use helm ls --deleted to list deleted releases:

$ helm ls --deleted
NAME REVISION UPDATED STATUS CHART
cautious-shrimp 1 Thu Jan 5 11:32:04 2017 DELETED hello -world-1.0.0

Use helm rollback RELEASE_NAME REVISION_NUMBER to restore a deleted release:

$ helm rollback cautious-shrimp 1
Rollback was a success! Happy Helming!
$ helm ls
NAME REVISION UPDA TED STATUS CHART
cautious-shrimp 2 Thu Jan 5 11:47:57 2017 DEPLOYED hello-world-1.0.0
$ helm status cautious-shrimp
LAST DEPLOYED: Thu Jan 5 11:47:57 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
hello-world 10.0.0.42 <nodes> 8080:32367/TCP 2m

==> extensions/Deployment
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE

```
AGE
hello-world  1       1       1         1         2m
```

Use helm delete --purge RELEASE_NAME to remove all Kubernetes resources associated with with the release and all records about the release from the store.

```
$ helm delete --purge cautious-shrimp
$ helm ls –deleted
```

Configuring releases

I have shown you how to manage a life cycle of a Helm release. That was cool. However, that is not even the reason why I started looking into Helm: I needed a tool to configure releases.

Helm Chart templates are written in the Go template language, with the addition of 50 or so add-on template functions from the Sprig library and a few other specialized functions. Values for the templates are supplied using values.yaml file, e.g.

```
$ cat <<'EOF' > ./values.yaml
image:
repository: gcr.io/google-samples/node-hello
tag: '1.0'
EOF
```

A generic warning about YAML configuration. Be careful to quote numeric values, e.g. if in the above example I have written `tag: 1.0` *(without quotes) it would have been interpreted as a number, i.e.* `1` .

Values that are supplied via a `values.yaml` file are accessible from the `.Values` object in a template.

```
$ cat <<'EOF' > ./templates/deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
name: hello-world
spec:
replicas: 1
template:
metadata:
labels:
app: hello-world
spec:
containers:
- name: hello-world
image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
ports:
- containerPort: 8080
protocol: TCP
EOF
```

Values in the `values.yaml` can be overwritten at the time of making the release using `--values YAML_FILE_PATH` or `--set key1=value1,key2=value2` parameters, e.g.

```
$ helm install --set image.tag='latest' .
```

Values provided using `--values` parameter are merged with values defined in `values.yaml` file and values provided using `--set` parameter are merged with the resulting values, i.e. `--set` overwrites provided values of `--value`, `--value` overwrites provided values of `values.yaml` .

Debugging

Using templates to generate the manifests require to be able to preview the result. Use `--dry-run --debug` options to print the values and the resulting manifests withou t deploying the release:

```
$ helm install . --dry-run --debug --set image.tag=latest
Created tunnel using local port: '49636'
SERVER: "localhost:49636"
CHART PATH: /Users/gajus/Documents/dev/gajus/hello -world
NAME:   eyewitness-turkey
REVISION: 1
RELEASED: Thu Jan  5 13:02:49 2017
CHART: hello-world-1.0.0
USER-SUPPLIED VALUES:
image:
  tag: latest
```

COMPUTED VALUES:
image:
repository: gcr.io/google-samples/node-hello
tag: latest

HOOKS:
MANIFEST:

 # Source: hello-world/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
name: hello-world
spec:
type: NodePort
ports:
- port: 8080
targetPort: 8080
protocol: TCP
selector:
app: hello-world

 # Source: hello-world/templates/deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
name: hello-world
spec:
replicas: 1

```
template:
  metadata:
    labels:
      app: hello-world
  spec:
    containers:
      - name: hello-world
        image: gcr.io/google-samples/node-hello:latest
        ports:
          - containerPort: 8080
            protocol: TCP
```

Using predefined values

In addition to user supplied values, there is a number of predefined values:

- .Release is used to refer to the resulting release values, e.g. .Release.Name, .Release.Time .
- .Chart is used to refer to the Chart.yaml configuration.
- .Files is used to refer to the files in the chart directory.

