**BASIC OF CONTAINERS**

**What is a container?**

**Containers** provide an excellent way of packaging our application with all of its dependencies in a package and delivering it irrespective of the target environment. This makes the deployment faster and consistent among all the environments (development, testing, production etc.,) maintaining the essence of the functionalities at the same time.

**What are Linux containers?**

A **Linux container** is a single or a group of processes that are isolated from other processes running in the system. The Linux kernel performs this isolation using the kernel features such as namespaces, cgroups, etc.

**LXC (LinuX Containers)** is an operating system-level virtualization method for running multiple isolated Linux systems (containers) on a single control host. It offers a simple command line interface that provides satisfying user-experience while working with containers.

**Features of Linux containers**

Following are few irresistible features of Linux containers which are the major reasons why most of the organizations are moving towards containers now:

* It offers lightweight virtualization.
* A container can run one or more applications in it depending upon the requirement.
* Provisioning a container will not take more than few seconds/milliseconds.
* Near bare metal runtime performance.
* Faster deployment due to its consistency across various environments.
* Linux containers run on the operating system of the host machine , making it extremely lightweight and can be densely deployed.
* Provides agility.

**Working with Linux containers**

Linux containers are considered as groups of isolated processes running in a host machine. Linux containers contain application packaged with all its dependencies and allow us to work on it. These isolated processes that are contained in the Linux system using several features built into the Linux kernel such as:

* Chroots
* Namespaces and
* Cgroups

Using these features, we can build a container in the Linux system without the use of Docker.

To build a Linux container, we need an OS image(Container image). The container image is an executable package of all the files and dependencies required to run a container in the OS.

Before following the steps, make sure we are logged in as root user or have root user privileges.

*[root@docker ~]$ whoami*

*root*

Step 1: Making filesystem

Since a container image is a package of executable that contain all the files and dependencies required to run a container. For a CentOS image, we will now make a filesystem. Here, we have the dependencies archived in a tar file, centos-7-docker.tar.xz. (If this tar file is not present, copy the basic necessary directories like /bin, /lib, etc., from the current root system "/", and paste it in a new directory. )

*[root@docker ~]$ ls -l /var/tmp/Images/centos-7-docker.tar.xz*

*-rw-rw-rw-. 1 root root 42517348 Jun 16 09:09 /var/tmp/Images/centos-7-docker.tar.xz*

This should be extracted into a directory. Make a directory 'rootfs' under /home/root

*[root@docker ~]$ mkdir rootfs*

*[root@docker ~]$ pwd*

*/home/podxuser*

Now, extract the centos-7-docker.tar.xz into the newly created directory.

*[root@docker ~]$ tar xf /var/tmp/Images/centos-7-docker.tar.xz -C rootfs/*

After extraction , the rootfs directory will look like a regular Linux system which has a "bin" directory with executables, an "etc" with system configuration, a "lib" with shared libraries, and so on.

[root@docker ~]$ ls rootfs/

anaconda-post.log bin dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp usr var

**Difference between container and virtual machine**

**Virtualization**

Virtualization is the creation of a virtual -- rather than actual -- version of something, such as an operating system, a server, a storage device or network resources.

**Containerization**

Containerization comes as a lightweight alternative to virtualization. The application with all its dependencies and shared libraries are packed and made into containers. These containers are then deployed to anywhere it is needed irrespective of the environment of the target machine. This makes it possible to run multiple containers in a single OS. This further reduces the overhead cost cause by virtualization.

**Virtualization vs Containerization**

Choosing whether to virtualize or containerize our microservices is a challenging task. Because both the process backs up different requirements that arise during software development process. Containerization has following upsides over virtualization.

* Containerization reduces the overhead costs caused by virtualization, since only the OS of the physical server is virtualised.
* Microservices are better isolated in case of containers. Multiple microservices requiring different environments cannot be packed in a single VM due to overlapping of environments. But multiple containers can be deployed in a single OS.
* VMs take more space than containers. The same physical host can contain more containers than VMs.
* Creation process is swift in case of containers when compared with VMs.
* Versioning of applications is consistent in case of containers.
* Rollback is easier in containers than virtual machines. Because, deploying a container hardly takes a few seconds.

Containers offer all the above advantages and few more which can be explored once we get our hands on containers.

**What is Docker?**

The official definition of Docker goes like this:

"**Docker** is an open platform for developers and sysadmins to build, ship, and run distributed applications. Consisting of Docker Engine, a portable, lightweight runtime and packaging tool, and Docker Hub, a cloud service for sharing applications and automating workflows, Docker enables apps to be quickly assembled from components and eliminates the friction between development, QA, and production environments. As a result, IT can ship faster and run the same app, unchanged, on laptops, data center VMs, and any cloud”

**Docker** is a container manager that enables us to develop, deploy and run applications with containers. As mentioned earlier, containerization is not new. But deploying applications using containerization is.

Docker is a build system:

1. Images are built from sources.
2. Images are built using Docker file.

* It is a set of REST APIs:
  + Engine API (controls the docker engine).
  + Plugin API (extends the engine → network, storage, authorization).
  + Registry API (publish/download images).

**Features of Docker**

Docker is being increasingly adopted by many organizations because of the following features:

• It offers lightweight virtualization.

• It adopts its foundation from the LXC format especially, the namespaces and cgroups of Linux containers.

• Docker offers immutable architecture.

• Immutable images - instance is ephemeral, persistent data is stored outside the container, on the host or volume-containers.

• Instant deployment is a key feature of Docker.

• Docker is more suitable for micro-services (one process, one container) - is designed to support a single application.

**Clustering**

Along with all the essential features offered by Docker, one such feature that is not to be missed is "Clustering". Docker allows us to cluster containers across different physical servers in the network. This ensures effective management of containers by a manager node. Two broadly applicable options available for clustering are Docker swarm and Kubernetes. Combined with clustering, docker depicts the following functionalities:

* Provides maximum application density per machine (currently in the hundreds) than virtualization.
* Improved 'application namespace' isolation across your cluster.
* No longer need to ensure stacks / sub-stacks are rolled onto your cluster.
* Enables multiple versioning across a cluster easily.
* Docker combined with clustering can obviate traditional deployment problems.

**Why docker?**

Containerization using Docker deals with the problems faced with traditional application deployment as follows:

* Docker does not possess any dependency problem. If it builds in your system, it can build anywhere.
* Multiple environments for production, QA, testing etc., can be created in minutes.
* Implement reliable CI easily.
* Container images can be used as build artifacts.
* The container image is an executable which contains all the files and dependencies required to run the application. Hence, creating an environment can be as easy as running an executable file.
* Images are bigger, but they are broken down into layers.
* Changes are updated by just updating the layer which underwent change.
* It reduces the cost overhead by saving disk, network and memory usage.

A screenshot of a computer

Description automatically generated

**Installing Docker**

Docker can be installed in Linux systems with the use of the yum or an installation script. The demos here are done using a CentOS 7 distribution. These steps can be used for RHEL distributions as well. To install docker, the user must have root user privilege or it can be done with use of sudo command.

Note: Execute the following commands with sudo if you don't have root user privilege.

**Step 1:** Update yum

Update the yum repository by executing the following command.

*[root@docker ~]# yum -y update*

**Step 2:** Install docker

After updating the yum repository, execute the following command to install docker.

*[root@docker ~]# yum install -y docker*

**Step 3:** Registry configuration

Since we are working in a lab environment, it is not mandatory to secure our docker registry. This change is done in the docker configuration file of docker, /etc/sysconfig/docker. Add the following content inside /etc/sysconfig/docker.

*echo "INSECURE\_REGISTRY='--insecure-registry registry.example.com:5000'" >> /etc/sysconfig/docker*

*echo "ADD\_REGISTRY='--add-registry registry.example.com:5000'" >> /etc/sysconfig/docker*

*echo "BLOCK\_REGISTRY='--block-registry all'" >> /etc/sysconfig/docker*

After updating, the /etc/sysconfig/docker file will look like this:

*[root@docker ~]# cat /etc/sysconfig/docker*

*echo "INSECURE\_REGISTRY='--insecure-registry registry.example.com:5000'" >> /etc/sysconfig/docker*

*echo "ADD\_REGISTRY='--add-registry registry.example.com:5000'" >> /etc/sysconfig/docker*

*echo "BLOCK\_REGISTRY='--block-registry all'" >> /etc/sysconfig/docker*

**Step 4:** Enable and start docker

The daemon service for docker is 'docker' which should be enabled and started in order to start working with docker.

*[root@docker ~]# systemctl enable docker*

*[root@docker ~]# systemctl start docker*

**Step 5:** Check the status of docker

After enabling and starting the service, check its status to ensure that it is up and running.

*[root@docker ~]# systemctl status docker*

Step 6: Check version

Check the version of docker that has been installed using the following command.

*[root@docker ~]# docker version*

*Client:*

*Version: 1.12.6*

*API version: 1.24*

*Package version: docker-1.12.6-61.git85d7426.el7.centos.x86\_64*

*Go version: go1.8.3*

*Git commit: 85d7426/1.12.6*

*Built: Tue Oct 24 15:40:21 2017*

*OS/Arch: linux/amd64*

*Server:*

*Version: 1.12.6*

*API version: 1.24*

*Package version: docker-1.12.6-61.git85d7426.el7.centos.x86\_64*

*Go version: go1.8.3*

*Git commit: 85d7426/1.12.6*

*Built: Tue Oct 24 15:40:21 2017*

*OS/Arch: linux/amd64*

**Loading Docker images**

**Docker** launches a container by running an image. The **container** is nothing but a running instance of a container image. A **container image** is a packaged executable containing all the code, dependencies, shared libraries etc., required to run an application.

Apparently, to launch a container, we need the required container image. Since containers follow layering, the base layer of the container will be the container image.

Since we are working in a lab environment, we are going to preload all the Docker images needed for the lab by running a script, which will load the Docker images.

*[root@docker ~]# cd /root/docker-images*

*[root@docker|/root/docker-images]# sh -x import-for-docker-lab.sh*

This script will preload all the Docker images required for our current lab in Docker engine.(This step can be skipped if registry is used to get the Docker images.)

Executing the following command gives us the information about the docker images that has been imported:

*[root@docker ~]# docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*docker.io/centos latest 3fa822599e10 2 weeks ago 203.5 MB*

*docker.io/busybox latest 6ad733544a63 5 weeks ago 1.129 MB*

*docker.io/mhausenblas/simpleservice 0.5.0 601917f29430 7 months ago 682.6 MB*

*docker.io/jpetazzo/clock latest 12068b93616f 2 years ago 2.43 MB*

**Life cycle of a container**

A Docker container is creating by running a Docker image. The Docker image is pulled from the repository for this purpose. They can also be created on our own to satisfy our requirements. Once the container is created, it behaves as an isolated process in the OS running with its full environment. These containers go through a life cycle of changes when they are alive. The life cycle of a container defines the the state the container is currently in.

There are different states to a container as listed below:

* Created: The state of the container when it gets created. In this state, the container has not yet started.
* Running: The container is currently running in this state, which means it has started.
* Paused: The container's processes have been paused. It is like temporarily stopping a container.
* Restarting: The container is in the process of getting restarted.
* Exited: The container is stopped either by the user or automatically due to a problem.
* Dead: The container that has been removed. It no longer lives in the Docker host.

**Container commands**

The following is a list of commands that can be used to work with the containers throughout their life cycle.

1. To create a container:

*$ docker run <image>*

2. To run a container in the interactive mode:

*$ docker run -it <image>*

3. To run a container in the background:

*$ docker run <image> -d*

4. To attach to a container running in the background:

*$ docker attach <containerID>*

5. To get the statistics of a running container:

*$ docker stat <containerID>*

6. To see the processes withing a container:

*$ docker top <containerID>*

7. To stop a running container:

*$ docker stop <containerID>*

8. To kill the processes inside a container:

*$ docker kill <containerID>*

9. To remove a container:

*$ docker rm <containerID>*

10. To list all the containers in the Docker host:

*$ docker ps -a*

**Creating the first container**

A container is a running instance of a docker image. Any number of containers can be created inside a Docker host. Any container can be created on any host. It has to be noted that anything that happens in the host machine or the container is exclusive to only themselves.

Docker engine launches a container by running a docker image.

Let us try to run our very first container. In this demo, we will run a simple "busybox" container in the interactive mode. The busybox image will be pulled from the repository.

**Step 1:** Execute the <docker run> command to create the container.

*$ docker run -it busybox*

*Unable to find image 'busybox:latest' locally*

*latest: Pulling from library/busybox*

*91f30d776fb2: Pull complete*

*Digest: sha256:9ddee63a712cea977267342e8750ecbc60d3aab25f04ceacfa795e6fce341793*

*Status: Downloaded newer image for busybox:latest*

*c4ef2e319cdf / #*

The busybox image has been pulled from the repository."c4ef2e319cdf" is the ID of the container. The container ID is an alphanumeric string which is used to identify the container throughout its lifecycle.

**Step 2:** We have launched the container in interactive mode. Exit from the container and check the list of containers using following command.

*c4ef2e319cdf / # exit*

*$ docker ps -a*

*CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES*

*c4ef2e319cdf busybox "sh" About a minute ago Exited (0) 10 seconds ago upbeat\_yonath*

The status of the container is exited which means the container was stopped.(We exited the container)

**Step 3:** Start the container using <docker start> command and execute <docker ps>.

*$ docker start c4ef*

*c4ef*

*$ docker ps -a*

*CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES*

*c4ef2e319cdf busybox "sh" 3 minutes ago Up 2 seconds upbeat\_yonath*

The status of the container is now "Up" which means the container is running. But we have not connected to the terminal of the container by starting it. Execute the next step to do so.

**Step 4:** Attach to the terminal of the container using <docker attach> command.

$ docker attach c4ef

c4ef2e319cdf / #

Step 5: Exit the container. Now, the container is not running. We can now remove the container by executing <docker rm> command.

*$ docker rm c4ef*

*c4ef*

*$ docker ps -a*

*CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES*

No container is running in the Docker host at the moment.

**Docker images**

**Docker images** are building blocks of containers. They contain the set of instructions necessary for creating the containers. Containers are born when the docker images are run. Essentially, a container is a running instance of a docker image.



Imagine, we want to create an Nginx container. To create a container, we need the necessary Docker image. This image can be accessed from a Docker registry (either Docker Hub or a self-hosted registry). For this purpose, we create a running instance of the Docker image by executing the following command.

*# docker run nginx*

In the above command, nginx is the name of the image used to create the Nginx container. So it is rightly said that docker images are building blocks of docker containers.

In essence, a Docker image is a bundle of files and metadata required to create a fully functional container environment which might include the filesystem, installation code, applications and dependencies, etc.,

**What are Docker images made up of?**

Docker images are made up of layers, one stacked on the other conceptually. Each layer refers to certain instructions that was given while creating the Docker image (through a Docker file).

In fact, when we built our first container, we noticed the images being pulled from the registry in the form of layers.

Docker images are mostly built from a parent image (maybe a Debian, CentOS, Ubuntu etc.,) of our requirement. Over the parent image, the changes are made in terms of layers. The instructions mentioned in the Docker file are executed one by one on the parent image in the form of layers in order to form the final Docker image.

Each newly stacked layer references the layer beneath it.

A screenshot of a container

Description automatically generated

Docker images can also be built using a base image, meaning you build your image from the scratch. This way, you can completely control the content of your image. But, building a Docker image from scratch is considered to be more time-consuming and complex for beginners.

Note: However, the terms 'parent image' and 'base image' are being used interchangeably.

**Immutability of Docker images**

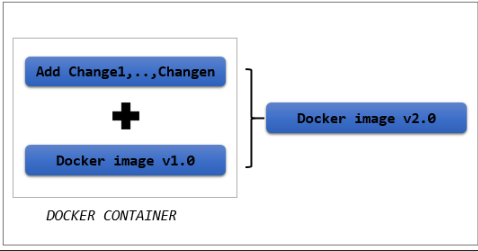
A Docker image once built, it immutable because it is only a read-only filesystem. So, how can we make changes to the existing image?

When a Docker container is instantiated by running a Docker image, Docker adds a thin writable layer which stores all the changes made to the container during the runtime. This is called as the container layer. This container layer is difference between the original Docker image used to build the container and the live container.

Since Docker images are immutable, any number of containers can be built from the same image while those containers can maintain their individual states without any change.

In essence, we don’t make changes to the existing image. Instead,

* We build a container out of the image.
* Make the necessary changes.
* Once the changes are complete, we convert it into a new layer.
* A new image is now created by adding this new layer to the existing image.



**Working with Docker images**

Docker client deals with the Docker images. Docker client pulls an image from the registry, pushes an image to the registry, tags an image, delete/remove, etc., A docker image when created will be given an ID, which can be used to refer to the image in future.

The docker command is used with the following . Using it with appropriate arguments and command line options lets us work with docker images seamlessly.

1. To list all the images present in the Docker engine:

*$ docker images*

2. To search for a particular image from the Docker registry (By default, DockerHub is searched):

*$ docker search nginx*

3. To download an image from the Docker registry:

*$ docker pull nginx*

4. To push a Docker image into the Docker registry:

*$ docker push newubuntu*

**Working with Docker images**

As discussed, it is very clear that a container is the running instance of a docker image. Let us assume that we want to run a CentOS container. There are two cases:

i) The image required is present: In this case, the container can be simply run using the "docker run" command.

ii) The image required is not present: In this case, the container can be run,

* By either pulling the image first and then running the container using "docker run" command or
* By executing the "docker run" command directly. The docker run command will search for the image locally. If not found, it automatically pulls the image from the registry and then runs the container

In this demo, let us assume that the image required is not present.

**Step 1:** List all the images available in the Docker host to check whether CentOS is present.

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*ubuntu latest 16508e5c265d 22 months ago 84.1MB*

*redis latest 4e8db158f18d 23 months ago 83.4MB*

*weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB*

*alpine latest 11cd0b38bc3c 24 months ago 4.41MB*

**Step 2:** Search for all the images in the Docker registry with the keyword "centos" in their names. Docker Hub also provides the description of each image present.

*$ docker search centos*

*NAME DESCRIPTION STARS OFFICIAL AUTOMATED*

*centos The official build of CentOS. 6064 [OK]*

*ansible/centos7-ansible Ansible on Centos7 130 [OK]*

*consol/centos-xfce-vnc Centos container with "headless" VNC session… 116 [OK]*

*.*

*.*

*pivotaldata/centos7-dev CentosOS 7 image for GPDB development 0*

*pivotaldata/centos6.8-dev CentosOS 6.8 image for GPDB development 0*

*smartentry/centos centos with smartentry 0 [OK]*

**Step 3:** Choose the appropriate image and pull it from the registry.

If no tags are mentioned, by default, the image with "latest" tag gets pulled. More on images and tags, later.

