

WA3003 Docker and Kubernetes Administration Training

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Creating a Docker Account and Obtain an Access Token.....	3
Lab 2 - Managing Containers.....	6
Lab 3 - Building Images.....	12
Lab 4 - Dockerfiles.....	16
Lab 5 - Getting Started with Docker Compose.....	20
Lab 6 - Docker Volumes.....	31
Lab 7 - Custom Network Management.....	36
Lab 8 - Configuring Minikube/Kubernetes to Use a Custom Docker Account.....	38
Lab 9 - Accessing the Kubernetes API.....	38
Lab 10 - Working with Kubernetes Workloads.....	46
Lab 11 - Scheduling and Node Management.....	54
Lab 12 - Accessing Applications.....	68
Lab 13 - Using Persistent Storage.....	77
Lab 14 - Getting Started with Helm.....	93
Lab 15 - Build CI Pipeline with Jenkins.....	112
Lab 16 - Appendix: Deploying Stateful Services in Docker.....	134

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 - Setting up the scene

In this part, you will open the terminal window and switch to the root user.

- __ 1. Open the VM image or connect to the lab environment.
- __ 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
```

It should show root.

Part 2 - Create a Docker account

IMPORTANT NOTE!!!

If you have been provided with a Docker account, skip to Part 3. Otherwise, complete Part 1 to create a Docker account

In this part, you will create a Docker account.

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

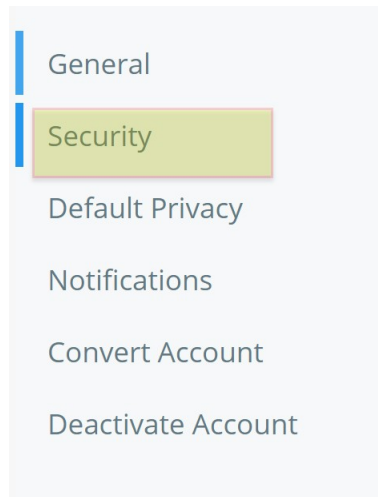
```
https://hub.docker.com/
```

- __ 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.



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

Part 3 - Log in with your Docker account

__1. Switch to the terminal window where you logged in as root.

__2. Run the following command to log into Docker.

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

__3. Close the Terminal.

Lab 2 - Managing Containers

Docker is an open IT automation platform widely used by DevOps, and in this lab, we will review the Docker commands for managing containers. The lab starts with Docker basics and then gets into managing containers.

Part 1 - Setup

__1. Open a new Terminal.

__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 system-wrecking consequences of issuing a wrong command.

__4. Enter the following command:

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

You should see that your system prompt has changed to:

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

__5. 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 - 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!'
```

Notice it displays 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 3 - List and Delete Container

In this lab part, we will need a second terminal also running a shell as **root**.

__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. In the new terminal (**T2**) become **root** (use the **sudo -i** command).

__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 and copy the id of the new container:

```
docker ps -a
```

__6. Switch to the original terminal window (**T1**).

__7. Enter the following command and replace the container id with the one 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.

__8. Switch to the **T2** terminal.

__9. Enter the following command:

```
docker ps
```

Now it should show an empty table.

Part 4 - 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 status as Exited.

__5. Enter the following command to restart the container (pick the new one 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.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	...
aed47e954acd	ubuntu	"/bin/bash"	3 minutes ago	Exited (0)	4 minutes ago

Leave the image as-is. We will be using it in the next lab.

Part 5 - Review

In this lab, we got started using Docker and looked at some commands for managing containers.

Lab 3 - Building Images

In this lab we will take an existing container and use it to create an image.

Part 1 - Setup

- __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. Verify the current user with the following command:

```
whoami
```

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

```
root
```

- __4. Navigate to the working directory:

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

You should see that your system prompt has changed to:

```
[root@localhost Works]#
```

Part 2 - Create a Custom Image

You should already have a container that was created in the previous lab. It will be used in this part to create a custom image.

- __1. Check for the container using the following command:

```
docker ps -a
```

If the container exists you should see something like this:

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

If the container DOES NOT exist you should instead see this:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	...
--------------	-------	---------	---------	--------	-----

__2. If the container does not exist then run the following command to re-create it:

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

The container will be created and the container prompt "root@basic_host:/# " will appear.

__3. Type 'exit' at the container prompt and hit enter.

__4. Check again for the container using 'docker ps -a' it should now appear in the list.

__5. 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.

__6. Enter the following command:

```
docker images
```

You should see the new image listed on top of the available images in our local image repository.

__7. 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.

__8. Enter the following command at the container prompt:

```
exit
```

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

Part 3 - 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:

NOTE: DO NOT delete any image or container other than the ones you have created in this lab. Leave any other images / containers as is.

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

__3. Verify there are no docker containers created by you:

```
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 has gone using the following command to list existing images:

```
docker images
```

__ 6. Close all terminals.

Part 4 - Review

In this lab, we created an image from a container.

Lab 4 - Dockerfiles

In this lab we will create a Dockerfile and use it to create a custom image. Then we will run a Java application using the image.

Part 1 - 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 - 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

# 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 will 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 3 - 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**.

__9. Close all terminals.

Part 4 - Review

In this lab we used a Dockerfile to create a new image and then ran a Java application using the image.

Lab 5 - 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

__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 commands to stop it and delete it:

```
docker stop <containerid>
```

```
docker rm <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. Edit the docker-compose.yml file:

```
edit docker/docker-compose.yml
```

__11. Remove all the link sections as as shown in red below:

```
version: '3'
services:
  zookeeper:
    image: wurstmeister/zookeeper:3.4.6
  kafka:
    image: wurstmeister/kafka:2.12-2.1.0
    links:
      - zookeeper
    environment:
      KAFKA_ADVERTISED_HOST_NAME: kafka
      KAFKA_ADVERTISED_PORT: 9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_CREATE_TOPICS: "order:5:1"
  apache:
    build: apache
    links:
      - order
      - shipping
      - invoicing
    ports:
      - "8080:80"
  postgres:
    build: postgres
    environment:
      POSTGRES_PASSWORD: dbpass
      POSTGRES_USER: dbuser
  order:
    build: ../microservice-kafka/microservice-kafka-order
    links:
      - kafka
      - postgres
    environment:
      SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
  shipping:
    build: ../microservice-kafka/microservice-kafka-shipping
    links:
      - kafka
      - postgres
    environment:
      SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
  invoicing:
    build: ../microservice-kafka/microservice-kafka-invoicing
    links:
      - kafka
      - postgres
    environment:
      SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
```

__12. Make sure your edited file looks like below:

```
version: '3'
services:
  zookeeper:
    image: wurstmeister/zookeeper:3.4.6
  kafka:
    image: wurstmeister/kafka:2.12-2.1.0
    environment:
      KAFKA_ADVERTISED_HOST_NAME: kafka
      KAFKA_ADVERTISED_PORT: 9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_CREATE_TOPICS: "order:5:1"
  apache:
    build: apache
    ports:
      - "8080:80"
  postgres:
    build: postgres
    environment:
      POSTGRES_PASSWORD: dbpass
      POSTGRES_USER: dbuser
  order:
    build: ../microservice-kafka/microservice-kafka-order
    environment:
      SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
  shipping:
    build: ../microservice-kafka/microservice-kafka-shipping
    environment:
      SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
  invoicing:
    build: ../microservice-kafka/microservice-kafka-invoicing
    environment:
      SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
```

__13. Save and close the file.

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.

```
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] microservice-kafka ..... SUCCESS [ 7.675 s]
[INFO] microservice-kafka-order ..... SUCCESS [ 17.597 s]
[INFO] microservice-kafka-shipping ..... SUCCESS [ 1.570 s]
[INFO] microservice-kafka-invoicing ..... SUCCESS [ 1.882 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

__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
```

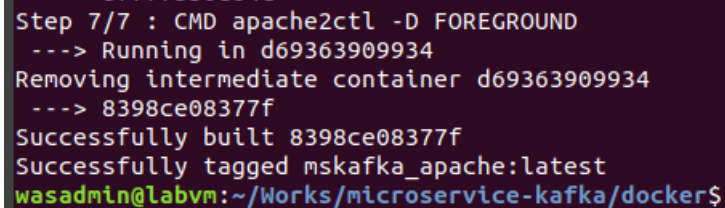
__2. Enable port forwarding to docker containers can connect to internet and download packages, enter **wasadmin** as password:

```
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.



```
Step 7/7 : CMD apache2ctl -D FOREGROUND
--> Running in d69363909934
Removing intermediate container d69363909934
--> 8398ce08377f
Successfully built 8398ce08377f
Successfully tagged mskafka_apache:latest
wasadmin@labvm:~/Works/microservice-kafka/docker$
```

__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...
...		

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 if 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:

```
721a465103fa: Pull complete
fc5a499bfd17: Pull complete
2dbf197dc259: Pull complete
Digest: sha256:2d61489969d56bc9341097ab621283d6803b4cf0245d6af9a83
Status: Downloaded newer image for wurstmeister/kafka:2.12-2.1.0
Creating mskafka_postgres_1 ... done
Creating mskafka_zookeeper_1 ... done
Creating mskafka_kafka_1 ... done
Creating mskafka_order_1 ... done
Creating mskafka_shipping_1 ... done
Creating mskafka_invoicing_1 ... done
Creating mskafka_apache_1 ... done
wasadmin@labvm:~/Works/microservice-kafka/docker$
```

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

docker ps

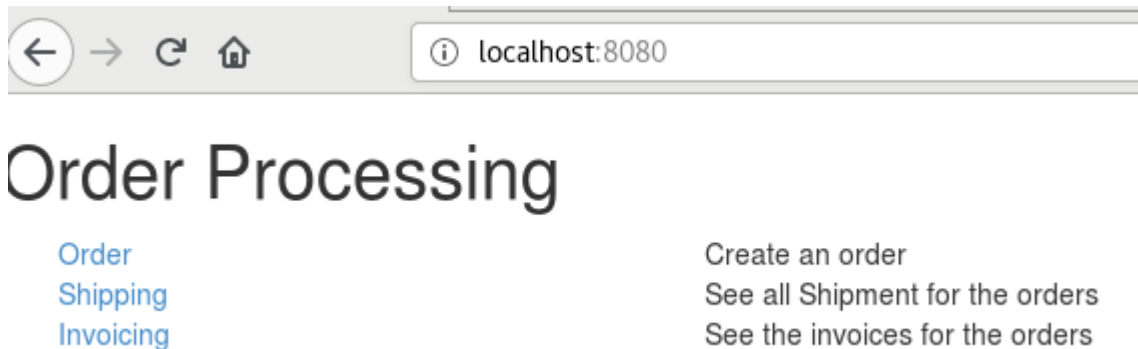
We should see a list of containers similar to the following:

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
```

You will see:

```
Removing mskafka_apache_1    ... done
Removing mskafka_order_1    ... done
Removing mskafka_invoicing_1 ... done
Removing mskafka_shipping_1 ... done
Removing mskafka_kafka_1    ... done
Removing mskafka_zookeeper_1 ... done
Removing mskafka_postgres_1 ... done
Removing network mskafka_default
```

__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 6 - Docker Volumes

Docker allows us to create Volumes which hold files, can be mounted into containers, and can be accessed by multiple containers. In this lab we will create a volume and use it from two separate containers.

Part 1 - Setup

Before starting the lab we will stop and remove any existing containers.

__ 1. Open a terminal and change to the root user:

```
sudo -i
```

__ 2. Navigate to the Works directory:

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

__ 3. Check for running containers:

```
docker ps
```

__ 4. If any containers are listed stop them

```
docker stop {container-name or container-id}
```

__ 5. Check again, the container should no longer be in the list.

__ 6. List all existing stopped volumes:

```
docker ps -a
```

__ 7. Execute the following cmd to remove each of the stopped volumes:

```
docker rm {container-name or container-id}
```

You are now ready to start the lab.

Part 2 - 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. Inspect the volume:

```
docker volume inspect hello
```

The output should look like this:

```
[
  {
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/hello/_data",
    "Name": "hello",
    "Options": {},
    "Scope": "local"
  }
]
```

__3. Create a container and mount it in the container:

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

__4. View mount point in container:

```
ls
```

The output includes the 'world' directory which is where the volume is mounted:

```
bin  dev  home  lib32  libx32  mnt  proc  run  srv  tmp  var
boot  etc  lib  lib64  media  opt  root  sbin  sys  usr  world
```

__5. Create a file in the volume:

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


__6. Verify that the file was created:

```
ls /world
```

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

__7. Exit out of the container:

```
exit
```

__8. List all containers:

```
docker ps -a
```

__9. Destroy the container: (Note: Make a note of the ID of your container and substitute it below)

```
docker stop <id>  
docker rm <id>
```

__10. Create another container (temp2) and attach the volume as a mount point:

```
docker run -v hello:/world --name temp2 -it ubuntu /bin/bash
```

__11. View mount point in container: (it will show the /world directory)

```
ls /world
```

__12. View contents of the file:

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

__13. Exit out of the container:

```
exit
```

__14. View all volumes:

```
docker volume ls
```

Part 3 - 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. In case, if there are any containers, stop (if running) 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:

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

Syntax is: docker rmi <REPOSITORY:TAG>

__7. Make sure your image has been deleted:

```
docker images
```

__8. In each Terminal type **exit** and then close the Terminal window.

Part 4 - Review

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

Lab 7 - Custom Network Management

Docker allows developers to create networks to link applications in separate containers. For example one container might host a web server while another one might include a client that needs to access the web server. The easiest way to do this in Docker is to have both containers set up on the same network.

