# Managing Kubernetes Applications with Helm (LFS244)

## Lab Exercises

### Table of Contents

# Lab 3.1 - Creating a New Cluster Using MicroK8s

If you already have an existing cluster, you may opt to skip this section and move on.

*Note: The following instructions assume running on a Linux system using Ubuntu 18.04.4 LTS.*

MicroK8s is an open source project led by Ubuntu which enables lightweight ephemeral Kubernetes clusters which are particularly useful for test purposes. The project homepage is located at https://microk8s.io/ and the GitHub repo at https://github.com/ubuntu/microk8s.

## Step 1

Run the following command to install MicroK8s:

```
$ sudo snap install microk8s --classic --channel=1.18/stable
```

## Step 2

Add your user to the group used by MicroK8s, give yourself ownership of the `~/.kube` directory, and re-initialize your terminal session:

```
$ sudo usermod -a -G microk8s $USER
$ sudo chown -f -R $USER ~/.kube
$ sudo su - $USER
```

## Step 3

Next, run the following command to check if the Kubernetes cluster is ready:

```
$ microk8s status --wait-ready
```

This will hang in the terminal until the cluster is ready (or there is an error). Once done, the output should look similar to the following:

```
microk8s is running
addons:
cilium: disabled
dashboard: disabled
...
```

If the output contains "`microk8s is running`", your cluster should be ready to use.

### Step 4

As part of MicroK8s starting, your environment should be automatically authenticated against the new cluster. Just to be extra safe, you can run the following command:

```
$ microk8s config > ~/.kube/config
```

This outputs the MicroK8s "kubeconfig" to the file path `~/.kube/config`, which is the default location for the YAML-based Kubernetes authentication file. Many Kubernetes-related tools (such as Helm) use this file to authenticate against the Kubernetes API.

This default is overridden by the KUBECONFIG environment variable. If this environment variable is set in your environment, you have two (2) options:

1. Unset the variable for the current terminal session:
   ```
   $ unset KUBECONFIG
   ```
2. Override the contents of the file at KUBECONFIG (**Warning**: Make sure you're not overwriting some other existing kubeconfig file first):
   ```
   $ microk8s config > "${KUBECONFIG}"
   ```

### Step 5

One of the nice things about MicroK8s is that it ships with its own version of kubectl, the Kubernetes CLI. You can verify cluster access with the following command:

```
$ microk8s kubectl cluster-info
```

Expected output should look like the following:

```
Kubernetes master is running at https://127.0.0.1:16443
```

Notice the “127.0.0.1” address indicating the cluster is running locally.

### Step 6

For the rest of the course you will see references to “kubectl”. Unless you’ve installed “kubectl” separately, you’ll need to run **`microk8s kubectl`**. In order to create and activate a simple alias for this, you should run the following:

```
$ echo "alias kubectl='microk8s kubectl'" >> "${HOME}/.bash_aliases"
$ source "${HOME}/.bash_aliases"
```

You can now use the **`kubectl`** command, which will run **`microk8s kubectl`** behind the scenes.

You are now ready to move on.

# Lab 3.2 - Helm Installation

## Obtaining the Helm Binary from GitHub Release Notes

We will now go over how to install the Helm CLI on your system.

### Step 1

First, create a directory somewhere on your system called **`helm-install`**:

```
$ mkdir ${HOME}/helm-install
```

Next, open your browser to the Helm releases page at https://github.com/helm/helm/releases. Find the latest release starting with “v3” (for example “Helm v3.2.0”).

*Note: You may see bug-fix releases for the old version, Helm 2. Do NOT use those releases for this course.*

### Step 2

You should see a section in the release titled “Installation and Upgrading”, followed by several download links. Find the links related to your operating system and architecture. On most modern systems, use **amd64**. If you are unsure about your system’s architecture, you can run the following command:

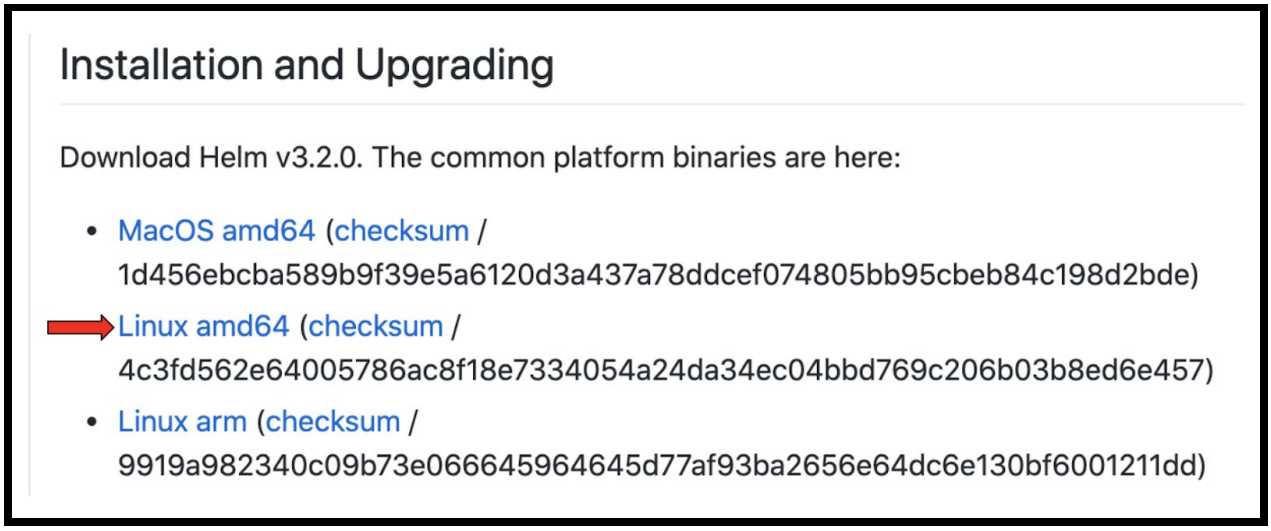
```
$ uname -m
```

If the result of that command is **`x86_64`**, use architecture **amd64**. If the result of that command is **`x86`**, use architecture **i386**. Other results should be self-explanatory. It is important to use the

correct architecture, otherwise Helm will not run on your system.

The following image shows an example of a download link for Linux amd64:




Step 3

Open a terminal and enter the install directory:

```
$ cd ${HOME}/helm-install
```

Step 4

Once you've determined the correct download link based on your system, right-click on the link and click "Copy Link Address". Then use wget to download the file:

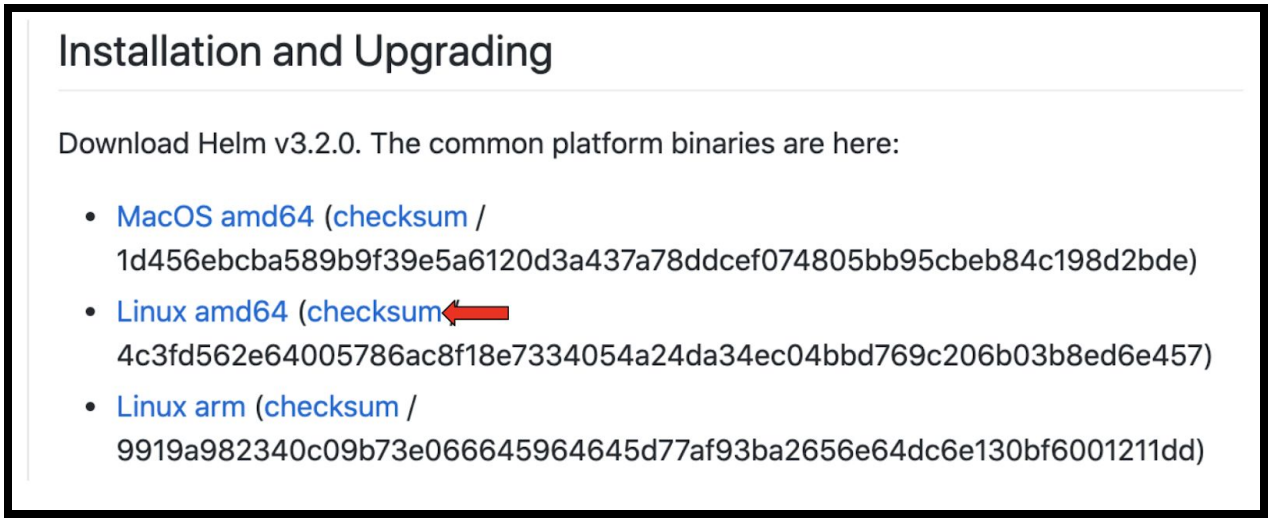
```
$ wget https://get.helm.sh/helm-v3.2.0-linux-amd64.tar.gz
```

## Optional: Verify the Checksum

You are encouraged to validate the `.tar.gz` file against the SHA-256 checksum. This validates that the released artifact has not been modified since the time it was released.

