WA2894 Booz Allen Hamilton Tech Excellence Modern Software Development Program - Phase 3

Student Labs

Web Age Solutions

Table of Contents

Environment

This course requires the following Virtual Machine (VM):

- `VM_WA2894_REL_1_1`
- `Project-VM-4.0-2023-02-07`

Make sure to use the right VM for each Lab, instructions are provided at the beginning of each Lab.

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.

Part 1 - Create a Docker account

In this Lab you will work in the 'VM_WA2894_REL_1_1'.

Start or connect to this VM if you don't have it opened yet.

Use wasadmin for user and password.

In this part, you will create a Docker account.

__1. Open a browser and navigate to the following URL:

```
https://hub.docker.com/
```

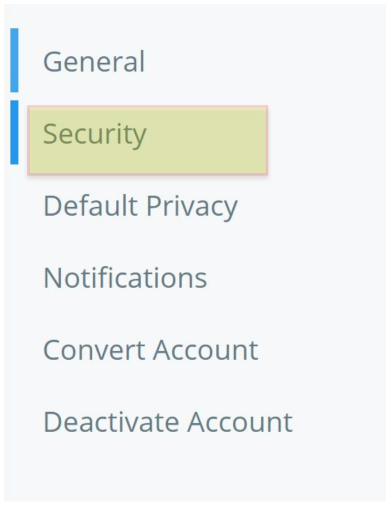
__2. Click **Sign Up** and create a new account.

__3. Sign in with your newly created account.

__4. 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.

__5. Select **Security**.



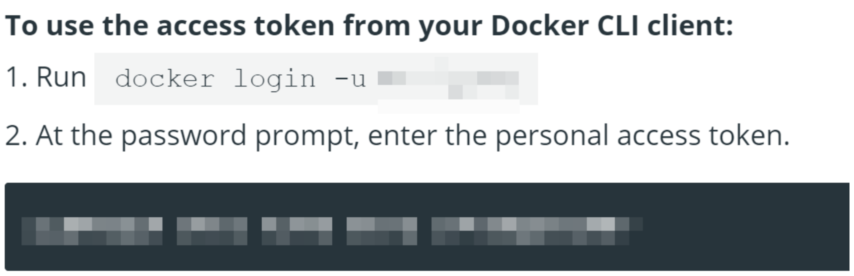
__6. Click **New Access Token**.

__7. Add the token description, such as *docker & Kubernetes labs.*

__8. Keep Access permissions as default (**Read, Write, Delete**).

__9. Click **Generate**

Notice it shows the following information on the page:



To use the access token from your Docker CLI client:

1. Run `docker login -u` [redacted]

2. At the password prompt, enter the personal access token.

[redacted]

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

Part 2 - Log in with your Docker account

__1. Start or connect to the '**VM_WA2894_REL_1_1**' virtual machine in case you don't have it opened yet. Use **wasadmin** for user and password.

__2. Open a new Terminal window.

__3. Execute the following command to switch to the root user:

```
sudo -i
```

__4. Verify you are logged in as root:

```
whoami
```

__5. Run the following command to log into Docker:

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

__6. You should see a "Login Succeeded" response.

__7. Close the Terminal.

Lab 2 - Getting Started with Docker

Docker is an open IT automation platform widely used by DevOps, and in this lab, we will review the main Docker commands. In this lab, you will install Docker and use its basic commands. You will also create a custom image by creating a Dockerfile.

Part 1 - Setting the Stage

In this Lab you will continue working in the 'VM_WA2894_REL_1_1'.

Start or connect to this VM if you don't have it opened yet. Use wasadmin for user and password.

Make sure you ran the following command in a previous lab:

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

__1. Open a new Terminal window.

__2. Change to sudo:

```
sudo -i
```

Enter wasadmin as password.

__3. Enter the following command:

```
cd /home/wasadmin/Works
```

__4. Get directory listing:

```
ls
```

Notice there is **SimpleGreeting-1.0-SNAPSHOT.jar** file. You will use this file later in this lab. It will be deployed in a custom Docker image and then you will create a container based on that image.

Part 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. Navigate through the output using your arrow keys and review Docker's commands.

__3. Enter q to exit.

Some commands are listed 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
...
```

__4. 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/***

Part 3 - Run the "Hello World!" Command on Docker

Let's check out what OS images are currently installed on your Lab Server.

__1. Enter the following command:

```
docker images
```

Notice there are a few images in the VM. You will use them in various labs.

One of the images is **ubuntu:12.04**.

__2. Enter the following command:

```
docker run ubuntu echo 'Yo Docker!'
```

The command will download an Ubuntu image and then display **Yo Docker!** message

So, what happened?

docker run executes the command that follows after it on a container provisioned on-the-fly for this occasion.

When the Docker command completes, you get back the command prompt, the container gets stopped and Docker creates an entry for that session in the transaction history table for your reference.

Part 4 - List and Delete Container

In this lab part, we will need a second terminal.

__1. Open a new terminal, expand it width wise (horizontally) to capture the output of the *docker ps* command we are going to run into it. Also make sure it does not completely overlap the original terminal window.

We will be referring to this new terminal as **T2;** the original terminal will be referred to as **T1**.

__2. Change to sudo:

```
sudo -i
```

Enter wasadmin as password.

__3. Change to the **~/Works** directory:

```
cd /home/wasadmin/Works
```

__4. Enter the following command:

```
docker ps
```

You should see an empty container table listing only the captions part of the table:

```
CONTAINER ID    IMAGE   COMMAND   CREATED    STATUS     PORTS     NAMES
```

__5. To see a list of all containers, whether active or inactive, run the following command:

```
docker ps -a
```

__6. Copy the **ubuntu Container ID** that you just created.

__7. Switch to the original terminal window (T1).

__8. Enter the following command and replace the container id with the **ubuntu** container from T2:

```
docker rm <container id>
```

The **docker ps** command only shows the running containers; if you repeat the **docker ps** command now after you have killed the container process, it will show the empty table.

In order to view all the containers created by docker with stopped ones stashed in the history table, use the **-a** flag with this command.

__9. Switch to the **T2** terminal.

__10. Enter the following command:

```
docker ps -a
```

Verify the ubuntu container was destroyed.

Part 5 - Working with Containers

__1. Switch to the **T1** terminal.

__2. Enter the following command:

```
docker run -it --hostname basic_host ubuntu /bin/bash
```

This command will create a container from the *ubuntu* OS image, it will give the instance of the container the hostname of **basic_host**, launch the *bash* program on it and will make the container instance available for interactive mode (**i**) over allocated pseudo TTY (**t**).

After running the above command, you should be dropped at the prompt of the **basic_host** container.

```
root@basic_host:/#
```

As you can see, you are **root** there (symbolized by the '#' prompt sign).

__3. Enter the following command to stop the container and exit out to the host OS:

```
exit
```

You should be placed back in the root's *Works* folder.

__4. Enter the following command to see the containers:

```
docker ps -a
```

You should see the ubuntu status as Exited.

__5. Enter the following command to restart the container (pick the new ubuntu from the list):

```
docker start <container id>
```

__6. Connect to the container:

```
docker exec -it <container id> /bin/bash
```

__7. Enter the following command:

```
exit
```

You should be logged off and placed back in the root's *Works* folder.

__8. Enter the following command:

```
docker stop <container id>
```

__9. Enter the following command:

```
docker ps -a
```

You should see that the container is listed in the transaction history and exited as status.

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
aed47e954acd        ubuntu              "/bin/bash"         3 minutes ago      Exited (0)
```

Part 6 - Create a Custom Image

Now that we have ourselves a container (which is currently in the **exited / stopped** status), let's create a custom image based on it.

__1. Enter the following command providing your container id (*aed47e954acd*, in our case):

```
docker commit <container id>  my_server:v1.0
```

You should get back the OS image id generated by Docker.

__2. Enter the following command:

```
docker images
```

You should see the new image [**my_server:v1.0**] listed on top of the available images in our local image repository.

__3. Enter the following command:

```
docker run -it my_server:v1.0 /bin/bash
```

This command will start a new container from the custom image we created in our local image repository.

__4. Enter the following command at the container prompt:

```
exit
```

You will be dropped back at the Lab Server's prompt.

Part 7 - Workspace Clean-Up

__1. Enter the following command:

```
docker ps -a
```

This command will show all the containers and not only the running ones.

It is always a good idea to clean-up after yourself, so we will remove all the containers we created so far.

__2. Enter the following command for every listed container id [ubuntu and my_server:v1.0]:

NOTE: DO NOT delete any image or container other than the ones you have created in this lab

```
docker rm <container id>
```

__3. Verify there are no docker containers created in this lab:

```
docker ps -a
```

__4. List all your images:

```
docker images
```

__5. Remove the *my_server:v1.0* image we created and persisted in our local image repository:

```
docker rmi my_server:v1.0
```

__6. Verify your image [my_server:v1.0] has gone:

```
docker images
```

Part 8 - Create a Dockerfile for Building a Custom Image

In this part you will download Ubuntu docker image and create a Dockerfile which will build a custom Ubuntu based image. It will automatically update the APT repository, install JDK, and copy the SimpleGreeting.jar file from host machine to the docker image. You will also build a custom image by using the Dockerfile.

__1. In the terminal, type following command to create Dockerfile:

```
gedit Dockerfile
```

__2. Type in following code:

```
# lets use OpenJDK docker image
FROM openjdk:8

# deploy the jar file to the container
COPY SimpleGreeting-1.0-SNAPSHOT.jar /root/SimpleGreeting-1.0-
SNAPSHOT.jar
```

Note: The Dockerfile creates a new image based on OpenJDK and copies host's /home/wasadmin/Works/SimpleGreeting-1.0-SNAPSHOT.jar to the image under the root directory.

__3. Click **Save** button.

__4. Close gedit. You may see some errors on the Terminal, ignore them.

__5. Type **ls** and verify your new file is there.

__6. Run following command to build a custom image:

```
docker build -t dev-openjdk: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.

Part 9 - 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 exists, and execute the jar file.

__1. In the terminal, run following command to verify the custom image exists:

```
docker images
```

Notice **dev-openjdk:v1.0** is there.

__2. Create a container based on the above image and connect to it:

```
docker run --name dev --hostname dev -it dev-openjdk:v1.0 /bin/bash
```

Note: You are naming the container dev, hostname is also dev, and it's based on your custom image dev-openjdk:v1.0

__3. Switch to the root directory in the container:

```
cd /root
```

__4. Get the directory list:

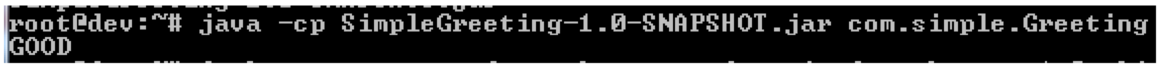
```
ls
```

Notice **SimpleGreeting*.jar** file is there.

__5. Execute the jar file:

```
java -cp SimpleGreeting-1.0-SNAPSHOT.jar com.simple.Greeting
```

Notice it displays the following message:



```
root@dev:~# java -cp SimpleGreeting-1.0-SNAPSHOT.jar com.simple.Greeting
GOOD
```

__6. Exit out from the container to the command prompt:

```
exit
```

__7. Get active docker container list:

```
docker ps
```

Notice there's no active container.

__8. Get list of all docker containers:

```
docker ps -a
```

Notice there's one inactive container named **dev**.

Part 10 - Working with a Docker Volume

In this part, you will manage a Docker volume. You will create a volume, mount it in a container, store content in it, and delete the volume.

__1. Create a volume:

```
docker volume create hello
```

__2. Create a container and mount it in the container:

```
docker run -v hello:/world -it ubuntu /bin/bash
```

__3. View mount point in container: (it will show the /world directory)

```
ls / -al
```

__4. Create a file in the volume:

```
echo "hello world" > /world/test.txt
```

__5. Exit out of the container:

```
exit
```

__6. Destroy the ubuntu container: (Note: Make a note of the ID of your container and substitute it below)

```
docker ps -a
docker stop <id>
docker rm <id>
```

__7. Create another container and attach the volume as a mount point:

```
docker run -v hello:/world -it ubuntu /bin/bash
```

__8. View mount point in container: (it will show the /world directory)

```
ls /world
```

__9. View contents of the file:

```
cat /world/test.txt
```

__10. Exit out of the container:

```
exit
```

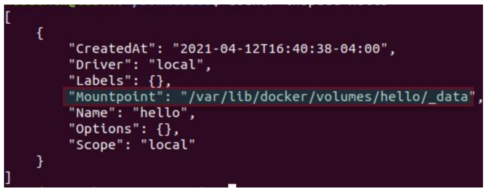
__11. View all volumes:

```
docker volume ls
```

__12. Check the mount point location.

```
docker inspect hello
```

__13. Notice the mount point is displayed in the JSON output.



```
[
    {
        "CreatedAt": "2021-04-12T16:40:38-04:00",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/hello/_data",
        "Name": "hello",
        "Options": {},
        "Scope": "local"
    }
]
```

Part 11 - Cleanup Docker

In this part you will clean up docker by removing containers and images.

__1. In the terminal, run following to get list of all containers:

```
docker ps -a
```

__2. Stop the **ubuntu** and **dev-openjdk:v1.0** containers and remove them by running following commands. Don't remove the ones you haven't created yourself in this lab:

```
docker stop <container>
docker rm <container>
```

__3. Get list of all docker containers, the ubuntu and dev-openjdk:v1.0 containers are gone:

```
docker ps -a
```

__4. View all volumes:

```
docker volume ls
```

__5. Delete the volume:

```
docker volume rm hello
```

__6. Get docker image list:

```
docker images
```

__7. Remove all images that you created by executing command
docker rmi <REPOSITORY:TAG> **:**

```
docker rmi dev-openjdk:v1.0
```

__8. Make sure your image has been deleted:

```
docker images
```

__9. In each Terminal type **exit** many times until all Terminals are closed.

Part 12 - Review

In this lab, we reviewed the main Docker command-line operations.

Lab 3 - 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.

In this Lab you will continue working in the 'VM_WA2894_REL_1_1'.

Make sure you ran the following command in a previous lab:

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

Part 1 - Configure the Docker account in Minikube/Kubernetes

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:

```
www.docker.com
```

__2. Open a terminal window.

__3. Switch to the root user by executing the following command.

```
sudo -i
```

Enter *wasadmin* as the password, if prompted.

__4. 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.

__5. Copy the Docker configuration file to the kubelet directory.

```
cp ~/.docker/config.json /var/lib/kubelet/config.json
```

__6. Restart the Kubelet service.

```
systemctl restart kubelet
```

__7. If you see the message **“Warning: The unit file, source configuration file or drop-ins of kubelet.service changed on disk. Run 'systemctl daemon-reload' to reload units”,** execute the following command:

```
systemctl daemon-reload
```

__8. Restart the kubelet service one more time.

```
systemctl restart kubelet
```

__9. In the Terminal type **exit** many times until Terminal is closed.

__10. Close the browser.

Lab 4 - Getting Started with 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 minikube, which allows you to create a Kubernetes environment with ease.

Part 1 - Setting the Stage

In this Lab you will continue working in the 'VM_WA2894_REL_1_1'.

Make sure you ran the following command in a previous lab:

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

__1. Open a browser and connect to the docker page to make sure you have internet connection:

```
www.docker.com
```

__2. Open a new Terminal window.

__3. Switch the logged-in user to **root**:

```
sudo -i
```

When prompted for the logged-in user's password, enter **wasadmin**

__4. Enter the following command:

```
whoami
```

You should see that you are **root** now.

```
root
```

__5. Verify you have minikube installed:

```
minikube version
```

__6. Get minikube help:

```
minikube --help
```

Part 2 - Start the Cluster

In this part, you will start the Kubernetes cluster and interact with it in various ways.

__1. First remove the stop minikube is it's running [you may see a timeout error after some minutes]:

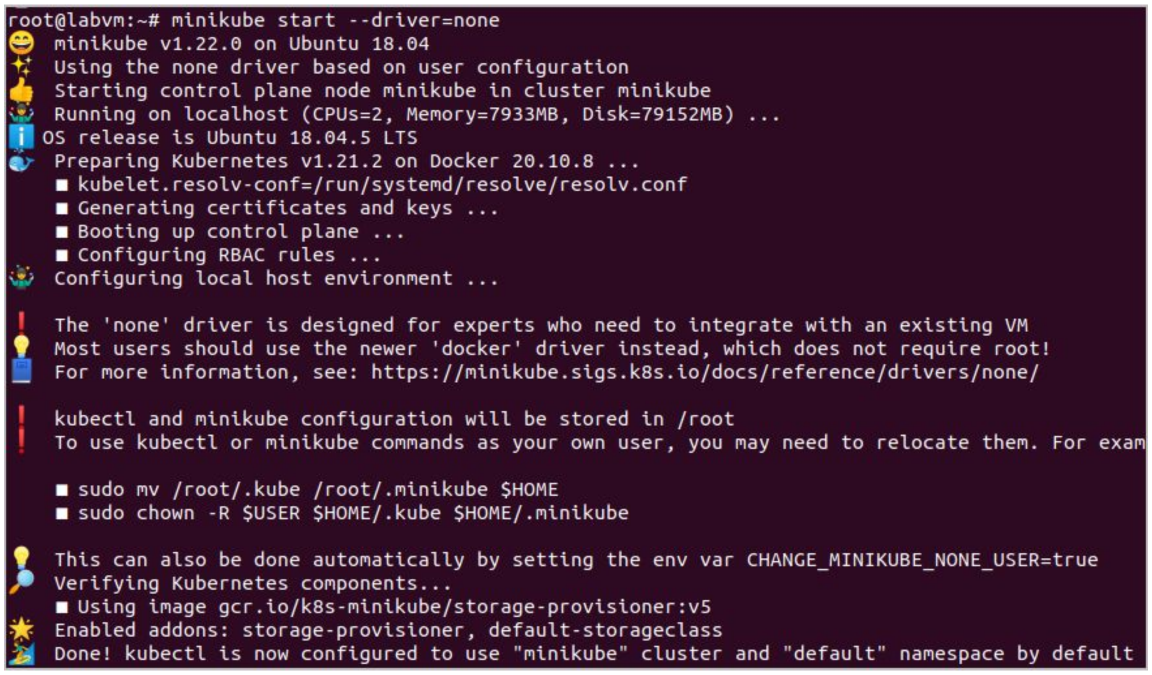
```
minikube stop
```

__2. Then delete minikube:

```
minikube delete
```

__3. Now, start minikube:

```
minikube start --driver=none
```



```
root@labvm:~# minikube start --driver=none
😄  minikube v1.22.0 on Ubuntu 18.04
✨  Using the none driver based on user configuration
👍  Starting control plane node minikube in cluster minikube
🤹  Running on localhost (CPUs=2, Memory=7933MB, Disk=79152MB) ...
ℹ️ OS release is Ubuntu 18.04.5 LTS
🐳  Preparing Kubernetes v1.21.2 on Docker 20.10.8 ...
    ▪ kubelet.resolv-conf=/run/systemd/resolve/resolv.conf
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🤹  Configuring local host environment ...

❗  The 'none' driver is designed for experts who need to integrate with an existing VM
💡  Most users should use the newer 'docker' driver instead, which does not require root!
📘  For more information, see: https://minikube.sigs.k8s.io/docs/reference/drivers/none/

❗  kubectl and minikube configuration will be stored in /root
❗  To use kubectl or minikube commands as your own user, you may need to relocate them. For exam

    ▪ sudo mv /root/.kube /root/.minikube $HOME
    ▪ sudo chown -R $USER $HOME/.kube $HOME/.minikube

💡  This can also be done automatically by setting the env var CHANGE_MINIKUBE_NONE_USER=true
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

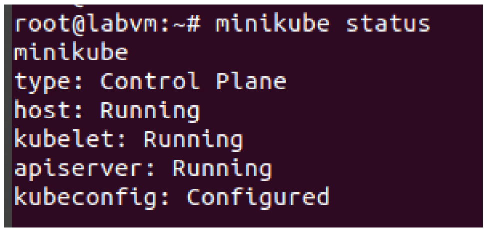
Note: The above command performs the following operations:

1. Generates the certificates and then proceeds to provision a local Docker host.
2. Starts up a control plane that provides various Kubernetes services.
3. Configures the default RBAC rules.

__4. In the terminal window, run the following command to verify the minikube status:

```
minikube status
```

Notice it shows messages like this:



```
root@labvm:~# minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

__5. Verify you can execute kubectl and also obtain Kubernetes version:

```
kubectl version
```

Notice it lists the major and minor versions in JSON format.

__6. Run the following command to obtain the cluster IP address:

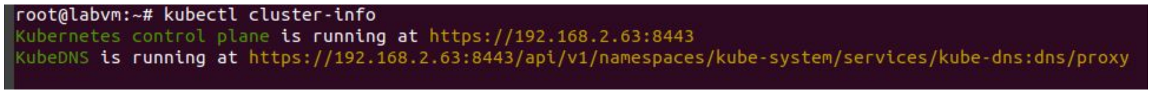
```
minikube ip
```

Notice it shows the IP address of our cluster.

__7. Get Kubernetes cluster information:

```
kubectl cluster-info
```

Notice it displays the IP address and port where the Kubernetes master is running. Don't worry about any connection message.



```
root@labvm:~# kubectl cluster-info
Kubernetes control plane is running at https://192.168.2.63:8443
KubeDNS is running at https://192.168.2.63:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

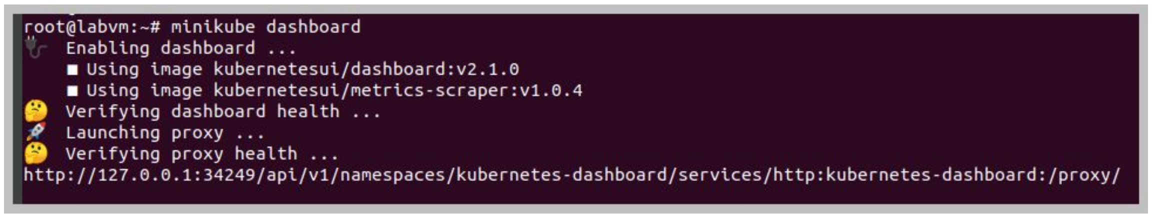
Part 3 - View Kubernetes Dashboard

In this part you will view the Kubernetes dashboard and interact with it.

