# Kubernetes Foundations Documentation

***Release 1.0.1***

**CNA Education Team - VMWare, Inc.**

**Sep 27, 2019**

# CONTENTS:

CHAPTER

ONE

# LAB 01: CONTAINERIZE APPLICATIONS

## 1.1 Environment Overview

Your lab environment is hosted with a cloud server instance. This instance is accessible by clicking the "My Lab" icon on the left side of the Strigo web page. You can access the terminal session using the built-in interface provided by Strigo or feel free to SSH directly as well (instructions will appear in the terminal).

The appropriate kubernetes related tools (docker, kubectl, kubernetes, etc.) have already been installed ahead of time.

## 1.2 Prepare Your Lab

### 1.2.1 Step 1: Start Kubernetes and your Docker Registry

```
k8s-start
```

## 1.3 Build Docker image for your frontend application

### 1.3.1 Step 1: Download the gowebapp code under your home directory and untar the file

```
cd ~/

wget https://s3.eu-central-1.amazonaws.com/heptio-edu-static/foundations/gowebapp.tar.
↪gz

tar -zxvf gowebapp.tar.gz
```

### 1.3.2 Step 2: Write Dockerfile for your frontend application

```
cd $HOME/gowebapp/gowebapp
```

Create a file named Dockerfile in this directory for the frontend Go application. Use vi or any preferred text editor. The template below provides a starting point for defining the contents of this file.

**Note:** Replace TODO comments with the appropriate commands

Dockerfile-gowebapp

```
#TODO --- Define this image to inherit from the "ubuntu" base image
#https://hub.docker.com/_/ubuntu
#https://docs.docker.com/engine/reference/builder/#from

EXPOSE 80

#TODO --- Copy source code in the ./code directory into /opt/gowebapp
#https://docs.docker.com/engine/reference/builder/#copy

#TODO --- Copy the application config in the ./config directory into
#/opt/gowebapp/config

WORKDIR /opt/gowebapp/

#TODO --- Define an entrypoint for this image which executes the gowebapp
#application (/opt/gowebapp/gowebapp) when the container starts
#https://docs.docker.com/engine/reference/builder/#entrypoint
```

### 1.3.3 Step 3: Build gowebapp Docker image locally

```
cd $HOME/gowebapp/gowebapp
```

Build the gowebapp image locally. Make sure to include "." at the end. Make sure the build runs to completion without errors. You should get a success message.

```
docker build -t gowebapp:v1 .
```

## 1.4 Build Docker image for your backend application

### 1.4.1 Step 1: Write Dockerfile for your backend application

```
cd $HOME/gowebapp/gowebapp-mysql
```

Create a file named Dockerfile in this directory for the backend MySQL database application. Use vi or any preferred text editor. The template below provides a starting point for defining the contents of this file.

**Note:** Replace TODO comments with the appropriate commands

Dockerfile-gowebapp-mysql

```
#TODO --- Define this image to inherit from the "mysql" version 5.6 base image
#https://hub.docker.com/_/mysql/
#https://docs.docker.com/engine/reference/builder/#from

#TODO --- Investigate the "Initializing a Fresh Instance" instructions for the mysql
```

(continues on next page)

(continued from previous page)

```
#parent image, and copy the local gowebapp.sql file to the proper container directory
#to be automatically executed when the container starts up
#https://hub.docker.com/_/mysql/
#https://docs.docker.com/engine/reference/builder/#copy
```

### 1.4.2 Step 2: Build gowebapp-mysql Docker image locally

```
cd $HOME/gowebapp/gowebapp-mysql
```

Build the gowebapp-mysql image locally. Make sure to include "." at the end. Make sure the build runs to completion without errors. You should get a success message.

```
docker build -t gowebapp-mysql:v1 .
```

## 1.5 Run and test Docker images locally

Before deploying to Kubernetes, let's run and test the Docker images locally, to ensure that the frontend and backend containers run and integrate properly.

### 1.5.1 Step 1: Create Docker user-defined network

To facilitate cross-container communication, let's first define a user-defined network in which to run the frontend and backend containers:

```
docker network create gowebapp
```

### 1.5.2 Step 2: Launch frontend and backend containers

Next, let's launch a frontend and backend container using the Docker CLI.

First, we launch the database container, as it will take a bit longer to startup, and the frontend container depends on it. Notice how we are injecting the database password into the MySQL configuration as an environment variable:

```
docker run --net gowebapp --name gowebapp-mysql --hostname gowebapp-mysql -d -e MYSQL_
↪ROOT_PASSWORD=mypassword gowebapp-mysql:v1
```

**Warning:** Wait at least 20 seconds after starting the backend before attempting to start the frontend. Right now, the frontend will crash if the backend is not ready.

