Kubernetes Foundations Documentation

Release 1.0.1

CNA Education Team - VMWare, Inc.

Sep 27, 2019

CONTENTS:

CHAPTER

ONE

LAB 06: RESOURCE ORGANIZATION

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
```

1.2.2 Step 2: Bootstrap Environment for Part 2

This will replay the labs from part one so we are ready to start the part two labs.

```
k8s-bootstrap-day2
```

1.3 Exploring Namespaces

Namespaces enable you to isolate objects within the same Kubernetes cluster. The isolation comes in the form of:

- Object & DNS Name Scoping
- Object Access Control
- Resource Quotas

1.3.1 Step 1: Define a Namespace

Start by defining a new Namespace.

```
cd $HOME
```

Create a file named `my-namespace.yaml` in this directory. 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

my-namespace.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  # TODO: give the namespace the name 'my-namespace'
```

1.3.2 Step 2: create the Namespace

Use kubectl to create the Namespace defined above

```
kubectl apply -f my-namespace.yaml
```

1.3.3 Step 3: verify the Namespace was created

Use kubectl to verify the Namespace was created

```
kubectl get namespaces
```

1.3.4 Step 4: List resources in Namespaces

Many objects in Kubernetes are created within a Namespace. By default, if no namespace is specified when creating a namespaced object, it will be placed in a namespace called 'default'.

Let's verify this by listing the pods in a few namespaces.

```
# There should be no pods in this namespace since we just created it
kubectl get pods --namespace my-namespace

# All of the gowebapp pods we created are in the 'default' namespace
kubectl get pods --namespace default

# Since the 'default' namespace is our current default namespace, the output of this␣
↪command
# should be the same as the last
kubectl get pods
```

1.4 DNS Namespacing

Namespaces also impact DNS resolution within the cluster. Pods can connect to services in their namespace using the service's short name. If a pod needs to connect to a service in a different Namespace, it must use the service's fully-qualified name.

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

$ kubectl get services

NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)     AGE
gowebapp          ClusterIP   10.104.159.241   <none>        80/TCP      8m11s
gowebapp-mysql    ClusterIP   10.100.107.166   <none>        3306/TCP    8m22s
kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP     11m
```

Let's test connecting to the `gowebapp` service using curl from within a pod three different ways:

- **within** the `default` namespace, using the short name, `gowebapp`
- **outside** the `default` namespace, using the short name, `gowebapp`
- **outside** `default` namespace, using the fully-qualified name `gowebapp.default.svc.cluster.local`

You can use the following two variant URL for testing with curl:

- http://gowebapp
- http://gowebapp.default.svc.cluster.local

The easiest way to "access a terminal" within a namespace is to launch a pod with an interactive terminal inside the desired namespace.

```
# In the 'default' namespace
kubectl run curl --image=radial/busyboxplus:curl -i --tty --rm

# In the 'my-namespace' namespace
kubectl run curl --namespace my-namespace --image=radial/busyboxplus:curl -i --tty --
↪rm
```

Use these commands to launch pods in both the `default` and `my-namepsace` namespaces. Then try curling the short and fully-qualified URLs above.

If successful you should see a response like the following

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">


    <meta name="description" content="">
    <meta name="keywords" content="">
    <meta name="author" content="">

    <title>Heptio Go Web App</title>

*** OUTPUT TRUNCATED ***
```

You should find that attempting to access a service in a different namespace, without using its fully-qualified name, should result in an error.

1.5 Changing Default Namespace

Right now, if we run any `kubectl` commands without specifying a namespace, the namespace `default` will be used. If desired, we can set a different namespace to be our default.

1.5.1 Step 01: view the kubectl configuration file

The `kubectl` configuration file, also called `kubeconfig` can be viewed in two ways.

```
kubectl config view

# or

cat ~/.kube/config
```

You'll notice that there are three sections:

- A list of clusters
- A list of users (credentials)
- A list of contexts that map users to clusters

1.5.2 Step 02: create a new context

In addition to mapping users to clusters, contexts are also where we can set a default namespace. Let's create a new context that uses the existing user and cluster, but defaults to the `my-namespace` namespace.

```
kubectl config set-context my-context --user kubernetes-admin --cluster kubernetes --
↪namespace my-namespace
```

1.5.3 Step 03: view the changes to kubeconfig

```
kubectl config view
```

You should see the new `my-context` context which includes `my-namespace` as it's default.