__1. In the terminal, run the following command to view the dashboard URL:

```
minikube dashboard
```

The command will show [It may take a while to show the IP]:



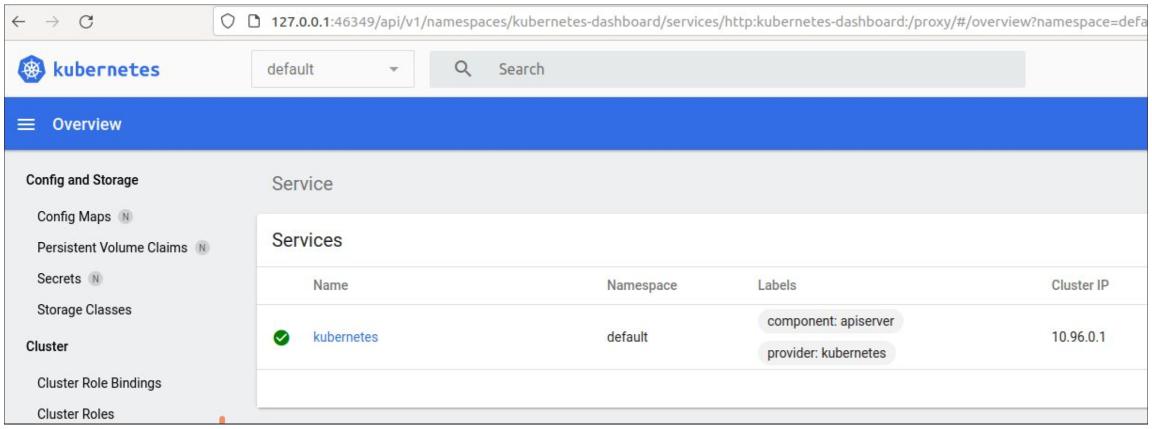
```
root@labvm:~# minikube dashboard
🔌  Enabling dashboard ...
    ■ Using image kubernetesui/dashboard:v2.1.0
    ■ Using image kubernetesui/metrics-scraper:v1.0.4
🤔  Verifying dashboard health ...
🚀  Launching proxy ...
🤔  Verifying proxy health ...
http://127.0.0.1:34249/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/
```

__2. Copy the URL which will show up as part of the minikube dashboard. The URL looks like this:

```
http://127.0.0.1:34249/api/v1/namespaces/kubernetes-dashboard/
services/http:kubernetes-dashboard:/proxy/
```

__3. Open the Firefox web browser and paste the URL.

__4. It will show the Kubernetes page. It may look different than below:



Services

Name	Namespace	Labels	Cluster IP
✓ kubernetes	default	component: apiserver provider: kubernetes	10.96.0.1

Notice that the Terminal is running, if you close the Terminal or terminate the command the browser will close, so keep it open.

If you get an error that Firefox is not supported then click on the link to open the URL.

__5. On the left side of the dashboard, click **Nodes** under **Cluster** section.

__6. Open a new Terminal window.

__7. Connect to sudo:

```
sudo -i
```

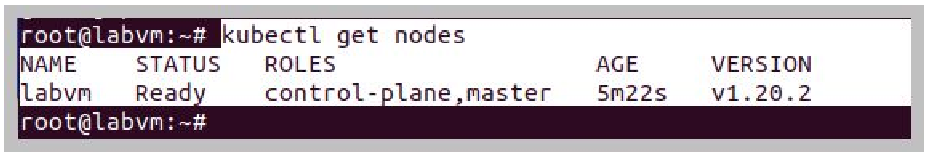
[Enter **wasadmin** as password]

__8. In the terminal window, run the following command to view the nodes:

```
kubectl get nodes
```

Notice that you can see the node in the terminal. It's the same information you saw in the dashboard earlier in this part of the lab.

Make a note of the node name because you will use it later in the lab. In this case **labvm**



```
root@labvm:~# kubectl get nodes
NAME    STATUS   ROLES                  AGE     VERSION
labvm   Ready    control-plane,master   5m22s   v1.20.2
root@labvm:~#
```

Part 4 - Create a Container

In this part you will create a container and run it in the node/cluster i.e. you will run a workload in the cluster.

__1. In the terminal window, run the following command:

```
kubectl create deployment my-web-server --image=nginx --port=80
```

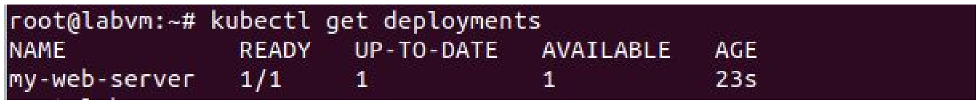
The command uses the nginx image to create a deployment named my-web-server and exposes the service on port 80. If the image is not available on your local machine, it connects to the Docker hub registry and downloads the image. You can specify other registries by specifying the full URL to the image.

In case if it container's image doesn't get downloaded properly, delete the cached data located under ~/.minikube/cache and retry.

__2. Run the following command to view the deployment list:

```
kubectl get deployments
```

You should see: [You may need to wait and repeat the command until you see 1/1]

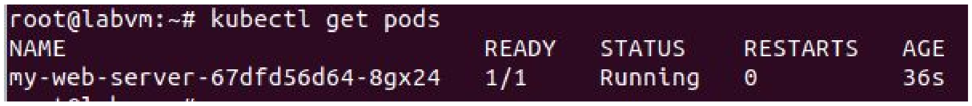


```
root@labvm:~# kubectl get deployments
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
my-web-server   1/1     1            1           23s
```

__3. Run the following command to view the pod list:

```
kubectl get pods
```

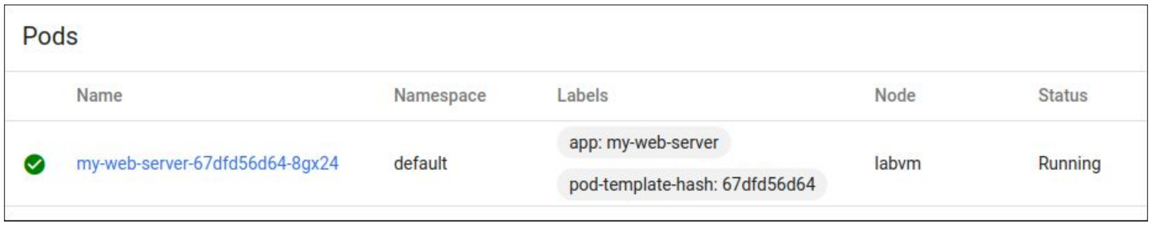
Notice it shows *my-web-server pod-{UUID}}*. [You may need to wait and repeat the command until you see 1/1]



```
root@labvm:~# kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
my-web-server-67dfd56d64-8gx24  1/1     Running   0          36s
```

__4. Switch to the Firefox window, where the dashboard is open, and click **Pods** on the left side of the page.

When the pod is created completely, it would show up like this:



Pods

	Name	Namespace	Labels	Node	Status
✓	my-web-server-67dfd56d64-8gx24	default	app: my-web-server pod-template-hash: 67dfd56d64	labvm	Running

Part 5 - View Logs and Details

In this part, you will view logs and details in various ways.

__1. On the dashboard, click **my-web-server-{{UUID}}** pod.

Notice it shows various pod details, such as creation date, status, and other events.

__2. Make a note of the pod name. (Note: Your UUID will most likely be different)

my-web-server-67dfd56d64-8gx24

__3. In the terminal run the following command to view pod details:

```
kubectl describe pod my-web-server-{{UUID}}
```

Notice you get to view similar details in the terminal as you saw in the previous steps of this part.

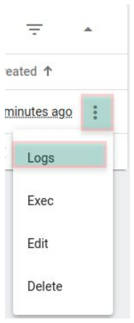
__4. Run following command to view the node details [replace <node name > with yours]:

```
kubectl describe node <node name>
```

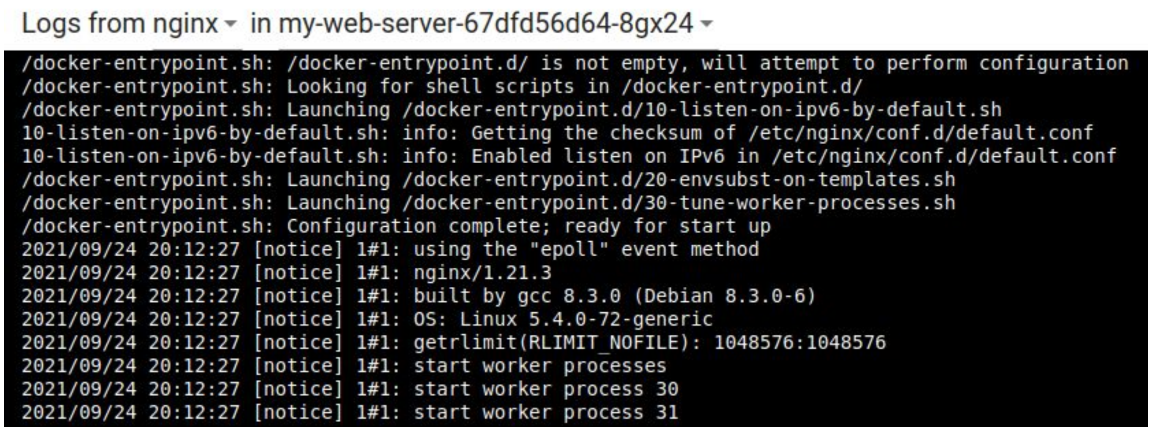
Notice it shows the node details. [You should be using **labvm** as node name in this VM]

__5. On the dashboard, click **Pods** under the **Workloads** section.

__6. Click the 3 dots (...) next to the **my-web-server**-*{{UUID}}* pod and click **Logs**.



Notice it shows the pod log:



Logs from nginx in my-web-server-67dfd56d64-8gx24

```
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2021/09/24 20:12:27 [notice] 1#1: using the "epoll" event method
2021/09/24 20:12:27 [notice] 1#1: nginx/1.21.3
2021/09/24 20:12:27 [notice] 1#1: built by gcc 8.3.0 (Debian 8.3.0-6)
2021/09/24 20:12:27 [notice] 1#1: OS: Linux 5.4.0-72-generic
2021/09/24 20:12:27 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2021/09/24 20:12:27 [notice] 1#1: start worker processes
2021/09/24 20:12:27 [notice] 1#1: start worker process 30
2021/09/24 20:12:27 [notice] 1#1: start worker process 31
```

You will view the log again, after accessing the Nginx service.

Part 6 - Expose a Service

In this part, you will expose the nginx service, which you deployed in the previous parts of this lab.

__1. In the terminal, run the following command:

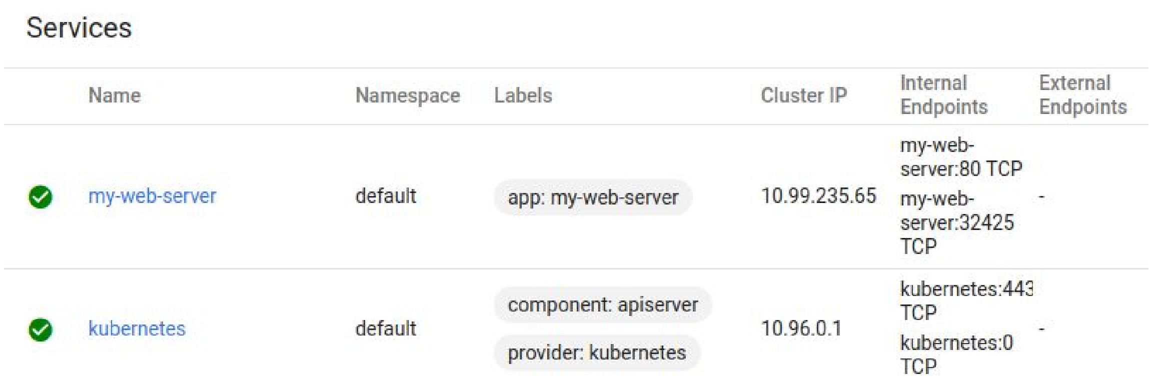
```
kubectl expose deployment my-web-server --type=NodePort
```

Notice it shows a message that the service has been exposed.

Make sure there is a **double dash** before *type=NodePort*

__2. In the Firefox browser, click **Services** on under the **Service** section.

Notice the page looks like this:



Services

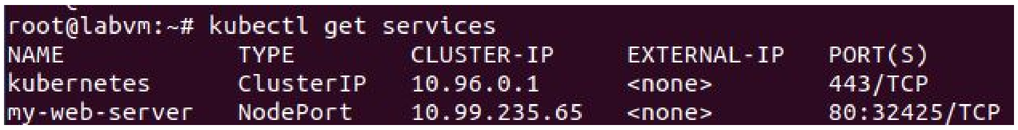
	Name	Namespace	Labels	Cluster IP	Internal Endpoints	External Endpoints
✓	my-web-server	default	app: my-web-server	10.99.235.65	my-web-server:80 TCP my-web-server:32425 TCP	-
✓	kubernetes	default	component: apiserver provider: kubernetes	10.96.0.1	kubernetes:443 TCP kubernetes:0 TCP	-

Note: The IP address is the internal IP address. You will access the public IP address later in the lab.

__3. In the terminal run the following command to view the exposed services:

```
kubectl get services
```

You should see something like this:

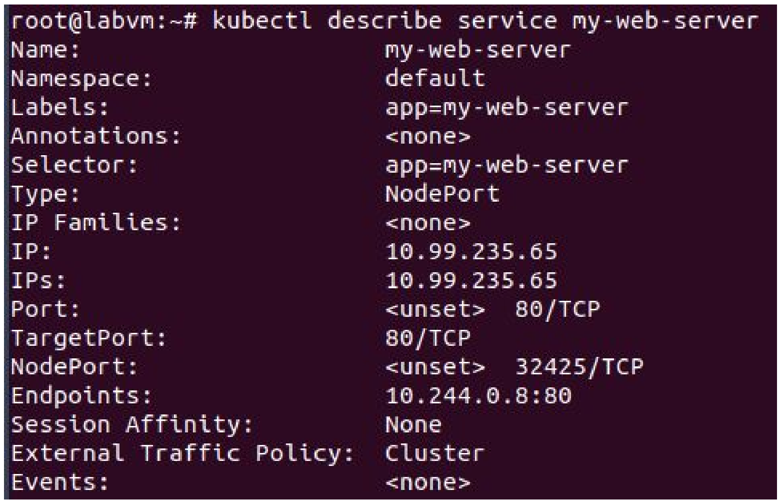


```
root@labvm:~# kubectl get services
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP
my-web-server   NodePort    10.99.235.65    <none>        80:32425/TCP
```

__4. View my-web-server service details:

`kubectl describe service my-web-server`

You should see something like this:



```
root@labvm:~# kubectl describe service my-web-server
Name:                     my-web-server
Namespace:                default
Labels:                   app=my-web-server
Annotations:              <none>
Selector:                 app=my-web-server
Type:                     NodePort
IP Families:              <none>
IP:                       10.99.235.65
IPs:                      10.99.235.65
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  32425/TCP
Endpoints:                10.244.0.8:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

__5. In the terminal, run the following command to access the service's public IP address:

`minikube service my-web-server --url=true`

Notice it shows an IP address like this: *http://192.168.2.63:32425*

Note. that the URL and port may be different.

Troubleshooting. If you don't get a URL then delete the service by running:

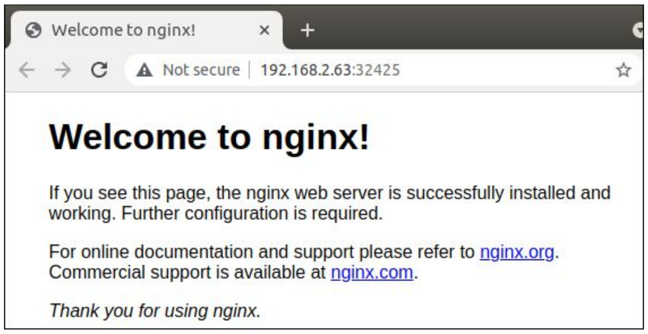
kubectl delete service my-web-server

and then create the service again.

__6. Ctrl+Click the URL in the terminal. It will launch the page in the Firefox web browser.

Alternatively, you can type in the URL manually in Firefox.

Notice it shows the familiar-looking Nginx home page.



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

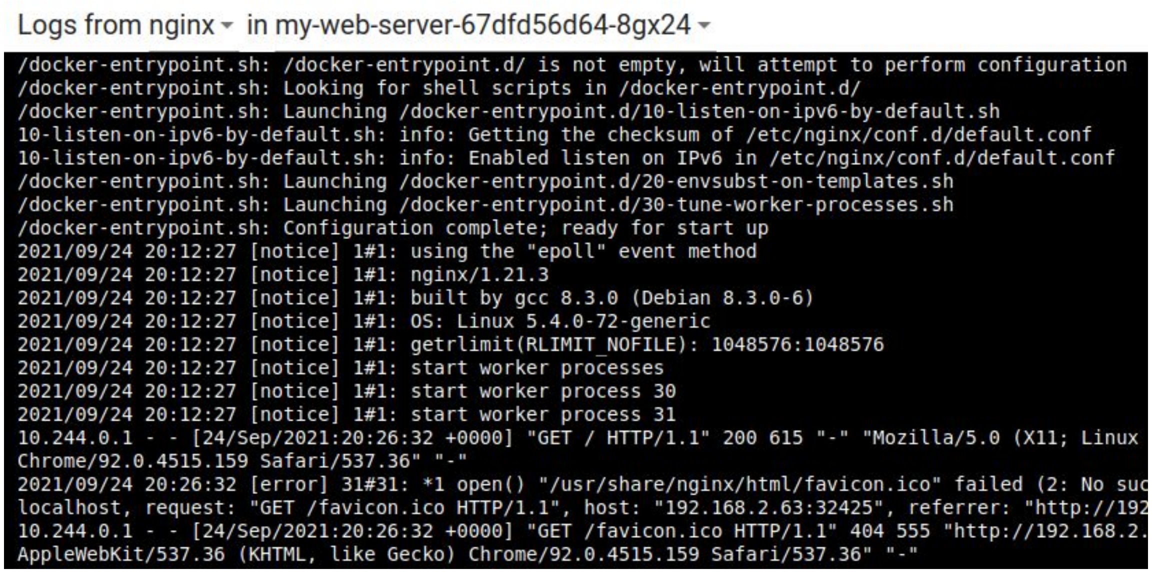
Thank you for using nginx.

__7. Back on the dashboard, click **Services**.

__8. Click **my-web-server**

__9. Scroll down to the **Pods** section and click the 3 dots (…) next to the pod name (**my-web-server-***{{UUID}}*) and click **Logs**.

Notice it shows the service access log like this:



Logs from nginx ▾ in my-web-server-67dfd56d64-8gx24 ▾

```
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2021/09/24 20:12:27 [notice] 1#1: using the "epoll" event method
2021/09/24 20:12:27 [notice] 1#1: nginx/1.21.3
2021/09/24 20:12:27 [notice] 1#1: built by gcc 8.3.0 (Debian 8.3.0-6)
2021/09/24 20:12:27 [notice] 1#1: OS: Linux 5.4.0-72-generic
2021/09/24 20:12:27 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2021/09/24 20:12:27 [notice] 1#1: start worker processes
2021/09/24 20:12:27 [notice] 1#1: start worker process 30
2021/09/24 20:12:27 [notice] 1#1: start worker process 31
10.244.0.1 - - [24/Sep/2021:20:26:32 +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0 (X11; Linux
Chrome/92.0.4515.159 Safari/537.36" "-"
2021/09/24 20:26:32 [error] 31#31: *1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No suc
localhost, request: "GET /favicon.ico HTTP/1.1", host: "192.168.2.63:32425", referrer: "http://192
10.244.0.1 - - [24/Sep/2021:20:26:32 +0000] "GET /favicon.ico HTTP/1.1" 404 555 "http://192.168.2.
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36" "-"
```

__10. Make a note of the pod name (It should be my-web-server-{{UUID}})

__11. In the terminal run the following command:

```
kubectl logs my-web-server-{{UUID}}
```

Notice it shows the service access log in the terminal.

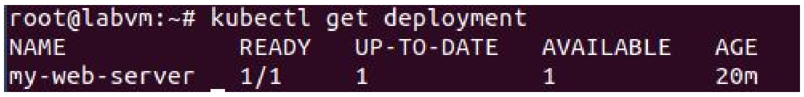
Part 7 - Scaling the Services

In the previous parts of the lab, you exposed a service. There was a single instance of the service, with one Pod that was provisioned on a single node. In this part, you will scale the service by having 3 Pods.

__1. In the terminal, run following command:

```
kubectl get deployment
```

Notice it shows result like this:



```
root@labvm:~# kubectl get deployment
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
my-web-server   1/1     1            1           20m
```

Notice there's a single instance running right now.

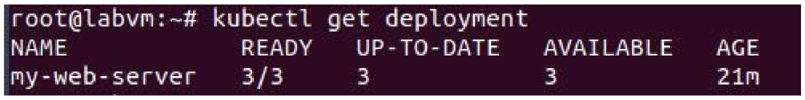
__2. Run the following command to scale up the service:

```
kubectl scale --replicas=3 deployment/my-web-server
```

__3. Run the following command to get the service instance count:

```
kubectl get deployment
```

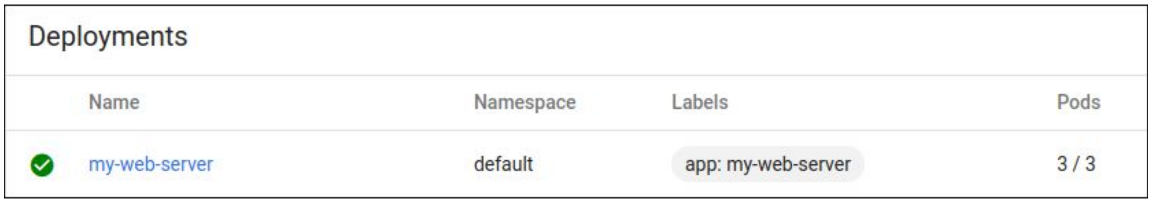
Notice how it shows result like this [make sure you see 3/3]:



```
root@labvm:~# kubectl get deployment
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
my-web-server   3/3     3            3           21m
```

__4. On the dashboard page, in the web browser, click **Deployments** on the left side of the page.

Notice it shows 3/3 Pods.

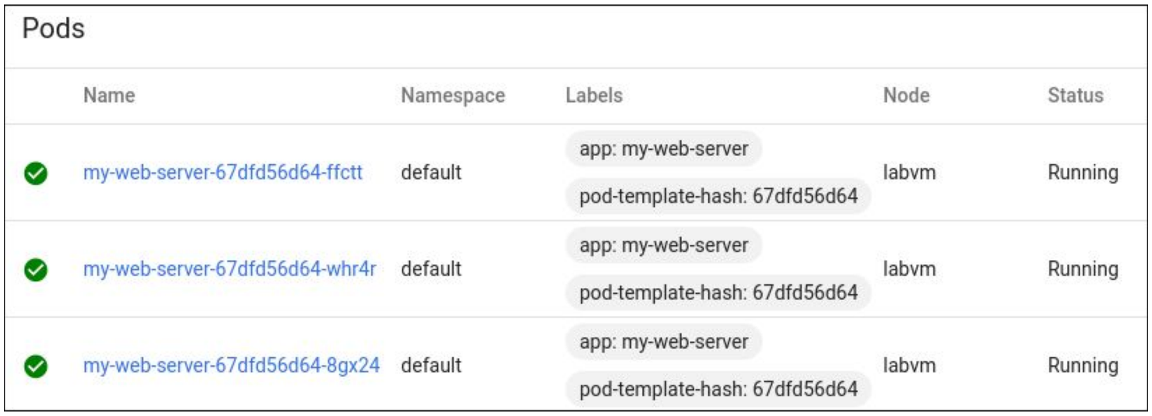


Deployments

	Name	Namespace	Labels	Pods
✓	my-web-server	default	app: my-web-server	3 / 3

__5. On the dashboard page, click **Services** on the left side of the page, then click **my-web-server** service.

__6. Scroll down to the Pods section and notice it shows Pods like these:



Pods

	Name	Namespace	Labels	Node	Status
✓	my-web-server-67dfd56d64-ffctt	default	app: my-web-server pod-template-hash: 67dfd56d64	labvm	Running
✓	my-web-server-67dfd56d64-whr4r	default	app: my-web-server pod-template-hash: 67dfd56d64	labvm	Running
✓	my-web-server-67dfd56d64-8gx24	default	app: my-web-server pod-template-hash: 67dfd56d64	labvm	Running

__7. On the left side of the page, click **Pods**.

Notice it shows the same Pod list.

__8. Run the following command to scale down the service:

```
kubectl scale --replicas=1 deployment/my-web-server
```

__9. Run the following command to verify the deployment is scaled down to 1 instance:

```
kubectl get deployment
```

Part 8 - Clean-Up

In this part, you will delete the deployment and stop the cluster you created in this lab.

__1. Close all open web browsers.

__2. Run the following command to delete the deployment:

```
kubectl delete deployment my-web-server
```

__3. Ensure the deployment is deleted by running the following command:

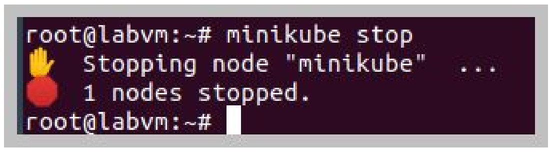
```
kubectl get deployments
```

__4. Switch to the Terminal running kubernetes and press CTRL+C to stop the running process.

__5. Stop the cluster:

```
minikube stop
```

Wait until the command is completed.



