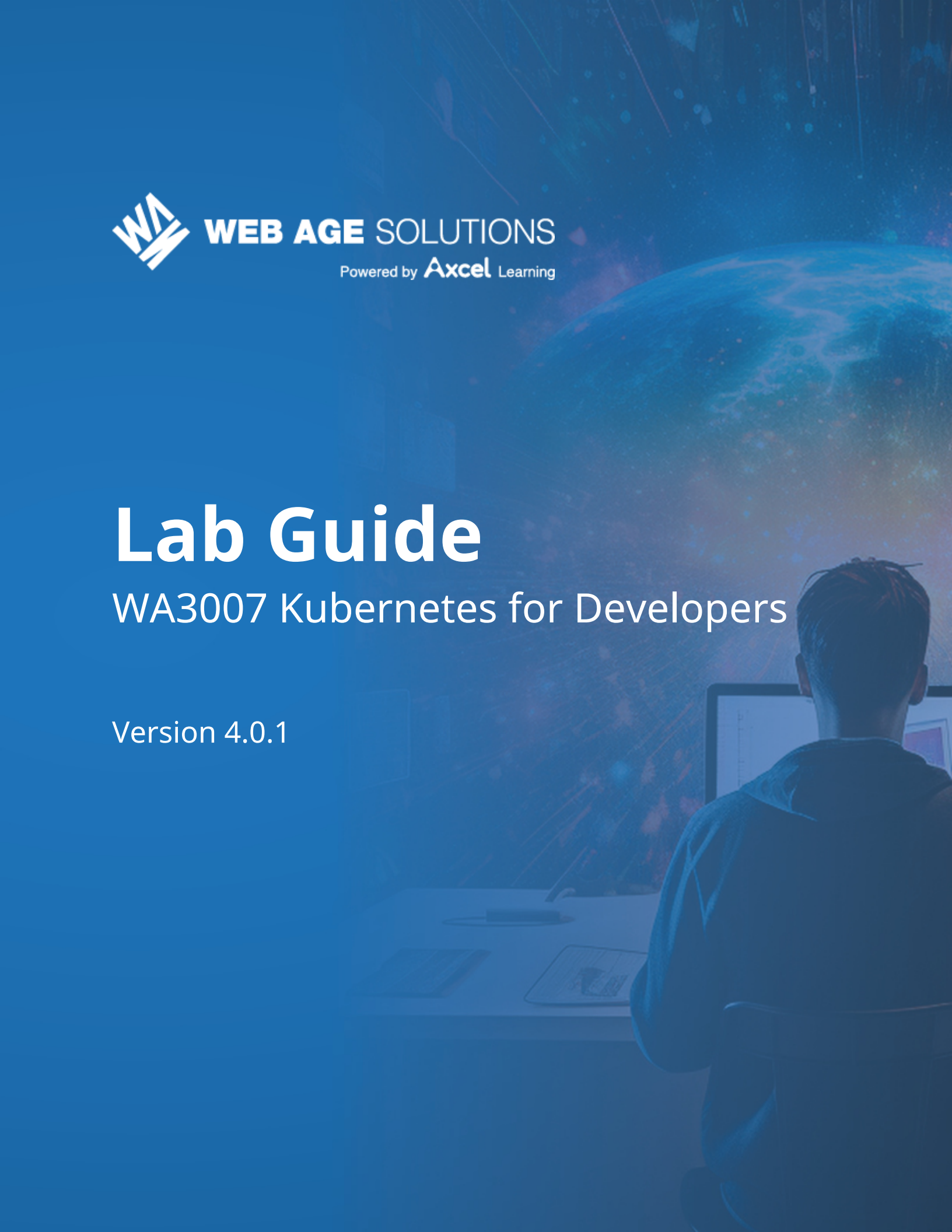


Lab Guide

WA3007 Kubernetes for Developers

Version 4.0.1

Revision 4.0.1 published on 2025-01-07.

Web Age Solutions, LLC
1 California Street Suite 2900
San Francisco, CA 94111
https://www.webagesolutions.com

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com
Canada: 1-877-812-8887 toll free, email: getinfo@webagesolutions.com

Table of Contents

1. Creating a Docker Account and Obtain an Access Token 5
1.1. Create a Docker account 5
1.2. Log in with your Docker account 7
2. Configuring Minikube/Kubernetes to Use a Custom Docker Account 8
2.1. Configure the Docker account in Minikube/Kubernetes 8
3. Getting Started with Kubernetes 12
3.1. Setting the Stage 12
3.2. Interact with the Cluster 13
3.3. The kubectl command line interface (CLI) 15
3.4. Kubernetes Dashboard 20
3.5. Review 24
4. Building a Docker Image with Dockerfile 25
4.1. Setting the Stage 25
4.2. Learning the Docker Command-line 25
4.3. Create Dockerfile for Building a Custom Image 29
4.4. Verify the Custom Image 32
4.5. Interacting with the Container 34
4.6. Stop and Delete the Container 35
4.7. Review 35
5. Deploying to Kubernetes 36
5.1. Setting the Stage 36
5.2. Deploy a Custom Image and Expose the Service 37
5.3. Exploring the Dashboard 42
5.4. View Logs and Run Commands in a Container 43
5.5. Delete the Service and Deployment 45
5.6. Redeploy the Custom Image and Expose the Service using a YAML Configuration File 46
5.7. Clean-Up 48

5.8. Review . . . 49
6. Implementing the Sidecar Pattern . . . 50
6.1. Setting the Stage . . . 50
6.2. Create an Application Pod . . . 51
6.3. Add a Second Container . . . 55
6.4. Create and Use a Shared Volume . . . 58
6.5. Deploy and Test the App and Sidecar . . . 61
6.6. Review . . . 63
7. Deploying Applications . . . 65
7.1. Setting the Stage . . . 65
7.2. Create a Deployment . . . 66
7.3. Update a Deployment . . . 69
7.4. Roll Back a Deployment . . . 71
7.5. Pausing and Resuming a Deployment . . . 74
7.6. Clean-Up . . . 75
7.7. Review . . . 76
8. Implementing RBAC Security . . . 77
8.1. Setting the Stage . . . 77
8.2. Downloading and Prepping the Security Project . . . 78
8.3. View the Cluster Configuration and Access the API Server . . . 79
8.4. Create a Certificate Signing Request and Verify the CSR . . . 81
8.5. Create a User/Principal with the Key and Certificate . . . 83
8.6. Create Contexts to Easily Switch Between Users . . . 84
8.7. Authorize the User to Access the Namespace . . . 90
8.8. Assign ClusterRole . . . 97
8.9. Clean-Up . . . 100
8.10. Review . . . 101
9. Accessing Applications . . . 102
9.1. Setting the Stage . . . 102
9.2. Create Deployment NX1 . . . 103

9.3. Create Deployment NX2 . . . 106
9.4. Check Current Network Situation . . . 109
9.5. Exposing Pods Outside the Cluster . . . 113
9.6. Cleanup . . . 115
9.7. Review . . . 116
10. Troubleshooting . . . 117
10.1. Setting the Stage . . . 117
10.2. Deploy an Application and Fix Yaml Syntax . . . 118
10.3. Troubleshooting a nodeSelector Issue. . . . 121
10.4. Troubleshooting a Failed Image Pull . . . 127
10.5. Troubleshooting Resource Issues when Scaling . . . 130
10.6. Review . . . 135

Lab 1. Creating a Docker Account and Obtain an Access Token

In this lab, you will create a Docker account and obtain an access token. With the free Docker account, you can pull up to 200 Docker images every 6 hours. Without the account, you will most likely run into Docker pull rate-limiting issue where anonymous users get restricted to 100 Docker image pulls every 6 hours and that can cause issues especially when multiple anonymous users connect from the same network.

1.1. Create a Docker account

IMPORTANT NOTE!!!

If you have been provided with a Docker account, skip to next **PART**.

In this part, you will create a Docker account.

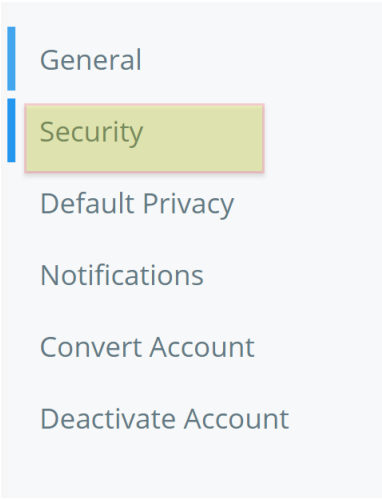
1. Open a browser and navigate to the following URL:

   https://hub.docker.com/

2. **Note**: If you get prompted for password, use **wasadmin**
3. Click **Sign Up** and create a new account.
4. Sign in with your newly created account.
5. Click on your username in the top right corner and select Account Settings.

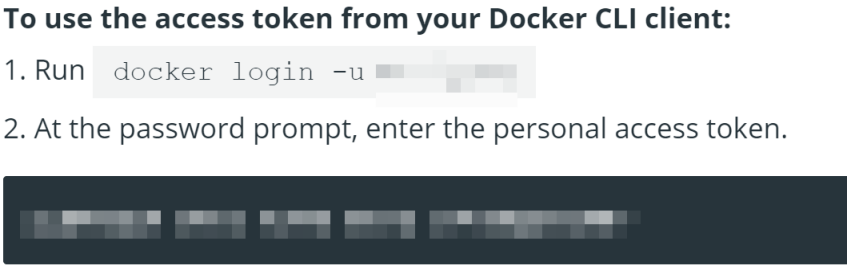
Alternatively, click navigate to **https://hub.docker.com/settings/general** to access Account Settings.

6. Select **Security**.



7. Click **New Access Token**.
8. Add the token description, such as *docker & Kubernetes labs.*
9. Keep Access permissions as default (**Read, Write, Delete**).
10. Click **Generate**

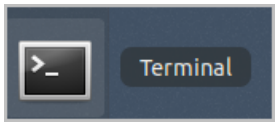
Notice it shows the following information on the page:



Make a note of both the steps displayed on the page.

1.2. Log in with your Docker account

1. Connect to the Virtual machine called "*VM_WA3007_REL_4_0**"*
2. Open a new Terminal.



3. Run the following command to log into Docker:

```
docker login -u \{your-docker-id} -p \{your-access-token}
```

   You should see a 'Login Succeeded' message.

```
root@labvm:~# sudo docker login -u [redacted] -p [redacted]
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
root@labvm:~#
```

4. Enter **exit** many times until the Terminal is closed.

Lab 2. Configuring Minikube/Kubernetes to Use a Custom Docker Account

In this lab, you will configure the Docker account/access token in Minikube. This will ensure Minikube/Kubernetes uses your custom Docker account to pull Docker images. With the free Docker account, you can pull up to 200 Docker images every 6 hours. Without the account, there is a risk you will run into Docker pull rate-limiting issue where anonymous users get restricted to 100 Docker image pulls every 6 hours and that can cause issues especially when multiple anonymous users connect from the same network.

2.1. Configure the Docker account in Minikube/Kubernetes

Make sure you run the following command:

```
docker login -u \{your-docker-id} -p \{your-access-token}
```

In this part, you will configure your Minikube/Kubernetes cluster to use the Docker account you configured earlier in the course.

1. Open a browser and connect to the docker page to make sure you have internet connection. **Note**: If you get prompted for password, use **wasadmin**:

```
www.docker.com
```

2. Open a Terminal window.
3. Ensure you have the following Docker configuration file:

   ```
   cat ~/.docker/config.json
   ```

   If you don’t see any contents, or it shows file not found, ensure you have executed “docker login” as mentioned in the note earlier in this lab.

4. Set Docker as the Minikube driver:

   ```
   minikube config set driver docker
   ```

5. For the driver change to take effect, run delete command (Minikube might complain that no profile exists, which is fine. Ignore the warning/error if it shows up):

   ```
   minikube delete
   ```

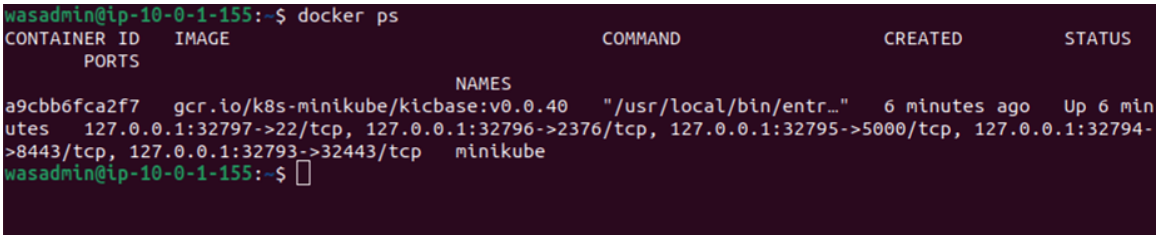
6. Now start Minikube. This step can take a couple of minutes as Minikube downloads the right image from Docker Hub.

   ```
   minikube start
   ```

7. Verify that Minikube is running as a Docker container. You should

see one container listed with the name of "minikube"

```
docker ps
```



8. Run the following four commands one after another to copy the Docker configuration to the right places within Minikube:

```
minikube ssh "mkdir -p .docker"

minikube cp ~/.docker/config.json minikube:/home/docker/.docker/

minikube cp ~/.docker/config.json minikube:/tmp/

minikube ssh "sudo cp /tmp/config.json /var/lib/kubelet/"
```

9. Restart the cluster for changes to take effect. Run these one after another:

```
minikube stop

minikube start
```

10. The above steps ensure that the Kubelet inside Minikube uses your Docker Hub credentials whenever downloading Docker images for the subsequent lab exercises.

11. Exit the Terminal and close the browser.

Lab 3. Getting Started with Kubernetes

The main terminal-based tool for working with Kubernetes is the kubectl command. With it you can deploy objects like deployments, pods and services and see what is going on inside the cluster. Another important tool is a GUI web interface called the Kubernetes Dashboard. It provides much of the same functionality as kubectl but does so in a way that for some operations is easier to navigate and operate. In this lab we will take a look at both of these tools and learn some of the things you can do with them.

1. kubectl Command
2. Kubernetes Dashboard

Make sure you run the following command:

```
docker login -u \{your-docker-id} -p \{your-access-token}
```

3.1. Setting the Stage

1. Open a new Terminal window.
2. Ensure you can access the Docker CLI by running the following:

   ```
   docker ps
   ```

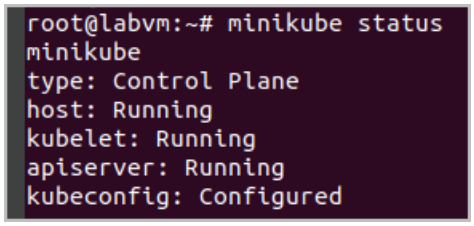
3.2. Interact with the Cluster

In this part, you will interact with the Minikube Kubernetes cluster in various ways.

1. Ensure that the cluster is running:

   ```
   minikube status
   ```

   You should see:

   If the cluster shows as not running, run the following (otherwise skip this step).

   ```
   minikube start
   ```

   Once the cluster starts, check status again as mentioned in the previous step.

2. Enter the following to enable ingress functionality:

   ```
   minikube addons enable ingress
   ```

   You should see something similar to the following screenshot (the versions might be different):

```
root@labvm:~# minikube addons enable ingress
    ▪ Using image k8s.gcr.io/ingress-nginx/controller:v0.44.0
    ▪ Using image docker.io/jettech/kube-webhook-certgen:v1.5.1
    ▪ Using image docker.io/jettech/kube-webhook-certgen:v1.5.1
🔎  Verifying ingress addon...
🌟  The 'ingress' addon is enabled
```

3. Enter the following to get the cluster's IP:

```
minikube ip
```

   You should see the following (yours may be different):

```
root@labvm:~# minikube ip
192.168.153.223
```

4. Get Kubernetes cluster information:

```
kubectl cluster-info
```

   It will display the IP address and port where the Kubernetes master is running. Yours will be different.

```
wasadmin@ip-10-0-1-155:~$ kubectl cluster-info
Kubernetes control plane is running at https://192.168.49.2:8443
CoreDNS is running at https://192.168.49.2:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
wasadmin@ip-10-0-1-155:~$
```

5. Create a working directory called Works inside your home directory:

```
mkdir ~/Works
```

6. Navigate to the working directory:

```
cd ~/Works
```

Check the command prompt. It should be:

```
wasadmin@<machine-id>:~/Works$
```

3.3. The kubectl command line interface (CLI)

The 'kubectl' command line interface is one of two main tolls available for you to configure and manage a Kubernetes cluster. The other tool is the Kubernetes Dashboard.

In this lab we will review a few of the high level kubectl resource commands you might expect to use on a daily basis including:

- explain
- create
- apply
- get
- delete

These commands work with most K8s resources like: **namespaces, nodes, pods, deployments, services, etc.**

Lets start by checking the cluster for any existing deployments. For this we will use the 'Get' command. The syntax of this command is **kubectl get {resource-type}**

1. Run the following **get** command:

   ```
   kubectl get deployments
   ```

   Your cluster shouldn't show any existing deployments.