*$ docker pull centos [OK]*

*Using default tag: latest [OK]*

*latest: Pulling from library/centos [OK]*

*6910e5a164f7: Pull complete [OK]*

*Digest: sha256:4062bbdd1bb0801b0aa38e0f83dece70fb7a5e9bce223423a68de2d8b784b43b*

*Status: Downloaded newer image for centos:latest*

**Step 4:** List the images to check whether "centos" image is now present in the Docker host.

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*centos latest 831691599b88 12 days ago 215MB [OK]*

*ubuntu latest 16508e5c265d 22 months ago 84.1MB*

*redis latest 4e8db158f18d 23 months ago 83.4MB [OK]*

*weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB [OK]*

*alpine latest 11cd0b38bc3c 24 months ago 4.41MB*

**Step 5:** Now that we have the image in our Docker host, we can run the "centos" container.

*$ docker run -it centos*

*[root@a9e73acbecd8 /]#*

The "centos" container is successfully created.

Now, Let us create a "debian" container. From the list of images displayed in "Step 4", it is clear that, there is no "debian" image present in the Docker host. Instead of pulling the image first, let us run the container directly.

*$ docker run -it debian*

*Unable to find image 'debian:latest' locally*

*latest: Pulling from library/debiane9afc4f90ab0: Pull complete*

*Digest: sha256:46d659005ca1151087efa997f1039ae45a7bf7a2cbbe2d17d3dcbda632a3ee9a*

*Status: Downloaded newer image for debian:latest*

*root@ba308272c106:/#*

You can notice the image being pulled layer by layer. Once the pull was complete, a new container was started. List the containers to see a "debian" container present.

*$ docker ps -a*

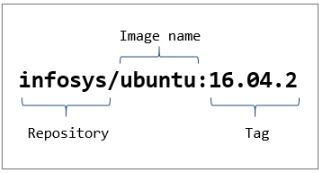
*CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES*

*ba308272c106 debian "bash" 2 minutes ago Exited (0) About a minute ago awesome\_kowalevski*

*a9e73acbecd8 centos "/bin/bash" 12 minutes ago Exited (0) 12 minutes ago priceless\_fermat*

**Images and tags**

In the last demo, when we pulled a "centos" image from the registry, it automatically pulls the image with tag "latest". "Latest" is the default tag assumed by Docker engine when no tag is mentioned while pulling an image. But the "latest" tag doesn't really reflect the same meaning as it looks like. More on the "latest" tag later.



Docker images are referred using their unique image IDs or names. A sample image ID might look like "16508e5c265d". But if you want to access an image with a more relevant name rather than either a broad name like "ubuntu" or a complex image ID like, "16508e5c265d", Docker tags will help you do it.

Using the docker tag command gives us the following scope:

* We are able to create an alias name for an image.
* We are able to perform version/variant controlling over docker images.

For example, Look at the list of images present in the Docker host currently.

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*ubuntu latest 16508e5c265d 22 months ago 84.1MB*

*redis latest 4e8db158f18d 23 months ago 83.4MB*

*weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB*

*alpine latest 11cd0b38bc3c 24 months ago 4.41MB*

Suppose I am using the weaveworks/scope image for my "Application1" , I might want to name it something like "imageforapp1". This is it is more relevant and also easier to remember.

Once the tagging is done, we will see that an alias name was given to the image weaveworks/scope.

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

ubuntu latest 16508e5c265d 22 months ago 84.1MB

redis latest 4e8db158f18d 23 months ago 83.4MB

imageforapp1 latest 4b07159e407b 23 months ago 68MB

weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB

alpine latest 11cd0b38bc3c 24 months ago 4.41MB

As we can see, "imageforapp1" alias was created and notice the "latest" tag given to it. If we do not mention a tag explicitly for an image, "latest" is the default tag assigned to it by the Docker engine.

Note that the "latest" tag does not always refer to the "latest" version of the particular image. It is the image that was named without any tag.

However tagging an image makes more sense, when we have a newer version of the image with some updates and we want the users to be aware of what they are getting into before downloading it.

Note: Notice that the image ID for weaveworks/scope and imageforapp1 have the same image ID, implying that both are same images with different names.

**Building a docker image**

Till the last demo, we were using the readily available docker images present in the Docker Hub. Some of those images are official images. Others are images created by Docker users for specific purposes. Whenever we are posed with requirements that are new, we cater to it by building a new solution. Same way, when we want a Docker image which can readily be used for a new purpose, we can easily build it.

Docker images can be built in two ways:

i) Interactive way( Create an image from existing container):

In this case, we start a container with an existing image. Doing this will create a thin writable layer over the container. Now, the changes can be customized inside the container, which will be written in the writable layer. From this, we can build a new Docker image.

ii) Using a Docker file:

A Docker file contains the instructions required to build the Docker image. There are two ways in which you can create Docker image using Docker file.

* We can specify a parent image and specify the changes we want to make to it. These changes will be made in the form of layers. Finally, we can build our own Docker image.
* We can create a fresh Docker image from scratch using the Docker file.

**Docker file**

A Dockerfile is a simple configuration file following a specific syntax. The Dockerfile contains the step by step instructions that are necessary to build a docker image. These instructions are available as commands that need to be run to assemble the final Docker image. Each of the instructions are added in the form of layers and each layer is the delta of the previous layer.

We can create a Dockerfile by easily creating a text file and editing it using a text editor of our choice. "Dockerfile" is the default name for the file used widely. But it can be named anything of our choice. A sample Docker file where a docker image is built from a parent image is shown below:

*FROM ubuntu:18.04*

*COPY . /app*

*RUN make /app*

*CMD python /app/app.py*

Each of the above instructions create a new layer. Let us break down the Dockerfile line by line.

*FROM ubuntu:18.04*

FROM is a keyword used to specify the parent image on which we are going to make changes to build our new Docker image. Here, an Ubuntu image of version 18.04 is used as the parent image.

*COPY . /app*

**COPY** is used to copy files/ directories from a source to target. This is used to put the required files in the Docker image from out host.

*RUN make /app*

**RUN** keyword lets us run different commands inside our Docker image. Multiple commands can be broken down into different RUN statements in order to improve readability of the Docker file.

*CMD python /app/app.py*

**CMD** keywork allows us to specify a default command to run once the container is run using this image. Multiple CMD statements can be specified but only the last one will take effect.

The "docker build" command is used to build an image from the Dockerfile.

**Building your own Docker image**

Imagine we are setting up a lab environment for a set of trainees to practice on how to configure an Apache webserver. There is no proper details about the Linux distribution the trainees have in their laptops. So we have decided to share a docker image for this purpose, which when run automatically spins up a CentOS container, appropriate for practicing web server configuration.

In such scenario, choosing to build an image from scratch is going to be unnecessarily additional work since we have numerous CentOS images available in the registry. So, ruling out this choice, we have two more choices left in hands:

1. Launch a centos container and make the necessary changes to it, which will be written in the writable layer of the container and build a new docker image out of it.
2. Use a Dockerfile by specifying a parent image, add the instructions and then build the image.

Before diving into writing our own Dockerfile, we will try to build a Docker image interactively.

**Build an image interactively**

Let us build our Docker image that has to be shared with the trainees, in order for them to be able to practice webserver configuration with ease. Follow the below steps to do the same.

**Step 1:** Make sure you have the parent image in the Docker host. In this case, it is CentOS 8. If not, pull one from the registry.

*$ docker pull centos:8*

*8: Pulling from library/centos*

*6910e5a164f7: Pull complete*

*Digest: sha256:4062bbdd1bb0801b0aa38e0f83dece70fb7a5e9bce223423a68de2d8b784b43b*

*Status: Downloaded newer image for centos:8*

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*centos 8 831691599b88 13 days ago 215MB*

*redis latest 4760dc956b2d 2 years ago 107MB*

*ubuntu latest f975c5035748 2 years ago 112MB*

*alpine latest 3fd9065eaf02 2 years ago 4.14MB*

**Step 2:** Launch a CentOS:8 container.

*$ docker run -it centos:8*

*[root@4914a0cea58c /]#*

**Step 3:** As matter of fact, we want the Apache web server package to be readily installed for the trainees to practice. Check whether "httpd" package is installed.

*[root@4914a0cea58c /]# rpm -q httpd*

*package httpd is not installed*

**Step 4:** Install the "httpd" package in the container using the yum repository. (Make sure the yum repository is configured.)

*[root@4914a0cea58c /]# yum install -y httpd*

*.*

*<snip>*

*.*

*Installed:*

*apr-1.6.3-9.el8.x86\_64 apr-util-1.6.1-6.el8.x86\_64*

*apr-util-bdb-1.6.1-6.el8.x86\_64 apr-util-openssl-1.6.1-6.el8.x86\_64*

*brotli-1.0.6-1.el8.x86\_64 centos-logos-httpd-80.5-2.el8.noarch*

*httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0afa8.noarch*

*httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 mailcap-2.1.48-3.el8.noarch*

*mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86\_64*

*Complete!*

**Step 5:** Check for the installed package.

*[root@4914a0cea58c /]# rpm -q httpd 10/11*

*httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64*

*The httpd package is successfully installed.*

This change done inside the container is written on the writable layer of the container.

**Step 6:** To inspect the changes made to the filesystem of the container, execute the "docker diff <container\_id>". This gives the difference between the container's current filesystem and the original image. Exit from the container before executing the command.

*[root@4914a0cea58c /]# exit*

*exit*

*$ docker diff 4914a0cea58c 2/11*

*C /etc 3/11*

*C /etc/gshadow 4/11*

*.*

*.*

*A /var/log/dnf.librepo.log*

*A /var/log/dnf.log*

*A /var/log/httpd*

We can see a list of files/directories prefixed with either C or A.

C - indicates files/directories that have been changed.

A - indicates files/directories that have been added.

**Step 7:** To add our changes as a new layer and to create a new image with this added layer, execute "docker commit" command. The output of this command is the image ID of our new Docker image.

*$ docker commit 4914a0cea58c*

*sha256:9ac3092f21147e8ff78ba2dfa38ccdd0b179ba96bdacf0f718510177be2ed2bb*

We have successfully created our own first docker image. The big alphanumeric output is the ID of the new image.

**Step 8:** List out the available image to see whether the new image has been added to the lot.

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*<none> <none> 9ac3092f2114 2 minutes ago 254MB*

*centos 8 831691599b88 13 days ago 215MB*

*redis latest 4760dc956b2d 2 years ago 107MB*

*ubuntu latest f975c5035748 2 years ago 112MB*

*alpine latest 3fd9065eaf02 2 years ago 4.14MB*

**Step 9:** Name the newly created docker image with an appropriate tag.

*$ docker tag 9ac3092f2114 centoswithhttpd:1*

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*centoswithhttpd 1 9ac3092f2114 5 minutes ago 254MB*

*centos 8 831691599b88 13 days ago 215MB*

*redis latest 4760dc956b2d 2 years ago 107MB*

*ubuntu latest f975c5035748 2 years ago 112MB*

*alpine latest 3fd9065eaf02 2 years ago 4.14MB*

**Step 10:** Launch a container with the new image to see the changes.

*$ docker run -it centoswithhttpd:1*

*[root@cae890485c74 /]# rpm -q httpd*

*httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64*

As we can see, the new centos image has "httpd" pre-installed in it unlike the base image.

The image is now ready to be shared with trainees for practice. Instead of sharing it individually, this image can be uploaded to a private registry where the users with access, can use this image. More on this later.

**Build an image using Dockerfile**

The docker image created in the previous demo involved human interaction with the container. In case there are loads of complex changes to be incorporated into the container, it can get really hard to do so without any errors. In order to automate the process of creating a docker image, Dockerfile can be used. The FROM keyword in the Dockerfile decides whether we are creating a new image based on a parent image or we are creating a brand new docker image from scratch.

**FROM centos:8** => indicates that we are going to build the new docker image with the base image as "centos:8".

**FROM scratch** => indicates that we are going to build an image from scratch.

Our Dockerfile is going to contain a series of instructions that will build our final docker image.

**Step 1:** The Dockerfile must be created in a new empty directory. Create a directory "centoswithhttpd". Under it, create a text file with the name, "Dockerfile".

*$ mkdir centoswithhttpd*

*$ cd centoswithhttpd*

*$ touch Dockerfile*

*$ ls*

*Dockerfile*

**Step 2:** Add the content to the Dockerfile using the VI editor(or any editor of your choice) as shown below.

*$ vi Dockerfile*

*$ cat Dockerfile*

*FROM centos:8*

*RUN yum install -y httpd*

**Step 3:** Now that we have instructed the steps to create the image, let us go ahead and build the image.

*$ docker build -t centoswithhttpd:1 .*

*Sending build context to Docker daemon 2.048kB*

*Step 1/2 : FROM centos:8*

*---> 831691599b88*

*Step 2/2 : RUN yum install -y httpd*

*---> Running in 62163df9c424*

*CentOS-8 - AppStream 9.8 MB/s | 5.8 MB 00:00*

*CentOS-8 - Base 5.5 MB/s | 2.2 MB 00:00*

*CentOS-8 - Extras 22 kB/s | 6.7 kB 00:00*

*Last metadata expiration check: 0:00:01 ago on Tue Jun 30 11:30:21 2020.*

*Dependencies resolved.*

*================================================================================*

*Package Arch Version Repo Size*

*================================================================================*

*Installing:*

*httpd x86\_64 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 1.7 M*

*Installing dependencies:*

*apr x86\_64 1.6.3-9.el8 AppStream 125 k*

*apr-util x86\_64 1.6.1-6.el8 AppStream 105 k*

*brotli x86\_64 1.0.6-1.el8 BaseOS 323 k*

*centos-logos-httpd*

*noarch 80.5-2.el8 BaseOS 24 k*

*httpd-filesystem noarch 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 36 k*

*httpd-tools x86\_64 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 103 k*

*mailcap noarch 2.1.48-3.el8 BaseOS 39 k*

*mod\_http2 x86\_64 1.11.3-3.module\_el8.2.0+307+4d18d695 AppStream 157 k*

*Installing weak dependencies:*

*apr-util-bdb x86\_64 1.6.1-6.el8 AppStream 25 k*

*apr-util-openssl x86\_64 1.6.1-6.el8 AppStream 27 k*

*Enabling module streams:*

*httpd 2.4*

*Transaction Summary*

*================================================================================*

*Install 11 Packages*

*Total download size: 2.6 M*

*Installed size: 7.5 M*

*Downloading Packages:*

*(1/11): apr-util-bdb-1.6.1-6.el8.x86\_64.rpm 456 kB/s | 25 kB 00:00*

*(2/11): apr-1.6.3-9.el8.x86\_64.rpm 1.5 MB/s | 125 kB 00:00*

*(3/11): apr-util-1.6.1-6.el8.x86\_64.rpm 975 kB/s | 105 kB 00:00*

*(4/11): apr-util-openssl-1.6.1-6.el8.x86\_64.rpm 424 kB/s | 27 kB 00:00*

*(5/11): httpd-filesystem-2.4.37-21.module\_el8.2 528 kB/s | 36 kB 00:00*

*(6/11): httpd-tools-2.4.37-21.module\_el8.2.0+38 1.0 MB/s | 103 kB 00:00*

*(7/11): mod\_http2-1.11.3-3.module\_el8.2.0+307+4 1.9 MB/s | 157 kB 00:00*

*(8/11): centos-logos-httpd-80.5-2.el8.noarch.rp 825 kB/s | 24 kB 00:00*

*(9/11): mailcap-2.1.48-3.el8.noarch.rpm 637 kB/s | 39 kB 00:00*

*(10/11): httpd-2.4.37-21.module\_el8.2.0+382+15b 5.1 MB/s | 1.7 MB 00:00*

*(11/11): brotli-1.0.6-1.el8.x86\_64.rpm 1.7 MB/s | 323 kB 00:00*

*--------------------------------------------------------------------------------*

*Total 2.6 MB/s | 2.6 MB 00:01*

*warning: /var/cache/dnf/AppStream-02e86d1c976ab532/packages/apr-1.6.3-9.el8.x86\_64.rpm: Header V3 RSA/SHA256 Signature, key ID 8483c65d: NOKEY*

*CentOS-8 - AppStream 1.6 MB/s | 1.6 kB 00:00*

*Importing GPG key 0x8483C65D:*

*Userid : "CentOS (CentOS Official Signing Key) <security@centos.org>"*

*Fingerprint: 99DB 70FA E1D7 CE22 7FB6 4882 05B5 55B3 8483 C65D*

*From : /etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial*

*Key imported successfully*

*Running transaction check*

*Transaction check succeeded.*

*Running transaction test*

*Transaction test succeeded.*

*Running transaction*

*Preparing : 1/1*

*Installing : apr-1.6.3-9.el8.x86\_64 1/11*

*Running scriptlet: apr-1.6.3-9.el8.x86\_64 1/11*

*Installing : apr-util-bdb-1.6.1-6.el8.x86\_64 2/11*

*Installing : apr-util-openssl-1.6.1-6.el8.x86\_64 3/11*

*Installing : apr-util-1.6.1-6.el8.x86\_64 4/11*

*Running scriptlet: apr-util-1.6.1-6.el8.x86\_64 4/11*

*Installing : httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8. 5/11*

*Installing : mailcap-2.1.48-3.el8.noarch 6/11*

*Installing : centos-logos-httpd-80.5-2.el8.noarch 7/11*

*Installing : brotli-1.0.6-1.el8.x86\_64 8/11*

*Running scriptlet: httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 9/11*

*Installing : httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 9/11*

*Installing : mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86 10/11*

*Installing : httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 11/11*

*Running scriptlet: httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 11/11*

*Verifying : apr-1.6.3-9.el8.x86\_64 1/11*

*Verifying : apr-util-1.6.1-6.el8.x86\_64 2/11*

*Verifying : apr-util-bdb-1.6.1-6.el8.x86\_64 3/11*

*Verifying : apr-util-openssl-1.6.1-6.el8.x86\_64 4/11*

*Verifying : httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 5/11*

*Verifying : httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 6/11*

*Verifying : httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8. 7/11*

*Verifying : mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86 8/11*

*Verifying : brotli-1.0.6-1.el8.x86\_64 9/11*

*Verifying : centos-logos-httpd-80.5-2.el8.noarch 10/11*

*Verifying : mailcap-2.1.48-3.el8.noarch 11/11*

*Installed:*

*apr-1.6.3-9.el8.x86\_64*

*apr-util-1.6.1-6.el8.x86\_64*

*apr-util-bdb-1.6.1-6.el8.x86\_64*

*apr-util-openssl-1.6.1-6.el8.x86\_64*

*brotli-1.0.6-1.el8.x86\_64*

*centos-logos-httpd-80.5-2.el8.noarch*

*httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64*

*httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0afa8.noarch*

*httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64*

*mailcap-2.1.48-3.el8.noarch*

*mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86\_64*

*Complete!*

*Removing intermediate container 62163df9c424*

*---> 2477edfffca1*

*Successfully built 2477edfffca1*

*Successfully tagged centoswithhttpd:1*

The -t stands for tagging the newly created image with an appropriate name.