The following image shows an example of a checksum link for Linux amd64:




Step 5

Once you've determined the correct checksum link, right-click on the link and click "Copy Link Address". Then use wget to download the file:

```
$ wget https://get.helm.sh/helm-v3.2.0-linux-amd64.tar.gz.sha256sum
```

Step 6

Run the following commands (*Note: The name of your* **`.tar.gz`** *and* **`.sha256sum`** *files will be different depending on the Helm version used as well as the download and checksum links you used for your system*):

```
$ cat helm-v3.2.0-linux-amd64.tar.gz.sha256sum
4c3fd562e64005786ac8f18e7334054a24da34ec04bbd769c206b03b8ed6e457
helm-v3.2.0-linux-amd64.tar.gz

$ shasum -a 256 helm-v3.2.0-linux-amd64.tar.gz
4c3fd562e64005786ac8f18e7334054a24da34ec04bbd769c206b03b8ed6e457
helm-v3.2.0-linux-amd64.tar.gz
```

If your system is missing the **`shasum`** command, try the **`sha256sum`** command instead:

```
$ sha256sum helm-v3.2.0-linux-amd64.tar.gz
4c3fd562e64005786ac8f18e7334054a24da34ec04bbd769c206b03b8ed6e457
helm-v3.2.0-linux-amd64.tar.gz
```

Notice in the example output above that the **`.tar.gz`** file's checksum matches the expected

checksum.

## Optional: Validate the Signature

*Note: This step requires that you have gpg (https://gnupg.org/) installed on your system.*

If you want to be extra safe about what you're installing onto your system, you can validate the signature of both the `.tar.gz` and the `.sha256sum` files you have downloaded.

Each release artifact is signed by the maintainer who released it, and those signatures are then uploaded to the GitHub release. So even in the rare event that a hacker compromises the `.tar.gz` and `.sha256sum` files and replaces them with evil ones, those artifacts would not contain a valid signature.

Navigate your browser back to the release notes in GitHub. At the bottom of the release, you will find a section labeled "Assets". There will be listed numerous download links for files ending in `.asc`. Find the two `.asc` files related to the `.tar.gz` and `.sha256sum`. They should be fairly easy to locate, as they are the name of the original file suffixed with "`.asc`":


### Step 7

Once you determined the correct links, save them into the `helm-install` directory. Once again, right-click on each link and click "Copy Link Address", then use wget to download the files:

```
$ wget
https://github.com/helm/helm/releases/download/v3.2.0/helm-v3.2.0-linu
x-amd64.tar.gz.asc

$ wget
https://github.com/helm/helm/releases/download/v3.2.0/helm-v3.2.0-linu
x-amd64.tar.gz.sha256sum.asc
```

Step 8

Next obtain the latest Helm **KEYS** file. This is a file which contains a collection of all of the maintainers' PGP keys which have been used to sign a Helm release:

```
$ wget https://raw.githubusercontent.com/helm/helm/master/KEYS
```

The **helm-install** directory should now contain 5 files total (**.tar.gz**, **.sha256sum**, two **.asc** files, and **KEYS**).

Step 9

Run the following commands to verify the signatures of both the **.tar.gz** package and the checksum using GNU Privacy Guard (gpg). This will create a temporary keyring based on the **KEYS** file, then use that to validate the signatures. Be sure to replace the **TARFILE** and **SUMFILE** with the **.asc** files you have downloaded:

```
$ TARFILE="helm-v3.2.0-linux-amd64.tar.gz.asc"
$ SUMFILE="helm-v3.2.0-linux-amd64.tar.gz.sha256sum.asc"

$ mkdir -p -m 0700 gnupgtemp
$ gpg --batch --quiet --homedir=gnupgtemp --import KEYS
$ gpg --batch --no-default-keyring --keyring "gnupgtemp/pubring.kbx"
--export > "tempkeyring.gpg"

$ COUNT=$(gpg --verify --keyring="${PWD}/tempkeyring.gpg"
--status-fd=1 "${TARFILE}" | grep -c -E '^\[GNUPG:\]
(GOODSIG|VALIDSIG)'); [[ ${COUNT} -ge 2 ]] && echo "Verified signature
of ${TARFILE}" || echo "FAILED to verify ${TARFILE}"

$ COUNT=$(gpg --verify --keyring="${PWD}/tempkeyring.gpg"
--status-fd=1 "${SUMFILE}" | grep -c -E '^\[GNUPG:\]
(GOODSIG|VALIDSIG)'); [[ ${COUNT} -ge 2 ]] && echo "Verified signature
of ${SUMFILE}" || echo "FAILED to verify ${SUMFILE}"
```

If the output for the last two commands end with "`Verified signature of...`", you are good to go. You can safely ignore any warnings. Move on to the next section to install Helm CLI.

If those last two commands end with "`FAILED to verify...`", please double-check, then discreetly send an email to cncf-helm-security@lists.cncf.io. This may be a security issue. More details on Helm security disclosures can be found here: https://github.com/helm/community/blob/master/SECURITY.md.

## Installing the Helm Binary

The final step in the installation process is to extract the `.tar.gz` file which contains the Helm binary, and move it into a permanent location on your system.

### Step 10

Inside the `helm-install` directory, extract the `.tar.gz` file:

```
$ tar -zxvf helm-v3.1.1-darwin-amd64.tar.gz
linux-amd64/
linux-amd64/LICENSE
linux-amd64/README.md
linux-amd64/helm
```

The output of that command should display the extracted contents. Look for the Helm binary, a file called `helm` (in this example, `linux-amd64/helm`).

### Step 11

Lastly, move the Helm binary into a permanent location, such as `/usr/local/bin`:

```
$ sudo mv linux-amd64/helm /usr/local/bin/helm
```

*Note: The last command may require some administrative privilege (hence the `sudo`). You can also move the Helm binary to another location in your PATH.*

### Step 12

Verify Helm is installed:

```
$ which helm
/usr/local/bin/helm
```

### Step 13

Finally, make sure that Helm runs properly:

```
$ helm version
version.BuildInfo{Version:"v3.2.0",
GitCommit:"e11b7ce3b12db2941e90399e874513fbd24bcb71",
GitTreeState:"clean", GoVersion:"go1.13.10"}
```

### Step 14

Congrats! You're ready to start using Helm. Feel free to discard the entire `helm-install` directory:

```
$ cd ../
$ rm -rf helm-install/
```

# Lab 3.3 - Running Your First Application

## Creating a New Chart

Helm has a command **`helm create`** which you can use to create a new Helm chart which follows the best practices.

### Step 1

Run the following command to create a sample chart:

```
$ helm create myapp
Creating myapp
```

This will create a directory called **`myapp`** in the current directory containing several Helm-specific files. We will not dive into the contents of the chart in this chapter, but feel free to take a look inside.

The example application in the boilerplate chart is a simple Nginx deployment (Nginx is a popular HTTP web server https://nginx.org/).

## Installing the Chart

### Step 2

Installing the chart will create a new Helm release in your Kubernetes cluster. So first come up with some unique release name, such as **demo**, and install the chart with the following command:

```
$ helm install demo myapp
NAME: demo
LAST DEPLOYED: Thu Apr 16 19:05:05 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l
"app.kubernetes.io/name=myapp,app.kubernetes.io/instance=demo" -o
jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl --namespace default port-forward $POD_NAME 8080:80
```

That's it! You've just installed your first application in Kubernetes using Helm. If you run into any issues at this point, double check that you have access to a running Kubernetes cluster (see earlier in this chapter).

### Step 3

Run a simple **`kubectl`** command to see a newly-created pod:

```
$ kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
demo-myapp-d96454b47-2q856     1/1     Running   0          24s
```

## Accessing the Application

Many charts, upon install, will share some output regarding how to access the application (see previous section).

### Step 4

In order to access the Nginx web server in this example, run the following commands which will forward local traffic on port 8080 to the pod:

```
$ export POD_NAME=$(kubectl get pods --namespace default -l
"app.kubernetes.io/name=myapp,app.kubernetes.io/instance=demo" -o
jsonpath="{.items[0].metadata.name}")
$ kubectl --namespace default port-forward $POD_NAME 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Note that this command will hang in the terminal (this is expected).

### Step 5

With the command still running, open another terminal window on the same machine. Run the following command to access the running Nginx application:

```
$ curl http://127.0.0.1:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully
installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

In the original terminal, you'll notice some output:

```
Handling connection for 8080
```

This indicates that the request from the **`curl`** command was successfully forwarded to the running Nginx pod. Click "Ctrl+C" in the original terminal window to stop the port forwarding. You

can close the other terminal window.

## Deleting a Release

### Step 6

To delete the release from this example, run the following command:

```
$ helm delete demo
release "demo" uninstalled
```

### Step 7

This will cause all Kubernetes resources associated with the release to be removed. For example, lets see if that pod is still there:

```
$ kubectl get pods
No resources found.
```

Gone! You've successfully deleted the release.

# Lab 4.1 - Installing a Chart With Custom Values

Let's combine our knowledge of Helm chart templating and chart repositories to install a third-party application. For this, we will use the **`bitnami/wordpress`** chart from before to install a new WordPress site.

