Experiment XX: Building a Helm Chart

In this experiment, we'll be walking through using Helm with k3d, with a single node and single server testing environment for Kubernetes. We will make a small Nginx web server application. To get set up, do the followiget set up, do the following.

- We have been using k3d, but for installation we can go to k3d.io
- We already have helm installed, but if not download and configure Helm using your favorite package manager listed here or manually from the releases.

Create a Helm chart

Start by confirming we have the prerequisites installed:

$ which helm *## this can be in any folder as long as it returns in the path*
/usr/local/bin/helm
$ C:\k3d> **k3d cluster create demo --servers 1 --agents 1**

[36mINFO[0m[0000] Created network 'k3d-demo'
[36mINFO[0m[0000] Created volume 'k3d-demo-images'
[36mINFO[0m[0000] Creating initializing server node
[36mINFO[0m[0000] Creating node 'k3d-demo-server-0'
[36mINFO[0m[0001] Pulling image 'docker.io/rancher/k3s:v1.18.6-k3s1'
[36mINFO[0m[0091] Creating node 'k3d-demo-agent-0'
[36mINFO[0m[0096] Creating LoadBalancer 'k3d-demo-serverlb'
[36mINFO[0m[0097] Pulling image 'docker.io/rancher/k3d-proxy:v3.0.1'
[36mINFO[0m[0158] Cluster 'demo' created successfully!
[36mINFO[0m[0158] You can now use it like this:
kubectl cluster-info

In our example, you'll see that we've setup 1 server (Kubernetes masters) in our control plane, and 1 agent (Kubernetes nodes) in our data plane.

You'll also see that we have the Load Balancer, k3d-demo-serverlb, which is our containerized Traefik instance running in our cluster.

Kubectl won't know about this cluster until we load and set our KUBECONFIG environment variable.

C:\k3d> **k3d cluster list**

NAME SERVERS AGENTS LOADBALANCER
demo 1/3   2/3   true

C:\k3d> **k3d kubeconfig get demo**

---

apiVersion: v1

```
clusters:
- cluster:
    certificate-authority-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUJWekNCL3FBREFnRUNBZ0VBTUFvR
0NDcUdTTTQ5QkFNQ01DTXhJVEFmQmdOVkJBTU1HR3N6Y3kxelpYSjIKWlhJdFkyRkFNVF
U1T1RZM01URTJOakFlRncweU1EQTVNRGt4TnpBMk1EWmFGdzB6TURBNU1EY3hOekEyT
URaYQpNQ014SVRBZkJnTlZCQU1NR0dzemN5MXpaWEoyWlhJdFkyRkFNVFU1T1RZM01U
RTJOakJaTUJNR0J5cUdTTTQ5CkFnRUdDQ3FHU000OUF3RUhBMElBQkR4cWlSWnI2cVUy
R25GYjQ1UjdTU2ljVmdFSC9RNEY3V3dBTkQxdU9uazUKOFIwVGVNRUh1eTYwN0ZXeWlqaz
VkeFJ3WjBOaUlybjcrSW1EOUVia2FmaWpJekFoTUE0R0ExVWREd0VCL3dRRQpBd0lDcERB
UEJnTlZIUk1CQWY4RUJUQURBUUgvTUFvR0NDcUdTTTQ5QkFNQ0EwZ0FNRVVDSUU2Nn
FaRkVuZ1BuClN3TmE2bU1wN1ZKd1UvN2FValdGM0s0Z1o1OWhzd29CQWlFQTgxY241UjA2
RTEzYndQdXJORjlMTlZXL0l5UzMKeEFEK1EyM2QwVUMvYk1nPQotLS0tLUVORCBDRVJUS
UZJQ0FURS0tLS0tCg==
    server: https://0.0.0.0:53948
  name: k3d-demo
contexts:
- context:
    cluster: k3d-demo
    user: admin@k3d-demo
  name: k3d-demo
current-context: k3d-demo
kind: Config
preferences: {}
users:
- name: admin@k3d-demo
  user:
    password: 6a4ad9aadd405b3dcffc77b5f12c46d5
    username: admin
```

C:\k3d> **k3d node list**

NAME	ROLE	CLUSTER	STATUS
k3d-demo-agent-0	agent	demo	running
k3d-demo-agent-1	agent	demo	running
k3d-demo-agent-2	agent	demo	exited
k3d-demo-server-0	server	demo	exited
k3d-demo-server-1	server	demo	exited
k3d-demo-server-2	server	demo	running
k3d-demo-serverlb	loadbalancer	demo	running

C:\k3d> **mkdir .kube**

C:\k3d> **cd .kube**

On Windows:

C:\k3d> **set KUBECONFIG_FILE=C:\k3d\.kube\demo**

C:\k3d> **k3d kubeconfig get demo > %KUBECONFIG_FILE%**

C:\k3d> **set KUBECONFIG=%KUBECONFIG_FILE%**

On MacOS or Linux

~/k3d/.kube $ **export KUBECONFIG_FILE=~/.kube/demo**

~/k3d/.kube $ **k3d kubeconfig get demo > $KUBECONFIG_FILE**

~/k3d/.kube $ **export KUBECONFIG=$KUBECONFIG_FILE**

Starting a new Helm chart requires one simple command:

```
$ helm create mychartname
```

For the purposes of this tutorial, name the chart **buildachart**:

```
$ helm create helmchartbuilder
Creating helmchartbuilder
$ ls helmchartbuilder/
Chart.yaml   charts/     templates/   values.yaml
```

Examine the chart's structure

Now that you have created the chart, take a look at its structure to see what's inside. The first two files you see—**Chart.yaml** and **values.yaml**—define what the chart is and what values will be in it at deployment.

Look at **Chart.yaml**, and you can see the outline of a Helm chart's structure:

```
apiVersion: v2
name: helmchartbuilder
description: A Helm chart for Kubernetes

# A chart can be either an 'application' or a 'library' chart.
#
# Application charts are a collection of templates that can be packaged into versioned archives
# to be deployed.
#
# Library charts provide useful utilities or functions for the chart developer. They're
```

```
included as
# a dependency of application charts to inject those utilities and functions into the rendering
# pipeline. Library charts do not define any templates and therefore cannot be deployed.
type: application

# This is the chart version. This version number should be incremented each time you make changes
# to the chart and its templates, including the app version.
version: 0.1.0

# This is the version number of the application being deployed. This version number should be
# incremented each time you make changes to the application.
appVersion: 1.16.0
```

The first part includes the API version that the chart is using (this is required), the name of the chart, and a description of the chart. The next section describes the type of chart (an application by default), the version of the chart you will deploy, and the application version (which should be incremented as you make changes).

The most important part of the chart is the template directory. It holds all the configurations for your application that will be deployed into the cluster. As you can see below, this application has a basic deployment, ingress, service account, and service. This directory also includes a test directory, which includes a test for a connection into the app. Each of these application features has its own template files under **templates/**:

```
$ ls templates/
NOTES.txt      _helpers.tpl    deployment.yaml    ingress.yaml    service.yaml
serviceaccount.yaml  tests/
```

There is another directory, called **charts**, which is empty. It allows you to add dependent charts that are needed to deploy your application. Some Helm charts for applications have up to four extra charts that need to be deployed with the main application. When this happens, the **values** file is updated with the values for each chart so that the applications will be configured and deployed at the same time. This is a far more advanced configuration (which I will not cover in this introductory article), so leave the **charts/** folder empty.

Understand and edit values

Template files are set up with formatting that collects deployment information from the **values.yaml** file. Therefore, to customize your Helm chart, you need to edit the values file. By default, the **values.yaml** file looks like:

```
# Default values for helmchartbuilder.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.
```

```yaml
replicaCount: 1

image:
  repository: nginx
  pullPolicy: IfNotPresent

imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""

serviceAccount:
 # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using the fullname template
  name:

podSecurityContext: {}
  # fsGroup: 2000

securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000

service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
  annotations: {}
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  hosts:
    - host: chart-example.local
      paths: []
  tls: []
  #  - secretName: chart-example-tls
  #    hosts:
  #      - chart-example.local

resources: {}
  # We usually recommend not to specify default resources and to leave this as a
conscious
  # choice for the user. This also increases chances charts run on environments with little
```

```
  # resources, such as Minikube. If you do want to specify resources, uncomment the
following
  # lines, adjust them as necessary, and remove the curly braces after 'resources:'.
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
  #   cpu: 100m
  #   memory: 128Mi

nodeSelector: {}

tolerations: []

affinity: {}
```

Basic configurations

Beginning at the top, you can see that the **replicaCount** is automatically set to 1, which means that only one pod will come up. You only need one pod for this example, but you can see how easy it is to tell Kubernetes to run multiple pods for redundancy.

The **image** section has two things you need to look at: the **repository** where you are pulling your image and the **pullPolicy**. The pullPolicy is set to **IfNotPresent**; which means that the image will download a new version of the image if one does not already exist in the cluster. There are two other options for this: **Always**, which means it will pull the image on every deployment or restart (I always suggest this in case of image failure), and **Latest**, which will always pull the most up-to-date version of the image available. Latest can be useful if you trust your image repository to be compatible with your deployment environment, but that's not always the case.

Change the value to **Always**.