```
root@labvm:~# minikube stop
✋  Stopping node "minikube"  ...
🛑  1 nodes stopped.
root@labvm:~#
```

__6. Type many times **exit** in each Terminal to close them.

Part 9 - Review

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

Lab 5 - Load Balancing using Kubernetes

In this lab, you will deploy a sample service to Kubernetes cluster and configure load balancing. Each time you send a request to your service, it will hit a different IP and a different pod will fulfill the request.

The overall architecture looks like this:

Part 1 - Setting the Stage

In this Lab you will continue working in the 'VM_WA2894_REL_1_1'.

Start or connect to this VM if you don't have it opened yet. Use wasadmin for user and password.

Make sure you ran the following command in a previous lab:

sudo docker login -u {your-docker-id} -p {your-access-token}

__1. Open a new Terminal window.

__2. Execute the following command to switch to the root user:

```
sudo -i
```

Enter **wasadmin** for password.

__3. Enter the following command:

```
whoami
```

You should see that you are **root** now.

```
root
```

Kubernetes networking addresses four concerns:

- Containers within a Pod use networking to communicate via loopback.
- Cluster networking provides communication between different Pods.
- The Service resource lets you expose an application running in Pods to be reachable from outside your cluster.
- You can also use Services to publish services only for consumption inside your cluster.

There are four types of Kubernetes services:

ClusterIP : This default type exposes the service on a cluster-internal IP.
You can reach the service only from within the cluster.

NodePort : This type of service exposes the service on each node's IP at a static port.
A ClusterIP service is created automatically, and the NodePort service will route to it.
From outside the cluster, you can contact the NodePort service by using "<NodeIP>:<NodePort>".

LoadBalancer : This service type exposes the service externally using the load balancer of your cloud provider. The external load balancer routes to your NodePort and ClusterIP services, which are created automatically.

ExternalName : This type maps the service to the contents of the externalName field (e.g., foo.bar.example.com). It does this by returning a value for the CNAME record.

See the documentation for further details.

https://kubernetes.io/docs/concepts/services-networking/

https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands

https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#expose

See the type flag usage in expose command.

```
$ kubectl expose (-f FILENAME | TYPE NAME) [--port=port] [--protocol=TCP|UDP|SCTP] [--target-port=number-or-name] [--name=name] [--external-ip=external-ip-of-service] [--type=type]
```

type	Type for this service: ClusterIP, NodePort, LoadBalancer, or External-Name. Default is 'ClusterIP'

Part 2 - Start-Up Kubernetes

In this part, you will start up Kubernetes. The lab environment is set up with MiniKube.

__1. Verify you have minikube installed:

```
minikube version
```

You should see:

```
minikube version: v1.22.0
```

__2. Stop MiniKube in case it is running:

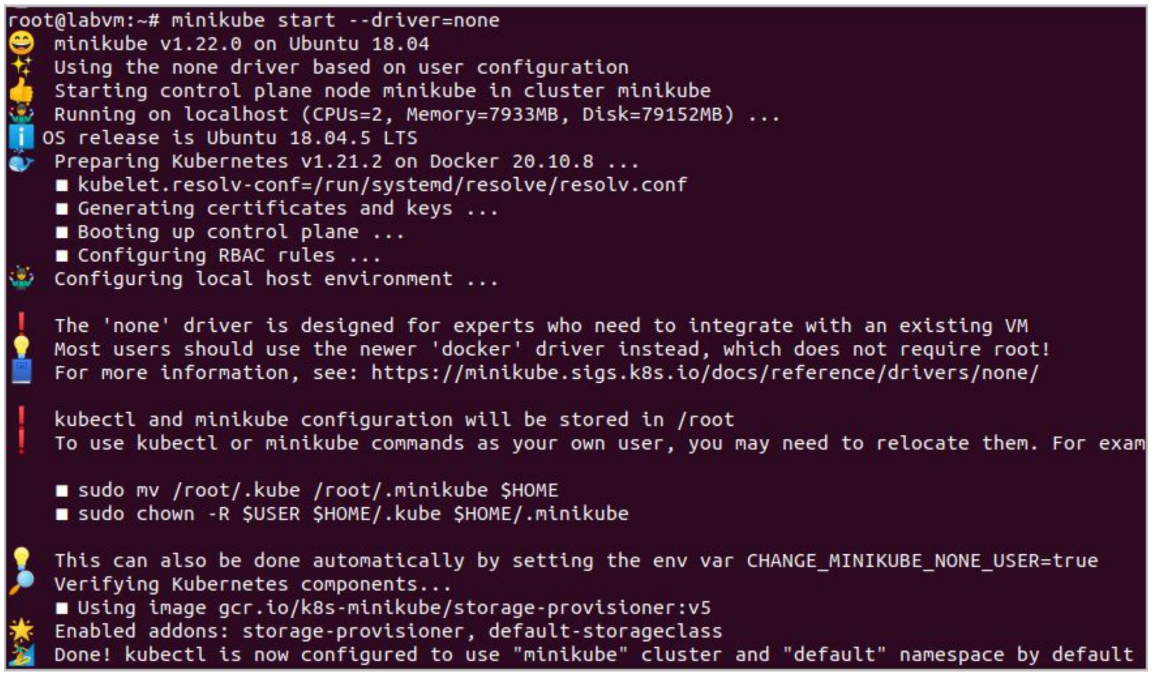
```
minikube stop
```

```
Note, if you see a time out error go to next step.
```

__3. Start minikube:

```
minikube start --driver=none
```

It may take a while to complete, you will see:
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default



```
root@labvm:~# minikube start --driver=none
😄  minikube v1.22.0 on Ubuntu 18.04
✨  Using the none driver based on user configuration
👍  Starting control plane node minikube in cluster minikube
🤹  Running on localhost (CPUs=2, Memory=7933MB, Disk=79152MB) ...
ℹ️ OS release is Ubuntu 18.04.5 LTS
🐳  Preparing Kubernetes v1.21.2 on Docker 20.10.8 ...
    ■ kubelet.resolv-conf=/run/systemd/resolve/resolv.conf
    ■ Generating certificates and keys ...
    ■ Booting up control plane ...
    ■ Configuring RBAC rules ...
🤹  Configuring local host environment ...

❗  The 'none' driver is designed for experts who need to integrate with an existing VM
💡  Most users should use the newer 'docker' driver instead, which does not require root!
📘  For more information, see: https://minikube.sigs.k8s.io/docs/reference/drivers/none/

❗  kubectl and minikube configuration will be stored in /root
❗  To use kubectl or minikube commands as your own user, you may need to relocate them. For exam

    ■ sudo mv /root/.kube /root/.minikube $HOME
    ■ sudo chown -R $USER $HOME/.kube $HOME/.minikube

💡  This can also be done automatically by setting the env var CHANGE_MINIKUBE_NONE_USER=true
🔎  Verifying Kubernetes components...
    ■ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Note: The above command performs the following operations:

1. Generates the certificates and then proceeds to provision a local Docker host.

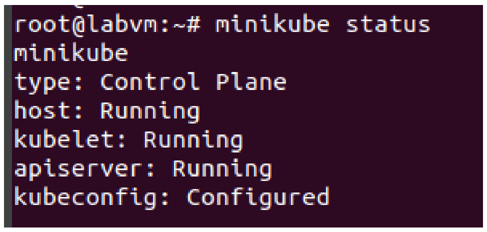
2. Starts up a control plane that provides various Kubernetes services.

3. Configures the default RBAC rules.

__4. In the terminal window, run the following command to verify the minikube status:

```
minikube status
```

It will show the following message:

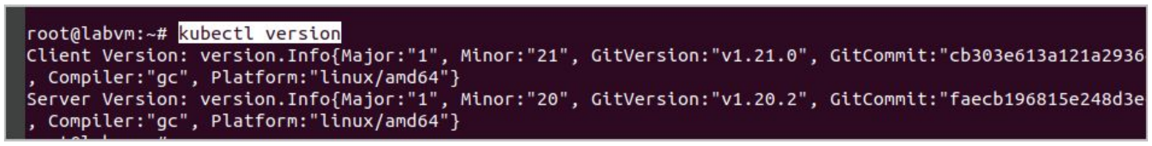


```
root@labvm:~# minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

__5. Verify you can execute kubectl and also obtain Kubernetes version:

```
kubectl version
```

Notice it lists the major and minor versions in JSON format.

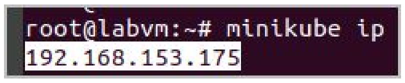


```
root@labvm:~# kubectl version
Client Version: version.Info{Major:"1", Minor:"21", GitVersion:"v1.21.0", GitCommit:"cb303e613a121a2936
, Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"20", GitVersion:"v1.20.2", GitCommit:"faecb196815e248d3e
, Compiler:"gc", Platform:"linux/amd64"}
```

__6. Run the following command to obtain the cluster IP address:

```
minikube ip
```

It will show the IP address of our cluster. Yours will be different.

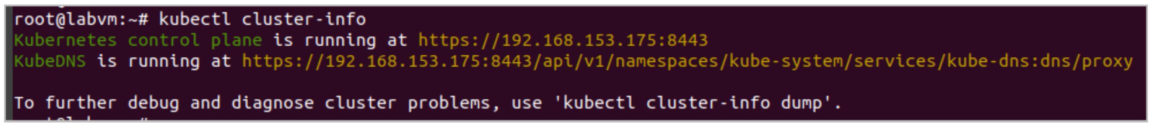


```
root@labvm:~# minikube ip
192.168.153.175
```

__7. 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.



```
root@labvm:~# kubectl cluster-info
Kubernetes control plane is running at https://192.168.153.175:8443
KubeDNS is running at https://192.168.153.175:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

__8. Navigate to the working directory:

```
cd /home/wasadmin/Works
```

__9. Check the command prompt. It should be:

```
root@labvm:/home/wasadmin/Works#
```

Part 3 - Deploy an Application

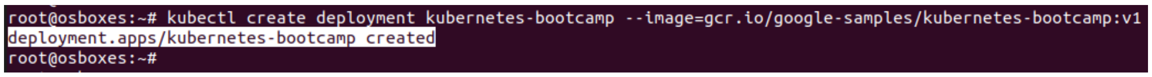
In this part, you will deploy a sample application to Kubernetes. After deployment, you will verify the status of the application.

__1. Run a sample application, on Kubernetes with the *kubectl* run command:

```
kubectl create deployment kubernetes-bootcamp
 --image=gcr.io/google-samples/kubernetes-bootcamp:v1
```

```
[Enter the command in 1 line]
```

The run command creates a new deployment. You need to provide the deployment name and app image location (include the full repository URL for images hosted outside Docker hub). You want to run the app on a specific port so you add the --port parameter.



```
root@osboxes:~# kubectl create deployment kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-bootcamp:v1
deployment.apps/kubernetes-bootcamp created
root@osboxes:~#
```

The above commands performed a few things for you:

* searched for a suitable node where an instance of the application could be run (we have only 1 available node)

* scheduled the application to run on that Node

* configured the cluster to reschedule the instance on a new Node when needed

kubernetes-bootcamp is a container which contains a simple service. When the service is executed, it returns “Hello Kubernetes Bootcamp” message.

__2. List the deployments:

```
kubectl get deployments
```

It should show the following outputs:

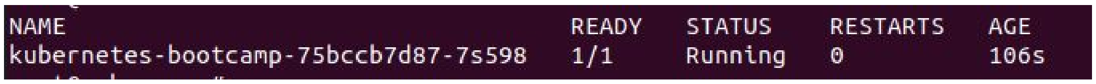
.

Wait and repeat the command until is shown 1/1 under Ready.

__3. View pods:

```
kubectl get pods
```

Notice it shows you the output similar to this:



```
NAME                                  READY   STATUS    RESTARTS   AGE
kubernetes-bootcamp-75bccb7d87-7s598  1/1     Running   0          106s
```

If it shows you ContainerCreating status, run the above command again after a few seconds. The status must show up as Running before you proceed to the next step of this lab.

Part 4 - Expose a Service and Test It

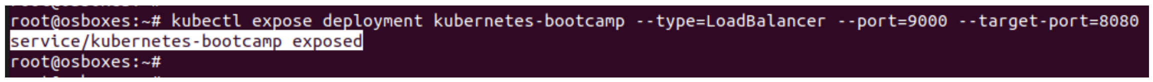
In this part, you will expose a service in the sample application which you deployed in the previous part of this lab. After exposing the service, you will test it by using curl command line tool.

__1. To create a new service in Kubernetes, based on the application which you have in the deployed image, and expose it:

```
kubectl expose deployment kubernetes-bootcamp --type=LoadBalancer --
port=9000 --target-port=8080
```

```
[Enter the command in 1 line]
```

The above command exposes the service in the kubernetes-bootcamp deployment. Since you want to load balance the service, you have specified the type as LoadBalancer. The service is hosted on port 8080 (which is specified in the Dockerfile of the sample application available in the kubernetes-bootcamp image which you deployed earlier in the lab. The load balancer port is configured as 9000.

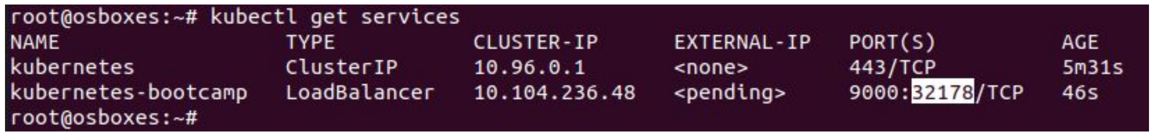


```
root@osboxes:~# kubectl expose deployment kubernetes-bootcamp --type=LoadBalancer --port=9000 --target-port=8080
service/kubernetes-bootcamp exposed
root@osboxes:~#
```

__2. Get the services list and ensure the service is exposed:

```
kubectl get services
```

Note: The output should be similar to this:



```
root@osboxes:~# kubectl get services
NAME                  TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes            ClusterIP      10.96.0.1       <none>        443/TCP          5m31s
kubernetes-bootcamp   LoadBalancer   10.104.236.48   <pending>     9000:32178/TCP   46s
root@osboxes:~#
```

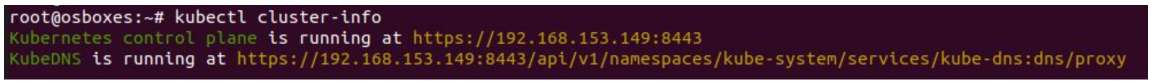
__3. Make a note of the port listed in front of /TCP (it's highlighted in the screen-shot displayed above. Your port number might be different)

9000 is the internal load balancer's port number which isn't accessible from outside the cluster. 30175 is accessible from outside the Kubernetes cluster.

__4. Obtain the cluster IP:

```
kubectl cluster-info
```

Make a note of only the IP. Yours will be different:



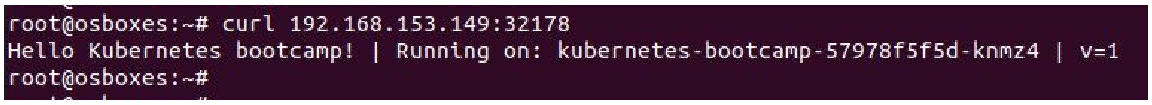
```
root@osboxes:~# kubectl cluster-info
Kubernetes control plane is running at https://192.168.153.149:8443
KubeDNS is running at https://192.168.153.149:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

__5. Run the following command to test the service:

```
curl <CLUSTER_IP>:<PORT>
```

Replace CLUSTER_IP and PORT with the values you noted in the earlier parts of this lab.

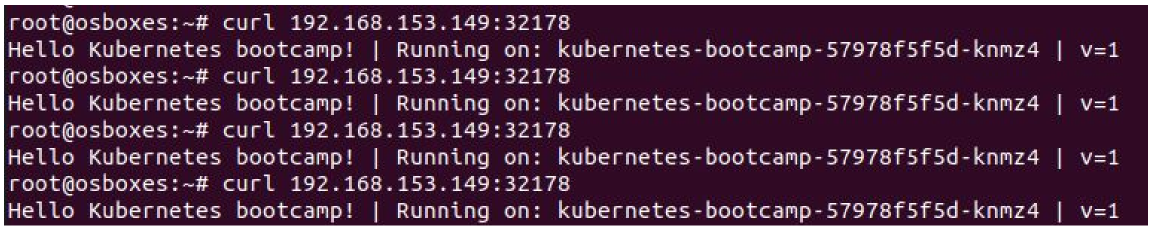
Notice the output looks like this: Hello Kubernetes Bootcamp! | Running on: <POD>



```
root@osboxes:~# curl 192.168.153.149:32178
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-57978f5f5d-knmz4 | v=1
root@osboxes:~#
```

__6. Run the curl command, mentioned in step above, multiple times.

Notice the same POD name is displayed each time like this:



```
root@osboxes:~# curl 192.168.153.149:32178
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-57978f5f5d-knmz4 | v=1
root@osboxes:~# curl 192.168.153.149:32178
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-57978f5f5d-knmz4 | v=1
root@osboxes:~# curl 192.168.153.149:32178
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-57978f5f5d-knmz4 | v=1
root@osboxes:~# curl 192.168.153.149:32178
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-57978f5f5d-knmz4 | v=1
```

Since the same pod name is displayed, it means load balancing is currently not implemented. You will implement load balancing in the next part of this lab.

Part 5 - Scale-Up the Pod(s) and Verify Load Balancing

In this part, you will scale up the pod and verify load balancing is working.

__1. Get the pods list:

```
kubectl get pods
```

Notice there's just one pod running for your kubernetes-bootcamp application.

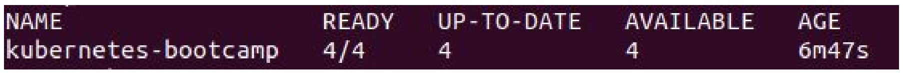
__2. Scale up the pod by adding 4 replicas:

```
kubectl scale deployments/kubernetes-bootcamp --replicas=4
```

__3. List the deployments:

```
kubectl get deployments
```

Notice it shows 4/4 for your kubernetes-bootcamp deployment.

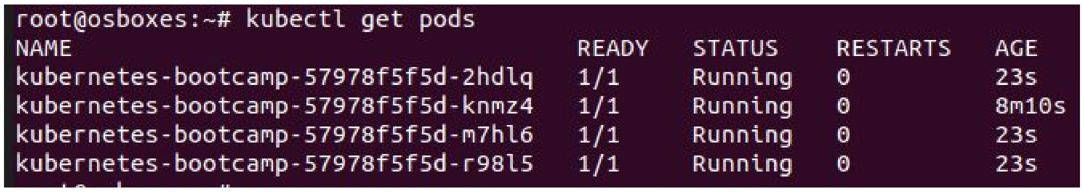


```
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
kubernetes-bootcamp   4/4     4            4           6m47s
```

__4. Get the pods list:

kubectl get pods

Notice there are 4 pods:

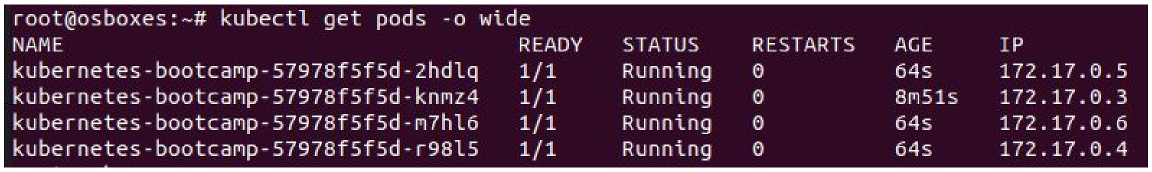


```
root@osboxes:~# kubectl get pods
NAME                                   READY   STATUS    RESTARTS   AGE
kubernetes-bootcamp-57978f5f5d-2hdlq   1/1     Running   0          23s
kubernetes-bootcamp-57978f5f5d-knmz4   1/1     Running   0          8m10s
kubernetes-bootcamp-57978f5f5d-m7hl6   1/1     Running   0          23s
kubernetes-bootcamp-57978f5f5d-r98l5   1/1     Running   0          23s
```

__5. Get the IP address of each pod:

kubectl get pods -o wide

Notice the IP address of each pod is different.

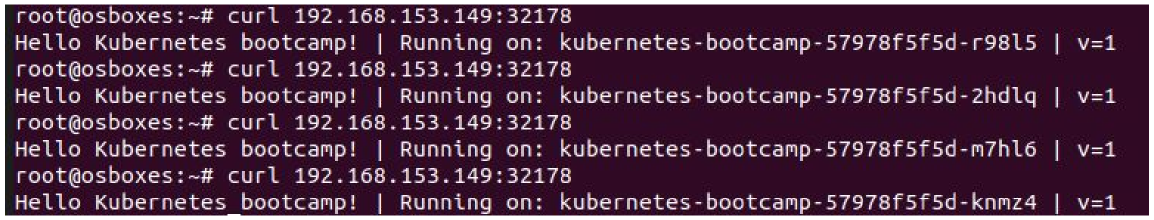


```
root@osboxes:~# kubectl get pods -o wide
NAME                                   READY   STATUS    RESTARTS   AGE     IP
kubernetes-bootcamp-57978f5f5d-2hdlq   1/1     Running   0          64s     172.17.0.5
kubernetes-bootcamp-57978f5f5d-knmz4   1/1     Running   0          8m51s   172.17.0.3
kubernetes-bootcamp-57978f5f5d-m7hl6   1/1     Running   0          64s     172.17.0.6
kubernetes-bootcamp-57978f5f5d-r98l5   1/1     Running   0          64s     172.17.0.4
```

__6. Run the same command that you ran before multiple times:

```
curl <CLUSTER_IP>:<PORT>
```

Notice it displays a different pod name each time. Due to load balancing, it hits a different IP each time. Kubernetes determines which pod has less load on it and sends the request to that pod. At times, it might show you the same pod name.