1.5.4 Step 04: use the new context

Next, we need to tell `kubectl` to use our new context.

```
kubectl config use-context my-context
```

At any time, you can see which context is currently active by running

```
kubectl config current-context
```

Or see a list of available contexts by running

```
kubectl config get-contexts
```

1.5.5 Step 05: deploy to my-namespace

Let's deploy a test deployment to `my-namespace`. Since it is now our default namespace, we shouldn't need to specify it manually.

```
kubectl run nginx --image nginx --replicas 3
```

1.5.6 Step 06: list the deployment and its pods

```
kubectl get deployments

kubectl get pods
```

1.5.7 Step 07: list the gowebapp deployment and its pods

We can still list the gowebapp deployment and pods, but we'll need to tell kubectl to target the namespace, `default`

```
kubectl get deployments --namespace default

# -n is short for --namespace
kubectl get pods -n default
```

1.5.8 Step 08: cleanup

Before continuing we'll need to clean up a few things.

Delete the nginx deployment

```
kubectl delete deployment nginx
```

Switch back to the default context

```
kubectl config use-context kubernetes-admin@kubernetes
```

1.6 kubectl Filtering with Labels

We can also filter queries with `kubectl` using labels.

For instance, we can list pods with the label `tier=frontend`.

```
kubectl get pods -l tier=frontend
```

In our small cluster with only a few pods and deployments, filtering using labels wont provide much benefit. In large production environments however, this feature is crucial for locating specific resources amongst hundreds or even thousands of pods and deployments.

1.7 Lab 06 Conclusion

Congratulations! You have successfully explored namespaces, managed your kubeconfig, and queried using labels.

CHAPTER

TWO

LAB 07: STORAGE & STATEFUL APPLICATIONS

2.1 Lab Goals

In this lab, you will be converting your MySQL deployment into a StatefulSet. You will statically provision a Persistent Volume that the StatefulSet will use. You will then demonstrate that the state of your MySQL database successfully persists after deleting and recreating the pod.

2.2 Provision a Persistent Volume (PV)

2.2.1 Step 1: Define a Persistent Volume

Define a Persistent Volume object to use with MySQL in a file named pv.yaml under $HOME/gowebapp/gowebapp-mysql

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

In this case, we will provision a volume that will leverage the local storage of our Kubernetes host. In a production environment, this will typically be remote storage (FC, iSCSI, EBS)

If you need help, please see reference: https://kubernetes.io/docs/concepts/storage/persistent-volumes/

Note: Replace TODO comments with the appropriate commands

pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv01
spec:
  storageClassName: local
  capacity:
    storage: # TODO: Set size to 5Gi
  accessModes:
  - ReadWriteOnce
  hostPath:
    path: /var/pv/pv01
```

2.2.2 Step 2: Create the Persistent Volume object

```
kubectl apply -f pv.yaml
```

2.2.3 Step 3: View the Persistent Volume

```
kubectl get pv pv01

kubectl describe pv pv01
```

2.3 Convert MySQL deployment to a StatefulSet

2.3.1 Step 1: Remove MySQL Deployment

Before converting the gowebapp-mysql deployment to a StatefulSet, you need to remove the existing deployment from the Kubernetes cluster.

```
kubectl delete deployment gowebapp-mysql
```

2.3.2 Step 2: Convert MySQL from Deployment to StatefulSet

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

Convert the MySQL deployment to a StatefulSet. You can use the following as a starting point.

Note: Replace TODO comments with the appropriate commands

gowebapp-mysql-sts.yaml

```
apiVersion: apps/v1
kind: # TODO: Set kind to StatefulSet
metadata:
  name: gowebapp-mysql
  labels:
    app: gowebapp-mysql
    tier: backend
spec:
  serviceName: # TODO: Set serviceName to gowebapp-mysql
  replicas: 1
  selector:
    matchLabels:
      app: gowebapp-mysql
      tier: backend
  template:
    metadata:
      labels:
        app: gowebapp-mysql
        tier: backend
    spec:
```

(continues on next page)

(continued from previous page)

```
      containers:
      - name: gowebapp-mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: mypassword
        image: localhost:5000/gowebapp-mysql:v1
        ports:
        - containerPort: 3306
        volumeMounts:
        - name: mysql-pv
          mountPath: # TODO: Set mountPath to /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-pv
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "local"
      resources:
        requests:
          storage: # TODO: Set storage request to 5Gi
```

2.3.3 Step 3: Create the MySQL StatefulSet

Warning: Ensure that you deleted the gowebapp-mysql-deployment (Step 1) before applying the new StatefulSet.

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

2.4 Test your application

2.4.1 Step 1: access your application

Access your application at the Ingress endpoint: http://<EXTERNAL-IP>

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

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.

2.5 Illustrate Failure and Recovery of Database State

2.5.1 Step 1: Find the name of the MySQL pod

```
kubectl get pod -l 'app=gowebapp-mysql'
```

2.5.2 Step 2: Kill the pod

The StatefulSet will automatically relaunch it.

