WA2919 Booz Allen Hamilton Tech Excellence Cloud Engineering Program - Phase 3

Student Labs

Web Age Solutions Inc.

Table of Contents

Environment

This course requires the following Virtual Machine (VM), version can me greater but not smaller:

- **`VM_WA2919_REL_1_0`**

You should find a shortcut in the desktop to run the VM or you may already connected directly in the VM.

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

IMPORTANT NOTE!!! **If you have been provided with a Docker account, skip to next part**.

In this part, you will create a Docker account.

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

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

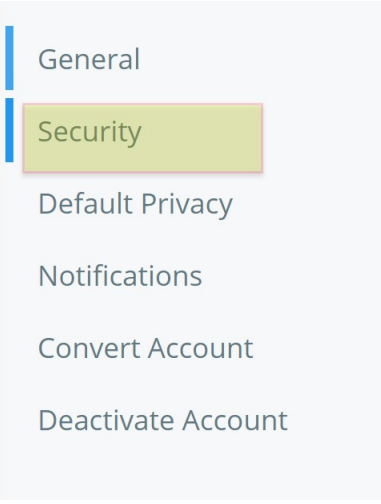
__2. Click **Sign Up** and create a new account.

__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**.



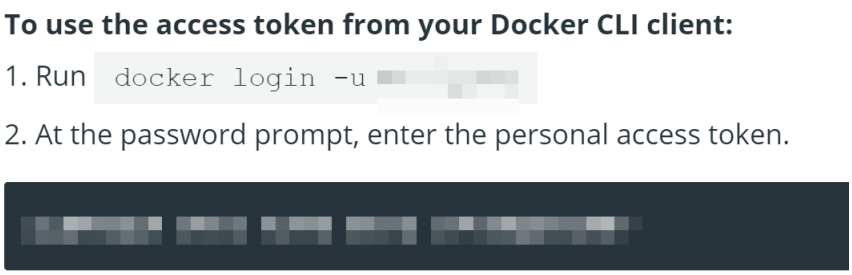
__6. Click **New Access Token**.

__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:



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_WA2919_REL_1_0**' 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_WA2919_REL_1_0'.

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. Enter the following command:

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

__3. 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.

The commands list is shown below for you reference.

```
  attach      Attach local standard input, output, and error streams to a
  build       Build an image from a Dockerfile
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's
  events      Get real time events from the server
  exec        Run a command in a running container
  export      Export a container's filesystem as a tar archive
  history     Show the history of an image
  images      List images
  import      Import the contents from a tarball to create a filesystem image
  info        Display system-wide information
  inspect     Return low-level information on Docker objects
  kill        Kill one or more running containers
  load        Load an image from a tar archive or STDIN
  login       Log in to a Docker registry
  logout      Log out from a Docker registry
  logs        Fetch the logs of a container
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  ps          List containers
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  rmi         Remove one or more images
  run         Run a command in a new container
  save        Save one or more images to a tar archive (streamed to STDOUT by
  search      Search the Docker Hub for images
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
  top         Display the running processes of a container
  unpause     Unpause all processes within one or more containers
  update      Update configuration of one or more containers
  version     Show the Docker version information
  wait        Block until one or more containers stop, then print their exit
```

__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 the **~/Works** directory:

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

__3. 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
```

__4. To see a list of all containers, whether active or inactive, run the following command:

```
docker ps -a
```

Make a note of the ubuntu container.

__5. Switch to the original terminal window (**T1**).

__6. Enter the following command and replace the container id with the **ubuntu** container from T2:

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

Note: You may only need to type the first two or three characters of the container id and then press the **Tab** key - Docker will go ahead and auto-complete it. The container id shown to the user is, actually, the first 12 bytes of a 64 hexadecimal character long hash used by Docker to create unique ids for images and containers.

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.

__7. Switch to the **T2** terminal.

__8. 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 ubutnu 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. Remove the *my_server:v1.0* image we created and persisted in our local image repository:

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

__5. 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:

__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
```

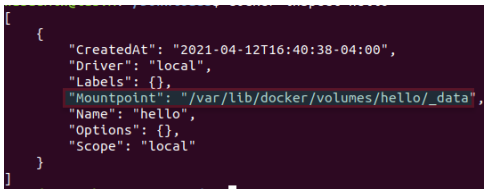
__11. View all volumes:

```
docker volume ls
```

__12. Check the mount point location.

```
docker inspect hello
```

Notice the mount point is displayed in the JSON output.



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. Delete the volume:

```
docker volume rm hello
```

__5. Get docker image list:

```
docker images
```

__6. Remove all images that you created by executing following command:

```
docker rmi <REPOSITORY:TAG>
```

e.g. **docker rmi dev-openjdk:v1.0**

__7. Make sure your image has been deleted:

```
docker images
```

__8. 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 - Getting Started with Docker Compose

To develop and operate a microservice application, it is not uncommon to require an environment to be brought up with many backing services. For instance, developing an online ordering system may require a backing database as well as a messaging system such as Kafka.

In this lab, we will explore the development of an order and invoicing system that requires a Kafka cluster, a Postgres backend database and three microservices written in Spring Boot.

Part 1 - Setting the Stage

In this Lab you will continue working in the 'VM_WA2919_REL_1_0'.

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. Enter the following command to switch to the workspace folder.