```
root@osboxes:~# curl 192.168.153.149:32178
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-57978f5f5d-r98l5 | v=1
root@osboxes:~# curl 192.168.153.149:32178
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-57978f5f5d-2hdlq | v=1
root@osboxes:~# curl 192.168.153.149:32178
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-57978f5f5d-m7hl6 | v=1
root@osboxes:~# curl 192.168.153.149:32178
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-57978f5f5d-knmz4 | v=1
```

Part 6 - Clean-Up

__1. Stop Kubernetes cluster:

```
minikube stop
```

__2. Type **exit** many times to close the terminal.

Part 7 - Review

In this lab, you deployed a sample service to Kubernetes cluster and configure load balancing. Each time you sent a request to your service, it hit a different IP and a different pod fulfilled the request.

Lab 6 - Working with Logs in Kubernetes

In this lab, you will view logs by using kubectl and also by using Fluentd, Elastic search, and Kibana. Although Kubernetes logs all messages written to standard out, if a pod is evicted from the cluster, all logs related to that pod are lost. Therefore, to write logs outside a container/pod, you will use fluentd, Elasticsearch, and Kibana. Fluentd is a logging agent that runs on each Kubernetes node and collects logs. Elasticsearch will index the contents of logs and Kibana will let you query the logs.

> When running multiple services and applications on a Kubernetes cluster, a centralized, cluster-level logging stack can help you quickly sort through and analyze the heavy volume of log data produced by your Pods. One popular centralized logging solution is the **E**lasticsearch, **F**luentd, and **K**ibana (EFK) stack.

Elasticsearch is a distributed and scalable full-text search database, that allows you to store and search large volumes of log events. Elasticsearch is a distributed, RESTful search and analytics engine capable of addressing a growing number of use cases. As the heart of the Elastic Stack, it centrally stores your data so you can discover the expected and uncover the unexpected.

https://www.elastic.co/products/elasticsearch

Fluentd is an open-source data collector, which lets you unify the data collection and consumption for a better use and understanding of data.

https://www.fluentd.org/

Kibana lets you visualize your Elasticsearch data and navigate the Elastic Stack so you can do anything from tracking query load to understanding the way requests flow through your apps.

https://www.elastic.co/products/kibana

Part 1 - Start up Kubernetes

In this Lab you will work in the 'Project-VM-4.0-2023-02-07'.

Start or connect to this VM if you don't have it opened yet. Use wasadmin for user and password.

__1. Open a new Terminal window.

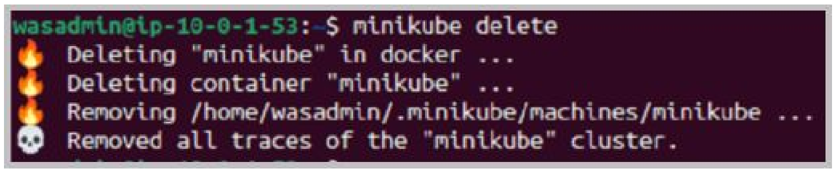
__2. Run the following command:

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

__3. Delete MiniKube:

```
minikube delete
```

You should see:



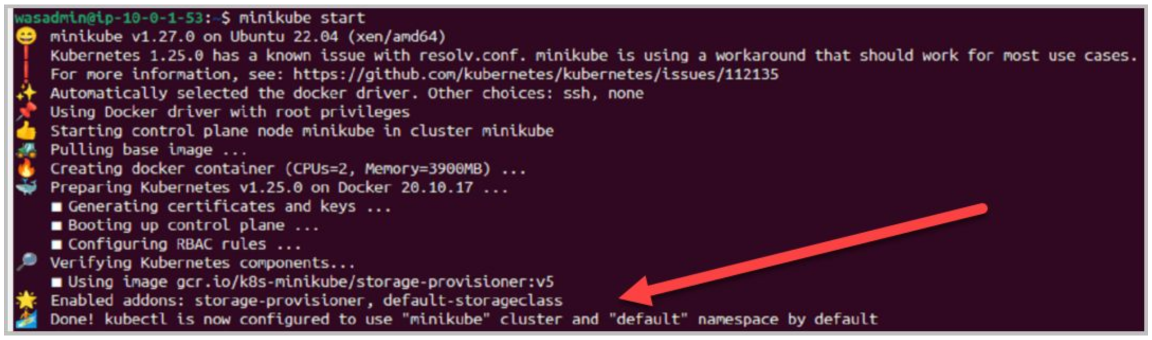
```
wasadmin@ip-10-0-1-53:~$ minikube delete
🔥  Deleting "minikube" in docker ...
🔥  Deleting container "minikube" ...
🔥  Removing /home/wasadmin/.minikube/machines/minikube ...
💀  Removed all traces of the "minikube" cluster.
```

__4. Start MiniKube:

```
minikube start
```

Wait for the service to start up. It can take up to 5 minutes.

You should see:

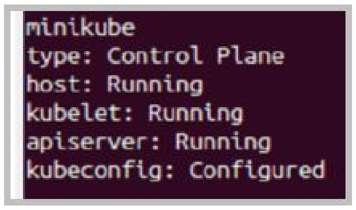


```
wasadmin@ip-10-0-1-53:~$ minikube start
😄  minikube v1.27.0 on Ubuntu 22.04 (xen/amd64)
❗  Kubernetes 1.25.0 has a known issue with resolv.conf. minikube is using a workaround that should work for most use cases.
❗  For more information, see: https://github.com/kubernetes/kubernetes/issues/112135
✨  Automatically selected the docker driver. Other choices: ssh, none
📌  Using Docker driver with root privileges
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
🔥  Creating docker container (CPUs=2, Memory=3900MB) ...
🐳  Preparing Kubernetes v1.25.0 on Docker 20.10.17 ...
    ■ Generating certificates and keys ...
    ■ Booting up control plane ...
    ■ Configuring RBAC rules ...
🔎  Verifying Kubernetes components...
    ■ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

__5. Get MiniKube status:

```
minikube status
```

Ensure your output looks like this:



```
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

__6. Switch to the home directory by entering the following commands:

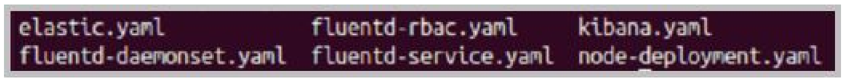
```
cd LabFiles/logging
```

__7. Make sure you can see the content of the directory:

```
ls
```

__8. Review the *.yaml files in /logging/kubernetes directory.

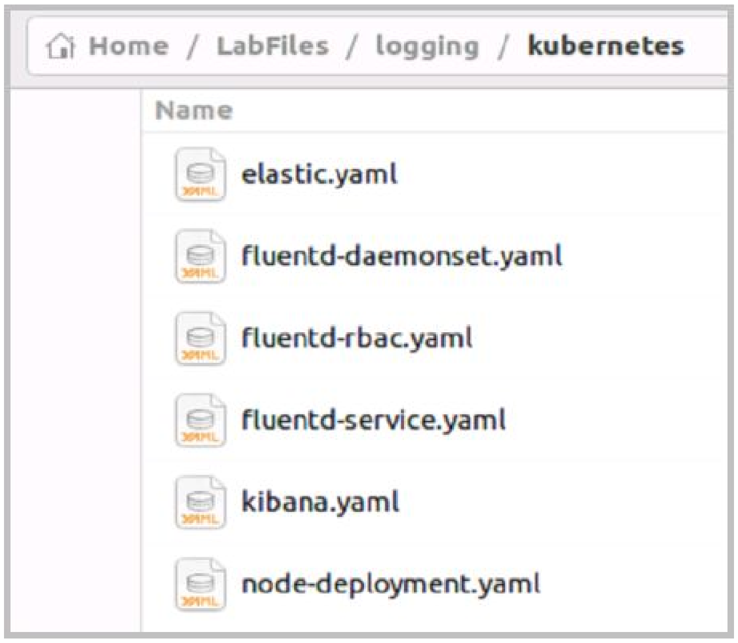
```
ls kubernetes
```



```
elastic.yaml            fluentd-rbac.yaml     kibana.yaml
fluentd-daemonset.yaml  fluentd-service.yaml  node-deployment.yaml
```

These yaml files have been pre-configured for Kubernetes deployment and ready to be deployed.

__9. Open a file browser and navigate to **Home \ LabFiles \ logging \ kubernetes**.



```
elastic.yaml
fluentd-daemonset.yaml
fluentd-rbac.yaml
fluentd-service.yaml
kibana.yaml
node-deployment.yaml
```

__10. Using a text editor, review elastic.yaml file. What can you do with this deployment?

__11. Review fluentd-daemonset.yaml. What can you do with this deployment?

__12. Review kibana.yaml. What can you do with this deployment?

__13. Review fluentd-service.yaml.

__14. Review fluentd-rbac.yaml

We are going to use kubectl create command with these logging stack preconfigured files to deploy Kubernetes resources.

kubectl create command can create a resource from a file or from stdin. JSON and YAML formats are accepted. Usage: $ kubectl create -f FILENAME

__15. Close all open text editors.

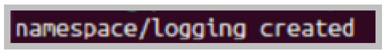
Part 2 - Deploy Elastic Search

In this part, you will deploy Elastic search portion of the Elastic Stack.

__1. Run the following command to create a namespace to organize the pods for logging:

```
kubectl create namespace logging
```

You should see:

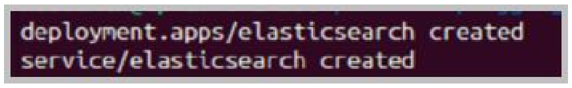


```
namespace/logging created
```

__2. Deploy Elastic Search component of the Elastic Stack:

```
kubectl create -f kubernetes/elastic.yaml -n logging
```

You should see:

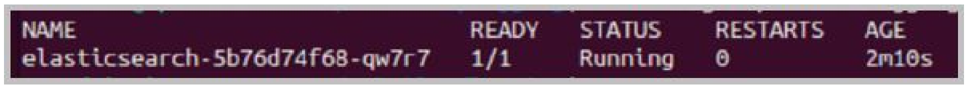


```
deployment.apps/elasticsearch created
service/elasticsearch created
```

__3. Verify the pods are created:

```
kubectl get pods -n logging
```

You should see:

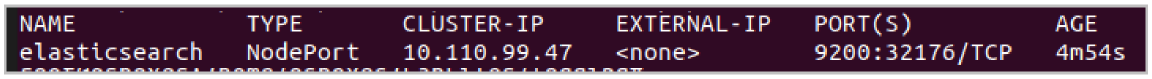

```
NAME                               READY   STATUS    RESTARTS   AGE
elasticsearch-5b76d74f68-qw7r7     1/1     Running   0          2m10s
```

If the pod(s) show 0/1, wait for a few minutes and run the command again. It must show 1/1 and Running status before you proceed to the next step.

__4. Verify the service was created:

```
kubectl get service -n logging
```

You should see something similar than this:


```
NAME            TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
elasticsearch   NodePort   10.110.99.47   <none>        9200:32176/TCP   4m54s
```

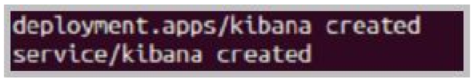
Part 3 - Deploy Kibana

In this part, you will deploy Kibana and verify it got deployed.

__1. Run the following command to deploy Kibana:

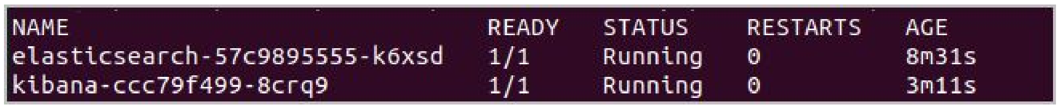
```
kubectl create -f kubernetes/kibana.yaml -n logging
```

You should see:


```
deployment.apps/kibana created
service/kibana created
```

__2. Verify the pods are created:

```
kubectl get pods -n logging
```

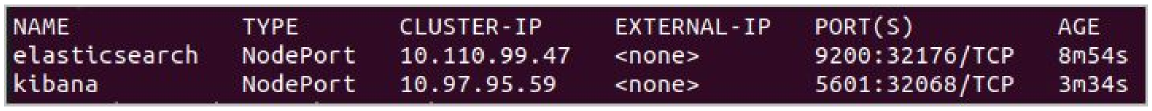


```
NAME                              READY   STATUS    RESTARTS   AGE
elasticsearch-57c9895555-k6xsd    1/1     Running   0          8m31s
kibana-ccc79f499-8crq9            1/1     Running   0          3m11s
```

If the pod(s) show 0/1, wait for a few minutes and run the command again. It must show 1/1 and Running status before you proceed to the next step.

__3. Verify the services were created:

```
kubectl get service -n logging
```



```
NAME            TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
elasticsearch   NodePort   10.110.99.47   <none>        9200:32176/TCP   8m54s
kibana          NodePort   10.97.95.59    <none>        5601:32068/TCP   3m34s
```

Part 4 - Deploy Fluentd

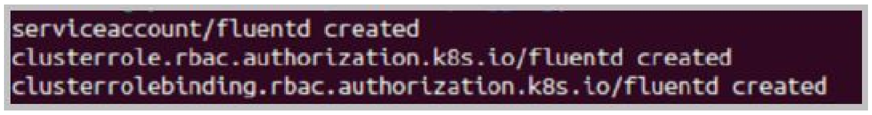
In this part, you will deploy a Fluentd logging agent to each node in the Kubernetes cluster, which will collect each container's log files running on that node. Fluentd will be deployed as a DaemonSet.

__1. Run the following command to configure RBAC (role-based access control) permissions so that Fluentd can access the logs:

```
kubectl create -f kubernetes/fluentd-rbac.yaml
```

The above command creates a ClusterRole which grants get, list, and watch permissions on pods and namespace objects. The ClusterRoleBinding then binds the ClusterRole to the ServiceAccount within the kube-system namespace.

You should see a message like this:



```
serviceaccount/fluentd created
clusterrole.rbac.authorization.k8s.io/fluentd created
clusterrolebinding.rbac.authorization.k8s.io/fluentd created
```

__2. Deploy fluentd as a DaemonSet so it can run on each Kubernetes node:

```
kubectl create -f kubernetes/fluentd-daemonset.yaml
```

You should see a message like this:

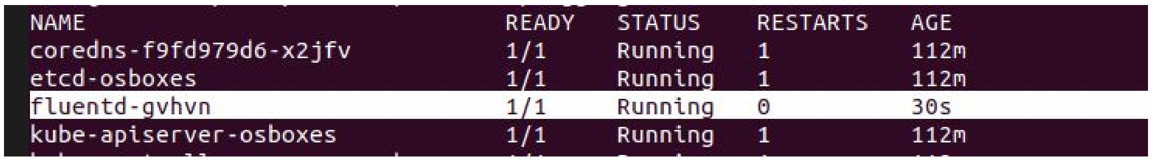
```
daemonset.apps/fluentd created
```

If you're running Kubernetes as a single node with Minikube, this will create a single Fluentd pod in the kube-system namespace.

__3. Verify fluentd is deployed and running:

```
kubectl get pods -n kube-system
```

Ensure fluentd is listed in the list. Make a note of the pod name. You will use it in the next step. Make sure fluentd shows 1/1. Wait a minute and repeat the command until shows 1/1.



```
NAME                             READY   STATUS    RESTARTS   AGE
coredns-f9fd979d6-x2jfv          1/1     Running   1          112m
etcd-osboxes                     1/1     Running   1          112m
fluentd-gvhvn                    1/1     Running   0          30s
kube-apiserver-osboxes           1/1     Running   1          112m
```

__4. Take a note of the logs:

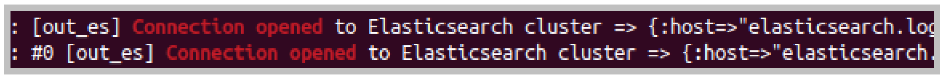
```
kubectl logs <fluentd-pod_name> -n kube-system | grep "Connection
opened"
```

```
[Enter this command in one line]
```

Use the fluentd pod name you noted down in the previous step of this part.

Ensure it shows a message like this:

Connection opened to Elasticsearch cluster =>
{:host=>"elasticsearch.logging", :port=>9200, :scheme=>"http"}



```
: [out_es] Connection opened to Elasticsearch cluster => {:host=>"elasticsearch.log
: #0 [out_es] Connection opened to Elasticsearch cluster => {:host=>"elasticsearch.
```

Part 5 - Connect to Kibana

__1. Obtain Kubernetes cluster IP address:

```
minikube ip
```

Make a note of the IP address. You will use it later in this part.

__2. Obtain port number assigned to Kibana service:

```
kubectl get services -n logging | grep kibana
```

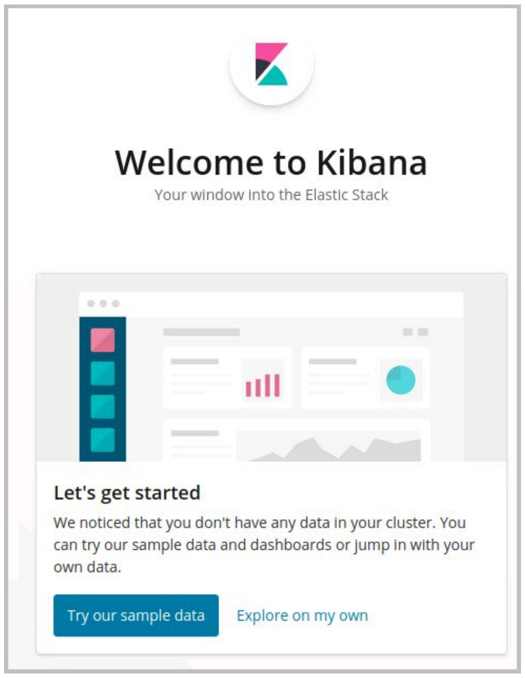
Make a note of the port number in front of /TCP. e.g. in the following example, the port number is 32683

```
kibana        NodePort   10.106.226.34    <none>       5601:32683/TCP   74s
```

__3. Open Firefox web browser in the VM and enter the following URL:

```
http://<MINIKUBE_IP>:<KIBANA_PORT>
```

Note: Ensure you use the values you noted in the previous steps of this part.
Ensure Kibana web page shows up.


Welcome to Kibana

Your window into the Elastic Stack

Let's get started

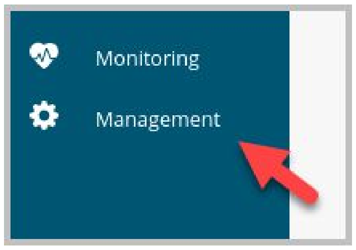
We noticed that you don't have any data in your cluster. You can try our sample data and dashboards or jump in with your own data.

__4. Click **Explore my own**.

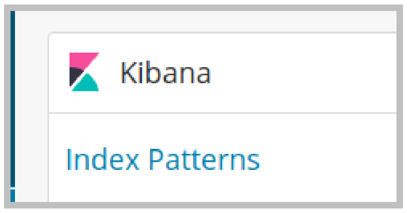
Part 6 - Configure Kibana

In this part, you will configure Kibana so it can get logs from fluentd.

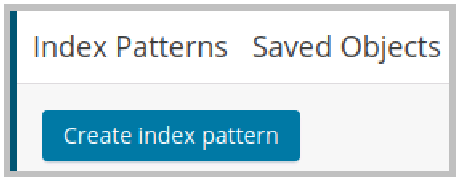
__1. In Firefox web browser where fluentd page is open, click **Management** on the left-hand side navigation.



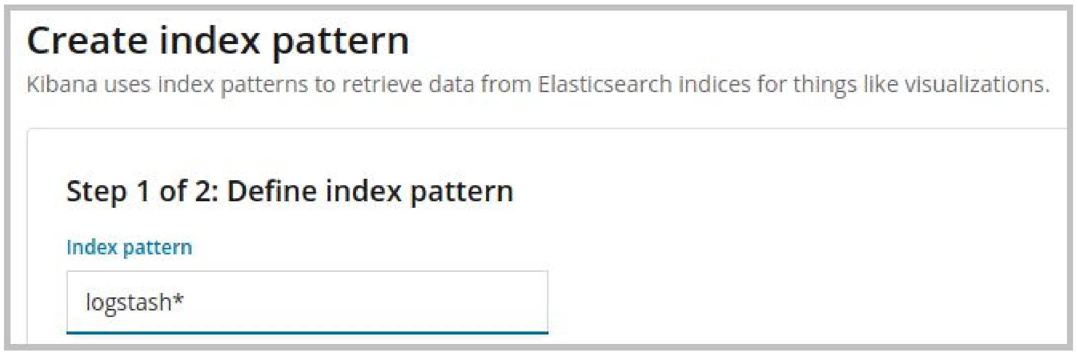
__2. Under **Kibana**, click **Index Patterns**.



__3. Click the **Create Index Pattern** button.



__4. In Index pattern field, type in **logstash***



Create index pattern

Kibana uses index patterns to retrieve data from Elasticsearch indices for things like visualizations.

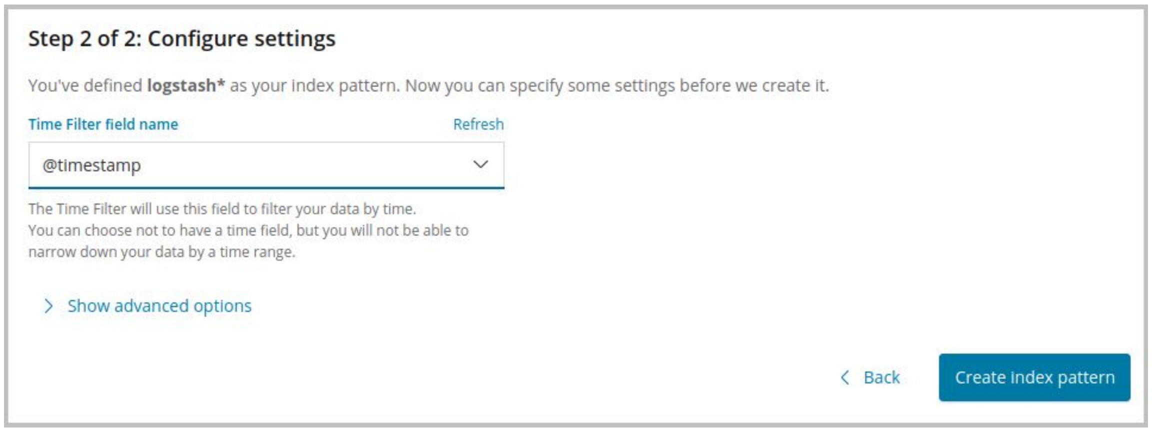
Step 1 of 2: Define index pattern

Index pattern: logstash*

__5. Click **Next step** button.

__6. In **Time Filter filed name** drop-down, select **@timestamp**

__7. Click **Create index pattern** button.



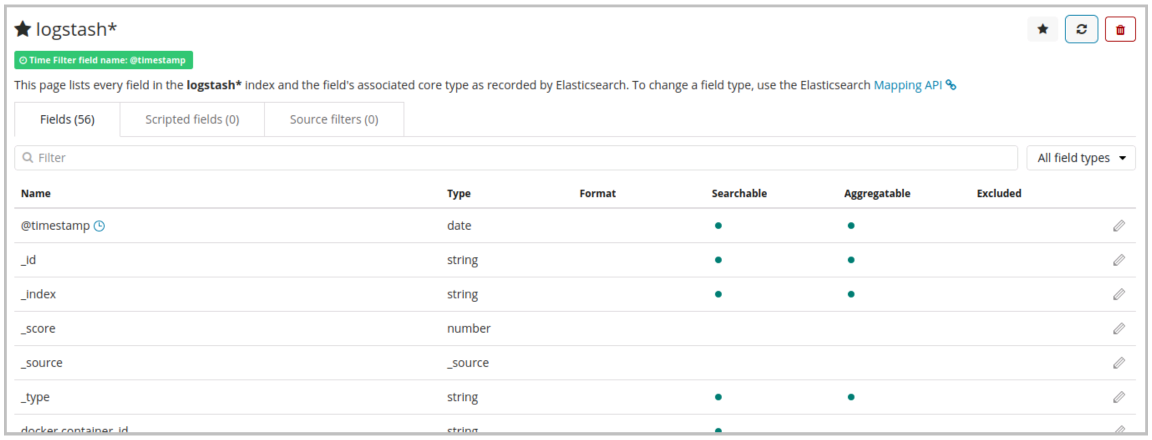
Step 2 of 2: Configure settings

You've defined **logstash*** as your index pattern. Now you can specify some settings before we create it.

Time Filter field name: @timestamp

The Time Filter will use this field to filter your data by time.
You can choose not to have a time field, but you will not be able to narrow down your data by a time range.

Note: It might show you a spinning wheel. Wait for about 5 seconds and proceed to the next part.



logstash*

Time Filter field name: @timestamp

This page lists every field in the **logstash*** index and the field's associated core type as recorded by Elasticsearch. To change a field type, use the Elasticsearch Mapping API

Name	Type	Format	Searchable	Aggregatable	Excluded
@timestamp	date		●	●	
_id	string		●	●	
_index	string		●	●	
_score	number				
_source	_source				
_type	string		●	●	
docker.container_id	string		●		

Part 7 - Deploy a Sample App and View Log using kubectl

In this part, you will build a Docker image with a custom node application that logs a message each second. You will deploy the image to Kubernetes cluster and view logs by using a basic technique.

__1. Back in the Terminal, view the source code of a sample node application:

```
gedit sample-app/index.js
```

Notice the code uses a logging package to log messages.

```
const SimpleNodeLogger = require('simple-node-logger');
const opts = {
  timestampFormat:'YYYY-MM-DD HH:mm:ss.SSS'
};
const log = SimpleNodeLogger.createSimpleLogger(opts);

function logMessage() {
  setTimeout(() => {
    log.info('log test');
    logMessage();
  }, 1000);
}

logMessage();
```

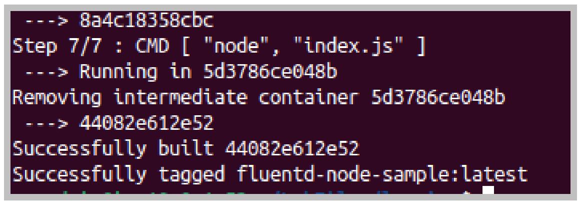
__2. Close the editor.

__3. Build a Docker image that contains your custom node application:

```
docker build -t fluentd-node-sample:latest -f sample-app/Dockerfile
sample-app
```

```
[Enter this command in one line]
```

You should see:



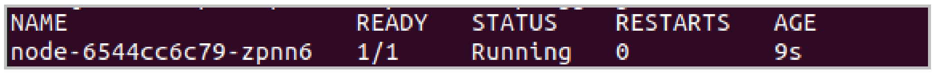
```
 ---> 8a4c18358cbc
Step 7/7 : CMD [ "node", "index.js" ]
 ---> Running in 5d3786ce048b
Removing intermediate container 5d3786ce048b
 ---> 44082e612e52
Successfully built 44082e612e52
Successfully tagged fluentd-node-sample:latest
```

__4. Deploy the image to Kubernetes:

```
kubectl create -f kubernetes/node-deployment.yaml
```

__5. Get pod list and make a note of the pod name:

```
kubectl get pods
```



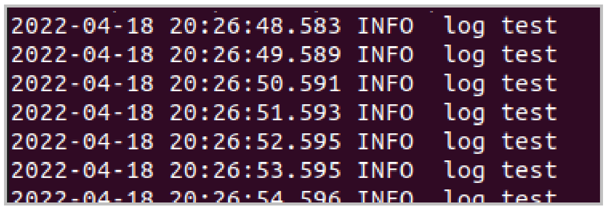
```
NAME                     READY   STATUS    RESTARTS   AGE
node-6544cc6c79-zpnn6    1/1     Running   0          9s
```

__6. View logs using kubectl:

```
kubectl logs <pod_name>
```

Note: Use the pod name which you noted down in the previous step of this lab.

__7. Verify the log shows the message log test.



```
2022-04-18 20:26:48.583 INFO  log test
2022-04-18 20:26:49.589 INFO  log test
2022-04-18 20:26:50.591 INFO  log test
2022-04-18 20:26:51.593 INFO  log test
2022-04-18 20:26:52.595 INFO  log test
2022-04-18 20:26:53.595 INFO  log test
2022-04-18 20:26:54.596 INFO  log test
```

Part 8 - View Logs using Kibana

__1. Back in the browser, click **Discover** on the left-hand side navigation.

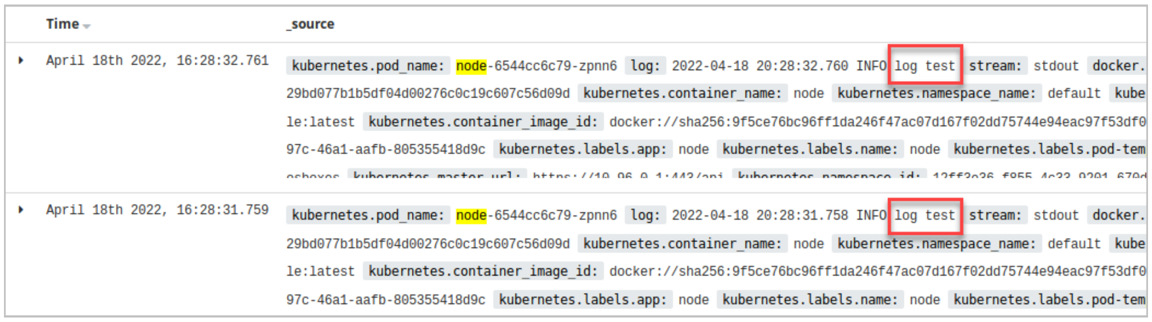
__2. Click **Add a filter** hyperlink.

__3. Add a filter with the following values:

```
Filter: kubernetes.pod_name
Operator: is
Value: node
```

__4. Click **Save** button.

__5. Ensure “log test” shows up in the logs.



```
Time          _source
April 18th 2022, 16:28:32.761   kubernetes.pod_name: node-6544cc6c79-zpnn6 log: 2022-04-18 20:28:32.760 INFO log test stream: stdout docker.
29bd077b1b5df04d00276c0c19c607c56d09d kubernetes.container_name: node kubernetes.namespace_name: default kube
le:latest kubernetes.container_image_id: docker://sha256:9f5ce76bc96ff1da246f47ac07d167f02dd75744e94eac97f53df0
97c-46a1-aafb-805355418d9c kubernetes.labels.app: node kubernetes.labels.name: node kubernetes.labels.pod-tem

April 18th 2022, 16:28:31.759   kubernetes.pod_name: node-6544cc6c79-zpnn6 log: 2022-04-18 20:28:31.758 INFO log test stream: stdout docker.
29bd077b1b5df04d00276c0c19c607c56d09d kubernetes.container_name: node kubernetes.namespace_name: default kube
le:latest kubernetes.container_image_id: docker://sha256:9f5ce76bc96ff1da246f47ac07d167f02dd75744e94eac97f53df0
97c-46a1-aafb-805355418d9c kubernetes.labels.app: node kubernetes.labels.name: node kubernetes.labels.pod-tem
```

BAH MSD Bootcamp students may skip the next part 10 Clean up activity. You may have to use these logging stack deployments in your project work.

Part 9 - Clean-Up

__1. Stop Kubernetes:

```
minikube stop
```

__2. Delete the Kubernetes cluster:

```
minikube delete
```

__3. Close the terminal window.

__4. Close the browser.

Part 10 - Review

In this lab, you explored logging using kubectl, fluentd, elasticsearch, and kibana.

Lab 7 - Monitoring Kubernetes with Prometheus

In this lab, you will use Prometheus to monitor Kubernetes. Prometheus is an open-source monitoring solution. You will use various manifest files to declaratively install and configure Prometheus in a Kubernetes cluster. You will view various metrics to monitor your Kubernetes cluster.

Part 1 - Setting the Stage

In this lab, you will continue working in the 'VM_WA2894_REL_1_1'.

Make sure you ran the following command in a previous lab:

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

__1. Open a new Terminal window.

__2. Switch the logged-in user to **root**:

```
sudo -i
```

When prompted for the logged-in user's password, enter **wasadmin**

__3. Enter the following command:

```
whoami
```

You should see that you are **root** now.

```
root
```

Note: Now that you are **root** on your Lab server, beware of the system-wrecking consequences of issuing a wrong command.

__4. Switch to the home directory:

```
cd ../home/wasadmin
```

__5. Switch to the LabFiles/monitoring directory:

```
cd LabFiles/monitoring
```

Part 2 - Start-Up Kubernetes

In this part, you will start up the Kubernetes cluster. The lab setup comes with MiniKube preinstalled.

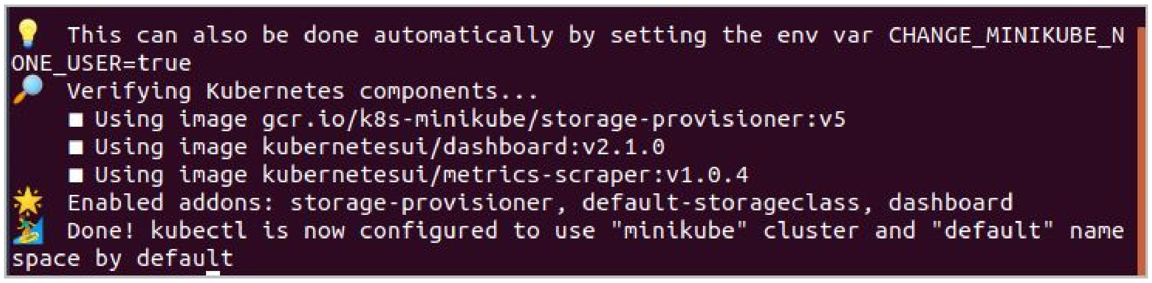
__1. Check if MiniKube is running [it may not even exist]:

```
minikube status
```

__2. If MiniKube is not running or it is not found, start MiniKube:

```
minikube start --vm-driver="none"
```

Wait for the service to start up. It can take up to 5 minutes. If it fails, try again. It’s a very resource-intensive service and takes time.

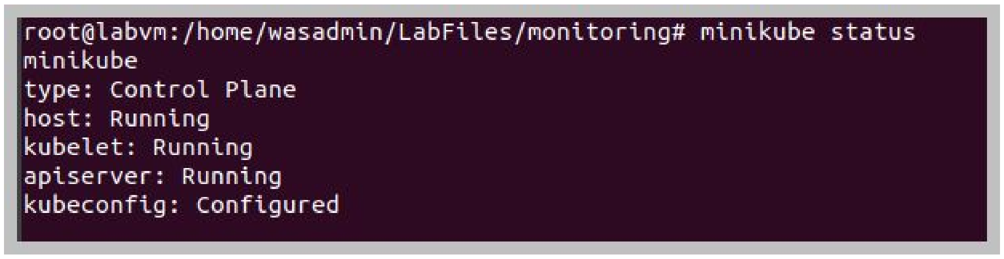


```
💡  This can also be done automatically by setting the env var CHANGE_MINIKUBE_N
ONE_USER=true
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
    ▪ Using image kubernetesui/dashboard:v2.1.0
    ▪ Using image kubernetesui/metrics-scraper:v1.0.4
🌟  Enabled addons: storage-provisioner, default-storageclass, dashboard
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" name
space by default
```

__3. Get MiniKube status again:

```
minikube status
```

Ensure your output looks like this. Your IP address might be different.



```
root@labvm:/home/wasadmin/LabFiles/monitoring# minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Part 3 - Create a Namespace and Configure Permissions

__1. Run the following command to create a namespace to organize the pods for monitoring:

```
kubectl create namespace monitoring
```

If you don't create a dedicated namespace, all the Prometheus kubernetes deployment objects get deployed on the default namespace.

__2. Run the following command to configure cluster read permission to this namespace so that Prometheus can fetch the metrics from Kubernetes API:

```
kubectl create -f prometheus-rbac.yaml
```

Prometheus utilizes Kubernetes APIs to read the available metrics from Nodes, Pods, and Deployments. That is why, an RBAC policy with read access to required API groups and bind the policy to the monitoring namespace is required.

Part 4 - Configure Kube State Metrics

In this part, you will configure the Kube State metrics service. It talks to the Kubernetes API server to get details about API objects, such as deployments, pods, and services. Without this service, Prometheus won't be able to scrape metrics, such as container memory usage, resource requests and limits, and monitor pod status. Kube state metrics service exposes all the metrics on /metrics URI.

__1. Execute the following command from the terminal to configure the Kube State metrics service:

```
kubectl create -f kube-state-metrics-configs
```

This command executes multiple YAML manifests located in the kube-state-metrics-configs directory. In summary, the following operations get performed:

• A service account is created that will be used to gather various metrics

• A cluster role is created for kube state metrics to access all Kubernetes API objects.

• A cluster role binding is created to bind the service account with the cluster role.

• Kube state metrics deployment is performed that contains the actual code to ensure metrics are processed.

• A service is created to expose the kube state metrics service. Prometheus will use this service to gather metrics.

__2. Execute the following command to verify kube metrics service is deployed:

```
kubectl get pods --all-namespaces | grep kube-state-metrics
```

Ensure the status shows as 1/1 Running before you proceed to the next step. It may take a minute before it shows that status.

__3. Execute the following command to obtain the ClusterIP assigned to the kube metrics service. Make a note of the ClusterIP. We will refer to it as “Kube metrics service Cluster IP” later in the lab:

```
kubectl get services --all-namespaces | grep kube-state-metrics
```

You should see something like:

```
kube-system   kube-state-metrics   ClusterIP   10.107.180.125   <none>
8080/TCP,8081/TCP      3m14s
```

__4. Execute the following command to obtain the Kubernetes ClusterIP. Make a note of the value. We will refer to it as “Kubernetes ClusterIP”:

```
kubectl get services
```

You should see something like:

```
kubernetes   ClusterIP   10.96.0.1   <none>      443/TCP   43d
```

Part 5 - Create a Config Map

In this part, you will create a config map with Prometheus configurations/rules that will be mounted to the Prometheus container under /etc/prometheus as prometheus.yaml.

__1. View config map file and keep the file open in the editor:

```
gedit prometheus-config-map.yaml
```

The file contains all the configurations to dynamically discover pods and services running in the Kubernetes cluster. There are the following configurations:

kubernetes-apiservers: It gets all the metrics from the API servers.

kubernetes-nodes: All Kubernetes node metrics will be collected with this job.

kubernetes-pods: All the pod metrics will be discovered if the pod metadata is annotated with prometheus.io/scrape and prometheus.io/port annotations.

kubernetes-cadvisor: Collects all cAdvisor metrics.

kubernetes-service-endpoints: All the Service endpoints will be scrapped if the service metadata is annotated with prometheus.io/scrape and prometheus.io/port annotations. It will be blackbox monitoring.

prometheus.rules will contain all the alert rules for sending alerts to alert manager.

The config map with all the Prometheus scrape config and alerting rules gets mounted to the Prometheus container in /etc/prometheus location as prometheus.yaml and prometheus.rules files.

prometheus.yaml: This is the main Prometheus configuration which holds all the scrape configs, service discovery details, storage locations, data retention configs, etc)

prometheus.rules: This file contains all the Prometheus alerting rules

__2. Search for “TODO1” and replace the value with the “Kube Metrics Service ClusterIP” value you noted earlier in the lab. (Make the change as shown in bold below. Ensure you use the value you noted earlier in the lab):

```
    - job_name: 'kube-state-metrics'
      static_configs:
        - targets: ['10.107.180.125:8080']
```

This configuration ensures Prometheus can scrape metrics provided by the kube metrics service.

__3. Search for “TODO2” and replace the value with the “Kubernetes ClusterIP” value you noted earlier in the lab. (Make the change as shown in bold below. Ensure you use the value you noted earlier in the lab):

```
        relabel_configs:
        - action: labelmap
          regex: __meta_kubernetes_node_label_(.+)
        - target_label: __address__
          replacement: 10.96.0.1:443
        - source_labels: [__meta_kubernetes_node_name]
```

This configuration ensures Prometheus can scrape metrics provided by Kubernetes.

__4. Save the file and close the editor.

__5. Run the following command to create the config map in Kubernetes:

```
kubectl create -f prometheus-config-map.yaml
```

__6. Verify the config map resource is created successfully:

```
kubectl describe configmap prometheus-server-conf -n monitoring
```

Part 6 - Create Prometheus Deployment

In this part, you will install Prometheus using the official image from Docker hub. You will also mount the Prometheus config map as a file under /etc/prometheus.

__1. Run the following command to deploy Prometheus to Kubernetes cluster’s monitoring namespace:

```
kubectl create -f prometheus-deployment.yaml
```

__2. Verify the deployment is done successfully:

```
kubectl get deployments --namespace=monitoring
```

Ensure Prometheus status shows 1/1 under Ready. Wait a few moments and repeat the command until the Ready shows 1/1

__3. Run the following command to expose Prometheus as a service:

```
kubectl create -f prometheus-service.yaml --namespace=monitoring
```

Note: Alternatively, you can use the following imperative command to expose Prometheus as a service.

kubectl port-forward <prometheus_pod> 8080:9090 -n monitoring.

Part 7 - Deploy a Sample App

In this part, you will deploy Nginx to Kubernetes so you can monitor it using Prometheus later in the lab.

__1. Execute the following command to deploy Nginx to Kubernetes:

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

Part 8 - Use Prometheus

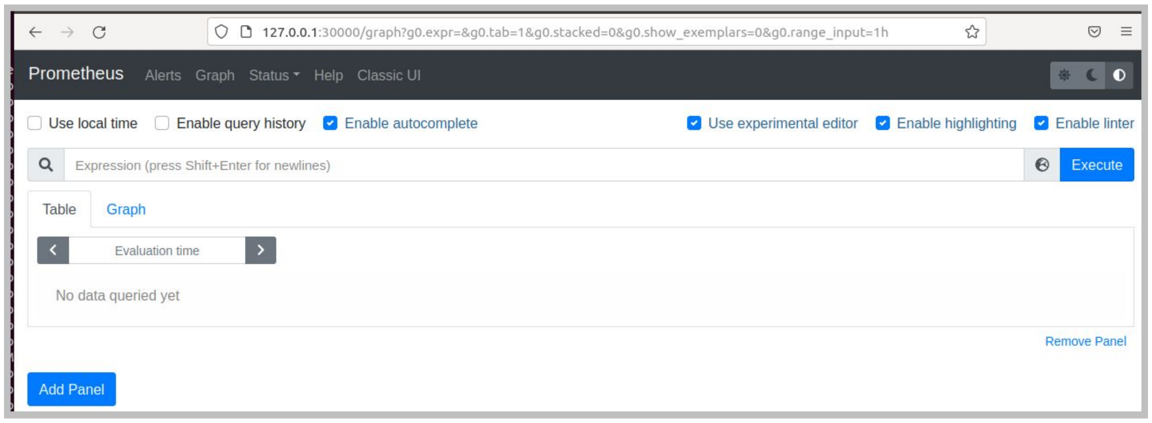
In this part, you will access the Prometheus service and use various metrics.