```
kubectl delete pod <mysql_pod_name>
```

2.5.3 Step 3: Monitor StatefulSet

Monitor the StatefulSet until it's ready again (AVAILABLE turns from 0 to 1):

```
kubectl get sts gowebapp-mysql -w

# Press <ctrl> + c to return to the prompt
```

2.6 Confirm Persistent Volume Kept Application State

2.6.1 Step 1: Find the name of the new MySQL pod and inspect it

```
kubectl get pod -l 'app=gowebapp-mysql'

kubectl describe pod <mysql_pod_name>
```

You should see that the volume has been re-mounted to the new pod

2.6.2 Step 2: Ensure gowebapp can still access the pre-failure data

Access your application at the Ingress endpoint: http://<EXTERNAL-IP>

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

---

Test that you can still login using the username and password you created earlier in this lab.

2.7 Lab 07 Conclusion

Congratulations! You have successfully converted your MySQL deployment to a StatefulSet and demonstrated that the data survives pod replacement.

CHAPTER

THREE

LAB 08: DYNAMIC APPLICATION CONFIGURATION

3.1 Build new Docker image for your frontend application

3.1.1 Step 1: Update Dockerfile for your frontend application

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

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

Note: Replace TODO comments with the appropriate commands.

Remove the line where the configuration file is copied to the image.

Dockerfile-gowebapp

```
1  FROM ubuntu
2
3  # TODO --- add an environment variable declaration for a default DB_PASSWORD
4  # of "mydefaultpassword"
5  # https://docs.docker.com/engine/reference/builder/#env
6
7  COPY ./code /opt/gowebapp
8  # TODO --- remove the following line. We no longer want to include the configuration
9  # file in the image.
10 COPY ./config /opt/gowebapp/config
11
12 # TODO --- add a volume declaration for the container configuration path we want to
13 # mount at runtime from the host file system: /opt/gowebapp/config
14 # https://docs.docker.com/engine/reference/builder/#volume
15
16 EXPOSE 80
17
18 WORKDIR /opt/gowebapp/
19 ENTRYPOINT ["/opt/gowebapp/gowebapp"]
```

3.1.2 Step 2: Build updated gowebapp Docker image locally

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

Build the gowebapp image locally. Make sure to include "." at the end. Notice the new version label.

```
docker build -t gowebapp:v2 .
```

3.2 Run and test new Docker image locally

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

3.2.1 Step 1: Launch frontend and backend containers

First, we launch the backend database container, using a previously created Docker image, as it will take a bit longer to startup, and the frontend container depends on it.

```
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 using the updated gowebapp image, mapping the container port 80, where the web application is exposed, to port 9000 on the host machine. Notice how we're mapping a host volume into the container for configuration, and setting a container environment variable with the MySQL DB password:

```
docker run -p 9000:80 \
-v $HOME/gowebapp/gowebapp/config:/opt/gowebapp/config \
-e DB_PASSWORD=mypassword --net gowebapp -d --name gowebapp \
--hostname gowebapp gowebapp:v2
```

3.2.2 Step 2: 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.

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.

3.2.3 Step 3: 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
```

3.3 Publish New Image

We built the second version of gowebapp from the last section and tested it locally. Now we can tag and push gowebapp:v2 to our Docker registry.

3.3.1 Step 1: update tag for gowebapp and publish to registry

```
docker tag gowebapp:v2 localhost:5000/gowebapp:v2

docker push localhost:5000/gowebapp:v2
```

3.4 Create a Secret

3.4.1 Step 1: create secret for the MySQL password

It's not a good idea to hard-code sensitive information in configurations. Let's create a secret for the MySQL database password, so that we can reference it from configurations. Additionally let's override the default password we placed and used inside the configuration file and Docker image.

```
kubectl create secret generic mysql --from-literal=password=mypassword

kubectl describe secret mysql
```

3.4.2 Step 2: update MySQL StatefulSet to incorporate the secret

Update gowebapp-mysql-sts.yaml If you need help, please see https://kubernetes.io/docs/concepts/configuration/secret/#using-secrets-as-environment-variable

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

Note: Replace TODO comments with the appropriate commands

gowebapp-mysql-sts.yaml

```
1 apiVersion: apps/v1
2 kind: StatefulSet
3 metadata:
4   name: gowebapp-mysql
5   labels:
```

(continues on next page)

(continued from previous page)

```
    app: gowebapp-mysql
    tier: backend
spec:
  serviceName: gowebapp-mysql
  replicas: 1
  selector:
    matchLabels:
      app: gowebapp-mysql
      tier: backend
  template:
    metadata:
      labels:
        app: gowebapp-mysql
        tier: backend
    spec:
      containers:
      - name: gowebapp-mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              #TODO: Set secret name to 'mysql'
              #TODO: Set key to 'password'
        image: localhost:5000/gowebapp-mysql:v1
        ports:
        - containerPort: 3306
        volumeMounts:
        - name: mysql-pv
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-pv
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "local"
      resources:
        requests:
          storage: 5Gi
```

3.5 Perform upgrade

3.5.1 Step 1: apply new StatefulSet

Start the upgrade

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

Verify that the deployment was successful. The gowebapp-mysql pod's status should be 'Running'