*Note: Before going any further, you must make sure your Kubernetes cluster has some default storage class defined as well as cluster DNS.*

## Step 1

You can check for if you have a default storage class using the following command:

```
$ kubectl get storageclass
No resources found in default namespace.
```

## Step 2

This indicates you do not have any storage providers. If running on MicroK8s, this is easily fixed by running the following command:

```
$ microk8s enable storage
Enabling default storage class
deployment.apps/hostpath-provisioner created
storageclass.storage.k8s.io/microk8s-hostpath created
serviceaccount/microk8s-hostpath created
clusterrole.rbac.authorization.k8s.io/microk8s-hostpath created
clusterrolebinding.rbac.authorization.k8s.io/microk8s-hostpath created
Storage will be available soon
```

### Step 3

After some time, you should see the **microk8s-hostpath** storage class available:

```
$ kubectl get storageclass
NAME                          PROVISIONER            RECLAIMPOLICY
VOLUMEBINDINGMODE    ALLOWVOLUMEEXPANSION    AGE
microk8s-hostpath (default)   microk8s.io/hostpath   Delete
Immediate            false                   41s
```

For more information about Kubernetes storage, please see https://kubernetes.io/docs/concepts/storage/storage-classes/.

### Step 4

To see if DNS is enabled in your cluster, check for the **kube-dns** service in the **kube-system** namespace:

```
$ kubectl get service -n kube-system
No resources found in kube-system namespace.
```

### Step 5

If running on MicroK8s, you can enable DNS with the following command:

```
$ microk8s enable dns
Enabling DNS
Applying manifest
serviceaccount/coredns created
configmap/coredns created
deployment.apps/coredns created
service/kube-dns created
clusterrole.rbac.authorization.k8s.io/coredns created
clusterrolebinding.rbac.authorization.k8s.io/coredns created
Restarting kubelet
DNS is enabled
```

### Step 6

After some time, you should see a **kube-dns** service in the **kube-system** namespace:

```
$ kubectl get service -n kube-system
NAME       TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
```

```
AGE
kube-dns   ClusterIP   10.152.183.10   <none>
53/UDP,53/TCP,9153/TCP   71s
```

You are now ready to move on.

Let's say we want to expose traffic to the WordPress site on the host machine at port 30001. To do so, we will need to use a **NodePort** service type, which is a type of Kubernetes service which listens on the same port on each node it is running on.

Upon inspecting the **values.yaml** file in the **bitnami/wordpress** chart, we see that we can modify the service type using the **service.type** value, and the port using the **service.nodePorts.http** value.

Lastly, let's use an exact version of the Helm chart and container image so we don't run into any unexpected issues when Bitnami updates this chart. We will use the chart version 9.2.5.

### Step 7

To install the **bitnami/wordpress** chart with our custom version and values, run the following:

```
$ helm install wordpress bitnami/wordpress \
     --version 9.2.5 \
      --set service.type=NodePort \
      --set service.nodePorts.http=30001
NAME: wordpress
LAST DEPLOYED: Tue May 19 05:51:08 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
** Please be patient while the chart is being deployed **

To access your WordPress site from outside the cluster follow the
steps below:

1. Get the WordPress URL by running these commands:

   export NODE_PORT=$(kubectl get --namespace default -o
jsonpath="{.spec.ports[0].nodePort}" services wordpress)
   export NODE_IP=$(kubectl get nodes --namespace default -o
```

```
jsonpath="{.items[0].status.addresses[0].address}")
   echo "WordPress URL: http://$NODE_IP:$NODE_PORT/"
   echo "WordPress Admin URL: http://$NODE_IP:$NODE_PORT/admin"

2. Open a browser and access WordPress using the obtained URL.

3. Login with the following credentials below to see your blog:

  echo Username: user
  echo Password: $(kubectl get secret --namespace default wordpress -o
jsonpath="{.data.wordpress-password}" | base64 --decode)
```

Step 8

After a couple minutes, after the container images have downloaded, and WordPress is connected successfully to its MariaDB database (defined as a dependency chart), you should see two pods running:

```
$ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
wordpress-5c8cc7769-897rh    1/1     Running   0          110s
wordpress-mariadb-0          1/1     Running   0          110s
```

Step 9

Additionally, you should see the NodePort service running on port 30001:

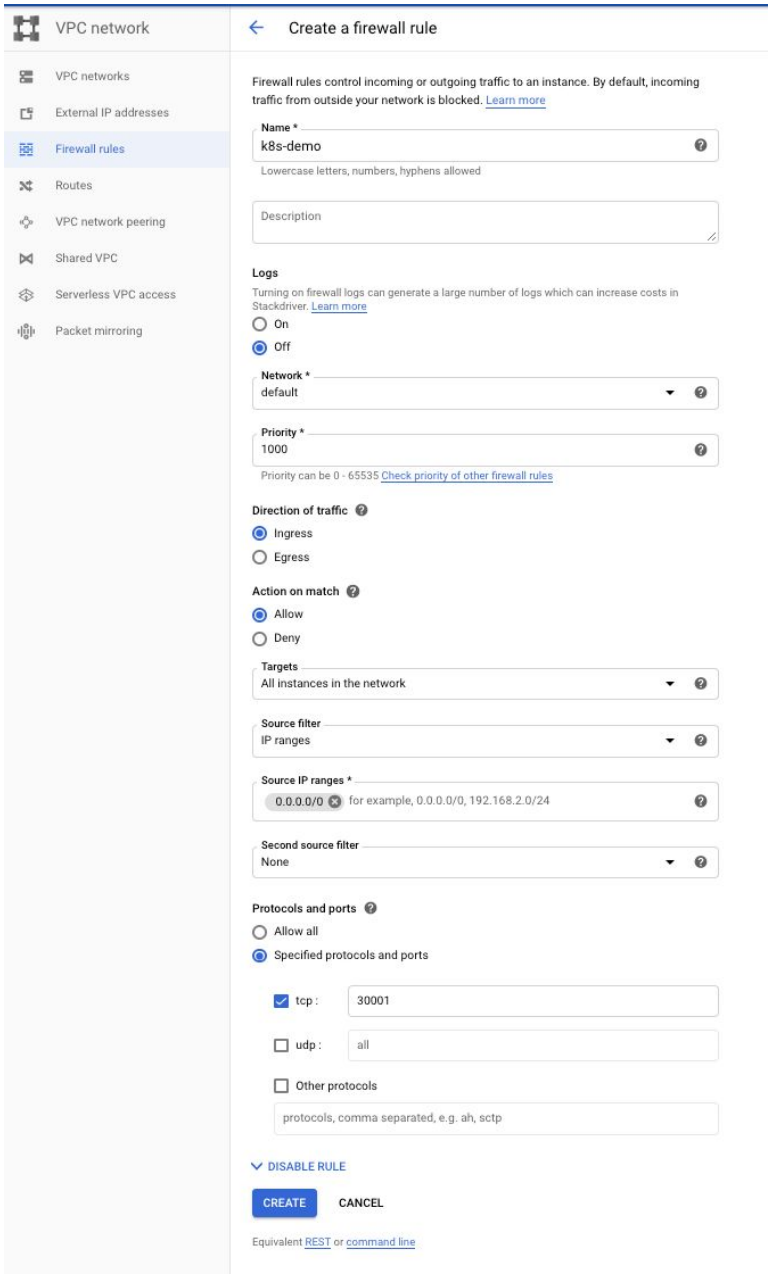
```
$ kubectl get service
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
kubernetes          ClusterIP   10.152.183.1     <none>        443/TCP
14d
wordpress           NodePort    10.152.183.210   <none>
80:30001/TCP,443:31265/TCP   2m20s
wordpress-mariadb   ClusterIP   10.152.183.178   <none>
3306/TCP                     2m20s
```

Identify the public IP on the server you have been running these commands on. Then, in your web browser, navigate to http://<ip>:30001.