Now launch a frontend container, mapping the container port 80 - where the web application is exposed - to port 9000 on the host machine:

```
docker run -p 9000:80 --net gowebapp -d --name gowebapp --hostname gowebapp␣
↪gowebapp:v1
```

### 1.5.3 Step 3: Test the application locally

Now that we've launched the application containers, let's try to test the web application locally. You should be able to access the application at http://<EXTERNAL-IP>:9000.

**Note:** To get the EXTERNAL-IP of your host, run the command `lab-info` in your lab terminal

Create an account and login. Write something on your Notepad and save it. This will verify that the application is working and properly integrates with the backend database container.

### 1.5.4 Step 4: Inspect the MySQL database

Let's connect to the backend MySQL database container and run some queries to ensure that application persistence is working properly:

```
docker exec -it gowebapp-mysql mysql -u root -pmypassword gowebapp
```

Once connected, run some simple SQL commands to inspect the database tables and persistence:

```
#Simple SQL to navigate
SHOW DATABASES;
USE gowebapp;
SHOW TABLES;
SELECT * FROM <table_name>;
exit;
```

### 1.5.5 Step 5: Cleanup application containers

When we're finished testing, we can terminate and remove the currently running frontend and backend containers from our local machine:

```
docker rm -f gowebapp gowebapp-mysql
```

## 1.6 Create and push Docker images to Docker registry

### 1.6.1 Step 1: Tag images to target another registry

We are finished testing our images. We now need to push our images to an image registry so our Kubernetes cluster can access them. First, we need to tag our Docker images to use the registry in your lab environment:

```
docker tag gowebapp:v1 localhost:5000/gowebapp:v1

docker tag gowebapp-mysql:v1 localhost:5000/gowebapp-mysql:v1
```

### 1.6.2 Step 2: publish images to the registry

```
docker push localhost:5000/gowebapp:v1
docker push localhost:5000/gowebapp-mysql:v1
```

## 1.7 Lab 01 Conclusion

Congratulations! By containerizing your application components, you have taken the first important step toward deploying your application to Kubernetes.

CHAPTER

TWO

# LAB 02: DEPLOY APPLICATIONS USING KUBERNETES

## 2.1 Getting Started with kubectl

### 2.1.1 Step 1: introduction to kubectl

By executing kubectl you will get a list of options you can utilize kubectl for. kubectl allows you to interact with the Kubernetes API server.

```
kubectl
```

### 2.1.2 Step 2: use kubectl to understand an object

Use explain to get documentation of various resources. For instance pods, nodes, services, etc.

```
kubectl explain pods
```

### 2.1.3 Step 3: understand kubeconfig

kubectl does not require us to enter information such as a password or the URL of our Kubernetes API server. Such client-side information is defined in the kubeconfig file:

```
cat ~/.kube/config
```

```
# EXAMPLE OUTPUT

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://172.31.34.173:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
```

(continues on next page)

(continued from previous page)

```
users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

- A cluster contains endpoint data for a kubernetes cluster.
- A user defines client credentials for authenticating to a kubernetes cluster.
- A context defines a named cluster,user,namespace tuple which is used to send requests to the specified cluster using the provided authentication info and namespace.

You can also run `kubectl config view` to view kubeconfig settings as well.

### 2.1.4 Step 4: get more information on an object

Get a list of available resource types:

```
kubectl api-resources
```

### 2.1.5 Step 5: autocomplete

Use TAB to autocomplete:

```
kubectl describe <TAB> <TAB>
```

## 2.2 Create Service Object for MySQL

### 2.2.1 Step 1: define a Kubernetes Service object for the backend MySQL database

```
cd $HOME/gowebapp/gowebapp-mysql
```

Use your preferred text editor to create a file called gowebapp-mysql-service.yaml. Follow instructions below to populate gowebapp-mysql-service.yaml.

If you need help, please see reference: https://kubernetes.io/docs/concepts/services-networking/service/#defining-a-service

**Note:** Replace TODO comments with the appropriate commands

gowebapp-mysql-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  #TODO: give the Service a name: gowebapp-mysql
  labels:
    #TODO: give the Service a label: app: gowebapp-mysql
    #TODO: give the Service a label: tier: backend
```

(continues on next page)

(continued from previous page)

```
spec:
  type: ClusterIP
  ports:
  - #TODO: expose port 3306
    targetPort: 3306
  selector:
    #TODO: define a selector: app: gowebapp-mysql
    #TODO: define a selector: tier: backend
```

### 2.2.2 Step 2: create a Service defined above

Use kubectl to create the service defined above

```
kubectl apply -f gowebapp-mysql-service.yaml
```

### 2.2.3 Step 3: test to make sure the Service was created

```
kubectl get service -l "app=gowebapp-mysql"
```

## 2.3 Create Deployment Object for MySQL

### 2.3.1 Step 1: define a Kubernetes Deployment object for the backend MySQL database

```
cd $HOME/gowebapp/gowebapp-mysql
```

Use your preferred text editor to create a file called gowebapp-mysql-deployment.yaml. Follow instructions below to populate gowebapp-mysql-deployment.yaml.

