**4. Docker – Containers & Build tool- Maven**

**4.1. Introduction:** What is a Docker, Use case of Docker, Platforms for Docker, Dockers vs. Virtualization

**4.2. Architecture:** Docker Architecture., Understanding the Docker components

**4.3. Installation:** Installing Docker on Linux. Understanding Installation of Docker on windows. Some Docker commands. Provisioning.

**4.4. Docker Hub:** Downloading Docker images. Uploading the images in Docker Registry and AWS ECS, Understanding the containers, Running commands in container. Running multiple containers.

**4.5. Custom images:** Creating a custom image. Running a container from the custom image. Publishing the custom image.

**4.6. Docker Networking:** Accessing containers, linking containers, Exposing container ports, Container Routing.

**What is Docker?**

🡺 Docker is a software development tool and a virtualization technology that makes it easy to develop, deploy, and manage applications by using containers. Container refers to a lightweight, stand-alone, executable package of a piece of software that contains all the libraries, configuration files, dependencies, and other necessary parts to operate the application. In other words, applications run the same irrespective of where they are and what machine they are running on because the container provides the environment throughout the software development life cycle of the application. Since containers are isolated, they provide security, thus allowing multiple containers to run simultaneously on the given host. Also, containers are lightweight because they do not require an extra load of a hypervisor.

**Docker Container**

🡺Containers provide the following benefits:

Reduced IT management resources

Reduced size of snapshots

Quicker spinning up apps

Reduced and simplified security updates

Less code to transfer migrate, and upload workloads

**Docker Snapshot**

🡺A docker that allows you to create snapshots (copies) of the current document, and to return to these states afterwards. The main part of the docker is a list of all saved snapshots.

**Docker Image**

🡺A Docker image is a file used to execute code in a Docker container. Docker images act as a set of instructions to build a Docker container, like a template.

**How Docker works?**

🡺Docker packages, provisions and runs containers. Container technology is available through the operating system: A container packages the application with all of the libraries, configuration files, dependencies and other necessary parts and parameters to operate. Each container shares the services of one basic operating system. Docker images contain all the dependencies needed to execute code inside a container, so containers that move between Docker environments with the same OS work with no changes.

**Use Case for Docker**

1. **Simplifying configuration**:  Docker provides the capability to put your environment and configuration in a code and deploy it. The benefit of it is that you can use the same Docker configuration in multiple environments. This minimizes infrastructure requirements from the application environment.

           (In short you can run your application across multiple IAAS/PAAS).

1. **Code Pipeline Management**: As the code travels from developers’ machine to the production, it has to go through many environments, and each environments have its own differences. Docker provides a consistent environment for the application from dev through production, easing the code development and deployment pipeline.
2. **App Isolations**: There are some scenarios where a developer has to use multiple applications on the same machine, so what docker do is that it combines multiple servers, which eventually lead to save cost.
3. **Debugging Capabilities**: Docker provides many tools that are not necessarily specific to containers, but they work well with the concept of containers. This includes the ability to checkpoint containers and container versions, as well as to diff two containers. This can be immensely useful in fixing an application.
4. **Multi-tenancy**: Using Docker, it was easy and inexpensive to create isolated environments for running multiple instances of app tiers for each tenant.

