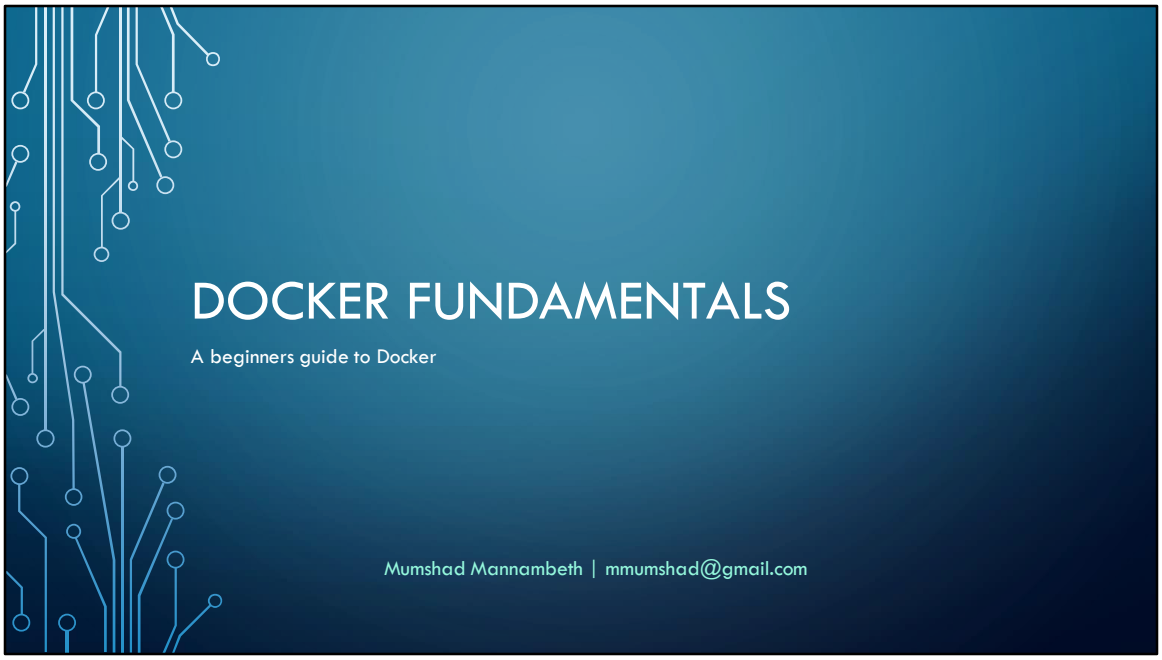
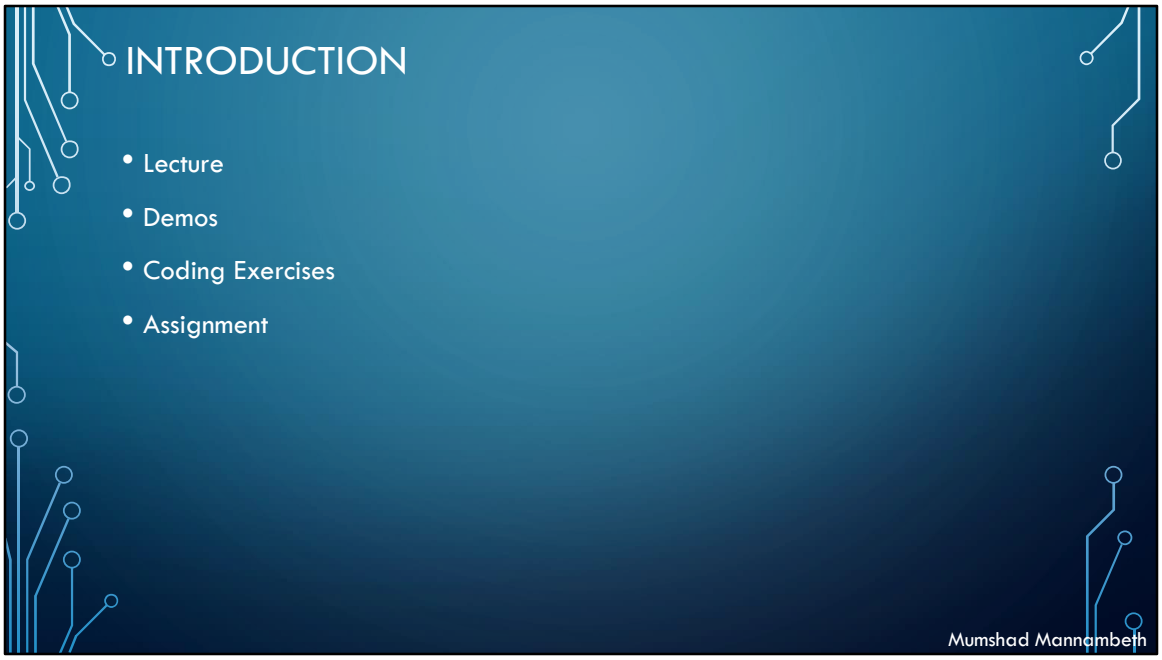