If you need help, please see reference: https://kubernetes.io/docs/concepts/workloads/controllers/deployment

**Note:** Replace TODO comments with the appropriate commands

gowebapp-mysql-deployment.yaml

```
apiVersion: apps/v1
#TODO: define the kind of object as Deployment
metadata:
  #TODO: give the deployment a name name: gowebapp-mysql
  labels:
    #TODO: give the Deployment a label: app: gowebapp-mysql
    #TODO: give the Deployment a label: tier: backend
spec:
  #TODO: set replica to one
  strategy:
    #TODO: define the type of strategy as Recreate
  selector:
    matchLabels:
```

(continues on next page)

(continued from previous page)

```
      app: gowebapp-mysql
      tier: backend
  template:
    metadata:
      labels:
        app: gowebapp-mysql
        tier: backend
    spec:
      containers:
      - name: gowebapp-mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: mypassword
        image: localhost:5000/gowebapp-mysql:v1
        ports:
        - #TODO: define the container port as 3306
```

### 2.3.2 Step 2: create the Deployment defined above

Use kubectl to create the service defined above

```
kubectl apply -f gowebapp-mysql-deployment.yaml
```

### 2.3.3 Step 3: test to make sure the Deployment was created

```
kubectl get deployment -l "app=gowebapp-mysql"
```

## 2.4 Create Service object for frontend application: gowebapp

### 2.4.1 Step 1: define a Kubernetes Service object for the frontend gowebapp

```
cd $HOME/gowebapp/gowebapp
```

Use your preferred text editor to create a file called gowebapp-service.yaml. Follow instructions below to populate gowebapp-service.yaml.

If you need help, please see reference: https://kubernetes.io/docs/concepts/services-networking/service/#defining-a-service

**Note:** Replace TODO comments with the appropriate commands

gowebapp-service.yaml

```
apiVersion: v1
#TODO: define the kind of object as Service
metadata:
  #TODO: give the Service a name: gowebapp
  labels:
```

(continues on next page)

(continued from previous page)

```
    #TODO: give the Service a label: app: gowebapp
    #TODO: give the Service a label: tier: frontend
spec:
  #TODO: Set service type to NodePort
  ports:
  - #TODO: expose port 80
  selector:
    #TODO: define a selector: app: gowebapp
    #TODO: define a selector: tier: frontend
```

### 2.4.2 Step 2: create a Service defined above

Use kubectl to create the service defined above

```
kubectl apply -f gowebapp-service.yaml
```

Step 3: test to make sure the Service was created

```
kubectl get service -l "app=gowebapp"
```

## 2.5 Create Deployment object for gowebapp

### 2.5.1 Step 1: define a Kubernetes Deployment object for the frontend gowebapp

```
cd $HOME/gowebapp/gowebapp
```

Use your preferred text editor to create a file called gowebapp-deployment.yaml. Follow instructions below to populate gowebapp-deployment.yaml.

If you need help, please see reference: https://kubernetes.io/docs/concepts/workloads/controllers/deployment

**Note:** Replace TODO comments with the appropriate commands

gowebapp-deployment.yaml

```
apiVersion: apps/v1
#TODO: define the kind of object as Deployment
metadata:
  #TODO: give the deployment a name name: gowebapp
  labels:
    #TODO: give the Deployment a label: app: gowebapp
    #TODO: give the Deployment a label: tier: frontend
spec:
  #TODO: set replica to 2
  selector:
    matchLabels:
      app: gowebapp
      tier: frontend
  template:
    metadata:
```

(continues on next page)

(continued from previous page)

```
      labels:
        app: gowebapp
        tier: frontend
    spec:
      containers:
      - name: gowebapp
        env:
        - #TODO: define name as MYSQL_ROOT_PASSWORD
          #TODO: define value as mypassword
        image: localhost:5000/gowebapp:v1
        ports:
        - #TODO: define the container port as 80
```

### 2.5.2 Step 2: create the Deployment defined above

Use kubectl to create the Deployment defined above

```
kubectl apply -f gowebapp-deployment.yaml
```

### 2.5.3 Step 3: test to make sure the Deployment was created

```
kubectl get deployment -l "app=gowebapp"
```

## 2.6 Test Your Application

### 2.6.1 Step 1: Access gowebapp through the NodePort service

Access your application at the NodePort Service endpoint: http://<EXTERNAL-IP>:<NodePort>.