I am using the predefined values to create labels that enable inspection of the Kubernetes resources:

```
$ cat <<'EOF' > ./templates/deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
name: {{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63
}}
```

```
labels:
app: {{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63 }}
version: {{ .Chart.Version }}
release: {{ .Release.Name }}
spec:
replicas: 1
template:
metadata:
labels:
app: {{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63 }}
version: {{ .Chart.Version }}
release: {{ .Release.Name }}
spec:
containers:
- name: hello-world
image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
ports:
- containerPort: 8080
protocol: TCP
EOF
$ cat <<'EOF' > ./templates/service.yaml
apiVersion: v1
kind: Service
metadata:
name: {{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63
}}
labels:
app: {{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63 }}
version: {{ .Chart.Version }}
release: {{ .Release.Name }}
spec:
type: NodePort
ports:
- port: 8080
```

```
targetPort: 8080
protocol: TCP
selector:
app: {{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63 }}
EOF
```

Note, that it is your responsibility to ensure that all resources have a unique name and labels. In the above example, I am using .Release.Name and .Chart.Name to create a resource name.

```
{{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63 }}
```

Note: Kubernetes resource labels and names are restricted to 63 characters. http://kubernetes.io/docs/user-guide/labels/#syntax-and-character-set

Partials

You have probably noticed a repeating pattern in the templates:

```
app: {{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63 }}
version: {{ .Chart.Version }}
release: {{ .Release.Name }}
```

We are using the same labels for all resources.

Furthermore, our resource name is a lengthy expression:

{{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63 }}

This repetition becomes even more apparent in large applications, with dozens of different resources. Lets see what we can do about it.

Lets create a `_helpers.tpl` file and use it to declare partials that we can later import into the templates:

Files in `./templates` directory that start with `_` are not considered Kubernetes manifests. The rendered version of these files are not sent to Kubernetes.

```
$ cat <<'EOF' > ./templates/_helpers.tpl
{{- define "hello-world.release_labels" }}
app: {{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63 }}
version: {{ .Chart.Version }}
release: {{ .Release.Name }}
{{- end }}

{{- define "hello-world.full_name" -}}
{{- printf "%s-%s" .Release.Name .Chart.Name | trunc 63  -}}
{{- end -}}
EOF
```

Now we have two partials: `hello-world.release_labels` and `hello-world.full_name` that we can use in the templates:

Template names are global. *Because templates in subcharts are compiled together with top -level templates, you should be careful to name your templates with chart -specific names. This is the reason I am using Chart name to prefix template names.*

```
$ cat <<'EOF' > ./templates/deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
name: {{ template "hello-world.full_name" . }}
labels:
{{- include "hello-world.release_labels" . | indent 4 }}
spec:
replicas: 1
template:
metadata:
labels:
{{- include "hello-world.release_labels" . | indent 8 }}
spec:
containers:
- name: hello-world
image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
ports:
- containerPort: 8080
protocol: TCP
EOF
$ cat <<'EOF' > ./templates/service.yaml
apiVersion: v1
kind: Service
```

```
metadata:
name: {{ template "hello-world.full_name" . }}
labels:
{{- include "hello-world.release_labels" . | indent 4 }}
spec:
type: NodePort
ports:
- port: 8080
targetPort: 8080
protocol: TCP
selector:
app: {{ template "hello-world.full_name" . }}
EOF
```

Notice that I sometimes use template *and sometimes* include *to include a partial. Read the named templates guide to learn the difference.*

One last time, lets print the resulting manifest:

```
$ helm install . --dry-run --debug
Created tunnel using local port: '50655'
SERVER: "localhost:50655"
CHART PATH: /Users/gajus/Documents/dev/gajus/hello-world
NAME:   kindred-deer
REVISION: 1
RELEASED: Thu Jan  5 14:46:41 2017
CHART: hello-world-1.0.0
USER-SUPPLIED VALUES:
{}
```

```
COMPUTED VALUES:
image:
  repository: gcr.io/google-samples/node-hello
  tag: "1.0"


HOOKS:
MANIFEST:
---
# Source: hello-world/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: kindred-deer-hello-world
  labels:
    app: kindred-deer-hello-world
    version: 1.0.0
    release: kindred-deer
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 8080
    protocol: TCP
  selector:
    app: kindred-deer-hello-world
---
# Source: hello-world/templates/deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: kindred-deer-hello-world
  labels:
    app: kindred-deer-hello-world
    version: 1.0.0
    release: kindred-deer
spec:
```

```
replicas: 1
template:
 metadata:
  labels:
   app: kindred-deer-hello-world
   version: 1.0.0
   release: kindred-deer
 spec:
  containers:
   - name: hello-world
    image: gcr.io/google-samples/node-hello:1.0
    ports:
     - containerPort: 8080
      protocol: TCP
```

Bonus round: generating a checksum

Have you noticed that I am using separate files for deployment and service resource definition? Helm does not require this. However, there is a good reason for keeping definitions separate.