```
kubectl get pods -l 'app=gowebapp-mysql'
```

3.6 Create ConfigMap

3.6.1 Step 1: create ConfigMap for gowebapp's config.json file

```
kubectl create configmap gowebapp --from-file=webapp-config-json=$HOME/gowebapp/
↪gowebapp/config/config.json

kubectl describe configmap gowebapp
```

Notice that the entire file contents from config.json are stored under the key webapp-config-json

3.6.2 Step 2: update gowebapp deployment to utilize ConfigMap

Update gowebapp-deployment.yaml.If you need help, please see https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#add-configmap-data-to-a-specific-path-in-the-volume

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

Note: Replace TODO comments with the appropriate commands

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: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              #TODO: Set secret name to 'mysql'
              #TODO: Set key name to 'password'
        # TODO: change image to v2
        image: localhost:5000/gowebapp:v1
        ports:
        - containerPort: 80
```

(continues on next page)

(continued from previous page)

```
        volumeMounts:
        - #TODO: Set volume name to 'config-volume'
          #TODO: Set mountPath to '/opt/gowebapp/config'
      volumes:
      - #TODO: define volume name: config-volume
        configMap:
          #TODO: Set ConfigMap name to 'gowebapp'
          items:
          - key: webapp-config-json
            path: config.json
```

3.7 Perform rolling upgrade

3.7.1 Step 1: apply new deployment

Start the rolling upgrade

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

Verify that rollout was successful. The two gowebapp pods' status should be 'Running'

```
kubectl get pods -l 'app=gowebapp'
```

3.8 Access your application

3.8.1 Step 1: access your application

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

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

You can now test the application. Since we added persistent storage in the last lab, your username, passoword, and notes should still be valid and present.

3.9 Lab 08 Conclusion

Congratulations! You have successfully updated your application to use dynamic configuration.

CHAPTER

FOUR

LAB 09: ADDITIONAL WORKLOADS

4.1 Manual Run Jobs

Jobs are a handy way to execute "run to completion" style workloads, unlike Deployments which are meant to run forever until they are terminated by an error or by a user. Let's explore running the job below which calculates the value of pi:

lab09-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

Download and apply lab09-job.yaml.

```
wget <url_of_yaml_above>

kubectl apply -f lab09-job.yaml
```

Notice the following observations:

- After the container exits successfully Kubernetes does not try to start another container like it does with Deployments
- The Pod and Job stick around after completion so that you can view the output

Try applying the yaml again with kubectl, does the job run a second time? What would you need to do if you wanted to rerun the job?

4.2 Parallel Jobs with a Work Queue

Parallel jobs with a work queue can create several pods which coordinate with themselves or with some external service which part of the job to work on. If your application has a work queue implementation for some remote data storage, for example, this type of Job can create several parallel worker pods that will independently access the work queue and process it. Parallel jobs with a work queue come with the following features and requirements:

- for this type of Jobs, you should leave .spec.completions unset.
- each worker pod created by the Job is capable to assess whether or not all its peers are done and, thus, the entire Job is done (e.g each pod can check if the work queue is empty and exit if so).
- when any pod terminates with success, no new pods are created.
- once at least one pod has exited with success and all pods are terminated, then the job completes with success as well.
- once any pod has exited with success, other pods should not be doing any work and should also start exiting.

Let's add parallelism to see how these type of Jobs work, notice the .spec.parallelism field was added and set to 3.

`lab09-parajob.yaml`

```
apiVersion: batch/v1
kind: Job
metadata:
  name: primes-parallel-wq
  labels:
    app: primes
spec:
  parallelism: 3
  template:
    metadata:
      name: primes
      labels:
        app: primes
    spec:
      containers:
      - name: primes
        image: debian:stable-slim
        command: ["bash"]
        args: ["-c",  "current=0; max=110; echo 1; echo 2; for((i=3;i<=max;)); do␣
↪for((j=i-1;j>=2;)); do if [  `expr $i % $j` -ne 0 ] ; then current=1; else␣
↪current=0; break; fi; j=`expr $j - 1`; done; if [ $current -eq 1 ] ; then echo $i;␣
↪fi; i=`expr $i + 1`; done"]
      restartPolicy: Never
```

Now, let's open two terminal windows. In the first terminal, watch the pods:

```
kubectl get pods -l app=primes -w
```

Download and apply lab09-job.yaml using a second terminal.