**Platforms for Docker**

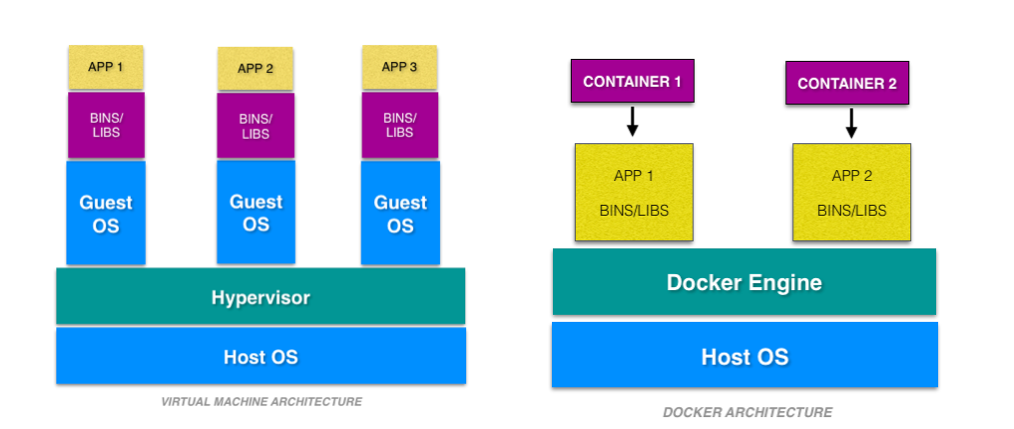
🡺The Docker platform runs natively on Linux (on x86-64, ARM and many other CPU architectures) and on Windows (x86-64). Docker Inc. builds products that let you build and run containers on Linux, Windows and macOS.

**VM vs Docker**

|  |  |  |
| --- | --- | --- |
|  | **Docker** | **Virtual Machine (VMs)** |
| **Boot time** | Boots in a few seconds. | It takes a few minutes for VMs to boot. |
| **Runs on** | Dockers make use of the execution engine. | VMs make use of the hypervisor. |
| **Memory Efficiency** | No space is needed to virtualize, hence less memory. | Requires entire OS to be loaded before starting the surface, so less efficient. |
| **Isolation** | Prone to adversities as no provisions for isolation systems. | Interference possibility is minimum because of the efficient isolation mechanism. |
| **Deployment** | Deploying is easy as only a single image, containerized can be used across all platforms. | Deployment is comparatively lengthy as separate instances are responsible for execution. |
| **Usage** | Docker has a complex usage mechanism consisting of both third party and docker managed tools. | Tools are easy to use and simpler to work with. |

SOME MORE CONSTRAINTS

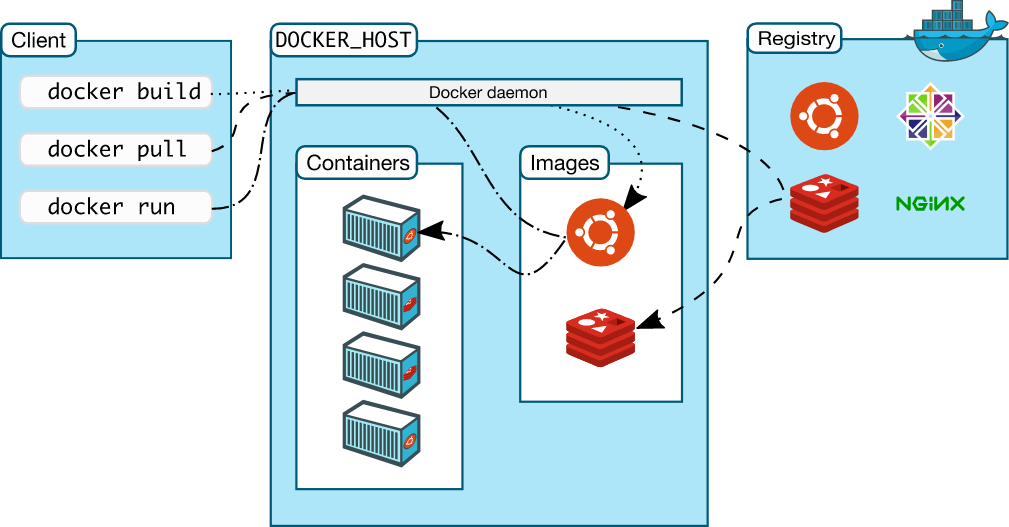
OS Support and Architecture



Virtual machines have host OS and the guest OS inside each VM. Guest OS can be any OS, like Linux or Windows, irrespective of host OS. In contrast, Docker containers host on a single physical server with a host OS, which shares among them. Sharing the host OS between containers makes them light and increases the boot time. Docker containers are considered suitable to run multiple applications over a single OS kernel; whereas, virtual machines are needed if the applications or services required to run on different OS.