---

**Note:** To get the EXTERNAL-IP of your host, run the command `lab-info` in your lab terminal

---

---

**Note:** To get the NodePort for your service, run the command `kubectl get svc gowebapp` in your lab terminal

---

```
*** EXAMPLE OUTPUT ****

$ kubectl get svc gowebapp

NAME        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)         AGE
gowebapp    NodePort    10.107.169.105   <none>        80:30500/TCP    2m

# In this example, the NodePort is 30500
```

You can now sign up for an account, login and use the Notepad.

**Warning:** Note: Your browser may cache an old instance of the gowebapp application from previous labs. When the webpage loads, look at the top right. If you see 'Logout', click it. You can then proceed with creating a new test account.

## 2.7 Lab 02 Conclusion

Congratulations! You have successfully deployed your applications using Kubernetes!

CHAPTER

THREE

# LAB 03: KUBERNETES TROUBLESHOOTING

When deploying to Kubernetes, it's important to understand the relationship between objects and how to troubleshoot when issues arise with your deployments. In this lab, we'll explore some of those relationships as well as various troubleshooting steps you can follow when things go wrong.

## 3.1 Exploring the Event Log

Kubernetes has an event log which captures a vast amount of information regarding changes, warnings, and errors on the cluster. If you encounter problems after a successful `kubectl apply`, the event log will often contain useful troubleshooting information.

**Note:** By default, the Kubernetes cluster retains one hour of event log history. The retention period can be adjusted by the cluster administrator. Ideally, the event log should be shipped to a centralized logging system.

### 3.1.1 Step 01: View the event log

You can quickly see the entire contents of the event log using `kubectl get events`.

```
kubectl get events
```

### 3.1.2 Step 02: Watch the event log

You can watch the event log in real time by adding a `-w` to the end of the command. It's helpful to run this in a separate terminal before running any `kubectl apply` commands. This will allow you to watch the events related to the `apply` process in real time.

```
kubectl get events -w

# Use ctrl + c to return to the shell
```

### 3.1.3 Step 03: View the event log for a specific object

If you only want to see the event log entries related to a specific object, you can do so by running `kubectl describe <resource_type> <resource_name>`. At the end of the output, you will see the most recent event log entries for that object.

```
kubectl describe node master

# The event log entries will be at the very end of the output.
```

## 3.2 Exploring a Deployment

When you created the gowebapp and gowebapp-mysql deployments earlier, Kubernetes automatically created a series of additional objects eventually resulting in the creation of containers running the application.

### 3.2.1 Step 01: List and describe the deployment

Let's start by getting a list of the deployments currently on the Kubernetes cluster.

```
kubectl get deployments
```

And now let's get more detail about the gowebapp frontend deployment.

```
kubectl describe deployment gowebapp
```

The event log at the end of that output is especially helpful. It will tell you the most recent cluster events related to that object. In this case we can see that the deployment controller scaled up a new replica set for our application.

### 3.2.2 Step 02: List and describe the replica set

Deployments create replica sets. Replica sets are used to make it easy to roll from one version of a deployment to another. Each time we update a deployment, a new replica set is created that contains the latest configuration.

Let's take a look at the replica sets currently on the Kubernetes cluster.

```
kubectl get replicasets
```

You can see that for each deployment, there is a corresponding replica set. Now let's get more detail about the gowebapp replicaset.

```
# You'll need the name of the gowebapp replicaset from the previous command
kubectl describe rs <replica_set_name>
```

**Note:** In the above command, we used the abbreviation `rs` instead of `replicasets`. Many resource types have abbreviated forms. For a complete list, you can run `kubectl api-resources`.

The event log for the replica set shows that it created two pods.

### 3.2.3 Step 03: List and describe the pods

You've now seen that deployments create replica sets, and replica sets create pods. Now let's look at the pods. Let's start by getting a list of the deployments currently on the Kubernetes cluster.

```
kubectl get pods
```

You can see that there are three pods. Two pods for the gowebapp frontend, and one for the MySQL backend. This corresponds to the `replicas` field we set in each deployment.

Now let's get more details about one of the gowebapp frontend pods

```
# You'll need the name of one of the gowebapp pods from the previous command
kubectl describe pod <pod_name>
```

The event log for the pod shows that the pod was scheduled to a node, and that the node created and started a container.

### 3.2.4 Step 04: Get pod logs

Now that we know how to get a list of pods, we can use a pod name to get its logs. This is the equivalent of running `docker logs <container_id>`. If your application streams logs to stdout and stderr, they will be accessible via this command.

```
# If your pod has only one container
kubectl logs <pod_name>

# If your pod has more than one container
kubectl logs <pod_name> -c <container_name>
```

## 3.3 Exploring a Service

Just like with a deployment, a service has relationships with other objects that are important to understand. Let's explore this now.