**Step 4:** Check the image list to see our newly created image.

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*centoswithhttpd 1 2477edfffca1 4 minutes ago 254MB*

*centos 8 831691599b88 13 days ago 215MB*

*ubuntu latest 16508e5c265d 22 months ago 84.1MB*

*redis latest 4e8db158f18d 23 months ago 83.4MB*

*weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB*

*alpine latest 11cd0b38bc3c 24 months ago 4.41MB*

**Step 5:** Spin a new container with the new docker image.

*$ docker run -it centoswithhttpd:1*

*[root@a69aec27a7b3 /]#*

*[root@a69aec27a7b3 /]# rpm -q httpd*

*httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64*

**Pushing Docker image to Docker hub**

Let us push the "centoswithhttpd:1" image, we created in the previous demo to our repository. Follow the below steps to push any number of local Docker image to your DockerHub repository.

Note: To push a local image to DockerHub repository, your Docker machine needs to connected to the internet.

**Step 1:** Before pushing a Docker image to a Dockerhub repository, the image must be named appropriately. The following convention must be followed while naming the image.

*<dockerid>/<repo\_name>:<tag>*

Using this convention, multiple images can be pushed inside a repository by naming the images with different tags. If the tag is not mentioned, the default tag "latest" is assumed. Images can be named as follows:

i) Naming the image while building the image:

*$ docker build -t <dockerid>/<repo\_name>:<tag> Dockerfile*

ii) Naming an existing image:

*$ docker tag <existing\_image> <dockerid>/<repo\_name>:<tag>*

iii) Naming the image while committing changes:

*$ docker commit <existing\_container> <dockerid>/<repo\_name>:<tag>*

In this demo, we are renaming an existing image, centoswithhttps:1.

*$ docker tag centoswithhttpd:1 basila123/mynewrepo:centoswithhttpd*

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*basila123/mynewrepo centoswithhttpd c3c247ab0923 52 seconds ago 254MB*

*centoswithhttpd 1 c3c247ab0923 52 seconds ago 254MB*

**Step 2:** Now the renamed image can be pushed to the repository. To do this, we have to login to the DockerHub account from the command line using "docker login" command.

*$ docker login*

*Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.*

*Username: basila123*

*Password:*

*Login Succeeded*

**Step 3:** Push the docker image to the repository now using "docker push" command.

*$ docker push basila123/mynewrepo:centoswithhttpd*

*The push refers to repository [docker.io/basila123/mynewrepo]*

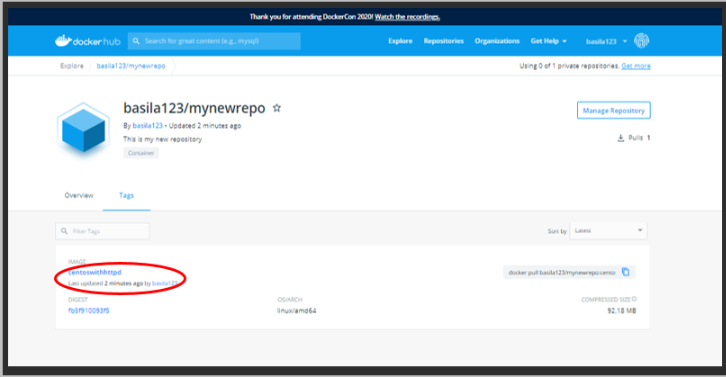
*80b77f3d336f: Pushed*

*eb29745b8228: Mounted from library/centos*

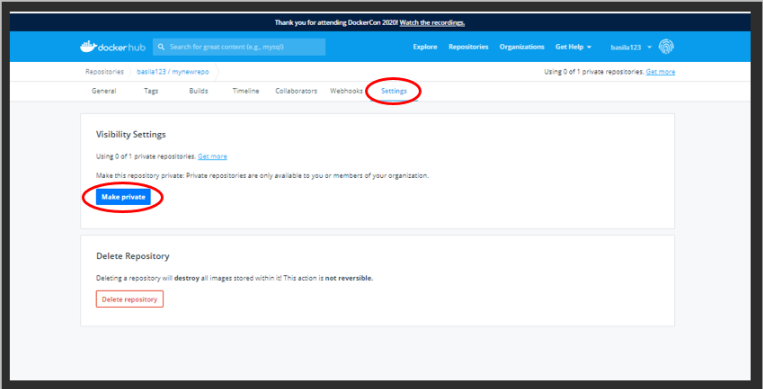
*centoswithhttpd: digest: sha256:fb5f910093f5dcfe529667e6d9551a7bc5d459184f92fa11bb98006fc17221fc*

*size: 741*

Now, this image is available to the Docker community for use. Click on the "tags" column of you repository to view the image being present in the DockerHub.



SNote: To create a Private DockerHub repository, select "Private" as the visibility. The visibility of existing repositories can be changed in the Settings tab of the repository.

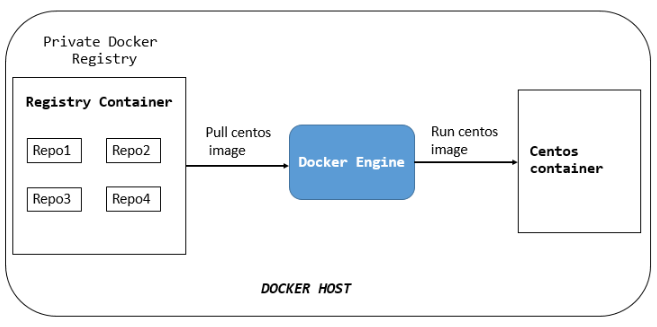


**Private Docker registry**

Hosting your Docker images on Docker Hub, as simple as it sounds, is not always chosen by organisations dealing with critical information. The data dealt within the organisation should be protected off breach at all times. That being said, hosting a Docker image created for an organisation level purpose on Docker Hub sounds ridiculous. Here is where Private registries come into picture.

Docker allows us to have a Private Docker registry of our own hosted inside our organisation. This Private Docker registry can then be used to store and manage images that are ought to be kept within the walls of organisation.

Docker provides us a "registry" container which can be used to host a private docker registry. This registry will remain as the source of our Docker images within the organisation.



**Create a Private Docker registry**

Follow the below steps to create local private docker registry.

**Step 1:** Run a registry container in your Docker host. We are going to expose a port of the container to the port of our host machine.

*$ docker run -d -p 5000:5000 --name registry registry:2*

*Unable to find image 'registry:2' locally*

*2: Pulling from library/registry*

*cbdbe7a5bc2a: Pull complete*

*47112e65547d: Pull complete*

*46bcb632e506: Pull complete*

*c1cc712bcecd: Pull complete*

*3db6272dcbfa: Pull complete*

*Digest: sha256:8be26f81ffea54106bae012c6f349df70f4d5e7e2ec01b143c46e2c03b9e551d*

*Status: Downloaded newer image for registry:2*

*23b142cfd4daedd01ad51e01227e4fcce157a460259a5b9e82e1b417dcb6c4ed*

*-p - to specify the ports.*

*registry - name of the container.*

*registry:2 - tagging the image with this name.*

**Step 2:** Check the list of docker images.

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*registry 2 2d4f4b5309b1 12 days ago 26.2MB*

*ubuntu latest 16508e5c265d 22 months ago 84.1MB*

*redis latest 4e8db158f18d 23 months ago 83.4MB*

*weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB*

*alpine latest 11cd0b38bc3c 24 months ago 4.41MB*

We have ubuntu:latest image in our Docker host. We will try to upload this to our private docker registry. Before that, the image has to be tagged appropriately.

Step 3: Tag the ubuntu image as follows.

*$ docker tag ubuntu localhost:5000/ubuntu*

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*registry 2 2d4f4b5309b1 12 days ago 26.2MB*

*ubuntu latest 16508e5c265d 22 months ago 84.1MB*

*localhost:5000/ubuntu latest 16508e5c265d 22 months ago 84.1MB*

*redis latest 4e8db158f18d 23 months ago 83.4MB*

*weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB*

*alpine latest 11cd0b38bc3c 24 months ago 4.41MB*

*localhost:5000 - name of the private docker registry*

ubuntu - name of the repository where the image is going to be pushed.

Specify a tag if needed.

**Step 4:** Push the newly tagged image to our private docker registry.

*$ docker push localhost:5000/ubuntu*

*The push refers to repository [localhost:5000/ubuntu]*

*ec8257ff6a7a: Pushed*

*7422efa72a14: Pushed*

*b6a02001ba33: Pushed*

*a26724645421: Pushed*

*a30b835850bf: Pushed*

*latest: digest: sha256:ac533e4ead4110211a4d67cbf44ed8b7d1aca2b8e6f15d1e8768eadaf433dd31 size: 1357*

**Step 5:** Delete both the ubuntu images present in our Docker host now.

*$ docker rmi ubuntu:latest*

*Untagged: ubuntu:latest*

*Untagged: ubuntu@sha256:72f832c6184b55569be1cd9043e4a80055d55873417ea792d989441f207dd2c7*

*$ docker rmi localhost:5000/ubuntu*

*Untagged: localhost:5000/ubuntu:latest*

*Untagged: localhost:5000/ubuntu@sha256:ac533e4ead4110211a4d67cbf44ed8b7d1aca2b8e6f15d1e8768eadaf433dd31*

*Deleted: sha256:16508e5c265dcb5c05017a2a8a8228ae12b7b56b2cda0197ed5411bda200a961*

*Deleted: sha256:9c86133a1c6d4396a658294f4327a5cf7bcd7572c272bf5f75200f29326e8b6c*

*Deleted: sha256:33eb1e02dcc9a707a475bf5aa739bd9f6d607b355beae5544b6576d1534fe949*

*Deleted: sha256:e54feb0ab7d782d04296e5d56bbf7efc843233497f0a16aab21bf8995f3385a7*

*Deleted: sha256:c2f76372c07b398c959f25ac97af0c4080e323d082e809de4316c1c728d5c096*

*Deleted: sha256:a30b835850bfd4c7e9495edf7085cedfad918219227c7157ff71e8afe2661f63*

List the docker images to make sure there are no ubuntu images currently.

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*registry 2 2d4f4b5309b1 12 days ago 26.2MB*

*redis latest 4e8db158f18d 23 months ago 83.4MB*

*weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB*

*alpine latest 11cd0b38bc3c 24 months ago 4.41MB*

**Step 6:** Pull the ubuntu image from our private docker registry.

*$ docker pull localhost:5000/ubuntu*

*Using default tag: latest*

*latest: Pulling from ubuntu*

*124c757242f8: Pull complete*

*2ebc019eb4e2: Pull complete*

*dac0825f7ffb: Pull complete*

*82b0bb65d1bf: Pull complete*

*ef3b655c7f88: Pull complete*

*Digest: sha256:ac533e4ead4110211a4d67cbf44ed8b7d1aca2b8e6f15d1e8768eadaf433dd31*

*Status: Downloaded newer image for localhost:5000/ubuntu:latest*

Check the image list to find the ubuntu image.

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*registry 2 2d4f4b5309b1 12 days ago 26.2MB*

*localhost:5000/ubuntu latest 16508e5c265d 22 months ago 84.1MB*

*redis latest 4e8db158f18d 23 months ago 83.4MB*

*weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB*

*alpine latest 11cd0b38bc3c 24 months ago 4.41MB*

Likewise, we will be able to push and pull Docker images within our organisation using the private Docker registry as long as the registry container is running in the environment.

**Build an Apache web server application**

If you have followed every resource of this course till here, You would now able to launch any containerized application on your own, test it and expose it to the outside world. In a way of consolidation what you have learnt till now, Let us go ahead and create an Apache web server application in Docker.

In this demo, The Apache web server will be hosted on a CentOS machine. We will start this demo right from creating a Docker Image for the application,

**Step 1:** We will build the image for our application with CentOS as the parent image. Make sure the CentOS image is present in the Docker Host.

*$ docker pull centos*

*Using default tag: latest*

*latest: Pulling from library/centos*

*6910e5a164f7: Pull complete*

*Digest: sha256:4062bbdd1bb0801b0aa38e0f83dece70fb7a5e9bce223423a68de2d8b784b43b*

*Status: Downloaded newer image for centos:latest*

**Step 2:** We have the parent image now. After this, the process can be carried out in two ways. We can either build the web server image using a Dockerfile or in the interactive way. Since interactive way involves a lot of manual commands, we will choose the Docker file method over it. Let us create a Dockerfile first. Create a new directory "webserver" and create a new text file, "Dockerfile".

*$ mkdir webserver*

*$ cd webserver*

*$ touch Dockerfile*

*$ ls*

Dockerfile

**Step 3:** In the webserver directory, create an index.html file for the webserver. This file will be copied from the host to the Docker image when we build it.

*$ vim index.html*

*$ cat index.html*

*<html>*

*<body>*

*<h1>Welcome to Webserver!<h1>*

*<p>You have successfully hosted a web application</p>*

*</body>*

*</html>*

**Step 4:** Edit the Docker file as per our requirement using appropriate Dockerfile commands.

*FROM centos:latest*

*MAINTAINER IMSAcademy*

*RUN yum -y install httpd*

*COPY index.html /var/www/html/*

*CMD [“/usr/sbin/httpd”, “-D”, “FOREGROUND”]*

*EXPOSE 80*

**FROM** - specifies the parent image, centos.

**MAINTAINER** - owner of this Dockerfile.

**RUN** - commands to be run.

**COPY** - file that has to be copied from host to the image.

**CMD** - the commands to be run as soon as the container is launched.

/usr/bin/httpd - starts the webservice.

**-D FOREGROUND** - argument which is used to run the webserver in the background.

**EXPOSE** - specifies the port number where the webserver is running. The default webserver port number 80 is used.

**Step 5:** Build the docker image. Also, add a tag to the image as below.

*$ docker build -t webserver:1 ./webserver*

*Sending build context to Docker daemon 3.072kB*

*Step 1/6 : FROM centos:latest*

*---> 831691599b88*

*Step 2/6 : MAINTAINER IMSAcademy*

*---> Running in 324b65e19c92*

*Removing intermediate container 324b65e19c92*

*---> 38e3b61373be*

*Step 3/6 : RUN yum -y install httpd*

*---> Running in 07f1c494f32c*

*CentOS-8 - AppStream 13 MB/s | 5.8 MB 00:00*

*CentOS-8 - Base 3.8 MB/s | 2.2 MB 00:00*

*CentOS-8 - Extras 17 kB/s | 6.7 kB 00:00*

*Dependencies resolved.*

*================================================================================*

*Package Arch Version Repo Size*

*================================================================================*

*Installing:*

*httpd x86\_64 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 1.7 M*

*Installing dependencies:*

*apr x86\_64 1.6.3-9.el8 AppStream 125 k*

*apr-util x86\_64 1.6.1-6.el8 AppStream 105 k*

*brotli x86\_64 1.0.6-1.el8 BaseOS 323 k*

*centos-logos-httpd*

*noarch 80.5-2.el8 BaseOS 24 k*

*httpd-filesystem noarch 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 36 k*

*httpd-tools x86\_64 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 103 k*

*mailcap noarch 2.1.48-3.el8 BaseOS 39 k*

*mod\_http2 x86\_64 1.11.3-3.module\_el8.2.0+307+4d18d695 AppStream 157 k*

*Installing weak dependencies:*

*apr-util-bdb x86\_64 1.6.1-6.el8 AppStream 25 k*

*apr-util-openssl x86\_64 1.6.1-6.el8 AppStream 27 k*

*Enabling module streams:*

*httpd 2.4*

*Transaction Summary*

*================================================================================*

*Install 11 Packages*

*Total download size: 2.6 M*

*Installed size: 7.5 M*

*Downloading Packages:*

*(1/11): apr-util-bdb-1.6.1-6.el8.x86\_64.rpm 537 kB/s | 25 kB 00:00*

*(2/11): apr-util-openssl-1.6.1-6.el8.x86\_64.rpm 1.2 MB/s | 27 kB 00:00*

*(3/11): apr-1.6.3-9.el8.x86\_64.rpm 1.6 MB/s | 125 kB 00:00*

*(4/11): apr-util-1.6.1-6.el8.x86\_64.rpm 1.2 MB/s | 105 kB 00:00*

*(5/11): httpd-filesystem-2.4.37-21.module\_el8.2 2.6 MB/s | 36 kB 00:00*

*(6/11): httpd-tools-2.4.37-21.module\_el8.2.0+38 3.4 MB/s | 103 kB 00:00*

*(7/11): mod\_http2-1.11.3-3.module\_el8.2.0+307+4 3.0 MB/s | 157 kB 00:00*

*(8/11): centos-logos-httpd-80.5-2.el8.noarch.rp 243 kB/s | 24 kB 00:00*

*(9/11): httpd-2.4.37-21.module\_el8.2.0+382+15b0 7.6 MB/s | 1.7 MB 00:00*

*(10/11): mailcap-2.1.48-3.el8.noarch.rpm 450 kB/s | 39 kB 00:00*

*(11/11): brotli-1.0.6-1.el8.x86\_64.rpm 978 kB/s | 323 kB 00:00*

*--------------------------------------------------------------------------------*

*Total 3.6 MB/s | 2.6 MB 00:00*

*warning: /var/cache/dnf/AppStream-02e86d1c976ab532/packages/apr-1.6.3-9.el8.x86\_64.rpm: Header V3 RSA/SHA256 Signature, key ID 8483c65d: NOKEY*

*CentOS-8 - AppStream 1.6 MB/s | 1.6 kB 00:00*

*Importing GPG key 0x8483C65D:*

*Userid : "CentOS (CentOS Official Signing Key) <security@centos.org>"*

*Fingerprint: 99DB 70FA E1D7 CE22 7FB6 4882 05B5 55B3 8483 C65D*

*From : /etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial*

*Key imported successfully*

*Running transaction check*

*Transaction check succeeded.*

*Running transaction test*

*Transaction test succeeded.*

*Running transaction*

*Preparing : 1/1*

*Installing : apr-1.6.3-9.el8.x86\_64 1/11*

*Running scriptlet: apr-1.6.3-9.el8.x86\_64 1/11*

*Installing : apr-util-bdb-1.6.1-6.el8.x86\_64 2/11*

*Installing : apr-util-openssl-1.6.1-6.el8.x86\_64 3/11*

*Installing : apr-util-1.6.1-6.el8.x86\_64 4/11*

*Running scriptlet: apr-util-1.6.1-6.el8.x86\_64 4/11*

*Installing : httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8. 5/11*

*Installing : mailcap-2.1.48-3.el8.noarch 6/11*

*Installing : centos-logos-httpd-80.5-2.el8.noarch 7/11*

*Installing : brotli-1.0.6-1.el8.x86\_64 8/11*

*Running scriptlet: httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 9/11*