**Architecture**

**Docker architecture:**



Docker uses a client-server architecture. The Docker client talks to the Docker daemon.

**Docker Daemon:** The Docker daemon listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. The daemon and the client communicate using Rest API.

**Docker Client**: The Docker client is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to daemon, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

**Docker Desktop**: Docker Desktop is an application that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon, the Docker client etc.

**Docker Registries**:A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

**Docker Objects:** When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects.

1. **Images**: An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization.
2. **Containers**: A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

**Installing Docker on Linux**

Install Docker

Log into your system as a user with sudo privileges.

Update your system: sudo yum update -y .

Install Docker: sudo yum install docker-engine -y.

Start Docker: sudo service docker start.

Verify Docker: sudo docker run hello-world.

**Step-By-Step Docker Installation on Windows**

**1.** Go to the website https://docs.docker.com/docker-for-windows/install/ and download the docker file. Note: A 64-bit processor and 4GB system RAM are the hardware prerequisites required to successfully run Docker on Windows 10.

**2.** Then, double-click on the Docker Desktop Installer.exe to run the installer.

Note: Suppose the installer (Docker Desktop Installer.exe) is not downloaded; you can get it from Docker Hub and run it whenever required.

**3.** Once you start the installation process, always enable Hyper-V Windows Feature on the Configuration page.

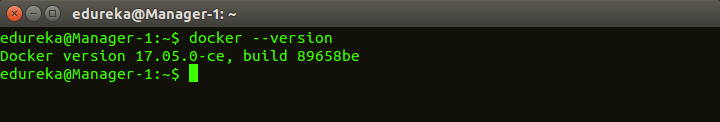
**4.** Then, follow the installation process to allow the installer and wait till the process is done.

**5.** After completion of the installation process, click Close and restart.

## **Docker Commands**

1.**docker –version**

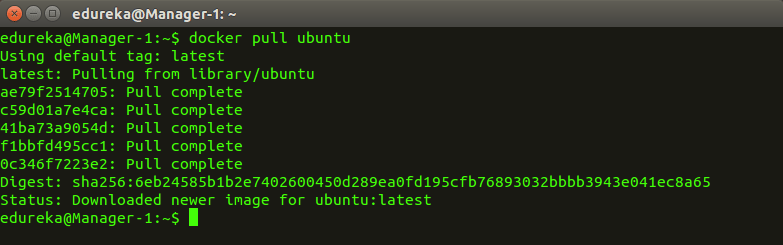
This command is used to get the currently installed version of docker



2.**docker pull**

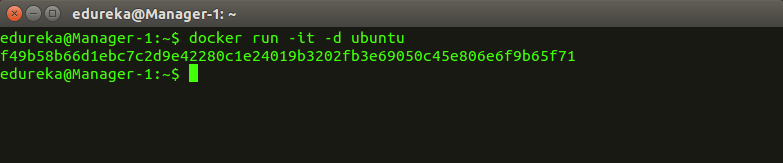
**Usage: docker pull <image name>**

This command is used to pull images from the **docker repository**(hub.docker.com)

  
  
3. **docker run**

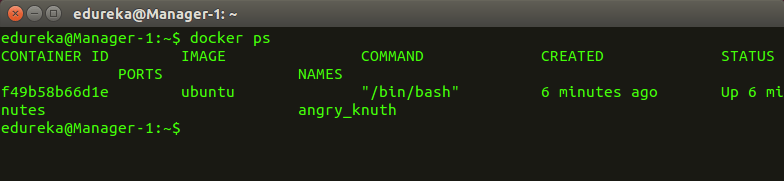
**Usage: docker run -it -d <image name>**