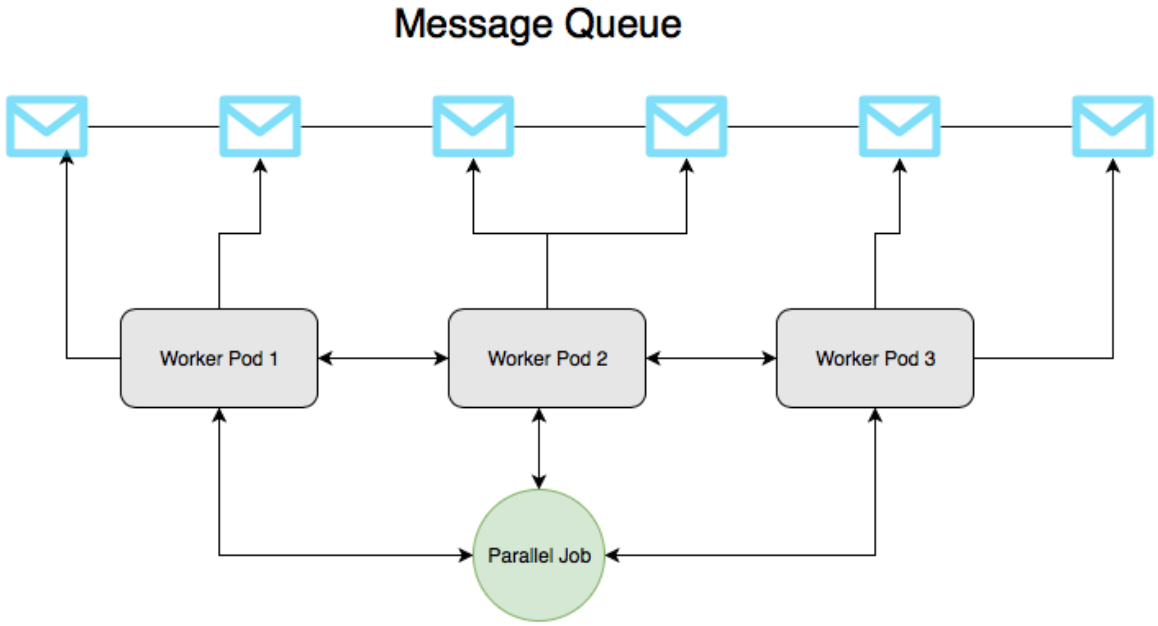
```
wget <url_of_yaml_above>
kubectl apply -f lab09-parajob.yaml
```

Next, let's see what's happening in the first terminal window:

```
kubectl get pods -l app=primes -w
NAME                       READY     STATUS    RESTARTS   AGE
primes-parallel-b2whq   0/1       Pending   0          0s
primes-parallel-b2whq   0/1       Pending   0         0s
primes-parallel-vhvqm   0/1       Pending   0         0s
primes-parallel-cdfdx   0/1       Pending   0         0s
primes-parallel-vhvqm   0/1       Pending   0         0s
primes-parallel-cdfdx   0/1       Pending   0         0s
primes-parallel-b2whq   0/1       ContainerCreating   0         0s
primes-parallel-vhvqm   0/1       ContainerCreating   0         0s
primes-parallel-cdfdx   0/1       ContainerCreating   0         0s
primes-parallel-b2whq   1/1       Running   0         4s
primes-parallel-cdfdx   1/1       Running   0         7s
primes-parallel-vhvqm   1/1       Running   0         10s
primes-parallel-b2whq   0/1       Completed   0         17s
primes-parallel-cdfdx   0/1       Completed   0         21s
primes-parallel-vhvqm   0/1       Completed   0         23s
```

As you see, the kubectl created three pods simultaneously. Each pod was calculating the prime numbers in parallel and once each of them completed the task, the Job was successfully completed as well.

In a real-world scenario, we could imagine a Redis list with some work items (e.g messages, emails) in it and three parallel worker pods created by the Job (see the Image above). Each pod could have a script to requests a new message from the list, process it, and check if there are more work items left. If no more work items exist in the list, the pod accessing it would exit with success telling the controller that the work was successfully done. This notification would cause other pods to exit as well and the entire job to complete. Given this functionality, parallel jobs with a work queue are extremely powerful in processing large volumes of data with multiple workers doing their tasks in parallel.



Example: https://kubernetes.io/docs/tasks/job/fine-parallel-processing-work-queue/

4.3 Scheduled Jobs

In the above scenario we ran a job manually and once. Sometimes you want to run a job at a regular scheduled interval or once at a particular time. CronJobs in Kubernetes provide this functionality.

lab09-cronjob.yaml

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

Download and apply lab09-cronjob.yaml.

```
wget <url_of_yaml_above>

kubectl apply -f lab09-cronjob.yaml
```

This will schedule the job to run every minute. We watch this in action by both view the job or watching the job:

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

$ kubectl get cronjob hello

NAME      SCHEDULE      SUSPEND   ACTIVE    LAST SCHEDULE   AGE
hello     */1 * * * *   False     0         <none>          9s