*Installing : httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 9/11*

*Installing : mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86 10/11*

*Installing : httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 11/11*

*Running scriptlet: httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 11/11*

*Verifying : apr-1.6.3-9.el8.x86\_64 1/11*

*Verifying : apr-util-1.6.1-6.el8.x86\_64 2/11*

*Verifying : apr-util-bdb-1.6.1-6.el8.x86\_64 3/11*

*Verifying : apr-util-openssl-1.6.1-6.el8.x86\_64 4/11*

*Verifying : httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 5/11*

*Verifying : httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 6/11*

*Verifying : httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8. 7/11*

*Verifying : mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86 8/11*

*Verifying : brotli-1.0.6-1.el8.x86\_64 9/11*

*Verifying : centos-logos-httpd-80.5-2.el8.noarch 10/11*

*Verifying : mailcap-2.1.48-3.el8.noarch 11/11*

*Installed:*

*apr-1.6.3-9.el8.x86\_64*

*apr-util-1.6.1-6.el8.x86\_64*

*apr-util-bdb-1.6.1-6.el8.x86\_64*

*apr-util-openssl-1.6.1-6.el8.x86\_64*

*brotli-1.0.6-1.el8.x86\_64*

*centos-logos-httpd-80.5-2.el8.noarch*

*httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64*

*httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0afa8.noarch*

*httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64*

*mailcap-2.1.48-3.el8.noarch*

*mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86\_64*

*Complete!*

*Removing intermediate container 07f1c494f32c*

*---> 6bff8362fee0*

*Step 4/6 : COPY index.html /var/www/html/*

*---> 9e947d72dc7f*

*Step 5/6 : CMD ["/usr/sbin/httpd", "-D", "FOREGROUND"]*

*---> Running in 066f114ff05a*

*Removing intermediate container 066f114ff05a*

*---> dc994f9dc0ac*

*Step 6/6 : EXPOSE 80*

*---> Running in a0220eb967c1*

*Removing intermediate container a0220eb967c1*

*---> aabd34dad820*

*Successfully built aabd34dad820*

*Successfully tagged webserver:1*

**-t** sepcifies the tag of the new image.

**./webserver** is the directory where the Dockerfile resides.

**Step 6:** List the Docker images to see the newly created image.

*$ docker images*

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*webserver 1 aabd34dad820 11 seconds ago 254MB*

*centos latest 831691599b88 2 weeks ago 215MB*

*ubuntu latest 16508e5c265d 22 months ago 84.1MB*

*redis latest 4e8db158f18d 23 months ago 83.4MB*

*weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB*

*alpine latest 11cd0b38bc3c 24 months ago 4.41MB*

**Step 7:** We are now good to run a container that is going to host a web application, as we wanted. Expose the port 80 of the container to port 80 of the host so that the web service is available to the outside world too.

*$ docker run -dit -p 80:80 webserver:1*

*5a78ab508974139be4c652f16ee17738dff07d8dc9ca45f37889904b51a5f556*

*-d runs the container in the background*

80:80 exposes the container's port no.80 to the Docker host's port no.80.

**Step 8:** Let us test the web application we just deployed by accessing it in a web browser. You can also use the command below.

*elinks http://192.168.10.10:80*

*192.168.10.10 is the IP address of the Docker host.*

**BASIC OF KUBERNATES LEX NOTES**

**Multiple reasons why people turn up to Containers and Kubernates:**

* Immutability
* Declarative configuration
* Self-Healing
* Decoupling
* Infrastructure Abstraction

**Container Orchestration**

Container orchestration is about managing the containers lifecycle, mainly for large scale deployments. Some of the popular Container orchestration tools are Kubernetes, Docker Swarm, and Apache Mesos. Most of administrators and developers go with container orchestration.

**Container orchestration controls and automates:**

* Provisioning and deployment of containers.
* Scaling up or scaling down containers to balance load evenly across cluster.
* Increase the count of masters or cluster managers based on load.
* If there is a shortage of resources in a host or if host dies, containers can be moved from one host to another.
* Resource allocation between containers.
* Expose services running in a container to external world.
* Monitoring health of containers and hosts.

**Kubernetes:**

Kubernetes is a system for coordinating and running containerized applications across a cluster. Kubernetes is a platform that is designed to manage the lifecycle of containerized applications and services completely.

**What is Kubernetes?**

* It is a project that is managed by CNCF (Cloud Native Computing Foundation).
* Kubernetes is one of the container management and orchestration system.
* It is a Google's open-source container orchestration platform project.
* It is known as the Linux kernel of distributed systems.
* Most importantly, it uses the same API across bare metal, virtual machines or across every cloud provider.

**Things we can do with Kubernetes:**

* We can Manage containerized applications across a cluster of nodes.
* Auto scaling of applications
* Efficient & Robust deployment of application can be achieved.
* Supports easy service deployments like Blue/green deployment, canary deployment - updates and scalability.
* Self-healing application
* We can Overcommit our cluster and we can evict low-priority jobs.
* Stateless and Stateful Applications (databases etc.) can be managed.
* What can be done by whom on which resources can be controlled, Fine-grained access control.
* Integration of third-party services (service catalog).
* Automation of complex tasks (operators).

**Key features that Kubernetes provides:**

* Scheduling: This feature helps in deciding where my containers should run.
* Lifecycle and health: This feature keep containers running despite failures.
* Scaling: The sets of containers can be made bigger or smaller.
* Storage volumes: Provides data to the containers.
* Naming and discovery: We can find where our containers are now.
* Logging and monitoring: We can Track what’s happening with our containers.
* Load balancing: Based on load distribute the traffic across the set of containers.
* Debugging and introspection: We can easily Enter or attach to the containers.
* Identity and authorization: Who can do what things on our containers can be controlled.

**Software deployment and management Automation:**

**What is a Deployment?**

Deployments represents a set of multiple pods that are identical, with no unique identities. Multiple replicas of our application can be run using deployment. In case any instances fail or becomes unresponsive, deployment automatically replaces those instances. Deployment provides self-healing mechanism to address machine failure or maintenance.

**Creating Deployments**

Deployments can be created using following commands.

* kubectl run
* kubectl apply
* kubectl create

**Kubernetes features that support automatic management of deployed software’s/applications:**

**Self-Healing:** Kubernetes Restarts the containers that fail, for containers that don't respond to user defined health checks it kills them, when nodes die it replaces and reschedules the container.

**Automated rollouts and rollbacks:** Changes to the applications or its configuration are rolled out progressively, while monitoring application health to ensure that it doesn't kill all our instances at the same time. In case something goes wrong in the new deployment, Kubernetes will roll back the change.

**Horizontal scaling:** Scaling the application up and down using simple commands, with an UI or based on CPU usage autoscaling.

**Automatic bin packing:** Places containers automatically based on its resource requirements and other constraints, without sacrificing availability.

**Kubernetes Architecture and Basic Components**

Kubernetes has just five basic components:

**Three Master Components :**

* API Server
* Controller manager
* Scheduler

**Two Node Components :**

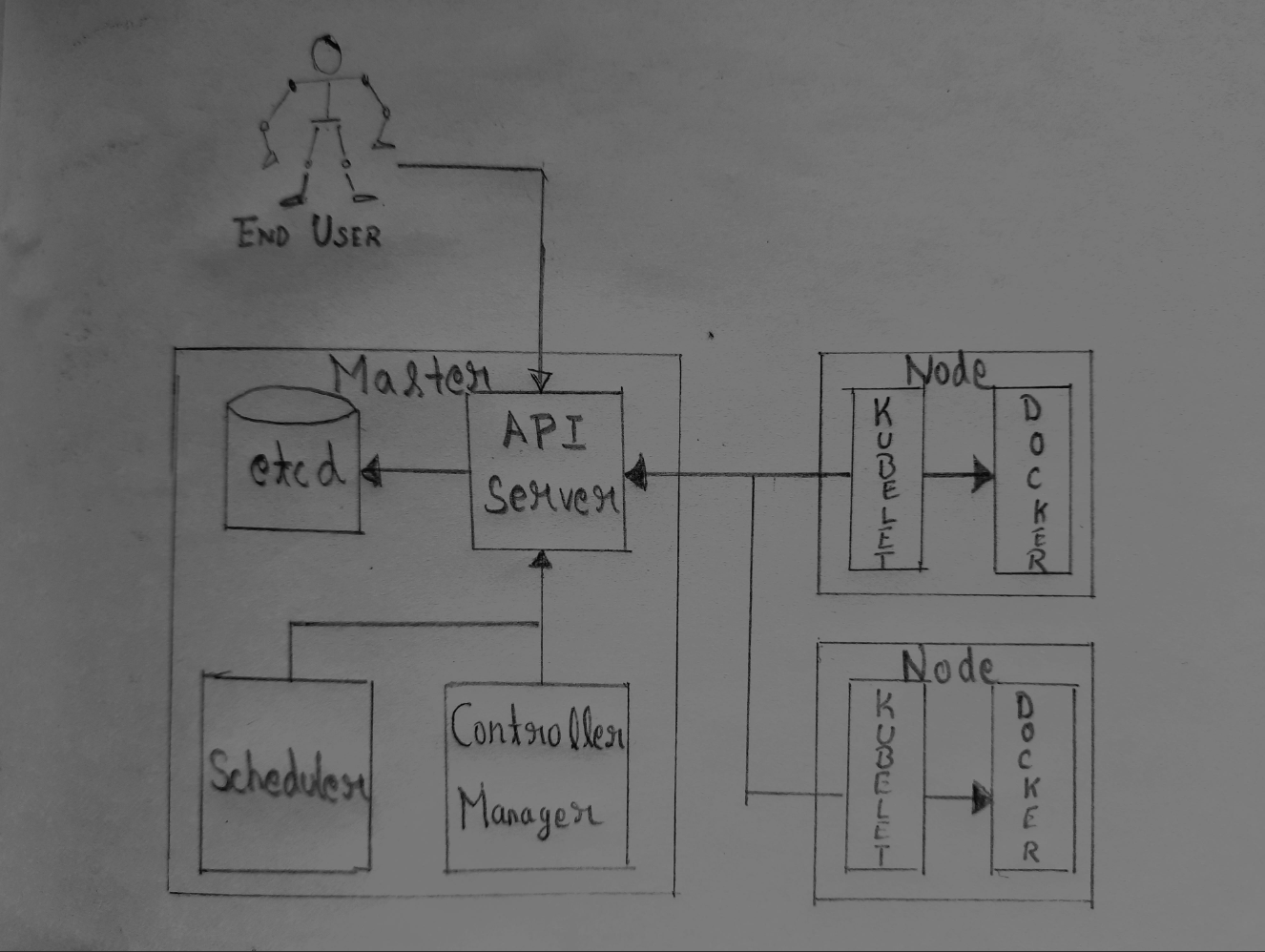
* Kubelet
* Proxy

**The Master Node's**

* The Kubernetes logic is a collection of services:
  + The API server (our point of entry to everything!)
  + Core services like the scheduler and controller manager
  + etcd(a highly available key/value store; the "database" of Kubernetes)
* Kubernetes logic is like the brain.
* All these services together form what is called the "master".
* These services can run in containers or straight on host machine(that's an implementation detail)
* etcd service can be run on separate machines ( i.e. first schema) or co-located ( i.e. second schema)
* At least one master is needed, but we can have more masters (for high availability)

**The Worker Node's**

* The nodes executing our containers run another collection of services:
  + a container Engine (like Docker,rkt,kata etc)
  + kubelet(the "node agent")
  + kube-proxy (it's a necessary but not a suﬃcient network component for nodes)
* It is always recommended to not run applications on the node's that run master components (Except when you are using small development clusters)



**What do kubernetes components do?**

**API Server**

* It provides a forward-facing REST interface into the Kubernetes control plane and datastore.
* Interaction of Kubernetes with all the clients and other applications is strictly through the API Server.
* Four APIs are exposed through API server:
  + Kubernetes API
  + Extensions API
  + Autoscaling API
  + Batch API.
* These components are used for communicating with the Kubernetes cluster and for executing container cluster operations.
* The API Server is the only component that connects to etcd server.
* Every other component to work with cluster state or access, must go through the API Server.
* It handles authentication and authorization, admission control, mutation, and request validation in addition it acts as front-end to the backing data store. It is like a gatekeeper to the cluster.

**Scheduler**

* The responsibility of Scheduler is to monitor the resource utilization of each node and scheduling the containers according to resource availability.
* Default scheduler uses bin packing.
* Requirements can include general hardware requirements, labels, affinity/anti-affinity, and various other custom resource requirements.

**Controller Manager**

* Serves as the primary daemon which manages all the core component control loops.
* Through the API server, it monitors the current state of the applications that are deployed on Kubernetes, and it makes sure that desired state is met.

**etcd**

* It’s a distributed, consistent key-value store and released as an open source by CoreOS.
* It’s used for:
  + Configuration management,
  + service discovery
  + coordinating distributed work
* Kubernetes uses etcd as the persistence storage of all of its API objects (what nodes exist in the cluster, what pods should be running, which nodes they are running on, etc).
* Etcd implements a watch feature, it provides an event-based interface for monitoring changes to keys asynchronously. Through these systems clients are allowed to perform a lightweight subscription for the changes to parts of the key namespace. Once a key is changed, “watchers” get notified immediately. In the context of Kubernetes, this is a crucial feature as the API Server component relies heavily on this to get notified and calls the appropriate business logic components in order to move the current state towards the desired state.

**Kubelet**

* It Acts as the node agent and manages the lifecycle of every pod on its host.
* It uses of the pod specification in order to create containers and manage them.
* Kubelet understands YAML container manifests which it can read from several sources:
  + File path
  + HTTP Endpoint
  + etcd watch acting on any changes.
  + HTTP Server mode accepting container manifests over a simple API.

**Kube-proxy**

* It runs in each node and manages the network rules on each node.
* Performs load balancing or connection forwarding for Kubernetes cluster services.
* It uses proxy modes rules for doing simple load balancing, TCP, UDP stream forwarding and round robin TCP, UDP forwarding.
* Available Proxy Modes:
  + Userspace
  + Iptables
  + IP Virtual Server (ipvs) - default if supported

**What is Etcd?**

Etcd is an open-source distributed key-value store. It is used to store and manage the critical information that is needed to keep distributed systems running. Prominently it manages the configuration data, metadata, and state data.

**Etcd properties:**

* Fully Replicated: The entire data store is available on every node in a cluster.
* Highly Available: Etcd is designed to avoid single points of failure and tolerate hardware failures and network issues.
* Consistent and Reliable: Every data read returns the most recent write across cluster. Using the Raft algorithm, the store is properly distributed.
* Simple: Using standard HTTP/JSON tools, any application, from simple web apps to complex container orchestration engines can read and write data to etcd.
* Secure: Implements automatic TLS with optional client certificate authentication.
* Fast: Etcd is benchmarked at 10,000 writes per second.

**What is Kubernetes scheduler?**

Kubernetes scheduler is a part of Kubernetes container orchestration platform. Scheduler makes sure that Pods are mapped to the Nodes.

Scheduler basically controls Performance, capacity, and availability in kubernetes cluster.

**Kube-scheduler**

The default scheduler for Kubernetes is kube-scheduler. Design of kube-scheduler is such that we can write our own scheduling component and use it.

How does node selection happen?

Node selection is a 2-step operation:

* Filtering
* Scoring

**Filtering** step will find all the nodes where it’s feasible to schedule a specific pod, once this step is done a list is ready with suitable nodes. After this step if list is empty, then that particular pod is not schedulable until required resource is available.

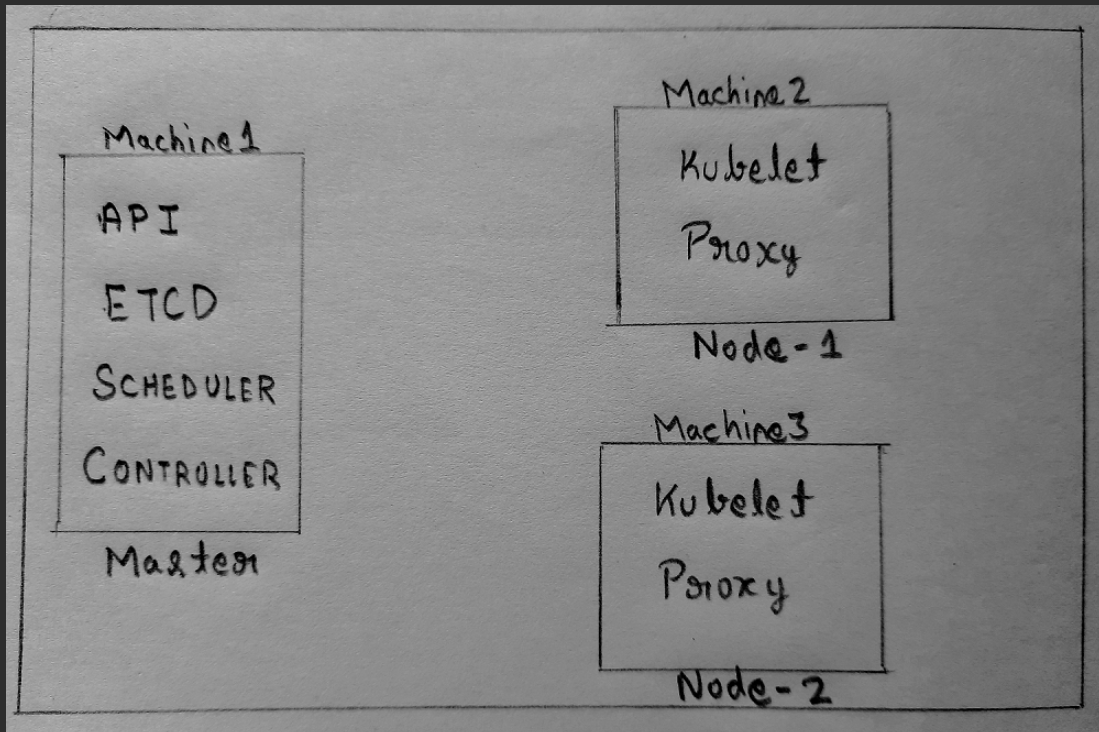
**Scoring** is the step, where the nodes from list in previous step get scores based on which one is most suitable for pod placement.