__1. Open **Firefox** browser in the VM and navigate to the following URL:

```
http://127.0.0.1:30000
```

Note: port 30000 is configured in prometheus-service.yaml.

The default site should look like this:



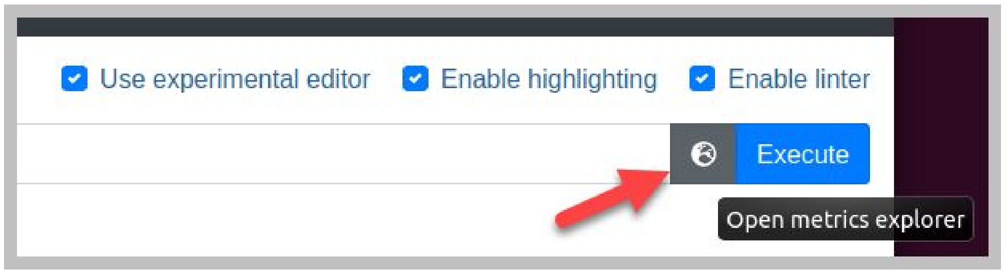
127.0.0.1:30000/graph?g0.expr=&g0.tab=1&g0.stacked=0&g0.show_exemplars=0&g0.range_input=1h

Use local time — Enable query history — ✓ Enable autocomplete — ✓ Use experimental editor — ✓ Enable highlighting — ✓ Enable linter

Expression (press Shift+Enter for newlines)

No data queried yet

__2. Next to the search text box, click **open metrics explorer**.



✓ Use experimental editor — ✓ Enable highlighting — ✓ Enable linter

Notice there are several metrics available.

__3. Scroll through the list and click **container_memory_usage_bytes**

__4. Click **Execute** button.

Notice in the Table view, it shows the list of all deployed pods along with memory usage for each deployment.

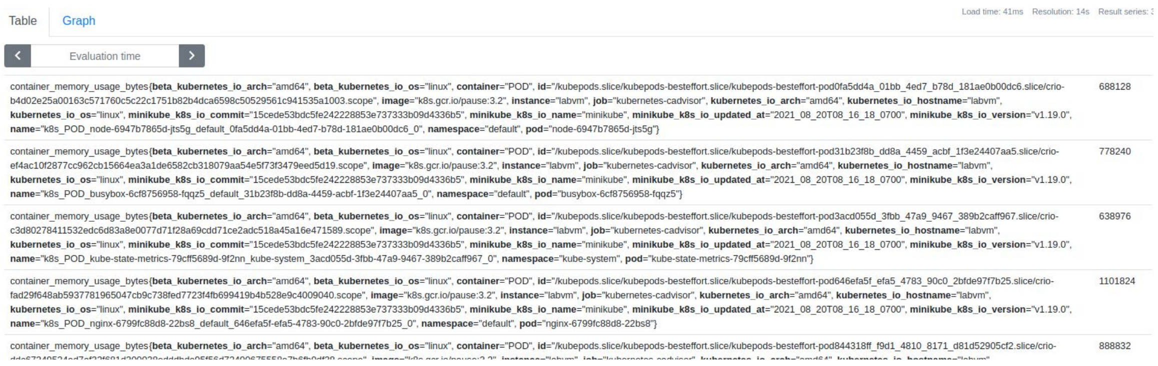


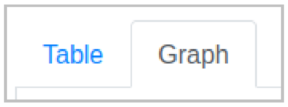
Table Graph

Load time: 41ms Resolution: 14s Result series: 3

Evaluation time

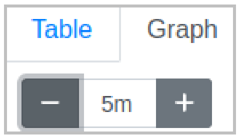
container_memory_usage_bytes{**beta_kubernetes_io_arch**="amd64", **beta_kubernetes_io_os**="linux", **container**="POD", **id**="/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-pod0fa5dd4a_01bb_4ed7_b78d_181ae0b00dc6.slice/crio-b4d02e25a00163c571760c5c22c1751b82b4dca6598c50529561c941535a1003.scope", **image**="k8s.gcr.io/pause:3.2", **instance**="labvm", **job**="kubernetes-cadvisor", **kubernetes_io_arch**="amd64", **kubernetes_io_hostname**="labvm", **kubernetes_io_os**="linux", **minikube_k8s_io_commit**="15cede53bdc5fe242228853e737333b09d4336b5", **minikube_k8s_io_name**="minikube", **minikube_k8s_io_updated_at**="2021_08_20T08_16_18_0700", **minikube_k8s_io_version**="v1.19.0", **name**="k8s_POD_node-6947b7865d-jts5g_default_0fa5dd4a-01bb-4ed7-b78d-181ae0b00dc6_0", **namespace**="default", **pod**="node-6947b7865d-jts5g"}	688128
container_memory_usage_bytes{**beta_kubernetes_io_arch**="amd64", **beta_kubernetes_io_os**="linux", **container**="POD", **id**="/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-pod31b23f8b_dd8a_4459_acbf_1f3e24407aa5.slice/crio-ef4ac10f2877cc962cb15664ea3a1de6582cb318079aa54e5f73f3479eed5d19.scope", **image**="k8s.gcr.io/pause:3.2", **instance**="labvm", **job**="kubernetes-cadvisor", **kubernetes_io_arch**="amd64", **kubernetes_io_hostname**="labvm", **kubernetes_io_os**="linux", **minikube_k8s_io_commit**="15cede53bdc5fe242228853e737333b09d4336b5", **minikube_k8s_io_name**="minikube", **minikube_k8s_io_updated_at**="2021_08_20T08_16_18_0700", **minikube_k8s_io_version**="v1.19.0", **name**="k8s_POD_busybox-6cf8756958-fqqz5_default_31b23f8b-dd8a-4459-acbf-1f3e24407aa5_0", **namespace**="default", **pod**="busybox-6cf8756958-fqqz5"}	778240
container_memory_usage_bytes{**beta_kubernetes_io_arch**="amd64", **beta_kubernetes_io_os**="linux", **container**="POD", **id**="/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-pod3acd055d_3fbb_47a9_9467_389b2caff967.slice/crio-c3d80278411532edc6d83a8e0077d71f28a69cdd71ce2adc518a45a16e471589.scope", **image**="k8s.gcr.io/pause:3.2", **instance**="labvm", **job**="kubernetes-cadvisor", **kubernetes_io_arch**="amd64", **kubernetes_io_hostname**="labvm", **kubernetes_io_os**="linux", **minikube_k8s_io_commit**="15cede53bdc5fe242228853e737333b09d4336b5", **minikube_k8s_io_name**="minikube", **minikube_k8s_io_updated_at**="2021_08_20T08_16_18_0700", **minikube_k8s_io_version**="v1.19.0", **name**="k8s_POD_kube-state-metrics-79cff5689d-9f2nn_kube-system_3acd055d-3fbb-47a9-9467-389b2caff967_0", **namespace**="kube-system", **pod**="kube-state-metrics-79cff5689d-9f2nn"}	638976
container_memory_usage_bytes{**beta_kubernetes_io_arch**="amd64", **beta_kubernetes_io_os**="linux", **container**="POD", **id**="/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-pod646efa5f_efa5_4783_90c0_2bfde97f7b25.slice/crio-fad29f648ab5937781965047cb9c738fed7723f4fb699419b4b528e9c4009040.scope", **image**="k8s.gcr.io/pause:3.2", **instance**="labvm", **job**="kubernetes-cadvisor", **kubernetes_io_arch**="amd64", **kubernetes_io_hostname**="labvm", **kubernetes_io_os**="linux", **minikube_k8s_io_commit**="15cede53bdc5fe242228853e737333b09d4336b5", **minikube_k8s_io_name**="minikube", **minikube_k8s_io_updated_at**="2021_08_20T08_16_18_0700", **minikube_k8s_io_version**="v1.19.0", **name**="k8s_POD_nginx-6799fc88d8-22bs8_default_646efa5f-efa5-4783-90c0-2bfde97f7b25_0", **namespace**="default", **pod**="nginx-6799fc88d8-22bs8"}	1101824
container_memory_usage_bytes{**beta_kubernetes_io_arch**="amd64", **beta_kubernetes_io_os**="linux", **container**="POD", **id**="/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-pod844318ff_f9d1_4810_8171_d81d52905cf2.slice/crio-	888832

__5. Click **Graph**.



Notice the graph displays a lot of lines, one line per container.

__6. Change the range to 5m to be able to see a bigger graph.



__7. To display memory usage for a specific container, change the query to the following and click **Execute**:

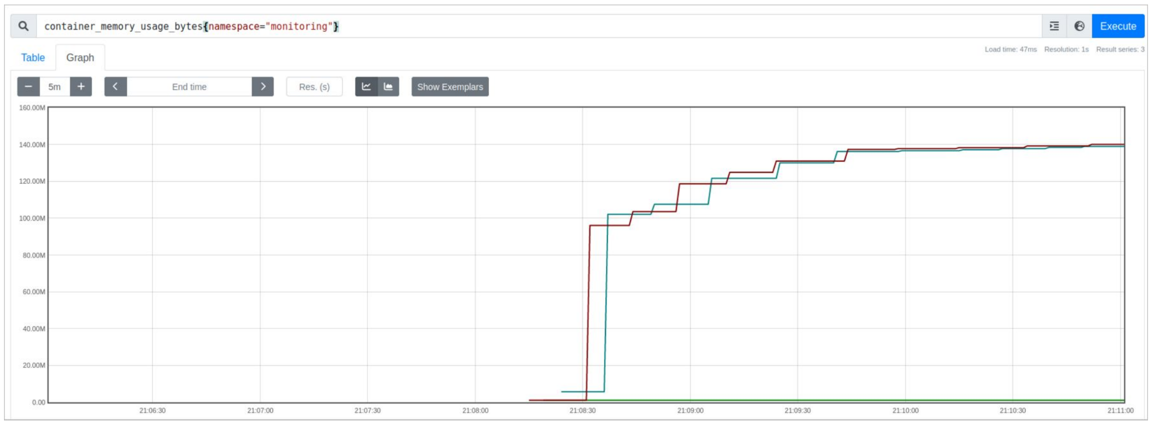
```
container_memory_usage_bytes{container="nginx"}
```

Ensure you use the curly-braces {} and not the parenthesis ()

```
container_memory_usage_bytes{container="nginx"}
```

__8. Change the query to the following and then click **Execute** to see the total memory consumed by deployments in the monitoring namespace:

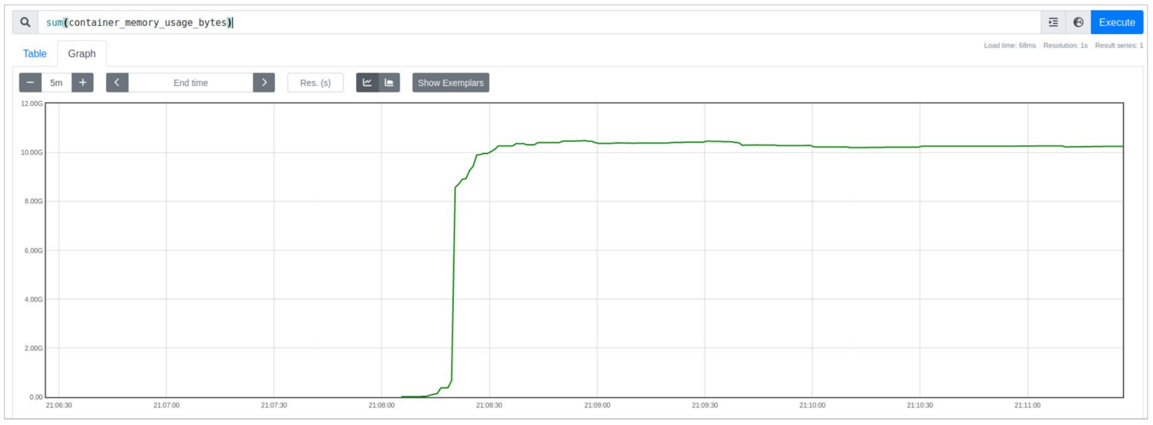
```
container_memory_usage_bytes{namespace="monitoring"}
```



__9. Change the query to the following and click **Execute** to see the total memory consumed by all the containers in the cluster:

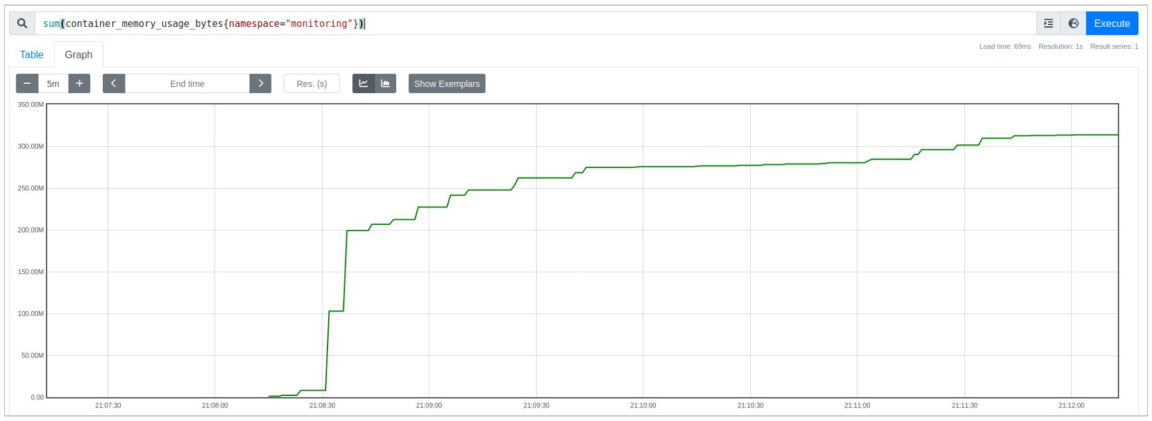
```
sum(container_memory_usage_bytes)
```

Ensure you use the parenthesis () for the sum aggregate function.



__10. Change the query to the following and click **Execute** to view the total memory consumed by containers in “monitoring” namespace of the cluster:

```
sum(container_memory_usage_bytes{namespace="monitoring"})
```

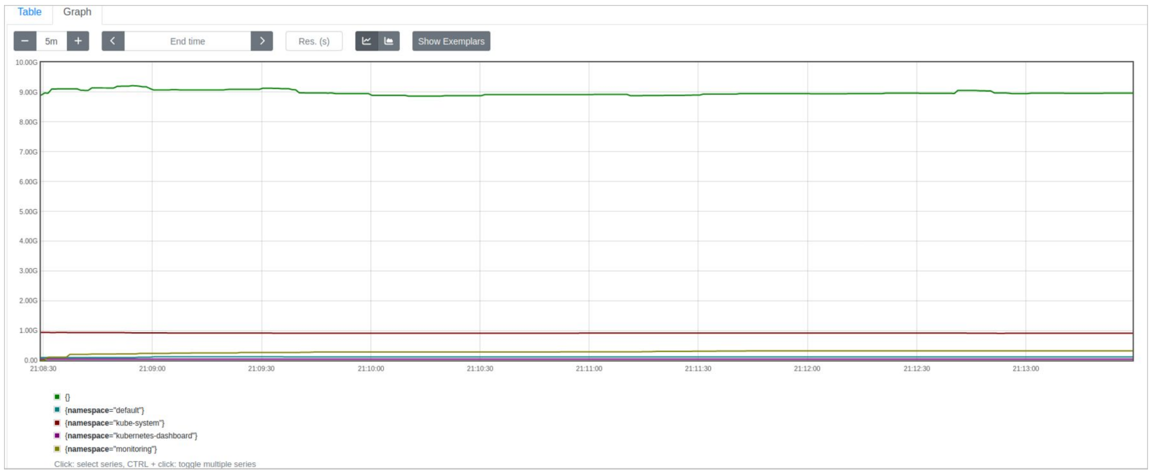


```
sum(container_memory_usage_bytes{namespace="monitoring"})
```

__11. Change the query to the following and click **Execute** to view memory consumption break-down by namespace:

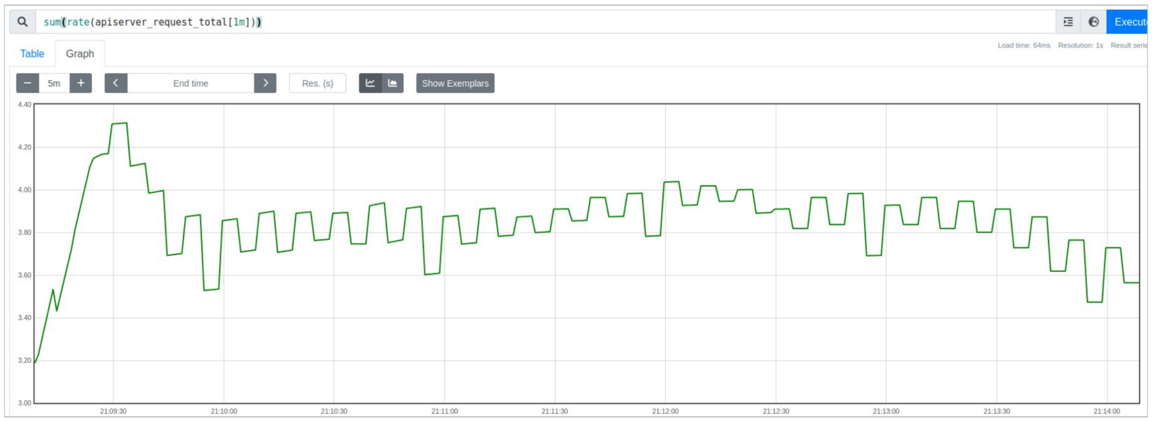
```
sum by(namespace) (container_memory_usage_bytes)
```

Your output may vary depending on the namespace you have created in the cluster.



__12. Change the query to the following and click **Execute** button to view the request count:

```
sum(rate(apiserver_request_total[1m]))
```



__13. Close the browser.

Part 9 - Clean-Up

__1. Delete the kube metrics service by executing the following command:

```
kubectl delete -f kube-state-metrics-configs
```

__2. Delete deployment, service, and cluster binding for Prometheus:

```
cd ..
kubectl delete -f monitoring
```

__3. Close the terminal window.

Part 10 - Review

In this lab, you used Prometheus to monitor a Kubernetes cluster.

Lab 8 - Using Jaeger for Tracing

In this lab, you will implement instrumentation using Jaeger. You will modify an existing Spring Boot application to add tracing to it.

Jaeger is used for monitoring and troubleshooting microservices-based distributed systems, including:

- Distributed context propagation
- Distributed transaction monitoring
- Root cause analysis
- Service dependency analysis
- Performance / latency optimization

Your applications must be instrumented before they can send tracing data to Jaeger backend. All Jaeger client libraries support the OpenTracing APIs. Review Jaeger java client libraries about how to use the OpenTracing API and how to initialize and configure Jaeger tracers.

https://github.com/jaegertracing/jaeger-client-java

Jaeger binaries are available for macOS, Linux, and Windows. Jaeger components can be downloaded in two ways: 1) Executable binaries, 2) Docket images.

https://www.jaegertracing.io/download/

The simplest way to start the all-in-one is to use the pre-built image published to DockerHub (a single command line). All-in-one is an executable designed for quick local testing, launches the Jaeger UI, collector, query, and agent, with an in-memory storage component.

You can navigate to http://localhost:16686 to access the Jaeger UI.

Part 1 - Setting the Stage

In this Lab you will work in the 'Project-VM-4.0-2023-02-07'.

Start or connect to this VM if you don't have it opened yet. Use wasadmin for user and password.

Make sure you ran the following command in a previous lab:

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

In this part, you will create a directory and copy an existing project to it.

__1. Open a new Terminal window.

__2. Create a directory where you will copy the Spring Boot application incase it doesn't exist:

```
mkdir -p ~/LabWorks
```

__3. Switch to the directory:

```
cd ~/LabWorks
```

__4. Unzip the existing application:

```
unzip ~/LabFiles/employees-rest-service
```

__5. Switch to the project directory:

```
cd employees-rest-service
```

__6. Open a Firefox browser and download the **build.gradle** file into the **Downloads** folder using the link below:

```
https://webage-downloads.s3.amazonaws.com/LabFiles/build.gradle
```

__7. Replace the **~/LabWorks/employees-rest-service/build.gradle** with the **Downloads/build.gradle** file that was just downloaded.