### 3.3.1 Step 01: List and describe the service

Let's start by getting a list of the services currently on the Kubernetes cluster.

```
kubectl get services
```

And now let's get more detail about the gowebapp frontend service.

```
kubectl describe service gowebapp
```

One of the notable fields is `Endpoints`. Each endpoint is the IP address of a pod that is backing that service. In this case, we should see two endpoints, one for each of the two gowebapp frontend pods created by the gowebapp deployment.

We can also see a list of endpoints for each service by running the following command.

```
kubectl get endpoints
```

If you see fewer endpoints than you expect. It's a good indicator that there's an issue with either your pods or service.

## 3.4 Troubleshooting a Deployment

Now that you've seen some of the commands used for exploring objects and their relationships in Kubernetes, let's simulate a failure and see how to troubleshoot it.

### 3.4.1 Step 01: Modify the gowebapp deployment

Let's start by modifying the gowebapp deployment to deliberately introduce a failure.

```
cd $HOME/gowebapp/gowebapp
```

Use your preferred text editor to edit gowebapp-deployment.yaml. Change the `image:` field from `localhost:5000/gowebapp:v1` to `localhost:5000/gowebapp:v99`.

The image `localhost:5000/gowebapp:v99` doesn't exist, so this should cause a number of failures.

### 3.4.2 Step 02: Apply the updated deployment

Use kubectl to apply the updated deployment

```
kubectl apply -f gowebapp-deployment.yaml
```

### 3.4.3 Step 03: Explore the results

Run the following commands to see what information Kubernetes provides when there's a problem with a deployment.

```
# Note that gowebapp deployment no longer shows 2 pods "UP-TO-DATE"
kubectl get deployments

# Note that there's a second, new replica set for gowebapp, but it shows '0' pods␣
↪ready
kubectl get rs

# Note that you still have 2 endpoints available for the gowebapp service. Kubernetes␣
↪won't
# destroy the old pods until the new ones are healthy. This has prevented the␣
↪application from
# failing despite the configuration error.
kubectl get ep

# Note that there's one new pod for gowebapp that is failing to start. The status is
# ``ImagePullBackOff`` which indicates that Docker is unable to pull the image␣
↪defined in the
# deployment.
kubectl get pods

# Finally, if you describe the pod and review its event log, you'll see an error␣
↪message that
# states ``Failed to pull image "localhost:5000/gowebapp:v99"``.
kubectl describe pod <name_of_failed_pod>
```

### 3.4.4 Step 04: Fix the Problem

```
cd $HOME/gowebapp/gowebapp
```

Use your preferred text editor to edit gowebapp-deployment.yaml. Change the `image:` field back to `localhost:5000/gowebapp:v1`. Then re-apply gowebapp-deployment.yaml.

```
kubectl apply -f gowebapp-deployment.yaml
```

### 3.4.5 Step 05: Verify the Fix

Re-run the commands from step 03 and verify that everything has returned to normal.

## 3.5 Troubleshooting a Service

Now let's look at how we can troubleshoot issues related to services. A typical issue you will encounter is a mismatch between a service's selectors and the target pods' labels.

### 3.5.1 Step 01: Modify the gowebapp service

Let's start by modifying the gowebapp service to deliberately introduce a failure.

```
cd $HOME/gowebapp/gowebapp
```

Use your preferred text editor to edit gowebapp-service.yaml. Change the selector `app: gowebapp` to `app: pythonwebapp`.

> **Warning:** Ensure you modify `app: gowebapp` in the `selector:` section, not the `labels:` section.

### 3.5.2 Step 02: Apply the updated service

Use kubectl to apply the updated service

```
kubectl apply -f gowebapp-service.yaml
```

### 3.5.3 Step 03: Explore the results

The first, most obvious result of introducing a failure here is that gowebapp will no longer be accessible from a browser. Go ahead and confirm this.

Next, run the following commands to see what information Kubernetes provides when there's a mismatch between a service's selectors and the target pods' labels.

```
# Note that just "getting" gowebapp gives no idications of a problem.
kubectl get service

# Note that the endpoints field is blank. This means the service can't find any pods␣
↪to send
# requests to.
kubectl describe service gowebapp

# You can also list the endpoints by running the following command. The ``ENDPOINTS``␣
↪field here
# should also be blank.
```

(continues on next page)

(continued from previous page)

```
kubectl get ep gowebapp

# Note the list of selectors
kubectl describe service gowebapp

# Get a list of pods
kubectl get pods

# Note the labels applied to the pod don't match the selectors from the service
kubectl describe pod <name_of_a_gowebapp_pod>
```

### 3.5.4 Step 04: Fix the Problem

```
cd $HOME/gowebapp/gowebapp
```

Use your preferred text editor to edit gowebapp-service.yaml. Change the selector back to `app: gowebapp`. Then re-apply gowebapp-service.yaml.

```
kubectl apply -f gowebapp-service.yaml
```

### 3.5.5 Step 05: Verify the Fix

Re-run the commands from step 03 and verify that everything has returned to normal.