**Cluster:**

Cluster is a loosely or tightly connected set of computers that work together. They work as a single system. Clusters use nodes to perform the tasks. They are usually managed by masters.

**Kubernetes Cluster** is a group of machines that act as one.

Note: The Masters and Nodes can have different specifications.



# Kubernetes Objects

**Kubernetes Objects**

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of the cluster.

Kubernetes contains a number of abstractions that represent the state of your system i.e. deployed containerized applications and workloads, their associated network and disk resources, and other information about what your cluster is doing.

**The basic Kubernetes objects include:**

* Namespace
* Pod
* Service
* Volume

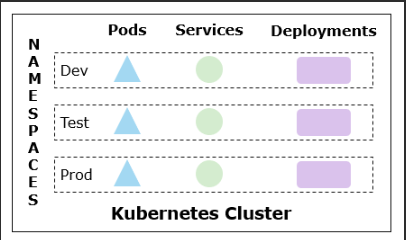
Kubernetes also contains higher-level abstractions that rely on controllers to build upon the basic objects and provide additional functionality and convenience features. These include:

* ReplicaSet
* Deployment
* DaemonSet
* StatefulSet
* Job

**Kubernetes Objects – Namespaces**

Namespaces provide a scope for Kubernetes object names. Many objects such as pods, services and deployments are namespaces, while some (like nodes) are not.

There are 3 user defined namespaces namely Dev, Test and Prod, each having its own set of Kubernetes objects such as Pods, Services, Deployments, etc.



When a Kubernetes cluster is initialized, it has three implicit namespaces:

* **default** - The default namespace for objects with no other namespace
* **kube-system** - The namespace for objects created by the Kubernetes system.
* **kube-public** - This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage.

You can list the current namespaces in a cluster using "**kubectl get ns**" command as well.

**Resource quota** tracks aggregate usage of resources in the Namespace and allows cluster operators to define hard resource usage limits that a Namespace may consume.

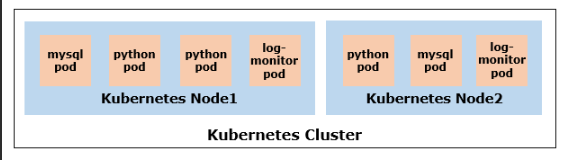
A **limit range** defines min/max constraints on the amount of resources a single entity can consume in a Namespace.

**Kubernetes Objects – Pods**

**A Pod** is the basic execution unit of a Kubernetes application–the smallest and simplest unit in the Kubernetes object model that you create or deploy.

**A Pod** represents processes running on your cluster. Cluster is a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

Below diagram depicts that there are various pods running on each cluster node.



**A Pod** encapsulates an application's container, storage resources, a unique network identity (IP address), as well as options that govern how the containers should run.

**A Pod** represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources.

**Pods in a Kubernetes cluster can be used in two main ways.**

* 1. **Pods that run a single container**

The "one-container-per-Pod" model is the most common Kubernetes use case. In this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly.

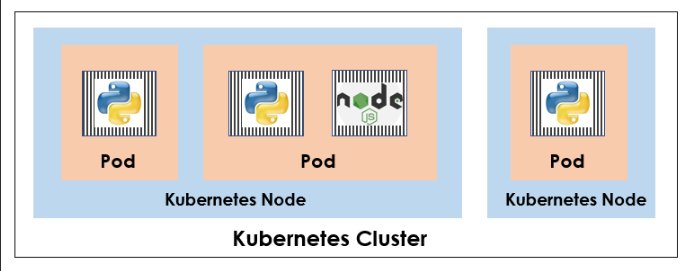
* 1. **Pods that run multiple containers that need to work together.**

A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. This topic will be discussed in Foundation course in detail.

Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (to provide more overall resources by running more instances), you should use multiple Pods, one for each instance. In Kubernetes, this is typically referred to as replication.

**Replicated Pods** are usually created and managed as a group by a workload resource and its controller.

In the below image, each Kubernetes node has one or more Pods and each pod contains one or more containers. It contains Python and Nodejs containers.



**Debugging Pods**

**If the pod is in pending status**

If a pod is stuck in Pending, it means that it cannot be scheduled onto a node. Generally, this is because there are insufficient resources i.e. the node in which it is scheduled exhausted the supply of CPU or Memory in the cluster. In this case you can try several things.

* Add additional nodes to the cluster.
* Terminate unwanted pods to make room for pending pods.
* Check that the pod is not larger than your nodes. For example, if all nodes have a capacity of cpu:1, then a pod with a request of CPU: 1.1 will never be scheduled.

The resource quota feature can be configured to limit the total amount of resources that can be consumed.

**If the pod is in waiting status**

If a pod is stuck in the waiting state, then it has been scheduled to a worker node, but it can't run on that machine. Again, the information from "kubectl describe" is informative in this case. The most common cause of waiting pods is a failure to pull the image. There are three things to check.

* Make sure that you have the name of the image correct.
* Have you pushed the image to the repository?
* Run a manual docker pull to see if the image can be pulled.

**Kubernetes Object Management**

The kubectl command-line tool supports several different ways to create and manage Kubernetes objects. Below are the different approaches.

* Imperative commands
* Imperative object configuration
* Declarative object configuration

**Imperative Command:**

The user provides operations to the kubectl command as arguments. This is the simplest way to get started or to run a one-off task in a cluster as well as generate a definition template easily. Because this technique operates directly on live objects.

Example:

To run an instance of the nginx container by creating a Deployment object:

*$ kubectl run nginx --image nginx*

Do the same thing using a different syntax:

*$ kubectl create deployment nginx --image nginx*

**Kubernetes Object Management - Imperative Object Configuration**

In imperative object configuration, the kubectl command specifies the operation (create, replace, etc.), and at least one file name. The file specified must contain a full definition of the object in YAML or JSON format.

Example:

Create the objects defined in a configuration file:

*$ kubectl create -f nginx.yaml*

"twocontainers" Pod has been created with 2 containers named "simplesrv" and "shell" using this method in the previous slide.

Delete the objects defined in two configuration files:

*$ kubectl delete -f nginx.yaml -f redis.yaml*

Update the objects defined in a configuration file by overwriting the live configuration:

*$ kubectl replace -f nginx.yaml*

**Demo - Genetate YAML File using Imperative Object Management**

Let us see 3 examples in creating a Kubenetes objects in this demo.

Example 1: Create Kubernetes deployment without creating a definition file

Step 1: Below "kubectl create deployment" command will create a pod named "nginx-test-deploy" using nginx image.

[root@k8s-master|/root/yaml]# kubectl create deployment --image=nginx nginx-test-deploy

deployment.apps/nginx-test-deploy created

Step 2: View the deployment (pod) created

*[root@k8s-master|/root/yaml]# kubectl get pods*

*NAME READY STATUS RESTARTS AGE*

*nginx-test-deploy-7d494598b4-q882x 1/1 Running 0 8s*

Example 2:

Now, if wish to generate the yaml configuration file alone for deployment without creating it, use the below two options in combination to generate a resource definition file quickly, then modify and create resources as required, instead of creating the files from scratch.

**--dry-run** By default as soon as the command is run, the resource will be created. If you simply want to test your command, then use the --dry-run option. This will not create the resource, instead, tell you whether the resource can be created and if your command is right

**-o yaml** This will output the resource definition in YAML format on screen

Below command shows the yaml configuration file for deployment on the screen.

*[root@k8s-master|/root/yaml]# kubectl create deployment --image=nginx ngix-dev-deploy --dry-run -o yaml*

*apiVersion: apps/v1*

*kind: Deployment*

*metadata:*

*creationTimestamp: null*

*labels:*

*app: ngix-dev-deploy*

*name: ngix-dev-deploy*

*spec:*

*replicas: 1*

*selector:*

*matchLabels:*

*app: ngix-dev-deploy*

*strategy: {}*

*template:*

*metadata:*

*creationTimestamp: null*

*labels:*

*app: ngix-dev-deploy*

*spec:*

*containers:*

*- image: nginx*

*name: nginx*

*resources: {}*

*status: {}*

Example 3:

If you wish to save the configuration definition in a file rather than showing it on the screen, you can run the below command. You can then edit the YAML file before creating the deployment.

Step 1: Below "kubectl create deployment" command will create a manifest file named "nginx-dev-defn.yaml"

*[root@k8s-master|/root/yaml]# kubectl create deployment --image=nginx ngix-dev-deploy --dry-run -o yaml > nginx-dev-defn.yaml*

Step 2: View the created yaml file content

*[root@k8s-master|/root/yaml]# cat nginx-dev-defn.yaml*

*apiVersion: apps/v1*

*kind: Deployment*

*metadata:*

*creationTimestamp: null*

*labels:*

*app: ngix-dev-deploy*

*name: ngix-dev-deploy*

*spec:*

*replicas: 1*

*selector:*

*matchLabels:*

*app: ngix-dev-deploy*

*strategy: {}*

*template:*

*metadata:*

*creationTimestamp: null*

*labels:*

*app: ngix-dev-deploy*

*spec:*

*containers:*

*- image: nginx*

*name: nginx*

*resources: {}*

*status: {}*

**Demo - Declarative Object Configuration**

When using declarative object configuration, a user operates on object configuration files stored locally, however the user does not define the operations to be taken on the files such as create, update, and delete. These operations are automatically detected per-object by kubectl. This enables working on directories, where different operations might be needed for different objects

Example:

It processes all object configuration files in the pod directory, and create or patch the live objects. You can first diff to see what changes are going to be made, and then apply.

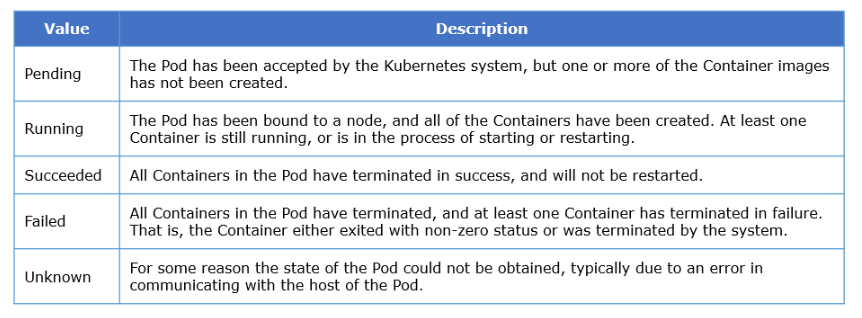
*$ kubectl diff -f pod*

*$ kubectl apply -f pod*

**Pod Lifecycle**

A Pod lifecycle has various states, conditions, policies, etc. A Pod's status field is a PodStatus object, which has a phase field. The phase of a Pod is a simple, high-level summary of where the Pod is in its lifecycle.

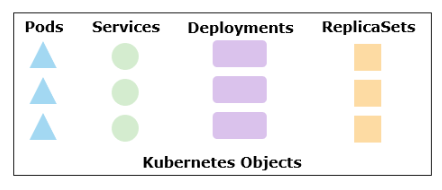
Here are the possible values for phase:



**Kubernetes Objects - Labels and Selectors**

Assume that in your Kubernetes environment, you have created the objects such as Pods, Deployments, Services, ReplicaSets to deploy various applications and its related components. These components may include front-end and back-end instances and monitoring agents with various versions.

Below diagram depicts that you can label and view the Kubernetes objects based on the need.



Once the appropriate labels have been added in the key/value pairs form for the Kubernetes objects, you can also filter the objects based on the application functionalities. For example, you can view all the front-end and back-end pods as shown below.

A diagram of a system

Description automatically generated

You can filter the objects using selectors. For example, application\_name="InfyTel" and functionality="front-end".

Labels are key/value pairs that are attached to Kubernetes objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users. Labels can be used to organize and to select subsets of objects.

Below are few commonly used labels:

* "release" : "stable", "release" : "beta"
* "environment" : "dev", "environment" : "qa", "environment" : "production"
* "tier" : "frontend", "tier" : "backend", "tier" : "cache"
* "partition" : "customerA", "partition" : "customerB"
* "track" : "daily", "track" : "weekly"
* "owner" : "teamCIS", "owner" : "teamDev"

Labels can be attached to objects at creation time and subsequently added and modified at any time. Many objects can carry the same labels.

How to Define Labels

Syntax:

Valid label values must be 63 characters or less and must be empty or begin and end with an alphanumeric character ([a-z0-9A-Z]) with dashes (-), underscores (\_), dots (.), and alpha-numerics between.

Example:

Below is the configuration file for a Pod that has two labels environment: production and app: nginx. Below yaml file creates a Pod named label-demo with the nginx (web server) container to deploy web application in it.

A screenshot of a computer code

Description automatically generated

**Overview of Selectors**

Labels selector are core grouping primitive in Kubernetes. They are used to select a set of objects. --selector option can be used with kubectl command to filter the objects.

Kubernetes API currently supports 2 type of selectors:

* Equality-based selectors
* Set-based selectors

**1. Equality-based Selectors:**

It allows filtering by key and value. Matching objects should satisfy all the specified labels. Three kinds of operators are admitted =, ==, != in matching. The first two represent equality while the latter represents inequality.

**Example**

* environment = production
* tier != frontend

In the above example, the former selects all resources with key equal to environment and value equal to production. The latter selects all resources with key equal to tier and value distinct from frontend, and all resources with no labels with the tier key.

**2. Set-based Selectors:**

It allows filtering of keys according to a set of values. Three kinds of operators are supported: in, notin and exists (only the key identifier).

Example

* environment in (production, qa)
* tier notin (frontend, backend)
* version
* !version

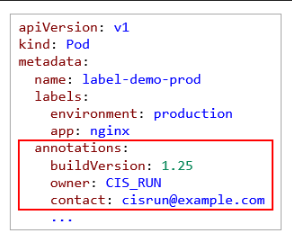
1. The first example selects all resources with key equal to environment and value equal to production or qa
2. The second example selects all resources with key equal to tier and values other than frontend and backend, and all resources with no labels with the tier key
3. The third example selects all resources including a label with key version; no values are checked
4. The fourth example selects all resources without a label with key version; no values are checked

A label selector can be made of multiple requirements which are comma-separated. In the case of multiple requirements, all must be satisfied so the comma separator acts as a logical AND operator.

**Overview of Annotations**

You can use either labels or annotations to attach metadata to Kubernetes objects. Labels can be used to select objects and to find collections of objects that satisfy certain conditions. In contrast, annotations are not used to identify and select objects. The metadata in an annotation can be small or large, structured or unstructured, and can include characters not permitted by labels.

Annotations, like labels, are key/value maps.



Few scenarios where to use annotations:

* Build, release, or image information like timestamps, release IDs, git branch, image hashes, and registry address
* Pointers to logging, monitoring, analytics, or audit repositories
* Client library or tool information that can be used for debugging purposes: for example, name, version, and build information
* User or tool/system provenance information, such as URLs of related objects from other ecosystem components
* Phone or pager numbers of persons responsible, or directory entries that specify where that information can be found, such as a team web site

Let us see the demonstration on annotation.

**Kubernetes Objects – Services**

Assume that there is a web application which has various piece of software components such as front-end instances (containers/pods), backend instances, monitoring instances, etc. All these pods that run on different nodes should connect with other pods to keep the application running.

A diagram of a company

Description automatically generated

**Overview of Services**

A Service in Kubernetes is an abstraction which defines a logical set of pods and a policy by which to access them. It enables a loose coupling between dependent pods. It is created as an object like other Kubernetes objects.

A Service routes traffic across a set of Pods. Services are the abstraction that allow pods to die and replicate in Kubernetes without impacting your application. Discovery and routing among dependent pods such as the front-end and back-end components in an application is handled by Kubernetes Services.

**How service discovers pods:**

Services match a set of pods using labels and selectors. Labels are key/value pairs attached to objects. For example, if you need to logically combine all the pods that runs with the labels "app":"app-a", you can a create a service definition as shown below. So that if any change in the IP address (if the pod is lost and recreated) of the pods which will be automatically discovered by the service through these label.

*# service-defn.yaml*

*apiVersion: v1*

*kind: Service*

*metadata:*

*name: myapp-service*

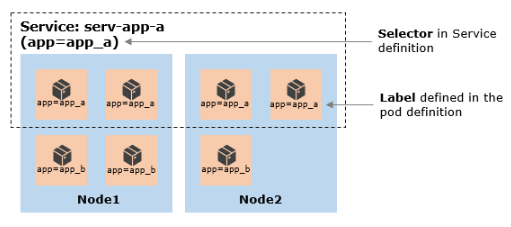
*spec:*

*selector:*

*app: app-a*

*...*

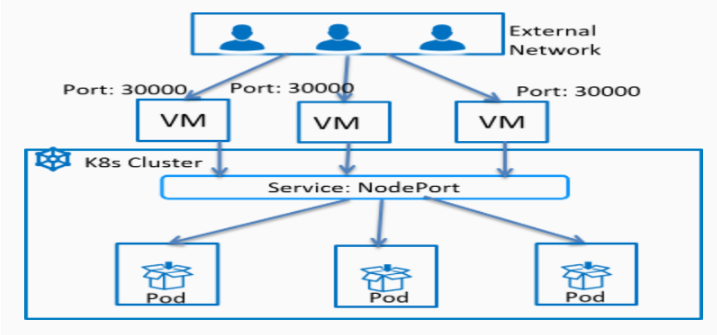
Below diagram depicts the points mentioned above.



**Service Types**

**1. NodePort:**

* It is the type of Kubernetes service that makes an internal pod or group of pods accessible to the external network by opening a specific port on the node machine
* From outside the cluster, one can contact the pod by using curl "<NodeIP>:<NodePort>"
* It exposes the port on every node in the cluster. Port can either be statically defined or dynamically taken from a range between 30000 and 32767



**2. ClusterIP:**

* This is the default service type in Kubernetes
* This service type makes a pod or group of pods accessible to other pods within the internal network of the Kubernetes cluster
* Kubernetes proxy is used to access ClusterIP service from outside the cluster

**3. LoadBalancer:**

* This service type exposes the service externally using the load-balancer functionality of your cloud provider(GCP, AWS, Azure and OpenStack)
* Helps in load balancing the requests/traffic coming towards the group of pods having connected to the service and having same working functionalities

**4. ExternalName:**

* External Name is used to reference endpoints outside the cluster
* It is used to map the Service to the contents of the externalName field
* It creates an internal CNAME DNS entry.

**Demo - Create NodePort Service**

Service is a Kubernetes REST object similar to Pod. It is created as an instance in the Kubernetes cluster and the control plane automatically assigns a IP address and DNS. Using these addresses the external service as well as the pods running inside the cluster can connect and interact with each other.