Consider a scenario: You have a ConfigMap resource that controls behaviour of your deployment, e.g.

```
$ cat <<'EOF' > ./templates/config-map.yaml
apiVersion: v1
kind: ConfigMap
metadata:
name: {{ template "hello-world.full_name" . }}
labels:
{{- include "hello-world.release_labels" . | indent 4 }}
```

```
data:
magic-number: '11'
EOF
$ cat <<'EOF' > ./templates/deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
name: {{ template "hello-world.full_name" . }}
labels:
{{- include "hello-world.release_labels" . | indent 4 }}
spec:
replicas: 1
template:
metadata:
labels:
{{- include "hello-world.release_labels" . | indent 8 }}
spec:
containers:
- name: hello-world
image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
ports:
- containerPort: 8080
protocol: TCP
env:
- name: MAGIC_NUMBER
valueFrom:
configMapKeyRef:
name: {{ template "hello-world.full_name" . }}
key: magic-number
EOF
$ helm install . --name demo
$ kubectl get po,svc,cm -l app=demo-hello-world
NAME READY STATUS RESTARTS AGE
po/demo-hello-world-2159237003-fr5gk 1/1 Running 0 8s
```

```
NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
svc/demo-hello-world 10.0.0.68 <nodes> 8080/TCP 8s
NAME DATA AGE
cm/demo-hello-world 1 8s
```

You've made a release.

Now `magic-number` has changed. You update the `config-map.yaml` and make a new release:

```
$ cat <<'EOF' > ./templates/config -map.yaml
apiVersion: v1
kind: ConfigMap
metadata:
name: {{ template "hello-world.full_name" . }}
labels:
{{- include "hello-world.release_labels" . | indent 4 }}
data:
magic-number: '12'
EOF
$ helm upgrade demo .
```

Notice that `ConfigMap` resource has been recreated but `Deployment` has not been recreated.

```
$ kubectl get po,svc,cm -l app=demo-hello-world
NAME READY STATUS RESTARTS AGE
po/demo-hello-world-2159237003-fr5gk 1/1 Running 0 1m
NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
```

```
svc/demo-hello-world 10.0.0.68 <nodes> 8080/TCP 1m
NAME DATA AGE
cm/demo-hello-world 1 1m
```

This is because nothing in the `deployment.yaml` has changed. However, we can use `include` function to include the contents of the `config-map.yaml` and `sha256sum` to create SHA256 checksum.

```
{{ include (print $.Chart.Name "/templates/config -map.yaml")
. | sha256sum }}
```

We can use this value to generate an annotation for the deployment:

```
$ cat <<'EOF' > ./templates/deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
name: {{ template "hello-world.full_name" . }}
labels:
{{- include "hello-world.release_labels" . | indent 4 }}
spec:
replicas: 1
template:
metadata:
labels:
{{- include "hello-world.release_labels" . | indent 8 }}
annotations:
checksum/config-map: {{ include (print $.Chart.Name
"/templates/config-map.yaml") . | sha256sum }}
spec:
```

```
containers:
- name: hello-world
image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
ports:
- containerPort: 8080
protocol: TCP
env:
- name: MAGIC_NUMBER
valueFrom:
configMapKeyRef:
name: {{ template "hello-world.full_name" . }}
key: magic-number
EOF
```

Now, we change magic-number value again and upgrade the release:

```
$ cat <<'EOF' > ./templates/config-map.yaml
apiVersion: v1
kind: ConfigMap
metadata:
name: {{ template "hello-world.full_name" . }}
labels:
{{- include "hello-world.release_labels" . | i ndent 4 }}
data:
magic-number: '13'
EOF
$ helm upgrade demo .
$ kubectl get po,svc,cm -l app=demo-hello-world
NAME READY STATUS RESTARTS AGE
po/demo-hello-world-2130866520-l77rf 1/1 Terminating 0
58s
po/demo-hello-world-3814420630-jrglp 0/1
ContainerCreating 0 2s
```

NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
svc/demo-hello-world 10.0.0.68 <nodes> 8080/TCP 4m
NAME DATA AGE
cm/demo-hello-world 1 4m

The checksum has changed, the release annotation has changed, and therefore the associated pods have been recreated.

My use case for Helm has been to abstract configuration and deployments in the CI/CD pipeline. However, I have learned along the way that Helm is much more versatile tool. I have not even touched on the aspect of using Helm to declare dependencies and setup Chart repositories. Therefore, I encourage the reader to continue exploring Helm by going through the documentation.