docker
For the Absolute Beginner

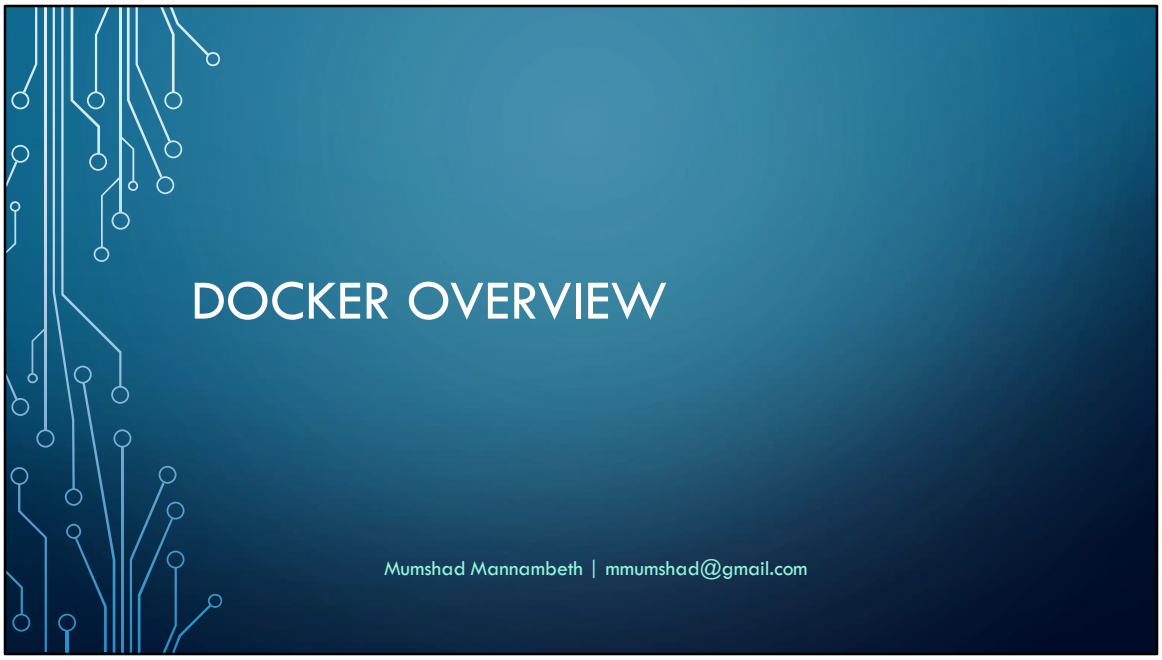


Hello and welcome to this course on Docker Fundametals. My name is Mumshad Mannambeth and I work as a Solutions Architect and I design solutions and cloud automation. This is a hands-on beginner's guide to Docker. And we learn Docker through some fun and interactive coding exercises.