This command is used to create a container from an image

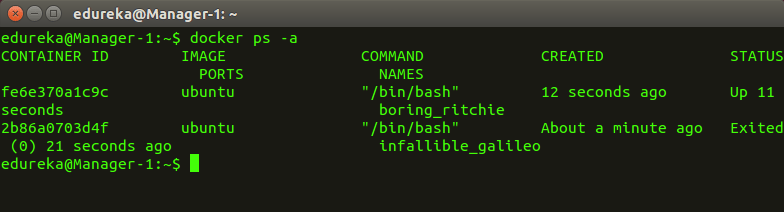


4. **docker ps**

This command is used to list the running containers

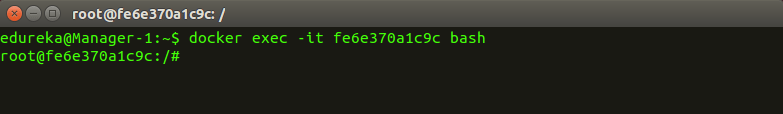
  
  
5. **docker ps -a**

This command is used to show all the running and exited containers

  
  
6. **docker exec**

**Usage: docker exec -it <container id> bash**

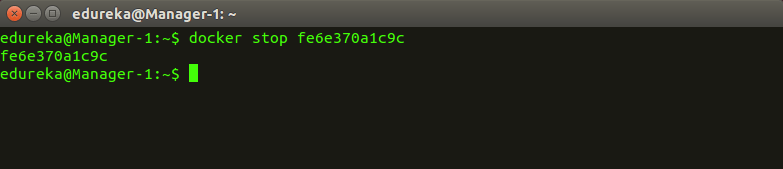
This command is used to access the running container



7. **docker stop**

**Usage: docker stop <container id>**

This command stops a running container



8. **docker kill**

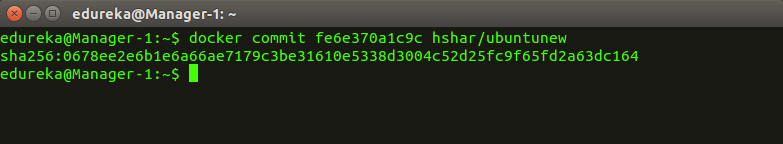
**Usage: docker kill <container id>**

This command kills the container by stopping its execution immediately. The difference between ‘docker kill’ and ‘docker stop’ is that ‘docker stop’ gives the container time to shutdown gracefully, in situations when it is taking too much time for getting the container to stop, one can opt to kill it

  
  
9. **docker commit**

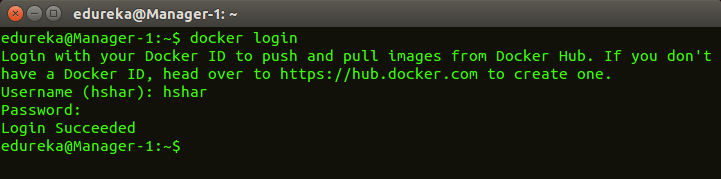
**Usage: docker commit <conatainer id> <username/imagename>**

This command creates a new image of an edited container on the local system



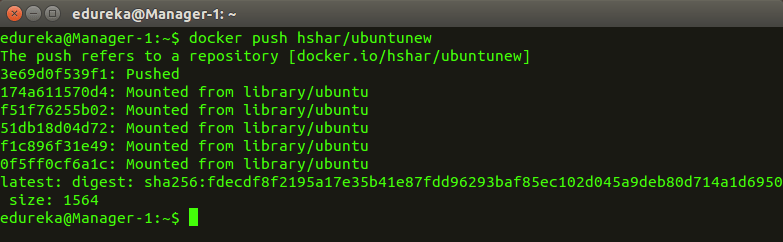
10. **docker login**

This command is used to login to the docker hub repository

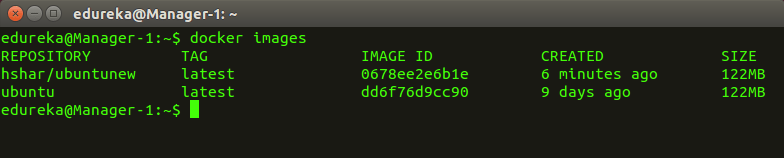
  
  
11.**docker push**

**Usage: docker push <username/image name>**

This command is used to push an image to the docker hub repository

  
  