__8. Open **build.gradle** with gedit:

```
gedit build.gradle
```

__9. Make sure the content looks like below:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'org.springframework.boot', name: 'spring-
boot-gradle-plugin', version: '2.1.8.RELEASE'
    }
}

apply plugin: 'java'
apply plugin: 'maven-publish'
apply plugin: 'org.springframework.boot'

group = 'com.webage.demo'
version = '0.0.1-SNAPSHOT'

description = """SpringBootDemo"""

sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
     mavenCentral()
}
dependencies {
    implementation group: 'org.springframework.boot', name: 'spring-
boot-starter-web', version:'2.1.8.RELEASE'
    testImplementation group: 'org.springframework.boot', name:
'spring-boot-starter-test', version:'2.1.8.RELEASE'
}
```

__10. Close the file.

Part 2 - Explore an Existing Spring Boot Application

In this part, you will explore an existing Spring Boot application.

__1. Verify Gradle version:

```
gradle --version
```

__2. Jenkins should be running, so let's stop it:

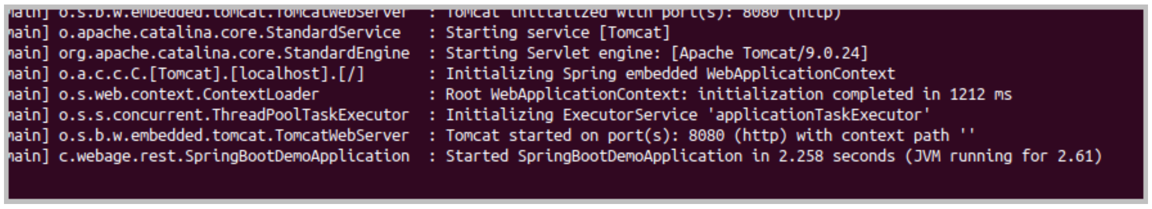
```
systemctl stop jenkins
```

__3. Enter **wasadmin** as password and click **Authenticate**.

__4. Build and run the sample project:

```
gradle clean build bootRun
```

You should see:



```
ain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
ain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
ain] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.24]
ain] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
ain] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 1212 ms
ain] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
ain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
ain] c.webage.rest.SpringBootDemoApplication  : Started SpringBootDemoApplication in 2.258 seconds (JVM running for 2.61)
```

__5. Leave the app running and open the Firefox web browser.

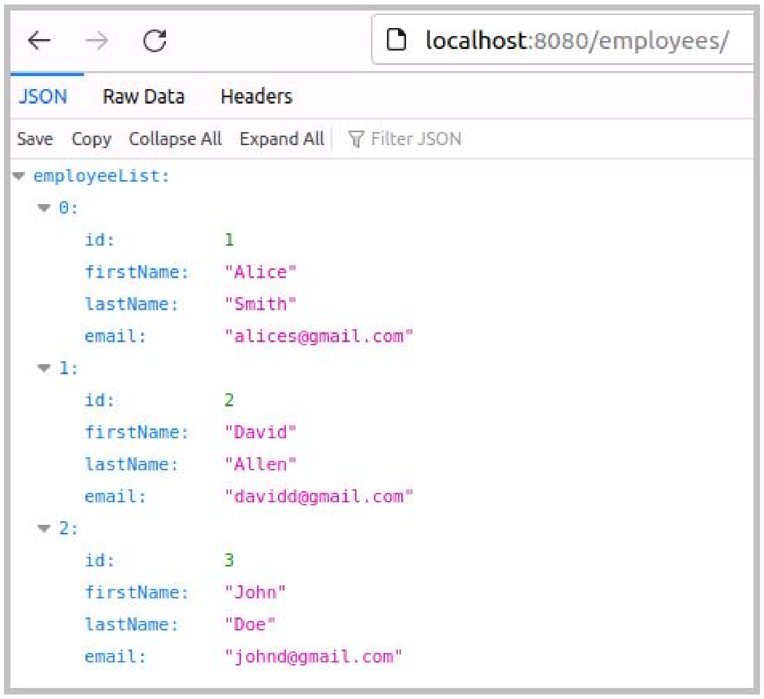
__6. Navigate to the following URL:

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

Note: Don’t forget to type in the / at the end.

The URL accesses Customers services implemented in the sample application. The service returns data in JSON format and it looks like this:

You should see:



```
employeeList:
  0:
    id:         1
    firstName:  "Alice"
    lastName:   "Smith"
    email:      "alices@gmail.com"
  1:
    id:         2
    firstName:  "David"
    lastName:   "Allen"
    email:      "davidd@gmail.com"
  2:
    id:         3
    firstName:  "John"
    lastName:   "Doe"
    email:      "johnd@gmail.com"
```

__7. In the terminal where Spring Boot application is running, press Ctrl+C to stop the application.

__8. Using gedit or vi/nano text editors, **examine** the code in the following files:

```
src/main/java/com/webage/rest/model/Employee.java
src/main/java/com/webage/rest/model/Employees.java
src/main/java/com/webage/rest/dao/EmployeeDAO.java
src/main/java/com/webage/rest/controller/EmployeeController.java
```

Note:
model folder contains the models (Employee and Employees)
dao folder contains Data Access Object which populates the data.
controller folder contains the REST service(s)

Part 3 - Add Jaeger Dependency to the Project and Configure It

In this part, you will modify build.gradle and add Jaeger dependency to the project and configure a bean in the main application file which returns a tracing instance.

__1. Open the project's main application file:

```
gedit src/main/java/com/webage/rest/SpringBootDemoApplication.java
```

__2. After the existing import statements, add the following imports:

```
import io.jaegertracing.Configuration;
import io.jaegertracing.internal.JaegerTracer;
import org.springframework.context.annotation.Bean;
```

__3. Inside SpringBootDemoApplication class, after the main function, add the following code:

```
@Bean
public static JaegerTracer getTracer() {

Configuration.SamplerConfiguration samplerConfig =
Configuration.SamplerConfiguration.fromEnv().withType("const").withParam(1);

Configuration.ReporterConfiguration reporterConfig =
Configuration.ReporterConfiguration.fromEnv().withLogSpans(true);

Configuration config = new Configuration("jaeger
tutorial").withSampler(samplerConfig).withReporter(reporterConfig);

return config.getTracer();
}
```

Note: These lines configure a bean which returns a JaegerTracing instance which will be used for tracing purpose. Also note, your application will traces will show up as "Jaeger tutorial" in Jaeger. You can customize this name to whatever application name you want.

__4. Save the file and exit gedit.

__5. Open **build.gradle**:

```
gedit build.gradle
```

__6. In the dependencies tag, add a new dependency:

```
implementation group: 'io.jaegertracing', name: 'jaeger-client',
version: '0.35.5'
```

```
[Enter this command in one line]
```

__7. Save and close the file.

__8. Run the following command to build your application:

```
gradle clean build
```

__9. Ensure there are no errors. If there any, resolve them before proceeding to the next part of this lab.

Part 4 - Modify the Existing REST Service

In this part, you will modify the existing REST service/controller and write code to trace messages.

__1. Open the REST service/controller file:

```
gedit src/main/java/com/webage/rest/controller/EmployeeController.java
```

__2. After the existing import statements, add the following imports:

```
import io.opentracing.Span;
import io.opentracing.Tracer;
```

__3. Inside EmployeeController class, add the following as the first statement:

```
@Autowired
private Tracer tracer;
```

__4. Inside **getEmployees** method, delete the existing return statement, and add the following code:

```
Span span = tracer.buildSpan("get employees").start();
span.setTag("http.status_code", 201);
Employees data = employeeDao.getAllEmployees();
span.finish();
return data;
```

Note: Your getEmployees method should look like this:

```
    public Employees getEmployees()
    {
        Span span = tracer.buildSpan("get employees").start();
        span.setTag("http.status_code", 201);
        Employees data = employeeDao.getAllEmployees();
        span.finish();
        return data;
    }
```

Note: These lines use the JaegerTracing object, which you configured in the main application file, to obtain a Span object. Span allows you to start the trace, optionally add some additional data in the form tags, and stop the trace.

__5. Save the file and close gedit.

__6. Run the following command to build your application:

```
gradle clean build
```

__7. Ensure there are no errors. If there any, resolve them before proceeding to the next part of this lab.

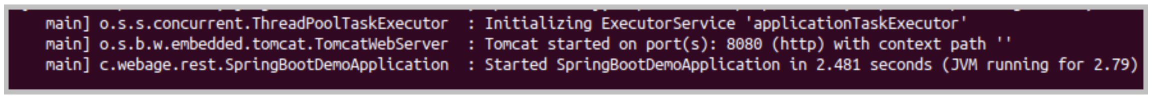
Part 5 - Verify Tracing in Jaeger

In this part, you will run your application, start Jaeger, and verify tracing is working properly.

__1. Run the following command to start up your Spring Boot application:

```
gradle bootRun
```

You should see:



```
main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
main] c.webage.rest.SpringBootDemoApplication  : Started SpringBootDemoApplication in 2.481 seconds (JVM running for 2.79)
```

__2. Open a web browser and navigate to the following URL:

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

Note: It's the same data you saw previously. Next, you will start Jaeger and view the traces.

__3. Open a new Terminal window.

__4. Run the following command to run Jaeger in a Docker container:

```
sudo docker run -d --name Jaeger --rm -it --network=host
jaegertracing/all-in-one:1.12.0
```

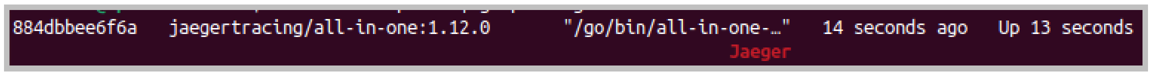
```
[Enter this command in one line]
```

__5. Enter **wasadmin** as password if prompt.

__6. Ensure container is up and running:

```
sudo docker ps -a | grep "Jaeger"
```

You should see:



```
884dbbee6f6a    jaegertracing/all-in-one:1.12.0       "/go/bin/all-in-one-…"   14 seconds ago    Up 13 seconds
                                                                     Jaeger
```

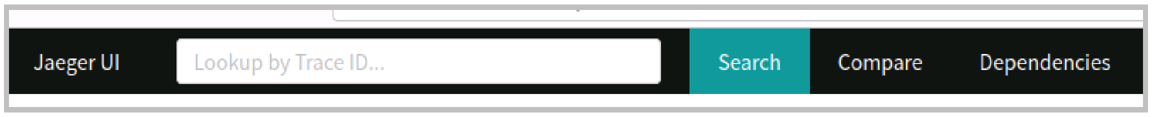
__7. Refresh the following page:

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

__8. In the web browser window, open a new tab, and navigate to the following URL:

`http://localhost:16686`

You should see:



__9. Refresh the page in the tab where **http://localhost:8080/employees/** page is open.

__10. On the **Jaeger** web console, refresh the page.

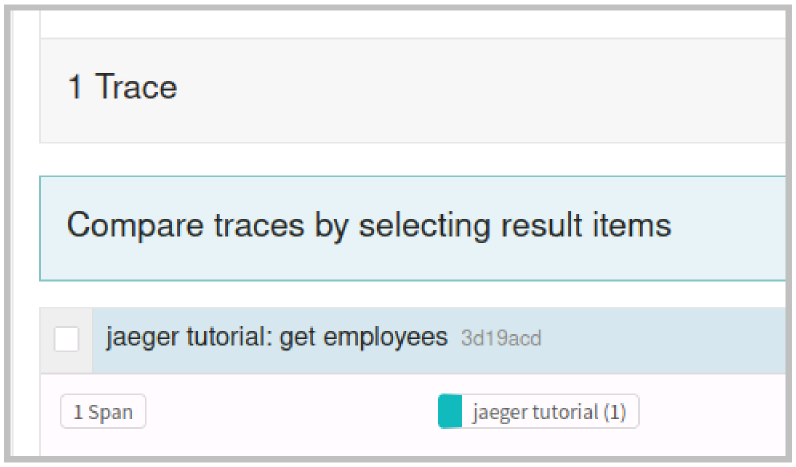
__11. Click **Service** drop-down and select **Jaeger tutorial**.

Note: If Service drop-down is blank, refresh the page in the tab where http://localhost:8080/employees/ page is open, and then refresh Jaeger web console. You might have to wait for a minute before it shows up in the drop down.

Jaeger tutorial is your custom application. Jaeger query is available OOB as part of Jaeger. It allows you to make REST API calls to query Jaeger in case if you don't want to use the web console.

__12. Scroll down and click **Find Traces** button:

Notice the trace looks like this:


1 Trace

Compare traces by selecting result items

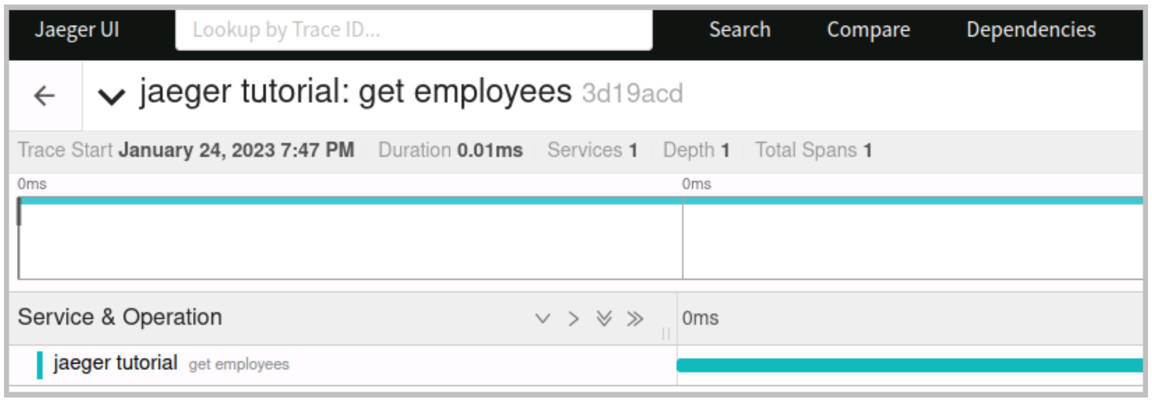
jaeger tutorial: get employees 3d19acd

1 Span

jaeger tutorial (1)

__13. Click the trace '**Jaeger tutorial**' to view details.

It should show up like this:

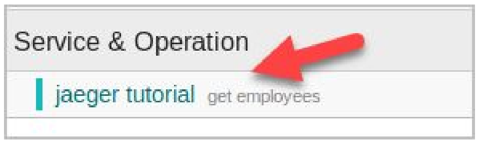


jaeger tutorial: get employees 3d19acd

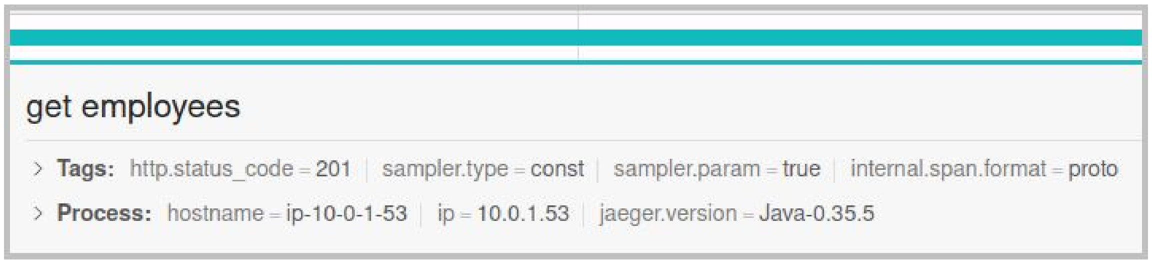
Trace Start **January 24, 2023 7:47 PM** Duration **0.01ms** Services **1** Depth **1** Total Spans **1**

jaeger tutorial get employees

__14. In **Service & Operation** section, click **Jaeger tutorial** again.



Notice it looks like this:



get employees

> Tags: http.status_code = 201 | sampler.type = const | sampler.param = true | internal.span.format = proto

> Process: hostname = ip-10-0-1-53 | ip = 10.0.1.53 | jaeger.version = Java-0.35.5

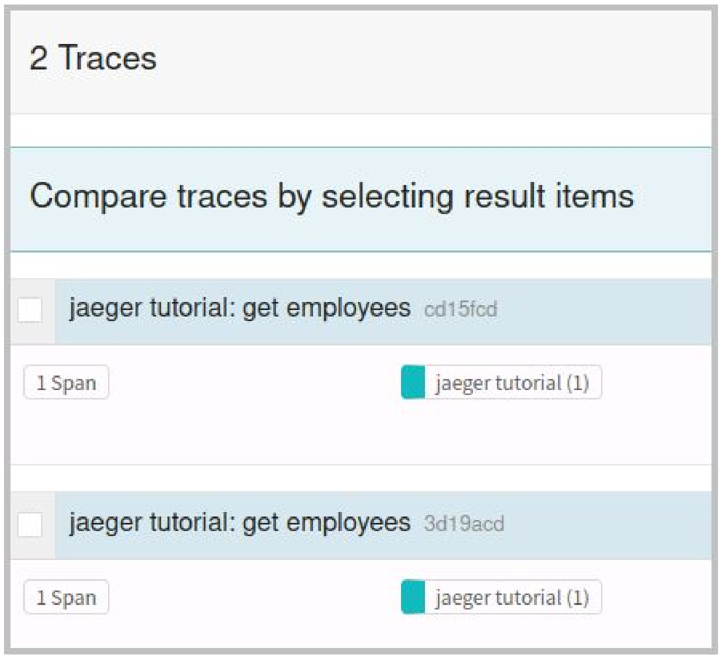
Notice Tags is showing your custom label http.status_code with value 201.

__15. Refresh the **http://localhost:8080/employees/** page.

__16. Go back to the Jaeger web console main page.

__17. Click **Find Traces**.

Notice the traces show up like this:



2 Traces

Compare traces by selecting result items

jaeger tutorial: get employees cd15fcd

1 Span | jaeger tutorial (1)

jaeger tutorial: get employees 3d19acd

1 Span | jaeger tutorial (1)

Note. Your values in the graph may be different.

Part 6 - Clean-Up

In this part, you will stop Prometheus and your application.

__1. Run the following command to stop Jaeger container:

```
sudo docker stop Jaeger
```

If prompted for a password, enter wasadmin

__2. Switch to the terminal where Spring Boot application is running and press Ctrl+C to stop the application.

__3. Close the Terminal windows.

__4. Close the web browser.

Part 7 - Review

In this lab, you implemented instrumentation using Jaeger.

Lab 9 - Getting Started with Istio

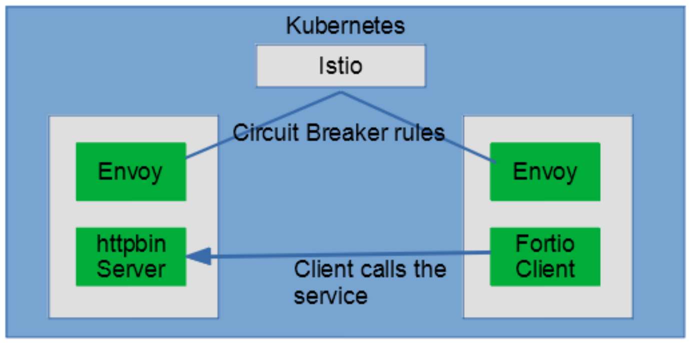
In this lab, you will set up Istio, install a sample app, configure circuit breaker rules, and test the rules by running a client application.

Istio is an open-source service mesh. A service mesh offers various features, such as service discovery, load balancing, and circuit breaking. In this lab, you will run Istio on Kubernetes and explore Istio's circuit breaker feature.

You will deploy a sample httpbin app and Fortio client. To run the sample with Istio requires no changes to the application itself. Instead, you simply need to configure and run the service in an Istio-enabled environment, with Envoy sidecars injected alongside each service.

You add Istio support to services by deploying a special sidecar proxy, also called Envoy, throughout your environment that intercepts all network communication between microservices. **Envoy** is a high-performance proxy developed in C++ to mediate all inbound and outbound traffic for all services in the service mesh. **Istio** leverages **Envoy's** many built-in features, for example, Istio provides automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic.

The end-to-end architecture of the application looks like this:



Part 1 - Setting the Stage

In this Lab you will continue working in the 'VM_WA2894_REL_1_1'.

Start or connect to this VM if you don't have it opened yet. Use wasadmin for user and password.

Make sure you ran the following command in a previous lab:

sudo docker login -u {your-docker-id} -p {your-access-token}

Note: make sure the VM has at least 8 GB RAM to run this lab

__1. Open a new Terminal window.

__2. Execute the following command to switch to the root user:

```
sudo -i
```

Enter **wasadmin** for password.

__3. Enter the following command:

```
whoami
```

You should see that you are **root** now.

```
root
```

Part 2 - Start-Up Kubernetes

In this part, you will start up Kubernetes. The lab environment is set up with MiniKube.

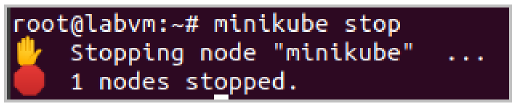
__1. Check if minikube is running:

```
minikube status
```

__2. If minikube is 'Running', stop it with this command:

```
minikube stop
```

Wait for it to output "minikube stopped" before moving on. It may take a while.



```
root@labvm:~# minikube stop
✋  Stopping node "minikube"  ...
🛑  1 nodes stopped.
```

__3. Delete the old cluster by running the following command:

```
minikube delete
```

__4. Start minikube with a new cluster using this command:

```
minikube start --driver=none
```

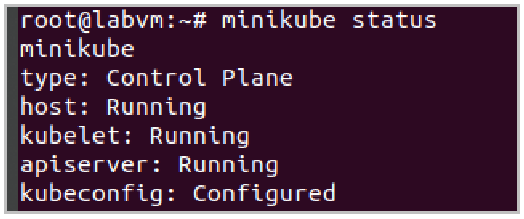
Wait for it to output the following line before moving on:

```
Done! kubectl is now configured to use "minikube"
```

__5. Check minikube status again:

```
minikube status
```

It should indicate that minikube is running.



```
root@labvm:~# minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Part 3 - Download Istio

In this part, you will download Istio.

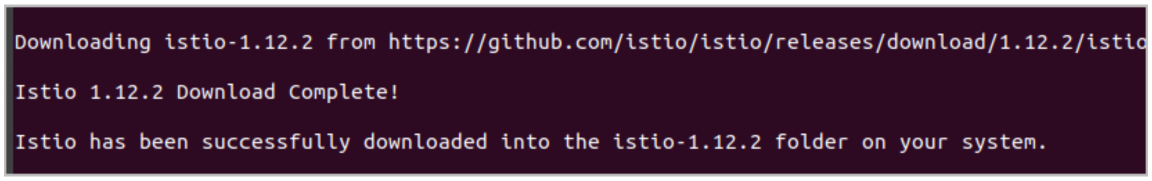
__1. Switch to the directory where Istio will be downloaded and configured:

```
cd /home/wasadmin/Works
```

__2. Run the following command to download the latest version of Istio:

```
curl -L https://istio.io/downloadIstio | sh -
```

Make a note of the Istio version that is downloaded on your machine. For example, at the time of writing of the lab, it's 1.12.2.



```
Downloading istio-1.12.2 from https://github.com/istio/istio/releases/download/1.12.2/istio

Istio 1.12.2 Download Complete!

Istio has been successfully downloaded into the istio-1.12.2 folder on your system.
```

__3. Switch to the directory where Istio is available:

```
cd istio-VERSION
```

Replace VERSION with the version you noted in the previous step. For example **cd istio-1.12.2**

__4. Add Istio to the path:

```
export PATH=$PWD/bin:$PATH
```

Part 4 - Setup and Verify Istio Installation

In this part, you will install and configure Istio. Although you can install and configure everything from scratch, to simplify the process, you can use various configuration profiles. For this lab, you will use the **demo** configuration profile.