$ kubectl get jobs --watch

NAME               DESIRED   SUCCESSFUL   AGE
hello-1543405740   1         1            23s
```

Also try check to see if you can find/filter on the logs specifically from these logs in Kibana.

4.3.1 Lab 09 Conclusion

You should now be familiar with running Jobs and CronJobs in Kubernetes essentially allowing it to be a distributed batch processing system.

CHAPTER

FIVE

LAB 10: SECURITY

5.1 NetworkPolicy

5.1.1 Test Pod to Pod connectivity

By default, all pods are allowed to communicate with each other in a Kubernetes cluster. Let's test this behavior. We will deploy a simple NGINX web server application, and a BusyBox client application which tries to access it. To make things simple, we will deploy both pods to the same namespace. However, what's shown here would apply to pods deployed across different namespaces as well. Namespaces by default do not prohibit traffic from other namespaces.

Step 1: deploy a simple web application

First, let's deploy the simple NGINX web server.

Create `nginx.yaml` under your home directory. If you already have a file with this name from a previous lab, overwrite it.

`nginx.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
```

(continues on next page)

(continued from previous page)

```
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: nginx
  type: NodePort
```

Now apply this manifest to your Kubernetes cluster:

```
kubectl apply -f nginx.yaml
```

Step 2: access the web application from another pod

Next, let's try to access our web application from another pod in the Kubernetes cluster. By default, this should work. Let's deploy a simple interactive BusyBox application to test this:

```
kubectl run busybox --rm -ti --image=busybox /bin/sh
```

Note: Ignore any deprecation warnings when running `kubectl run`

When the prompt comes up, run the following command to access the NGINX server:

```
wget nginx --timeout 5 -O -
```

You should see some HTML come back, which indicates successful connectivity between the BusyBox client and the NGINX service.

To terminate and remove the BusyBox deployment, just exit from the shell.

5.1.2 Restrict Pod to Pod access with NetworkPolicy

For security reasons, in a multi-tenant environment, we may want to restrict network connectivity between pods. This can be done in Kubernetes by configuring ingress isolation with a NetworkPolicy. Let's enable this on the `default` namespace.

Step 1: create a NetworkPolicy

Create `default-deny.yaml` under your home directory.

Note: Replace TODO comments with the appropriate commands

`default-deny.yaml`

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: 00-default-deny
  #TODO: scope this policy to namespace: default
spec:
  #TODO: define an empty podSelector which applies to all pods
```

Now apply the manifest to your Kubernetes cluster:

```
kubectl apply -f default-deny.yaml
```

Step 2: verify network isolation

To verify NetworkPolicy enforcement, run the BusyBox client application again:

```
kubectl run busybox --rm -ti --image=busybox /bin/sh
```

When the prompt comes up, run the following command to access the NGINX server:

```
wget nginx --timeout 5 -O -
```

You should see a timeout occur after 5 seconds, indicating that the BusyBox client was unable to connect to the NGINX service. With the `NetworkPolicy` we applied in Step 1, all ingress to pods in the default namespace will be denied, even from other pods in the same namespace.

To terminate and remove the BusyBox deployment, just exit from the shell.

5.1.3 Define NetworkPolicy to enable Pod to Pod communication

Let's create a `NetworkPolicy` that whitelists pods with specific labels to communicate with each other.

Step 1: create a NetworkPolicy

Create `network-whitelist.yaml` under your home directory.

Note: Replace TODO comments with the appropriate commands

`network-whitelist.yaml`

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: 10-nginx-allow-from-lab
spec:
  podSelector:
    matchLabels:
      #TODO: add a label selector for app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          #TODO: add a label selector for lab: network
```

Now apply the manifest to your Kubernetes cluster:

```
kubectl apply -f network-whitelist.yaml
```

Step 2: test Pod to Pod communication without labels

To test connectivity, run the BusyBox client application again:

```
kubectl run busybox --rm -ti --image=busybox /bin/sh
```

When the prompt comes up, run the following command to access the NGINX server:

```
wget nginx --timeout 5 -O -
```

You should see a timeout occur after 5 seconds, indicating that the BusyBox client was still unable to connect to the NGINX service. The `NetworkPolicy` we added above only allows ingress from pods with the label `lab=network`, which this deployment did not have.

To terminate and remove the BusyBox deployment, just exit from the shell.

Step 3: test Pod to Pod communication with labels

Run the BusyBox client application again, this time with a label matching the NetworkPolicy whitelist:

```
kubectl run busybox --rm -ti --labels="lab=network" --image=busybox /bin/sh
```

When the prompt comes up, run the following command to access the NGINX server:

```
wget nginx --timeout 5 -O -
```

You should see some HTML returned, indicating a successful connection to the NGINX service.

To terminate and remove the BusyBox deployment, just exit from the shell.