   If it does then the output would look something like this:

   ```
   NAME        READY   UP-TO-DATE  AVAILABLE   AGE
   nginx-web   1/1     1           1           20h
   ```

   If you have any existing deployments lets go ahead and delete them. You will use the *kubectl delete {resource-type} {resource-name}* command. Run the following delete command, make sure to substitute the name that was output from the earlier get command:

   ```
   kubectl delete deployments nginx-web
   ```

   Repeat the above command for each deployment you found in your get listing.

   Another important command is **apply**. It is used to create and update resources and has the following syntax:

   ```
   kubectl apply -f \{resource-definition-filename}
   ```

2. To use the apply command, we need a resource definition file. Lets create one using the VSCode editor:

```
code myapp.yaml
```

3. Enter the following text into the file. You are welcome to use VSCode's ability to help you with indentations to ensure the YAML is syntactically correct:

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

4. Save and close VSCode.

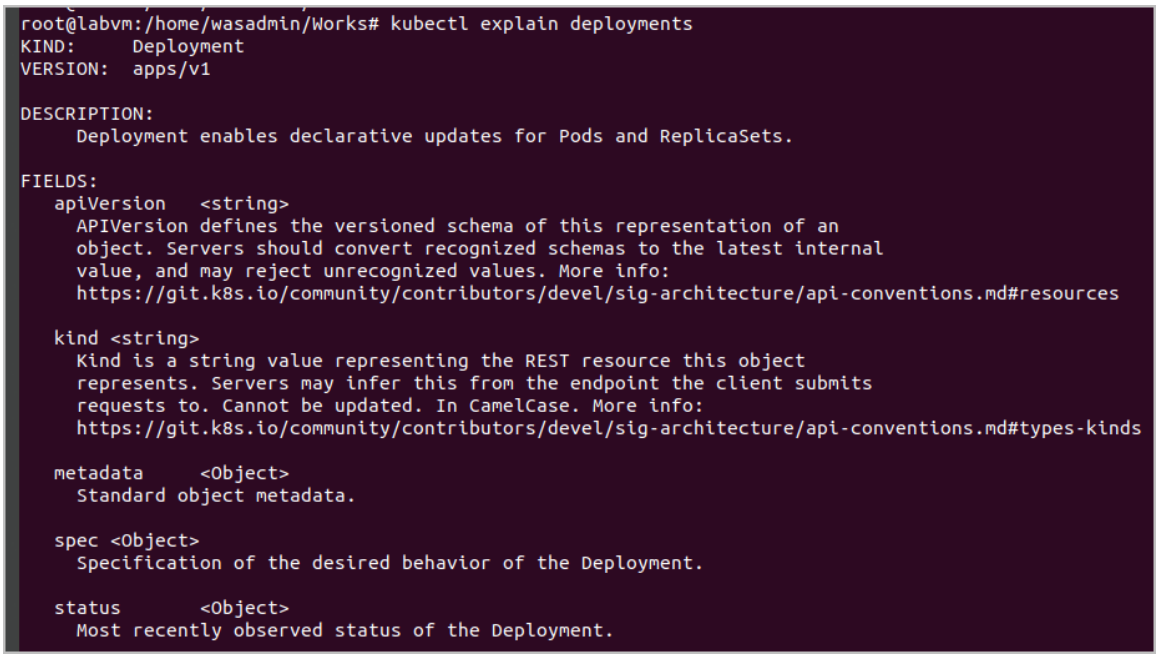
   But what do all of those settings mean? We can get a basic understanding by using the 'kubectl explain' command which has this syntax:

```
kubectl explain \{resource-type}
```

5. Try running the following explain command:

   ```
   kubectl explain deployments
   ```

   The output of the command is:



6. Its also possible to drill down on this information. For example what about if we wanted to know more about the fields under **"metadata:"**? Try running the following command:

   ```
   kubectl explain deployments.metadata
   ```

   There are a lot of possible fields that could go under 'metadata' so the output of the above command is long. The first part of the

output looks like this:

```
root@labvm:/home/wasadmin/Works# kubectl explain deployments.metadata
KIND:     Deployment
VERSION:  apps/v1

RESOURCE: metadata <Object>

DESCRIPTION:
     Standard object metadata.

     ObjectMeta is metadata that all persisted resources must have, which
     includes all objects users must create.
```

Now that we better understand the deployment yaml file lets use it to create a deployment.

7. Execute the following **apply** command to create a deployment based on the yaml file we just created:

```
kubectl apply -f myapp.yaml
```

8. Check that the deployment was created:

```
kubectl get deployments
```

The deployment 'apache' should be in the list and the AVAILABLE field should say '1'. If if shows 0/1 wait a minute and run the command again until 1/1 is shown.

Another way to deploy an application is to use the **create** command. It is less flexible than **apply** but it does come in handy when you need to create a quick deployment.

```
root@labvm:/home/wasadmin/Works# kubectl get deployments
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
apache   1/1     1            1           10m
```

9. Deploy the '**alpine**' docker image using the following **create** command:

```
kubectl create deployment nginx --image nginx:latest
```

10. Check deployments:

```
kubectl get deployments
```

You may need to wait a minute until nginx is running. The output should look like this:

```
root@labvm:/home/wasadmin/Works# kubectl get deployments
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
apache   1/1     1            1           12m
nginx    1/1     1            1           59s
```

Next we will look at the kubernetes dashboard. Leave the deployments running.

3.4. Kubernetes Dashboard

Another API tool is the Kubernetes dashboard. The dashboard is a GUI web application that makes it easy to check how the cluster is working.

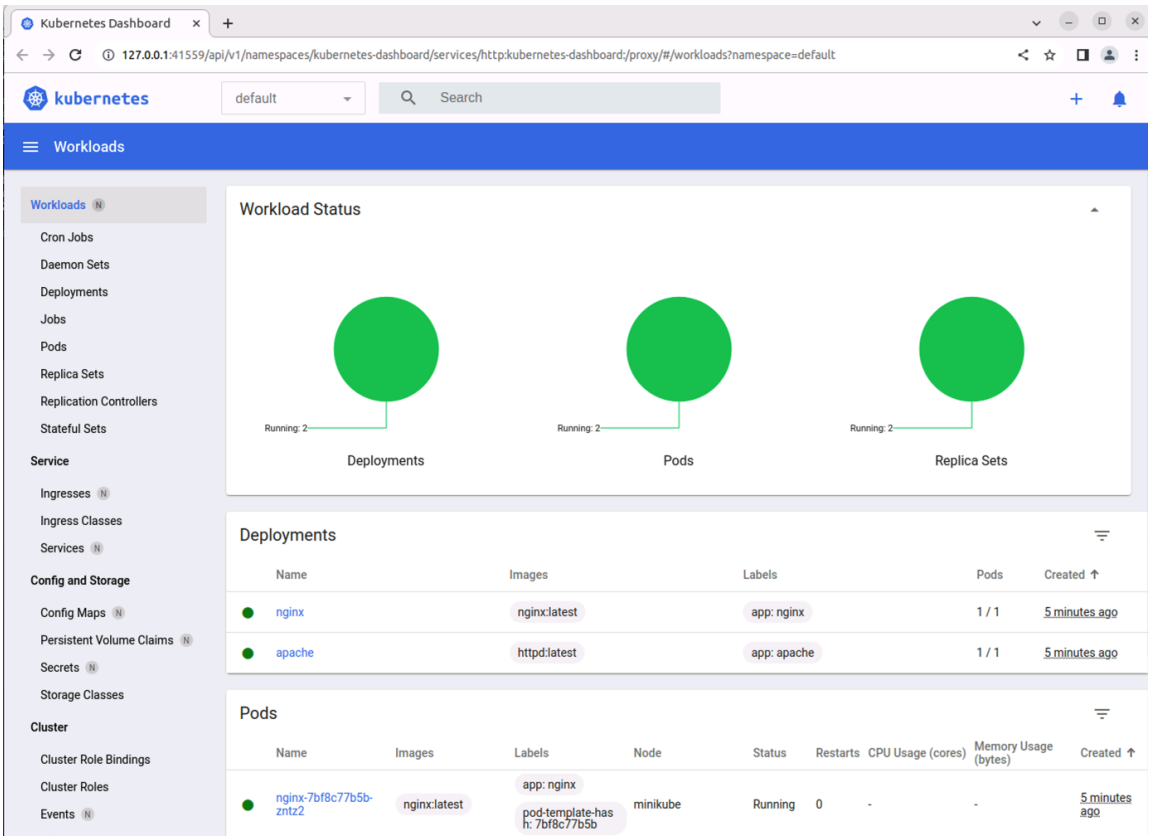
1. Run the following command to start the dashboard:

```
minikube dashboard
```

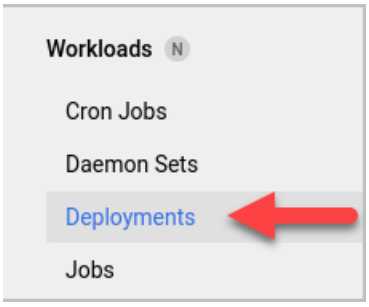
After a few moments it should output the following to the console.

```
wasadmin@ip-10-0-1-155:~/Works$ minikube dashboard
🔌  Enabling dashboard ...
    ▪ Using image docker.io/kubernetesui/dashboard:v2.7.0
    ▪ Using image docker.io/kubernetesui/metrics-scraper:v1.0.8
💡  Some dashboard features require the metrics-server addon. To enable all features please run:

        minikube addons enable metrics-server


🤔  Verifying dashboard health ...
🚀  Launching proxy ...
🤔  Verifying proxy health ...
🎉  Opening http://127.0.0.1:41559/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/ in you
[325334:325334:0817/231706.062187:ERROR:object_proxy.cc(590)] Failed to call method: org.freedesktop.portal.Settings.Read:
ortal/desktop: org.freedesktop.DBus.Error.AccessDenied: Portal operation not allowed: Unable to open /proc/325334/root
[325376:325376:0817/231706.478078:ERROR:viz_main_impl.cc(186)] Exiting GPU process due to errors during initialization
[325412:8:0817/231706.705817:ERROR:command_buffer_proxy_impl.cc(128)] ContextResult::kTransientFailure: Failed to send GpuC
```

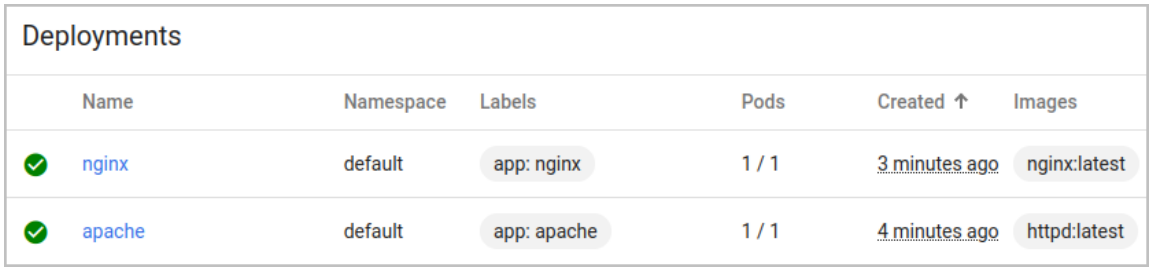
2. Typically, the browser will launch automatically. If not, look for the URL in the terminal output and browse to it in your browser manually.
3. There are some known Minikube issues that result in warnings even though the dashboard launches successfully. You might see these errors in the terminal. Ignore them.
4. The dashboard app looks like this:



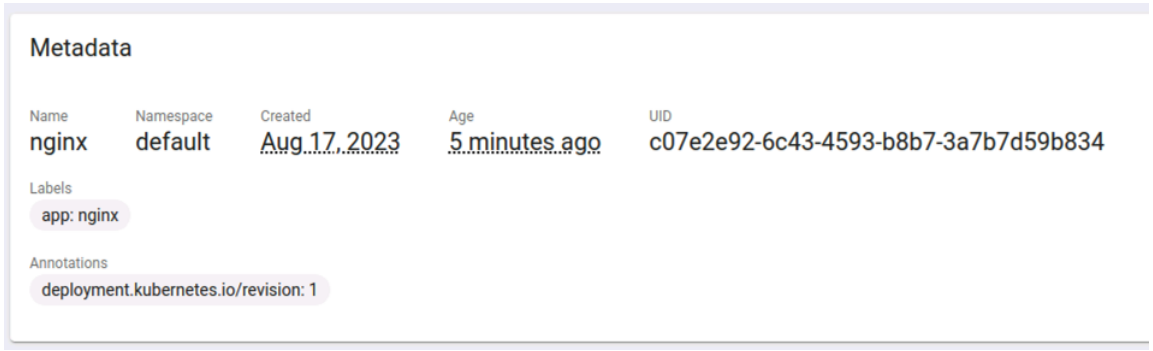
5. The app has a navigation area to the left of the screen with links. Click on the **deployments** link under **Workloads**.



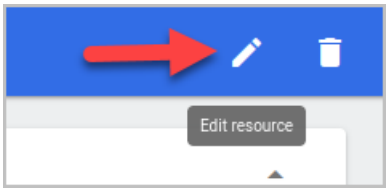
   The main part of the screen should change to show your deployments:



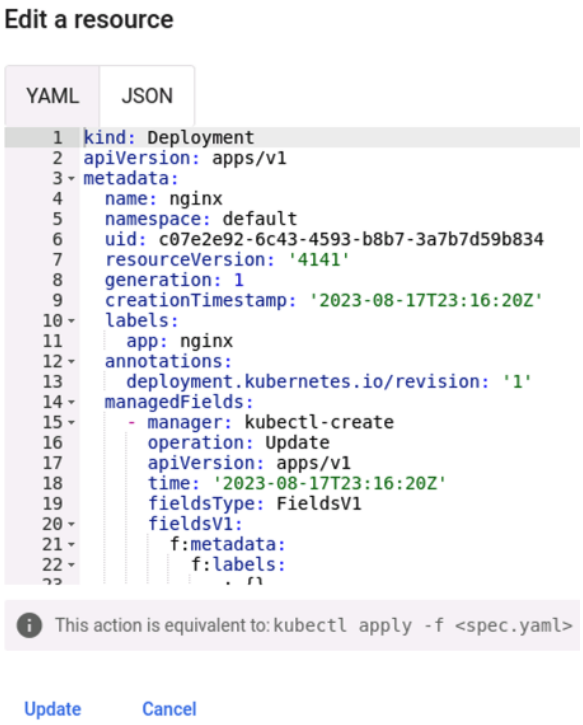
6. Click the **nginx** deployment. This will show you the deployment details.



7. Scroll down to see more sections.
8. Click on the **Edit resource** button at the upper right of the screen.



This will pop up an edit screen with the deployments information in it. From this screen you can edit and apply updates to the deployment.



9. Click **Cancel** to dismiss the edit window.
10. Scale, Edit and Delete resources are also available at the top right of the window.

Feel free to try out some of the other links. As you become more familiar with what's available in Dashboard and as you work more

with Kubernetes you may find some operations are easier to execute from the dashboard's GUI than they are from a terminal using kubectl commands.

11. Close the browser.
12. In the Terminal, stop the command by pressing **CTRL+C**.
13. Type **exit** to close the Terminal.

3.5. Review

In this lab we looked at two tools for interacting with Kubernetes clusters kubectl CLI and the Kubernetes Dashboard.

Lab 4. Building a Docker Image with Dockerfile

Docker is an open-source containerization solution. Docker containers can run on any OS in an on-prem environment and also on any cloud platform. In this lab, you will create a custom Docker image by creating a Dockerfile.

Make sure you run the following:

```
docker login -u \{your-docker-id} -p \{your-access-token}
```

4.1. Setting the Stage

1. Open a new Terminal window.
2. Ensure you can access the Docker CLI by running the following:

   ```
   docker ps
   ```

4.2. Learning the Docker Command-line

Get quick information about Docker by running it without any arguments.

1. Run the following command:

```
docker | less
```

2. Enter **q** to exit.

   The commands list is shown below for you reference.

attach	Attach local standard input, output, and error streams to a
build	Build an image from a Dockerfile
commit	Create a new image from a container’s changes
cp	Copy files/folders between a container and the local filesystem
create	Create a new container
diff	Inspect changes to files or directories on a container’s
events	Get real time events from the server
exec	Run a command in a running container
export	Export a container’s filesystem as a tar archive
history	Show the history of an image
images	List images
import	Import the contents from a tarball to create a filesystem image
info	Display system-wide information
inspect	Return low-level information on Docker objects
kill	Kill one or more running containers

load	Load an image from a tar archive or STDIN
login	Log in to a Docker registry
logout	Log out from a Docker registry
logs	Fetch the logs of a container
pause	Pause all processes within one or more containers
port	List port mappings or a specific mapping for the container
ps	List containers
pull	Pull an image or a repository from a registry
push	Push an image or a repository to a registry
rename	Rename a container
restart	Restart one or more containers
rm	Remove one or more containers
rmi	Remove one or more images
run	Run a command in a new container
save	Save one or more images to a tar archive (streamed to STDOUT by
search	Search the Docker Hub for images
start	Start one or more stopped containers
stats	Display a live stream of container(s) resource usage statistics
stop	Stop one or more running containers
tag	Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE

top	Display the running processes of a container
unpause	Unpause all processes within one or more containers
update	Update configuration of one or more containers
version	Show the Docker version information
wait	Block until one or more containers stop, then print their exit

3. You can get command-specific help by using the **--help** flag added to the commands invocation line, e.g. to list containers created by Docker (the command is called **ps**), use the following command:

```
docker ps --help
```

More information on Docker's command-line tools can be obtained at https://docs.docker.com/reference/commandline/cli/

4. Enter the following command:

```
docker images
```

This command displays docker images available on the machine.