Let's start writing a Kubernetes Service specification file called "simple-service.yaml".

**Step 1:** Create a file called "simple-service.yaml" and navigate inside that

* Service specification yaml file starts with apiVersion followed by kind, metadata and specs sections
* The apiVersion is set to "v1" and is enabled by default
* The kind field is defined as "Service"
* The metadata field will have a name of the service to be created. Labels can also be added here

*[root@k8s-master|/root/yamls]# vi simple-service.yaml*

*apiVersion: v1*

*kind: Service*

*metadata:*

*name: simple-service*

**Step 2**: Next the "spec" field, as always this is the most crucial part of the file and that is where the actual definition of service lies

It consists of type and ports and selector sub-fields. The type refers to the type of service to be created (NodePort, ClusterIP and LoadBalancer). In this demo, NodePort Service type is implemented.

*apiVersion: v1*

*kind: Service*

*metadata:*

*name: simple-service*

*spec:*

*type: NodePort*

Note: Service specification file without a type field under spec will create service of type Cluster IP by default

**Step 3:** The next part of a spec is ports

* The first type of port ID is the target Port which points to the port number dedicated by the pod
* The second one is a port on the service object. By default it will be set to 80
* The third is NodePort set of the node machine used to connect to External clients/services. The node port range should be between 30000 and 32767

*apiVersion: v1*

*kind: Service*

*metadata:*

*name: simple-service*

*spec:*

*type: NodePort*

*ports:*

*- port: 80*

*targetPort: 8080*

*NodePort: 30180*

Step 4: The selectors in the definition file connects the service to the pods. The .spec.selector field acts a label selector. Pods which are already in the running state as well as any pods and deployments created in future having the same label will be automatically get connected that particular service

For example, the .spec.selector field in the service specification file is mapped to label "app=service-impl-lab", meaning any running pods or future pods having label "service-impl-lab" will get connected to "simple-service" as endpoints.

*apiVersion: v1*

*kind: Service*

*metadata:*

*name: simple-service*

*spec:*

*type: NodePort*

*ports:*

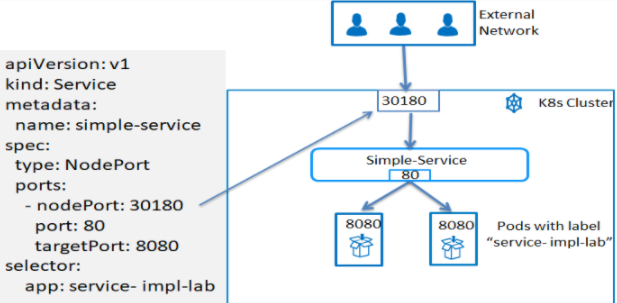
*- port: 80*

*targetPort: 8080*

*nodePort: 30180*

*selector:*

*app: service-impl-lab*

**

**Step 5:** Create a Service using kubectl create command

*[root@k8s-master|/root/yamls]# kubectl create -f simple-service.yaml*

*service/simple-service created*

**Step 6:** View the Service that is created

[root@k8s-master|/root/yamls]# kubectl get svc

*NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE*

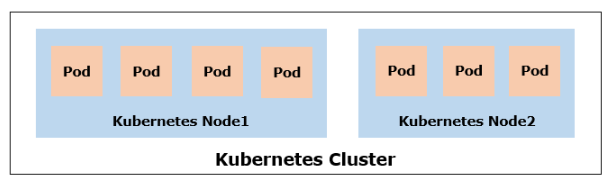
*kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 66m*

*simple-service NodePort 10.105.125.168 <none> 80:30180/TCP 7s*

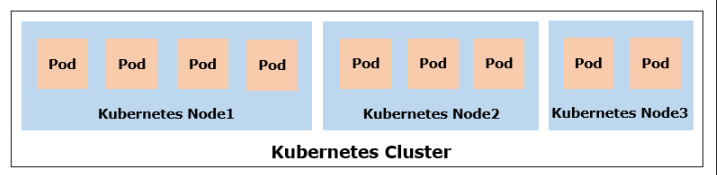
**Kubernetes Objects – ReplicaSet**

**Overview of ReplicaSet**

A ReplicaSet in Kubernetes ensures the stable set of replica Pods running at any given time. Hence, it guarantees the availability of a specified number of identical Pods all the time. It ensures high availability and scalability.



For scenario 1, ReplicaSet will automatically creates the pods in the existing or new node and ensures that the specified number of pods (7) running all time. Thus it ensure the high availability of the web application all the time as shown below.

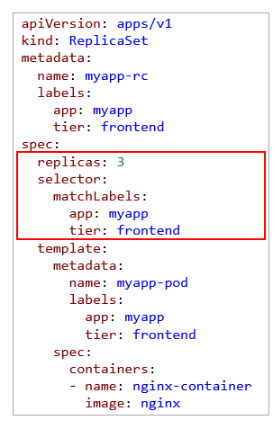


For scenario 2, as the web application's demand increases, you can scale the pods according to the need. Assume that you need to scale the replica to 9, then ReplicaSet will create the additional pods in the cluster to distribute the load on the pods evenly.

**How ReplicaSet works**

Let us understand the various sections of ReplicaSet configuration file.

A ReplicaSet is defined with below fields using which it creates and deletes the Pods as needed to reach the desired number.



**Selector ->** specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining. Below selector code describes that ReplicaSet should identify all the pods that runs with the labels "myapp" and "frontend", also 3 replicas of the pod should be running all the time. It can also identify the pod that is not created by the replicaset.

A screenshot of a computer program

Description automatically generated

**Pod template ->** When a ReplicaSet needs to create new Pods, it uses its Pod template. It specifies the data of new Pods it should create to meet the number of replicas criteria. As per the configuration file mentioned below, it will nginx container with the label "myapp" and "frontend".

**Scale ReplicaSet**

Assume that the demand of the web application increases during the specific period and you need to scale up the replicaset. Similarly, once the demand is over, you can scale down by updating the replicas field.

The ReplicaSet Controller ensures that a desired number of Pods with a matching label selector are available and operational.

You can scale the ReplicaSet in 3 methods.

* 1. **ReplicaSet configuration file**

In this method, update the field "replicas" with the new value and run kubectl with replace option as shown below.

*$ kubectl replace -f <yaml\_file>*

* 1. **scale option with kubectl command**

In this method, you can directly specify the new replicas value using "kubectl" command with "scale" option as shown below. You need not update the yaml file.

*$ kubectl scale --replicas=<n> -f <yaml\_file>*

* 1. **Horizontal Pod AutoScalers (HPA)**

A ReplicaSet can also be a target for Horizontal Pod Autoscalers (HPA). That is, a ReplicaSet can be auto-scaled by an HPA. It automatically scales the number of pods in a replica set, deployment or replication controller based on observed CPU utilization or on some other application-provided metrics.

Let us see the demonstration on scaling the ReplicaSet using configuration file.

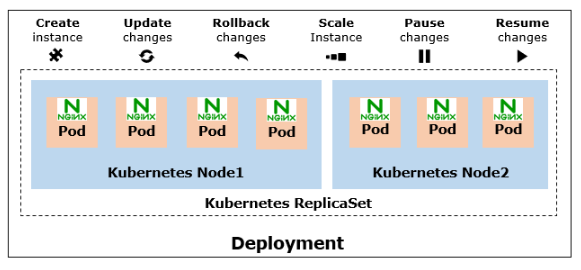
**Kubernetes Objects – Deployment**

**Why Deployment?**

Assume that you need to deploy the identical copies of the the nginx web server containers in the ReplicaSet environment. After creating the instances, you will update the instances if any changes are there in the application. Similarly, you will scale up or down the instance according to the load on the application.

You can also rollback the changes if an unexpected error is occurred during the update. If you wish to change the pod template specification of the deployment or any other configurations, you can pause the changes and resume whenever required.

Below diagram depicts the various operations that can be performed through deployment.



**Demo - Create Deployment**

Configuration file for deployment is same as ReplicaSet configuration except the "kind". In this demo, "simple-webapp" application will be deployed on 2 replicas. You will access this simple web application after the deployment.

Follow the steps mentioned below to create and view the deployment.

**Step 1:** Create a yaml file to create a deployment named "simpleservice-deploy" with 2 nginx pods for hosting frontend web application server

*[root@k8s-master|/root/yamls]# cat deployment-simple-webapp-v1.yaml*

*apiVersion: apps/v1*

*kind: Deployment*

*metadata:*

*name: simpleservice-deploy*

*spec:*

*replicas: 2*

*selector:*

*matchLabels:*

*app: simpleservice*

*template:*

*metadata:*

*labels:*

*app: simpleservice*

*spec:*

*containers:*

*- name: simpleservice*

*image: registry.example.com:5000/simple-webapp:v1*

*ports:*

*- containerPort: 8080*

*env:*

*- name: SIMPLE\_SERVICE\_VERSION*

*value: "1.0"*

Note that simple-webapp:v1 image is used for this deployment.

**Step 2:** Create a deployment using "kubectl create or apply" command

*[root@k8s-master|/root/yamls]# kubectl create -f deployment-simple-webapp-v1.yaml*

*deployment.apps/simpleservice-deploy created*

**Step 3:** You can view the status of the deployment as shown below. There are 2 replicas running for this deployment as per the definition

*[root@k8s-master|/root/yamls]# kubectl get deploy*

*NAME READY UP-TO-DATE AVAILABLE AGE*

*simpleservice-deploy 2/2 2 2 10s*

*[root@k8s-master|/root/yamls]# kubectl get rs*

*NAME DESIRED CURRENT READY AGE*

*simpleservice-deploy-c7c8898bf 2 2 2 16s*

**Step 4:** View the pods as shown below

*[root@k8s-master|/root/yamls]# kubectl get pods -o wide*

*NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES*

*simpleservice-deploy-c7c8898bf-cv7nb 1/1 Running 0 22s 10.244.2.69 worker-node2.pod2.example.com <none> <none>*

*simpleservice-deploy-c7c8898bf-nsf95 1/1 Running 0 22s 10.244.1.68 worker-node1.pod2.example.com <none> <none>*

Each pod runs on a different node with its own IP address.

**Step 5:** You can also deployment, replicaset and pods details altogether using "kubectl get all" command as shown below

*[root@k8s-master|/root/yamls]# kubectl get all*

*NAME READY STATUS RESTARTS AGE*

*pod/simpleservice-deploy-c7c8898bf-cv7nb 1/1 Running 0 17m*

*pod/simpleservice-deploy-c7c8898bf-nsf95 1/1 Running 0 17m*

*NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE*

*service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 15d*

*service/webapp NodePort 10.97.3.30 <none> 8080:32112/TCP 13m*

*NAME READY UP-TO-DATE AVAILABLE AGE*

*deployment.apps/simpleservice-deploy 2/2 2 2 17m*

*NAME DESIRED CURRENT READY AGE*

*replicaset.apps/simpleservice-deploy-c7c8898bf 2 2 2 17m*

Note the IP address of the pod to access it.

**Step 6:** Now you can access this application inside the cluster as shown below using "curl" command. The pod that runs on worker-node2 is accessed through its IP address

*[root@k8s-master|/root/yamls]# curl 10.244.2.69:8080*

*<!DOCTYPE html>*

*<html lang="en">*

*<head>*

*<meta charset="utf-8">*

*<title>Deployment Demonstration</title>*

*<meta name="viewport" content="width=device-width, initial-scale=1.0">*

*<style>*

*HTML{height:100%;}*

*BODY{font-family:Helvetica,Arial;display:flex;display:-webkit-flex;align-items:center;justify-content:center;-webkit-align-items:center;-webkit-box-align:center;-webkit-justify-content:center;height:100%;}*

*.box{background:#f2c341;color:white;text-align:center;border-radius:10px;display:inline-block;}*

*H1{font-size:10em;line-height:1.5em;margin:0 0.5em;}*

*H2{margin-top:0;}*

*</style>*

*</head>*

*<body>*

*<div class="box"><h1>v1</h1><h2></h2>*

*<p>The date today is : <b> 08-07-2020 </b> </p>*

*<p>And the time is : <b> 10:26:49 </b> </p>*

*<p>This is served from POD : <b> simpleservice-deploy-c7c8898bf-cv7nb </b> </p>*

*<p>and the POD IP is :<b> 10.244.2.69 </b> </p>*

*</div>*

*</body>*

*</html>*

**Step 7:** Now if you wish to access it from outside the cluster, then you need to expose it as a service.

*[root@k8s-master|/root/yamls]# kubectl expose deployment simpleservice-deploy --type=NodePort --name=webapp*

Deployment "simpleservice-deploy" has been exposed as a service named "webapp". Service webapp has been created with the nodePort 32112 as shown below.

*[root@k8s-master|/root/yamls]# kubectl get services/webapp -o wide*

*NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE SELECTOR*

*webapp NodePort 10.97.3.30 <none> 8080:32112/TCP 27m app=simpleservice*

Kubernetes Service will be discussed in a different resource in detail.

**Step 8:** Export the nodePort of the service

*[root@k8s-master|/root/yamls]# export NODE\_PORT=$(kubectl get services/webapp -o go-template='{{(index .spec.ports 0).nodePort}}')*

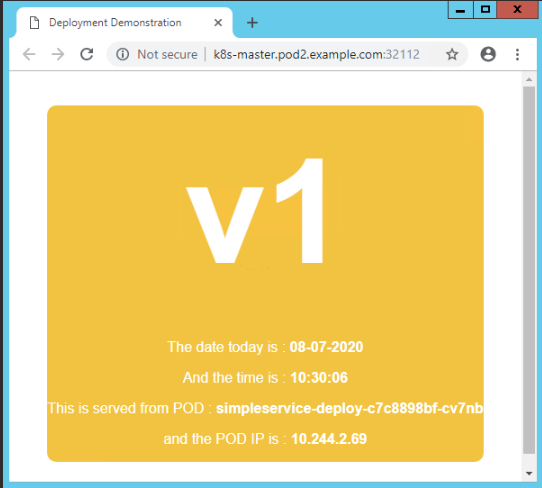
A NodePort is an open port on every node of your cluster. Kubernetes transparently routes incoming traffic on the NodePort to your service, even if your application is running on a different node.

**Step 9:** View the Pod URL with the nodePort that is exported. You can access the application through this URL

*[root@k8s-master|/root/yamls]# echo ${PODURL}:${NODE\_PORT}/*

*http://k8s-master.pod2.example.com:32112/*

**Step 10:** Open any browser and access the simple web application through this URL



**Demo - Scale Deployment**

Assume that the load on the simple web application is increased due to high user base of the company. Then, you need to scale the deployment according to the need to balance the load. You can scale a deployment by using the following command.

**Step 1:** Now 2 replicas of the pod run for deployment as shown below

*[root@k8s-master|/root/yamls]# kubectl get deploy*

*NAME READY UP-TO-DATE AVAILABLE AGE*

*simpleservice-deploy 2/2 2 2 35m*

*[root@k8s-master|/root/yamls]# kubectl get rs*

*NAME DESIRED CURRENT READY AGE*

*simpleservice-deploy-c7c8898bf 2 2 2 35m*

*[root@k8s-master|/root/yamls]# kubectl get pods -o wide*

*NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES*

*simpleservice-deploy-c7c8898bf-cv7nb 1/1 Running 0 35m 10.244.2.69 worker-node2.pod2.example.com <none> <none>*

*simpleservice-deploy-c7c8898bf-nsf95 1/1 Running 0 35m 10.244.1.68 worker-node1.pod2.example.com <none> <none>*

**Step 2**: Due to load, you need to add 2 more replicas in the deployment. Scale the deployment using "--replicas" field in "kubectl scale" command. It is similar to replicaset scaling

*[root@k8s-master|/root/yamls]# kubectl scale deployment.apps/simpleservice-deploy --replicas=4*

*deployment.apps/simpleservice-deploy scaled*

**Step 3:** View the status of the deployment, replicaset and pods. There are 4 pods run in the replicaset as shown below

*[root@k8s-master|/root/yamls]# kubectl get all*

*NAME READY STATUS RESTARTS AGE*

*pod/simpleservice-deploy-c7c8898bf-2xvlb 1/1 Running 0 2m11s*

*pod/simpleservice-deploy-c7c8898bf-b69zn 1/1 Running 0 2m11s*

*pod/simpleservice-deploy-c7c8898bf-cv7nb 1/1 Running 0 37m*

*pod/simpleservice-deploy-c7c8898bf-nsf95 1/1 Running 0 37m*

*NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE*

*service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 15d*

*service/webapp NodePort 10.97.3.30 <none> 8080:32112/TCP 33m*

*NAME READY UP-TO-DATE AVAILABLE AGE*

*deployment.apps/simpleservice-deploy 4/4 4 4 37m*

*NAME DESIRED CURRENT READY AGE*

*replicaset.apps/simpleservice-deploy-c7c8898bf 4 4 4 37m*

**Step 4:** View the pods with it's IP address and the node in which it runs

*[root@k8s-master|/root/yamls]# kubectl get pods -o wide NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES*

*simpleservice-deploy-c7c8898bf-2xvlb 1/1 Running 0 2m51s 10.244.1.69 worker-node1.pod2.example.com <none> <none>*

*simpleservice-deploy-c7c8898bf-b69zn 1/1 Running 0 2m51s 10.244.2.70 worker-node2.pod2.example.com <none> <none>*

*simpleservice-deploy-c7c8898bf-cv7nb 1/1 Running 0 38m 10.244.2.69 worker-node2.pod2.example.com <none> <none>*

*simpleservice-deploy-c7c8898bf-nsf95 1/1 Running 0 38m 10.244.1.68 worker-node1.pod2.example.com <none> <none>*

Along with 2 existing pods i.e. cv7nb and nsf95, 2 more pods 2xvlb and b69zn are created newly.

**Demo - Update Deployment**

Assume that there is a change in the simple web application that is already deployed. The existing instances should be upgraded with the new version of the simple-webapp image (v2 instead of v1). Follow the steps given below to update the deployment.

Step 1: View the image used for the deployment "simpleservice-deploy" as shown below. It is simple-webapp:v1

*[root@k8s-master|/root/yamls]# kubectl get deploy -o wide*

*NAME READY UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR*

*simpleservice-deploy 4/4 4 4 43m simpleservice registry.example.com:5000/simple-webapp:v1 app=simpleservice*

Step 2: Create a yaml file using the new image "simple-webapp:v2" as shown below

*[root@k8s-master|/root/yamls]# cat deployment-simple-webapp-v2.yaml*

*apiVersion: apps/v1*

*kind: Deployment*

*metadata:*

*name: simpleservice-deploy*

*spec:*

*replicas: 4*

*selector:*

*matchLabels:*

*app: simpleservice*

*template:*

*metadata:*

*labels:*

*app: simpleservice*

*spec:*

*containers:*

*- name: simpleservice*

*image: registry.example.com:5000/simple-webapp:v2*

*ports:*

*- containerPort: 8080*

*env:*

*- name: SIMPLE\_SERVICE\_VERSION*

*value: "2.0"*

You can edit the image name of the current deployment configuration by running the below command.