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

__3. Let's make sure nothing is running in Docker. Execute the following command and make sure no containers are running:

```
docker ps
```

__4. If a container is running, use the following command:

```
docker stop <containerid>
```

__5. Let's take a look at the images currently available on our system:

```
docker images
```

Part 2 - Exploring the project

__1. We are ready to get started We will clone an existing project using git:

```
git clone https://github.com/jfbilodeau/microservice-kafka
```

__2. Change directory into the project:

```
cd microservice-kafka
```

__3. Let's take a look at the project structure:

```
ls
```

__4. There are two directories: **docker** and **microservice-kafka**. Let's now take a look inside the microservice-kafka directory:

```
ls microservice-kafka
```

There are three projects: **microservice-kafka-invoicing, microservice-kafka-order** and **microservice-kafka-shipping**. Each one is a Spring Boot project and are designed to run in a container.

__5. Let's take a look at one of the Dockerfile:

```
cat microservice-kafka/microservice-kafka-order/Dockerfile
```

Notice how port 8080 is exposed. Feel free to take a look at the other two Dockerfile, or any of the code of the project if interested. They are very similar and they all expose port 8080. You might also notice that nowhere do the Dockerfile mention Kafka or Postgres.

__6. Let's now take a look in the docker directory:

```
ls docker
```

Notice the presence of the **docker-compose.yml** file.

__7. Before we concern ourselves with Docker Compose, let's take a look at the apache and postgres directories:

```
ls docker/apache

ls docker/postgres
```

__8. Let's also take a quick look at their Dockerfile:

```
cat docker/apache/Dockerfile

cat docker/postgres/Dockerfile
```

The Apache Dockerfile is especially of interest. In this project, Apache is strictly a front-end proxy for our three microservices. You might notice the use of mod_proxy.

__9. Let's take a look at the Apache proxy configuration:

```
cat docker/apache/000-default.conf
```

Of interest are the three ProxyPass and ProxyPassReverse instructions. Notice how Apache is expected to resolve three domain names, order, shipping and invoicing on port 8080.

__10. Let's now take a look at the docker-compose.yml file:

```
cat docker/docker-compose.yml
```

How many services are defined in the file?

Notice the links: configuration node. Take a moment and draw a dependency diagram of all the services.

Which service(s) should start first? Which service(s) should start last?

Part 3 - Building the microservices

We are ready to build our three microservices.

__1. Change into the microservice-kafka directory:

```
cd microservice-kafka
```

__2. Confirm that the compiled jar does not exist:

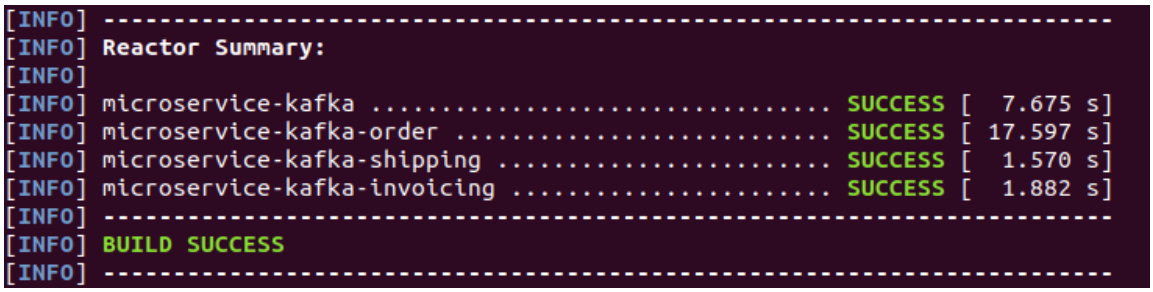
```
ls microservice-kafka-order/target
```

We should get an error indicating that the directory does not exist. Feel free to check the other two service project as well.

__3. Run the build script:

```
./mvnw clean package -Dmaven.test.skip=true
```

__4. The build will take a couple of minutes to run. Once completed, the target jar files are created.



__5. Confirm that the jar file was created as follows:

```
ls microservice-kafka-order/target
```

In the output, we should see a file named **microservice-kafka-order-0.0.1-SNAPSHOT.jar**. Feel free to verify the other two projects.

__6. Finally, let's exit the microservice-kafka folder:

```
cd ..
```

The project is now build and ready to rock.

Part 4 - Creating the images

A number of Docker images need to be created. Let's use Docker Compose to help us.

__1. Change directory into the docker directory:

```
cd docker
```

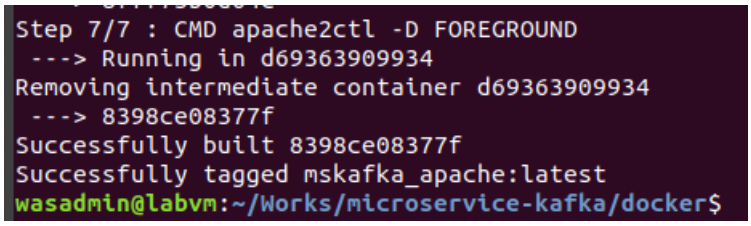
__2. Enable port forwarding to docker containers can connect to internet and download packages, enter **wasadmin** as password:

```
sudo /sbin/sysctl -w net.ipv4.conf.all.forwarding=1
```

__3. Build the application images:

```
docker-compose build
```

__4. This task will take a couple of minutes to run. You may see some errors but it should complete successfully.



__5. Let's make sure the images were created:

```
docker images
```

__6. We should see an output similar to the following:

```
REPOSITORY              TAG                 IMAGE ID...
mskafka_invoicing       latest              564ffae9f625...
mskafka_shipping        latest              aa0e02e3903b...
mskafka_order           latest              2624e818fc09...
mskafka_apache          latest              6c67a4b774a2...
mskafka_postgres        latest              aca66cac4816...
ubuntu                  16.04               2a697363a870...
openjdk                 11.0.2-jre-slim     b7a931ed7d37...
postgres                9.6.3               33b13ed6b80a...
...
```

We have created our images.