*Note: If you have trouble connecting, make sure to add a firewall rule allowing inbound access to port 30001. For source, use either 0.0.0.0/0 (open to internet), or <your_ip>/32 (just your IP address).*

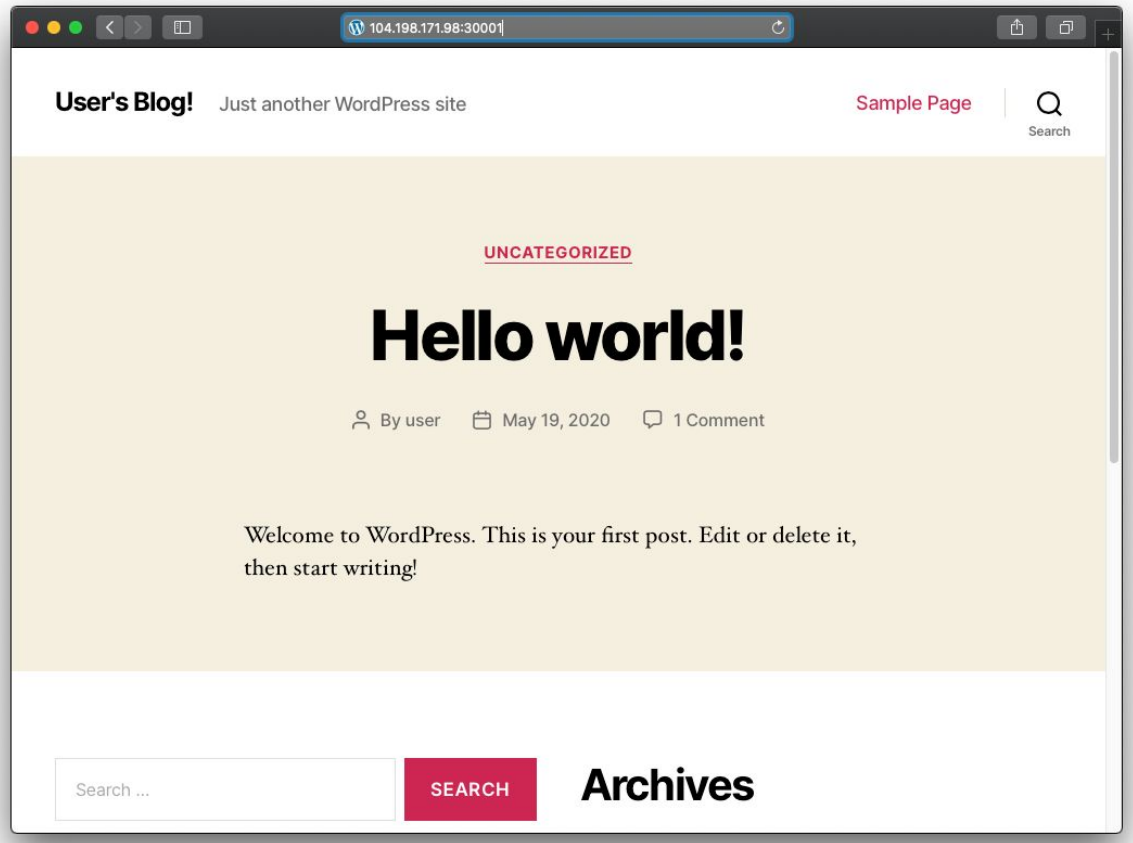
 

In Google Cloud, this looks like the following:




If things are working properly, you should see a fresh WordPress installation in your browser:

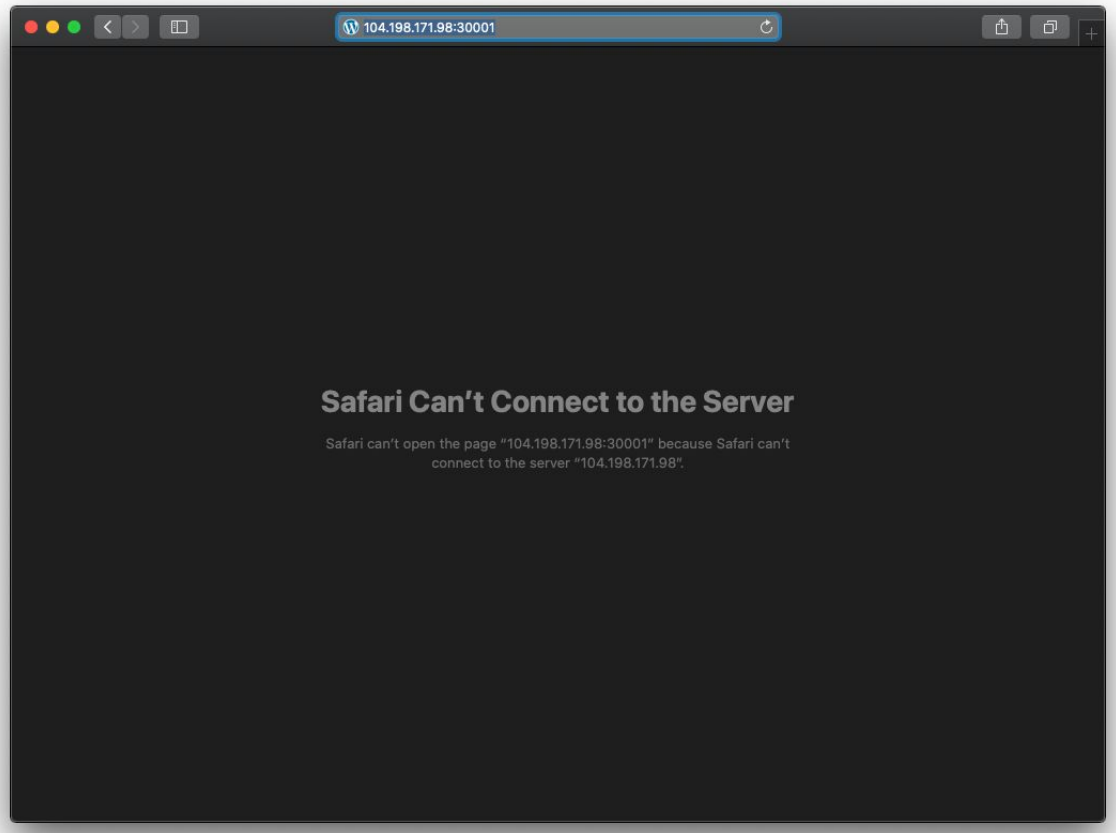



Step 10

To cleanup this install, you can run **`helm delete`**:

```
$ helm delete wordpress
release "wordpress" uninstalled
```

You'll notice if you refresh your browser after this, the site should be down:




Optionally, you can also remove the firewall rule you created if that step was required.

# Lab 5.1 - Using Helm to Manage the Lifecycle of a Sample Python Application

In this lab we will learn how to manage the lifecycle of a simple Python (**`python3`**) web application on Kubernetes using Helm.

We will start by explaining the source code of the app and what it does. We will then briefly walk through the steps of publishing a container image to a registry so that Kubernetes can fetch our application and run it as a containerized process.

Afterwards, we will start building a Helm chart for this application from scratch, walking through the various files in the chart and how everything works together to manage this application on Kubernetes.

Lastly, we will run various Helm commands to go step-by-step through the full lifecycle of this application.

## Setup

This lab should be supplemented by a .zip file containing all of the files referenced in this lab (go to Chapter 1: Introduction > Course Information > Course Resources). Copy this .zip file onto your machine and extract it, then enter the **`workspace/`** directory:

Step1

```
$ unzip ch5_workspace.zip
Archive:  ch5_workspace.zip
   creating: workspace/
  inflating: workspace/server.py
```

```
  inflating: workspace/index.html
  inflating: workspace/Dockerfile
   creating: workspace/quotegen/
  inflating: workspace/quotegen/Chart.yaml
   creating: workspace/quotegen/templates/
  inflating: workspace/quotegen/templates/pre-delete-hook.yaml
  inflating: workspace/quotegen/templates/deployment.yaml
  inflating: workspace/quotegen/templates/service.yaml
  inflating: workspace/quotegen/templates/_helpers.tpl
  inflating: workspace/quotegen/templates/api-test.yaml
  inflating: workspace/quotegen/values.yaml

$ cd workspace/
```

Note: The filename may be slightly different from “ch5_workspace.zip”. Modify the “unzip” to reference the correct filename.

### Step 2

The **`docker`** command is required to build container images. If running on Ubuntu, use the following command to install Docker:

```
$ sudo snap install docker
docker 19.03.11 from Canonical✓ installed
```

Run the following command to verify that Docker is installed:

```
$ sudo docker info | grep "Server Version"
 Server Version: 19.03.11
```

### Step 3

Lastly, we must have access to a container registry to push images to. Using MicroK8s, you can start a local registry at **`localhost:32000`** by running the following command:

```
$ microk8s enable registry
Enabling the private registry
storage is already enabled
Applying registry manifest
namespace/container-registry created
persistentvolumeclaim/registry-claim created
deployment.apps/registry created
service/registry created
```

```
The registry is enabled
```

You can also use public registries such as Docker Hub, Quay, and others if you have access to that.

## The Application

The application itself is a random quote generator called “quotegen”. When a user visits the web application in a browser, they will be met with a thoughtful phrase to ponder upon.

This web app will be comprised of two primary files:

- **`index.html`**: A static HTML file which requests new quotes.
- **`server.py`**: A backend web server written in Python that returns great quotes.

### Step 4

Here is the source code for **`index.html`**:

```
<html>
    <head>
        <title>quotegen - worlds first quote generator</title>
        <style>
            body {
                background: black;
            }
            #main {
                color: white;
                text-align: center;
                font-weight: bold;
                font-family: monospace;
                padding-top: 60px;
            }
        </style>
    </head>
    <body>
        <div id="main"></div>
        <script>
            document.addEventListener('DOMContentLoaded', function() {
                var xhr = new XMLHttpRequest();
                xhr.onreadystatechange = function() {
                    if (xhr.readyState === 4){
```

```
                    var data = JSON.parse(xhr.responseText);
                    var elem = document.getElementById('main');
                    elem.style.fontSize = data['size']+'px';
                    elem.innerHTML = data['text'];
                }
            };
            xhr.open('GET', '/quote');
            xhr.send();
        });
    </script>
  </body>
</html>
```

When this page is loaded in the browser, using JavaScript, we make an AJAX request to an API endpoint to retrieve a JSON payload containing a new quote. Then, the quote is extracted from the JSON body, and displayed on the page for the user.