Here is a list of the built-in configuration profiles:

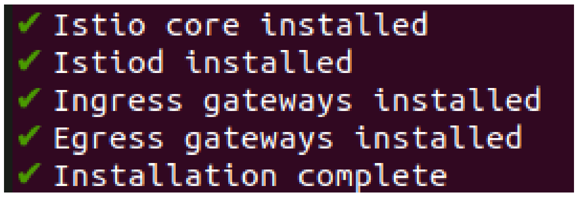
1. **default**: This profile is recommended for production deployments.
2. **demo**: This profile has modest resource requirements.
3. **minimal**: This can be useful as a base profile for custom configuration.
4. **preview**: This profile contains experimental features.

__1. Install and setup Istio with the demo configuration profile:

```
istioctl install --set profile=demo
```

__2. Enter y when prompt.

Ensure it shows you output like this:



```
✓ Istio core installed
✓ Istiod installed
✓ Ingress gateways installed
✓ Egress gateways installed
✓ Installation complete
```

Note. Make sure the VM has at least 8 GB on RAM or Istio won't be installed.

__3. Run the following command to configure Istio to automatically inject Envoy sidecar proxies when you deploy your applications:

```
kubectl label namespace default istio-injection=enabled
```

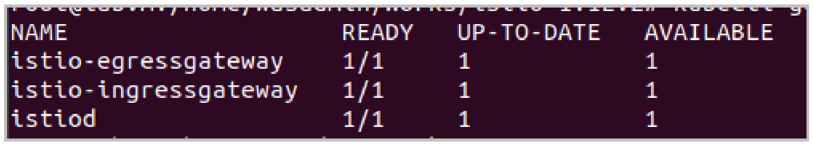
Ensure it shows the following output:

namespace/default labeled

__4. Run the following command to verify Istio deployments:

```
kubectl get deployments -n istio-system
```

Verify it shows the following result: (Note: Ensure the READY status is 1/1).

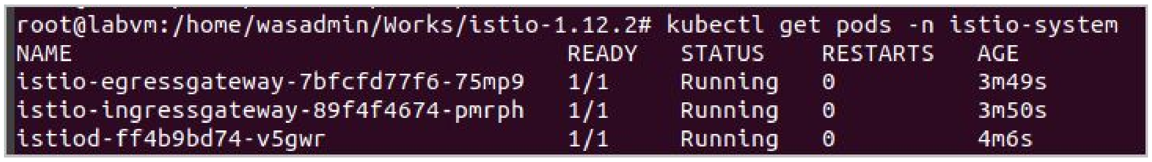


```
NAME                    READY   UP-TO-DATE   AVAILABLE
istio-egressgateway     1/1     1            1
istio-ingressgateway    1/1     1            1
istiod                  1/1     1            1
```

__5. Run the following command to verify all Istio pods are running:

```
kubectl get pods -n istio-system
```

Verify it shows the following result: (Note: Ensure the READY status is 1/1)

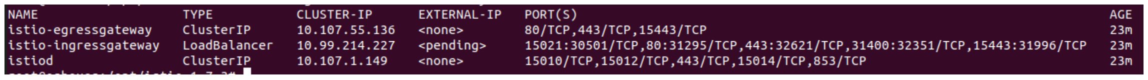


```
root@labvm:/home/wasadmin/Works/istio-1.12.2# kubectl get pods -n istio-system
NAME                                    READY   STATUS    RESTARTS   AGE
istio-egressgateway-7bfcfd77f6-75mp9    1/1     Running   0          3m49s
istio-ingressgateway-89f4f4674-pmrph    1/1     Running   0          3m50s
istiod-ff4b9bd74-v5gwr                  1/1     Running   0          4m6s
```

__6. Run the following command to verify all Istio services are running:

```
kubectl get services -n istio-system
```

Verify it shows the following result:



```
NAME                   TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)                                                                      AGE
istio-egressgateway    ClusterIP      10.107.55.136   <none>        80/TCP,443/TCP,15443/TCP                                                     23m
istio-ingressgateway   LoadBalancer   10.99.214.227   <pending>     15021:30501/TCP,80:31295/TCP,443:32621/TCP,31400:32351/TCP,15443:31996/TCP   23m
istiod                 ClusterIP      10.107.1.149    <none>        15010/TCP,15012/TCP,443/TCP,15014/TCP,853/TCP                                23m
```

In the next section of this lab, you will test Istio's circuit breaker capabilities. Circuit breaking is an important pattern for creating resilient microservice applications. Circuit breaking allows you to write applications that limit the impact of failures, latency spikes, and other undesirable effects of network peculiarities.

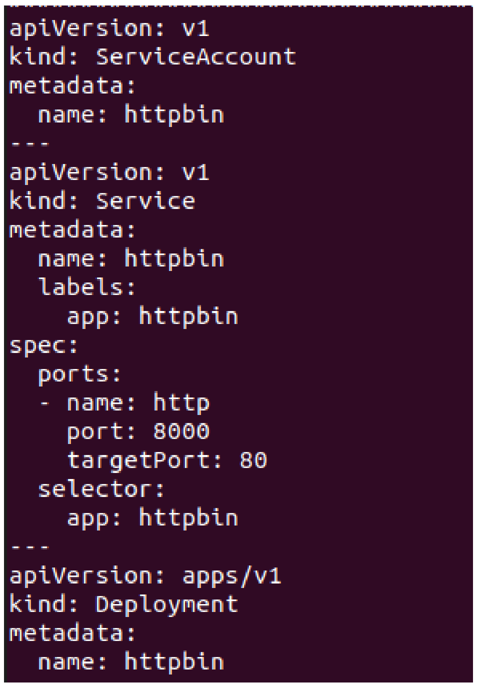
Part 5 - Setup a Sample Backend Application

In this part, you will deploy and start a sample application, httpbin, that is a well known simple HTTP request and response service available at http://httpbin.org. httpbin will act as the backend service and you will later access from a client app and test Istio's circuit breaker capabilities.

__1. View httpbin.yaml:

```
cat samples/httpbin/httpbin.yaml
```

The contents of the file look like this (only part is shown):



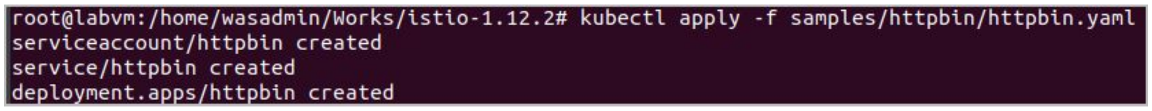
```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: httpbin
---
apiVersion: v1
kind: Service
metadata:
  name: httpbin
  labels:
    app: httpbin
spec:
  ports:
  - name: http
    port: 8000
    targetPort: 80
  selector:
    app: httpbin
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpbin
```

If the docker image for httpbin application isn't available, it downloads it from docker.io/kennerthreitz/httpbin. In the container, you will have httpbin service listening at port 8000.

__2. Deploy the httpbin sample backend service:

```
kubectl apply -f samples/httpbin/httpbin.yaml
```

You should see:



```
root@labvm:/home/wasadmin/Works/istio-1.12.2# kubectl apply -f samples/httpbin/httpbin.yaml
serviceaccount/httpbin created
service/httpbin created
deployment.apps/httpbin created
```

Note: In case if you have haven't enabled automatic sidecar injection, you can run the following command to manually inject sidecar.

kubectl apply -f <(istioctl kube-inject -f samples/httpbin/sample-client/fortio-deploy.yaml)

This command is equivalent to:

istioctl kube-inject -f samples/httpbin/httpbin.yaml | kubectl apply -f -

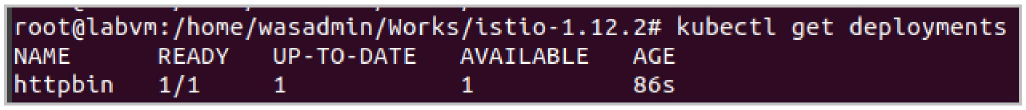
Kube apply would usually create one pod per. But, since you are injecting Istio sidecar, it will end up creating 2 pods.

istioctl kube-inject generates a new yaml file where your Istio proxy is injected into your application yaml.

__3. Ensure the application is deployed:

```
kubectl get deployments
```

Verify it shows the following result:



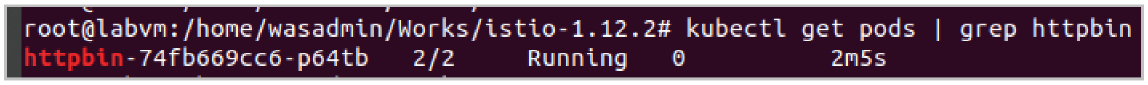
```
root@labvm:/home/wasadmin/Works/istio-1.12.2# kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
httpbin   1/1     1            1           86s
```

If you see 0/1, try again. It may take some minutes to show 1/1.

__4. Verify a pod for the service is running:

```
kubectl get pods | grep httpbin
```

Ensure it shows the following:



```
root@labvm:/home/wasadmin/Works/istio-1.12.2# kubectl get pods | grep httpbin
httpbin-74fb669cc6-p64tb   2/2     Running   0          2m5s
```

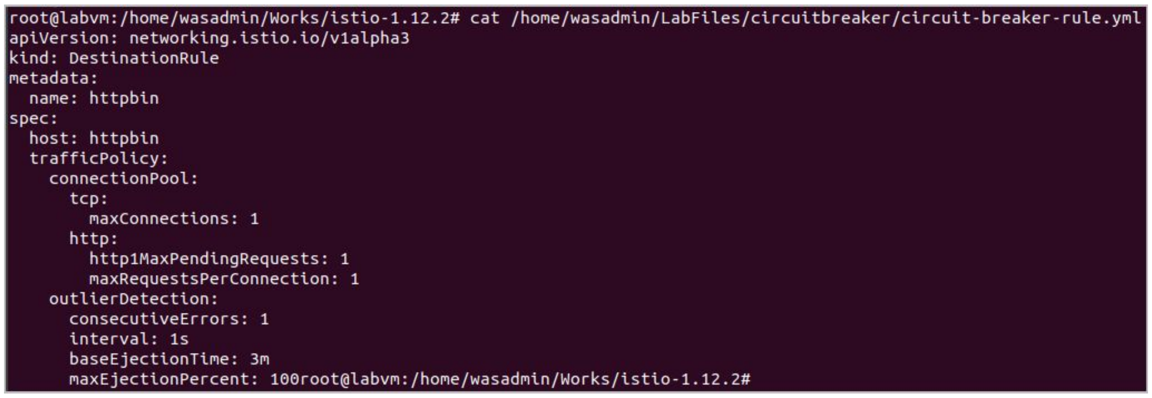
Part 6 - Configure the Circuit Breaker

In this part, you will configure the circuit breaker so that when you access the backend service, httpbin, from a client application, it should follow the circuit breaker rules.

__1. Inspect the contents of a yaml file which specifies circuit breaker rule:

```
cat /home/wasadmin/LabFiles/circuitbreaker/circuit-breaker-rule.yml
```

You should see:



```
root@labvm:/home/wasadmin/Works/istio-1.12.2# cat /home/wasadmin/LabFiles/circuitbreaker/circuit-breaker-rule.yml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: httpbin
spec:
  host: httpbin
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      consecutiveErrors: 1
      interval: 1s
      baseEjectionTime: 3m
      maxEjectionPercent: 100root@labvm:/home/wasadmin/Works/istio-1.12.2#
```

__2. Create a destination rule to apply circuit breaking settings when calling the `httpbin` service:

```
kubectl apply -f /home/wasadmin/LabFiles/circuitbreaker/circuit-
breaker-rule.yml
```

[This command must be entered in one line]

Ensure it shows the message:

destinationrule.networking.istio.io/httpbin created

__3. Verify the destination rule was created correctly:

```
kubectl get destinationrule httpbin -o yaml
```

Verify it shows the following output:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
...
spec:
  host: httpbin
  trafficPolicy:
    connectionPool:
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
      tcp:
        maxConnections: 1
    outlierDetection:
      baseEjectionTime: 3m
      consecutiveErrors: 1
      interval: 1s
      maxEjectionPercent: 100
```

In the circuilt-breaker-rule.yml file's `DestinationRule` settings, you have `maxConnections: 1` and `http1MaxPendingRequests: 1`. These rules indicate that if you exceed more than one connection and request concurrently, you should see some failures when the `istio-proxy` opens the circuit for further requests and connections.

Part 7 - Setup a Client Application

In this part, you will use Fortio as a client application. Fortio started as Istio's load testing tool and later was turned into an open-source project. Fortio comes from Greek and means load/burden. With Fortio, you can control the number of connections, concurrency, and delays for outgoing HTTP calls. You will use this tool to trip the circuit breaker policies you set in DestinationRule in the previous part of this lab.

You will use Fortio to send traffic to the httbin backend service.

__1. View the manifest you will use to deploy the Fortio client:

```
cat samples/httpbin/sample-client/fortio-deploy.yaml
```

The yaml file contains deployment and service configuration for the Fortio client.

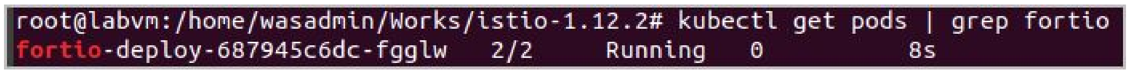
__2. Run the following command to inject the client with the Istio sidecar proxy so network interactions are governed by Istio:

```
kubectl apply -f samples/httpbin/sample-client/fortio-deploy.yaml
```

__3. Verify the pods are created:

```
kubectl get pods | grep fortio
```

Wait until the status is Running 2/2 before you proceed to the next step.



```
root@labvm:/home/wasadmin/Works/istio-1.12.2# kubectl get pods | grep fortio
fortio-deploy-687945c6dc-fgglw   2/2     Running   0          8s
```

Before you test the circuit breaker functionality, you will verify you can connect to the backend (httpbin) pod from the client (Fortio) pod.

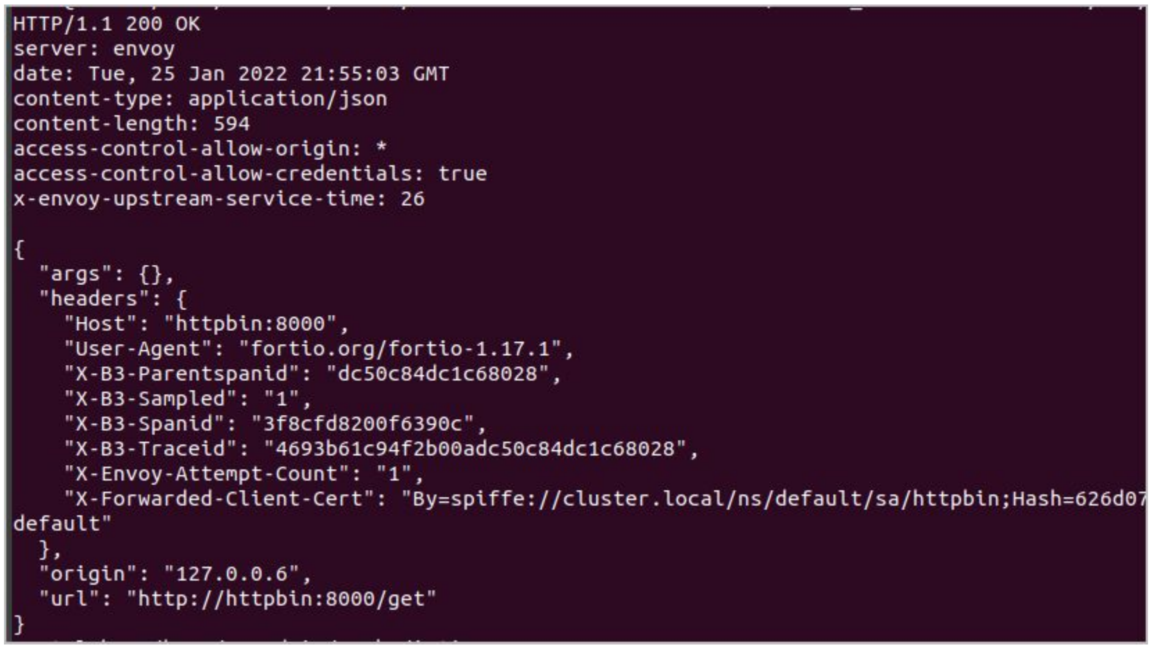
__4. Run the following command to obtain the Fortio pod name: (Note: Ensure you run it as a single command.)

```
export FORTIO_POD=$(kubectl get pods -lapp=fortio -o
'jsonpath={.items[0].metadata.name}')
```

__5. Run the following command to use curl from the client (Fortio) pod to connect to the backend (httpbin) pod: (Note: Ensure you run it as a single command.)

```
kubectl exec "$FORTIO_POD" -c fortio -- /usr/bin/fortio curl -quiet
http://httpbin:8000/get
```

Ensure it shows the output is similar than this:



```
HTTP/1.1 200 OK
server: envoy
date: Tue, 25 Jan 2022 21:55:03 GMT
content-type: application/json
content-length: 594
access-control-allow-origin: *
access-control-allow-credentials: true
x-envoy-upstream-service-time: 26

{
  "args": {},
  "headers": {
    "Host": "httpbin:8000",
    "User-Agent": "fortio.org/fortio-1.17.1",
    "X-B3-Parentspanid": "dc50c84dc1c68028",
    "X-B3-Sampled": "1",
    "X-B3-Spanid": "3f8cfd8200f6390c",
    "X-B3-Traceid": "4693b61c94f2b00adc50c84dc1c68028",
    "X-Envoy-Attempt-Count": "1",
    "X-Forwarded-Client-Cert": "By=spiffe://cluster.local/ns/default/sa/httpbin;Hash=626d07
default"
  },
  "origin": "127.0.0.6",
  "url": "http://httpbin:8000/get"
}
```

Part 8 - Tripping the Circuit Breaker

In this part, you will the circuit breaker functionality. If you recall, the custom circuilt-breaker-rule.yml file's `DestinationRule` settings specified `maxConnections: 1` and `http1MaxPendingRequests: 1`. These rules indicate that if you exceed more than one connection and request concurrently, you should see some failures when the `istio-proxy` opens the circuit for further requests and connections.

__1. Call the service with one connection (`-c 1`) and send 20 requests (`-n 20`):

```
kubectl exec -it $FORTIO_POD  -c fortio -- /usr/bin/fortio load -c 1 -
qps 0 -n 20 -loglevel Warning http://httpbin:8000/get
```

`[This command must be entered in one line]`

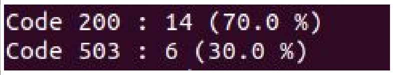
Notice it shows Code 200 : 20 (100.0 %), which means all calls were successful since the circuit breaker rules didn't kick in.

__2. Call the service with two concurrent connections (`-c 2`) and send 20 requests (`-n 20`):

```
kubectl exec -it $FORTIO_POD  -c fortio -- /usr/bin/fortio load -c 2 -
qps 0 -n 20 -loglevel Warning http://httpbin:8000/get
```

```
[This command must be entered in one line]
```

The output on your side might be slightly different:



```
Code 200 : 14 (70.0 %)
Code 503 : 6 (30.0 %)
```

Code 200 represents the successful calls and 503 represents the server side failures due to concurrency rules which you defined earlier in the lab.

It’s interesting to see that almost all requests, 90%, made it through! The `istio-proxy` does allow for some leeway.

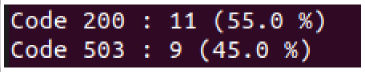
You may need to run this command twice.

__3. Bring the number of concurrent connections up to 3:

```
kubectl exec -it $FORTIO_POD  -c fortio -- /usr/bin/fortio load -c 3 -
qps 0 -n 20 -loglevel Warning http://httpbin:8000/get
```

```
[This command must be entered in one line]
```

Note: The output on your side may differ.



```
Code 200 : 11 (55.0 %)
Code 503 : 9 (45.0 %)
```

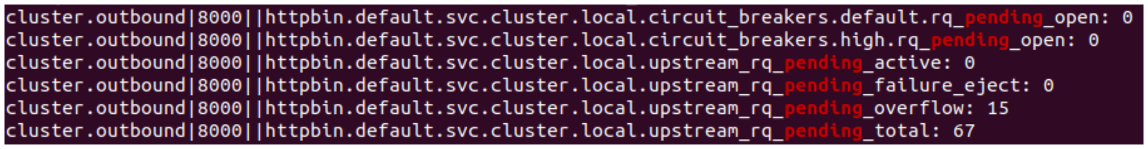
Now you start to see the expected circuit breaking behavior. Only 65% of the requests succeeded and the rest were trapped by circuit breaking. Your % may vary.

__4. Query the `istio-proxy` stats to see more:

```
kubectl exec -it $FORTIO_POD  -c istio-proxy  -- sh -c 'curl
localhost:15000/stats' | grep httpbin | grep pending
```

```
[This command must be entered in one line]
```

Depending on how many times you have executed the test, your numbers will vary. In the sample picture, you can see `24` for the `upstream_rq_pending_overflow` value which means `24` calls so far have been flagged for circuit breaking.



```
cluster.outbound|8000||httpbin.default.svc.cluster.local.circuit_breakers.default.rq_pending_open: 0
cluster.outbound|8000||httpbin.default.svc.cluster.local.circuit_breakers.high.rq_pending_open: 0
cluster.outbound|8000||httpbin.default.svc.cluster.local.upstream_rq_pending_active: 0
cluster.outbound|8000||httpbin.default.svc.cluster.local.upstream_rq_pending_failure_eject: 0
cluster.outbound|8000||httpbin.default.svc.cluster.local.upstream_rq_pending_overflow: 15
cluster.outbound|8000||httpbin.default.svc.cluster.local.upstream_rq_pending_total: 67
```

Part 9 - Clean-Up

__1. Remove the rules:

```
kubectl delete destinationrule httpbin
```

__2. Shutdown the **httpbin** service and client:

```
kubectl delete deploy httpbin fortio-deploy
```

```
kubectl delete svc httpbin
```

__3. Type **exit** until the Terminal is closed.

Part 10 - Review

In this lab, you set up Istio, set up circuit breaker rules for a sample application, and tested the rules.