In this lab we will:

1. Create a custom Docker network
2. Run a web server container that uses the network
3. Test the Network by Calling the Server from a Client Container

Part 1 - Setup

Before starting the lab we should stop and remove any existing containers.

__1. Open a terminal and change to the root user:

```
sudo -i
```

__2. Navigate to the Works directory:

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

__3. Check for running containers:

```
docker ps
```

__4. If any containers are listed stop them:

```
docker stop {container-name or container-id}
```

__5. Check again, the container should no longer be in the list.

__6. List all existing stopped volumes:

```
docker ps -a
```

__7. Execute the following command to remove each of the stopped volumes:

```
docker rm {container-name or container-id}
```

You are now ready to start the lab.

Part 2 - Create a Custom Docker network

__1. List the existing docker networks using this command:

```
docker network ls
```

__2. You should get the following output (ids will differ):

NETWORK ID	NAME	DRIVER	SCOPE
b95340483036	bridge	bridge	local
c6759ab4e678	host	host	local
1bd12a3a5484	none	null	local

These are the default networks that are created when docker starts.

__3. Lets take a closer look at the default 'bridge' network using the following 'inspect' command:

```
docker network inspect bridge
```

A lot of information is output:

```
[
  {
    "Name": "bridge",
    "Id": "b9534048303667f6258fcababb68483200dc2ac7a365d6a943181ee9fee4ee8a",
    "Created": "2020-08-16T14:21:25.920506173-04:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "172.17.0.0/16",
          "Gateway": "172.17.0.1"
        }
      ]
    }
  },
]
```

```

    "Internal": false,
    "Attachable": false,
    "Containers": {},
    "Options": {
      "com.docker.network.bridge.default_bridge": "true",
      "com.docker.network.bridge.enable_icc": "true",
      "com.docker.network.bridge.enable_ip_masquerade": "true",
      "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
      "com.docker.network.bridge.name": "docker0",
      "com.docker.network.driver.mtu": "1500"
    },
    "Labels": {}
  }
]

```

Notice that the "Containers" section is empty.

__ 4. Lets run a container and see how that affects the default network. Run the following command:

```
docker run -d -it --name client alpine
```

Note that the above command does not specify a network.

__ 5. Now inspect the bridge network again and take a look at the "Containers" section:

```
docker network inspect bridge
```

Notice how the 'client' container shows up:

```

"Containers": {
  "bb5ca32222abcb7c992190fa0084c76036006d4ef617c143f24c37a6d245cf4a": {
    "Name": "client",
    "EndpointID": "0f5fcceafe8739d30c4a1f26165b7bc34a55b21ba15d31f7a4841961f076823b",
    "MacAddress": "02:42:ac:11:00:02",
    "IPv4Address": "172.17.0.2/16",
    "IPv6Address": ""
  }
},

```

__ 6. Lets create a custom Docker network named 'mynet'. Use the following command:

```
docker network create mynet
```

__7. Inspect the new network and verify that its 'Containers' section is empty:

```
docker network inspect mynet
```

The 'Containers' section should look like this:

```
"Containers": {},
```

__8. Use the following command to connect the custom network 'mynet' with the container:

```
docker network connect mynet client
```

__9. Inspect both the custom 'mynet' and default 'bridge' networks using these commands? What do you see?

```
docker network inspect bridge
```

```
docker network inspect mynet
```

It looks like they are BOTH connected. Lets fix that.

__10. Run the following command:

```
docker network disconnect bridge client
```

__11. Inspect the 'bridge' network again. You should see that it is no longer associated with the 'client' container.

__12. Stop and remove the 'client' container:

```
docker stop client
```

```
docker rm client
```

__13. Instead of connecting to a network after the container is created we can connect as part of the run command. Use the following command to recreate and start the 'client' container on the 'mynet' network:

```
docker run -d -it --network mynet --name client alpine
```

__14. Inspect the bridge and mynet networks again. You should see that only 'mynet' is connected to the 'client' container:

```
docker network inspect bridge
```

```
docker network inspect mynet
```

__15. To prepare for the next part lets remove the 'client' container using these commands:

```
docker stop client
```

```
docker rm client
```

Part 3 - Run a Web Server Container that uses the Custom Network

The following command runs an nginx web server. Notice that does not include the 'network' parameter. If executed it would run the container on Docker's default bridge network:

```
docker run -it -d --name web nginx
```

To be useful we would have to expose the port it sits on as well using the '-p' parameter:

```
docker run -it -d -p 8080:80 --name web nginx
```

Now we could access it from the host using a browser or the following curl command:

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

But what if we wanted instead to access it from another container? That's when a custom docker network would help.

__1. The following command runs the web server container on our custom Docker network 'mynet'. Go ahead and run the command:

```
docker run -it -d --name web --network mynet nginx
```


Part 4 - Call the Server from a Client Container

__1. Run the following command, it will start up a client and leave us at a the client container's shell prompt:

```
docker run -it --network mynet --name client alpine
```

The prompt should look like this: `/ #`

The image we are using 'alpine' is a minimal Linux image which by default does not include the curl networking command. Installing the command will allow us to make network calls from the client container.

__2. Run the following command at the client container prompt to install Curl:

```
apk add curl
```

Troubleshooting: If curl cannot be installed then you will need to restart docker, delete the containers, runs the web server container and then restart this part.

1. This command will restart docker:

```
sudo systemctl restart docker
```

2. Then find the containers: `docker ps -a`

Remove them: `docker rm xxxx`

3. Runs the web server container on our custom Docker network 'mynet':

```
docker run -it -d --name web --network mynet nginx
```

4. Restart this Part.

__3. Now we will use curl to call the nginx server in the 'web' container. Run this command from the client container prompt:

```
curl http://web
```

Notice how the server name in the command matches the name of the nginx container - "web". The server should respond with the following web page:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

We can use the same type of custom network between any client and server or between multiple clients and a server or between one server and another.

__4. Try pinging the 'web' container by executing this command at the 'client' container prompt:

ping web

Your output should be similar to this:

```
PING web (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.208 ms
64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.134 ms
64 bytes from 172.18.0.3: seq=2 ttl=64 time=0.147 ms
```

__5. Stop it by pressing Ctrl+C.

__6. We can also ping the 'client' container:

ping client

Your output should be similar to this:

```
PING client (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.141 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.067 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.134 ms
```

From these outputs we can see that the custom Docker network 'mynet' has assigned an IP address of 172.18.0.3 to the 'web' container and 172.18.0.2 to the 'client' container.

__ 7. Stop it by pressing Ctrl+C.

__ 8. Close the 'client' container shell prompt:

```
exit
```

__ 9. Close all Terminals.

Part 5 - Review

In this lab we set up a custom Docker network and used it to communicate between client and server applications.

Lab 8 - 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.

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. You may 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 9 - Accessing the Kubernetes API

All access, for configuration and management of the Kubernetes cluster goes through the Kubernetes API-Server component. In this lab we will work with two tools that allow us to work with Kubernetes through the API:

1. kubectl
2. Kubernetes API Proxy

Part 1 - 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
```

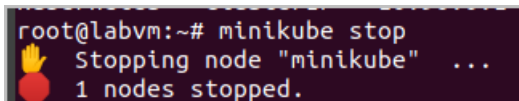
Part 2 - Start the Cluster

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

- __ 1. Stop minikube:

```
minikube stop
```

You should see:



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

But you may see a timeout message as below:

```
root@labvm:~# minikube stop
👉 Stopping node "minikube" ...
❌ Exiting due to GUEST_STOP_TIMEOUT: Temporary Error: stop: Maximum number of retries (60) exceeded
🐱 If the above advice does not help, please let us know:
👉 https://github.com/kubernetes/minikube/issues/new/choose
```

__2. Delete minikube:

```
minikube delete
```

It may take a while to complete:

```
root@labvm:~# minikube delete
🔄 Uninstalling Kubernetes v1.20.2 using kubeadm ...
🔥 Deleting "minikube" in none ...
💀 Removed all traces of the "minikube" cluster.
root@labvm:~#
```

__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 v1.19.0 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=5917MB, Disk=79152MB) ...
OS release is Ubuntu 18.04.5 LTS
Preparing Kubernetes v1.20.2 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 example,

  ■ 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
root@labvm:~#
```

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 - The kubectl command line interface (CLI)

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

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

- explain
- create
- apply
- get
- delete

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

Lets start by checking the cluster for any existing deployments. For this we will use the 'Get' command. The syntax of this command is

```
kubectl get {resource-type}
```

__1. Run the following **get** command:

```
kubectl get deployments
```

Your cluster may or may not show any existing deployments. If it does then the output would look something like this:

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

If you have any existing deployments lets go ahead and delete them. You will use the `kubectl delete` command which has this syntax:

```
kubectl delete {resource-type} {resource-name}
```

__2. Run the following **delete** command. Make sure to substitute the name that was output from the earlier `get` command. Repeat this command for each deployment you found in your `get` listing.

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

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

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

__3. To use this command we'll need a resource definition file. Lets create one using the `gedit` editor:

```
gedit myapp.yaml
```

__4. Enter the following text into the file, then save it, and close the editor.

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

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

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

__5. Try running the following explain command:

```
kubectl explain deployments
```

The output of the command is:

```
KIND:      Deployment
VERSION:   apps/v1

DESCRIPTION:
  Deployment enables declarative updates for Pods and ReplicaSets.

FIELDS:
  apiVersion      <string>
    APIVersion defines the versioned schema of this representation of an
    object. Servers should convert recognized schemas to the latest internal
    value, and may reject unrecognized values. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-
    conventions.md#resources
```

```

kind      <string>
  Kind is a string value representing the REST resource this object
  represents. Servers may infer this from the endpoint the client submits
  requests to. Cannot be updated. In CamelCase. More info:
  https://git.k8s.io/community/contributors/devel/sig-architecture/api-
  conventions.md#types-kinds

metadata  <Object>
  Standard object metadata.

spec      <Object>
  Specification of the desired behavior of the Deployment.

status    <Object>
  Most recently observed status of the Deployment.

```

___6. Its also possible to drill down on this information. For example what about if we wanted to know more about the fields under "**metadata**"? Try running the following command:

```
kubectl explain deployments.metadata
```

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

```

KIND:      Deployment
VERSION:   apps/v1

RESOURCE: metadata <Object>

DESCRIPTION:
  Standard object metadata.

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

FIELDS:
  annotations          <map[string]string>
    Annotations is an unstructured key value map stored with a resource that
    may be set by external tools to store and retrieve arbitrary metadata. They
    are not queryable and should be preserved when modifying objects. More
    info: http://kubernetes.io/docs/user-guide/annotations
  ...

```

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

__7. Execute the following **apply** command to create a deployment based on the yaml file we just created:

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

__8. Check that the deployment was created:

```
kubectl get deployments
```

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

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

__9. Deploy the '**alpine**' docker image using the following **create** command:

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

__10. Check deployments:

```
kubectl get deployments
```

The output should look like this:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
apache	1/1	1	1	30m
nginx	1/1	1	1	46s

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

Part 4 - Kubernetes Dashboard

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

__1. Run the following command to start the dashboard:

```
minikube dashboard
```

After a minute it should output the following to the console.

```
root@labvm:/home/wasadmin/Works# minikube dashboard
Enabling dashboard ...
  ■ Using image kubernetesui/metrics-scraper:v1.0.4
  ■ Using image kubernetesui/dashboard:v2.1.0
🤖 Verifying dashboard health ...
🚀 Launching proxy ...
🤖 Verifying proxy health ...
http://127.0.0.1:36049/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/
```

__2. Copy the URL.

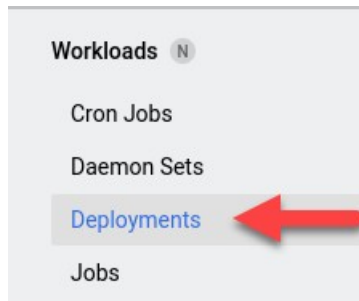
__3. Open the Mozilla browser inside the VM and paste the URL.

The dashboard app looks like this:

The screenshot shows the Kubernetes Dashboard interface. The top navigation bar includes the Kubernetes logo, a dropdown menu set to 'default', and a search bar. The left sidebar contains a menu with categories: Workloads (with a count of 8), Service (with a count of 1), and Config and Storage. The main content area is titled 'Overview' and features a 'Workloads' section with three large green circular indicators for Deployments, Pods, and Replica Sets. Below this is a 'Deployments' table with columns for Name, Namespace, Labels, Pods, Created, and Images.

Name	Namespace	Labels	Pods	Created	Images
✓ nginx	default	app: nginx	1 / 1	an hour ago	nginx:latest
✓ apache	default	app: apache	1 / 1	an hour ago	httpd:latest

__4. The app has a navigation area to the left of the screen with links. Click on the **deployments** link under **Workloads**.