## 3.6 Keep Exploring

You can use `kubectl get` and `kubectl describe` on any resource type in Kubernetes. You can run `kubectl api-resources` to get a list of all the available resource types. As we introduce more resource types throughout the remainder of the course, run these commands to see what information each of them provides.

## 3.7 Lab 03 Conclusion

In this lab we explored the relationships between Deployments, ReplicaSets, Pods, Services, and Endpoints. We then explored various commands that can be used to explore and troubleshoot problems with those resources.

CHAPTER

FOUR

# LAB 04: EXPOSING SERVICES

In lab 02, we exposed the gowebapp frontend using a NodePort service. In production, users will expect to access a web service on the standard http(s) ports, 80 and 443. With Kubernetes we can accomplish this with either `serviceType: LoadBalancer` (layer 4) or an `Ingress` (layer 7). In this lab, we will modify the application to expose the frontend using an Ingress Controller and Ingress Resource, which will allow users to connect to the application on 80 and 443.

## 4.1 Modify gowebapp Service

Since we will be exposing the frontend using an Ingress controller, we no longer need to expose it using a NodePort service. We still need a ClusterIP service to provide loadbalancing for the pods. Let's modify the service to change it to `type: ClusterIP`.

### 4.1.1 Step 01: Delete the gowebapp service

Kubernetes will not allow us to change the type of a running service. In this case, we will need to delete and recreate it. Start by deleting the existing gowebapp frontend service.

```
kubectl delete service gowebapp
```

### 4.1.2 Step 02: Modify gowebapp-service.yaml

```
cd $HOME/gowebapp/gowebapp
```

Open `gowebapp-service.yaml` in your editor and change the service type to `ClusterIP`

### 4.1.3 Step 03: Update the service

Use kubectl to update the service

```
kubectl apply -f gowebapp-service.yaml
```

### 4.1.4 Step 04: Verify the service type has changed

```
kubectl get service -l "app=gowebapp"
```

The service type should now be ClusterIP.

## 4.2 Deploy an Ingress Controller

An ingress controller acts a central point for external traffic to enter a cluster. It's essentially a layer 7 reverse proxy that Kubernetes manages for you. In this class we will deploy an ingress controller based on NGINX.

### 4.2.1 Step 01: Deploy the Ingress Controller

`ingress-controller.yaml`

---

**Note:** The contents of `ingress-controller.yaml` are not displayed here. If you're curious, you can view the contents of the file on your lab machine after downloading it.

---

Download and apply the above yaml file.

```
cd $HOME/gowebapp/gowebapp

wget <url_of_yaml_above>

kubectl apply -f ingress-controller.yaml
```

### 4.2.2 Step 02: Verify the Ingress Controller is Working

Run the following command to verify the Ingress Controller pod is running

```
kubectl get pods -n ingress-nginx
```

Test access to the Ingress Controller by accessing the following URL in your browser: http://<EXTERNAL-IP>.

---

**Note:** To get the EXTERNAL-IP of your host, run the command `lab-info` in your lab terminal

---

The page should display `404 Not Found`. This is expected since we have not configured a default site for the ingress controller yet.

## 4.3 Create an Ingress Resource

### 4.3.1 Step 01: Define the ingress Resource

```
cd $HOME/gowebapp/gowebapp
```

Use your preferred text editor to create a file called `gowebapp-ingress.yaml`. Follow instructions below to populate gowebapp-ingress.yaml.

If you need help, please see reference: https://kubernetes.io/docs/concepts/services-networking/ingress/

**Note:** Replace TODO comments with the appropriate commands

gowebapp-ingress.yaml

```
apiVersion: extensions/v1beta1
#TODO: define the kind of object as Ingress
metadata:
  #TODO: give the Ingress a name: gowebapp
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
  labels:
    app: gowebapp
    tier: frontend
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          #TODO: set the backend service name to gowebapp
          servicePort: 80
```

**Note:** In a typical production use case, you would also specify a hostname for this site, for example, `gowebapp.com`. In this lab, we don't have access to public DNS, so we will omit the hostname. This will allow us to connect using the lab machine's public IP instead.

### 4.3.2 Step 02: Apply the ingress resource

```
kubectl apply -f gowebapp-ingress.yaml
```

### 4.3.3 Step 03: Test access to gowebapp

Run the following command to verify the Ingress resource has been created

```
kubectl get ingress -l "app=gowebapp"
```

Test access to the application by accessing the following URL in your browser: http://<EXTERNAL-IP>.