Part 5 - Running the application

Now that all the pieces are in place, let's bring up the application.

__1. We need to make sure port 8080 is not running, enter the following command to list process running on port 8080:

```
sudo lsof -i:8080
```

__2. Run the following command to kill process on port 8080 is running:

```
sudo kill $(sudo lsof -t -i:8080)
```

__3. Verify the that port 8080 is not used:

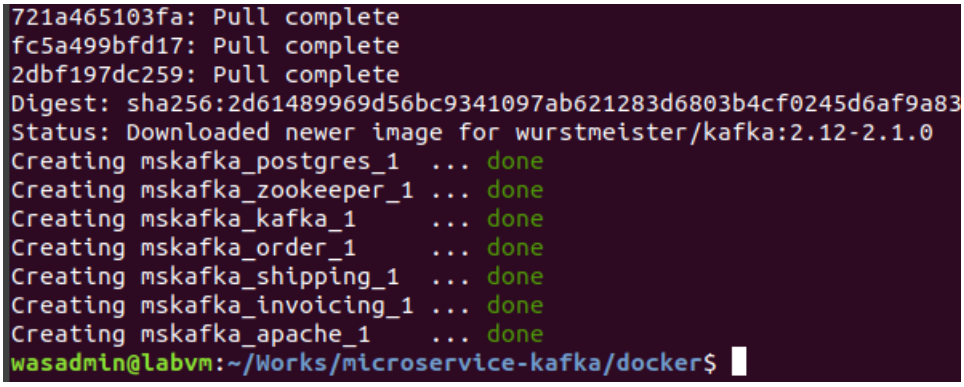
```
sudo lsof -i:8080
```

__4. Run docker-compose up:

```
docker-compose up -d
```

Wait for the containers to come up. Did they come up in the order you predicted?

You should get a result similar than this:



__5. Let's take a look at the running containers:

```
docker ps
```

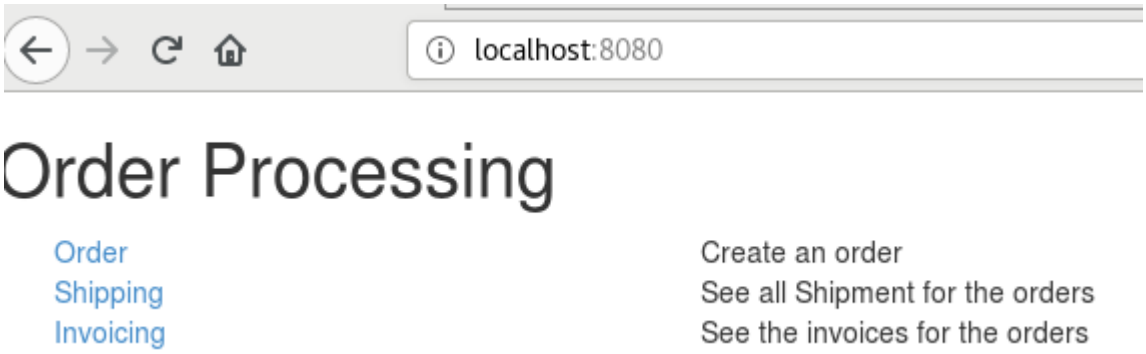
We should see a list of containers similar to the following:

```
CONTAINER ID        IMAGE                               COMMAND...
39f87b605a0e        mskafka_apache                      "/bin/sh...
efab6112b8a8        mskafka_invoicing                   "/bin/sh...
1f7a0ef5f53c        mskafka_shipping                    "/bin/sh...
2a6b6cf3abd9        mskafka_order                       "/bin/sh...
638623f27d23        mskafka_postgres                    "docker-...
aeab11a5474b        wurstmeister/kafka:2.12-2.1.0       "start-kafka.sh"...
cac702dcb14f        wurstmeister/zookeeper:3.4.6        "/bin/sh…
```

__6. Open a Firefox web browser and visit the following address:

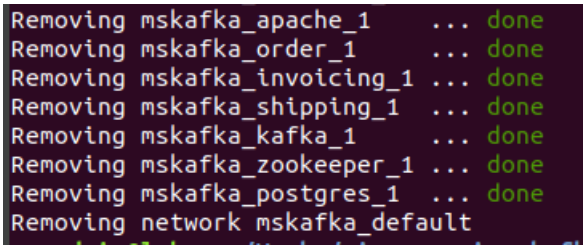
```
localhost:8080
```

__7. The order processing application is now available. Take a moment to explore the application.



__8. Once you are satisfied that the three microservices are working, let bring it down:

```
docker-compose down
```



__9. Confirm that everything was brought down successfully:

```
docker ps
```

__10. We can also revisit the order site. We should now get an error:

```
localhost:8080
```

We have successfully brought up and down a set of containers.

__11. Close the web browser.

__12. In the Terminal type **exit** many times until Terminal is closed.

Part 6 - Review

In this lab, we explored a project that made use of Docker Compose to conveniently bring up and down an entire microservice application and its dependencies.

Lab 4 - 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_WA2919_REL_1_0'.

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
```

__4. Enter *wasadmin* as the password, if prompted.

__5. 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.

__6. Copy the Docker configuration file to the kubelet directory.

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

__7. Restart the Kubelet service.