12. **docker images**

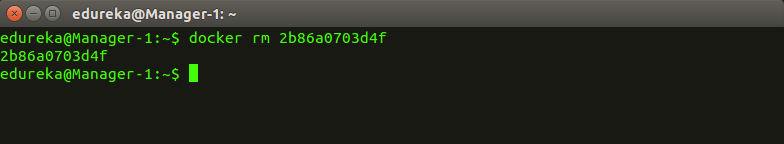
This command lists all the locally stored docker images



13. **docker rm**

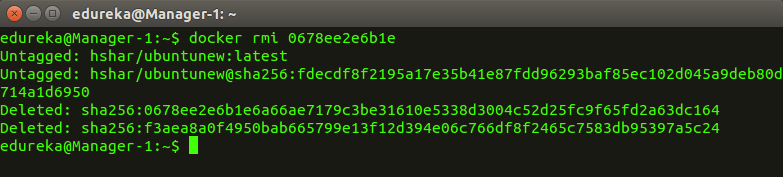
**Usage: docker rm <container id>**

This command is used to delete a stopped container

  
  
14. **docker rmi**

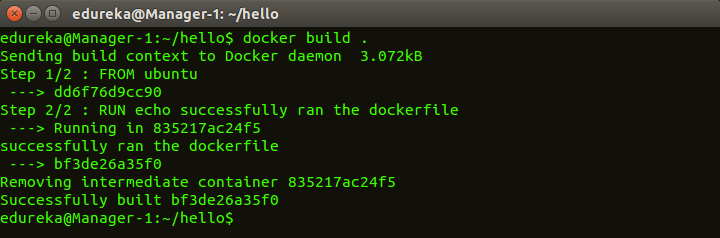
**Usage: docker rmi <image-id>**

This command is used to delete an image from local storage

  
  
15. **docker build**

**Usage: docker build <path to docker file>**

This command is used to build an image from a specified docker file



Docker Based Provisioning

Docker-based provisioning involves the creation of a Micro gateway Docker file from an existing installation, building the image, and running it multiple times in a container environment

# **Docker - Hub**

Docker Hub is a registry service on the cloud that allows you to download Docker images that are built by other communities. You can also upload your own Docker built images to Docker hub.

**Downloading Docker Images**

Pull an image from Docker Hub

To download a particular image, or set of images use docker pull. If no tag is provided, Docker Engine uses the :latest image:

$ docker pull debian

Using default tag: latest

latest: Pulling from library/debian

fdd5d7827f33: Pull complete

a3ed95caeb02: Pull complete

Digest: sha256:e7d38b3517548a1c71e41bffe9c8ae6d6d29546ce46bf62159837aad072c90aa

Status: Downloaded newer image for debian:latest

Docker images can consist of multiple layers. In the example above, the image consists of two layers; fdd5d7827f33 and a3ed95caeb02.

Layers can be reused by images. For example, the debian:jessie image shares both layers with debian:latest. Pulling the debian:jessie image therefore only pulls its metadata, but not its layers, because all layers are already present locally: [

$ docker pull debian:jessie

jessie: Pulling from library/debian

fdd5d7827f33: Already exists

a3ed95caeb02: Already exists

Digest: sha256:a9c958be96d7d40df920e7041608f2f017af81800ca5ad23e327bc402626b58e

Status: Downloaded newer image for debian:jessie

**Create and Upload a Docker Image**

1.Run docker login images.sbgenomics.com and enter your Seven Bridges credentials. Don't forget: enter your [authentication token](https://docs.sevenbridges.com/docs/get-your-authentication-token) when prompted for a password.