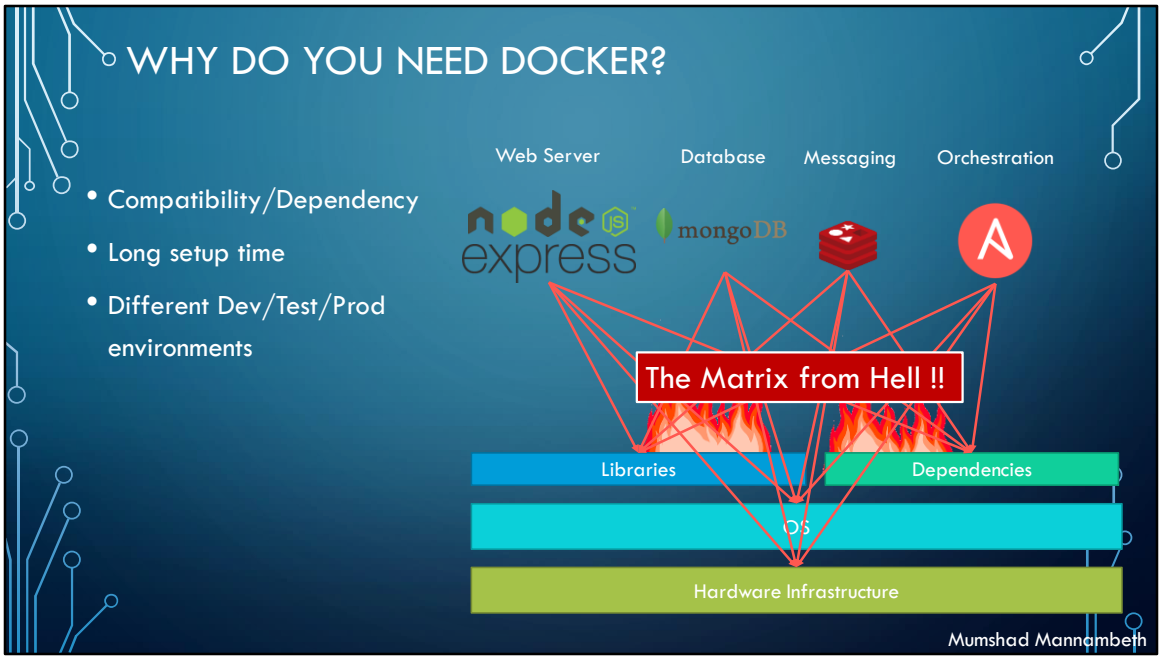


So how exactly does this course work? This course contains lectures on various topics followed by some demos showing you how to setup and get started with docker. We then go through some coding exercises were you will practice writing Docker commands, build your own Docker images using Dockerfiles and setup your own stack using Docker compose. You will be developing Docker Images for different use cases, which will give you a pretty good idea on how to start creating your own images and how to share them in the community. Finally we will take a practice test to test your knowledge.

In this course we are going to get introduced to Docker basics. What Docker is, how to run Docker Containers, How Docker handles networking, how to create a Docker image and finally we will look at what Docker Compose and Docker swarm are. This course is intended to give an absolute beginner some idea on Docker and enough information on getting started, playing around and exploring Docker. So let's get started.



Hello and welcome to this lecture on Docker Overview. My name is Mumshad Mannambeth and we are learning Docker Fundamentals. In this lecture we are going to look at a high level overview on why you need Docker and what it can do for you.



Let me start by sharing how I got introduced to Docker. In one of my previous projects, I had this requirement to setup an end-to-end stack including various different technologies like a Web Server using NodeJS and a database such as MongoDB/CouchDB, messaging system like Redis and an orchestration tool like Ansible. We had a lot of issues developing this application with all these different components. First, their compatibility with the underlying OS. We had to ensure that all these different services were compatible with the version of the OS we were planning to use. There have been times when certain version of these services were not compatible with the OS, and we have had to go back and look for another OS that was compatible with all of these different services.