### Step 5

Here is the source code for **server.py**:

```
import os
import json
import random
import http.server
import socketserver

FONT_SIZE = os.environ.get('FONT_SIZE', '20')
RAW_INDEX_CONTENTS = bytes(open('./index.html').read(), 'UTF-8')
QUOTES = [
   'Live, laugh, love.',
   'Keep calm and carry on.',
   'Do what you feel in your heart to be right - for you'll be
criticized anyway.',
   'You may not control all the events that happen to you, but you can
decide not to be reduced by them',
   'Truth is, everybody is going to hurt you; you just gotta find the
ones worth suffering for'
   'If life gives you lemons, make lemonade',
   'You miss 100 percent of the shots you dont take',
   'Be the change you wish to see in the world',
   'Be kind, for everyone you meet is fighting a hard battle.',
   'Learning to unlearn is the highest form of learning.'
```

```
]

def get_quote():
   return {'size': FONT_SIZE, 'text': random.choice(QUOTES)}

class Server(http.server.SimpleHTTPRequestHandler):
   def do_GET(self):
       path = self.path
       if path == '/' or path == '/index.html':
           self.send_response(200, 'OK')
           self.send_header('Content-type', 'text/html')
           self.end_headers()
           self.wfile.write(RAW_INDEX_CONTENTS)
       elif path == '/quote':
           quote = get_quote()
           self.send_response(200, 'OK')
           self.send_header('Content-type', 'application/json')
           self.end_headers()
           self.wfile.write(bytes(json.dumps(quote), 'UTF-8'))
       else:
           self.send_response(404, 'NOT FOUND')

if __name__ == '__main__':
   print('Starting server...')
   socketserver.TCPServer(('0.0.0.0', 8080), Server).serve_forever()
```

This source file starts a web server on port 8080. When it receives an HTTP request for the path `/`, it will simply return the contents of `index.html`. If it receives an HTTP request for the path `/quote`, it will determine a random quote and write it back to the client encoded as JSON.

## Containerizing and Publishing the Application with Docker

In order for Kubernetes to run our application as a container, it must first be published to a container registry which Kubernetes has access to.

Once we have Docker up-and-running, we will create a simple `Dockerfile` which will provide instructions for how to package our app as a container image.

### Step 6

Here are the contents of `Dockerfile`:

```
FROM python:3-alpine
WORKDIR /app
COPY server.py .
COPY index.html .
EXPOSE 8080
CMD ["python3", "server.py"]
```

This file should be placed directly next to `index.html` and `app.py`.

### Step 7

Using Docker, build the image with the reference `localhost:32000/quotes:v1`:

```
$ sudo docker build . -t localhost:32000/quotes:v1
Sending build context to Docker daemon  6.656kB
Step 1/6 : FROM python:3-alpine
3-alpine: Pulling from library/python
df20fa9351a1: Pull complete
36b3adc4ff6f: Pull complete
7031d6d6c7f1: Pull complete
81b7f5a7444b: Pull complete
0f8a54c5d7c7: Pull complete
Digest:
sha256:c5623df482648cacece4f9652a0ae04b51576c93773ccd43ad459e2a195906d
d
Status: Downloaded newer image for python:3-alpine
 ---> 8ecf5a48c789
Step 2/6 : WORKDIR /app
 ---> Running in 525fe7b000f8
Removing intermediate container 525fe7b000f8
 ---> e20e2c3b744e
Step 3/6 : COPY server.py .
 ---> 0e5415a83174
Step 4/6 : COPY index.html .
 ---> 48e60ea78545
Step 5/6 : EXPOSE 8080
 ---> Running in 269551c36b5f
Removing intermediate container 269551c36b5f
 ---> 24d4dc5c0d1f
Step 6/6 : CMD ["python3", "server.py"]
 ---> Running in 82f756418802
Removing intermediate container 82f756418802
```

```
 ---> 059e859346fd
Successfully built 059e859346fd
Successfully tagged localhost:32000/quotes:v1
```

Step 8

You could test to see if your application runs properly:

```
$ sudo docker run -it --rm localhost:32000/quotes:v1
Starting server...
```

(press CTRL+C to exit the running container)

Step 9

Finally, publish the container image to the registry:

```
$ sudo docker push localhost:32000/quotes:v1
The push refers to repository [localhost:32000/quotes]
5dd75bb60e7f: Pushed
008a5aac9e66: Pushed
da5caef63bd4: Pushed
fffdb84c36f2: Layer already exists
50205a7df19a: Layer already exists
ef833453b9c7: Layer already exists
408e53c5e3b2: Layer already exists
50644c29ef5a: Layer already exists
v1: digest:
sha256:76166f6332d33581fec218ea8fe58c6650d3daf2a705775cf9f6544ff46b03d
9 size: 1989
```

## The Helm Chart for the Application

Now that we have a container image, we need to create a Kubernetes Deployment object which references this image for the current version of our application.

The Helm chart for the application will generate all of the Kubernetes resources necessary for running this application, which will allow us to dynamically set not only the version of the container image to run, but also various other things such as the container's table of environment variables.

Step 10

The Helm chart for our application is contained in the **quotegen** directory. Enter this directory and examine the files:

```
$ cd quotegen/
```

Step 11

Checkout the **Chart.yaml** for this chart:

```
$ cat Chart.yaml
name: quotegen
version: 0.1.0
description: worlds first quote generator
```

*Note: The **name:** field of **Chart.yaml** must always match the directory it is contained in.*

Next 12

Next, inspect **values.yaml** containing the default configuration settings for our chart:

```
$ cat values.yaml
image:
  repo: localhost:32000/quotes
  tag: v1

env:
  FONT_SIZE: 20

replicas: 2

service:
  type: NodePort
  port: 30001
```

Step 13

Next, let's create the chart's **templates/** directory:

```
$ mkdir -p templates/
```

Inside the **templates/** directory, we will have five files:

- `_helpers.tpl`: A file with a few partial definitions.
- `deployment.yaml`: A template for a Deployment resource.
- `service.yaml`: A template for a Service resource.
- `api-test.yaml`: A template for a Job to run to test API connectivity.
- `pre-delete-hook.yaml`: A template for a Job to run on the "pre-delete" event.

Take some time to check out the source code for each of these files within the `templates/` directory.

### Managing the Application's Lifecycle Using Helm Commands

Step 14

Re-enter the top-level directory for this lab:

```
$ cd ../ # workspace
```

Step 15

We should now have a directory layout that looks like the following:

```
$ tree .
.
├── Dockerfile
├── index.html
├── quotegen
│   ├── Chart.yaml
│   ├── _helpers.tpl
│   ├── api-test.yaml
│   ├── pre-delete-hook.yaml
│   ├── templates
│   │   ├── _helpers.tpl
│   │   ├── api-test.yaml
│   │   ├── deployment.yaml
│   │   ├── pre-delete-hook.yaml
│   │   └── service.yaml
│   └── values.yaml
└── server.py

2 directories, 13 files
```

Step 16

Now we enter the very first stage in the lifecycle of this application: its first installation. Use Helm to install the **quotegen** chart, creating an initial release:

```
$ helm install quotegen-prod quotegen/
NAME: quotegen-prod
LAST DEPLOYED: Wed Jun 17 10:08:13 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
```

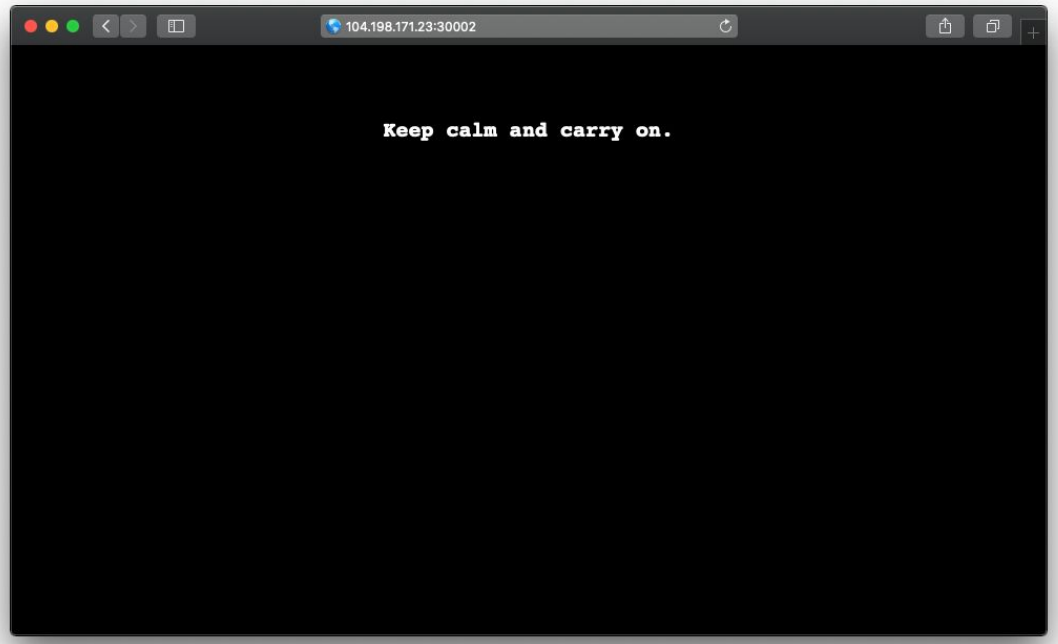
Step 17

Next, let's verify that the application is working properly by executing our test Job:

```
$ helm test quotegen-prod --timeout=30s
NAME: quotegen-prod
LAST DEPLOYED: Wed Jun 17 10:08:13 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE:     quotegen-prod-api-test
Last Started:   Wed Jun 17 10:08:24 2020
Last Completed: Wed Jun 17 10:08:26 2020
Phase:          Succeeded
```