4.3. Create Dockerfile for Building a Custom Image

In this part, you will create a Dockerfile which will build a custom Docker image.

1. In the Terminal, navigate to the Works directory you created earlier:

```
cd ~/Works/
```

2. Download the sample Java application by using the following wget command:

```
wget https://course-sw.s3.amazonaws.com/sample-webapp.zip
```

3. Extract the contents of the sample Java application:

```
unzip sample-webapp.zip
```

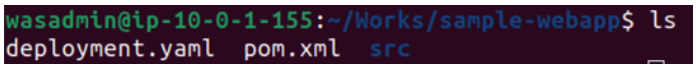
4. Switch to the sample-webapp directory:

```
cd sample-webapp
```

5. Check the content:

```
ls
```

You should see:



6. Type the following command to create Dockerfile:

```
code Dockerfile
```

7. Enter following code:

```
FROM maven:3.6-jdk-8 AS builder
COPY . /app
WORKDIR /app
RUN mvn install

FROM openjdk:8-jdk-alpine
ARG TARGET=app/target
COPY --from=builder $\{TARGET}/ /app/target
RUN ls /app/target
EXPOSE 8080
ENTRYPOINT ["java","-jar","./app/target/sample-webapp-1.0.jar"]
```

The Dockerfile script is a 2-stage script. The first stage will obtain a base image from Docker Hub that will already have Maven and JDK8 preinstalled. After downloading the base image, your custom Java application source will be copied to it and Maven will build your application to generate the deployable artifact (.jar). The second stage will use a very small base image that already has all the dependencies required to run your custom Java application.

8. Click **Save** button.
9. Close VSCode.
10. Type `ls` and verify **Dockerfile** new file is there.
11. Run following command to build a custom image:

```
docker build -t sample-webapp:v1.0 .
```

```
This command builds/updates a custom image named _dev-
openjdk:v1.0_.

Don't forget to add the period at the end of docker build
command.

Notice, Docker creates the custom image with JDK installed in it
and the jar file deployed in the image. The first time you build
the job, it will be slow since the image will get built for the
first time. Subsequent runs will be faster since image will just
get updated, not rebuilt from scratch.
```

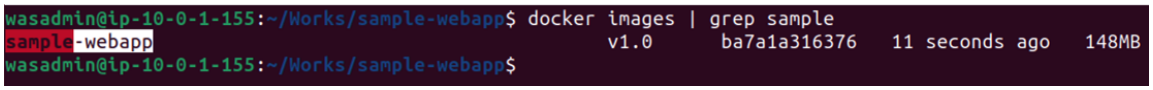
Wait until the process is completed, at the end of the messages you should see:

4.4. Verify the Custom Image

In this part you will create a container based on the custom image you created in the previous part. You will also connect to the container, verify the jar file for the application exists, and execute the jar file.

1. In the terminal, run following command to verify the custom image exists:

```
docker images | grep sample
sample-webapp v1.0 is shown:
```



2. Create a container based on the above image and connect to it:

```
docker run -d --name sample-webapp-container -p 8080:8080 sample-
webapp:v1.0
```

Note: docker run is the short form of docker container run.

-d runs the container in background as a daemon.

--name assigns a name to the container. You can create multiple containers based on the same image. Name can be utilized with commands to stop, start, and remove containers.

-p lets you perform port mapping. The number in front of the colon (:) symbol is where the port will be available.

The command will show a message similar to this:

```
wasadmin@ip-10-0-1-155:~/Works/sample-webapp$ docker run -d --name sample-webapp-container -p 8080:8080 sample-webapp:v1.0
760470d6b59782e587b476c0e10c260f6f6d9f3b15b94763b734e5b12137e4d2
```

Troubleshoot:

In case you make a mistake in the command and when you are running the command again it shows that the container already exists, then you have to stop it and remove it using the following commands:

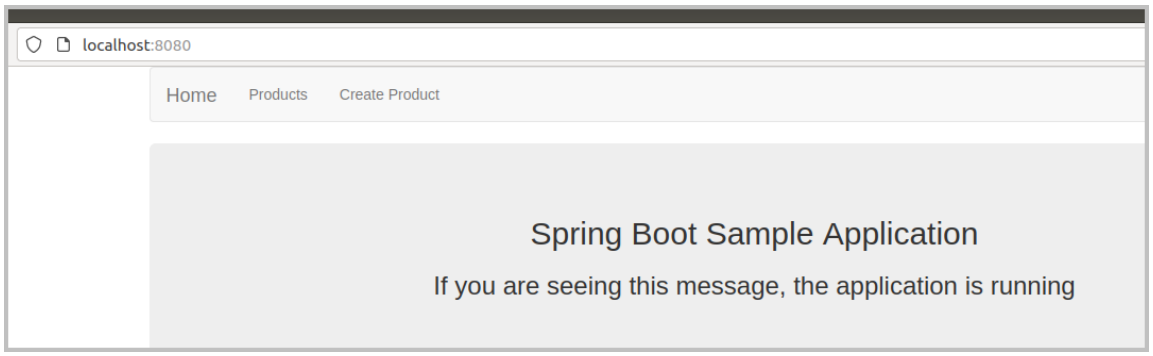
```
docker stop sample-webapp-container
docker rm sample-webapp-container
```

and then run again the command to create the container.

3. Open a web browser window and navigate to the following URL:

```
http://localhost:8080
```

4. Verify the following page shows up:



4.5. Interacting with the Container

In this part, you will check container logs and execute commands in the container.

1. Run the following command to check container logs:

   ```
   docker logs sample-webapp-container
   ```

   You can inspect logs to troubleshoot your application running in a container. If the application in a container crashes, the container will crash as well.

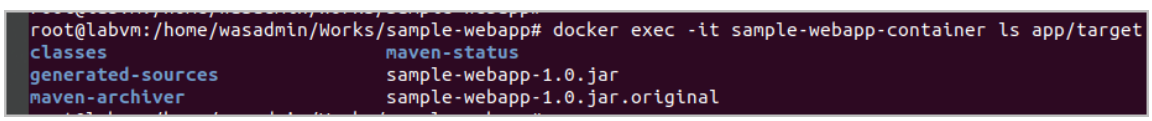
2. Run the following command to inspect the Java version in the container:

   ```
   docker exec -it sample-webapp-container java -version
   ```

3. Run the following command to inspect the directory structure and verify your sample web application is deployed:

   ```
   docker exec -it sample-webapp-container ls app/target
   ```

   Notice **sample-webapp-1.0.jar** is available in the container. ls app/target command got executed in the container. You could you cat or use any other command to view contents of any file available in the container.



4.6. Stop and Delete the Container

In this part, you will stop and delete the docker container you created in the previous parts of the lab.

1. Stop the container:

```
docker stop sample-webapp-container
```

You can also obtain the container ID by using “docker ps -a” and use the ID instead of the container name.

2. Destroy the container:

```
docker rm sample-webapp-container
```

You can also delete an unused docker image by running “docker rmi <image_name>:<tag>

3. Type **exit** until the Terminal is closed.
4. Close all.

4.7. Review

In this lab, we reviewed the main Docker command-line operations, created a Dockerfile script, and ran it to build a Docker image to containerize a custom Java application.

Lab 5. Deploying to Kubernetes

Kubernetes is an open-source container orchestration solution. It is used for automating deployment, scaling, and management of containerized applications. In this lab, you will explore the basics of Kubernetes. You will use the Minikube distribution of Kubernetes which allows you to create a Kubernetes environment with ease. Minikube is a single-node Kubernetes cluster. In more advanced cases, you will want to use OpenShift to set up a multi-node cluster.

Make sure you run the following command:

+

```
docker login -u \{your-docker-id} -p \{your-access-token}
```

5.1. Setting the Stage

1. Open a new Terminal window.
2. Ensure you can access the Docker CLI by running the following:

   ```
   docker ps
   ```

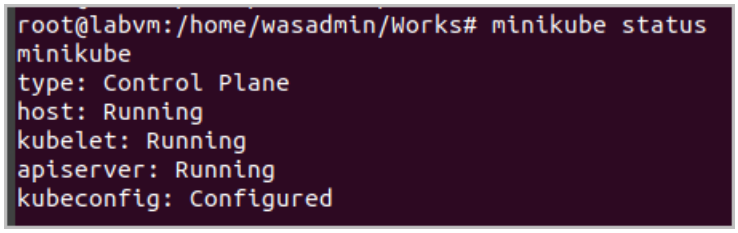
3. In the Terminal, navigate to the Works directory you created earlier:

   ```
   cd ~/Works/
   ```

4. Check minikube status:

```
minikube status
```

You should see that minikube is Running:



If minikube is not running then start it using this command:

```
minikube start
```

5.2. Deploy a Custom Image and Expose the Service

In this part, you will manually deploy a custom docker image and verify your service is working in Kubernetes.

1. Switch to the sample-webapp directory:

```
cd ~/Works/sample-webapp
```

2. Minikube has a separate Docker daemon from the host lab machine. When we build the Docker image in the previous lab, it was cached into the environment of the host machine. In this step,

we will switch the environment. Run the following in the terminal:

```
eval $(minikube -p minikube docker-env)
```

3. Run **docker images** and notice how the list of images now is very different. You should see all the Kubernetes component images listed. If you search, you won't see the sample-webapp, since it was not cached in this environment.
4. Next rebuild the Docker image so it caches in the Minikube Docker environment:

```
docker build -t sample-webapp:v1.0 .
```

5. Create a new application deployment using this freshly created Docker image:

```
kubectl create deployment sample-webapp --image=sample
-webapp:v1.0
```

6. Get the deployment list:

```
kubectl get deployments
```

It might take a while before the READY status shows 1/1 for sample-webapp. You may need to wait for a few minutes and run the command again until it shows 1/1.

```
root@labvm:/home/wasadmin/Works/sample-webapp# kubectl get deployments
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
apache          1/1     1            1           131m
nginx           1/1     1            1           130m
sample-webapp   1/1     1            1           6s
```

7. Get the pod list:

```
kubectl get pods
```

It might take a while before the pod READY status shows 1/1. You may need to wait for a few minutes and run the command again until it shows 1/1.

```
root@labvm:/home/wasadmin/Works/sample-webapp# kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
apache-68cf467b84-5tcgw           1/1     Running   0          131m
nginx-55649fd747-qgll5            1/1     Running   0          130m
sample-webapp-588bd6c7c7-5mts7    1/1     Running   0          55s
```

8. In the terminal, run the following command to see exposed services:

```
kubectl get services
```

You won't see your custom node-app service since it's not exposed.

9. In the terminal, run the following command to expose the service:

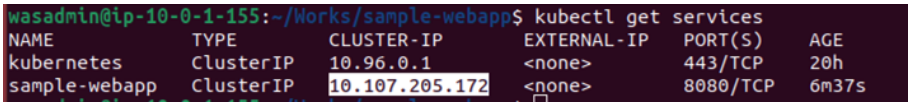
```
kubectl expose deployment sample-webapp --port 8080
```

You can define custom port mapping by using --target -port=<internal_port> --port <external_port>

10. Get the services list:

```
kubectl get services
```

11. Notice how now you have a CLUSTER-IP assigned to the **sample-webapp** service:



12. Take note of the ClusterIP. In this case, it is 10.107.205.172. Yours will be different.
13. Since this is an *internal* address, you won't be able to access it from your host machine directly (we will look into it in the Services lab later on in the course). For now, you can **ssh** into the cluster and then use **curl** to see if the webpage is up and running.
14. Use the following to ssh into the Minikube cluster:

```
minikube ssh
```

15. Now use **curl** to see if the website is up and running. Use the ClusterIP you noted earlier. Also, use port 8080 for this.

```
curl http://10.107.205.172:8080
```

16. You should see the html of the home page in the terminal, like this:

```
wasadmin@ip-10-0-1-155:~/Works/sample-webapp$ minikube ssh
docker@minikube:~$ curl http://10.107.205.172:8080
<!DOCTYPE html>
<html>
<head lang="en">

    <title>Sample Spring Boot Application</title>


    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <link href="/webjars/bootstrap/3.3.4/css/bootstrap.min.css"
          rel="stylesheet" media="screen" />

    <script src="/webjars/jquery/2.1.4/jquery.min.js"></script>

    <link href="/css/guru.css"
          rel="stylesheet" media="screen"/>

</head>
<body>

<div class="container">

        <nav class="navbar navbar-default">
            <div class="container-fluid">
                <div class="navbar-header">
                    <a class="navbar-brand" href="/">Home</a>
                    <ul class="nav navbar-nav">
                        <li><a href="/products">Products</a></li>
                        <li><a href="/product/new">Create Product</a></li>
                    </ul>

                </div>
            </div>
        </nav>

        <div class="jumbotron">
            <div class="row text-center">
                <div class="">
                    <h2>Spring Boot Sample Application</h2>

                    <h3>If you are seeing this message, the application is running</h3>
                </div>
            </div>
        </div>

</div>
</body>
</html>docker@minikube:~$ exit
logout
wasadmin@ip-10-0-1-155:~/Works/sample-webapp$
```

17. Exit the ssh session by typing **exit** at the prompt. This will take you back to your main terminal session.
18. Finally, let's revert our host machine's Docker daemon to the original host machine Docker environment.:

```
eval $(minikube docker-env -u)
```

19. If you run **docker images** now, you should notice the original list

of images.

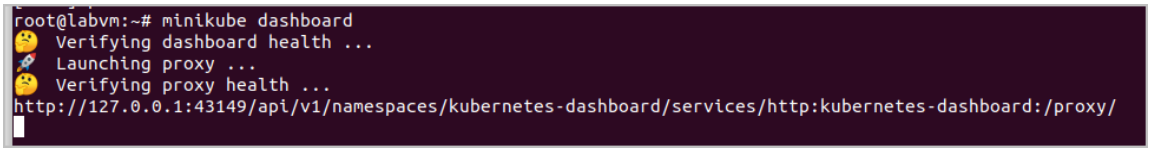
5.3. Exploring the Dashboard

In this part, you will start the Minikube dashboard and explore some of the steps that you previously performed from the terminal.

1. In the terminal, run the following command to launch the Minikube dashboard:

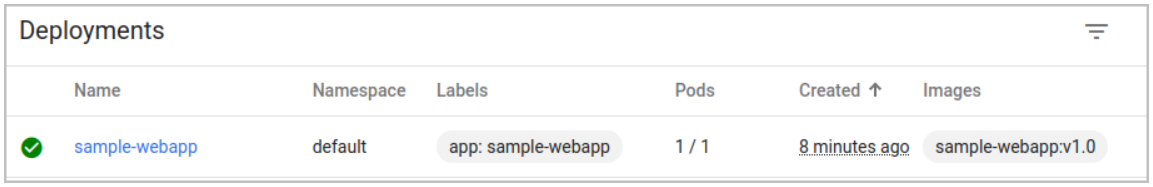
   ```
   minikube dashboard
   ```

   The browser should launch automatically with the dashboard. If that doesn't happen, look for the URL in the terminal output and use that to launch the dashboard manually by CTRL+Clicking on it:

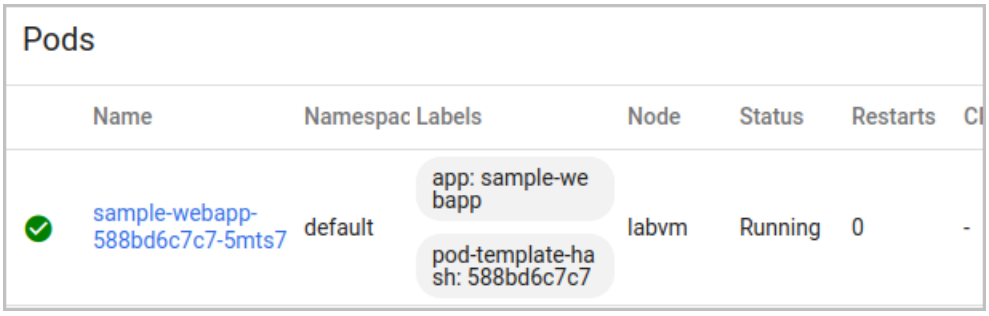
2. On the left side of the dashboard, click **Deployments**.

   Verify you have the sample-webapp deployment that you deployed from the terminal window. You may see more deployments.

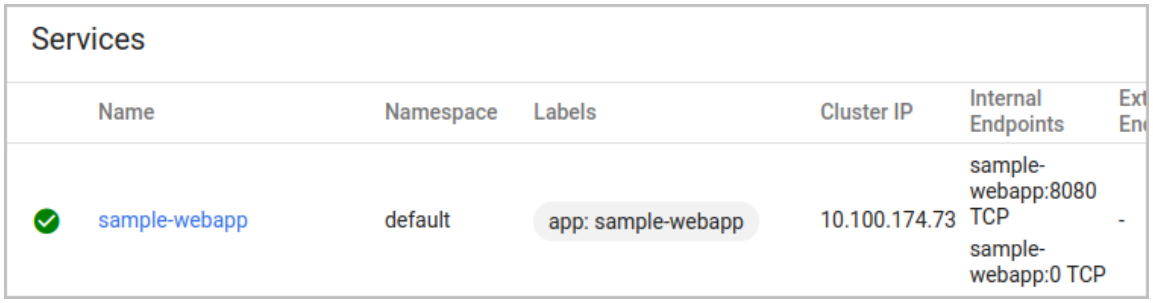
3. On the left side of the dashboard, click **Pods**.