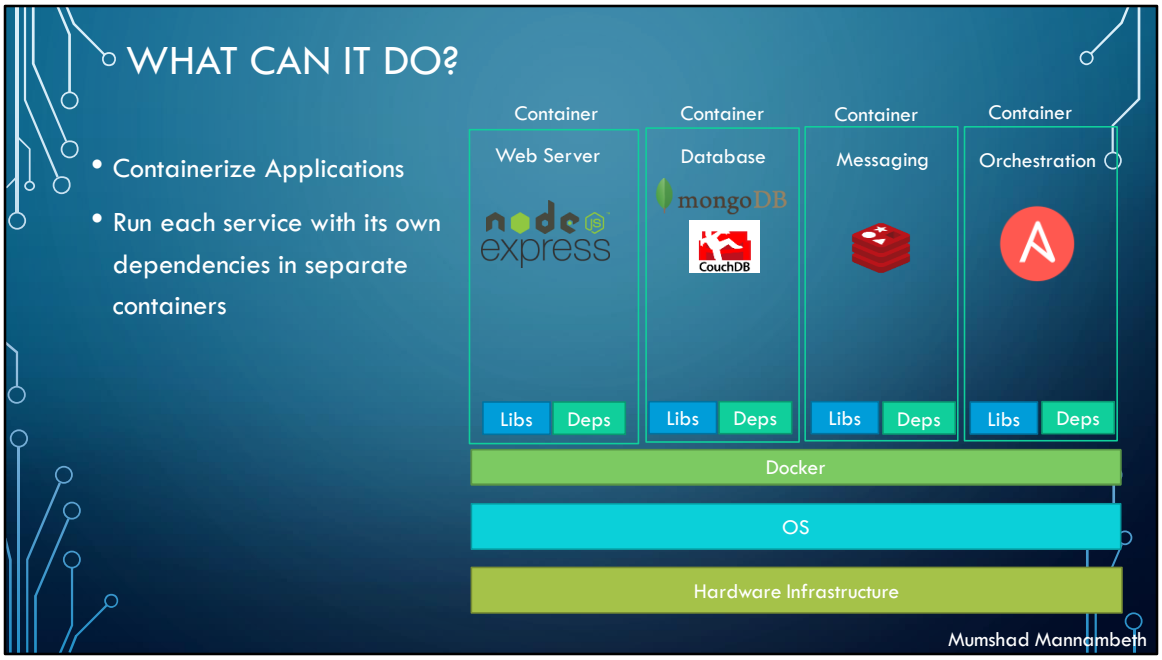
Secondly, we had to check the compatibility between these services and the libraries and dependencies on the OS. We have had issues were one service requires one version of a dependent library whereas another service required another version.

The architecture of our application changed over time, we have had to upgrade to newer versions of these components, or change the database etc and everytime something changed we had to go through the same process of checking compatibility between these various components and the underlying infrastructure. This

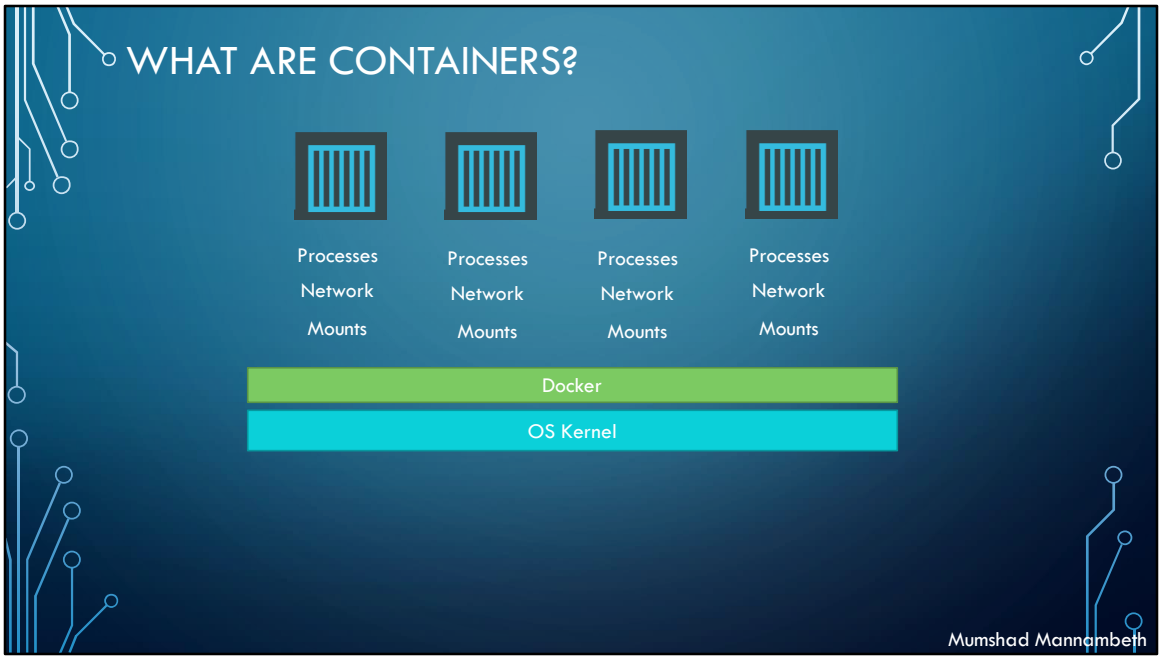
compatibility matrix issue is usually referred to as the matrix from hell.