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

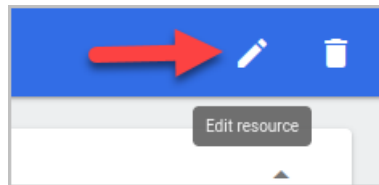
Deployments					
	Name	Namespace	Labels	Pods	Created ↑ Images
✓	nginx	default	app: nginx	1 / 1	3 minutes ago nginx:latest
✓	apache	default	app: apache	1 / 1	4 minutes ago httpd:latest

__5. Click the **nginx** deployment. This will show you the deployment details.

Metadata				
Name	Namespace	Created	Age	UID
nginx	default	Dec 23, 2021	3 minutes ago	53858eae-f3fd-40d7-91e2-c482d02c5bc4
Labels				
app: nginx				
Annotations				
deployment.kubernetes.io/revision: 1				

__6. Scroll down to see more sections.

__7. Click on the **Edit resource** button at the upper right of the screen.



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

Edit a resource

YAML

JSON

```
1 kind: Deployment
2 apiVersion: apps/v1
3 metadata:
4   name: nginx
5   namespace: default
6   uid: 5fbfcd4f-8597-4dcd-9031-8834b581a9da
7   resourceVersion: '3741'
8   generation: 1
9   creationTimestamp: '2021-12-22T17:04:52Z'
10  labels:
11    app: nginx
12  annotations:
13    deployment.kubernetes.io/revision: '1'
14  managedFields:
15    - manager: kubectl-create
16      operation: Update
17      apiVersion: apps/v1
18      time: '2021-12-22T17:04:52Z'
19      fieldsType: FieldsV1
20      fieldsV1:
21        'f:metadata':
```

This action is equivalent to: `kubectl apply -f <spec.yaml>`

Update

Cancel

__8. Click **Cancel** to dismiss the edit window.

__9. Scale, Edit and Delete resources are also available at the top right of the window.



Feel free to try out some of the other links. As you become more familiar with what's available in Dashboard and as you work more with Kubernetes you may find some operations are easier to execute from the dashboard's GUI than they are from a terminal using `kubectl` commands.

__10. Close the browser.

__11. In the Terminal, stop the command by pressing CTRL+C.

Feel free to try out some of the other links. As you become more familiar with what's available in Dashboard and as you work more with Kubernetes you may find some operations are easier to execute from the dashboard's gui than they are from a terminal using `kubectl` commands.

Part 5 - The Kubernetes API Proxy

In this part we will start up the Kubernetes API Proxy and work with the cluster by making REST calls to it. The API is setup as a REST service and can be accessed through HTTP networking calls. A REST call's URL looks like this:

```
http://server-name:port/call_uri  
http://localhost:8080/api/nodes
```

The server-name and port identify the running service instance while the `call_uri` identifies the operation being requested. This is also referred to as a service endpoint.

We will be using the linux '`curl`' command to make HTTP REST calls to the API.

In this part we'll make calls to the API proxy service and take a look at their responses.

__1. You will need two terminals, one for the proxy and another to make calls to the proxy. Open up the two terminals now and switch to the one you are going to use to run the proxy.

__2. Start the proxy:

```
kubectl proxy --port=8001
```

The output should be:

```
Starting to serve on 127.0.0.1:8001
```

Leave the proxy running here. Later when you are done you can stop it with Ctrl-C.

__3. Switch to the terminal where you are going to execute calls to the API.

__4. Lets run the command for the root endpoint "/"

```
curl http://localhost:8001/
```

The output should look like this:

```
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/logs",
    "/metrics",
    "/openapi/v2",
    "/version"
  ]
}
```

The output is fairly long, this is just a part of it. What you see here are various endpoints that can be called.

__5. Run a curl command to access the **"/version"** endpoint which returns information about the Kubernetes version.

```
curl http://localhost:8001/version
```

The output should look like this:

```
{
  "major": "1",
  "minor": "20",
  "gitVersion": "v1.20.2",
  "gitCommit": "faecb196815e248d3ecfb03c680a4507229c2a56",
  "gitTreeState": "clean",
  "buildDate": "2021-01-13T13:20:00Z",
  "goVersion": "go1.15.5",
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

This contains some of the same information that is returned by the 'kubectl version' command.

__6. The following commands get information about pods and services. Try running both commands:

```
curl http://localhost:8001/api/v1/namespaces/default/pods
```

```
curl http://localhost:8001/api/v1/namespaces/default/services
```

You'll notice that All the information about the given items are returned in JSON format. The amount of data returned is typically more than you would be interested in when interacting with the data manually. While its possible to run one-off commands manually against the API like we've been doing, the API was made primarily to enable programmatic access to the K8s API.

For most of your day to day manual command line queries the 'kubectl' tool is a better choice.

__7. Stop the proxy.

__8. Close all terminals.

Part 6 - Review

In this lab we looked at three tools for interacting with Kubernetes clusters through its API: kubectl CLI, Dashboard and API REST proxy.

Lab 10 - 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.

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
```

Output should be:

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

__2. If minikube is 'Stopped', start it with this command:

```
minikube start --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. Check minikube status again with 'minikube status', it should indicate that minikube is running.

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.

__ 5. Create the Deployment by running the following command:

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

You should get:

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

__ 6. Run the following command to verify the Deployment was created:

```
kubectl get deployments
```

You should see "nginx-deployment 3/3". You may see other deployments.

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
apache	1/1	1	1	11m
nginx	1/1	1	1	10m
nginx-deployment	3/3	3	3	8s

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
```

A similar output is returned:

NAME	READY	STATUS	RESTARTS	AGE	LABELS
apache-68cf467b84-cqjxn	1/1	Running	0	13m	app=apache,pod-template-hash=68cf467b84
nginx-55649fd747-7t8vl	1/1	Running	0	13m	app=nginx,pod-template-hash=55649fd747
nginx-deployment-5d59d67564-2wrz9	1/1	Running	0	2m40s	app=nginx,pod-template-hash=5d59d67564
nginx-deployment-5d59d67564-hrsbw	1/1	Running	0	2m40s	app=nginx,pod-template-hash=5d59d67564
nginx-deployment-5d59d67564-pjz5c	1/1	Running	0	2m40s	app=nginx,pod-template-hash=5d59d67564

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:

```
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.

```
nginx-deployment-d645d84b6-m6mts    0/1    ImagePullBackOff    0    27s
nginx-deployment-d645d84b6-zplf4    0/1    ImagePullBackOff    0    27s
```

__5. Undo the recent change:

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

__6. Verify the invalid pods are removed [it make take few minutes to be deleted]:

```
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. Close the terminal window.

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 11 - Scheduling and Node Management

Kubernetes allows us to decide how we want Pods to be loaded. We can manually set which node we want to run a pod on or let the Kubernetes scheduler decide based on certain conditions like resource availability. In this lab we will learn how to:

- Schedule pods based on the nodeSelector field.
- Run pods based on resource availability.

Part 1 - 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
```

__6. 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 v1.19.0 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=5917MB, Disk=79152MB) ...
🔧 OS release is Ubuntu 18.04.5 LTS
🔧 Preparing Kubernetes v1.20.2 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 example,

  ▪ 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
root@labvm:~#
```

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.

__7. 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
```

Part 2 - Scheduling Pods Based on nodeSelector

In this section we will setup a pod to execute on a specific node using nodeSelector. First lets take a look at the Pod specification we plan to use:

```
# label-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: label-pod
spec:
  containers:
  - name: web1
    image: nginx:latest
    ports:
    - containerPort: 80
  nodeSelector:
    quality: bestnode
```

This specification includes a nodeSelector that includes the label

```
nodeSelector:
  quality: bestnode
```

This tells the Kubernetes scheduler to place the node on a node that is tagged with the given label. By default the label shown here does not exist on the node and will not show up until we add it explicitly which we will do a bit later.

__1. Create the file 'label-pod.yaml' in the current directory:

```
gedit label-pod.yaml
```

__2. Enter the following text into the file, then save it and close the editor:

```
# label-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: label-pod
spec:
  containers:
  - name: web1
    image: nginx:latest
    ports:
    - containerPort: 80
  nodeSelector:
    quality: bestnode
```

__3. Lets Apply the specification file:

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

__4. Now check the pod's status:

```
kubectl get pods label-pod
```

Notice that the pod is pending. No matter how long you wait it will continue in this state unless something changes.

NAME	READY	STATUS	RESTARTS	AGE
label-pod	0/1	Pending	0	116s

__5. Lets check the pod for errors using the describe command:

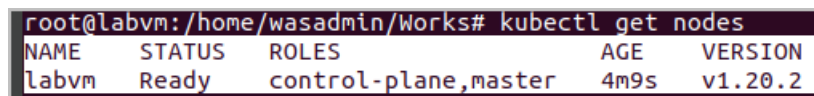
```
kubectl describe pods label-pod
```


You should see this in the commands's output:

```
Events:
  Type      Reason             Age   From                    Message
  ----      -
  Warning    FailedScheduling   48s   default-scheduler       0/1 nodes are
available: 1 node(s) didn't match node selector.
```

__6. Get the Kubernetes node list.

```
kubectl get nodes
```

A terminal window showing the command 'kubectl get nodes' and its output. The output is a table with columns: NAME, STATUS, ROLES, AGE, and VERSION. The single node listed is 'labvm' with status 'Ready', roles 'control-plane,master', age '4m9s', and version 'v1.20.2'.

NAME	STATUS	ROLES	AGE	VERSION
labvm	Ready	control-plane,master	4m9s	v1.20.2

__7. Check to see what labels the node has:

```
kubectl describe nodes labvm
```

__8. Scrolling back up to the top of the output you should see:

```
Name:          labvm
Roles:         control-plane,master
Labels:        beta.kubernetes.io/arch=amd64
               beta.kubernetes.io/os=linux
               kubernetes.io/arch=amd64
               kubernetes.io/hostname=labvm
               kubernetes.io/os=linux

minikube.k8s.io/commit=15cede53bdc5fe242228853e737333b09d4336b5
minikube.k8s.io/name=minikube
```

The node has plenty of labels but none that match the node selector from the pod:

```
nodeSelector:
  quality: bestnode
```

__9. Add the label to the node:

```
kubectl label nodes labvm quality=bestnode
```

__10. Check the node's labels again:

```
kubectl describe nodes labvm
```

Scrolling back up to the top of the output you should see:

```
Name:                labvm
Roles:               control-plane,master
Labels:              beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    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=2022_04_29T09_54_09_0700
minikube.k8s.io/version=v1.19.0
node-role.kubernetes.io/control-plane=
node-role.kubernetes.io/master=
quality=bestnode
```

The node now has the required label.

__11. Lets check on the pod again, the status should have changed:

```
kubectl get pods label-pod
```

NAME	READY	STATUS	RESTARTS	AGE
label-pod	1/1	Running	0	116s

Notice that the pod is now in the 'Running' state.

Using this method we can manually define the node or nodes we would like to run our pod.

Part 3 - Running Pods Based on Resource Availability

Some applications have specific resource needs. They may only run well with a certain amount of memory and/or cpu. In this section we will create an application that requests a specific amount of resources. This will allow the Kubernetes scheduler to only choose nodes that can satisfy the requirements to run the pods.

__1. Before we start lets delete any existing deployments and pods. The following commands should be helpful in identifying and removing these items:

```
kubectl get deployments
kubectl get pods
kubectl delete deployments {deployment-name}
kubectl delete pods {pod-name}
```

Make sure to run the delete commands for each item shown by the get commands.

__2. Check nodes for the current cluster:

```
kubectl get nodes
```

Your output should be:

NAME	STATUS	ROLES	AGE	VERSION
labvm	Ready	control-plane,master	15m	v1.20.2

Notice that minikube is the only node. If this were a production system you would most likely have several nodes.

In this lab we will make extensive the node describe command. This command outputs a lot of data so it is important that we understand how to find our way around. Data returned from the command includes the following sections:

```
Name:
Roles:
Labels:
Annotations:
CreationTimestamp:
Taints:
Unschedulable:
Conditions:
Addresses:
Capacity:
Allocatable:
System Info:
Non-terminated Pods:
Allocated resources:
Events:
```

Based on what we are looking for, we will reference one of more of the various sections.

The first thing we need to do is to check the size and CPU capabilities of the minikube node.

__3. Run the node describe command:

```
kubectl describe nodes labvm
```

We will first focus on the 'Capacity' section of the output.

The Capacity section should look like this:

```
Capacity:
  cpu:                2
  ephemeral-storage:  17784772Ki
  hugepages-2Mi:      0
  memory:              1989472Ki
  pods:               110
```

The CPU figure states that we have a total of 2 CPU cores that we can allocate to running pods. Our pods can request fractional portions of this number.

The memory figure states that we have 1989472 kilobytes.

```
1989472Ki (kilobytes)
```

Which is approximately equal to these other measurement units.

```
1989Mi (megabytes)
2Gi (gigabytes)
```

Memory that pods request must come out of and can not exceed this amount.

The Allocatable section of the output is similar to the Capacity section only it shows what has not already been used.