Verify you have the sample-webapp pod. Your pod id will be different.



4. On the left side of the dashboard, click **Services.**

   Verify you have the sample-webapp service



5. Exit the browser and go back to the terminal. Enter **Ctrl+C** to end the Dashboard application.

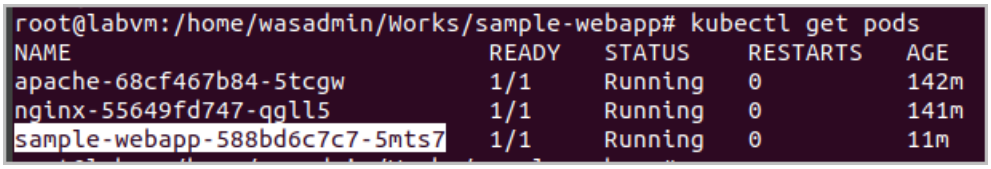
5.4. View Logs and Run Commands in a Container

In this part, you will view logs and run commands in a container.

1. In the terminal window, run the following command to get the pod list:

```
kubectl get pods
```

2. Make a note of the **sample-webapp** pod name:



3. In the terminal, run the following command to view logs of the container running in a pod:

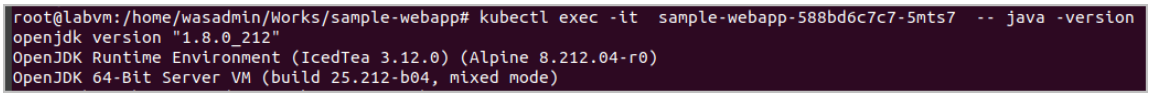
```
kubectl logs <pod_name>
```

   Notice the command displays the logs from the application running in the container of the pod specified in the command.

4. Run the following command to get Java version in the container:

```
kubectl exec -it <pod_name> -- java -version
```

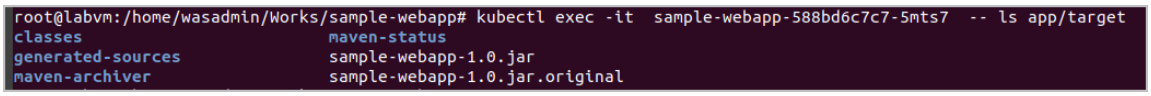
   Ensure there's a space in between — and java



5. Run the following command to view directory structure and verify the application is deployed:

```
kubectl exec -it <pod_name> -- ls app/target
```

Ensure there's a space in between — and ls.



5.5. Delete the Service and Deployment

In this part, you will delete the sample-webapp service and deployment.

1. In the Kubernetes terminal, run the following command to delete the sample-webapp service:

   ```
   kubectl delete service sample-webapp
   ```

2. Run the following command to delete the deployment:

   ```
   kubectl delete deployment sample-webapp
   ```

3. Verify the service and deployment are deleted:

   ```
   kubectl get deployments
   kubectl get pods
   kubectl get services
   ```

   NOTES: You might see a service named Kubernetes – this is a system service and is fine.

5.6. Redeploy the Custom Image and Expose the Service using a YAML Configuration File

In this part, you will redeploy the custom image and expose the service using a YAML configuration file. YAML configuration is the preferred way to deploy images and configure services. You can check in the YAML files to a source control repository, such as Git, to version the changes.

1. View the YAML configuration file:

```
cat deployment.yaml | more
```

represents comments.

The first section deploys sample-webapp:v1.0 image to the Kubernetes cluster.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: sample-webapp
    namespace: default
spec:
    replicas: 1
    selector:
        matchLabels:
            app: sample-webapp
    template:
        metadata:
            labels:
                app: sample-webapp
```

```
            spec:
                containers:
                - image: sample-webapp:v1.0
                  name: sample-webapp
---
```

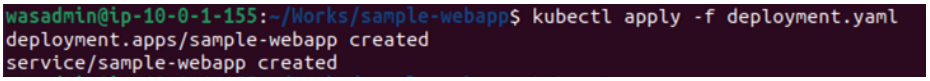
The second section exposes the deployed application as a service.

```
apiVersion: v1
kind: Service
metadata:
    labels:
        app: sample-webapp
    name: sample-webapp
    namespace: default
spec:
    ports:
    - port: 8080
    selector:
        app: sample-webapp
```

2. Run the following command to perform the deployment and expose the service:

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

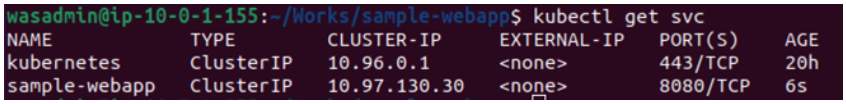
Notice it displays the following message:



3. Get the services list:

```
kubectl get services
```

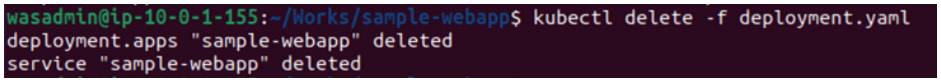
4. You should see the service is up and running.



5. *Optional Extra Credit Step*: Follow the steps detailed in **Part 2** to **ssh** into the Minikube cluster and verify that the website is running using the **curl** command.
6. Delete the service and deployment by running the following command:

```
kubectl delete -f deployment.yaml
```

Notice both the service and deployment are deleted.



5.7. Clean-Up

1. Close all open browser windows.
2. Switch to the Dashboard Terminal and press Ctrl+C to stop any processes running.
3. Type **exit** many times in all Terminals to close them.

5.8. Review

In this lab, you learned the basics of Kubernetes with minikube and kubectl.

Lab 6. Implementing the Sidecar Pattern

In this lab we will use Kubernetes to implement a pod that uses the sidecar pattern. The pod will have two containers:

1. Application Container - The application will be a simple script that appends some data to a log file every few seconds.
2. Sidecar Container - The application in the sidecar will be an nginx server that provides http access to the file that the application container's app is logging to.

Instead of providing HTTP access a real-world sidecar might copy the log file to a log aggregation server.

Make sure you run the following command:

```
docker login -u \{your-docker-id} -p \{your-access-token}
```

6.1. Setting the Stage

1. Open a new Terminal window.
2. Ensure you can access the Docker CLI by running the following:

   ```
   docker ps
   ```

3. In the Terminal, navigate to the Works directory you created

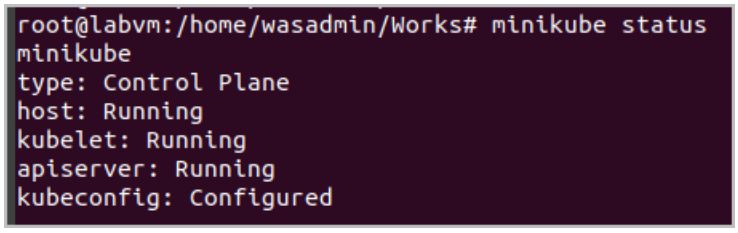
earlier:

```
cd ~/Works/
```

4. Check minikube status:

```
minikube status
```

You should see that minikube is Running:



If minikube is not running then start it using this command:

```
minikube start
```

6.2. Create an Application Pod

First we will create an application pod. The application in this pod will save data to a log file every five seconds. In this section we will create the application pod in kubernetes and shell into it to observe the work it does.

1. Create a file using VSCode:

```
code application-pod.yaml
```

2. Add the following content:

```
# application-pod.yaml
apiVersion: v1
kind: Pod
metadata:
    name: application-pod
spec:
    containers:
    - name: app-container
      image: ubuntu
      command: ["/bin/sh"]
      args: ["-c", "while true; do date >> /var/log/app.log;
sleep 5;done"]
```

3. Save the file and close the editor.
4. Create the application pod in kubernetes:

```
kubectl apply -f application-pod.yaml
```

5. Check that the pod was created:

```
kubectl get pods
```

You should get the following output:

```
wasadmin@ip-10-0-1-155:~/Works$ kubectl get pods
NAME                    READY   STATUS    RESTARTS      AGE
application-pod         1/1     Running   0             15s
nginx-7bf8c77b5b-zntz2  1/1     Running   1 (51m ago)   19h
```

6. Describe the pod:

```
kubectl describe pods application-pod
```

The command produces a large amount of output. Scroll through the output until you see the "containers" section. It should look like this:

```
Containers:
  app-container:
    Container ID:  docker://a3791c3888b50110cd60c8a8834dd7aa81ee32e4099efaaec3cfa2fd5885f1b5
    Image:         ubuntu
    Image ID:      docker-pullable://ubuntu@sha256:ec050c32e4a6085b423d36ecd025c0d3ff00c38ab93
a3d71a460ff1c44fa6d77
    Port:          <none>
    Host Port:     <none>
    Command:
      /bin/sh
    Args:
      -c
      while true; do date >> /var/log/app.log; sleep 5;done
    State:          Running
      Started:      Fri, 18 Aug 2023 18:48:02 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-w4xj2 (ro)
```

The pod has a single container named 'app-container' hosting an ubuntu Docker image and running a single line script (see Args) that appends the date to an app.log file. Next, we will check this log file to verify that the application is working.

7. Shell into the app-container using the following command:

```
kubectl exec application-pod -c app-container -it -- /bin/bash
```

The prompt will change to:

```
root@application-pod:/#
```

8. Check that the app.log file exists:

   ```
   ls -l /var/log/app.log
   ```

   Your output should look similar to this:

   ```
   -rw-r--r-- 1 root root 493 Dec 23 20:20 /var/log/app.log
   ```

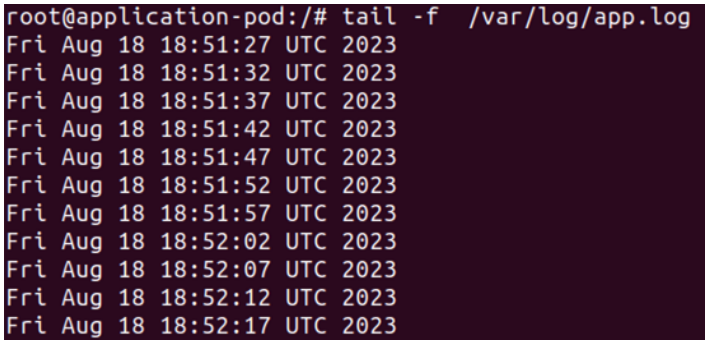
9. Try running the list command a few more times. You should notice that the size of the app.log file is larger each time:

   ```
   ls -l /var/log/app.log
   ```

10. Use the linux tail command to take a look at the app.log file's contents:

    ```
    tail -f /var/log/app.log
    ```

    The output should look similar to this:



The tail command displays the most recent lines from the end of the app.log file. If you wait a few moments you will see new dates as they are appended to the file by the application's script. This verifies that the application is working.

We would like a way to check this log without having to shell into the application container. To do that we will develop a sidecar container.

11. Hit **Ctrl-C** to stop the tail command.
12. Exit the container shell using the following command:

```
exit
```

6.3. Add a Second Container

In this section we will add a second container to the pod. The second container will hold an nginx web server.

1. First, delete the current deployment. The deletion might take a few moments. Wait for it to complete before moving on:

```
kubectl delete -f application-pod.yaml
```

2. Open the *application-pod.yaml* again in VSCode:

```
code application-pod.yaml
```

3. Add the content shown below:

```
...
        - name: sidecar-container
          image: nginx:latest
          ports:
            - containerPort: 80
```

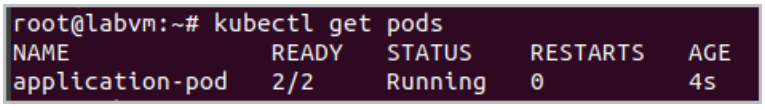
4. Save the file and close the editor. Be careful with the indentation.
5. Create the application pod in kubernetes using the updated yaml file:

```
kubectl apply -f application-pod.yaml
```

6. Check out the pods:

```
kubectl get pods
```

   You should get the following output:



READY should now say 2/2 meaning 2 out of 2 containers are running. Wait and repeat the command until you see 2/2.

7. Describe the pod:

```
kubectl describe pods application-pod
```

8. Scroll through the output until you see the "containers" section.

   You can see the second container here named "sidecar-container". It is running an nginx image. Let’s verify that the nginx server is working. We will do that by shelling into the second container.

9. Shell into the sidecar-container using the following command:

```
kubectl exec application-pod -c sidecar-container -it --
/bin/bash
```

   The prompt will change to:

```
root@application-pod:/#
```

10. Use the linux curl command to make a call to the nginx server.

```
curl http://localhost/index.html
```

11. The first few lines of output should be:

```
<!DOCTYPE html>
```

```
<html>
<head>
<title>Welcome to nginx!</title>
...
```

This is the nginx welcome page. If the server were not running, we would have received a 'Connection refused" message instead. This verifies that the nginx server is running in the second container.

At this point both containers are running but the nginx server does not have access to the app.log file which is on the first container's file system. Next, we will use a shared volume to solve this issue.

12. Exit the container shell using the following command:

```
exit
```

6.4. Create and Use a Shared Volume

For our design to work we need the application to place its log file somewhere that the nginx server can access it. We can use a shared volume for this. This will involve three steps:

1. Add a shared volume
2. Setup the app-container to use the shared volume
3. Setup the sidecar-container to use the shared volume

All of this can be done in the yaml file.

1. First, delete the current deployment. The deletion might take a few moments. Wait for it to complete before moving on:

```
kubectl delete -f application-pod.yaml
```

2. Edit the application-pod.yaml in VSCode:

```
code application-pod.yaml
```

3. Add the following in bold after the 'spec:' line and before the 'containers:' line:

```
spec:
    volumes:
    - name: shared-logs
      emptyDir: \{}
    containers:
```

   This establishes a shared volume called 'shared-logs'.

4. Add the following code in bold to the app-container after the 'args' line and before the '- name: sidecar-container' line:

```
args: ["-c", ...]
volumeMounts:
- name: shared-logs
  mountPath: /var/log
- name: sidecar-container
```

   This mounts the shared-logs volume to the /var/log directory in

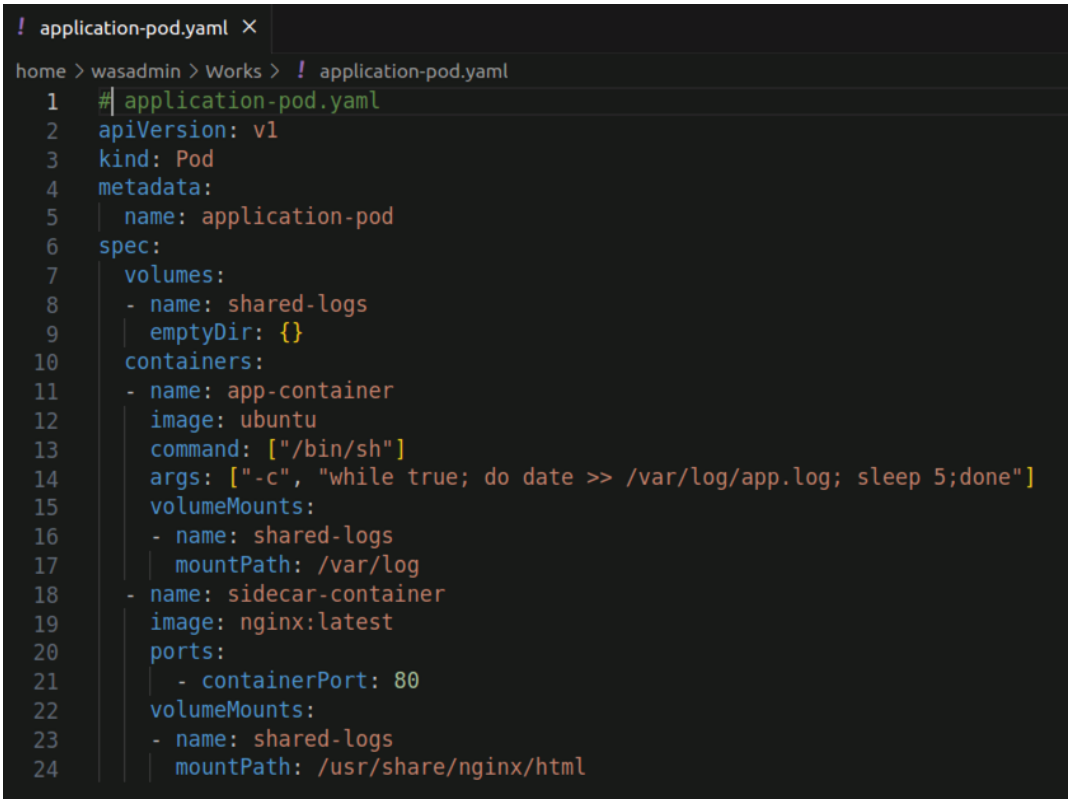
the app-container. Remember the application in that container is appending to /var/log/app.log. Now any file in that directory, including the app.log, will be saved to the root of the shared volume.

5. Add the following code in bold at the end of the sidecar-container after the '- containerPort' line:

```
ports:
    - containerPort: 80
volumeMounts:
- name: shared-logs
  mountPath: /usr/share/nginx/html
```

These lines tell kubernetes to mount the shared-logs volume to the **/usr/share/nginx/html** directory in the sidecar-container. The mount point is the same directory nginx serves files from. That means that any file in the root of the shared-logs volume is available to be served by nginx - including the app.log.

6. Save the file.
7. Verify that you have the right indentation in the file as shown below or it won't run:



8. Close the editor.

6.5. Deploy and Test the App and Sidecar

In this part we will update the application again. At this point it includes the app-container and sidecar-container with both accessing the app.log file from a shared volume. In testing we should see the contents of this file changing as we access it through the nginx server.

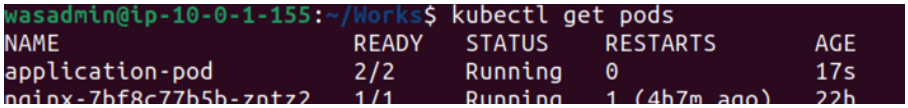
1. Create the application pod in kubernetes using the updated yaml file:

```
kubectl apply -f application-pod.yaml
```

2. Check out the pods:

```
kubectl get pods
```

   You should get the following output:



   READY should say 2/2 meaning both containers are running.

3. Shell into the sidecar-container using the following command:

```
kubectl exec application-pod -c sidecar-container -it --
/bin/bash
```

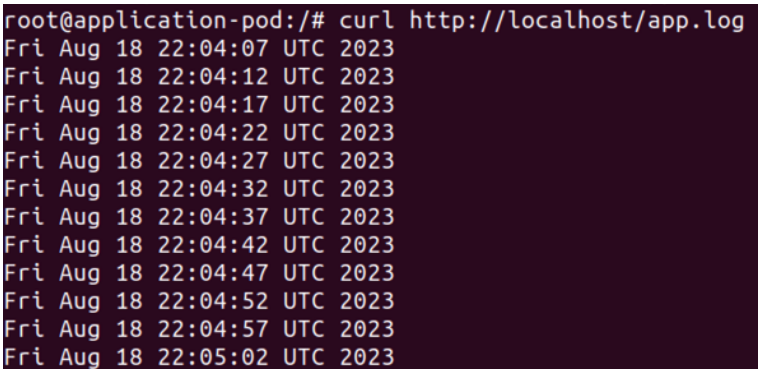
   The prompt will change to:

```
root@application-pod:/#
```

4. Use the linux curl command to make a call to the nginx server.

```
curl http://localhost/app.log
```

5. The output should be similar to this:



If you run the command a few times you will see the time on the last line is changing which shows that new dates are being appended.

We have successfully implemented a sidecar container that accesses a log file used by the application in the app-container.

6. Exit the sidecar-container shell using the following command:

```
exit
```

7. Delete the deployment from the terminal (make sure you exited the sidecar-container shell first):

```
kubectl delete -f application-pod.yaml
```

8. Close Terminal by typing **exit**.

6.6. Review

In this lab we implemented a pod that uses the sidecar pattern in Kubernetes. Our main application created logs while the sidecar

container provided HTTP access to the log file. As implemented, we still had to shell into the sidecar to access the log file. Adding a Kubernetes service would complete the design by allowing us to access the sidecar from outside the Kubernetes cluster.

Lab 7. Deploying Applications

In this lab, you will see how to Deployment Kubernetes applications. You will also perform various operations on the deployment such as upgrade, pause, resume, and scale.

Make sure you run the following command:

```
docker login -u \{your-docker-id} -p \{your-access-token}
```

7.1. Setting the Stage

1. Open a new Terminal window.
2. Ensure you can access the Docker CLI by running the following:

   ```
   docker ps
   ```

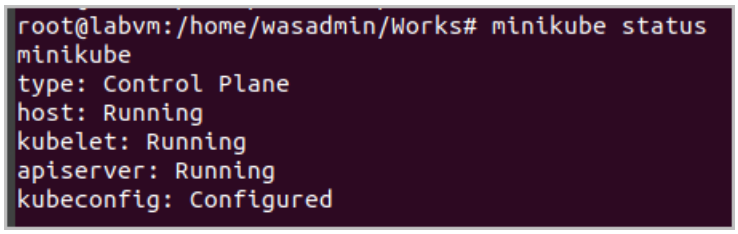
3. In the Terminal, navigate to the Works directory you created earlier:

   ```
   cd ~/Works/
   ```

4. Check minikube status:

   ```
   minikube status
   ```

You should see that minikube is Running:



If minikube is not running then start it using this command:

```
minikube start
```

7.2. Create a Deployment

1. Use VSCode to create a deployment manifest file. This command will create the file and bring it up in edit mode:

```
code nginx-deployment.yaml
```

2. Add the following contents to the file.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
    labels:
        app: nginx
spec:
    replicas: 3
    selector:
        matchLabels:
```

```
            app: nginx
    template:
        metadata:
            labels:
                app: nginx
    spec:
        containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
          - containerPort: 80
```

3. Save the file.

```
A Deployment named nginx-deployment is created, indicated by the
.metadata.name field.

The Deployment creates three replicated Pods, indicated by the
replicas field.

The selector field defines how the Deployment finds which Pods to
manage. In this case, you simply select a label that is defined in the
Pod template (app: nginx). However, more sophisticated selection rules
are possible, as long as the Pod template itself satisfies the rule.

The template field contains the following sub-fields:

* The Pods are labeled app: nginx using the labels field.
* The Pod template's specification, or .template.spec field, indicates
that the Pods run one container, nginx, which runs the nginx Docker
Hub image at version 1.7.9.
* Create one container and name it nginx using the name field.
```

1. Close the editor.
2. Create the Deployment by running the following command:

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

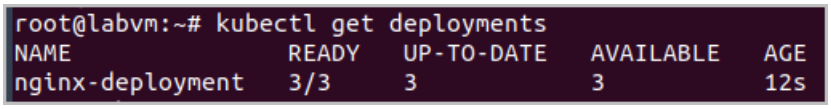
You should see:

```
deployment.apps/nginx-deployment created
```

3. Run the following command to verify the Deployment was created:

```
kubectl get deployments
```

You should see:



If you don't get the above result then it might take a few minutes for the nginx image to get downloaded, wait and don't move to the next step until you see 3/3.

4. To see the Deployment rollout status, run the following command:

```
kubectl rollout status deployment.v1.apps/nginx-deployment
```

The "deployment.v1.apps" prefix on the deployment name specifies the API version (v1.apps). You could also simply use "deployments/nginx-deployment". API versions allow Kubernetes to introduce changes without breaking existing clients. Different

versions can have different functionality. One API server can support multiple API versions simultaneously. You should see:

```
wasadmin@ip-10-0-1-155:~/Works$ kubectl rollout status deployment.v1.apps/nginx-deployment
deployment "nginx-deployment" successfully rolled out
```

5. To view the details of the deployment, run the following command:

```
kubectl get deployment nginx-deployment -o yaml
```

6. Labels are automatically generated for each Pod, to see what they are run:

```
kubectl get pods --show-labels
```

```
wasadmin@ip-10-0-1-155:~/Works$ kubectl get pods --show-labels
NAME                                READY   STATUS    RESTARTS      AGE   LABELS
nginx-7bf8c77b5b-zntz2              1/1     Running   1 (20h ago)   39h   app=nginx,pod-template-hash=7bf8c77b5b
nginx-deployment-f7599d4c-56lj6     1/1     Running   0             56s   app=nginx,pod-template-hash=f7599d4c
nginx-deployment-f7599d4c-5r8vk     1/1     Running   0             56s   app=nginx,pod-template-hash=f7599d4c
nginx-deployment-f7599d4c-dc5hr     1/1     Running   0             56s   app=nginx,pod-template-hash=f7599d4c
```

7.3. Update a Deployment

1. Run the following command to check the nginx version image used by the current Deployment:

```
kubectl describe deployment nginx-deployment
```

Notice image is displayed as 1.7.9

2. Open the Deployment manifest for editing:

```
code nginx-deployment.yaml
```

3. Change image version from 1.7.9 to 1.9.1
4. Change replicas count from 3 to 4
5. Save and close the editor.
6. Apply the updates:

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

   You will see:

   ```
   deployment.apps/nginx-deployment configured
   ```

7. Check rolling update status:

   ```
   kubectl rollout status deployment.v1.apps/nginx-deployment
   ```

   You will see:

   ```
   deployment "nginx-deployment" successfully rolled out
   ```

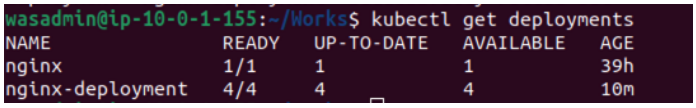
   

   It might take a few minutes for all replicas to get updated to the new nginx version.

8. View the Deployment:

```
kubectl get deployments
```

Ensure there are 4 replicas.



9. Verify nginx version is updated to 1.9.1:

```
kubectl describe deployment nginx-deployment
```

Ensure Image version is 1.9.1

7.4. Roll Back a Deployment

In this part, you will revert an update. One use-case where it can be useful is when you try to upgrade your Deployment to a version that doesn't exist. You can rollback such a deployment to make your application functional again by reverting to the previous version.

1. Suppose that you made a typo while updating the Deployment, by putting the image name as nginx:1.91 instead of nginx:1.9.1. Enter the following into the terminal:

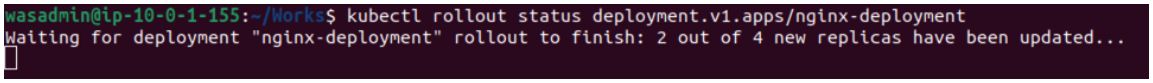
```
kubectl set image deployment.v1.apps/nginx-deployment
nginx=nginx:1.91
```

You should see:

```
deployment.apps/nginx-deployment image updated
```

2. The rollout gets stuck. You can verify it by checking the rollout status:

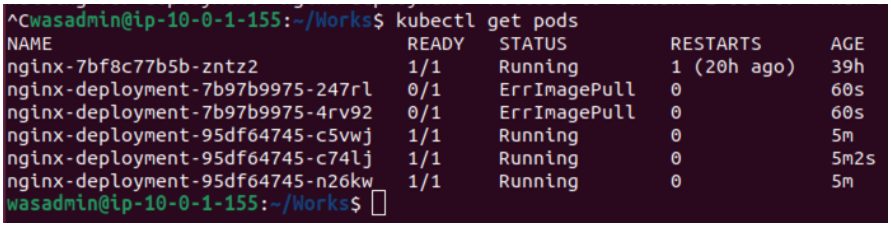
```
kubectl rollout status deployment.v1.apps/nginx-deployment
```



3. Press Ctrl+C to stop the above rollout status watch.
4. Run the following command to get pod list:

```
kubectl get pods
```

Looking at the Pods created, you see that Pod(s) created by the new ReplicaSet is stuck in an image pull loop.



5. Undo the recent change:

```
kubectl rollout undo deployment.v1.apps/nginx-deployment
```

You should see:

```
deployment.apps/nginx-deployment rolled back
```

6. Verify the invalid pods are removed:

```
kubectl get pods
```

Notice that the problematic pods are either in "Terminating" state or have completely disappeared after successfully terminating.

7. Scale a deployment by using imperative command:

```
kubectl scale deployment.v1.apps/nginx-deployment --replicas=1
```

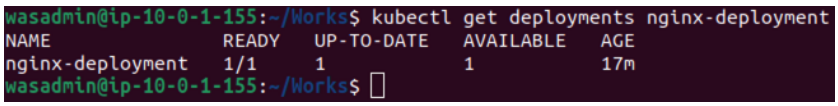
You should see:

```
deployment.apps/nginx-deployment scaled
```

8. Verify scaling is configured:

```
kubectl get deployments nginx-deployment
```

You should see:



7.5. Pausing and Resuming a Deployment

You can pause a Deployment before triggering one or more updates and then resume it. This allows you to apply multiple fixes in between pausing and resuming without triggering unnecessary rollouts.

1. Pause the Deployment by running the following command:

```
kubectl rollout pause deployment.v1.apps/nginx-deployment
```

2. While the rollout is paused, set the Deployment image to a different version:

```
kubectl set image deployment.v1.apps/nginx-deployment
nginx=nginx:1.9.2
```

3. Run the following command to verify the image version:

```
kubectl rollout status deployment.v1.apps/nginx-deployment
```

This command will keep showing you “Waiting for deployment” since the rollout is paused.

4. Press Ctrl+C to exit out to the terminal.
5. Resume rollout:

```
kubectl rollout resume deployment.v1.apps/nginx-deployment
```

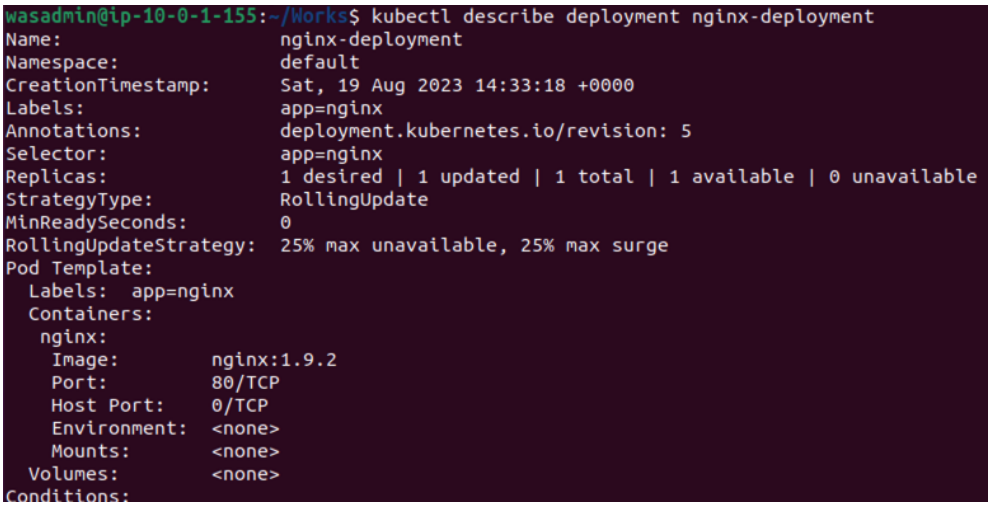
6. Verify the rollout status:

```
kubectl rollout status deployment.v1.apps/nginx-deployment
```

   You may have to wait a few seconds for the command to be completed.

7. Verify the Deployment image version:

```
kubectl describe deployment nginx-deployment
```



The Image version should show as 1.9.2

7.6. Clean-Up

1. Delete your Deployment:

```
kubectl delete deployments/nginx-deployment
```

2. Verify the deployment is deleted:

```
kubectl get deployments
```

3. Verify the nginx-deployment pods are deleted, you may see other pods:

```
kubectl get pods
```

It might take a few moments for the pods to get deleted. You may see other pods left.

4. Type **exit** to exit the Terminal.

7.7. Review

In this lab, you Deployed Kubernetes applications and performed various operations such as upgrade, pause, resume, and scale on the Deployment.

Lab 8. Implementing RBAC Security

In this lab, you will implement security using RBAC. You will see how to create users with basic security and also with certificates. You will explore how to create contexts so you can switch between users. You will also implement authorization by using Role and RoleBinding. A role is essentially the permissions that you want to grant and a role binding is assigning the permissions to a specific user or group.

Make sure you run the following command:

```
docker login -u \{your-docker-id} -p \{your-access-token}
```

8.1. Setting the Stage

1. Open a new Terminal window.
2. Ensure you can access the Docker CLI by running the following:

   ```
   docker ps
   ```

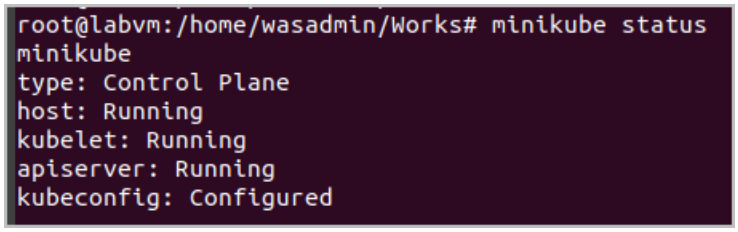
3. In the Terminal, navigate to the Works directory you created earlier:

   ```
   cd ~/Works/
   ```

4. Check minikube status:

```
minikube status
```

You should see that minikube is Running:



If minikube is not running then start it using this command:

```
minikube start
```

8.2. Downloading and Prepping the Security Project

In this part, you will download the security project from the web and unzip it.

1. Make sure you are in the **Works** folder:

```
cd ~/Works
```

2. Download the security project by using the following **wget** command::

```
wget https://course-sw.s3.amazonaws.com/ecurity.zip
```

3. Open a browser.
4. Enter the following link to download the application:

```
https://course-sw.s3.amazonaws.com/security.zip
```

5. Extract the contents of **security.zip**:

```
unzip security.zip
```

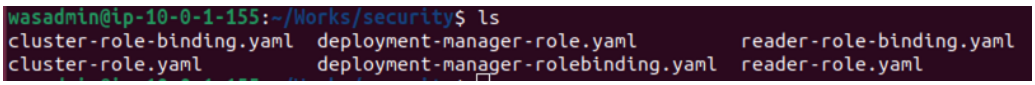
6. Change to the **security** subdirectory:

```
cd security
```

7. Verify the files were extracted:

```
ls
```

You should see the files that you just extracted:



8.3. View the Cluster Configuration and Access the API Server

In this part, you will view the Kubernetes configuration and access the API server using cURL. In a real-world scenario, you won't use cURL.