*[root@k8s-master|/root/yaml]# kubectl set image deployment.app/simpleservice-deploy simpleservice=simple-webapp:v2 --record*

*simpleservice -> is the name of the container.*

Alternatively, you can edit the Deployment and change .spec.template.spec.containers.image from simple-webapp:v1 to simple-webapp:v2 using "kubectl edit -f <deployment\_name>" command to update the deployment.

**Step 3:** Run the deployment

*[root@k8s-master|/root/yamls]# kubectl apply -f deployment-simple-webapp-v2.yaml*

*Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply*

*deployment.apps/simpleservice-deploy configured*

**Step 4:** You can check the status of the rollout as shown below

*[root@k8s-master|/root/yamls]# kubectl rollout status deploy/simpleservice-deploy*

*deployment "simpleservice-deploy" successfully rolled out*

**Step 5:** View the status of the deployment or replicaset

*[root@k8s-master|/root/yamls]# kubectl get deploy -o wide*

*NAME READY UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR*

*simpleservice-deploy 4/4 4 4 58m simpleservice registry.example.com:5000/simple-webapp:v2 app=simpleservice*

[root@k8s-master|/root/yamls]# kubectl get rs

NAME DESIRED CURRENT READY AGE

simpleservice-deploy-68bbcbc6bf 4 4 4 7m23s

simpleservice-deploy-c7c8898bf 0 0 0 58m

The older one "c7c8898bf" replicaset is automatically terminated once the new replicas are deployed using the latest version 2 as shown above.

**Step 6:** Below are the pods new replicas created as part of the deployment

[*root@k8s-master|/root/yamls]# kubectl get pods -o wide*

*NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES*

*simpleservice-deploy-68bbcbc6bf-2k62g 1/1 Running 0 11m 10.244.1.71 worker-node1.pod2.example.com <none> <none>*

*simpleservice-deploy-68bbcbc6bf-7brcl 1/1 Running 0 11m 10.244.2.71 worker-node2.pod2.example.com <none> <none>*

*simpleservice-deploy-68bbcbc6bf-gblr2 1/1 Running 0 11m 10.244.1.70 worker-node1.pod2.example.com <none> <none>*

*simpleservice-deploy-68bbcbc6bf-xlclh 1/1 Running 0 11m 10.244.2.72 worker-node2.pod2.example.com <none> <none>*

**Step 7:** You can also view the history of the rollout as shown below. So far, 2 revisions happened on the deployment simpleservice-deploy

*[root@k8s-master|/root/yamls]# kubectl rollout history deploy/simpleservice-deploy*

*deployment.apps/simpleservice-deploy*

*REVISION CHANGE-CAUSE*

*1 <none>*

*2 <none>*

**Step 8:** Access the simple application through browser. You can see the change in the service on the same port as shown below

**A screenshot of a computer

Description automatically generated**

**Demo - Rollback Deployment**

Let us take the new version deployed in the previous demo. There are 2 revisions happened on simpleservice-deploy. For some reason if you wish to go back to the previous version i.e. you need to undo the deployment, follow the steps mentioned below.

**Step 1:** View the current image used for the deployment and also revision happened

*[root@k8s-master|/root/yamls]# kubectl get deploy -o wide*

*NAME READY UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR*

*simpleservice-deploy 4/4 4 4 70m simpleservice registry.example.com:5000/simple-webapp:v2 app=simpleservice*

*[root@k8s-master|/root/yamls]# kubectl rollout history deploy/simpleservice-deploy*

*deployment.apps/simpleservice-deploy*

*REVISION CHANGE-CAUSE*

*1 <none>*

*2 <none>*

**Step 2:** Now explicitly roll back to a specific revision i.e. undo the deployment

*[root@k8s-master|/root/yamls]# kubectl rollout undo deploy/simpleservice-deploy --to-revision=1*

*deployment.apps/simpleservice-deploy rolled back*

**Step 3:** View the status of the deployment or replicaset

*[root@k8s-master|/root/yamls]# kubectl get deploy -o wide*

*NAME READY UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR*

*simpleservice-deploy 4/4 4 4 71m simpleservice registry.example.com:5000/simple-webapp:v1 app=simpleservice*

*See that the new replicaset "68bbcbc6bf" is replaced by older one "c7c8898bf" as shown below.*

*[root@k8s-master|/root/yamls]# kubectl get rs*

*NAME DESIRED CURRENT READY AGE*

*simpleservice-deploy-68bbcbc6bf 0 0 0 22m*

*simpleservice-deploy-c7c8898bf 4 4 4 73m*

**Step 4:** View the new pods that are running. 4 new pods are created, each node has 2 pods

*[root@k8s-master|/root/yamls]# kubectl get pods -o wide*

*NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES*

*simpleservice-deploy-c7c8898bf-fvzs2 1/1 Running 0 18s 10.244.1.72 worker-node1.pod2.example.com <none> <none>*

*simpleservice-deploy-c7c8898bf-h2c95 1/1 Running 0 16s 10.244.1.73 worker-node1.pod2.example.com <none> <none>*

*simpleservice-deploy-c7c8898bf-pzfhn 1/1 Running 0 18s 10.244.2.73 worker-node2.pod2.example.com <none> <none>*

*simpleservice-deploy-c7c8898bf-xb2gf 1/1 Running 0 16s 10.244.2.74 worker-node2.pod2.example.com <none> <none>*

**Step 5:** View the rollout history

*[root@k8s-master|/root/yamls]# kubectl rollout history deploy/simpleservice-deploy*

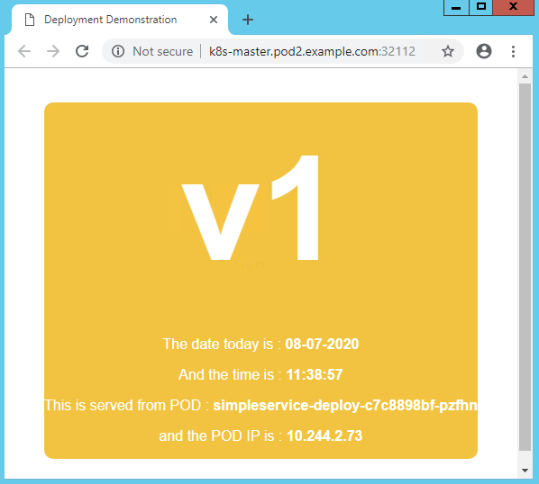
*deployment.apps/simpleservice-deploy*

*REVISION CHANGE-CAUSE*

*2 <none>*

*3 <none>*

**Step 6:** Access the simple application through browser. You can see the change in the service (older version - v1) on the same port as shown below.



**Demo - Rollback Deployment**

For some reason, if you wish to rollback the changes applied on the deployment, you can use "kubectl rollback undo" command. To simulate the issue in the update, here image nginx:1.17.1 is used and it does not exist in the worker-nodes. Follow the steps given below to rollback the deployment.

**Step 1:** View the image used for the deployment "myapp-dep" as shown below. It is nginx:1.16.1

*[root@k8s-master|/root/yaml]# kubectl get deploy -o wide*

*NAME READY UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR*

*myapp-dep 3/3 3 3 8s myapp-nginx-cont docker.io/nginx:1.16.1 app=myapp,tier=frontend*

*[root@k8s-master|/root/yaml]# kubectl get rs*

*NAME DESIRED CURRENT READY AGE*

*myapp-dep-b89fbcc48 3 3 3 12s*

*[root@k8s-master|/root/yaml]# kubectl get pods*

*NAME READY STATUS RESTARTS AGE*

*myapp-dep-b89fbcc48-8jwd2 1/1 Running 0 14s*

*myapp-dep-b89fbcc48-nfpm9 1/1 Running 0 14s*

*myapp-dep-b89fbcc48-nlcn7 1/1 Running 0 14s*

**Step 2:** update the deployment to the image nginx:1.17.1

*[root@k8s-master|/root/yaml]# kubectl set image deployment.app/myapp-dep myapp-nginx-cont=nginx:1.17.1 --record*

*deployment.apps/myapp-dep image updated*

*myapp-nginx-container -> is the name of the container.*

**Step 3:** You can check the status of the rollout as shown below

*[root@k8s-master|/root/yaml]# kubectl rollout status deployment.apps/myapp-dep*

*Waiting for deployment "myapp-dep" rollout to finish: 1 out of 3 new replicas have been updated...*

**Step 4:** View the status of the deployment or replicaset

*[root@k8s-master|/root/yaml]# kubectl get deploy -o wide*

*NAME READY UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR*

*myapp-dep 3/3 1 3 3m45s myapp-nginx-cont nginx:1.17.1 app=myapp,tier=frontend*

*[root@k8s-master|/root/yaml]# kubectl get rs*

*NAME DESIRED CURRENT READY AGE*

*myapp-dep-b89fbcc48 3 3 3 3m54s*

*myapp-dep-d599ff956 1 1 0 35s*

*[root@k8s-master|/root/yaml]# kubectl get pods*

*NAME READY STATUS RESTARTS AGE*

*myapp-dep-b89fbcc48-8jwd2 1/1 Running 0 3m56s*

*myapp-dep-b89fbcc48-nfpm9 1/1 Running 0 3m56s*

*myapp-dep-b89fbcc48-nlcn7 1/1 Running 0 3m56s*

*myapp-dep-d599ff956-sg7rh 0/1 ErrImagePull 0 37s*

Look at the Pods created, you see that 1 Pod created by new ReplicaSet is stuck in an image pull loop. The Deployment controller stops the bad rollout automatically, and stops scaling up the new ReplicaSet.

To fix this, you need to rollback to a previous revision of Deployment that is stable.

Step 5: Check the rollout history (revision) of a deployment as shown below

*[root@k8s-master|/root/yaml]# kubectl rollout history deployment.apps/myapp-dep*

*deployment.apps/myapp-dep*

*REVISION CHANGE-CAUSE*

*1 <none>*

*2 kubectl set image deployment.app/myapp-dep myapp-nginx-cont=nginx:1.16.1 --record=true*

*3 kubectl set image deployment.app/myapp-dep myapp-nginx-cont=nginx:1.17.1 --record=true*

You can also view the details of the specific revision using "--revision=2" in the above command.

**Step 6:** Undo the current rollout and rollback to the previous revision

*[root@k8s-master|/root/yaml]# kubectl rollout undo deployment.apps/myapp-dep*

*deployment.apps/myapp-dep rolled back*

You can also rollback to a specific revision by specifying it with "--to-revision=2" in the above command.

Step 7: Check whether the rollback was successful and the deployment is running as expected. The pod "myapp-dep-d599ff956" created as part of the version "nginx:1.17.1" is automatically deleted and deployment is back to nginx:1.16.1 image

*[root@k8s-master|/root/yaml]# kubectl get deploy -o wide*

*NAME READY UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR*

*myapp-dep 3/3 3 3 12m myapp-nginx-cont docker.io/nginx:1.16.1 app=myapp,tier=frontend*

*[root@k8s-master|/root/yaml]# kubectl get rs*

*NAME DESIRED CURRENT READY AGE*

*myapp-dep-b89fbcc48 3 3 3 11m*

*myapp-dep-d599ff956 0 0 0 8m31s*

*[root@k8s-master|/root/yaml]# kubectl get pods*

*NAME READY STATUS RESTARTS AGE*

*myapp-dep-b89fbcc48-8jwd2 1/1 Running 0 11m*

*myapp-dep-b89fbcc48-nfpm9 1/1 Running 0 11m*

*myapp-dep-b89fbcc48-nlcn7 1/1 Running 0 11m*

**Demo - Pause and Resume 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.

**Step 1:** You can pause the current deployment

*[root@k8s-master|/root/yaml]# kubectl rollout pause deployment.apps/myapp-dep*

Step 2: After pausing the deployment, you can do the required changes in the image, resource, etc. Few examples are listed below

*$ kubectl set image deployment.apps/myapp-dep myapp-nginx-cont=nginx:1.16.2*

*$ kubectl set resources deployment.apps/myapp-dep -c=nginx --limits=cpu=250m,memory=512Mi*

**Step 3:** Once the changes are done, resume the deployment

*[root@k8s-master|/root/yaml]# kubectl rollout resume deployment.apps/myapp-dep*

Step 4: Check the status of the rollout until it is done

*[root@k8s-master|/root/yaml]# kubectl get rs -w*

Note that you cannot rollback a paused Deployment until you resume it.

Delete Deployment

To delete the deployment, run the below command.

*[root@k8s-master|/root/yamls]# kubectl delete deploy simpleservice-deploy*

Let us next test your understanding on Kubernetes Deployment.

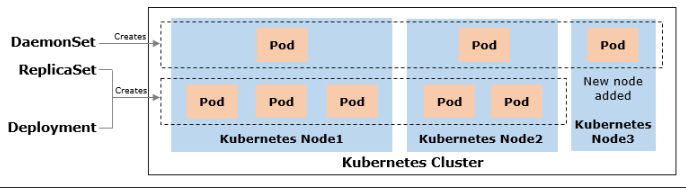
**Kubernetes Objects – DaemonSet**

**Why DaemonSet?**

So far, you have created a replicaset and deployments in the cluster environment. Assume that you need to deploy 5 identical copies of the web server pod on 3 node cluster. Then multiple pods run on the same node to ensure the desired number of pods (5) running at any given time.

Assume that you have created an image to run the log collection or monitoring daemon on every node of a cluster, then you can create a DaemonSet to accomplish the same. If you are in a need to monitor the newly added node in the cluster without manually installing any agent related software separately, then you can use DaemonSet. It will automatically create a pod in the new node like replicaset or deployment and only one copy of the DaemonSet runs in every cluster node.

Below diagram depicts the points that is mentioned above.



A DaemonSet ensures that all or some nodes run a copy of a pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

When to use of a DaemonSet:

* running a cluster storage daemon on every cluster node
* running a logs collection daemon on every cluster node
* running a node monitoring daemon on every cluster node

**Demo - Create DaemonSet**

In this demo, DaemonSet named "myapp-ds" is created to deploy the fluentd pod to collect the logs on every node. Follow the steps mentioned below to create the same.

**Step 1:** Create a yaml file for creating a DaemonSet as shown below, it is same as ReplicaSet except 2 "kind" and "replicas" fields. kind is of DaemonSet and replicas field is not applicable for DaemonSet

*[root@k8s-master|/root/yaml]# cat ds.yaml*

*apiVersion: apps/v1*

*kind: DaemonSet*

*metadata:*

*name: myapp-ds*

*labels:*

*app: myapp-agent*

*agent: log-collect*

*spec:*

*selector:*

*matchLabels:*

*app: myapp-agent*

*agent: log-collect*

*template:*

*metadata:*

*name: myapp-pod*

*labels:*

*app: myapp-agent*

*agent: log-collect*

*spec:*

*containers:*

*- name: myapp-log-cont*

*image: docker.io/fluentd:latest*

**Step 2:** Run the DaemonSet

*[root@k8s-master|/root/yaml]# kubectl create -f ds.yaml*

*daemonset.apps/myapp-ds created*

**Step 3:** You can view the DaemonSet

*[root@k8s-master|/root/yaml]# kubectl get ds -o wide*

*NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE CONTAINERS IMAGES SELECTOR*

*myapp-ds 2 2 2 2 2 <none> 3s myapp-log-cont fluentd agent=log-collect,app=myapp-agent*

**Step 4:** View the Pods of Deployment (created 3 replicas already) and DaemonSet. You can see multiple pods are running in the same node (node2) for deployment whereas only one pod runs in every node for DaemonSet

*[root@k8s-master|/root/yaml]# kubectl get pods -o wide*

*NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES*

*myapp-ds-7s44x 1/1 Running 0 12m 10.244.1.49 worker-node1.pod2.example.com <none> <none>*

*myapp-ds-hrjr8 1/1 Running 0 12m 10.244.2.53 worker-node2.pod2.example.com <none> <none>*

*myapp-dep-b89fbcc48-7lzrf 1/1 Running 0 13m 10.244.2.52 worker-node2.pod2.example.com <none> <none>*

*myapp-dep-b89fbcc48-sgwmf 1/1 Running 0 13m 10.244.2.51 worker-node2.pod2.example.com <none> <none>*

*myapp-dep-b89fbcc48-fjj56 1/1 Running 0 13m 10.244.1.48 worker-node1.pod2.example.com <none> <none>*

**DaemonSet Fields**

Let us understand few DaemonSet fields.

**Pod selector:**

If the .spec.selector is specified, it must match the .spec.template.metadata.labels. Config with these not matching will be rejected by the API.

**Running pods on select nodes:**

As mentioned you can run the DaemonSet pods on all or some nodes in the cluster. If you need to create in the specific node, then you need to mention the node details using nodeSelector or nodeAffinity fields of the pod template, then the DaemonSet controller will create Pods on nodes which match that node selector or affinity. These two fields will be discussed later in Kubernetes Foundation course.

**Rolling update on a DaemonSet:**

You can also update the DaemonSet in Kubernetes version 1.6 or later. DaemonSet has 2 update strategy types.

1. **OnDelete:**

With OnDelete update strategy, after you update a DaemonSet template, new DaemonSet pods will only be created when you manually delete old DaemonSet pods.

1. **RollingUpdate:**

This is the default update strategy. With RollingUpdate update strategy, after you update a DaemonSet template, old DaemonSet pods will be killed, and new DaemonSet pods will be created automatically, in a controlled fashion.

Refer Kubernetes documentation to know more about rolling update on a DaemonSet.

**Delete a DaemonSet:**

You can delete the DaemonSet if it is not required. It deletes the pods associated with the DaemonSet.

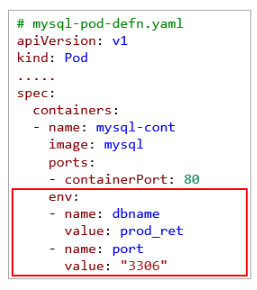
*[root@k8s-master|/root/yaml]# kubectl delete ds myapp-ds*

*daemonset.apps "myapp-ds" deleted*

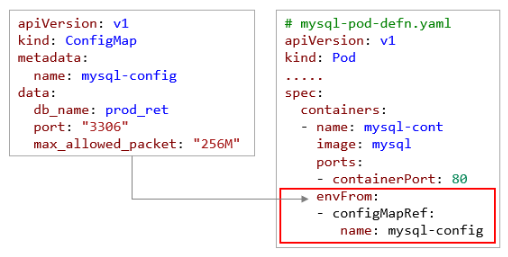
**Kubernetes Objects – ConfigMap**

**Why ConfigMap?**

Assume that you have various micro services such as frontend (Python), backend (MySQL), in-memory database (Redis), etc. in your application. There are many variables that need to be passed to all these (pods) containers while composing it to run the application. If few variables are there in each pod then it can be defined in the pod definition itself using "env" property as shown below.