**Note:** To get the EXTERNAL-IP of your host, run the command `lab-info` in your lab terminal

Now, when you access the URL, the gowebapp application should appear.

## 4.4 Lab 04 Conclusion

Congratulations! You have successfully exposed your application to the internet using an ingress controller.

CHAPTER

FIVE

# LAB 05: DEPLOYMENT MANAGEMENT

## 5.1 Lab Goals

In this lab we'll work with resource requests and limits to ensure workloads have sufficient resources to operate. Then, we will work with liveness and readiness probes to proactively monitor the health of our applications.

## 5.2 Resource Requests

Resource requests are a way for us to let Kubernetes know the minimun required resources for a container in a pod to run. Kubernetes will use this to intelligently schedule a pod on the cluster by placing the pod only on a node with sufficient resources to satisfy the request.

### 5.2.1 Step 01: Add resource requests to gowebapp

Modify `gowebapp-deployment.yaml` to add resource requests for

- 256Mi of memory
- 250 millicpu (250m)

### 5.2.2 Step 02: Re-apply the gowebapp deployment

Use kubectl to update the gowebapp deployment.

```
kubectl apply -f gowebapp-deployment.yaml
```

### 5.2.3 Step 03: Verify the requests have been applied

You can verify that the resource requests were successfully applied by describing the deployment.

```
kubectl describe deployment gowebapp
```

## 5.3 Resource Limits

Resource limits define the maximum amount of resources a container in a pod should consume. If a container exceeds this limit, it may be terminated.

### 5.3.1 Step 01: Add resource limits to gowebapp

Modify `gowebapp-deployment.yaml` to add resource limits for

- 512Mi of memory
- 500 millicpu (500m)

### 5.3.2 Step 02: Re-apply the gowebapp deployment

Use kubectl to update the gowebapp deployment.

```
kubectl apply -f gowebapp-deployment.yaml
```

### 5.3.3 Step 03: Verify the limits have been applied

You can verify that the resource limits were successfully applied by describing the deployment.

```
kubectl describe deployment gowebapp
```

## 5.4 Probes

Liveness and readiness probes are health checks that Kubernetes can perform that proactively monitor the health of pods and the containers within.

- Liveness probes check if a container in a pod is still "alive". If the probe fails, the container will be restarted.
- Readiness probes check if a container in a pod is ready to serve requests from a service. If the probe fails, the pod will be removed as an endpoint for any matching services.

In this section we will configure both liveness and readiness probes. To do this we need to create some clever scenarios make our probes valid and to see them in action. We ultimately want to create the following three scenarios:

1. regular nginx deployment
2. nginx deployment that delays becoming **alive** and then delays more past that to become **ready**
3. nginx deployment that delays becoming **alive** and will periodically become **not ready** for a period of time

Scenario is a basic nginx deployment `nginx-scenario1.yaml`

Scenario #2 is a little trickier. It will need an `initContainer` to delay startup and something else to show up after nginx is up and alive. We encourage you to use what you know of Kubernetes and this course to figure out a way to make this happen.

If not we do have a solution you can use: `nginx-scenario2.yaml`

Scenario #3 is a variation of number #2 where we continually alternate the container from being ready to being not ready every 30 seconds for example. Again we encourage you to use what you know of Kubernetes and this course to figure out a way to make this happen.

If not we do have a solution you can use: `nginx-scenario3.yaml`

Clear out work from earlier in this lab by doing a `kubectl delete all --all` and then create the Deployment yaml files for the above scenarios. Test them and make sure they work before we add in the probes.

Take the above three yaml files created for each scenario and add probes to them accordingly. When doing so we are also going to need to override the following two options on the probes:

- `successThreshold: 1`
- `periodSeconds: 1`

This is to ensure the intervals work appropriately and we don't wait forever or fall into a timing issue.

Once the probes are added you can validate the probes are working correctly a few ways:

- run a `kubectl describe pod <pod_id>` and look at the events at the bottom
- look at the logs of the nginx container and verify the probe is invoking the 200s and 404s in 30 chunk blocks
- look at the containerStatuses section of a pod

```
kubectl get pod nginx-4247535729-vhjzc -o jsonpath='{range.status.
↪containerStatuses[*]}{@.name}{" is ready: "}{@.ready}{"\n"}'
```

## 5.5 Lab 05 Conclusion

Congratulations! You have successfully added resource requests/limits and probes to your application. These features are essential to ensuring that your application has the resources it requires and that Kubernetes can effectively monitor and address issues that may arrise with it.