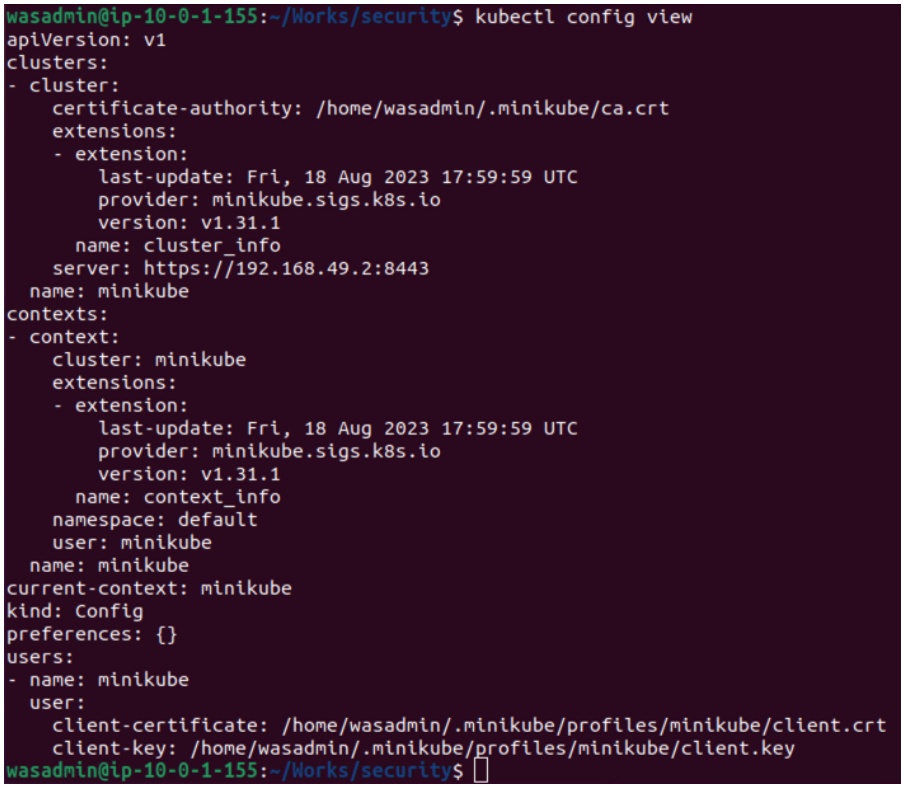
Instead, you will use kubectl and various other commands to communicate with the API server.

1. Run the following command to view Kubernetes configuration:

   ```
   kubectl config view
   ```

   The above command displays the contents of ~/.kube/config file.

   Notice by default, there's just one user/principal, minikube in this case, and there's one context. A context represents a combination of Kubernetes cluster and user/principal that can connect to a cluster.



   The file ca.crt represents the CA used by the cluster and the files, client.crt, and client.key, maps to the user minikube. Kubectl uses

these certificates and keys from the current context to encode the request.

2. To add a new user/principal using a plain-text password (basic security), run the following command:

```
kubectl config set-credentials alice --password=abc123
--username=alice
```

3. View the Kubernetes configuration:

```
kubectl config view
```

Notice there are now 2 users. Alice is added as a user with a redacted plain-text password whereas the default user is added with a certificate.

8.4. Create a Certificate Signing Request and Verify the CSR

In the next few parts, you will create another user/principal. This time you will use a key and certificate. In this part, you will create CSR and verify it. The CSR will be assigned a user/principal, Joe, later in the lab.

1. Run the following command to generate a private key for Joe:

```
openssl genrsa -out joe.key 2048
```

2. Run the following command to create a Certificate Signing Request from the private key:

```
openssl req -new -key joe.key -out joe.csr -subj
"/CN=joe/O=sales"\n
```

3. View Kubernetes configuration and locate the directory where certificates are stored:

```
kubectl config view
```

   Make a note of the folder as highlighted in the screenshot.

4. You will use the folder you noted in the previous part quite often throughout this lab. Let's create a variable and store the location, enter the following command:

```
LOCATION=/home/wasadmin/.minikube
```

5. Generate the final certificate employee.crt by approving the

certificate sign request, joe.csr, you made earlier:

```
openssl x509 -req -in joe.csr -CA $LOCATION/ca.crt -CAkey
$LOCATION/ca.key -CAcreateserial -out joe.crt -days 500
```

[Enter the command in 1 line]

You should see:

```
Certificate request self-signature ok
subject=CN = joe, O = salesn
```

6. Verify the joe.crt file contains the certificate issued by Kubernetes:

```
cat joe.crt
```

You should see:

```
-----BEGIN CERTIFICATE-----
<YOUR CERTIFICATE CONTENT>
-----END CERTIFICATE-----
```

8.5. Create a User/Principal with the Key and Certificate

In this part, you will create a Kubernetes user/principal with the key and certificate you generated in the previous parts of the lab.

1. Add Joe as a user to Kubernetes:

```
kubectl config set-credentials joe --client-certificate=joe.crt
--client-key=joe.key
```

[Enter the command in 1 line]

Verify it shows the following message:

```
User "joe" set.
```

2. View Kubernetes configuration and verify Joe is listed:

```
kubectl config view
```

Notice it shows the following:

```
- name: joe +
  user: +
    client-certificate: <YOUR_WORKING_DIRECTORY>/security/joe.crt
+
    client-key: <YOUR_WORKING_DIRECTORY>/security/joe.key
```

8.6. Create Contexts to Easily Switch Between Users

In this part, you will create contexts so you can easily switch between the users. You have two users: alice and joe. Unlike OpenShift that let

you use the oc login command to login as a different user, Kubernetes relies on contexts and the kubectl command.

Before creating the contexts, let's create a namespace. Namespaces are like projects that let you group related resources together. For example, order-management-system namespace could hold order, shipping, and invoicing deployments (pods + services). You can grant authorization at a namespace level and assign project-specific users.

1. Create a Sales namespace:

   ```
   kubectl create namespace sales
   ```

2. Verify the namespace has been created:

   ```
   kubectl get namespaces
   ```

   You should see:

   Let's perform a deployment so that we can later verify if we can see it as a certain user, such as joe and alice.

3. Let's ensure that any previous deployment called **nginx** is removed. Run the following. You will either get a confimation of the delete or an "not found" error if it didn't exist. Either way is

fine.

```
kubectl delete deployment nginx
```

4. Deploy a sample image to the sales namespace:

```
kubectl create deployment nginx --image=nginx --namespace=sales
```

5. Run the following command to create a context for Joe that should limit Joe's permissions to a custom namespace named **sales**:

```
kubectl config set-context _joe_-context --namespace=sales
--user="_joe_" --cluster="minikube"
```

[Enter the command in 1 line]

Notice it shows the message: Context "joe-context" created.

6. Create a context for Alice:

```
kubectl config set-context alice-context --namespace=sales
--user="alice" --cluster="minikube"
```

[Enter the command in 1 line]

Notice it shows the message: Context "alice-context" created.

7. What exactly happened when you executed the above two commands? Let's find out. Run the following command to view the Kubernetes configuration:

```
kubectl config view
```

You have 3 contexts. The default is the last one in the contexts section. The first context will let you connect to the minikube cluster as alice and the second one lets you connect to the minikube cluster as joe.

```
contexts:
- context:
    cluster: minikube
    namespace: sales
    user: alice
  name: alice-context
- context:
    cluster: minikube
    namespace: sales
    user: joe
  name: joe-context
- context:
    cluster: minikube
    extensions:
    - extension:
        last-update: Fri, 18 Aug 2023 17:59:59 UTC
        provider: minikube.sigs.k8s.io
        version: v1.31.1
      name: context_info
    namespace: default
    user: minikube
  name: minikube
current-context: minikube
```

8. Let’s test the contexts. Run the following command to check the current context:

```
kubectl config current-context
```

Notice **minikube** is displayed which means you are using the cluster admin’s context and have access to all Kubernetes cluster resources.

9. Verify you can get the deployments list as the cluster-admin:

```
kubectl get deployments -n=sales
```

10. Even as a cluster admin, you can run commands as a different user without actually switching the context. Run the following command to get the deployments list as Joe:

    ```
    kubectl get deployments -n=sales --as joe-context
    ```

    Notice it's forbidden to access the resources as joe.

    The alternative to the above command is:

    ```
    kubectl get deployments --context=joe-context
    ```

11. If you don't want to pass --as <user> with each command, you can switch the context to another user. Run the following command to use the context created for Joe:

    ```
    kubectl config use-context joe-context
    ```

    Notice it shows the message:

    ```
    Switched to context "joe-context".
    ```

12. Verify the current context is set to joe-context:

```
kubectl config get-contexts
```

You should see:

13. Try viewing existing deployments as Joe:

```
kubectl get deployments
```

Notice it shows an error: the server doesn't have a resource type "deployments"

The error means the current context, user, doesn't have access to the deployments resource type.

Also notice that you didn't have to specify --namespace=sales since the context is already configured to use the sales namespace by default.

14. Switch back to the default, minikube, context so you can run commands as the cluster-admin:

```
kubectl config use-context minikube
```

Ensure the minikube context is set.

In the next few steps, you will use an alternative technique to

check if a user/context has access to the sales namespace.

15. Check if the cluster-admin has permission to access the sales namespace:

```
kubectl auth can-i list pods --namespace sales
```

Ensure it says **yes**

16. Get the pods list as the cluster-admin.

```
kubectl get pods -n=sales
```

You should be able to see the nginx deployment-based pod.

17. Check if joe has permission to access the sales namespace:

```
kubectl auth can-i list pods --namespace sales --as _joe_
```

Notice it says **no**

You are yet to authorize the users to access the sales namespace. You will do that in the next part.

8.7. Authorize the User to Access the Namespace

In this part, you will create Role and RoleBinding to authorize Joe to access the sales namespaces.

1. View the contents of reader-role.yaml file:

```
cat reader-role.yaml
```

the configuration file creates a custom sales-reader role in the sales namespace with the read-only permission to access pods, services, and deployments. The contents of the file are listed here for reference.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
    namespace: sales
    name: sales-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods", "services", "deployments"]
  verbs: ["get", "watch", "list"]
```

2. Run the YAML configuration file to create a Role:

```
kubectl apply -f reader-role.yaml
```

3. Run the following command to verify a custom sales-reader role has been created in the sales namespace:

```
kubectl get roles --namespace=sales
```

Notice it shows the following output:

```
NAME            CREATED AT
sales-reader    <date_and_time_stamp>
```

4. View the contents of reader-rolebinding.yaml file:

```
cat reader-role-binding.yaml
```

The configuration file creates a custom role-binding in the sales namespace and binds the user Joe to the custom sales-reader role.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
    name: sales-read-access
    namespace: sales
subjects:
- kind: User
  name: joe
  apiGroup: ""
roleRef:
  kind: Role
  name: sales-reader
  apiGroup: ""
```

5. Run the YAML configuration file to create a role binding:

```
kubectl apply -f reader-role-binding.yaml
```

Using the YAML configuration technique is the best practice. But, if you want to do the same thing without using the YAML configuration file you can also do it imperatively using the kubectl CLI.

6. Run the following command to verify a custom role binding has been created in the sales namespace:

```
kubectl get rolebindings --namespace=sales
```

Notice it shows the following output:

```
NAME                ROLE                AGE
sales-read-access   Role/sales-reader   <time>
```

7. Verify you *cannot* access the sales namespace as Alice:

```
kubectl get pods --namespace sales --as alice
```

8. Verify you can access the sales namespace as Joe:

```
kubectl get pods --namespace sales --as joe
```

9. Switch to Joe's context so we can verify it's a read-only role that has been assigned to Joe:

```
kubectl config use-context joe-context
```

10. Try to perform another deployment and verify you cannot do it as Joe.

```
kubectl create deployment nginx2 --image=nginx
```

Notice it shows the following error message:

> Error from server (Forbidden): deployments.apps is forbidden: User "joe" cannot create resource "deployments" in API group "apps" in the namespace "sales"

11. Let's promote Joe's role to a deployment manager so he can perform deployments. Before you can do that, you need to switch to the cluster admin context:

```
kubectl config use-context minikube
```

12. View deployment-manager-role.yaml:

```
cat deployment-manager-role.yaml
```

The configuration file creates a custom deployment-manager role in the sales namespace with permissions to manage deployments. The contents are the file are listed here for reference.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
```

```
    namespace: sales
    name: deployment-manager
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["deployments", "replicasets", "pods"]
  verbs: ["get", "list", "watch", "create", "update", "patch",
"delete"]
z
```

13. Apply the YAML configuration to create a custom deployment manager role:

```
kubectl apply -f deployment-manager-role.yaml
```

14. View deployment-manager-rolebinding.yaml:

```
cat deployment-manager-rolebinding.yaml
```

The configuration file creates a custom role-binding in the sales namespace and binds the user Joe to the custom deployment-manager role.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
    name: deployment-manager-binding
    namespace: sales
subjects:
- kind: User
  name: joe
  apiGroup: ""
roleRef:
```

```
  kind: Role
  name: deployment-manager
  apiGroup: ""
```

15. Apply the YAML configuration to create a custom deployment manager role:

```
kubectl apply -f deployment-manager-rolebinding.yaml
```

16. Switch back to Joe's context and see if we can perform a deployment:

```
kubectl config use-context joe-context
```

17. Perform a deployment:

```
kubectl create deployment nginx2 --image=nginx
```

    Ensure you use nginx2 since you already have a deployment named nginx.

18. Delete the deployments as Joe:

```
kubectl delete deployment nginx

kubectl delete deployment nginx2
```

19. Switch back to the main cluster-admin context:

```
kubectl config use-context minikube
```

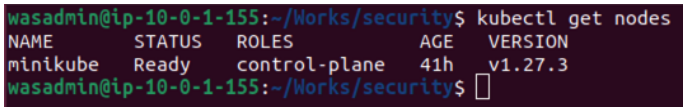
8.8. Assign ClusterRole

In the previous part, you authorized Joe and Alice to access the sales namespace. Likewise, you can authorize users to access additional namespaces. In case, if you want a user to have access to the cluster-level resources, such as nodes, you have to use CluserRole. In this part, you will assign ClusterRole to a user so the entire cluster can be managed.

1. Try accessing nodes as a cluster-admin:

   ```
   kubectl get nodes
   ```

   It should show a node like this:

2. Try accessing the nodes as Joe:

   ```
   kubectl get nodes --as joe
   ```

   You will get Error from server (Forbidden): nodes is forbidden: User "joe" cannot list resource "nodes" in API group "" at the cluster scope

3. View the cluster-role.yaml file:

```
cat cluster-role.yaml
```

The configuration file creates a custom cluster-node-reader role with a read-only permission to the cluster.

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: cluster-node-reader
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "watch", "list"]
```

4. Run the YAML configuration file to assign a cluster-level role to Joe:

```
kubectl apply -f cluster-role.yaml
```

5. Verify a cluster role has been created:

```
kubectl get clusterroles cluster-node-reader
```

6. View cluster-role-binding.yaml:

```
cat cluster-role-binding.yaml
```

The configuration file creates a custom role-binding named read-cluster-nodes and binds the user Joe to the custom cluster-node-reader role.

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
name: read-cluster-nodes
subjects:
- kind: User
name: joe
apiGroup: rbac.authorization.k8s.io
roleRef:
kind: ClusterRole
name: cluster-node-reader
apiGroup: rbac.authorization.k8s.io
```

7. Apply the YAML configuration file to create a cluster role binding, i.e. authorize Joe to perform operations on nodes:

```
kubectl apply -f cluster-role-binding.yaml
```

8. Verify the cluster role binding has been created:

```
kubectl get clusterrolebindings read-cluster-nodes
```

9. Access the nodes list as Joe and ensure you can view the node information:

```
kubectl get nodes --as joe
```

```
root@labvm:/home/wasadmin/Works/security# kubectl get nodes --as joe
NAME    STATUS   ROLES                  AGE    VERSION
labvm   Ready    control-plane,master   153d   v1.20.2
```

10. Access the nodes list as Alice and ensure you *cannot* view the node information:

```
kubectl get nodes --as alice
```

8.9. Clean-Up

In this part, you will delete all custom roles, rolebindings, and users that you created in the previous parts of this lab.

1. Switch to the main cluster admin context:

```
kubectl config use-context minikube
```

2. Delete all roles and rolebindings by running the following command:

```
kubectl delete -f .
```

   Don’t forget the dot at the end.

3. Delete all custom contexts:

```
kubectl config unset contexts.alice-context

kubectl config unset contexts.joe-context
```

4. Delete all custom users:

```
kubectl config unset __user__s.alice

kubectl config unset __user__s._joe_
```

5. Type **exit** to close the Terminal.

8.10. Review

In this lab, you explored how to implement RBAC security in a Kubernetes cluster.