Next, everytime we had a new developer on board, we found it really difficult to setup a new environment. The new developers had to follow a large set of instructions and run 100s of commands to finally setup their environments. They had to make sure they were using the right Operating System, the right versions of each of these components and each developer had to set all that up by himself each time.

We also had different development test and production environments. One developer may be comfortable using one OS, and the others may be using another one and so we couldn't gurantee the application that we were building would run the same way in different environments. And So all of this made our life in developing, building and shipping the application really difficult.

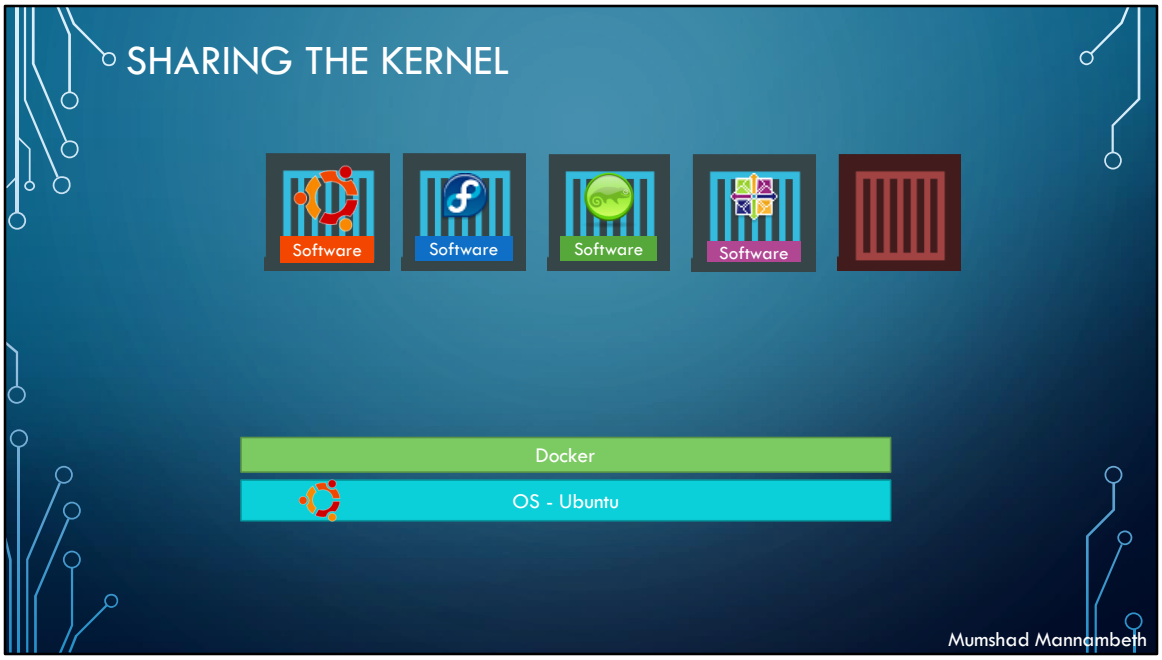


So I needed something that could help us with the compatibility issue. And something that will allow us to modify or change these components without affecting the other components and even modify the underlying operating systems as required. And that search landed me on Docker. With Docker I was able to run each component in a separate container – with its own libraries and its own dependencies. All on the same VM and the OS, but within separate environments or containers. We just had to build the docker configuration once, and all our developers could now get started with a simple “docker run” command. Irrespective of what underlying OS they run, all they needed to do was to make sure they had Docker installed on their systems.