2. Open a Docker base image.

3. Install your tool in the image.

4. Commit your image.

5. Push your image to the Seven Bridges registry.

**To push a Docker image to an Amazon ECR repository**

1. Create the Docker image.

2. Create an ECR registry.

3. Tag the image.

4. Give the Docker CLI permission to access your Amazon account.

5. Upload your docker image to ECR.

6. Create a Fargate Cluster for ECS to use for the deployment of your container.

7. Create an ECS Task.

8. Run the ECS Task!

**Docker Containers.**

Docker Container is a standardized unit which can be created on the fly to deploy a particular application or environment. It could be an Ubuntu container, CentOs container, etc. to full-fill the requirement from an operating system point of view. Also, it could be an application oriented container like CakePHP container or a Tomcat-Ubuntu container etc.

* Containers are fast and boots quickly as it uses host operating system and shares the relevant libraries.
* Containers do not waste or block host resources unlike virtual machines.
* Containers have isolated libraries and binaries specific to the application they are running.
* Containers are handled by Containerization engine.
* Docker is one of the containerization platforms which can be used to create and run containers.

Container Commands:

When developing or deploying containers you’ll often need to look inside a running container to inspect its current state or debug a problem. To this end, Docker provides the docker exec command to run programs in containers that are already running.

|  |  |
| --- | --- |
| **Command** | **Description** |
| docker container attach | Attach local standard input, output, and error streams to a running container |
| docker container commit | Create a new image from a container’s changes |
| docker container cp | Copy files/folders between a container and the local filesystem |
| docker container create | Create a new container |
| docker container diff | Inspect changes to files or directories on a container’s filesystem |
| docker container exec | Run a command in a running container |
| docker container export | Export a container’s filesystem as a tar archive |
| docker container inspect | Display detailed information on one or more containers |
| docker container kill | Kill one or more running containers |
| docker container logs | Fetch the logs of a container |
| docker container ls | List containers |
| docker container pause | Pause all processes within one or more containers |
| docker container port | List port mappings or a specific mapping for the container |
| docker container prune | Remove all stopped containers |
| docker container rename | Rename a container |
| docker container restart | Restart one or more containers |
| docker container rm | Remove one or more containers |
| docker container run | Run a command in a new container |
| docker container start | Start one or more stopped containers |
| docker container stats | Display a live stream of container(s) resource usage statistics |
| docker container stop | Stop one or more running containers |
| docker container top | Display the running processes of a container |
| docker container unpause | Unpause all processes within one or more containers |
| docker container update | Update configuration of one or more containers |
| docker container wait | Block until one or more containers stop, then print their exit codes |

**Running Multiple Containers**

Docker Compose

It is the tool used to handle the Multiple Containers

Create a docker-compose.yaml file that defines the services (containers) that make up your application. So they can be run together in an isolated environment. In this compose file, we define all the configurations that need to build and run the services as docker containers.

There are several steps to follow to use docker-compose.

1. Split your app into services

The first thing to do is to think about how you’re going to divide the components of your application into different services(containers).

In a simple client-server web application, it could contain three main layers (frontend, backend, and the database). So we can split the app in that way. Likewise, you will have to identify your services of the application, respectively.

## 2. Pull or build images

For some of your services, you may not need to build from a custom Dockerfile , and a public image on [DockerHub](https://hub.docker.com/" \t "_blank) will suffice.

For example, if you have a MySQL database in your application, you can pull MySQL image from the hub instead of building it. For others, you will have to create a Dockerfile and build them.

## 3. Configure environment variables, declare dependencies

Most applications use environment variables for initialization and startup. And also, after we divide the application into services, they have dependencies on each other. So we need to identify those things before we declare the compose file.

## 4. Configure networking

Docker containers communicate with each other through their internal network that is created by compose (eg service\_name:port). If you want to connect from your host machine, you will have to expose the service to a host port.

## 5. Set up volumes

In most cases, we would not want our database contents to be lost each time the database service is brought down. A simple way to persist our DB data is to mount a volume.

## 6. Build & Run

Now, you are set to go and create the compose file and build the images for your services and generate containers from those images.

**Create a Custom Image**

We can create a docker image easily with few commands. There are two ways of creating a docker image depending upon the purpose for which you want to create the image. The first method is using commit command and another method is by using Docker file concept.