It is not a recommended method if the number of variables for each pod is huge as it makes the pod definition file lengthy and complex. In this scenario, you can define all the variables in a ConfigMap definition and add it in a pod definition as shown below.



You can store this information in a central location and reference this from an application. When you want to change the configuration, simply change it in the file and you are good to go. You need not search every location where the data is referenced.

If confidential values need to be passed such SSH key, database password, you can use Kubernetes Secret. ConfigMap is used to store non-confidential data in key-value pairs.

**Overview of ConfigMap**

A ConfigMap is an API object used to store non-confidential data in key-value pairs and allow other objects (Pod) to use. Pods can consume ConfigMaps in the below 3 methods.

1. As an environment variables
2. Command-line arguments
3. Configuration files in a volume

**When to use ConfigMap:**

Use a ConfigMap for setting configuration data separately from application code.

**How to define ConfigMap object:**

Unlike most Kubernetes objects that have a "spec", a ConfigMap has a "data" section to store items (keys) and their values. ConfigMap can be created in either imperative or declarative method.

**Note**: ConfigMap does not provide secrecy or encryption. If the data you want to store are confidential, use a Secret rather than a ConfigMap, or use additional (third party) tools to keep your data private.

**Demo - Create ConfigMap (Imperative)**

Follow the steps mentioned below to create a ConfigMap using imperative method.

**Step 1:** Create a ConfigMap using "kubectl create config" command

*[root@k8s-master|/root/yaml]# kubectl create configmap \*

*> mysqlp-config --from-literal=db\_name=prod\_ret \*

*> --from-literal=port=3306 \*

*> --from-literal=max\_allowed\_packet=256M*

*configmap/mysqlp-config created*

mysql-config -> name of the configmap

--from-literal -> use it to specify each key-value pairs

You can also define these variables in a file and use the it through "--from-file" option in the above command.

**Step 2:** View the ConfigMap created

*[root@k8s-master|/root/yaml]# kubectl get configmap*

*NAME DATA AGE*

*mysqlp-config 3 109s*

**Step 3:** Describe the ConfigMap to view the key-value pairs as shown below

*[root@k8s-master|/root/yaml]# kubectl describe cm*

*Name: mysqlp-config*

*Namespace: default*

*Labels: <none>*

*Annotations: <none>*

*Data*

*====*

*port:*

*----*

*3306*

*db\_name:*

*----*

*prod\_ret*

*max\_allowed\_packet:*

*----*

*256M*

*Events: <none>*

**Demo - Create ConfigMap (Declarative)**

Follow the steps mentioned below to create a ConfigMap using declarative method.

Step 1: Create a ConfigMap definition file named "mysql-config" to store mysql environment related variables with the value as shown below

*[root@k8s-master|/root/yaml]# cat mysql-config.yaml*

*apiVersion: v1*

*kind: ConfigMap*

*metadata:*

*name: mysql-config*

*data:*

*db\_name: prod\_ret*

*port: "3306"*

*max\_allowed\_packet: "256M"*

Step 2: Run the "kubectl create" command to create a ConfigMap

*[root@k8s-master|/root/yaml]# kubectl create -f mysql-config.yaml*

*configmap/mysql-config created*

You can also use "kubectl apply" command.

Step 3: View the ConfigMap created, it has 3 variables

*[root@k8s-master|/root/yaml]# kubectl get cm*

*NAME DATA AGE*

*mysql-config 3 6s*

Step 4: Describe the ConfigMap to view the variable details

*[root@k8s-master|/root/yaml]# kubectl describe cm*

*Name: mysql-config*

*Namespace: default*

*Labels: <none>*

*Annotations: <none>*

*Data*

*====*

*db\_name:*

*----*

*prod\_ret*

*max\_allowed\_packet:*

*----*

*256M*

*port:*

*----*

*3306*

*Events: <none>*

**Demo - Inject ConfigMap as an Environment Variable**

To use the ConfigMap in pod, both ConfigMap and Pod must be in the same namespace. Follow the steps mentioned below to use the ConfigMap variables as an environment variables inside a pod.

**Step 1:** Refer the ConfigMap "mysql-config" that is created already in a Pod using "envFrom" property

*apiVersion: v1*

*kind: Pod*

*metadata:*

*name: mysql-prod*

*labels:*

*environment: prod*

*app: backend*

*spec:*

*containers:*

*- name: mysql-cont*

*image: mysql*

*ports:*

*- containerPort: 80*

*envFrom:*

*- configMapRef:*

*name: mysql-config*

**Step 2:** Run "kubectl" to create a pod as shown below

*[root@k8s-master|/root/yaml]# kubectl create -f prod\_mysql\_pod.yaml*

*pod/mysql-prod created*

**Step 3:** View the pod that is running

*[root@k8s-master|/root/yaml]# kubectl get pods*

*NAME READY STATUS RESTARTS AGE*

*mysql-prod 1/1 Running 0 29s*

**Step 4:** Connect to the pod and run the "env" command as shown below. Observe that all 3 ConfigMap variables are added in the environment variables

*[root@k8s-master|/root/yaml]# kubectl exec -it mysql-prod sh*

*# env*

*KUBERNETES\_SERVICE\_PORT=443*

*KUBERNETES\_PORT=tcp://10.96.0.1:443*

*HOSTNAME=mysql-prod*

*PHP\_APACHE\_PORT\_80\_TCP=tcp://10.102.217.138:80*

*HOME=/root*

*db\_name=prod\_ret*

*max\_allowed\_packet=256M*

*TERM=xterm*

*KUBERNETES\_PORT\_443\_TCP\_ADDR=10.96.0.1*

*NGINX\_VERSION=1.13.12-1~stretch*

*PHP\_APACHE\_SERVICE\_HOST=10.102.217.138*

*PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin*

*port=3306*

You can also check this as a command-line argument as shown below.

*[root@k8s-master|/root/yaml]# kubectl exec -it mysql-prod -- /bin/sh -c 'echo "$db\_name\n$port\n$max\_allowed\_packet"'*

*prod\_ret*

*3306*

*256M*

In this example, all the variables have been injected into the container environment. Also, the name of the container variable is same as ConfigMap variable. If required, you can inject few ConfigMap variables in the container and change the name of the variable as well.

**Demo - Inject ConfigMap as Command-line Argument**

In the previous demo, If you wish to change the name of the variable while accessing it through the Pod, then you can customize the name of the ConfigMap variables.

Follow the steps mentioned below to use the ConfigMap variables as a command-line arguments inside a pod.

**Step 1:** Refer the ConfigMap "mysql-config" that is created already in a Pod using "env" property

*[root@k8s-master|/root/yaml]# cat prod\_mysql\_pod.yaml*

*apiVersion: v1*

*kind: Pod*

*metadata:*

*name: mysql-prod*

*labels:*

*environment: prod*

*app: backend*

*spec:*

*containers:*

*- name: mysql-cont*

*image: mysql*

*ports:*

*- containerPort: 80*

*env:*

*- name: MYSQL\_DB\_NAME*

*valueFrom:*

*configMapKeyRef:*

*name: mysql-config*

*key: db\_name*

*- name: MYSQL\_DB\_PORT*

*valueFrom:*

*configMapKeyRef:*

*name: mysql-config*

*key: port*

In the above pod definition, only db\_name and port variables have injected from the ConfigMap "mysql-config". Also, ConfigMap variables db\_name and port have been changed into MYSQL\_DB\_NAME and MYSQL\_DB\_PORT respectively.

**Step 2:** Run "kubectl" to create a pod as shown below

*[root@k8s-master|/root/yaml]# kubectl create -f prod\_mysql\_pod.yaml*

*pod/mysql-prod created*

**Step 3:** View the pod that is running

*[root@k8s-master|/root/yaml]# kubectl get pods*

*NAME READY STATUS RESTARTS AGE*

*mysql-prod 1/1 Running 0 2s*

**Step 4:** Connect to the pod and access the ConfigMap variables with the new name as a command-line argument as shown below. Only 2 variables have injected in the pod

[*root@k8s-master|/root/yaml]# kubectl exec -it mysql-prod -- /bin/sh -c 'echo "$MYSQL\_DB\_NAME\n$MYSQL\_DB\_PORT"'*

*prod\_ret*

*3306*

**Demo - Inject ConfigMap as Volume**

Follow the steps mentioned below to use the ConfigMap variables as a volume inside a pod.

**Step 1:** Refer the ConfigMap "mysql-config" that is created already in a Pod using volume properties as shown below

*[root@k8s-master|/root/yaml]# cat prod\_mysql\_pod.yaml*

*apiVersion: v1*

*kind: Pod*

*metadata:*

*name: mysql-prod*

*labels:*

*environment: prod*

*app: backend*

*spec:*

*containers:*

*- name: mysql-cont*

*image: mysql*

*ports:*

*- containerPort: 80*

*volumeMounts:*

*- name: "config-volume"*

*mountPath: "/root/config"*

*volumes:*

*- name: "config-volume"*

*configMap:*

*name: mysql-config*

name of the volumeMounts and volumes must be same. You can mount the variables as a file in the volume mouthPath "/root/config".

**Step 2:** Run "kubectl" to create a pod as shown below

*[root@k8s-master|/root/yaml]# kubectl create -f prod\_mysql\_pod.yaml*

*pod/mysql-prod created*

**Step 3:** View the pod that is running

*[root@k8s-master|/root/yaml]# kubectl get pods*

*NAME READY STATUS RESTARTS AGE*

*mysql-prod 1/1 Running 0 2m46s*

Step 4: Connect to the pod as shown below

*[root@k8s-master|/root/yaml]# kubectl exec -it mysql-prod sh*

*#*

**Step 5:** Move to the respective mount path and list the files and it's content as shown below

*# cd /root/config*

*# ls*

*db\_name max\_allowed\_packet port*

*# cat db\_name*

*prod\_ret*

*# cat port*

*3306*

*# cat max\_allowed\_packet*

*256M*

*# exit*

ConfigMap variables are mapped as a volume files in a Pod.

**Kubernetes Objects – Secrets**

**Why Secret?**

There are many confidential details configured with a container image such as OS credential, SSH keys, tokens, web server, application server and database server login passwords, etc. You need to maintain such sensitive details safe. **Secret** in Kubernetes helps to **store the confidential information** safer and more flexible than putting it as a plain text in a Pod definition or in a container image.

You can create a secret in 2 ways.

1. using "kubectl create secret" command (imperative method)
2. using secret definition file (declarative method)

Once the secret is created, it can be injected into a Pod in 2 ways.

1. As a volume in a Pod
2. As container variables in a Pod

**Demo - Create Secret using kubectl create secret Command (Imperative Method)**

In this example, a MongoDB database connection string consists of a username and password. Follow the steps mentioned below to create a secret.

**Step 1**: You can store the username in a file db\_username.txt and the password in a file db\_password.txt on your local machine

*[root@k8s-master|/root/yaml]# echo -n 'db\_username=app\_admin' > db\_username.txt*

*[root@k8s-master|/root/yaml]# echo -n 'db\_password=Infy@123+' > db\_password.txt*

**Step 2:** Create a secret named "mongodb-user-pass" using "kubectl create secret generic" command as shown below

*[root@k8s-master|/root/yaml]# kubectl create secret generic mongodb-user-pass --from-file=db\_username.txt --from-file=db\_password.txt*

*secret/mongodb-user-pass created*

Rather than storing these details in a file, you can also directly mention it in the kubectl command using "--from-literal" with key-value pair.

*[root@k8s-master|/root/yaml]# kubectl create secret generic db-user-pass --from-literal=db\_username=app\_admin --from-literal=db\_password=Infy@123+*

**Step 3:** View the secrets created by running the below command

*root@k8s-master|/root/yaml]# kubectl get secrets*

*NAME TYPE DATA AGE*

*mongodb-user-pass Opaque 2 2m15s*

Opaque -> means it stores the data in the key-value pair.

**Step 4:** You can view a description of the secret as shown below

*[root@k8s-master|/root/yaml]# kubectl describe secret mongodb-user-pass*

*Name: mongodb-user-pass*

*Namespace: default*

*Labels: <none>*

*Annotations: <none>*

*Type: Opaque*

*Data*

*====*

*db\_password.txt: 9 bytes*

*db\_username.txt: 9 bytes*

The commands kubectl get and kubectl describe avoid showing the contents of a secret by default. This is to protect the secret from being exposed accidentally to an onlooker, or from being stored in a terminal log. However, here you can view the plain text stored as is using the files.

**Creating a Secret**

The Secret contains two maps, data and stringData. The data field is used to store arbitrary data, encoded using base64. The stringData field is provided for convenience, and allows you to provide secret data as unencoded strings.

For example, to store mongodb username and password, convert the string to base64 (data) as follows.

*[root@k8s-master|/root/yaml]# echo -n 'app\_admin' | base64*

*YXBwX2FkbWlu*

*[root@k8s-master|/root/yaml]# echo -n 'Infy@123+' | base64*

*SW5meUAxMjMr*

If you need to view the database credentials in an unencoded form, decode it (stringData) as shown below.

*[root@k8s-master|/root/yaml]# echo -n 'YXBwX2FkbWlu' | base64 --decode*

*app\_admin*

*[root@k8s-master|/root/yaml]# echo -n 'SW5meUAxMjMr' | base64 --decode*

*Infy@123+*

**Demo - Create Secret using Definition File (Declarative Method)**

If few strings need to be secured, you can use the kubectl command to store in a Secret. However there are chances for the onlooker to view the secrets using terminal log using this method. Hence, it is not a recommended method. You can store the sensitive information in the encoded form in a secret (kind) definition file if there are many secrets.

Follow the steps mentioned below to create a secret in a declarative method and configure it with the pod.

**Step 1:** Create a secret definition file as shown below using the kind Secret

*[root@k8s-master|/root/yaml]# cat mongodb-secret.yaml*

*apiVersion: v1*

*kind: Secret*

*metadata:*

*name: mongodb-user-pass*

*type: Opaque*

*data:*

*db\_username: YXBwX2FkbWlu*

*db\_password: SW5meUAxMjMr*

For some reason, if you wish to store both data and stringData, you can add "stringData" field in the above yaml file.

**Step 2:** Run kubectl command to create a secret "mongodb-secret"

*[root@k8s-master|/root/yaml]# kubectl apply -f mongodb-secret.yaml*

*secret/mongodb-user-pass created*

**Step 3:** View the secret created as shown below

*[root@k8s-master|/root/yaml]# kubectl get secret*

*NAME TYPE DATA AGE*

*mongodb-user-pass Opaque 2 3s*

**Step 4:** Once the secret is created, it can be injected (referred) using a container variable "envFrom" in the pod definition as shown below

*[root@k8s-master|/root/yaml]# cat qa\_nginx\_pod.yaml*

*apiVersion: v1*

*kind: Pod*

*metadata:*

*name: label-demo-qa*

*labels:*

*environment: quality*

*app: nginx*

*spec:*

*containers:*

*- name: nginx*

*image: nginx*

*ports:*

*- containerPort: 80*

*envFrom:*

*- secretRef:*

*name: mongodb-user-pass*

**Step 5:** Run the pod

*[root@k8s-master|/root/yaml]# kubectl create -f qa\_nginx\_pod.yaml*

*pod/label-demo-qa created*

**Step 6:** View the pod created

*[root@k8s-master|/root/yaml]# kubectl get pods*

*NAME READY STATUS RESTARTS AGE*

*label-demo-qa 1/1 Running 0 6s*

**Step 7:** You can access the secret data inside the container as shown below

*[root@k8s-master|/root/yaml]# kubectl exec -i -t label-demo-qa -- /bin/sh -c 'echo "$db\_username\n$db\_password"'*

*app\_admin*

*Infy@123+*

**Demo - Inject the Secret as a Volume in a Pod**

In the previous demo, secret named "mongodb-secret" is created and it contains mongodb username and password in the data (encoded) form. If these data needs to be mapped to a volume inside the container, you can mount the secret as a volume. Follow the below steps to create a secret using volume.

**Step 1:** Create a secret definition file as shown below using the kind Secret and refer the secret "mongodb-secret" through volume

*[root@k8s-master|/root/yaml]# cat dev\_nginx\_pod.yaml*

*apiVersion: v1*

*kind: Pod*

*metadata:*

*name: label-demo-dev*

*labels:*

*environment: development*

*app: nginx*

*spec:*

*containers:*

*- name: nginx*

*image: nginx*

*ports:*

*- containerPort: 80*

*volumeMounts:*

*- name: "mongodb-secret"*

*mountPath: "/root/mongodb-volume"*

*readOnly: true*

*volumes:*

*- name: "mongodb-secret"*

*secret:*

*secretName: mongodb-user-pass*

name -> name of the volumeMounts and volumes must be similar

mountPath -> path that will container the secrets inside the container

secretName -> name of the secret that is already created

**Step 2:** Run the pod using this yaml file

*[root@k8s-master|/root/yaml]# kubectl apply -f dev\_nginx\_pod.yaml*

*pod/label-demo-dev created*

**Step 3:** View the pod that is created

*[root@k8s-master|/root/yaml]# kubectl get pods*

*NAME READY STATUS RESTARTS AGE*

*label-demo-dev 1/1 Running 0 5s*

Step 4: Connect to the container

**[root@k8s-master|/root/yaml]# kubectl exec -i -t label-demo-dev -- /bin/bash**

**root@label-demo-dev:/#**

Step 5: View the volume path and files inside it with the content

*root@label-demo-dev:/# ls /root/mongodb-volume/*

*db\_username db\_password*

*root@label-demo-dev:/# cat /root/mongodb-volume/db\_username*

*app\_admin*

*root@label-demo-dev:/# cat /root/mongodb-volume/db\_password*

*Infy@123+*

**Demo - Inject the Secret as Container Variables in a Pod**

You can inject the Secret as container environment variables as shown below.

*[root@k8s-master|/root/yaml]# cat dev\_nginx\_pod.yaml*

*apiVersion: v1*

*kind: Pod*

*metadata:*

*name: label-demo-dev*

*labels:*

*environment: development*

*app: nginx*

*spec:*

*containers:*

*- name: nginx*

*image: nginx*

*ports:*

*- containerPort: 80*

*envFrom:*

*- secretRef:*

*name: mongodb-secret*

You can also define a container environment variable with data from a single Secret or multiple Secrets. Refer Kubernetes documentation to know more details on this.