```
Allocatable:
  cpu:                2
  ephemeral-storage:  16390445849
  hugepages-2Mi:      0
  memory:              1887072Ki
  pods:               110
```

To see how much CPU and memory have been used we would look at the "Allocated resources" section:

```
Allocated resources:
(Total limits may be over 100 percent, i.e., overcommitted.)
Resource           Requests          Limits
-----
cpu                 755m (37%)        0 (0%)
memory              190Mi (10%)       340Mi (18%)
ephemeral-storage   0 (0%)            0 (0%)
```

And for the details of where the CPU and memory are being used we look at the 'Non-Terminated Pods' section: (some columns have been removed to make it easier to display here)

Non-terminated Pods:		(7 in total)				
Namespace	Name	CPU Requests	CPU Limits	Memory Requests	Memory Limits	Age
-----	----	-----	-----	-----	-----	---
kube-system	coredns-74ff55c5b-fk44h	100m (5%)	0 (0%)	70Mi (1%)	170Mi (2%)	16m
kube-system	etcd-labvm	100m (5%)	0 (0%)	100Mi (1%)	0 (0%)	16m
kube-system	kube-apiserver-labvm	250m (12%)	0 (0%)	0 (0%)	0 (0%)	16m
kube-system	kube-controller-manager-labvm	200m (10%)	0 (0%)	0 (0%)	0 (0%)	16m
kube-system	kube-proxy-pdfq7	0 (0%)	0 (0%)	0 (0%)	0 (0%)	16m
kube-system	kube-scheduler-labvm	100m (5%)	0 (0%)	0 (0%)	0 (0%)	16m
kube-system	storage-provisioner	0 (0%)	0 (0%)	0 (0%)	0 (0%)	16m

Notice that since we deleted all of our own deployments and pods what is left here are all pods that are part of Kubernetes itself that reside in the kube-system namespace. After creating a new deployment we will start to see some pods from the default namespace.

Lets Define a deployment with CPU and Memory requirements.

__ 4. Create the file scaler-deployment.yaml using the following command:

```
gedit scaler-deployment.yaml
```

__5. Add the following content to the file, then save it and close the editor:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: scaler-deployment
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: web1
          image: nginx:latest
          ports:
            - containerPort: 80
          env:
          resources:
            requests:
              memory: "64Mi"
              cpu: "125m"
            limits:
              memory: "128Mi"
              cpu: "250m"
```

The containers in the above specification have the following resource requests and limits:

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

Each container will take up 125/2000 of the total CPU cores and 64 Megabytes of memory.

__6. Create the deployment by executing the following command:

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

__7. Check that the deployment is up and running:

```
kubectl get deployments
```

You should see that it is ready and available:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
scaler-deployment	1/1	1	1	103s

__8. Run the following command to get the updated node information:

```
kubectl describe nodes labvm
```

Take a look at the Non-terminated Pods section. You should now see a line for pod in the default namespace from the scaler-deployment:

Non-terminated Pods:		(8 in total)		
Namespace	Name		CPU Requests	Memory Requests
-----	----		-----	-----
default	scaler-deployment-6f69dd64c9-dx964		125m (6%)	64Mi (3%)

Here 6% refers to 6% of 2 cpu cores which is:

$.12 = .6 \times 2.0 \approx 125m$

The Allocated resources section is now:

Allocated resources:		
Resource	Requests	Limits
-----	-----	-----
cpu	880m (44%)	250m (12%)
memory	254Mi (13%)	468Mi (25%)
ephemeral-storage	0 (0%)	0 (0%)

CPU request are up from 37% to 44%

Memory requests are up from 10% to 13%

__9. Check the deployment:

```
kubectl get deployment scaler-deployment -owide
```

You will see that its scaled to a single replica:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS ...
scaler-deployment	1/1	1	1	93m	web1

__10. Run the following command to scale the deployment up to 20 replicas in 1 line:

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

The output should say:

```
deployment.apps/scaler-deployment scaled
```

__11. Wait a minute and then check the state of the deployment:

```
kubectl get deployment scaler-deployment -owide
```

Not all Pods would get deployed due to the lack of sufficient resources. You numbers may vary.

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS ...
scaler-deployment	10/20	20	10	98m	web1

__12. So what's up with the pods? Lets check the pods:

```
kubectl get pods
```

You will notice, some Pods are running and others are in the Pending state.

__13. Lets take a closer look at a Pod in the pending state (change it for your id):

```
kubectl describe pods scaler-deployment-6f69dd64c9-zjrjj
```

The 'Events' section of the output says this:

```
Events:
  Type            Reason              Age             From              Message
  ----            -
  Warning         FailedScheduling    10s (x7 over 8m37s)  default-scheduler  0/1 nodes
are available: 1 Insufficient cpu.
```

Take a look at the Message field:

```
0/1 nodes are available: 1 Insufficient cpu.
```

Its saying that it can't deploy the pod because the node does not have enough memory to support it.

__14. Lets double check by looking at the node information: (Replace {node-name} with your actual node name)

```
kubectl describe nodes {node-name}
```

Take a look at the Allocated resources section. You will see cpu and memory utilization in percent. We can see that in this case that puts the requests over 100%. When the scheduler tried to schedule the last pod onto the node it would have seen this and stopped the pod from being deployed.

In systems that have more than one node the scheduler could go ahead and schedule the pod on one of the other available nodes.

__15. Scale down the deployment by running the following command.

```
kubectl scale --replicas=5 deployment scaler-deployment
```

__16. Get the Pods list to verify all Pods are running.

```
kubectl get pods.
```

You may see pods in Terminating status.

__17. Now that you are done with the lab, execute the following command to delete the deployment.

```
kubectl delete deployment scaler-deployment
```

__18. Close any Terminal.

Part 4 - Review

In this lab we learned how to:

- Schedule pods based on the nodeSelector field.
- Run pods based on resource availability.

Lab 12 - Accessing Applications

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

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

Part 1 - Setup

__1. Open a new Terminal.

In next steps we will referring to this Terminal as 'nx1'.

__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 the Cluster

__1. Check if minikube is running:

```
minikube status
```

Output should say "Running" for each component and "Correctly Configured " for kubectl.

__2. If minikube is 'Stopped', start it with this command:

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

It can take up several minutes for it to start up.

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

```
Done! kubectl is now configured to use "minikube" cluster and "default"
namespace by default
```

__3. Check minikube status again with "minikube status", it should indicate that minikube is running.

__4. Check for any existing deployments left over from previous labs and delete any that you find. How to do this should be familiar from previous labs. Commands to use are listed here for convenience:

```
kubectl get deployments
kubectl get services
kubectl get replicaset
kubectl delete deployments {deployment-name}
kubectl delete services {service-name}
kubectl delete replicaset {replicaset-name}
```

Ensure there are no existing deployments.

Part 3 - Create Deployment NX1

__1. Run the following command to create the nx1 deployment:

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

__2. Verify that the deployment has been added:

```
kubectl get deployments -owide
```

You should see:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS	IMAGES	SELECTOR
nx1	1/1	1	1	33m	nginx	nginx:latest	app=nx1

Repeat the command until you see 1/1.

___3. Check that a pod was created for the deployment and is running:

```
kubect1 get pods -owide
```

You should see:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nx1-775fd7649c-kn7kv	1/1	Running	0	34m	172.17.0.6	labvm

___4. Now we will try calling the nginx server using the curl command: (Note: Ensure you use the IP address shown on your side as part of the previous command)

```
curl http://172.17.0.6/index.html
```

The nginx web server should return the default contents of index.html.

Part 4 - Create Deployment NX2

___1. Open a new Terminal.

In next steps we will referring to this Terminal as 'nx2'.

The rest of the instructions are similar to those in the previous section except for the name 'nx2'. Also, all these instructions should be entered into the 'nx2' terminal except where otherwise noted.

___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
```

__6. Run the following command to create the nx1 deployment:

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

__7. Verify that the deployment has been added:

```
kubectl get deployments -owide
```

You should see:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS	IMAGES	SELECTOR
nx1	1/1	1	1	33m	nginx	nginx:latest	app=nx1
nx2	1/1	1	1	14s	nginx	nginx:latest	app=nx2

__8. Check that a pod was created for the deployment and is running:

```
kubectl get pods -owide
```

You should see:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nx1-775fd7649c-kn7kv	1/1	Running	0	34m	172.17.0.6	labvm
nx2-577775c7c4-pl72v	1/1	Running	0	86s	172.17.0.7	labvm

__9. Now we will try calling the nginx server using the curl command: (Note: Ensure your use the IP address shown on your side as part of the previous command)

```
curl http://172.17.0.7/index.html
```

The nginx web server should return default contents of index.html.

Part 5 - Check Current Network Situation

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

__ 1. Open a New Terminal.

We will referring to this Terminal as 'kubectl'

__ 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
```

__ 6. Check the current directory using 'pwd' you should be here:

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

__ 7. Run this command to obtain your Kubernetes node name.

```
kubectl get nodes
```

__ 8. Run this command to get information about the node that Kubernetes is running on:
(Replace {node-name} with your node name)

```
kubectl describe nodes {node-name}
```

__9. Scroll down until you come to the this line "Non-terminated Pods: ...". You should see pods in the default namespace for the two deployments, nx1 & nx2. This is evidence that the two pods are running on the same node.

Non-terminated Pods: (13 in total)

Namespace	Name	CPU Requests	CPU Limits
-----	----	-----	-----
default	nx1-775fd7649c-kn7kv	0 (0%)	0 (0%)
default	nx2-577775c7c4-pl72v	0 (0%)	0 (0%)
kube-system	coredns-5c98db65d4-5s5zq	100m (5%)	0 (0%)
kube-system	coredns-5c98db65d4-bv5gk	100m (5%)	0 (0%)

__10. Run this command to describe the nx1 pod, remember your pod names will be slightly different:

```
kubectl describe pods nx1-775fd7649c-kn7kv
```

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

```
Name:          nx1-775fd7649c-kn7kv
Namespace:     default
Node:          labvm/10.0.2.15
Status:        Running
IP:            172.17.0.6
Controlled By: ReplicaSet/nx1-775fd7649c
Containers:
  nginx:
    Container ID:  docker://915e4...c2
    Image:         nginx:latest
```

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

__11. Call the following command and check the IP address for nx1:

```
kubectl get pods -owide
```

This returns:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nx1-775fd7649c-kn7kv	1/1	Running	0	3h15m	172.17.0.6	labvm
nx2-577775c7c4-pl72v	1/1	Running	0	162m	172.17.0.7	labvm

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

Part 6 - Exposing Pods Outside the Cluster

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

- __1. Make sure you are in the terminal named '**kubectl**'.
- __2. Check that you are in the '/home/wasadmin/Works' directory using the 'pwd' command.
- __3. Create and edit a file named nx1-service.yaml using the following command:

```
gedit nx1-service.yaml
```

This will bring up the gedit gui editor.

- __4. Enter the following text into the editor:

```
apiVersion: v1  
kind: Service  
metadata:  
  name: nx1  
spec:  
  selector:  
    app: nx1  
  ports:  
  - port: 80  
    protocol: TCP  
  type: NodePort  
  externalTrafficPolicy: Cluster
```

- __5. Save the file and close the editor.
- __6. Run the following command. It will create a Kubernetes service for nx1:

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

The response should be:

```
service/nx1 created
```

___7. Check that the service exists with this command:

```
kubectl get services nx1
```

This should return:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
nx1	NodePort	10.103.137.140	<none>	80:30090/TCP	2m20s

___8. Take a note of the PORT in this case 30090

___9. Get the cluster ip:

```
minikube ip
```

It returns:

```
192.168.99.100
```

Yours will be different.

___10. Make a call to the port using the cluster ip:

```
curl http://192.168.99.100:30090
```

Replace the IP and PORT with yours.

This should return contents of the index.html file.

This is evidence of accessing nx1 from outside the cluster!

Next you will close all Terminals.

- __ 11. Close kubectl Terminal.
- __ 12. Switch to nx1 Terminal.
- __ 13. Type exit and then close the Terminal.
- __ 14. Switch to nx2 Terminal.
- __ 15. Type exit and then close the Terminal.

Part 7 - Review

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

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

Lab 13 - Using Persistent Storage

Most applications need some kind of persistent storage. Most deployments make use of Docker images that include some kind of operating system which includes a file system that can be used as persistent storage. In many cases though there are advantages to placing files that the deployment needs outside the deployed image. Kubernetes includes Volumes for this purpose.

In this lab we will deploy an application that uses files in a separate volume. We will see how to create and manage volumes from two perspectives:

- Temporary storage space (emptyDir)
- Persistent local storage (local-storage)

Part 1 - 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
```

__ 6. Check if minikube is running:

```
minikube status
```

__ 7. If minikube is stopped, start with this command:

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

__ 8. Check minikube status again with **minikube status**, it should indicate that minikube is running.

Part 2 - Using emptyDir for Temporary Volumes

Some applications require 'scratch' or 'temporary' space to place files they are working with. Work that would normally be done in memory is often done by saving data to disk while a process is running. Once a result is obtained the 'temporary' data files can be deleted. Since they were only being used as a scratch pad they do not need to be persisted.

emptyDir volumes are created when a Pod is created and destroyed when the Pod is deleted. They provide storage space that can be used while the Pod is running.

In this section we will create a pod that includes an emptyDir volume. We will open up a shell into the Pod, create and use a file. Then we will delete and recreate the pod showing that the file we created previously is now gone.

__1. Use gedit to create the file empty-dir-pod.yaml:

```
gedit empty-dir-pod.yaml
```

__2. Enter the following text into the file:

```
# empty-dir-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: empty-dir-pod
spec:
  containers:
    - image: nginx:1.21.4
      name: empty-dir-container
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

- The name of the pod is 'empty-dir-pod'
- The pod holds a container based on an nginx image
- The 'cache-volume' emptyDir volume is mounted in the container at '/cache'

__3. Save it and close the editor.

__4. Use the apply command to create the container from the yaml file:

```
kubectl apply -f empty-dir-pod.yaml
```

The resulting output should show that the pod was created.

```
root@labvm:/home/wasadmin/Works# kubectl apply -f empty-dir-pod.yaml
pod/empty-dir-pod created
```

__5. Show all existing pods

```
kubectl get pods
```

The pod 'empty-dir-pod' should show up with status=running. If status is not yet as running then wait a minute and try the get pods command again. You may see other pods.

Note. If after many time the pod fails to start, ensure you have logged into docker using "docker login -u {user-name} -p {password}".

```
root@labvm:/home/wasadmin/Works# kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
empty-dir-pod       1/1     Running   0           39s
```

__6. Open a shell into the pod:

```
kubectl exec -it empty-dir-pod -- bash
```

Your prompt should change to the container's shell prompt:

```
root@empty-dir-pod:/#
```

The commands that follow, prefaced with '#', should be executed from the container's shell prompt.