Commands Required

The first command is pull command. This command will download/pull the complete operating system within seconds depending upon your internet connectivity. The syntax is like, docker pull image\_name for eg we are pulling alex43/ubuntu-with-git:v1.0 which is my own customized image.

docker pull alex43/ubuntu-with-git:v1.0

The second command is run command which we will use to run the pulled image. This command will launch our image and we will get an interactive shell/terminal of that image. The syntax is like, -it for an interactive terminal, –name to give reference name for our image launched and then our image\_name.

docker run -it --name myos alex43/ubuntu-with-git:v1.0

The third command and the most important command for creating our own image is commit command. By using this command we can simply create our own image with the packages which we want from the existing image. The syntax is like, docker commit Name of Running Image your own name:tag.

docker commit myos ubuntu-basicbundle:v1.0

The fourth command is tag command. By using this command we need to rename our image with syntax username/image-name:tag. Before executing this command you need to create an account on the Docker hub and you have to give the same username which you have given in the docker hub profile.

docker tag alex43/ubuntu-with-git:v1.0 alex43/ubuntu-basicbundle:v1.0

The fifth command is login command. By using this command we will logged in to the docker hub account through our terminal and it is required to upload our docker image to the docker hub profile.

docker login --username alex43 --password your\_passwd

The sixth command is push command. By using this command we can upload our own created docker image to the docker hub profile and can use it anywhere from our local system to cloud by pulling it.

docker push alex43/ubuntu-basicbundle:v1.0

So these were the few commands with the concept which we will be using in this tutorial and I will be uploading one fresh image so that you guys can understand it in a better way.

My Own Image

Now we will create our own image from existing alex43/ubuntu-with-git:v1.0 image and we will customize it with our needs and we will upload it.

Step 1: The very first step is to pull the image as I have shown in the upper commands. Use the command and pull the image into your system.

docker pull alex43/ubuntu-with-git:v1.0

Step 2: Launch that image so that we can customize it as per our needs. In the below picture you can see that at 1st we were not having vim editor in our image so it was giving error of command not found. Then I installed vim in our system by using apt-get install vim command.

In the below image you can see that after installing vim editor when I opened abhi.txt with vim editor it didn’t gave any error and I was able to write content in that. You can also see the content using cat command which we already have in alex43/ubuntu-with-git:v1.0 image.

So it was only the step that you need to understand and apply your own concepts. You can create lots of more useful docker images to solve some industry use cases like launching a WordPress blogging site using a docker image, creating your own MySQL database image, etc. It is the basic concept which you need to understand and apply as per your needs.

Step 3: Come out of the image by pressing ctrl+p+q and commit the modified image as I have explained in the above command.

Step 4: Create an account at docker hub and change the name of the image with proper syntax as explained above.

Step 5: Log in into docker hub profile from terminal using docker login command as explained above.

Step 6: The final task is to upload the image using push command to docker hub profile as explained above.

For step-3, step-5, and step-6 you can see the image below. Pushing your image might take some time depending upon the size of the image and the internet speed of your system. In the image, you can see I have used the commands which I explained above and it successfully uploaded my image at my docker hub profile.

So in this way we can create our own docker image and push it to docker hub just by using few commands.

**Docker Networking**

Networking is about communication among processes, and Docker’s networking is no different. Docker networking is primarily used to establish communication between Docker containers and the outside world via the host machine where the Docker daemon is running.

Docker supports different types of networks, each fit for certain use cases. We’ll be exploring the network drivers supported by Docker in general, along with some coding examples.

Docker networking differs from virtual machine (VM) or physical machine networking in a few ways:

1. Virtual machines are more flexible in some ways as they can support configurations like [NAT and host networking](https://superuser.com/questions/227505/what-is-the-difference-between-nat-bridged-host-only-networking). Docker typically uses a bridge network, and while it can support host networking, that option is [only available on Linux](https://docs.docker.com/network/host/).
2. When using Docker containers, network isolation is achieved using a network namespace, not an entirely separate networking stack.
3. You can run hundreds of containers on a single-node Docker host, so it’s required that the host can support networking at this scale. VMs usually don’t run into these network limits as they typically run fewer processes per VM.