CHAPTER

SIX

# LAB 01: CONTAINERIZE APPLICATIONS

Dockerfile-gowebapp

```
FROM ubuntu

COPY ./code /opt/gowebapp
COPY ./config /opt/gowebapp/config

EXPOSE 80

WORKDIR /opt/gowebapp/
ENTRYPOINT ["/opt/gowebapp/gowebapp"]
```

Dockerfile-gowebapp-mysql

```
FROM mysql:5.6

COPY gowebapp.sql /docker-entrypoint-initdb.d/
```

CHAPTER

SEVEN

# LAB 02: DEPLOY APPLICATIONS USING KUBERNETES

gowebapp-mysql-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: gowebapp-mysql
  labels:
    app: gowebapp-mysql
    tier: backend
spec:
  type: ClusterIP
  ports:
  - port: 3306
    targetPort: 3306
  selector:
    app: gowebapp-mysql
    tier: backend
```

gowebapp-mysql-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gowebapp-mysql
  labels:
    app: gowebapp-mysql
    tier: backend
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: gowebapp-mysql
      tier: backend
  template:
    metadata:
      labels:
        app: gowebapp-mysql
        tier: backend
    spec:
      containers:
      - name: gowebapp-mysql
        env:
```

(continues on next page)

(continued from previous page)

```
        - name: MYSQL_ROOT_PASSWORD
          value: mypassword
        image: localhost:5000/gowebapp-mysql:v1
        ports:
        - containerPort: 3306
```

gowebapp-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: gowebapp
  labels:
    app: gowebapp
    tier: frontend
spec:
  type: NodePort
  ports:
  - port: 80
  selector:
    app: gowebapp
    tier: frontend
```

gowebapp-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gowebapp
  labels:
    app: gowebapp
    tier: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: gowebapp
      tier: frontend
  template:
    metadata:
      labels:
        app: gowebapp
        tier: frontend
    spec:
      containers:
      - name: gowebapp
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: mypassword
        image: localhost:5000/gowebapp:v1
        ports:
        - containerPort: 80
```

CHAPTER

EIGHT

# LAB 03: KUBERNETES TROUBLESHOOTING

CHAPTER

NINE

# LAB 04: EXPOSING SERVICES

gowebapp-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: gowebapp
  labels:
    app: gowebapp
    tier: frontend
spec:
  type: ClusterIP
  ports:
  - port: 80
  selector:
    app: gowebapp
    tier: frontend
```

gowebapp-ingress.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: gowebapp
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
  labels:
    app: gowebapp
    tier: frontend
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: gowebapp
          servicePort: 80

```

CHAPTER

TEN

# LAB 05: DEPLOYMENT MANAGEMENT

## 10.1 Resource Requests

gowebapp-deployment-requests.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gowebapp
  labels:
    app: gowebapp
    tier: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: gowebapp
      tier: frontend
  template:
    metadata:
      labels:
        app: gowebapp
        tier: frontend
    spec:
      containers:
      - name: gowebapp
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: mypassword
        image: localhost:5000/gowebapp:v1
        ports:
        - containerPort: 80
        resources:
          requests:
            memory: "256Mi"
            cpu: "250m"
```

## 10.2 Resource Limits

gowebapp-deployment-limits.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gowebapp
  labels:
    app: gowebapp
    tier: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: gowebapp
      tier: frontend
  template:
    metadata:
      labels:
        app: gowebapp
        tier: frontend
    spec:
      containers:
      - name: gowebapp
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: mypassword
        image: localhost:5000/gowebapp:v1
        ports:
        - containerPort: 80
        resources:
          requests:
            memory: "256Mi"
            cpu: "250m"
          limits:
            memory: "512Mi"
            cpu: "500m"
```

## 10.3 Probes

nginx-scenario3-withProbes.yaml

```yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  selector:
    app: nginx
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
apiVersion: apps/v1
```

(continues on next page)

(continued from previous page)

```
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      volumes:
      - name: www
        emptyDir: {}
      containers:
      - name: nginx
        image: nginx:1.13.0
        ports:
        - containerPort: 80
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
        livenessProbe:
          successThreshold: 1
          periodSeconds: 1
          httpGet:
            path: /alive
            port: 80
        readinessProbe:
          successThreshold: 1
          periodSeconds: 1
          httpGet:
            path: /ready
            port: 80
      # alternate every minute the pod being "ready" by removing a file that would be used by a readiness probe
      - name: flapper
        image: busybox
        command: ["sh", "-c", "while true; do var=$((1-var)); if [ $var -eq 0 ];then echo OK > /usr/share/nginx/html/ready; else rm /usr/share/nginx/html/ready;fi; sleep 30; done"]
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
      initContainers:
      - name: slowness
        image: busybox
        command: ['sh', '-c', 'echo "Sleeping 10 seconds..." && sleep 20 && echo OK > /usr/share/nginx/html/alive']
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html

```