```
systemctl restart kubelet
```

__8. 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
```

__9. Restart the kubelet service one more time.

```
systemctl restart kubelet
```

__10. In the Terminal type **exit** many times until Terminal is closed.

__11. Close the browser.

Lab 5 - 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_WA2919_REL_1_0'.
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:

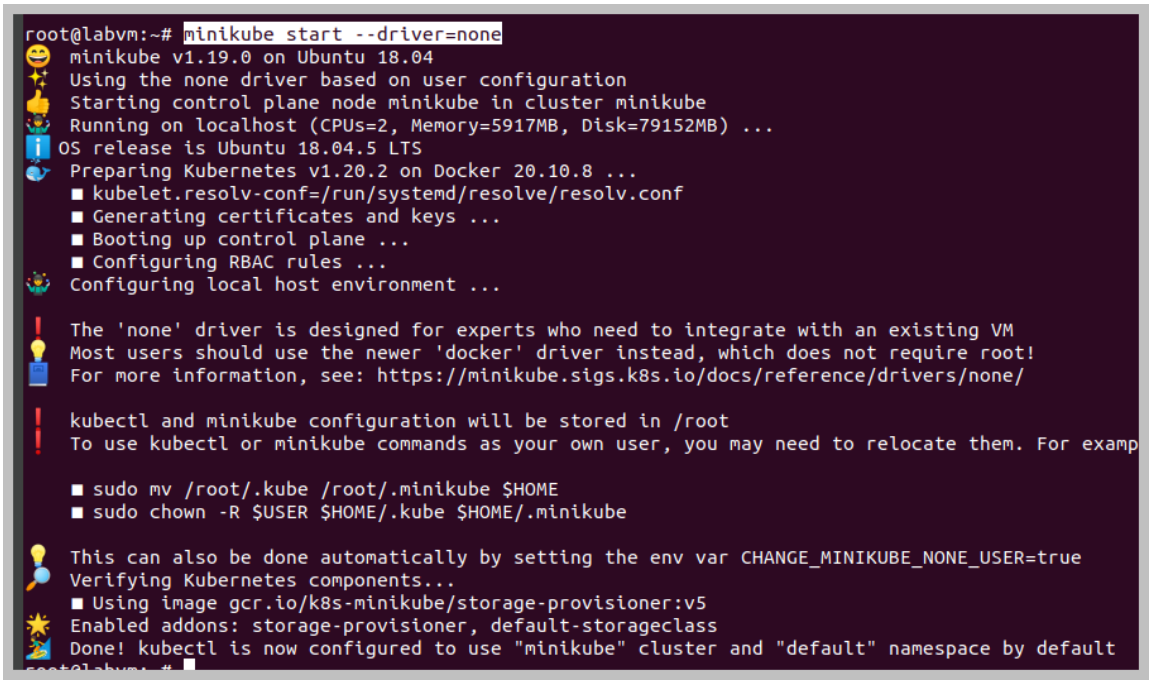
```
minikube stop
```

__2. Then delete minikube:

```
minikube delete
```

__3. Now, start minikube:

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

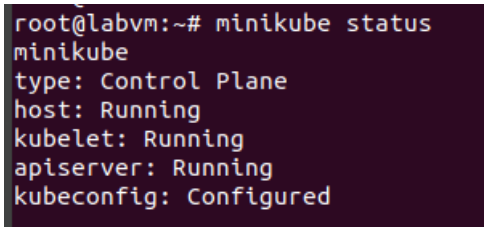


Note: The above command performs the following operations: 1. Generates the certificates and then proceeds to provision a local Docker host. 2. Starts up a control plane that provides various Kubernetes services. 3. Configures the default RBAC rules.

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

```
minikube status
```

Notice it shows messages like this:



__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]:

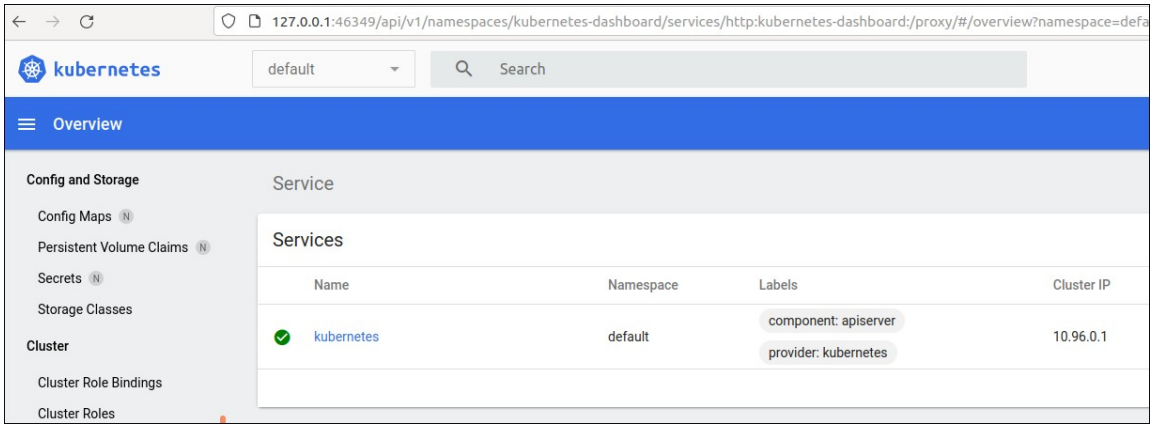
```
root@labvm:~# minikube dashboard
🔌  Enabling dashboard ...
    ▪ Using image kubernetesui/dashboard:v2.1.0
    ▪ Using image kubernetesui/metrics-scraper:v1.0.4
🤔  Verifying dashboard health ...
🚀  Launching proxy ...
🤔  Verifying proxy health ...
http://127.0.0.1:34249/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/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

__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:



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:

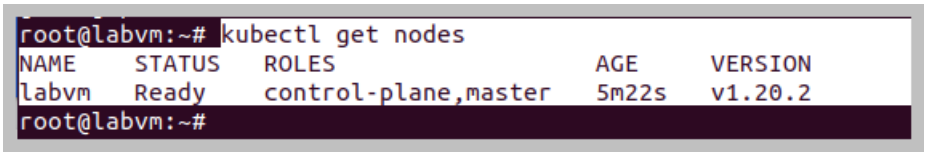
```
sudo -i
```

[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.



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
```