**Accessing the Docker**

To check the container status and run Workload Automation commands, you need to access the containers as described below:

1. Obtain the container ID by running the following command: **docker ps**

An output similar to the following one is returned:

CONTAINER ID IMAGE NAMES ........ .......

b02459af2b9c ...... wa-console ........ .......

1. Access the Docker container by running the following command: **docker exec -it <container\_id> /bin/bash**

Where

***container\_id***

Is the ID of the container obtained with the command explained in the first step, for example **b02459af2b9c**.

**Linking Containers**

Docker container linking allows multiple containers to be linked to each other. It allows the recipient container to get connection information relating to the source container. You should use Docker container linking when you are using default bridge networks and when you want to share environmental variables.

How do I link two Docker containers together?

To allow two Docker containers on the same host to communicate with each other by name:

Create a user-defined bridge network: Create your own custom bridge network first using docker network create . ...

Start a container and connect it to the bridge: Start your container as normal.

**Exposing Container Ports**

Need of exposing ports.

When a container is created using docker create or docker run, by default it does not publish any of its ports to the outside world.

In order to make a port available to services outside of Docker, or to Docker containers which are not connected to the container’s network, we can use the -P or -p flag. This creates a firewall rule which maps a container port to a port on the Docker host to the outside world.

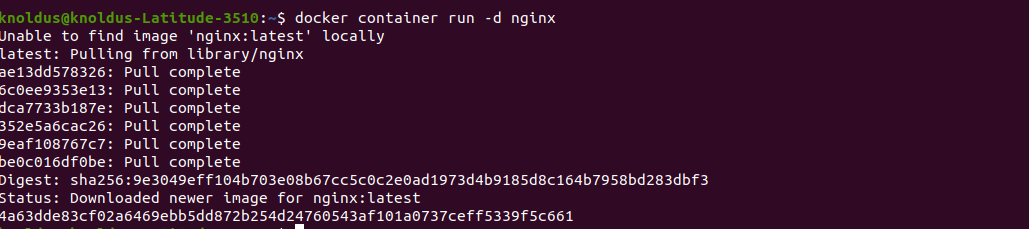
-p or –publish list : Publishes a container’s port(s) to the host.

-P or –publish-all : Publishes all exposed ports to random ports.

#### Let’s see how we can achieve this,

##### **Create a nginx container without any port mapping,**

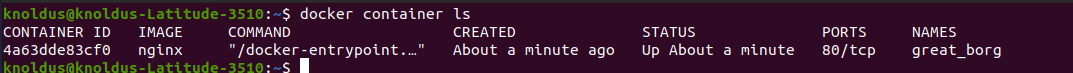
docker container run -d nginx



This will pull the nginx image from the docker hub and create a container for us.

##### **Check the port for nginx container**

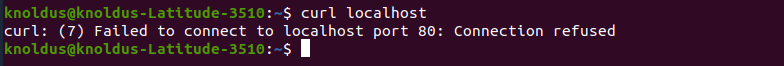
docker container ls



This shows the nginx container. We have tcp port for port 80. However it is yet not mapped.

##### **Check if we get any response from localhost**

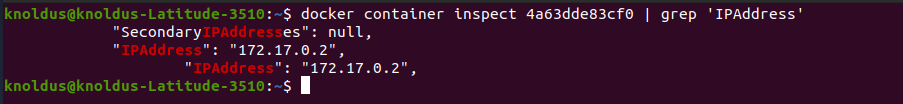
curl localhost



Connection refused because we don’t have port mapping set for port 80.

##### **Get the privateIP of the container**

docker inspect container



Execute

curl This will give content of nginx home page.

##### **Expose port 3000 on container**

Once we expose the port, it means now this port is available to be mapped. We can re-map them using -p or -P flag.

docker container run -d --expose 3000 nginx



##### **Verify if port 3000 is open**

docker container ls