__7. List the current directory, you should see a dir named 'cache':

```
ls
```

Output should be:

```
bin boot cache dev docker-entrypoint.d ...
```

__8. List the contents of the cache dir - it should be empty.

```
ls cache
```

__9. Create a file in the cache dir as shown below:

```
echo "some file content" > cache/file1.txt
```

__10. Check for the file you just created:

```
ls cache
```

The output should be:

```
file1.txt
```

__11. Check the contents of the file you just created:

```
cat cache/file1.txt
```

Its contents should be:

```
some file content
```

__12. Exit the container's shell prompt.

```
exit
```

We've just used an attached emptyDir volume. The file we created is temporary. It should go away if we delete and recreate the Pod. Lets do that now.

Recreate the Pod

__13. Delete the pod using its name:

```
kubectl delete pod empty-dir-pod
```

Watch this command and wait for it to return before you move on.

__14. Recreate the pod with the apply command:

```
kubectl apply -f empty-dir-pod.yaml
```

The resulting output should show that the pod was created.

__15. Show all existing pods:

```
kubectl get pods
```

The pod 'empty-dir-pod' should show up with status=running. If status is not yet running then wait a minute and try the get pods command again.

__16. Open a shell into the pod:

```
kubectl exec -it empty-dir-pod -- bash
```

Your prompt should change to the container's shell prompt:

```
root@empty-dir-pod:/#
```

The commands that follow, prefaced with '#', should be executed from the container's shell prompt.

__17. List the contents of the cache dir - it should be empty:

```
ls cache
```

As you can see, the 'file1.txt' file we created before no longer exists. The attached storage is empty as we should expect from it being of type emptyDir.

__18. Close the container's shell prompt:

```
exit
```

Part 3 - Using a Persisted local-storage Type Volume

Some files created by applications are meant to stick around. They may be used multiple times before they are deleted or they may never be deleted and we need them to survive Pod restarts.

In the next few parts of this lab we will create a persisted volume. Its storage space will be used by a Pod that we create. The pod will request to use local-storage space through a persisted volume claim and Kubernetes will honor the request by identifying the persisted volume and configuring the pod to use its space.

To do that we will need to create yaml specification files for the following:

- A local-storage persisted volume claim (lpvc)
- A local-storage persisted volume (lpv)
- A pod that requests storage space using the lpvc (lpvpod)

After that we will:

- Open a shell to the pod
- Create and use a file
- Delete the pod
- Recreate the pod
- Show that the file we created still exists in the persisted volume

Part 4 - Create a Local-Storage Persisted Volume Claim

__1. Use gedit to create the file lpvc.yaml:

```
gedit lpvc.yaml
```

__2. Enter the following text into the file, then save it and close the editor:

```
# lpvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: lpv-claim
spec:
  storageClassName: local-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

- The name of the claim is 'lpv-claim'
- The claim requests 1Gi (1 gigabyte) of 'local-storage' type storage
- The volume can be mounted as read-write by a single node

__3. Save it and close the editor.

__4. Use the apply command to create the container from the yaml file:

```
kubectl apply -f lpvc.yaml
```

The resulting output should show that the pod was created.

```
root@labvm:/home/wasadmin/Works# kubectl apply -f lpvc.yaml
persistentvolumeclaim/lpv-claim created
```

__5. Show persistent volume claims:

```
kubectl get pvc
```

Note that the status=pending. Later, when a persistent volume exists to satisfy the request, the status will change to 'Bound' indicating that Kubernetes has linked the two.

```
root@labvm:/home/wasadmin/Works# kubectl get pvc
NAME          STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
lpv-claim     Pending                                     local-storage   32s
```

Part 5 - Create a Local-Storage Persisted Volume

__1. Use gedit to create the file lpv.yaml:

```
gedit lpv.yaml
```

__2. Enter the following text into the file:

```
#lpv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-volume
spec:
  capacity:
    storage: 2Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/pvol
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - example-node
                - minikube
                - labvm
```

- The size of the volume is 2 gigabytes
- It will take its space from the node's filesystem
- When a pvc is deleted the volume will also be deleted
- The type of storage used will be local-storage (on the node)
- /mnt/pvol will be the volume's location on the node (note: we will be creating the directory soon)
- The volume wants to be created on the node named 'example-node'

__3. Save it and close the editor.

__4. Use the apply command to create the persistent volume from the yaml file:

```
kubectl apply -f lpv.yaml
```

The resulting output should show that the volume was created.

```
root@labvm:/home/wasadmin/Works# kubectl apply -f lpv.yaml
persistentvolume/local-volume created
```

__5. Wait a minute or two and then show persistent volumes using this command:

```
kubectl get pv
```

The status should be 'Bound' meaning that Kubernetes has linked the claim to the volume. If shows 'available' wait few seconds and try again.

```
root@labvm:/home/wasadmin/Works# kubectl get pv
NAME             CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  CLAIM             STORAGECLASS
local-volume     2Gi       RWO           Delete          Bound   default/lpv-claim local-storage
```

__6. Check the claims using this command:

```
kubectl get pvc
```

The status of the claim should also be 'Bound'.

```
root@labvm:/home/wasadmin/Works# kubectl get pvc
NAME             STATUS  VOLUME             CAPACITY  ACCESS MODES  STORAGECLASS  AGE
lpv-claim        Bound   local-volume       2Gi       RWO           local-storage  4m10s
```

Now that we have the claim and the volume lets create a pod that uses them.

Part 6 - Create a Directory for the Volume in the Kubernetes Node

We referenced the '/mnt/pvol' directory in the lpv.yaml persistent volume specification. This directory does not exist by default on the file system of the node where minikube is running. In this part, we will create the '/mnt/pvol' directory.

__1. Open up a new terminal.

__2. Run the following command to create the '/mnt/pvol' directory:

```
sudo mkdir /mnt/pvol
```

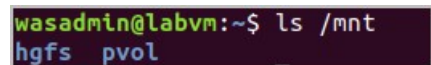
__3. Enter **wasadmin** as password.

Note: In the lab environment, Minikube is set up to run with the 'none' driver instead of running it in a VM. Therefore, Kubernetes will be access the folder that you just created without requiring SSH.

__4. Check that the directory was created:

```
ls /mnt
```

This should return:



```
wasadmin@labvm:~$ ls /mnt
hgfs  pvol
```

The node is now prepared so that the directory can be used for the persisted volume.

Part 7 - Create a Pod that Uses the Local-Storage Persisted Volume

__1. Switch to the Terminal were you are the 'root' user.

__2. Use gedit to create the file lpvpod.yaml:

```
gedit lpvpod.yaml
```

__3. Enter the following text into the file:

```
# lvpod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: lpv-storage
      persistentVolumeClaim:
        claimName: lpv-claim
  containers:
    - name: lpv-container
      image: nginx:1.21.4
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: lpv-storage
```

- The name of the pod is 'task-pv-pod'
- It requests a volume using the claim 'lpv-claim'
- It defines a container based on an nginx image
- nginx is a web server, it will be accessible from the container's port 80
- The requested volume is mounted at nginx's html dir: /usr/share/nginx/html

__4. Save it and close the editor.

__5. Use the apply command to create the pod from the yaml file:

```
sudo kubectl apply -f lvpod.yaml
```

The resulting output should show that the pod named 'task-pv-pod' was created.

```
root@labvm:/home/wasadmin/Works# sudo kubectl apply -f lvpod.yaml
pod/task-pv-pod created
```

__6. Wait a minute or two and then show pods using this command:

```
kubectl get pods
```

The status of the pod should be 'Running'.

```
root@labvm:/home/wasadmin/Works# kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
empty-dir-pod  1/1     Running   0           20m
task-pv-pod    1/1     Running   0           51s
```

__7. Check the claims using this command:

```
kubectl get pvc
```

The status of the claim should be 'Bound'.

```
root@labvm:/home/wasadmin/Works# kubectl get pvc
NAME          STATUS   VOLUME      CAPACITY   ACCESS MODES   STORAGECLASS
lpv-claim     Bound    local-volume  2Gi        RWO             local-storage
```

Now that we have the claim and the volume lets create a pod that uses them.

Part 8 - Add a File to Persistent Volume and Check after Pod Restart

Now that we have a Pod running with a Persistent Local Volume we can try adding a file to the volume and check that it still exists after deleting and recreating the pod. To do this we will follow these steps:

- Shell into the Pod
- Create a file
- Check that the file is being used by the application (nginx)
- Exit the shell and delete the pod
- Recreate the pod
- Shell into the pod
- Check for the file we created earlier

__1. Shell into the pod:

```
kubectl exec -it task-pv-pod -- bash
```

The prompt should change to:

```
root@task-pv-pod:/#
```

__2. Check the persistent volume mount point:

```
ls /usr/share/nginx/html
```

The directory should currently be empty.

__3. Use the echo command to create a file in the mount point directory (type the command all on one line):

```
echo "welcome to nginx index.html on persistent volume" >  
/usr/share/nginx/html/index.html
```

The above command placed index.html in the mount point directory which happens to also be the nginx web server's root directory. That means that we should be able to send a request to the web server that returns the file.

__4. Execute the following command to access the file via the web server:

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

You should get the following response:

```
welcome to nginx index.html on persistent volume
```

__5. The file should also be available (as the default web server response) using this command:

```
curl http://localhost
```

Everything looks good so far.

__6. Exit the pod's shell:

```
exit
```

Next we want to stop the Pod. Then we will recreate it pointing to the same persistent volume and see if the file is still there.

Part 9 - Check for the File after Deleting and Recreating the Pod

Now delete and recreate the pod and then check for the file again. Because we are using a persistent volume it should still be there. Here are the steps:

- Delete the pod
- Recreate the pod
- Shell into the pod and try accessing the file

__1. Execute the following command to delete all the resources you provisioned in this lab. (Note: Don't forget the dot at the end of the command)

```
kubectl delete -f .
```

Wait for the command to be completed. You may see some not found errors.

__2. Check that they have all been deleted:

```
kubectl get pv
```

```
kubectl get pvc
```

```
kubectl get pods
```

You may see pods from other labs.

__3. Run the apply commands again to recreate the pod, claim and volume object:

```
kubectl apply -f lpv.yaml
kubectl apply -f lpvc.yaml
kubectl apply -f lpvpod.yaml
```

Now we'll check to see if the file we created is still there.

__4. Shell into the pod:

```
kubectl exec -it task-pv-pod -- bash
```

The prompt should change to:

```
root@task-pv-pod:/#
```

__5. Check the persistent volume mount point:

```
ls /usr/share/nginx/html
```

The directory should show that the index.html file is in there.

```
root@task-pv-pod:/# ls /usr/share/nginx/html
index.html
```

__6. Execute the following command to access the file via the web server:

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

You should get the following response:

```
root@task-pv-pod:/# curl http://localhost/index.html
welcome to nginx index.html on persistent volume
```

This is the proof that the persistent volume did not get deleted or otherwise go away when the pod was removed and replaced!

__7. Exit the pod's shell:

```
exit
```

__8. Close all Terminals.

Part 10 - Review

In this lab we created pods that accessed different types of volumes: one temporary and the other persistent. In both cases we proved that the volumes were acting as expected when it comes to persistence.

Lab 14 - Getting Started with Helm

In this lab we will install Helm and start working with Helm charts. We will pull a preexisting chart from a repository and run it in Kubernetes using Helm commands. Using this chart template allows us to speed up the changes required when deploying our system. In addition, since we will make changes to it, it ensures those changes will be successful with the current Kubernetes setup and can easily upgrade the release using another helm command.

We will learn to:

- Install helm
- Pull a chart from a repository
- Install a chart on Kubernetes
- Make changes to a chart
- Upgrade a release with the new chart
- Test our changes

Part 1 - Setup

Before we install and start working with Helm we to have a clean Kubernetes cluster to work with.

In this part you will delete any existing cluster and start up minikube with a fresh cluster.

__1. Open a new Terminal.

__2. Switch to sudo:

```
sudo -i
```

__3. Enter **wasadmin** as password.

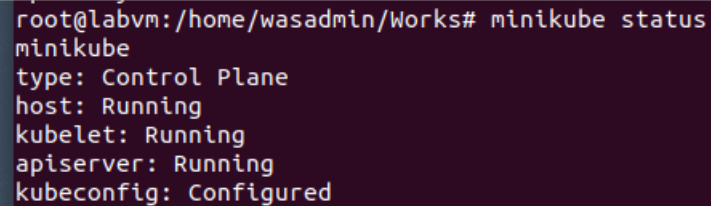
__4. Verify the current user with the command '**whoami**', it should be 'root'.

__5. Navigate to the working directory:

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

__6. Check if minikube is running:

```
minikube status
```

A terminal window with a dark purple background. The prompt is 'root@labvm:/home/wasadmin/Works#'. The command 'minikube status' has been entered and the output is displayed: 'minikube', 'type: Control Plane', 'host: Running', 'kubelet: Running', 'apiserver: Running', and 'kubeconfig: Configured'.

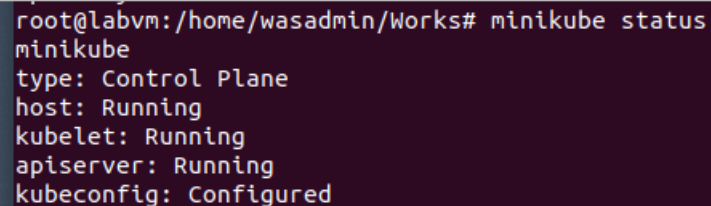
```
root@labvm:/home/wasadmin/Works# minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

__7. If minikube is stopped, run it with this command:

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

__8. Check minikube status again, it should indicate that minikube is running:

```
minikube status
```

A terminal window with a dark purple background. The prompt is 'root@labvm:/home/wasadmin/Works#'. The command 'minikube status' has been entered and the output is displayed: 'minikube', 'type: Control Plane', 'host: Running', 'kubelet: Running', 'apiserver: Running', and 'kubeconfig: Configured'.