Lab 9. Accessing Applications

Kubernetes is designed to easily run and scale multi-part applications. For that to happen it needs to allow the different parts of an application to communicate. In this lab we'll explore how the underlying Kubernetes network makes this possible.

We will start by deploying two applications - to keep things simple we will use a basic nginx web server image for these apps. Each of these runs on a linux image that we can shell into to show the access we have to our apps from within the kubernetes cluster. Later we will expose the apps so they can be accessed from outside the cluster.

Make sure you run the following command:

```
docker login -u \{your-docker-id} -p \{your-access-token}
```

9.1. Setting the Stage

1. Open a new Terminal window.
2. Ensure you can access the Docker CLI by running the following:

   ```
   docker ps
   ```

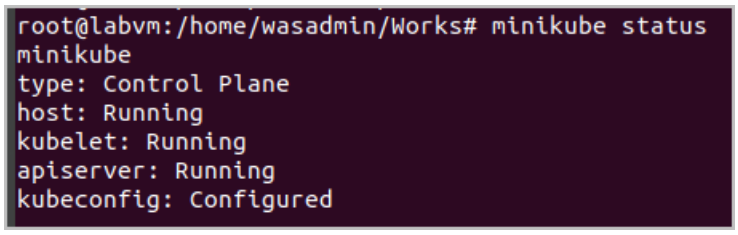
3. In the Terminal, navigate to the Works directory you created earlier:

```
cd ~/Works/
```

4. Check minikube status:

```
minikube status
```

You should see that minikube is Running:



If minikube is not running then start it using this command:

```
minikube start
```

9.2. Create Deployment NX1

You will deploy the nginx application and give it the name 'nx1'.

We will refer to the current terminal as nx1

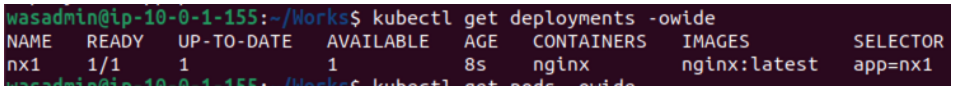
1. Run the following command to create the nx1 deployment:

```
kubectl create deployment nx1 --image nginx:latest
```

2. Verify that the deployment has been added:

```
kubectl get deployments -owide
```

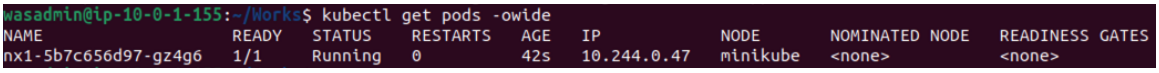
You should see:



3. Check that a pod was created for the deployment and is running:

```
kubectl get pods -owide
```

You should see:



4. Open a shell into the Pod. *Make sure to use the name listed in the pod command from the previous step - yours will differ from the one you see here*:

```
kubectl exec -it nx1-5b7c656d97-gz4g6 -- /bin/bash
```

This will open up a command prompt (again, your prompt name will differ): `root@nx1-5b7c656d97-gz4g6:/#`

5. Execute the following commands at the prompt to install the 'nano' editor:

```
apt-get update
apt-get install nano
```

In the next few commands we will modify nginx's index.html file and customize it for this instance of the application.

6. Execute the following command. It will open index.html up in the nano editor:

```
nano /usr/share/nginx/html/index.html
```

7. Use Ctrl-K *repeatedly* to delete all the current contents of the file.
8. Type the following into the file:

```
Welcome from NX1!
```

9. Enter these commands in this order to save the file:

```
Ctrl-x
y

[hit Enter]
```

10. Check out the file and verify that the changes were saved:

```
cat /usr/share/nginx/html/index.html
```

It should return:

```
Welcome from NX1!
```

11. Now we will try calling the nginx server using the curl command:

    ```
    curl http://localhost/index.html
    ```

    The nginx web server should return:

    ```
    Welcome from NX1!
    ```

12. Leave this terminal window open, as is.

9.3. Create Deployment NX2

Deploy the nginx application and give it the name 'nx2'.

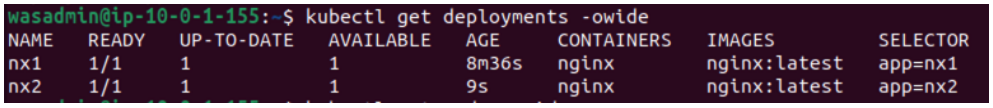
1. Open a new Terminal window.
    - We will refer to this terminal as NX2
2. FYI: The rest of the following instructions are similar to those in the previous section except for the name 'nx2'. Also, all these instructions should be entered into the 'NX2' terminal except where otherwise noted.
3. Run the following command to create the nx2 deployment:

    ```
    kubectl create deployment nx2 --image nginx:latest
    ```

4. Verify that the deployment has been added:

```
kubectl get deployments -owide
```

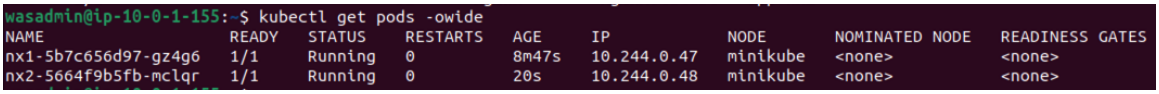
You should see:



5. Check that a pod was created for the deployment and is running:

```
kubectl get pods -owide
```

You should see:



6. Open a shell into the Pod. It's the one with the **nx2** prefix. Make sure to use the name listed in the pod command from the previous step - yours will differ from the one you see here:

```
kubectl exec -it nx2-5664f9b5fb-mclqr -- /bin/bash
```

This will open up a command prompt (again, yours will differ): `root@nx2-5664f9b5fb-mclqr:/#`

7. Execute the following commands at the prompt to install the 'nano' editor:

```
apt-get update
apt-get install nano
```

```
[Press Y when prompt to continue]
```

In the next few commands we will modify nginx's index.html file and customize it for this instance of the application.

8. Execute the following command. It will open index.html up in the nano editor:

```
nano /usr/share/nginx/html/index.html
```

9. Use Ctrl-K *repeatedly* to delete all the current contents of the file.
10. Type the following into the file:

```
Welcome from NX2!
```

11. Enter these command in this order to save the file:

```
Ctrl-x
y

[Press Enter]
```

12. Check out the file and verify that the changes were saved:

```
cat /usr/share/nginx/html/index.html
```

It should return:

```
Welcome from NX2!
```

13. Now we will try calling the nginx server using the curl command:

    ```
    curl http://localhost/index.html
    ```

    The nginx web server should return:

    ```
    Welcome from NX2!
    ```

9.4. Check Current Network Situation

In this part we will check to see what sort of visibility we have between the two deployments. We will check to see what was created for each deployment as well.

1. Open a third Terminal window.
    - We will refer to this terminal as kubectl
2. Run this command to get information about the node that Kubernetes is running on:

    ```
    kubectl describe nodes
    ```

3. Scroll down until you come to this line "Non-terminated Pods: ...". You should see pods in the default namespace for the two deployments, nx1 & nx2. This is evidence that the two pods are

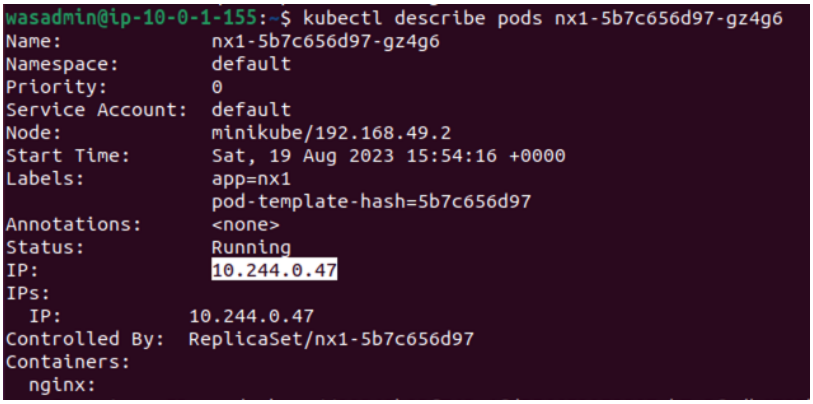
running on the same node.

```
Non-terminated Pods: (13 in total)
    Namespace       Name    CPU Requests        CPU Limits
    ---------       ----    ------------        ----------
    default         nx1-5b7c656d97-gz4g6        0 (0%) 0 (0%)
    default         nx2-5664f9b5fb-mclqr        0 (0%) 0 (0%)
    kube-system     coredns-5c98db65d4-5s5zq    100m (5%) 0 (0%)
```

4. Run this command to describe the nx1 pod, remember your pod names will be different:

```
kubectl describe pods nx1-5b7c656d97-gz4g6
```

Below are some of the more important lines and sections when it comes to networking. Try to locate them in your own output:



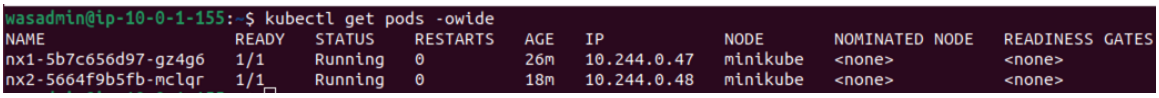
The IP address listed here is the address where the pod can be found *inside* the cluster. We can also find that information with the 'kubectl get pods -owide' command.

Take note of the IP address. You will use this in a later step. Yours will be different from the screenshot here.

5. Call the following command and check the IP address for nx1:

```
kubectl get pods -owide
```

This returns:



How can we verify that the IP we see here points to nx1? Well it should point to the container where we are running our first instance of nginx. so, we can try using it to make a call to nginx.

6. Switch to the Terminal named as *nx1 *_(you can always identify the terminal by looking at the prompt. The prompt in nx1 starts with "root@nx1-....."_.

7. Remember the 'nx1' terminal is running a shell inside the cluster. We can try calling our web server from there (replace the IP with the IP in your VM):

```
curl http://10.244.0.47
```

It should return:

```
Welcome to NX1!
```

Here we are making the call from the nx1 pod and accessing the nginx web server application in the same pod.

8. Switch to the Terminal named as **nx2**.

9. Try calling our web server again (replace the IP with the IP in your VM):

   ```
   curl http://10.244.0.47
   ```

   This should also return:

   ```
   Welcome to NX1!
   ```

   In this case we are making the call from the nx2 pod and it is successfully communicating with the nginx application in the nx1 pod! This is evidence that pods have their own IP addresses within the cluster and that the pods can see each other. By the way we could do the same with nx2, if you try calling it based on its IP address it should return the index.html page we customized for nx2.

   Imagine you had a RESTful web service running on one pod and a related data store in another pod. The web service would see the pod with the data store and would be able to pull data from it as needed.

   It's great that applications in the various pods can see each other but what if we want to access the application from outside the cluster? Let's give it a try.

10. Switch to the terminal named as **kubectl**. *(This is the third terminal. Since you didn't shell into either of the pods, the prompt here should start with "wasadmin@...")*
11. The shell here is NOT inside the cluster. It is outside. How do we

know? Try running this command:

```
kubectl cluster-info
```

It brings back information about the cluster itself. This would not have worked from inside the cluster. The kubectl command always runs from outside the cluster.

9.5. Exposing Pods Outside the Cluster

In this section we will add services to expose our nginx pods to the network outside the cluster.

1. Switch to the terminal named as **kubectl**.
2. Change directory to Works:

   ```
   cd ~/Works
   ```

3. Create and edit a file named nx1-service.yaml using the following command:

   ```
   code nx1-service.yaml
   ```

   This will bring up the gedit gui editor.

4. Enter the following text into the editor:

   ```
   apiVersion: v1
   ```

```
kind: Service
metadata:
    name: nx1
spec:
    selector:
        app: nx1
    ports:
    - port: 80
      protocol: TCP
    type: NodePort
    externalTrafficPolicy: Cluster
```

5. Save the file and close the editor.
6. Run the following command. It will create a Kubernetes service for nx1:

```
kubectl apply -f nx1-service.yaml
```

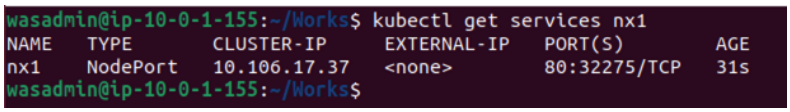
The response should be:

```
service/nx1 created
```

7. Check that the service exists with this command:

```
kubectl get services nx1
```

This should return (your cluster-ip value might be different):

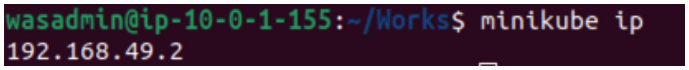


Take note of the port. *In the case above, it is 32275. Yours will be different.*

8. Get the cluster ip:

```
minikube ip
```

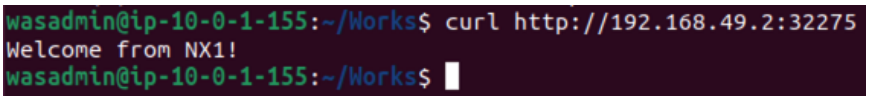
It returns (your minikube ip might be different.):



9. Make a call to the port you noted using *your* minikube ip: (your port number may be different. Refer to the output above to check the port number)

```
curl http://**<Minikube IP>:NodePort
```

This should return:



This is evidence of accessing **nx1** from outside the cluster!

9.6. Cleanup

1. Close the **nx1** and **nx2** terminal windows by using the command **exit** as many times as needed.
2. Delete the **nx1** service from the **kubectl** terminal:

```
kubectl delete service nx1
```

3. Delete both deployments from the **kubectl** terminal:

```
kubectl delete deployment nx1
kubectl delete deployment nx2
```

4. Use **exit** to close the **kubectl** terminal

9.7. Review

In this lab we deployed two applications. The deployment process created pods within which ran containers where instances of the nginx web server were running. We gained shell access to the containers and were able to call the servers from within the cluster network.

The nginx servers were not accessible from outside the cluster network until we created kubernetes services. Then we were able to access each nginx server off separate ports on the cluster IP address.

Lab 10. Troubleshooting

When deploying objects to Kubernetes things can go wrong in various ways. In this lab we will take a look at some errors and how to fix them. We will troubleshoot various issues that come up when deploying an application including:

- Yaml Syntax Exceptions
- nodeSelector Issues
- Failed Image Pull
- Resource Issues while Scaling

For each issue we will see how the issue appears, look into errors and warning, fix the issue and re-test to verify the fix.

Make sure you run the following command:

```
docker login -u \{your-docker-id} -p \{your-access-token}
```

10.1. Setting the Stage

1. Open a new Terminal window.
2. Ensure you can access the Docker CLI by running the following:

   ```
   docker ps
   ```

3. In the Terminal, navigate to the Works directory you created

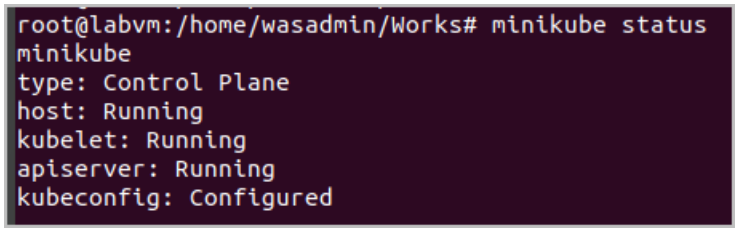
earlier:

```
cd ~/Works/
```

4. Check minikube status:

```
minikube status
```

You should see that minikube is Running:



If minikube is not running then start it using this command:

```
minikube start
```

10.2. Deploy an Application and Fix Yaml Syntax

In this section we will work to deploy an application based on an nginx web server.

1. Create a deployment specification using VSCode:

```
code webapp-deployment.yaml
```

2. Enter the following content into the file. Enter the text EXACTLY as it is shown here. Some lines contain error on purpose so that we can demonstrate those errors and how to troubleshoot them:

```
# app-deployment.yaml
# v0
apiVersion: apps/v1
kind: Deployment
metadata:
    name: webapp-deployment
    labels:
    app: nginx
spec:
    replicas: 1
    selector:
        matchLabels:
            app: nginx
    template:
        metadata:
            labels:
                app: nginx
        spec:
            nodeSelector:
                kubernetes.io/hostname: mycluster
            containers:
            - name: webapp
              image: ngix:latest
              ports:
              - containerPort: 80
              env:
              resources:
              requests:
                memory: "64Mi"
```

```
                cpu: "125m"
              limits:
                memory: "128Mi"
                cpu: "250m"
```

3. Save and close the file.
4. Apply the deployment yaml file:

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

   An error message will be returned:

   ```
   Error from server (BadRequest): error when creating "webapp-
   deployment.yaml": Deployment in version "v1" cannot be handled as
   a Deployment: strict decoding error: unknown field "metadata.app"
   ```

   One part of the message is most important to us: 'unknown field "app"' This tells us that there is a problem in the yaml file related to the text "app" in the metadata section of the file.