So what are containers? Containers are completely isolated environments, as in they can have their own processes or services, their own network interfaces, their own mounts, just like Virtual machines, except that they all share the same OS kernel. We will look at what that means in a bit. But its also important to note that containers are not new with Docker. Containers have existed for about 10 years now and some of the different types of containers are LXC, LXD , LXCFS etc. Docker utilizes LXC containers. Setting up these container environments is hard as they are very low level and that is were Docker offers a high-level tool with several powerful functionalities making it really easy for end users like us.

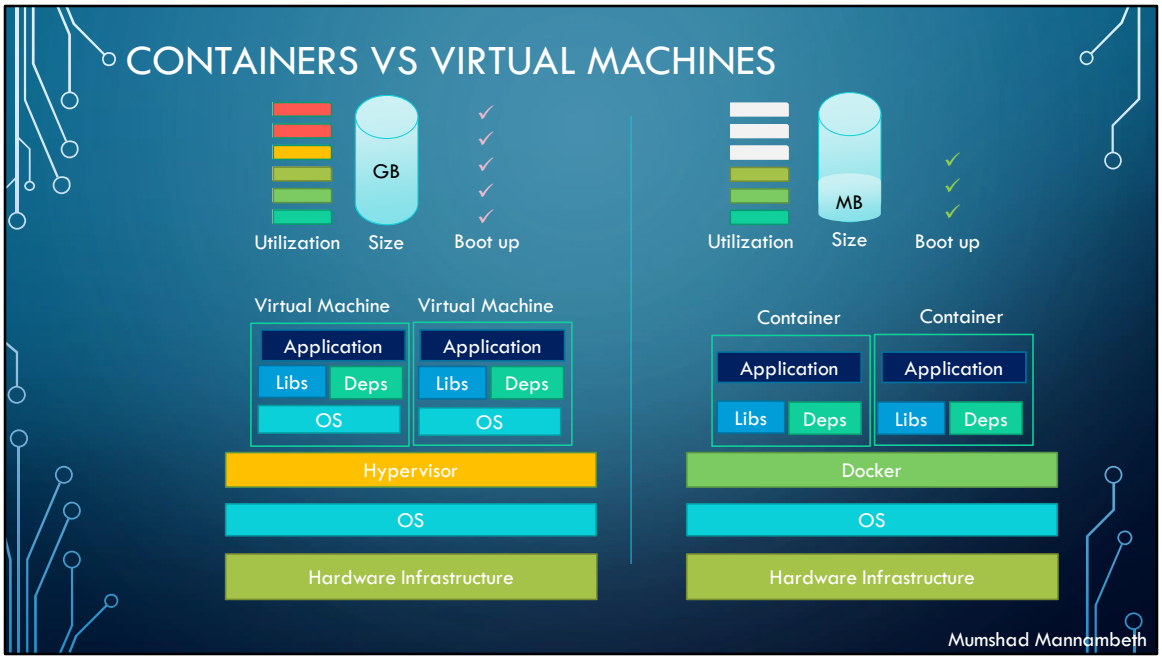
To understand how Docker works let us revisit some basics concepts of Operating Systems first. If you look at operating systems like Ubuntu, Fedora, Suse or Centos – they all consist of two things. An OS Kernel and a set of software. The OS Kernel is responsible for interacting with the underlying hardware. While the OS kernel remains the same– which is Linux in this case, it's the software above it that make these Operating Systems different. This software may consist of a different User Interface, drivers, compilers, File managers, developer tools etc. SO you have a common Linux Kernel shared across all Oses and some custom softwares that differentiate Operating systems from each other.