Looks like things appear to be working correctly. The final step is to do some manual quality assurance. Load the app in your web browser and refresh the page a couple times to see a few different quotes:




*Note: Keep in mind that in order to access port 30001 on this server, you must open up the firewall to allow inbound access from your IP address to port 30001 as we went over in a Lab 4.1.*

Let's say our users aren't happy with font size on the screen. They want it to be bigger (much bigger!). This is something we can easily modify with one of the nested configuration values we have in our chart, **`env.FONT_SIZE`**.

### Step 18

Let's upgrade our application to set **`env.FONT_SIZE`** to **80** (for 80 pixels):

```
$ helm upgrade quotegen-prod quotegen/ --set env.FONT_SIZE=80
Release "quotegen-prod" has been upgraded. Happy Helming!
NAME: quotegen-prod
LAST DEPLOYED: Wed Jun 17 10:09:12 2020
NAMESPACE: default
STATUS: deployed
REVISION: 2
```

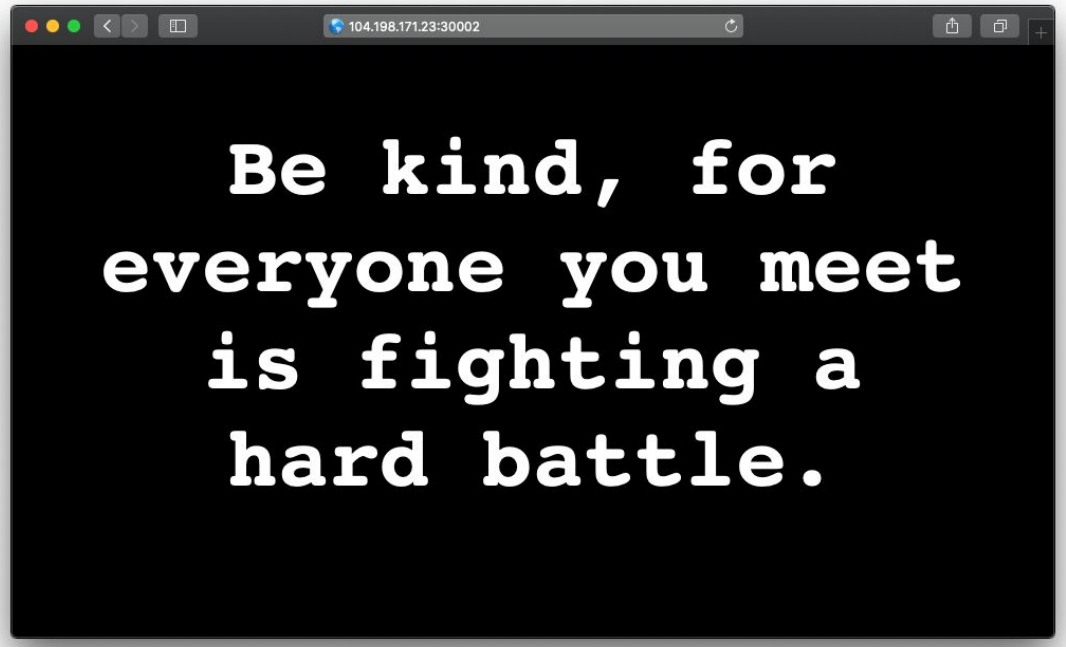
 

Step 19

This will create a second release revision. We can view the current revision using helm status:

```
$ helm status quotegen-prod
NAME: quotegen-prod
LAST DEPLOYED: Wed Jun 17 10:09:12 2020
NAMESPACE: default
STATUS: deployed
REVISION: 2
```

If you navigate to the application in the browser once again, you should now see bigger, better quotes on the screen:




Your boss isn't happy with the quote content. They say that the quotes aren't thought-provoking enough. The quotes need to really inspire people.

Step 20

A change like this will require us to make an update to the Python code. Here's a new version of `server.py` with a modified `get_quote()` function:

```
import os
import json
import random
import http.server
import socketserver

FONT_SIZE = os.environ.get('FONT_SIZE', '20')
RAW_INDEX_CONTENTS = bytes(open('./index.html').read(), 'UTF-8')
QUOTES = [
   'Live, laugh, love.',
   'Keep calm and carry on.',
   'Do what you feel in your heart to be right - for you'll be
criticized anyway.',
   'You may not control all the events that happen to you, but you can
decide not to be reduced by them',
   'Truth is, everybody is going to hurt you; you just gotta find the
ones worth suffering for'
   'If life gives you lemons, make lemonade',
   'You miss 100 percent of the shots you dont take',
   'Be the change you wish to see in the world',
   'Be kind, for everyone you meet is fighting a hard battle.',
   'Learning to unlearn is the highest form of learning.'
]

def get_quote():
   text = 'Be very inspired: ' + random.choice(QUOTES)
   return {'size': FONT_SIZE, 'text': QUOTES[0]}


class Server(http.server.SimpleHTTPRequestHandler):
   def do_GET(self):
       path = self.path
       if path == '/' or path == '/index.html':
           self.send_response(200, 'OK')
           self.send_header('Content-type', 'text/html')
           self.end_headers()
           self.wfile.write(RAW_INDEX_CONTENTS)
       elif path == '/quote':
           quote = get_quote()
           self.send_response(200, 'OK')
           self.send_header('Content-type', 'application/json')
```

```
            self.end_headers()
            self.wfile.write(bytes(json.dumps(quote), 'UTF-8'))
        else:
            self.send_response(404, 'NOT FOUND')

if __name__ == '__main__':
   print('Starting server...')
   socketserver.TCPServer(('0.0.0.0', 8080), Server).serve_forever()
```

Step 21

Build and push a new container image with these changes with a new tag, **v2**:

```
$ sudo docker build . -t localhost:32000/quotes:v2
Sending build context to Docker daemon  334.8kB
Step 1/6 : FROM python:3-alpine
 ---> 8ecf5a48c789
Step 2/6 : WORKDIR /app
 ---> Using cache
 ---> e20e2c3b744e
Step 3/6 : COPY server.py .
 ---> 976b42c82d6b
Step 4/6 : COPY index.html .
 ---> 9ec306a9828a
Step 5/6 : EXPOSE 8080
 ---> Running in 658fa87aa621
Removing intermediate container 658fa87aa621
 ---> b8b33d7f06b4
Step 6/6 : CMD ["python3", "server.py"]
 ---> Running in 9676d51b9cc6
Removing intermediate container 9676d51b9cc6
 ---> 13c0f11f0469
Successfully built 13c0f11f0469
Successfully tagged localhost:32000/quotes:v2

$ sudo docker push localhost:32000/quotes:v2
The push refers to repository [localhost:32000/quotes]
ab4875387f73: Pushed
761063c46537: Pushed
da5caef63bd4: Layer already exists
fffdb84c36f2: Layer already exists
50205a7df19a: Layer already exists
```

```
ef833453b9c7: Layer already exists
408e53c5e3b2: Layer already exists
50644c29ef5a: Layer already exists
v2: digest:
sha256:e1ec54d6d4ab83062d98da089a0e046df6c45be1778a87c7fc72e25a4fdea68
7 size: 1989
```

Step 22

Now let's upgrade our application to use this new container image:

```
$ helm upgrade quotegen-prod quotegen/ --reuse-values --set
image.tag=v2
Release "quotegen-prod" has been upgraded. Happy Helming!
NAME: quotegen-prod
LAST DEPLOYED: Wed Jun 17 10:21:32 2020
NAMESPACE: default
STATUS: deployed
REVISION: 3
```

*Note: The `--reuse-values` flag used here allows us to merge into the existing set of values, so we do not accidentally reset `env.FONT_SIZE`.*

Step 23

Next, let's verify that the application is working properly by executing our test Job:

```
$ helm test quotegen-prod --timeout=30s
NAME: quotegen-prod
LAST DEPLOYED: Wed Jun 17 10:21:32 2020
NAMESPACE: default
STATUS: deployed
REVISION: 3
TEST SUITE:     quotegen-prod-api-test
Last Started:   Wed Jun 17 10:23:08 2020
Last Completed: Wed Jun 17 10:23:18 2020
Phase:          Failed
Error: timed out waiting for the condition
```

Oh no! The test failed! Looks like the same terrible quote is being sent back by the API over and over.