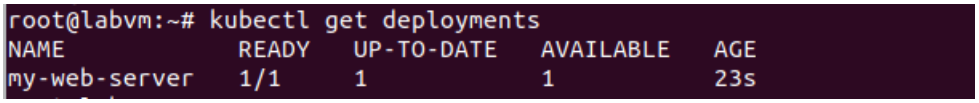
The command uses the nginx image to create a deployment named my-web-server and exposes the service on port 80. If the image is not available on your local machine, it connects to the Docker hub registry and downloads the image. You can specify other registries by specifying the full URL to the image.

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
```

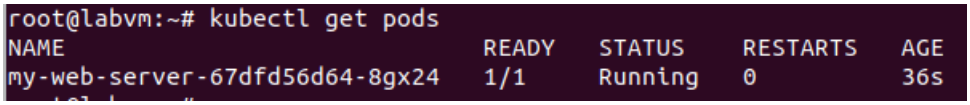
You should see: [You may need to wait and repeat the command until you see 1/1]



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

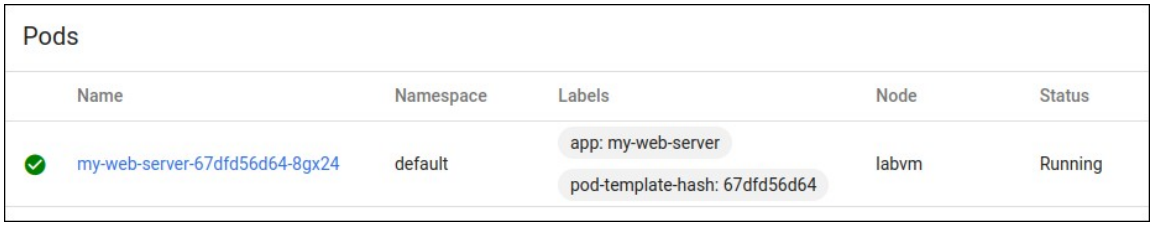
```
kubectl get pods
```

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



__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:



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.

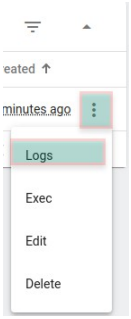
__4. Run following command to view the node details [replace <node name > with yours]:

```
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.