5. Open up webapp-deployment.yaml in a text editor:

   ```
   code webapp-deployment.yaml
   ```

6. Take a look at the metadata section and locate the text 'app'. The section should look like this:

   ```
   metadata:
   name: webapp-deployment
   labels:
   ```

```
app: nginx
```

The problem is that 'app: nginx' is a label and based on yaml syntax it should be indented from the 'labels:' tag above it. Add an indentation (two spaces should do it.) like this:

```
metadata:
name: webapp-deployment
labels:
app: nginx
```

7. Make the change shown above then save the yaml file and exit the editor.
8. Re-run the failed apply statement:

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

You should now get the output:

```
deployment.apps/webapp-deployment created
```

Our first problem, a YAML syntax issue, has been fixed.

10.3. Troubleshooting a nodeSelector Issue.

We completed our last section by deploying the webapp-application. It is time now for us to check and make sure the deployment completed as expected. Some of the automated steps involved include:

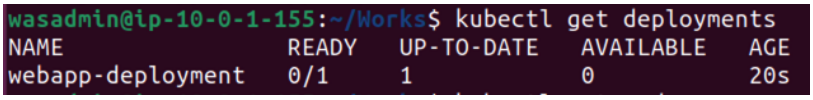
- Deploying Pods
- Pulling images
- Creating containers

If something goes wrong with any of these things the application will be unusable.

1. Check on deployments:

   ```
   kubectl get deployments
   ```

   You should see the following output:

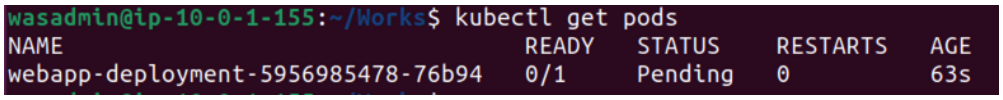
   

   The READY column shows that zero of one pods were deployed.

2. Let's verify that by checking out the pods:

   ```
   kubectl get pods
   ```

   You should see the following output: (your pod name will be different):

   The READY column is indeed the same here and the STATUS is 'pending'

3. Drill down and get more details by describing the pod listed above

(use the pod name from your machine):

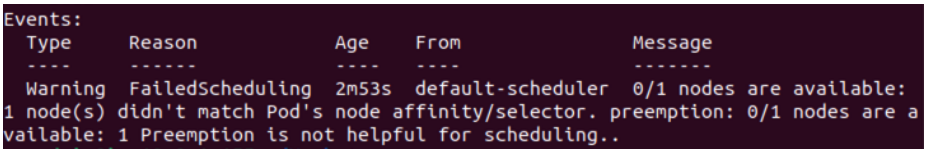
```
kubectl describe pods webapp-deployment-5956985478-76b94
```

4. Scroll down through the output until you see the 'Conditions:' section:

```
Conditions:
    Type              Status
    PodScheduled      False
```

You can see here that PodScheduled is False. This means that the scheduler could not identify a node to run the pod on.

5. Scroll down some more to find the 'Events:' section.



Here you see a warning with the message: '1 node(s) didn’t match node selector'. This must mean that a node selector was set in the yaml file but the conditions it requested were not met by any active nodes.

6. Open the yaml in the editor again:

```
code webapp-deployment.yaml
```

7. Search for 'nodeSelector'. You should find the following section:

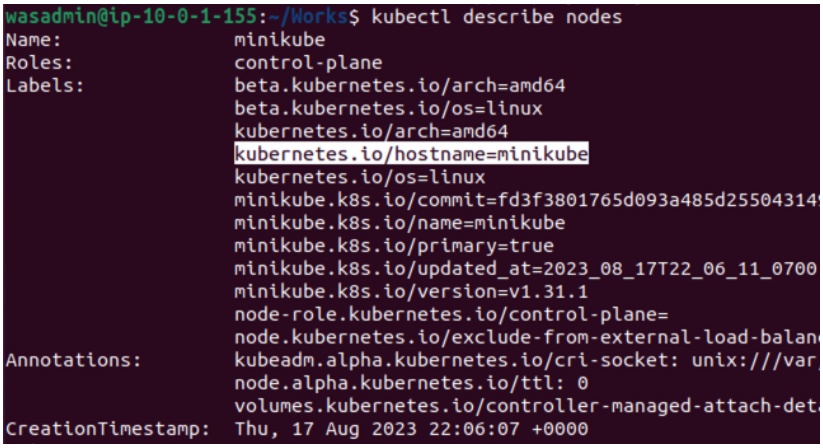
```
nodeSelector:
kubernetes.io/hostname: mycluster
```

The node selector holds a single name-value pair that is supposed to match a label in one of the running nodes. Our current cluster has a single node named 'minikube'. Let's check the minikube node to see if it has this label.

8. Leave the editor open and switch back to the terminal window.
9. Describe the node:

```
kubectl describe nodes
```

10. Scroll down in the command output to find the "Labels:" section. It should look like this:



The fourth label looks similar to the one we saw in the nodeSelector but there the value was 'mycluster' and here it is '**minikube**':

```
kubernetes.io/hostname=__minikube__
```

11. Go back to the YAML file in the editor and set the node selector to match the label from the node:

```
nodeSelector:
kubernetes.io/hostname: minikube
```

12. Save the yaml file and close it.
13. Apply the updated yaml deployment specification file:

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

Take note that we did not delete the existing deployment before applying the update. This can cause all kinds of problems, one of which we will see soon.

14. Check the pods:

```
kubectl get pods
```

Now there are two pods. The first one is 'pending' and is still trying to be deployed to a node that that matches the incorrect label in its specification. That pod was created from the previous flawed deployment. Trying to delete the old pod will be problematic because Kubernetes will keep trying to restart it even though it is broken.

```
wasadmin@ip-10-0-1-155:~/Works$ kubectl get pods
NAME                                READY   STATUS             RESTARTS   AGE
webapp-deployment-5944b5866d-slv7g  0/1     ImagePullBackOff   0          78s
webapp-deployment-5956985478-76b94  0/1     Pending            0          9m15s
```

The easiest way to fix this is to delete the initial deployment and recreate it.

15. Remove the deployment with this command:

```
kubectl delete -f webapp-deployment.yaml
```

16. Verify that the deployment and pods were removed using the following commands:

```
kubectl get deployments
kubectl get pods
```

You may need to wait until the pod is completely removed.

17. Apply the updated yaml again:

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

18. Check the deployment:

```
kubectl get deployments
```

Your output should look like this:

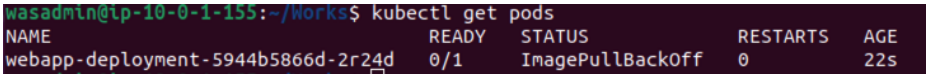
```
wasadmin@ip-10-0-1-155:~/Works$ kubectl get deployments
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
webapp-deployment   0/1     1            0           5s
```

READY indicates that zero of one pods were deployed.

19. Let's check the pods:

```
kubectl get pods
```

Although the status is no longer 'Pending' and the issue with the node selector has been fixed a new problem has come up based on the new status - ImagePullBackOff. In the next section we will take a closer look and troubleshoot this new issue.

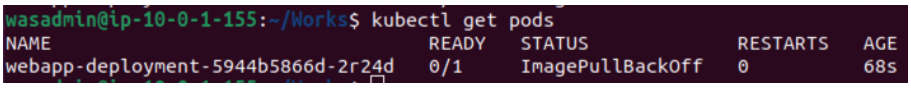


10.4. Troubleshooting a Failed Image Pull

In this section we will pick up where we left off in the last section and continue troubleshooting the ImagePullBackOff issue.

1. Start by listing the pods:

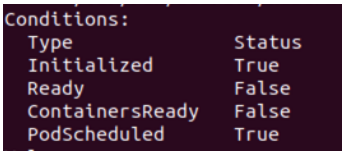
```
kubectl get pods
```



2. Describe the pod shown in the previous command (use the pod name specific to you as it will be different from the one listed here):

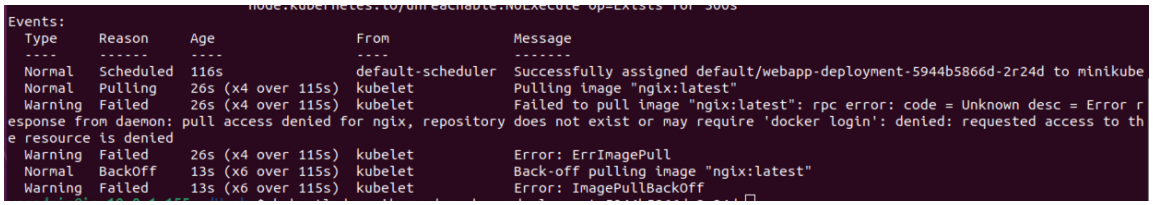
```
kubectl describe pods webapp-deployment-5944b5866d-2r24d
```

3. Scroll down in the output to the 'Conditions:' section:



   Here we see that the pod has been initialized and scheduled but is not yet ready.

4. For more information we will scroll down further and check out the 'Events:' section:



   The information we need is listed in the third item:

   Failed to pull image "ngix:latest": rpc error: code = Unknown desc = Error response from daemon: pull access denied for ngix, repository does not exist or may require 'docker login': denied: requested access to the resource is denied

   Here is says that the kubelet failed to pull the requested "ngix:latest " image. This looks like a typo where 'ngix' was entered instead of the correct value 'nginx'

5. First delete the deployment:

```
kubectl delete -f webapp-deployment.yaml
```

6. Open the deployment YAML file in your editor.

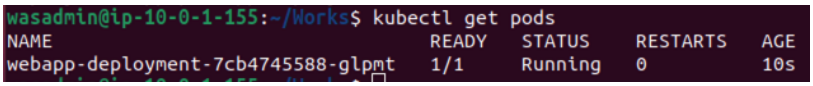
```
code webapp-deployment.yaml
```

7. Find the string '**ngix**' and replace it with '**nginx**'.
8. Save the file and close the editor.
9. Run the following command:

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

10. Try running "**kubectl get deployments**" a few times until the READY field shows up as "1/1". This may take a minute or two. This means that the pod was finally deployed successfully.
11. Get the pod name:

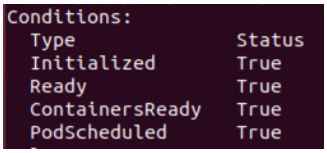
```
kubectl get pods
```



12. Describe the pod (use *your* pod's name):

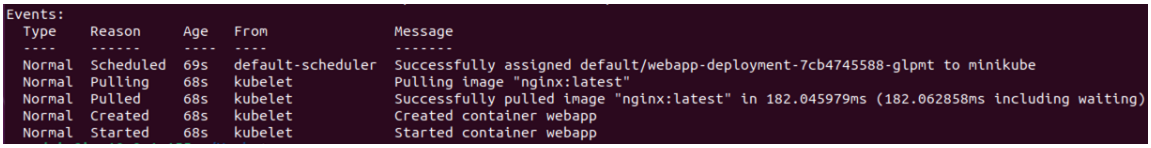
```
kubectl describe pods webapp-deployment-7cb4745588-glpmt
```

Scroll down until you see the 'Conditions:' section:



The output shows that the pod has been scheduled, initialized and is ready as is the container.

The 'Events:' section tells the same story.



This is what a successful pod deployment looks like.

10.5. Troubleshooting Resource Issues when Scaling

Now that we have the deployment running, we will try scaling it up from 1 to 20 replicas.

1. Execute the following commands to scale the deployment:

```
kubectl scale --current-replicas=1 --replicas=20 deployment
webapp-deployment

[Enter the command in 1 line]
```

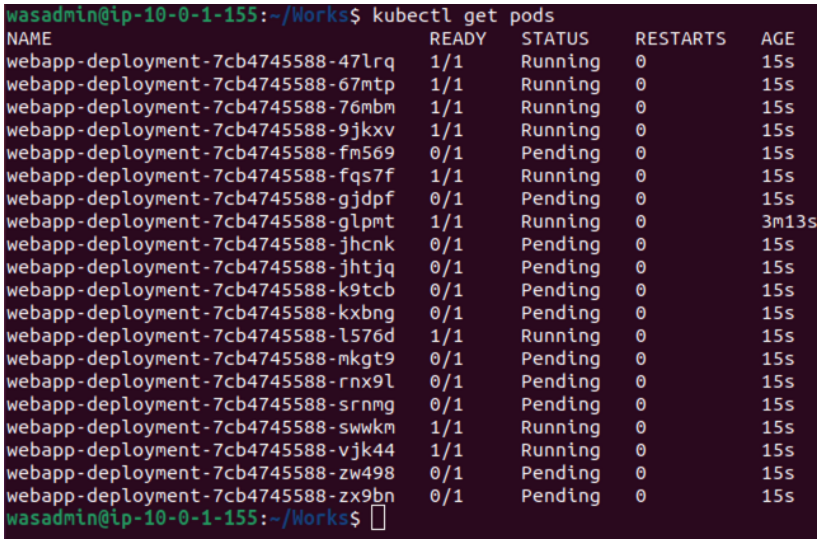
You should see the output:

```
deployment.apps/webapp-deployment scaled
```

2. Scaling should affect the number of pods so we will check that:

```
kubectl get pods
```

   You should see the output:



   Notice that many the pods are ready except many others are READY=0/1 and STATUS is 'pending'. We need to find out why those pods are not running.

3. Pick a pod that is not running and use **kubectl describe** on it (your pod name will be different):

```
kubectl describe pods webapp-deployment-7cb4745588-zx9bn
```

4. Under the 'Conditions:' section you will find that the pod has not been scheduled:

```
Conditions:
    Type            Status
```

```
    PodScheduled    False
```

5. Under the 'Events:' section you will find this warning:

   Warning FailedScheduling 112s default-scheduler 0/1 nodes are available: **1 Insufficient cpu**. preemption: 0/1 nodes are available: 1 No preemption victims found for incoming pod..

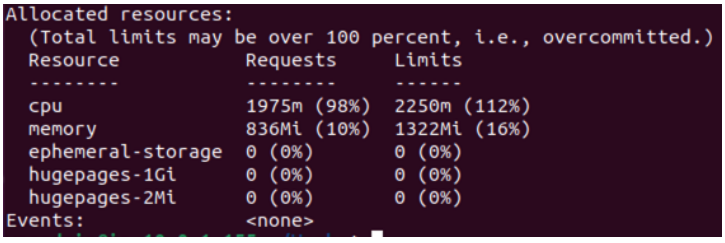
6. This is a resource issue. We can verify the problem by checking the node to see how much resources it has. Use the following command:

```
kubectl describe nodes
```

7. Find the 'Non-terminated Pods:' section in the output. You will see that each pod from our deployment is taking about 6% of the total CPU resource (125m):

```
Namespace   Name        CPU Requests    CPU Limits  Memory
Requests   Memory Limits     AGE
---------   ----------  ------------    ----------
---------------   -------------     ---
default     webapp-...  125m (6%)       250m (12%)  64Mi (3%)
128Mi (6%)        16m
```

8. Now check the "Allocated resources" section:



Here we see that 98% of the CPU is already allocated (your numbers might be slightly different) which means that there is not enough left for more pods. We have two choices here:

9. Increase the CPU resource allocated to the nodes VM.
10. Get by less replicas for the time being.
11. Reduce the amount of CPU requested by each pod.

    For now let's rescale the deployment down from 20 to 9 replicas.

12. Execute the following scale command in 1 line:

    ```
    kubectl scale --current-replicas=20 --replicas=9 deployment
    webapp-deployment
    ```

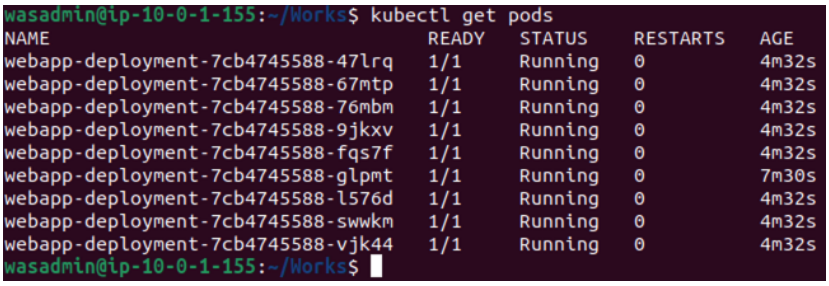
    The output should say:

    ```
    deployment.extensions/webapp-deployment scaled
    ```

13. Check the pods again:

    ```
    kubectl get pods
    ```

    You will see that there are now only 9 pods and all of them are running.



If you wanted instead to reduce the amount of CPU used by each container you would modify the following section in the webapp-deployment.yaml:

```
resources:
    requests:
        memory: "64Mi"
        cpu: "125m"
    limits:
        memory: "128Mi"
        cpu: "250m"
```

Reducing the request CPU from 125 to 100 should allow for more pods to be scheduled.

```
resources:
    requests:
        memory: "64Mi"
        cpu: "100m"
    limits:
        memory: "128Mi"
        cpu: "250m"
```

After this change you would need to delete the current deployment and recreate it with the updated yaml. At that point you would be able to scale the deployment up to 10 replicas

without running out of space.

14. Type **exit** until the Terminal is closed.

10.6. Review

In this lab we troubleshot issues that come up when deploying an application including:

- Yaml Syntax Exceptions
- nodeSelector Issues
- Failed Image Pull
- Resource Issues while Scaling

For each issue we saw how the issue appeared, looked into errors and warning, fixed the issue and re-tested to verify the fix.