Step 24

Before we try to determine what exactly went wrong with the changes, let's quickly rollback the changes to get to the previous version of the application:

```
$ helm rollback quotegen-prod 2
Rollback was a success! Happy Helming!
```

Step 25

Let's try testing again to make sure things are back to normal:

```
$ helm test quotegen-prod --timeout=30s
NAME: quotegen-prod
LAST DEPLOYED: Wed Jun 17 10:23:54 2020
NAMESPACE: default
STATUS: deployed
REVISION: 4
TEST SUITE:     quotegen-prod-api-test
Last Started:   Wed Jun 17 10:24:25 2020
Last Completed: Wed Jun 17 10:24:27 2020
Phase:          Succeeded
```

Looks like we're in the clear. Phew, that was close. Thankfully Helm came to the rescue there.

Step 26

After inspecting our Python code we find the root cause, a bad line in our **`get_quote()` function**:

```
def get_quote():
   text = 'Be very inspired: ' + random.choice(QUOTES)
   return {'size': FONT_SIZE, 'text': QUOTES[0]}
```

We call your boss to deliver the disappointing news about the failed feature. Before we're able to explain, they tell us the entire project has been cancelled. It turns out the competition launched a bulletproof quote generator just last week. Your company needs to rethink its strategy, and frankly, its entire existence.

So you're instructed to permanently take down the **`quotegen`** app. You poured the last 8 months of your life into this project, but whatever. You suppose you must do what is best for the company.

Step 27

Let's uninstall the application using **`helm delete`**:

```
$ helm delete quotegen-prod
release "quotegen-prod" uninstalled
```

Step 28

Goodbye autogen... But wait! You recall adding a hook to the Helm chart to generate one last quote. A final farewell, per se:

```
$ kubectl get pods | grep pre-delete-hook
quotegen-prod-pre-delete-hook-dm9rr   0/1     Completed     0
35s

$ kubectl logs quotegen-prod-pre-delete-hook-dm9rr
{"size": "20", "text": "Be the change you wish to see in the world"}
```

# Lab 6.1 - Hosting a Simple Chart Repository with Python

## The Repository Index (index.yaml)

At the core of each chart repository is a file called `index.yaml`. This is known as the **repository index**. This file contains a list of all available charts in the repository, and all of their versions. This file should be accessible at the relative root of the chart repo URL.

### Step 1

Let's use an example from Chapter 4, the public chart repository provided by Bitnami. You may recall you ran the following command to add the repository:

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

Behind the scenes, Helm is fetching the repository index from **https://charts.bitnami.com/bitnami/index.yaml**.

### Step 2

Let's see what this file looks like if we download it directly:

```
$ curl -s https://charts.bitnami.com/bitnami/index.yaml | less
apiVersion: v1
entries:
  airflow:
  - apiVersion: v1
```

```
    appVersion: 1.10.10
    created: "2020-07-02T07:45:13.7728364Z"
    description: Apache Airflow is a platform to programmatically
author, schedule
      and monitor workflows.
    digest:
d72687d8ca8f85411d7273ad64cee41afb19b007dae0acf1c0d50e96333be2e5
    engine: gotpl
    home: https://airflow.apache.org/
    icon:
https://bitnami.com/assets/stacks/airflow/img/airflow-stack-110x117.pn
g
    keywords:
    - apache
    - airflow
    - workflow
    - dag
    maintainers:
    - email: containers@bitnami.com
      name: Bitnami
    name: airflow
    sources:
    - https://github.com/bitnami/bitnami-docker-airflow
    urls:
    - https://charts.bitnami.com/bitnami/airflow-6.3.5.tgz
    version: 6.3.5
  - apiVersion: v1
    appVersion: 1.10.10
    created: "2020-06-28T21:39:48.486454019Z"
    description: Apache Airflow is a platform to programmatically
author, schedule
      and monitor workflows.
    digest:
508f6c22d45963bdcdfbfb26c1b2b1ae5eab2d17522bfccf6f241aabc226bde1
    engine: gotpl
    home: https://airflow.apache.org/
    icon:
https://bitnami.com/assets/stacks/airflow/img/airflow-stack-110x117.pn
g
    keywords:
    - apache
    - airflow
```

```
    - workflow
    - dag
    maintainers:
    - email: containers@bitnami.com
      name: Bitnami
    name: airflow
    sources:
    - https://github.com/bitnami/bitnami-docker-airflow
    urls:
    - https://charts.bitnami.com/bitnami/airflow-6.3.4.tgz
    version: 6.3.4
…
generated: "2020-07-02T08:39:50.316281483Z"
```

The whole file is over 4 MB, so it has been shortened for this example. The most important part of this file is the **entries** section. This is a map which maps unique chart names in this repository to a list of their available versions. For example, the example above shows two (2) versions of the “airflow” chart. Let’s look as one of these versions for the “airflow” chart:

```
  - apiVersion: v1
    appVersion: 1.10.10
    created: "2020-07-02T07:45:13.7728364Z"
    description: Apache Airflow is a platform to programmatically
author, schedule
      and monitor workflows.
    digest:
d72687d8ca8f85411d7273ad64cee41afb19b007dae0acf1c0d50e96333be2e5
    engine: gotpl
    home: https://airflow.apache.org/
    icon:
https://bitnami.com/assets/stacks/airflow/img/airflow-stack-110x117.pn
g
    keywords:
    - apache
    - airflow
    - workflow
    - dag
    maintainers:
    - email: containers@bitnami.com
      name: Bitnami
    name: airflow
    sources:
```

```
  - https://github.com/bitnami/bitnami-docker-airflow
  urls:
  - https://charts.bitnami.com/bitnami/airflow-6.3.5.tgz
  version: 6.3.5
```

Does this look familiar? That's because the majority of this information is gathered from the chart's **Chart.yaml** file when the index is created. One section in each entry that is unique to repository indexes is the **urls** section. This contains a list of available places to download this specific chart version (usually just a single location is provided).

When you run **helm install bitnami/airflow**, Helm will first determine the latest semantic version of the chart based on the cached version of the index (**6.3.5**). After this, Helm will attempt to download the chart based on the contents of the url field of the entry (i.e. **https://charts.bitnami.com/bitnami/airflow-6.3.5.tgz**). After the chart has been downloaded, Helm then proceeds to install it.

Helm provides commands for building your own **index.yaml** which you can use to host a chart repository.

### Step 3

First, let's create a test chart:

```
$ helm create mychart
Creating mychart
```

### Step 4

Next, let's create a directory which will contain all of our chart tarballs:

```
$ mkdir -p lab/
```

### Step 5

Now let's package the **mychart** chart into this directory:

```
$ helm package mychart/ -d lab/
Successfully packaged chart and saved it to: lab/mychart-0.1.0.tgz
```

### Step 6

Next, all you need to do is run **helm repo index**, pointing to the directory containing this chart:

```
$ helm repo index lab/
```

Step 7

This should create a new `index.yaml` file inside the directory containing a single entry for the `mychart` chart:

```
$ cat lab/index.yaml
apiVersion: v1
entries:
  mychart:
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2020-06-02T11:50:33.588597349Z"
    description: A Helm chart for Kubernetes
    digest:
64be7ad4e2ab08a57229c83f7f6b34f324a5a16305817d549052d20c69633ee0
    name: mychart
    type: application
    urls:
    - mychart-0.1.0.tgz
    version: 0.1.0
generated: "2020-06-02T11:50:33.58767312Z"
```

## Updating the Index with New Chart Versions

As you iterate on your chart, you will want to make sure to regenerate the repository index to contain the latest versions.

Step 8

Let's make a slight modification to the `mychart` chart, just bumping the version to `0.2.0`:

```
$ sed -i 's/^version:.*$/version: 0.2.0/' mychart/Chart.yaml
```

Step 9

Now, let's create another tarball for this new chart version:

```
$ helm package mychart/ -d lab/
Successfully packaged chart and saved it to: lab/mychart-0.2.0.tgz
```

Step 10

Now, all we need to do to regenerate our index file is to re-run the **`helm repo index`** command:

```
$ helm repo index lab/
```

Step 11

You will see that our index file is now larger, with a new entry for the latest chart version:

```
$ cat lab/index.yaml
apiVersion: v1
entries:
  mychart:
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2020-06-02T11:55:00.70151907Z"
    description: A Helm chart for Kubernetes
    digest:
2abb7e97678c65a5c671dc20f3c54d9e48834810304b77f77aa95d4d0bdace2b
    name: mychart
    type: application
    urls:
    - mychart-0.2.0.tgz
    version: 0.2.0
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2020-06-02T11:55:00.700913049Z"
    description: A Helm chart for Kubernetes
    digest:
64be7ad4e2ab08a57229c83f7f6b34f324a5a16305817d549052d20c69633ee0
    name: mychart
    type: application
    urls:
    - mychart-0.1.0.tgz
    version: 0.1.0
generated: "2020-06-02T11:55:00.700108583Z"
```

In some instances you might not have access to all of the chart package (`.tgz`) files at the time which you are trying to generate the index.