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

Part 2 - Install Helm

In this part we will install Helm.

__1. Execute the following command to download the Helm install script. This will connect to GitHub and download to our local machine the required files. The whole command should appear on a single line (no line breaks):

```
curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3
```

The command copies the file "get_helm.sh" into the current directory.

__2. In Linux we need to make it executable. Set permissions on the script and then use ./ in front to the file name to run it:

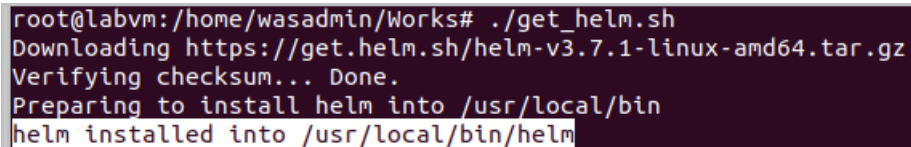
```
chmod 700 get_helm.sh
```

```
./get_helm.sh
```

You will see a message "Downloading..."

When the script is done it will output:

```
helm installed into /usr/local/bin/helm
```

A terminal window with a dark background showing the output of the ./get_helm.sh script. The output includes the download URL, checksum verification, and the final installation path.

```
root@labvm:/home/wasadmin/Works# ./get_helm.sh
Downloading https://get.helm.sh/helm-v3.7.1-linux-amd64.tar.gz
Verifying checksum... Done.
Preparing to install helm into /usr/local/bin
helm installed into /usr/local/bin/helm
```

__3. Check that helm is installed correctly with this command:

```
helm version
```

This should output: (If you do not see this inform your instructor)

```
root@labvm:/home/wasadmin/Works# helm version
version.BuildInfo{Version:"v3.7.1", GitCommit:"1d11fcb5d3f3bf00dbe6fe31b8412839a96b3dc4", GitTreeState:
"clean", GoVersion:"go1.16.9"}
```

Version may vary. Helm supports a variety of commands. The --help command displays a helm command reference.

__4. Run the following command to show the helm command reference:

```
helm --help
```

Some common commands are listed near the top:

```
helm search:  search for charts
helm pull:    download a chart to your local directory to view
helm install: upload the chart to Kubernetes
helm list:    list releases of charts
```

A more complete list of commands is listed further down in the "Available Commands" section:

```
Available Commands:
  completion  generate autocompletions script for the specified shell
  create      create a new chart with the given name
  dependency  manage a chart's dependencies
  env         helm client environment information
  get         download extended information of a named release
  ...
```

In the rest of this lab we will be working with many of these commands.

Part 3 - Searching for Charts

In this section we will search for existing helm charts. Shared Chart repositories can be searched from the hub.helm.sh site or using the 'helm search hub' command. Repositories can be associated with a local helm installation using the 'helm repo add' command. Once a repository has been added locally you will also be able to search using the 'helm search repo' command.

__1. Enter the following helm search command:

```
helm search repo 'web server'
```

You should see the output:

```
Error: no repositories configured
```

__2. Try listing the local repositories:

```
helm repo list
```

You should see the output:

```
Error: no repositories to show
```

That's because the helm install does not provide any repositories by default.

__3. Lets search the hub.helm site with the same keywords:

```
helm search hub 'web server'
```


You should see a list.

```
root@labvm:/home/wasadmin/Works# helm search hub 'web server'
URL                                CHART VERSION  APP VERSION  DESCRIPTION
https://artifacthub.io/packages/helm/http-folder... 2.0.0          1.0.0        A simple web server with file upload and download
https://artifacthub.io/packages/helm/sagikazarm... 0.0.14         2.4.5        A powerful, enterprise-ready, open source web s...
https://artifacthub.io/packages/helm/linzhengen... 0.1.7          0.1.0        A web server Helm chart for Kubernetes
https://artifacthub.io/packages/helm/halkeye/cops 1.0.1          1.1.3-ls86   Calibre OPDS (and HTML) PHP Server - web-based ...
https://artifacthub.io/packages/helm/rocketchat... 3.1.0          3.14.5       Prepare to take off with the ultimate chat plat...
https://artifacthub.io/packages/helm/cloudve/cl... 0.5.1          2.0.2        A Helm chart for CloudLaunch, including the ser...
https://artifacthub.io/packages/helm/yggyq2/de... 1.0.0          3.4.0        Chart for the nginx server
https://artifacthub.io/packages/helm/cloudnativ... 3.0.1          9.0.20       Chart for Apache Tomcat
https://artifacthub.io/packages/helm/bitnami/to... 9.6.1          10.0.13      Chart for Apache Tomcat
https://artifacthub.io/packages/helm/bitnami-ak... 9.6.1          10.0.13      Chart for Apache Tomcat
```

__4. Open a web browser to:

`https://hub.helm.sh/charts/http-folder/http-folder`

The screenshot shows the Artifact Hub web interface for the 'http-folder' Helm chart. The top navigation bar includes the 'ArtifactHUB' logo, a search bar, and links for 'STATS', 'SIGN UP', and 'SIGN IN'. The main content area features the chart's icon, name 'http-folder', and its description: 'A simple web server with file upload and download'. Below this, there is a section for 'http-folder' with a link to the README. On the right side, there are buttons for 'INSTALL', 'TEMPLATES', 'DEFAULT VALUES', and 'CHANGELOG'. The 'Star' count is shown as 0.

The URL shown here points to a chart page on the hub.helm.sh web site.

__5. Lets use this to add the chart's repository to our local helm installation. Execute this command in your terminal:

```
helm repo add http-folder https://aureliengasser.github.io/charts
```

You should see the output:

```
"http-folder" has been added to your repositories
```

__6. Check the local helm repositories with the following command:

```
helm repo list
```

You should see the output:

NAME	URL
http-folder	https://aureliengasser.github.io/charts

__7. Try searching for a chart again locally:

```
helm search repo 'web server'
```

You should see the output:

```
root@labvm:/home/wasadmin/Works# helm search repo 'web server'
NAME                CHART VERSION  APP VERSION     DESCRIPTION
http-folder/http-folder 2.0.0          1.0.0           A simple web server with file upload and download
```

If we wanted to, we could install the chart now as-is using the 'helm install' command. Instead we are going to 'pull' the chart and all of its files down to a local directory where we can take a look at it and make modifications if needed.

__8. Execute the following command. It will create an 'http-folder' directory in the current directory:

```
helm pull http-folder/http-folder --untar
```

(without the -untar flag this command downloads a *.tgz archive of the chart)

__9. List the contents of the http-folder directory:

```
ls http-folder
```

__10. The output you find in the http-folder directory has the following structure:

```
http-folder/
├── Chart.yaml
├── README.md
├── templates
│   ├── deployment.yaml
│   ├── helpers.tpl
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── pvc.yaml
│   ├── serviceaccount.yaml
│   └── service.yaml
└── values.yaml
```

The templates directory contains templates that along with values in the values.yaml file are used by helm to create yaml files that are then applied to Kubernetes to create deployments, services, etc.

__11. To find out what the Kubernetes yaml files will look like run the following command:

```
helm template http-folder > http-folder.txt
```

__12. To open the file we can use a Graphical Linux Editor, like gedit, vim or nano. We will use gedit on the http-folder.txt file that was just created in a text editor:

```
gedit http-folder.txt
```

__13. Scroll down to line 54 in the file. You should see a line that details the following "# Source: http-folder/templates/deployment.yaml".

This section shows the effective K8s deployment yaml that results from Helm processing the deployment.yaml template along with configuration values.

```
# Source: http-folder/templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: RELEASE-NAME-http-folder
  labels:
    app.kubernetes.io/name: http-folder
    helm.sh/chart: http-folder-2.0.0
    app.kubernetes.io/instance: RELEASE-NAME
    app.kubernetes.io/version: "1.0.0"
    app.kubernetes.io/managed-by: Helm
spec:
  replicas: 1
  ...
```

In the next section we will be installing this chart.

__14. Close the file.

Part 4 - Installing the Chart

In this section we will install the chart and try accessing the application.

__1. Execute the following command to install the chart and give it the release name 'web1'. This gives us a way to reference the application later, similar to adding a label.

```
helm install web1 http-folder
```

You should see the output:

```
root@labvm:/home/wasadmin/Works# helm install web1 http-folder
NAME: web1
LAST DEPLOYED: Thu Dec  2 09:28:41 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l "app.kubernetes.io/name=http-folder,app.kubernetes.io/instance=web1" -o jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl port-forward $POD_NAME 8080:80
```

__2. Verify that Helm now knows about this new release:

```
helm list
```

You should see the output:

```
root@labvm:/home/wasadmin/Works# helm list
NAME      NAMESPACE    REVISION    UPDATED                               STATUS    CHART          APP VERSION
web1      default      1           2021-12-02 09:28:41.013584104 -0500 EST deployed  http-folder-2.0.0  1.0.0
```

'Revision' is the revision number of the 'web1' release. Since we have only just created it, there are no updates as of yet, so 1 makes perfect sense here.

The 'app version' is the version of the chart.

'2.0.0' is the release number of the http-folder application.

'Release' means that the chart has been applied to the local Kubernetes installation. We should see evidence of the release if we check the cluster.

__3. Check the Kubernetes cluster using these commands:

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

You will find a deployment named '**web1-http-folder**'

A service named '**web1-http-folder**'

A pod named '**web1-http-folder-#####**'

```
root@labvm:/home/wasadmin/Works# kubectl get deployments
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
web1-http-folder    1/1     1            1           3m35s
root@labvm:/home/wasadmin/Works# kubectl get services
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
kubernetes          ClusterIP   10.96.0.1     <none>        443/TCP    104d
web1-http-folder    ClusterIP   10.107.215.46 <none>        80/TCP     3m35s
root@labvm:/home/wasadmin/Works# kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
web1-http-folder-7985d4fcd9-jw55d  1/1     Running   0          3m36s
```

Actually the number at the end of the pod name that you see will differ slightly from the one above but should still start with 'web1-http-folder'. Note that you may see more deployments, services and pods from previous labs.

These were all created by Helm from the information in the chart.

4. Lets take a closer look. Execute the following command to show more detailed information about the service:

```
kubectl get services -o wide
```

You should see output similar to this:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	SELECTOR
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	4h	<none>
web1-http-folder	ClusterIP	10.107.215.46	<none>	80/TCP	12m	app.kube...

The application can be accessed from inside the cluster at the address:

http://10.107.215.46 (yours will be different)

Server IP is 10.107.215.46 (yours will be different)

Port number is 80

__5. Try accessing the server application (use the IP address in your terminal):

```
curl http://10.107.215.46
```

At this point all you will see is a pair of empty brackets: []

This is showing the contents of the folder that is currently being served and which is currently empty.

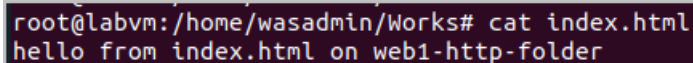
__6. We will use the application's file upload capability to add a file. Run the following command to create a file that can be uploaded:

```
echo "hello from index.html on web1-http-folder" > index.html
```

Here we are setting some simple text to be returned via the index.html file on the http server.

__7. Check that the file was created:

```
cat index.html
```



```
root@labvm:/home/wasadmin/Works# cat index.html
hello from index.html on web1-http-folder
```

__8. Post the file to the server application using the following curl command:

(remember to use the ip address you got from the services command above)

```
curl --data-binary "@index.html" http://10.107.215.46/index.html
```

You should see the output:

```
Uploaded successfully
```

__9. Now try accessing the server again:

```
curl http://10.107.215.46
```

This time you should get:

```
["index.html"]
```

__10. Now that the server has a file to server lets try and access it:

```
curl http://10.107.215.46/index.html
```

You should get:

```
hello from index.html on web1-http-folder
```

We have verified that the server application is up and running and is working as expected. We accessed it by opening a shell inside the cluster and using the server's cluster ip address. This won't work once we've exited the shell. We could replace the current service being used with a new one that exposes the application outside the cluster but since we are working with Helm there is another way which we will try in the next section.

Part 5 - Modifying the Chart and Upgrading the Release

In this section we will fix our problem of not being able to access the service outside the cluster by making a modification to the chart and using helm to push the modification to Kubernetes.

We are going to change the service type to NodePort. This should allow us to access the service from outside the cluster. Ensure you use the proper case for the NodePort name.

__1. Open the chart's values.yaml file in a text editor:

```
gedit http-folder/values.yaml
```


__2. Find the following section in the file:

```
service:
  type: ClusterIP
  port: 80
```

__3. Change service.type to NodePort:

```
service:
  type: NodePort
  port: 80
```

__4. Save the file and exit the editor.

Now that we've changed the chart we should update its version.

__5. Open the chart's Chart.yaml file in a text editor:

```
gedit http-folder/Chart.yaml
```

__6. Find and change appVersion as follows:

```
change: appVersion: 1.0.0
       to: appVersion: 2.0.0
```

__7. Save the file and exit the editor.

Now we can update the release. But before we do let's check a few things.