Before:

```
image:
  repository: nginx
  pullPolicy: IfNotPresent
```

After:

```
image:
  repository: nginx
  pullPolicy: Always
```

Naming and secrets

Next, take a look at the overrides in the chart. The first override is **imagePullSecrets**, which is a setting to pull a secret, such as a password or an API key you've generated as credentials for a private registry. Next are **nameOverride** and **fullnameOverride**. From the moment you ran **helm create**, its name (helmchartbuilder) was added to a number of configuration files—from the YAML ones above to the **templates/helper.tpl** file. If you need to rename a chart after you create it, this section is the best place to do it, so you don't miss any configuration files.

Change the chart's name using overrides.

Before:

```
imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""
```

After:

```
imagePullSecrets: []
nameOverride: "nginx-awesome-app"
fullnameOverride: "nginx-chart"
```

Accounts

Service accounts provide a user identity to run in the pod inside the cluster. If it's left blank, the name will be generated based on the full name using the **helpers.tpl** file. I recommend always having a service account set up so that the application will be directly associated with a user that is controlled in the chart.

As an administrator, if you use the default service accounts, you will have either too few permissions or too many permissions, so change this.

Before:

```
serviceAccount:
 # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using the fullname template
  Name:
```

After:

```
serviceAccount:
 # Specifies whether a service account should be created
```

```
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using the fullname template
  Name: nginxrocks
```

Security

You can configure pod security to set limits on what type of filesystem group to use or which user can and cannot be used. Understanding these options is important to securing a Kubernetes pod, but for this example, I will leave this alone.

```
podSecurityContext: {}
  # fsGroup: 2000

securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000
```

Networking

There are two different types of networking options in this chart. One uses a local service network with a **ClusterIP** address, which exposes the service on a cluster-internal IP. Choosing this value makes the service associated with your application reachable only from within the cluster (and through **ingress**, which is set to **false** by default). The other networking option is **NodePort**, which exposes the service on each Kubernetes node's IP address on a statically assigned port. This option is recommended for running minikube, so use it for this how-to.

Before:

```
service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
```

After:

```
service:
  type: NodePort
  port: 80
```

```
ingress:
  enabled: false
```

Resources

Helm allows you to explicitly allocate hardware resources. You can configure the maximum amount of resources a Helm chart can request and the highest limits it can receive. Since I'm using Minikube on a laptop, I'll set a few limits by removing the curly braces and the hashes to convert the comments into commands.

Before:

```
resources: {}
  # We usually recommend not to specify default resources and to leave this as a
conscious
  # choice for the user. This also increases chances charts run on environments with little
  # resources, such as Minikube. If you do want to specify resources, uncomment the
following
  # lines, adjust them as necessary, and remove the curly braces after 'resources:'.
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
  #   cpu: 100m
  #   memory: 128Mi
```

After:

```
resources:
 # We usually recommend not to specify default resources and to leave this as a
conscious
  # choice for the user. This also increases chances charts run on environments with little
  # resources, such as Minikube. If you do want to specify resources, uncomment the
following
  # lines, adjust them as necessary, and remove the curly braces after 'resources:'.
   limits:
     cpu: 100m
     memory: 128Mi
   requests:
     cpu: 100m
     memory: 128Mi
```

Tolerations, node selectors, and affinities

These last three values are based on node configurations. Although I cannot use any of them in my local configuration, I'll still explain their purpose.

nodeSelector comes in handy when you want to assign part of your application to specific nodes in your Kubernetes cluster. If you have infrastructure-specific applications, you set the node selector name and match that name in the Helm chart. Then, when the application is deployed, it will be associated with the node that matches the selector.

Tolerations, **tainting**, and **affinities** work together to ensure that pods run on separate nodes. Node affinity is a property of *pods* that *attracts* them to a set of nodes (either as a preference or a hard requirement). Taints are the opposite—they allow a *node* to *repel* a set of pods.

In practice, if a node is tainted, it means that it is not working properly or may not have enough resources to hold the application deployment. Tolerations are set up as a key/value pair watched by the scheduler to confirm a node will work with a deployment.

Node affinity is conceptually similar to **nodeSelector:** it allows you to constrain which nodes your pod is eligible to be scheduled based on labels on the node. However, the labels differ because they match rules that apply to scheduling.

```
nodeSelector: {}

tolerations: []

affinity: {}
```

Deploy ahoy!