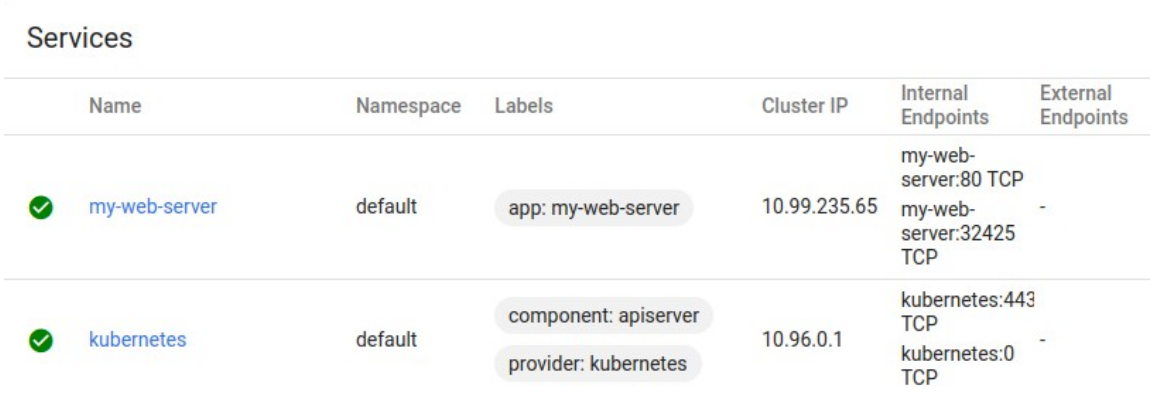
__1. In the terminal, run the following command:

```
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:



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.

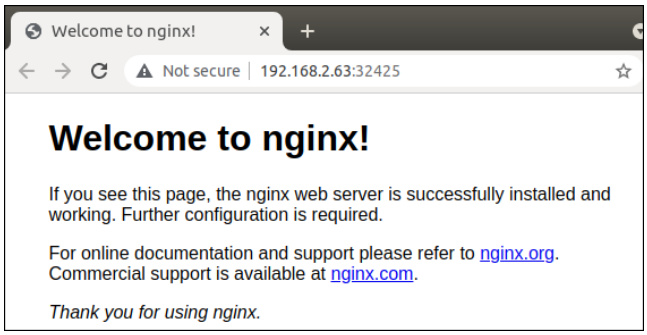
Troubleshooting. If you don't get a URL then delete the service by running:

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.



__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 **Log**.

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:

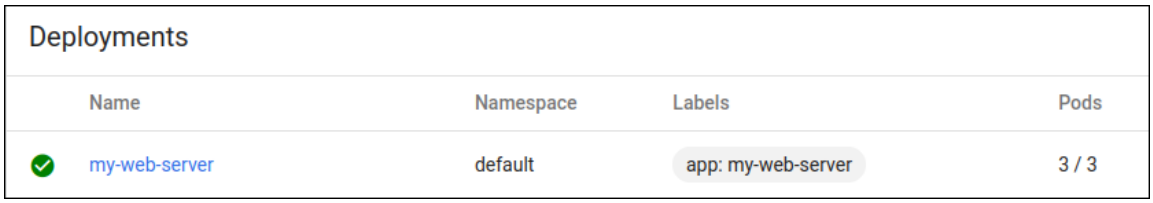
```
kubectl get deployment
```

Notice how it shows result like this:

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

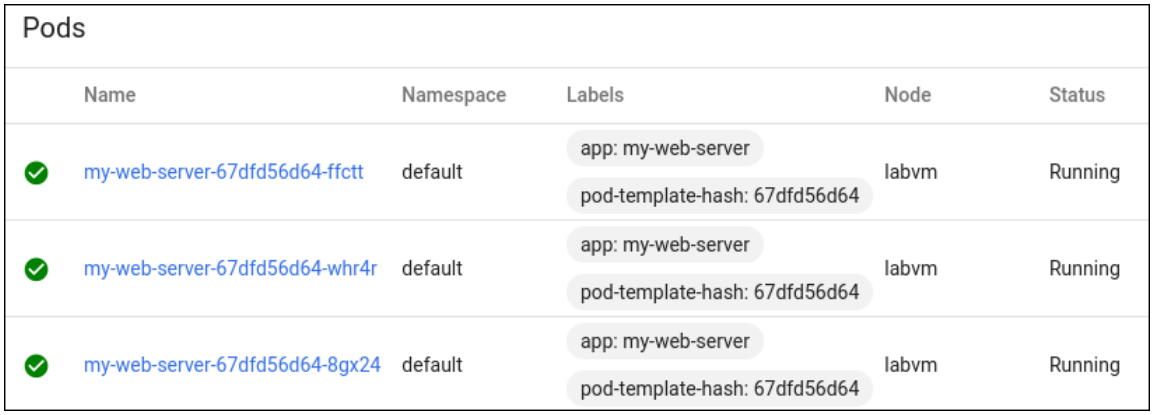
__4. On the dashboard page, in the web browser, click **Deployments** on the left side of the page.

Notice it shows 3/3 Pods.



__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:



__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:

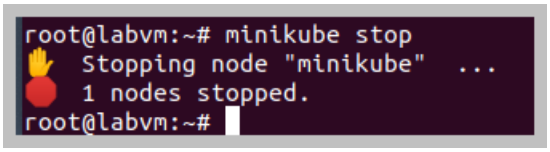
```
kubectl get deployments
```

__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.



__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 6 - Working with Kubernetes Workloads

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

In this Lab you will continue working in the 'VM_WA2919_REL_1_0'.

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

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

Part 1 - Pre-Setup

__1. Open a new Terminal.

__2. Switch to sudo:

```
sudo -i
```

__3. Enter **wasadmin** as password.

__4. Verify the current user with the command 'whoami', it should be 'root'.

__5. Navigate to the working directory:

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

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:

```
minikube status
```

__2. If should show stopped. If it is running then stop it using this command:

```
minikube stop
```

__3. Start minikube:

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

__4. Check that minikube is running:

```
minikube status
```

Part 3 - Create a Deployment

__1. Use gedit to create a deployment manifest file. This command will create the file and bring it up in edit mode:

```
gedit nginx-deployment.yaml
```

__2. Add the following contents to the file.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

__3. Save the file.

A Deployment named nginx-deployment is created, indicated by the .metadata.name field. The Deployment creates three replicated Pods, indicated by the replicas field. The selector field defines how the Deployment finds which Pods to manage. In this case, you simply select a label that is defined in the Pod template (app: nginx). However, more sophisticated selection rules are possible, as long as the Pod template itself satisfies the rule.

The template field contains the following sub-fields:

• The Pods are labeled app: nginx using the labels field.

• The Pod template's specification, or .template.spec field, indicates that the Pods run one container, nginx, which runs the nginx Docker Hub image at version 1.7.9.

• Create one container and name it nginx using the name field.

__4. Close the editor.

Don't worry about the warnings.

__5. Create the Deployment by running the following command:

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

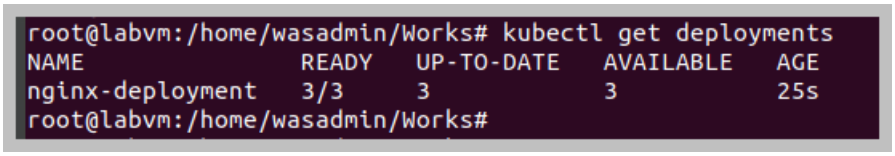
You should see:

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

__6. Run the following command to verify the Deployment was created:

```
kubectl get deployments
```

You should see:



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

__7. To see the Deployment rollout status, run the following command:

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

The output may similar to this:

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

__8. Run the **kubectl get deployments** command again a few seconds later.

__9. To view the details of the deployment run the following command:

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

__10. Labels are automatically generated for each Pod, to see what they are run:

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

The following output is returned:

```
root@osboxes:~/workspace# kubectl get pods --show-labels
NAME                                READY   STATUS    RESTARTS   AGE     LABELS
nginx-deployment-5754944d6c-g2b8n   1/1     Running   0          6m48s   app=nginx,pod-template-hash=5754944d6c
nginx-deployment-5754944d6c-rhw4m   1/1     Running   0          6m48s   app=nginx,pod-template-hash=5754944d6c
nginx-deployment-5754944d6c-tg5kv   1/1     Running   0          6m48s   app=nginx,pod-template-hash=5754944d6c
root@osboxes:~/workspace#
```

Part 4 - Update a Deployment

__1. Run the following command to check the nginx version image used by the current Deployment:

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

Notice image is displayed as 1.7.9

__2. Open the Deployment manifest for editing:

```
gedit nginx-deployment.yaml
```

__3. Change image version from 1.7.9 to 1.9.1

__4. Change replica count from 3 to 4

__5. Save and close the editor.

__6. Apply the updates:

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

__7. Check rolling update status:

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

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

__8. View the Deployment:

```
kubectl get deployments
```

Ensure there are 4 replicas.

__9. Verify nginx version is updated to 1.9.1:

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

Ensure Image version is 1.9.1

Part 5 - Roll Back a Deployment

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

__1. Suppose that you made a typo while updating the Deployment, by putting the image name as nginx:1.91 instead of nginx:1.9.1, enter this command:

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

The output is similar to this:

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

__2. The rollout gets stuck. You can verify it by checking the rollout status:

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

__3. Press Ctrl+C to stop the above rollout status watch.

__4. Run the following command to get pod list:

```
kubectl get pods
```

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

__5. Undo the recent change:

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

__6. Verify the invalid pods are removed (you may need to repeat the command until they are gone):

```
kubectl get pods
```

__7. Scale a deployment by using imperative command:

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

__8. Verify scaling is configured:

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

Part 6 - Pausing and Resuming a Deployment

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

__1. Pause the Deployment by running the following command:

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

__2. While the rollout is paused, set the Deployment image to a different version:

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

__3. Run the following command to verify the image version:

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

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

__4. Press Ctrl+C to exit out to the terminal.

__5. Resume rollout:

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

__6. Verify the rollout status:

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

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

__7. Verify the Deployment image version:

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

Note: The Image version should show as 1.9.2

Part 7 - Clean-Up

__1. Delete your Deployment:

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

__2. Verify the deployment is deleted:

```
kubectl get deployments
```

__3. Verify the nginx-deployment pods are deleted:

```
kubectl get pods
```

Note: It might take a few moments for the pods to get deleted.

__4. Stop Kubernetes:

```
minikube stop
```

__5. Delete the Kubernetes cluster:

```
minikube delete
```

__6. When the command is done then close the terminal window [type exit].

Part 8 - Review

In this lab, you used the Deployment Kubernetes workload. You also performed various operations, such as upgrade, pause, resume, and scale the Deployment.

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_WA2919_REL_1_0'.

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.

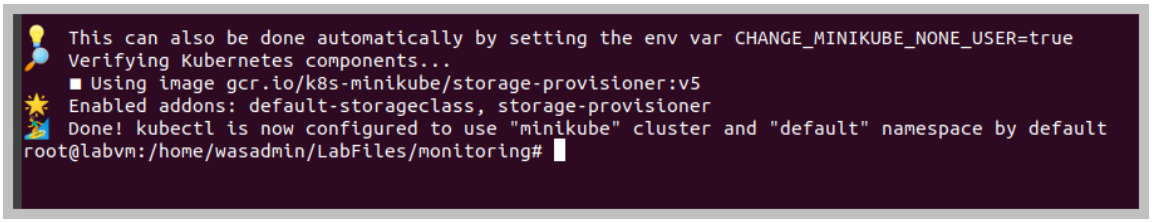
__1. Check if MiniKube is already running:

```
minikube status
```

__2. If MiniKube is not running, start MiniKube:

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

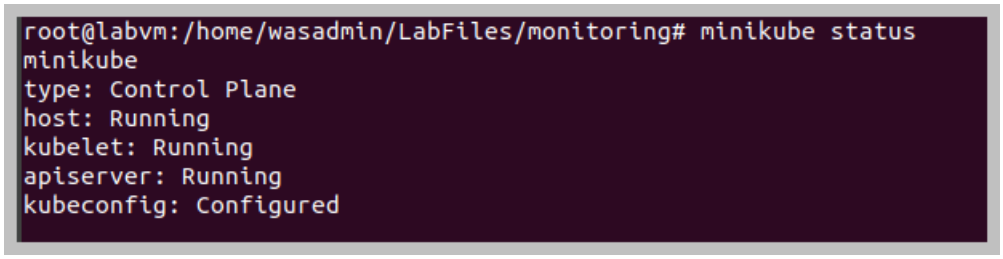
Wait for the service to start up. It can take up to 5 minutes. If it fails, try again. It's a very resource-intensive service and takes time.



__3. Get MiniKube status again:

```
minikube status
```

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



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: ['10.98.46.92:8080']
```