For example, you might upload charts to your chart repository storage as part of a CI/CD pipeline. Instead of downloading every single chart version to generate the index, you can use the current version of the index and simply append to it. This can be achieved using the **`--merge`** flag.

### Step 12

To demonstrate this, let's create a second chart called **`otherchart`**:

```
$ helm create otherchart
Creating otherchart
```

### Step 13

Next, we will create a new workspace directory called **`workspace`** and package the new chart into it:

```
$ mkdir -p workspace/
$ helm package otherchart/ -d workspace/
Successfully packaged chart and saved it to:
workspace/otherchart-0.1.0.tgz
```

### Step 14

Finally, run **`helm repo index`** using the **`--merge`** option to build a new repository index containing the new chart package based on the existing file:

```
$ helm repo index --merge lab/index.yaml workspace/
```

### Step 15

Now if we inspect this new index, we will see all three (3) chart versions (**`mychart-0.1.0`**, **`mychart-0.2.0`**, and **`otherchart-0.1.0`**):

```
$ cat workspace/index.yaml
apiVersion: v1
entries:
  mychart:
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2020-06-02T11:55:00.70151907Z"
    description: A Helm chart for Kubernetes
    digest:
2abb7e97678c65a5c671dc20f3c54d9e48834810304b77f77aa95d4d0bdace2b
```

```
    name: mychart
    type: application
    urls:
    - mychart-0.2.0.tgz
    version: 0.2.0
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2020-06-02T11:55:00.700913049Z"
    description: A Helm chart for Kubernetes
    digest:
64be7ad4e2ab08a57229c83f7f6b34f324a5a16305817d549052d20c69633ee0
    name: mychart
    type: application
    urls:
    - mychart-0.1.0.tgz
    version: 0.1.0
  otherchart:
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2020-06-02T12:13:06.685534862Z"
    description: A Helm chart for Kubernetes
    digest:
660492730dd4dc154dd44a7b4580abdd1d7f915bb2c04d13559f24716161a9b1
    name: otherchart
    type: application
    urls:
    - otherchart-0.1.0.tgz
    version: 0.1.0
generated: "2020-06-02T12:13:06.684391664Z"
```

## Standing Up a Web Server with Python

The only thing left to do in hosting a chart repository is to leverage a web server to host `index.yaml` and the chart `.tgz` files.

In a production environment, you might use a performant web server such as Nginx: https://nginx.org/.
For the sake of this lab, we will simply use Python's built-in, command-line web server (Python 3).

Step 16

Prior to starting the web server, make sure all files are in the proper location. Let's move the `.tgz` packages for the `mychart` chart into the `workspace` directory which contains our most up-to-date index file:

```
$ mv lab/*.tgz workspace/
$ ls workspace/
index.yaml  mychart-0.1.0.tgz  mychart-0.2.0.tgz  otherchart-0.1.0.tgz
```

Next, open a new terminal window in the current directory.

Step 17

To start a Python web server to statically serve these files at http://localhost:8000 (http://127.0.0.1:8000), open a new terminal window to the current directory and run the following commands:

```
$ cd workspace/
$ python3 -m http.server --bind 127.0.0.1 8000
Serving HTTP on 127.0.0.1 port 8000 (http://127.0.0.1:8000/) ...
```

*Note: If at any time you wish to stop the web server, just use CTRL+C.*

Your chart repository is now ready to be used with Helm.

## Using Your Chart Repository with Helm

Open the original terminal (the one not running your web server).

Now that our chart repository is up-and-running, we can start to use it from the Helm CLI.

Step 18

The first step is to add our chart repository as a new local repository with a unique identifier (`lab`):

```
$ helm repo add lab http://localhost:8000
"lab" has been added to your repositories
```

As we do this, in our other terminal running the Python web server, you should see the incoming request for `index.yaml` being logged:

```
127.0.0.1 - - [02/Jun/2020 12:30:03] "GET /index.yaml HTTP/1.1" 200 -
```

The repository index was downloaded by Helm and cached locally.

### Step 19

Next, let's install one of our charts from the repo (**mychart**):

```
$ helm install lab-demo lab/mychart
NAME: lab-demo
LAST DEPLOYED: Thu Jul  2 12:34:41 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l
"app.kubernetes.io/name=mychart,app.kubernetes.io/instance=lab-demo"
-o jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl --namespace default port-forward $POD_NAME 8080:80
```

If we look again at the logs of our web server, we should see another incoming request, this time for **mychart-0.2.0.tgz**:

```
127.0.0.1 - - [02/Jun/2020 12:34:41] "GET /mychart-0.2.0.tgz HTTP/1.1"
200 -
```

Why did Helm choose to use version **0.2.0** of the **mychart** chart vs. version **0.1.0**? This is because Helm is able to compare semantic versions and automatically choose the latest chart if no specific version is specified.

### Step 20

If we wanted to, we could deploy the old version of the **mychart** chart by specifying the **--version** flag:

```
$ helm install lab-demo-010 lab/mychart --version 0.1.0
NAME: lab-demo-010
LAST DEPLOYED: Thu Jul  2 12:39:05 2020
NAMESPACE: default
STATUS: deployed
```

```
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l
"app.kubernetes.io/name=mychart,app.kubernetes.io/instance=lab-demo-01
0" -o jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl --namespace default port-forward $POD_NAME 8080:80
```

This time we should see an incoming request in our web server logs for **mychart-0.1.0.tgz**:

```
127.0.0.1 - - [02/Jun/2020 12:39:05] "GET /mychart-0.1.0.tgz HTTP/1.1"
200 -
```

Next, let's see what happens when we update the repository index again.

### Step 21

Make another modification to the **mychart** chart, bumping the version to **0.3.0**:

```
$ sed -i 's/^version:.*$/version: 0.3.0/' mychart/Chart.yaml
```

### Step 22

Next, package it into the **workspace** directory and regenerate the index:

```
$ helm package mychart/ -d workspace/
Successfully packaged chart and saved it to:
workspace/mychart-0.3.0.tgz
$ helm repo index workspace/
```

### Step 23

Now our repository index should contain a total of four (4) entries (**mychart-0.1.0**, **mychart-0.2.0**, **mychart-0.3.0**, and **otherchart-0.1.0**):

```
$ cat workspace/index.yaml | grep version: | wc -l
4
```

So our chart repository has an update available for us. Our Helm release we installed titled **lab-demo** is based on version **0.2.0** of the **mychart** chart, but now there is a **0.3.0** available.

Step 24

Let's try to make the upgrade:

```
$ helm upgrade lab-demo lab/mychart --version 0.3.0
Error: failed to download "lab/mychart" (hint: running `helm repo
update` may help)
```

What went wrong? If we inspect the logs of our Python web server, we see no request for `mychart-0.3.0.tgz` as we might expect.

The problem is that we did not update the locally-cached version of the repository index. When we run **`helm install`**, Helm is determining available versions based on the contents of the `index.yaml` file that was downloaded when we ran **`helm repo add`**.

Step 25

If we want to update the local version of the repository index, we need to run **`helm repo update`**:

```
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "lab" chart repository
Update Complete. ⎈ Happy Helming!⎈
```

Now if we look at the web server logs, we should see an additional request for `index.yaml`:

```
127.0.0.1 - - [02/Jun/2020 12:52:16] "GET /index.yaml HTTP/1.1" 200 -
```

Step 26

Now we can retry our original command to upgrade to version `0.3.0`:

```
$ helm upgrade lab-demo lab/mychart --version 0.3.0
Release "lab-demo" has been upgraded. Happy Helming!
NAME: lab-demo
LAST DEPLOYED: Thu Jun  2 12:53:25 2020
NAMESPACE: default
STATUS: deployed
REVISION: 2
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l
```

```
"app.kubernetes.io/name=mychart,app.kubernetes.io/instance=lab-demo"
-o jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl --namespace default port-forward $POD_NAME 8080:80
```

Nice! We were able to download and install version `0.3.0` since we fetched the newer version of the index which contains it.

If we inspect out web server logs one last time, we should see the incoming request for:

```
127.0.0.1 - - [02/Jun/2020 12:53:25] "GET /mychart-0.3.0.tgz HTTP/1.1"
200
```

That's it! You should now have a thorough understanding of how to operate chart repositories.

### Step 27

Lastly, let's do some cleanup. Exit the Python web server using CTRL+C, and delete the Helm releases we created:

```
$ helm delete lab-demo lab-demo-010
release "lab-demo" uninstalled
release "lab-demo-010" uninstalled
```