__8. Check the current service:

```
kubectl get services web1-http-folder
```

You should see the output:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
web1-http-folder	ClusterIP	10.107.215.46	<none>	80/TCP	17m

Note how the TYPE is ClusterIP. This should change once we've made the update.

__9. Check the release information in helm:

```
helm list
```

You should see the output:

NAME	NAMESPACE	REVISION	UPDATED	STATUS	CHART	APP VERSION
web1	default	1	2021-..	deployed	http-folder-2.0.0	1.0.0

Notice the revision = 1 and the app-version = 1.0.0

Now lets upgrade the release to include the changes we made to the helm chart.

__10. Run the following command to do the upgrade:

```
helm upgrade web1 http-folder
```

You should see the output:

```
Release "web1" has been upgraded. Happy Helming!
```

```
root@labvm:/home/wasadmin/Works# helm upgrade web1 http-folder
Release "web1" has been upgraded. Happy Helming!
NAME: web1
LAST DEPLOYED: Thu Dec  2 09:52:13 2021
NAMESPACE: default
STATUS: deployed
REVISION: 2
TEST SUITE: None
```

__11. Now lets check the release information in helm:

```
helm list
```

You should see the output:

NAME	NAMESPACE	REVISION	UPDATED	STATUS	CHART	APP VERSION
web1	default	2	2021-..	deployed	http-folder-2.0.0	2.0.0

Notice the changes from before, the revision = 2 and the app-version = 2.0.0

__12. We should also check the service, use the following command:

```
kubectl get services web1-http-folder
```

You should see the output:

```
root@labvm:/home/wasadmin/Works# kubectl get services web1-http-folder
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)          AGE
web1-http-folder    NodePort    10.107.215.46 <none>       80:30993/TCP     26m
```

Note how the TYPE shows the change we made and is now **NodePort**. Note also that the PORT setting now includes the external port number **30993** (yours maybe different). These changes will allow us to access the service from outside the cluster.

__13. First we need to get the external ip of the cluster:

```
minikube ip
```

Your output should be similar to:

```
192.168.153.149
```

__14. Run the following command to create a file that can be uploaded:

```
echo "hello from index.html on web1-http-folder" > index.html
```

__15. Check that the file was created:

```
cat index.html
```

```
root@labvm:/home/wasadmin/Works# cat index.html
hello from index.html on web1-http-folder
```

__16. Post the file to the server application using the following curl command:

(remember to use the external ip address you got from the command above)

```
curl --data-binary "@index.html" http://192.168.153.149:30993/index.html
```

You should see the output:

```
Uploaded successfully
```

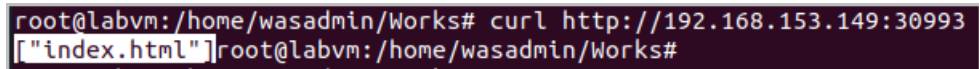
```
root@labvm:/home/wasadmin/Works# curl --data-binary "@index.html" http://192.168.153.149:30993/index.html
Uploaded successfullyroot@labvm:/home/wasadmin/Works#
```

__17. Now try accessing the server:

```
curl http://192.168.153.149:30993
```

You should get:

```
["index.html"]
```

A terminal window with a dark background. The prompt is root@labvm:/home/wasadmin/Works#. The command curl http://192.168.153.149:30993 is entered. The output is ["index.html"]root@labvm:/home/wasadmin/Works#.

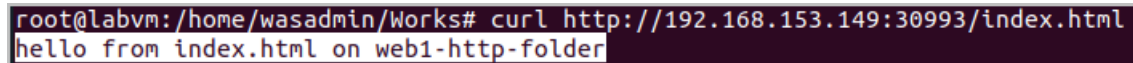
```
root@labvm:/home/wasadmin/Works# curl http://192.168.153.149:30993
["index.html"]root@labvm:/home/wasadmin/Works#
```

__18. Now that the server has a file to server lets try and access it:

```
curl http://192.168.153.149:30993/index.html
```

You should get the file output:

```
hello from index.html on web1-http-folder
```

A terminal window with a dark background. The prompt is root@labvm:/home/wasadmin/Works#. The command curl http://192.168.153.149:30993/index.html is entered. The output is hello from index.html on web1-http-folder.

```
root@labvm:/home/wasadmin/Works# curl http://192.168.153.149:30993/index.html
hello from index.html on web1-http-folder
```

This shows that we have accessed the application we created using Helm from outside the cluster.

__19. Clean up Kubernetes by running the following Helm uninstall command:

```
helm uninstall web1
```

You should get:

```
release "web1" uninstalled
```

__20. Check Kubernetes with the following commands. All of the deployments, pods and services that were created for the web1 release should have been removed [may take time to delete them]:

```
kubect1 get deployments
kubect1 get services
kubect1 get pods
```

__21. Type 'exit' many times until the Terminal is closed.

__22. Close all.

Part 6 - Review

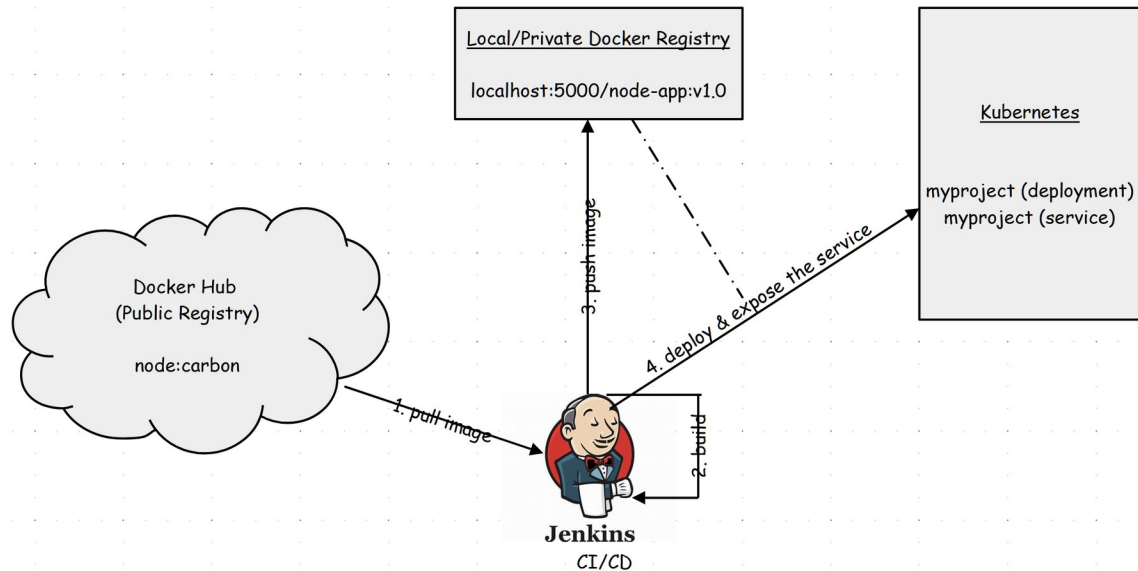
In this lab we learned how to:

- Install helm
- Pull a chart from a repository
- Install the chart on Kubernetes
- Make changes to the chart
- Upgrade the release with the new chart
- Test our changes

Lab 15 - Build CI Pipeline with Jenkins

In this lab, we will implement Continuous Integration(CI) with Jenkins, Docker, and Kubernetes. You will implement CI/CD using Jenkins to create a custom Docker image with a Node.js application, host it on our local/private Docker registry, and then deploy it in Kubernetes.

Here's the high-level architecture diagram.



This lab includes the following sections:

1. Setup
2. Create a Node.js Application
3. Check Code into GIT
4. Host a Local/Private Docker Registry
5. Create a Docker Image
6. Manually Deploy the Image
7. Expose the node-app Service and Access it
8. Configure Jenkins
9. Create a Jenkins Job
10. Test the Jenkins Job

This Lab currently does not work with a proxy server. If you can't run this Lab because you're network uses a proxy server then the instructor will demonstrate the Lab.

Part 1 - Setup

__1. Open a new Terminal.

In next steps we will referring to this Terminal as 'nx1'.

__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. Stop minikube:

```
minikube stop
```

You should see:

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

But you may see a timeout message.

__6. Delete minikube:

```
minikube delete
```

It may take a while to complete:

```
root@labvm:~# minikube delete
🔄 Uninstalling Kubernetes v1.20.2 using kubeadm ...
🔥 Deleting "minikube" in none ...
💀 Removed all traces of the "minikube" cluster.
root@labvm:~#
```


__7. Start minikube:

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

It may take a while to complete.

Part 2 - Create a Node.js Application

In this part, you will create a Node.js application. Later in the lab, you will create a Docker image hosting the Node.js service.

__1. Run the following command:

```
mkdir -p /var/lib/jenkins/repos/hello-node
```

__2. Switch to the directory:

```
cd /var/lib/jenkins/repos/hello-node
```

__3. Initialize a new node application:

```
npm init -y
```

__4. Download express node module:

```
npm install express --save
```

Express is node module used to create network applications. We will use it in the app we are creating.

__5. Create index.js file:

```
gedit index.js
```

__ 6. Enter the following JavaScript code:

```
'use strict';

const express = require('express');
// Constants
const PORT = 9090;
const HOST = '0.0.0.0';

// App
const app = express();
app.get('/', (req, res) => {
  res.send('Hello Node.js v1.0\n');
});

app.listen(PORT, HOST);
```

This is a basic network application. When a network connection comes in the application replies with the text: 'Hello Node.js v1.0'.

__ 7. Click the **Save** button to save the file. Then close gedit.

__ 8. Run the service:

```
node index.js
```

The command uses the console but does not output any messages. It may look like it is hanging but should still work fine.

__9. Open a new terminal and try calling the service using Curl as shown below:

```
curl http://localhost:9090
```

The output should be:

```
Hello Node.js v1.0
```

__10. Close this Terminal.

__11. Go back to the previous terminal and stop the application by pressing **CTRL+C**.

Now we have an application to work with. The next thing we'll do is create a Docker image to containerize the app.

Part 3 - Create the Dockerfile script for your custom application

In this part, you will write the Dockerfile script that will be used to generate the Docker image of your Node.js service.

__1. Create Dockerfile by executing the following command:

```
gedit Dockerfile
```

Note: D in Dockerfile must be uppercase.
--

__2. Enter the following text into the Dockerfile in the editor. To save time, you can skip the comment lines that start with #:

```
FROM node:carbon

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies. Note: Don't forget the period at the end
COPY package.json .

RUN npm install

# Bundle app source. Note: There's period space period at the end
COPY . .

EXPOSE 9090
CMD [ "node", "index.js" ]
```

__3. Save the file and close gedit.

__4. Create a .dockerignore file:

```
gedit .dockerignore
```

Note: There's a period in front of dockerignore. This file specifies files and directories which shouldn't become part of a docker image.

__5. Enter the following text:

```
node_modules
npm-debug.log
```

node_modules contains node.js packages which are equivalent to binaries in other technologies. We don't want these packages to be deployed in a docker container. These packages will be installed in the container by the *npm install* command.

Also, *Node Package Manager* debug file (*npm-debug.log*) should also not be deployed to the container.

__6. Save the file and close gedit.

Part 4 - Check Code into Git Repository

In this part, you will check in the code into the Git repository. You will utilize the repository in Jenkins, later in the lab.

__ 1. Make sure are using the Terminal where you were logged as root.

__ 2. Ensure you are under */var/lib/jenkins/repos/hello-node* directory:

```
pwd
```

__ 3. Create .gitignore file:

```
gedit .gitignore
```

Note: There's a period in front of gitignore. In a bit, you will check in your app source code into a git repository, so you can utilize it in Jenkins. .gitignore specifies files and directories which shouldn't get checked into the repository.

__ 4. Enter the following text:

```
node_modules  
npm-debug.log
```

__ 5. Save the file and close gedit.

__ 6. Initialize a Git repository:

```
git init .
```

```
root@labvm:/var/lib/jenkins/repos/hello-node# git init .  
Initialized empty Git repository in /var/lib/jenkins/repos/hello-node/.git/  
root@labvm:/var/lib/jenkins/repos/hello-node#
```

__ 7. Enter these commands:

```
git config --global user.name "webage"  
git config --global user.email "webage@webage.com"
```

__ 8. Stage changed files:

```
git add .
```

__9. Commit the changes:

```
git commit -m "nodejs microservice v1.0"
```

```
root@labvm:/var/lib/jenkins/repos/hello-node# git commit -m "nodejs microservice v1.0"
[master (root-commit) 3dc0573] nodejs microservice v1.0
6 files changed, 422 insertions(+)
create mode 100644 .dockerignore
create mode 100644 .gitignore
create mode 100644 Dockerfile
create mode 100644 index.js
create mode 100644 package-lock.json
create mode 100644 package.json
```

Part 5 - Building and Pushing the Custom Docker Image in to a Private Docker Registry

In this part, you will perform the following operations:

- Run a custom/private Docker registry.
- Build the Docker image of your custom Node.js service
- Push the Docker image in to your private hosted Docker registry

__1. **Open a new Terminal.**

__2. Switch to the root user by executing the following command: (Note: use wasadmin as the password if prompted)

```
sudo -i
```

__3. Execute the following command host a private local Docker registry.

```
docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

In the real world, it should be started on a separate machine and all Docker images should be pushed in to it.

You can use Artifactory or any other Docker registry solution. In this lab, you will use Docker's registry image to host the registry.

Wait until the above command completes and you are returned the prompt.

__4. Switch to the directory:

```
cd /var/lib/jenkins/repos/hello-node
```

__5. Run the following command to build a custom image and tag it so it can later be pushed in to your local Docker registry.