5.2 Kubernetes API Authentication

In this section, we are going to - Create a service account - Configure kubectl to use the service account

5.2.1 Create a Service Account

With Kubernetes, there are many ways to authenticate requests to it's API. Typically, human users are authenticated by integrating the cluster with a centralized identity provider such as LDAP. For services, Kubernetes provides a Service Account resource that can be used to authenticate requests.

Step 1: Define a Service Account Object

Create a file called my-user-sa.yaml with the following contents

Note: Replace TODO comments with the appropriate commands

`my-user-sa.yaml`

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: #TODO: define the ServiceAccount name as 'my-user'
  namespace: #TODO: define the namespace as 'my-namespace'
```

Step 2: Apply the New Service Account

```
kubectl apply -f my-user-sa.yaml
```

Step 3: View the Service Account and it's Token

Describe the service account and retrieve the name of the secret containing it's token.

```
kubectl describe sa my-user -n my-namespace
```

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

$ kubectl describe sa my-user -n my-namespace

Name:                my-user
Namespace:           my-namespace
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   my-user-token-tnqq6
Tokens:              my-user-token-tnqq6
Events:              <none>

# In this example, the secret's name is my-user-token-tnqq6
```

Now view the secret containing the service account's token

```
kubectl get secret <token_secret_name> -n my-namespace -o yaml
```

The token contained in the secret can be used to authenticate with the Kubernetes API. It is base64 encoded and needs to be decoded before passed to the API.

5.2.2 Configure kubeconfig with the New Service Account

To test the new service account, you can add it's autogenerated token to your lab system's `kubeconfig` file. This will allow you run `kubectl` commands as the service account.

Step 1: Retrieve the Service Account Token

Retrieve the secret, decode it, and assign it to an environment variable.

Note: Replace `<token_secret_name>` with the name of the secret you retrieved in the previous step.

```
export SA_TOKEN=$(kubectl get secret <token_secret_name> -n my-namespace -o jsonpath='
↪{.data.token}' | base64 --decode)

echo $SA_TOKEN
```

Step 2: Add the Token as a Credential in kubeconfig

```
kubectl config set-credentials my-user --token $SA_TOKEN
```

Step 3: Add a kubeconfig Context

```
kubectl config set-context my-user --user my-user --cluster=kubernetes
```

Step 4: Switch to the my-user Context

```
kubectl config use-context my-user
```

Step 5: Test the New Service Account

```
kubectl get nodes
```

Note: The above command should fail stating that my-user user cannot list nodes. This is expected since you have not yet granted any permissions to the service account. You'll address that next.

Step 6: Temporary Grant Administrator Privileges

Temporarily grant administrator privileges to the my-user service account.

```
kubectl create clusterrolebinding my-user-admin --serviceaccount my-namespace:my-user␣
↪--clusterrole cluster-admin --context kubernetes-admin@kubernetes
```

Note: The last command used the `--context` option. This option allows you to run a single command with a different context. In this case the command was run using the `kubernetes-admin@kubernetes` context.

Step 7: Test the service account again

```
kubectl get nodes
```

This time, with permissions applied, you should see a list of pods.

5.3 Kubernetes API Access Control

In this section, we are going to create an RBAC role and role binding to grant access to the namespace

5.3.1 Getting Started

Step 1: Ensure kubectl is using the my-user context

```
kubectl config use-context my-user
```

Step 2: launch an nginx deployment to a namespace

Now let's launch a deployment into the default namespace:

```
kubectl run nginx --namespace default --image=nginx
```

And into the my-namespace namespace

```
kubectl run nginx --namespace my-namespace --image=nginx
```

That worked, because my-user currently has cluster-admin privileges.

Now delete the deployments before continuing

```
kubectl delete deployment nginx --namespace default

kubectl delete deployment nginx --namespace my-namespace
```

Step 3: remove cluster-admin permissions from my-user

```
kubectl delete clusterrolebinding my-user-admin --context kubernetes-admin@kubernetes
```

Step 4: Verify that we can no longer deploy applications

```
kubectl run nginx --namespace my-namespace --image=nginx
```

```
# EXAMPLE OUTPUT

Error from server (Forbidden): User "my-user" cannot create deployments.extensions in␣
↪the
namespace "my-namespace". (post deployments.extensions)
```

5.3.2 Define Namespace Scoped RBAC Policies

Step 1: Grant my-user admin privileges in my-namespace

The admin role is a default ClusterRole. We are using a RoleBinding to bind this cluster-wide role to a single namespace for this user:

Create my-user-my-namespace-admin.yaml with the following contents

Note: Replace TODO comments with the appropriate commands

my-user-my-namespace-admin.yaml

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: my-user-my-namespace-admin
  #TODO: define the namespace for the role to be bound: my-namespace
  labels:
    lab: authz
subjects:
- #TODO: define the kind as ServiceAccount
  #TODO: define user name: my-user
  namespace: my-namespace
roleRef:
  #TODO: reference role kind as cluster-wide by using ClusterRole
  #TODO: reference pre-defined admin role
  apiGroup: rbac.authorization.k8s.io
```

Step 2: create the RoleBinding object in Kubernetes

```
kubectl apply -f my-user-my-namespace-admin.yaml --context kubernetes-admin@kubernetes
```

Step 3: try to deploy an application

First, try to deploy an application to the default namespace:

```
kubectl run nginx --namespace default --image=nginx
```

That didn't work, because we didn't grant privileges in the default namespace.

Now let's try the same thing in the my-namespace namespace:

```
kubectl run nginx --namespace my-namespace --image=nginx
```

Since we're targeting the application to the my-namespace namespace where my-user has admin privileges, that worked.

5.3.3 Change Default namespace for User

Step 1: set default namespace

It's inconvenient having to specify the namespace parameter every time we run a command. Let's make the my-namespace namespace the default for my-user in kubeconfig:

```
kubectl config set-context my-user --cluster=kubernetes --user=my-user --namespace my-
↪namespace
```

Now try deleting the deployment without specifying the –namespace parameter. It should work:

```
kubectl delete deployment nginx
```

5.3.4 Switch back to the administrator user

```
kubectl config use-context kubernetes-admin@kubernetes
```

5.4 Lab 10 Conclusion

Congratulations! You have successfully deployed network policies to control network communications between pods on your cluster and you have successfully RBAC policies to limit access to actions and resources on the Kubernetes API.

CHAPTER

SIX

LAB 06: RESOURCE ORGANIZATION

6.1 Namespaces:

my-namespace.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace
```

CHAPTER

SEVEN

LAB 07: STORAGE & STATEFUL APPLICATIONS

pv.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv01
spec:
  storageClassName: local
  capacity:
    storage: 5Gi
  accessModes:
  - ReadWriteOnce
  hostPath:
    path: /var/pv/pv01
```

gowebapp-mysql-sts.yaml

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: gowebapp-mysql
  labels:
    app: gowebapp-mysql
    tier: backend
spec:
  serviceName: gowebapp-mysql
  replicas: 1
  selector:
    matchLabels:
      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:
```

(continues on next page)

(continued from previous page)

```
        - containerPort: 3306
        volumeMounts:
        - name: mysql-pv
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-pv
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "local"
      resources:
        requests:
          storage: 5Gi
```

CHAPTER

EIGHT

LAB 08: DYNAMIC APPLICATION CONFIGURATION

Dockerfile-gowebapp

```
FROM ubuntu

ENV DB_PASSWORD=mydefaultpassword

COPY ./code /opt/gowebapp

VOLUME /opt/gowebapp/config

EXPOSE 80

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

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: DB_PASSWORD
          valueFrom:
            secretKeyRef:
                name: mysql
                key: password
```

(continues on next page)

(continued from previous page)

```
        image: localhost:5000/gowebapp:v2
        ports:
        - containerPort: 80
        volumeMounts:
        - name: config-volume
          mountPath: /opt/gowebapp/config
      volumes:
      - name: config-volume
        configMap:
          name: gowebapp
          items:
          - key: webapp-config-json
            path: config.json
```

gowebapp-mysql-sts.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: gowebapp-mysql
  labels:
    app: gowebapp-mysql
    tier: backend
spec:
  serviceName: gowebapp-mysql
  replicas: 1
  selector:
    matchLabels:
      app: gowebapp-mysql
      tier: backend
  template:
    metadata:
      labels:
        app: gowebapp-mysql
        tier: backend
    spec:
      containers:
      - name: gowebapp-mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql
              key: password
        image: localhost:5000/gowebapp-mysql:v1
        ports:
        - containerPort: 3306
        volumeMounts:
        - name: mysql-pv
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-pv
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "local"
      resources:
```

(continues on next page)

(continued from previous page)

```
42          requests:
43            storage: 5Gi
```

CHAPTER

NINE

LAB 09: ADDITIONAL WORKLOADS

CHAPTER

TEN

LAB 10: SECURITY

default-deny.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: 00-default-deny
  namespace: default
spec:
  podSelector: {}

```

network-whitelist.yaml

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: 10-nginx-allow-from-lab
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          lab: network
```

my-user-sa.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-user
  namespace: my-namespace
```

my-user-my-namespace-admin.yaml

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: my-user-my-namespace-admin
  namespace: my-namespace
  labels:
     lab: authz
```

(continues on next page)

(continued from previous page)

```
subjects:
  - kind: ServiceAccount
    name: my-user
    namespace: my-namespace
roleRef:
  kind: ClusterRole
  name:  admin
  apiGroup: rbac.authorization.k8s.io
```