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

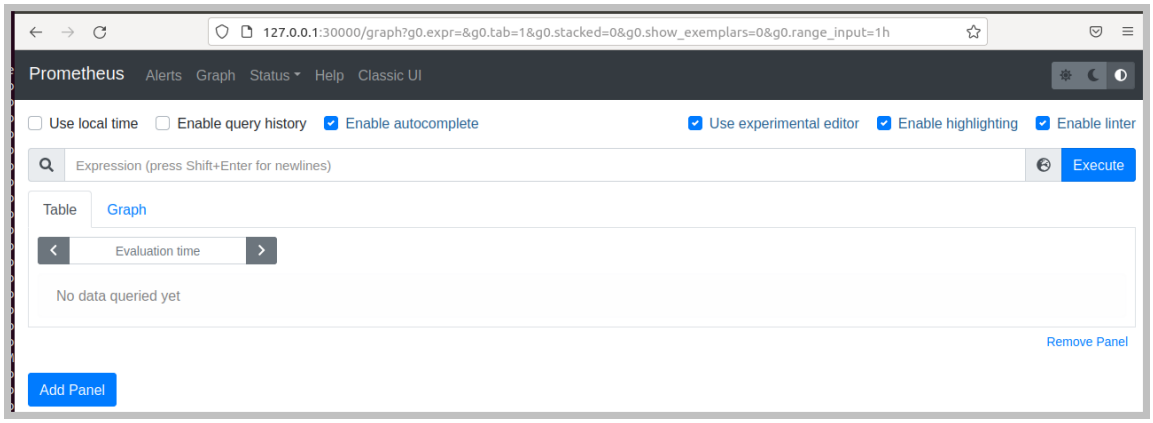
In this part, you will access the Prometheus service and use various metrics.

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

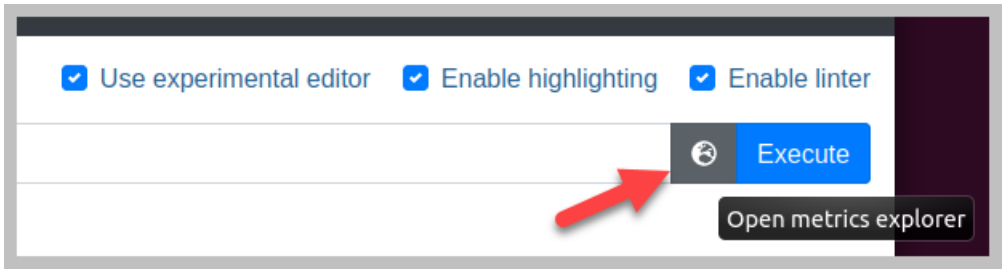
```
http://127.0.0.1:30000
```

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

The default site should look like this:



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

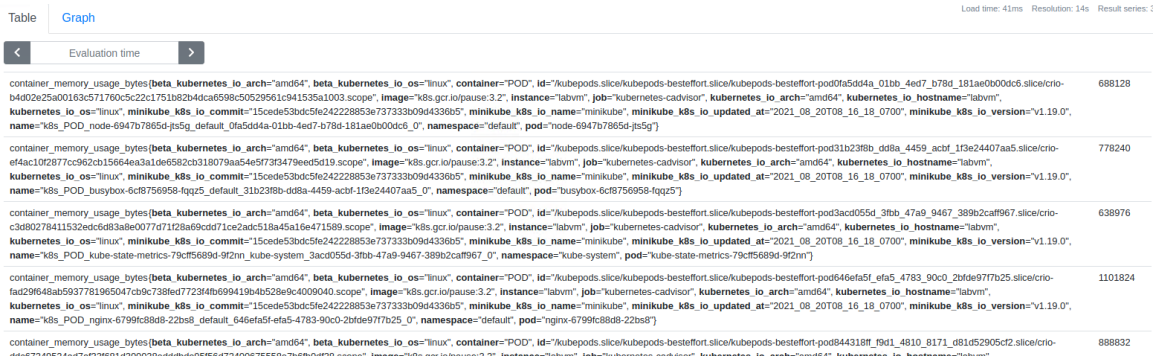


Notice there are several metrics available.

__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.



__5. Click **Graph**.

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

__6. 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 ()

__7. 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"}
```

__8. 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.

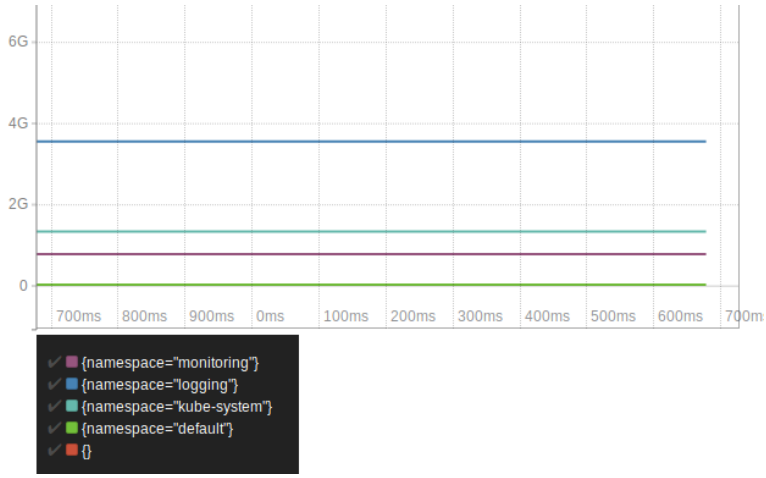
__9. Change the query to the following and click **Execute** to view the total memory consumed by containers in “monitoring” namespace of the cluster:

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

__10. 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.



__11. Change the query to the following and click **Execute** button to view the request count:

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

__12. 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.

__4. Close the browser.

Part 10 - Review

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