Now that you've made the necessary modifications to create a Helm chart, you can deploy it using a Helm command, add a name point to the chart, add a values file, and send it to a namespace:

```
$ helm install my-nginx-chart helmchartbuilder/ --values helmchartbuilder/values.yaml
Release “my-nginx-chart” has been upgraded. Happy Helming!
```

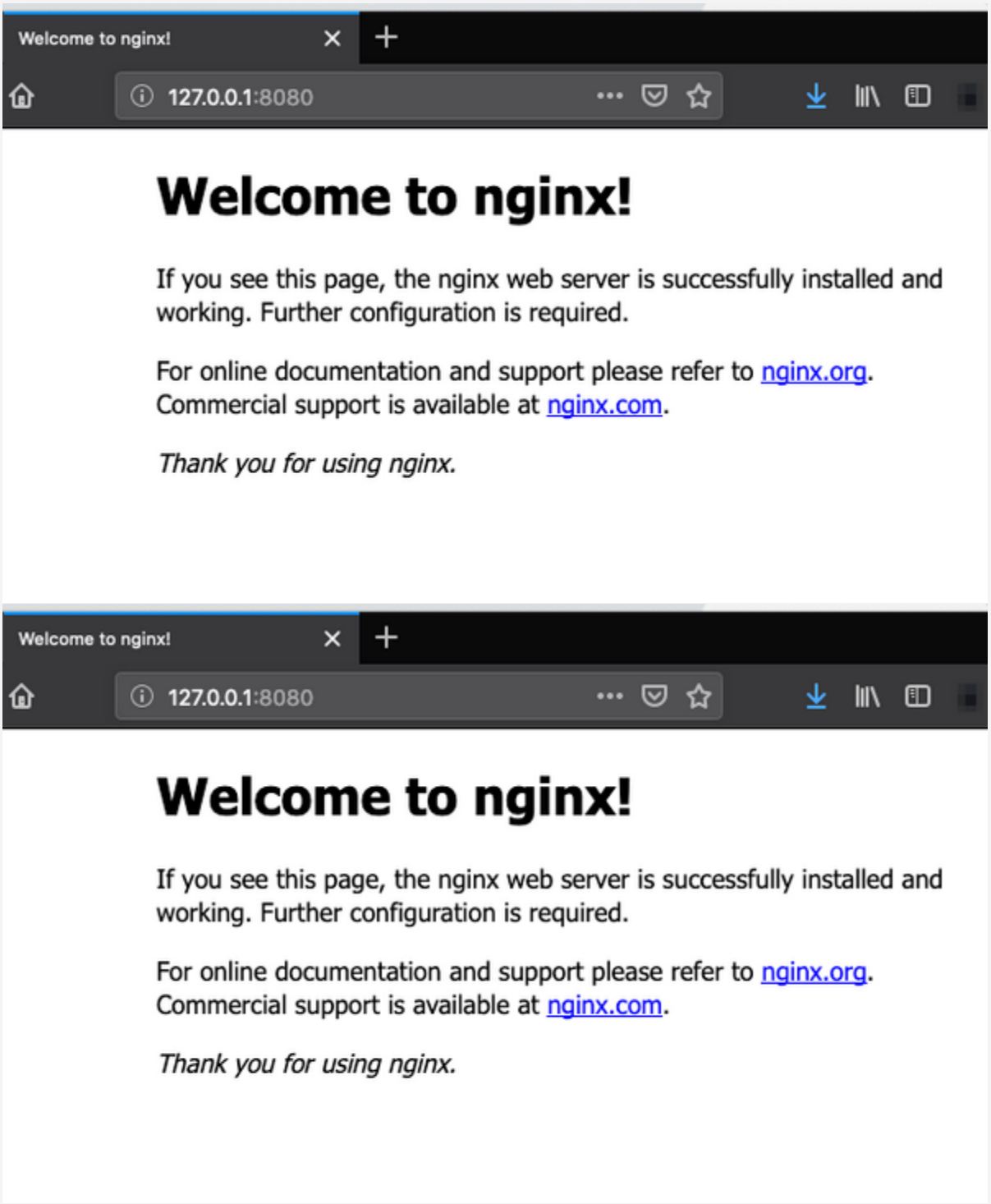
The command's output will give you the next steps to connect to the application, including setting up port forwarding so you can reach the application from your localhost. To follow those instructions and connect to an Nginx load balancer:

```
$ export POD_NAME=$(kubectl get pods -
l "app.kubernetes.io/name=helmchartbuilder,app.kubernetes.io/instance=my-nginx-
chart" -o jsonpath="{.items[0].metadata.name}")
$ echo "Visit http://127.0.0.1:8080 to use your application"
Visit http://127.0.0.1:8080 to use your application
$ kubectl port-forward $POD_NAME 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

View the deployed application

To view your application, open your web browser:

nginx_screen.png



Congratulations! You've deployed an Nginx web server by using a Helm chart!

A GitHub repo with the files for this experiment are here https://github.com/GeorgeNiece/chart-builder