```
docker build -t localhost:5000/node-app:v1.0 .
```

Make sure you add the dot at the end of the command.

You should see:

```
Step 6/7 : EXPOSE 9090
--> Running in 3d21b6bf0ad3
Removing intermediate container 3d21b6bf0ad3
--> fc7db97457d7
Step 7/7 : CMD [ "node", "index.js" ]
--> Running in d91ddb839391
Removing intermediate container d91ddb839391
--> 1d6d6312436b
Successfully built 1d6d6312436b
Successfully tagged localhost:5000/node-app:v1.0
```

__6. After the above command completes, verify the custom image exists:

```
docker images
```

You should see:

```
root@labvm:/var/lib/jenkins/repos/hello-node# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost:5000/node-app	v1.0	1d6d6312436b	53 seconds ago	905MB
httpd	2.4	8362f2615893	2 days ago	144MB
httpd	latest	8362f2615893	2 days ago	144MB

__7. Execute the following command to push the Docker image in to your local Docker registry.

```
docker push localhost:5000/node-app:v1.0
```

Part 6 - Manually Deploy the Image

In this part, you will manually deploy the custom docker image and verify your custom service is working in Kubernetes. After testing the manual deployment, you will automate CI/CD using Jenkins later in the lab.

__1. Switch to the first terminal where you logged in as *root*.

__2. Change to the */var/lib/jenkins/repos/hello-node* directory.

```
cd /var/lib/jenkins/repos/hello-node
```

__3. Create a new application by deploying your custom docker image [enter the command in 1 line]:

```
kubectl create deployment myproject --image="localhost:5000/node-app:v1.0"
```

You should see:

```
deployment.apps/myproject created
```

__4. Check out deployments, services and pods. Note the name of the service and check for Ready status for the pod:

```
kubectl get deployments
```

Wait and repeat the command until you see 1/1.

```
root@labvm:/var/lib/jenkins/repos/hello-node# kubectl get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
myproject     1/1     1            1           54s
```


Part 7 - Expose the node-app Service and Access it

In this part, you will expose node-app service and access it via the web browser.

__1. Run the following command to create a service to expose the application outside the cluster:

```
kubectl expose deployment myproject --type=LoadBalancer --port=9090
```

You should see:

```
service/myproject exposed
```

__2. Check services, note that the new service has the same name as the underlying deployment 'myproject':

```
kubectl get services
```

The output looks like this [your IP will be different]:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	26h
myproject	NodePort	10.101.214.73	<pending>	9090:32214/TCP	16s

__3. Run the following command to the external URL for the service:

```
minikube --url=true service myproject
```

The output should look similar to this [your IP will be different]:

```
http://192.168.99.100:32214
```

The port number is auto-generated. It's useful when you use the NodePort option when exposing the service.

Since you used the LoadBalancer option, you can use the custom 9090 port.

__4. Use curl and the url you just obtained to call the myproject service:

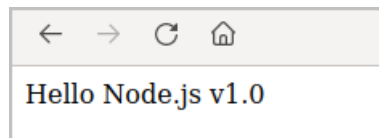
```
curl http://192.168.99.100:32214
```

The output should be:

```
Hello Node.js v1.0
```

We have successfully accessed the service from outside the cluster.

__5. Try opening up the URL in Firefox (inside the VM). We should see the same output as above. In later lab steps we will refer to it as the "Hello Node Tab" in firefox.



Part 8 - Configure Jenkins

Now you are gearing towards automating the CI/CD solution using Jenkins. You need to configure Jenkins before you can work on the actual CI/CD part. In this part, you will connect to Jenkins and configure it so you can create jobs which can deploy to Kubernetes. You will also run some commands so Jenkins has the access Docker and also to the Kubernetes cluster.

__1. Open Firefox and enter the following URL:

```
https://webage-downloads.s3.amazonaws.com/LabFiles/jenkins-configure.txt
```

__2. The following content will open in the browser:

```
echo net.ipv4.ip_forward=1 >> /etc/sysctl.d/enable-ip-forward.conf

export
PATH=$PATH:/bin:/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/
bin:/sbin:/home/wasadmin/.local/bin:/home/wasadmin/bin

cp -R /root/.kube /home/wasadmin
cp -R /root/.minikube /home/wasadmin
chown -R wasadmin /home/wasadmin/.kube
chown -R wasadmin /home/wasadmin/.minikube
chmod -R +777 /home/wasadmin/.kube
chmod -R +777 /home/wasadmin/.minikube
usermod -a -G docker jenkins
systemctl restart jenkins
```

__3. Select all and copy its contents to the clipboard.

__4. In the terminal where you logged in as root, paste and execute the clipboard contents. Make sure to execute all the commands.

__5. Open the Kubernetes configuration file for editing by executing the following command:

```
gedit /home/wasadmin/.kube/config
```

__6. Find the following three lines and make changes as shown in bold.

```
certificate-authority: /home/wasadmin/.minikube/ca.crt

client-certificate:
/home/wasadmin/.minikube/profiles/minikube/client.crt

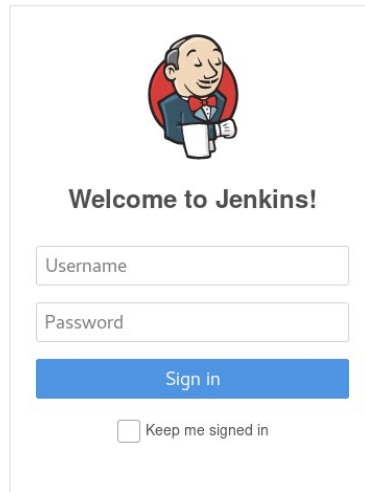
client-key: /home/wasadmin/.minikube/profiles/minikube/client.key
```

__7. Save it and close the editor.

__8. Open a new tab in the web browser and enter the following URL:

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

Notice Jenkins login page shows up.

The image shows the Jenkins login page. At the top is the Jenkins logo, a cartoon character with a red bow tie. Below the logo is the text "Welcome to Jenkins!". Underneath is a form with two input fields: "Username" and "Password". Below these fields is a blue "Sign in" button. At the bottom of the form is a checkbox labeled "Keep me signed in".

Welcome to Jenkins!

Username

Password

Sign in

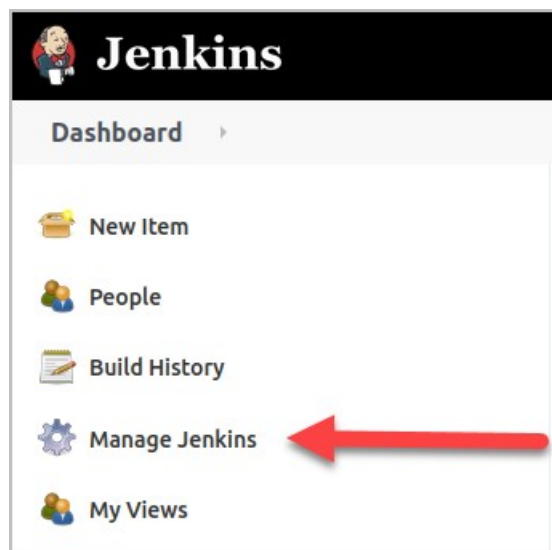
☐ Keep me signed in

__9. Enter the following credentials to log in:

User Name: wasadmin
Password: wasadmin

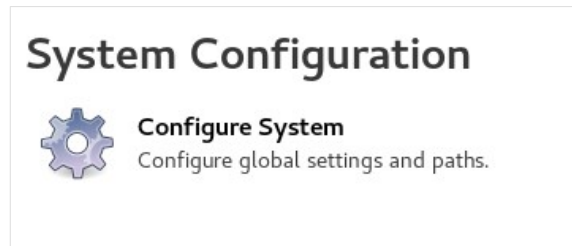
[don't save the credentials]

__10. On the left side of the page, click **Manage Jenkins**.

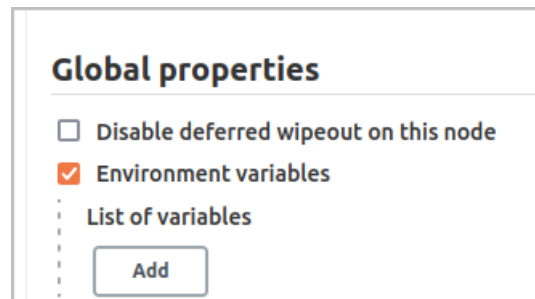


Note that Jenkins is slow so be patient and wait after you click for something to happen. You may see a bunch of warning, don't worry about them for now.

__11. Scroll down until you see **Configure System** then click on it.



__12. Scroll down until you see **Global properties**, then select the checkbox in front of **Environment variables**.



__13. Click **Add**

Name and Value text fields will appear.

__14. Enter the following into the text fields:

Name: KUBECONFIG
Value: /home/wasadmin/.kube/config

A screenshot of the 'Add' form for a new environment variable. It has two text input fields. The first field is labeled 'Name' and contains the text 'KUBECONFIG'. The second field is labeled 'Value' and contains the text '/home/wasadmin/.kube/config'.

Note: The config directory contains Kubernetes configuration. It's required so you can execute the "kubectl" commands from a Jenkins job.

__15. Click **Save** button.



Part 9 - Create a Jenkins Job

In this part, you will implement Continuous Integration by creating a Jenkins job. It will check out the Node.js code from the Git repository, build a new Docker image, delete the existing Kubernetes service and deployment configuration, and redeploy the app to Kubernetes.

__1. On the left side of the Jenkins home page, click **New Item**.

__2. Enter **CI-CD** as the job name.

__3. Select **Freestyle project** type.

A screenshot of the Jenkins 'Enter an item name' form. At the top, there's a header 'Enter an item name'. Below it is a text input field containing 'CI-CD'. Under the input field, there's a small text label '» Required field'. Below the input field, there's a section for selecting the project type. It features an icon of a box with a blue sphere on top, followed by the text 'Freestyle project'. Below this, there's a description: 'This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any b used for something other than software build.'

__4. Click **OK** button and wait for the next page to come up.

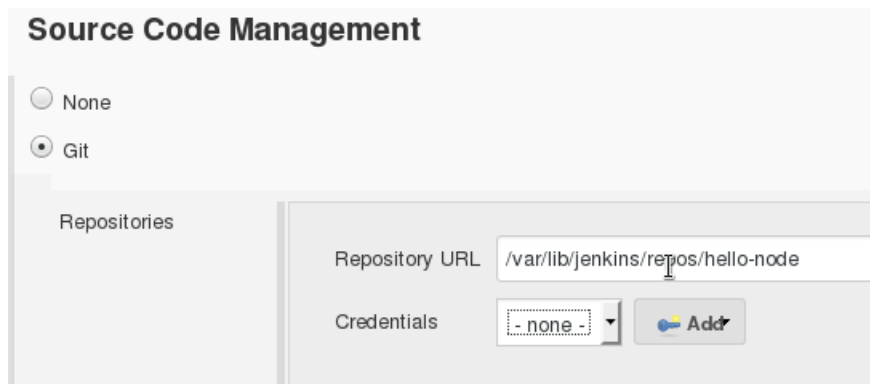
The page will have tabs. The first tab is General, the second is 'Source Code Management', etc.

__5. Click on the **Source Code Management** tab.

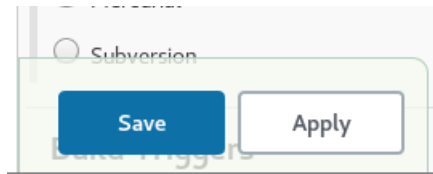
__6. Select **Git**.

__7. In Repository URL enter the following location and then hit Tab: (the message in red should go away after you hit tab)

`/var/lib/jenkins/repos/hello-node`



__8. Click the **Apply** button.



__9. Select the **Build Triggers** tab at the top of the page.

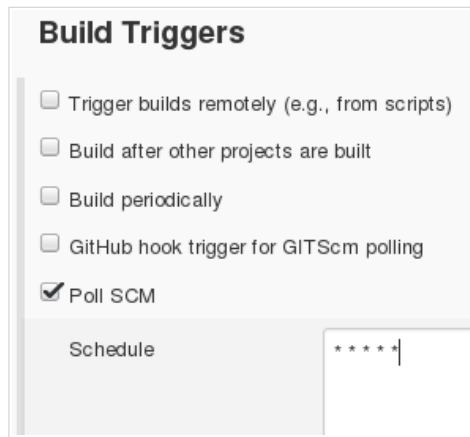
__10. Click on the checkbox next to **Poll SCM**.

A schedule text field will appear.

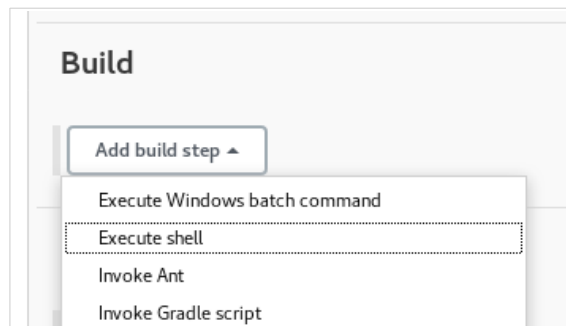
__11. Enter the following into the field (5 asterisks with a space in between each):

`* * * * *`

This tells Jenkins to check the SCM every 1 minute and triggers a build if changes are detected in the repository.



- __ 12. Click the **Apply** button.
- __ 13. Select the **Build** tab at the top of the screen.
- __ 14. Click **Add build step** and select **Execute shell**.



A text box labeled 'Command' will appear.