We said earlier that Docker containers share the underlying kernel. What does that actually mean – sharing the kernel? Let's say we have a system with an Ubuntu OS with Docker installed on it. Docker can run any flavor of OS on top of it as long as they are all based on the same kernel – in this case Linux. If the underlying OS is Ubuntu, docker can run a container based on another distribution like debian, fedora, suse or centos. Each docker container only has the additional software, that we just talked about in the previous slide, that makes these operating systems different and docker utilizes the underlying kernel of the Docker host which works with all Oses above.

So what is an OS that do not share the same kernel as these? Windows ! And so you wont be able to run a windows based container on a Docker host with Linux OS on it. For that you would require docker on a windows server.

You might ask isn't that a disadvantage then? Not being able to run another kernel on the OS? The answer is No! Because unlike hypervisors, Docker is not meant to virtualize and run different Operating systems and kernels on the same hardware. The main purpose of Docker is to containerize applications and to ship them and run them.



So that brings us to the differences between virtual machines and containers. Something that we tend to do, especially those from a Virtualization.

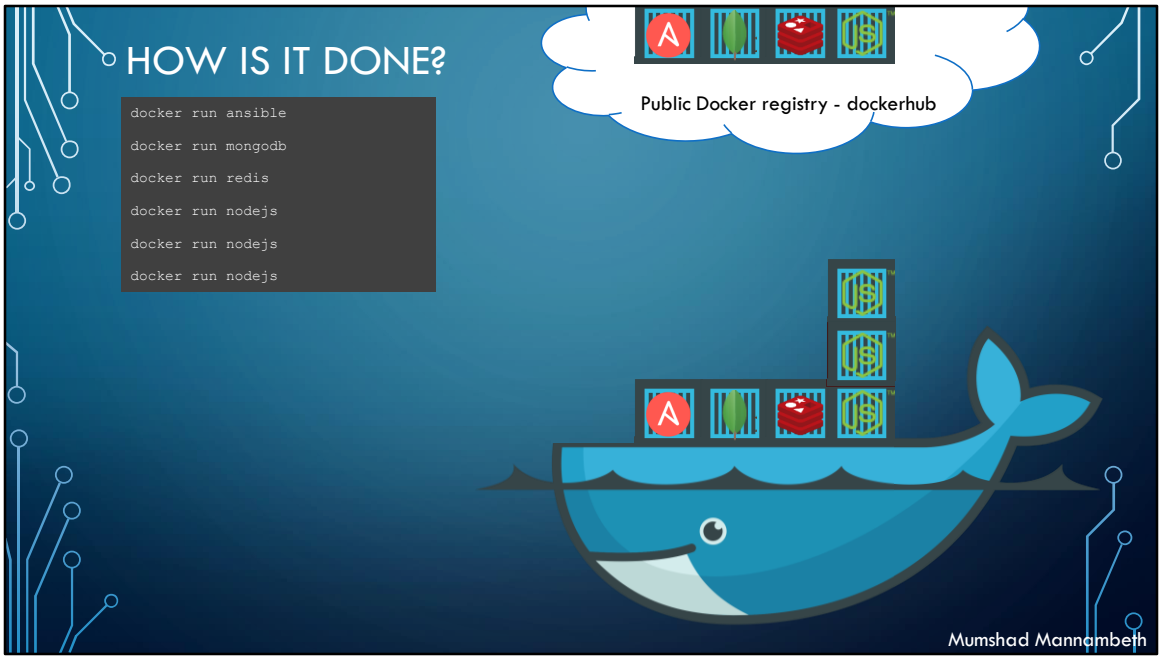
As you can see on the right, in case of Docker, we have the underlying hardware infrastructure, then the OS, and Docker installed on the OS. Docker then manages the containers that run with libraries and dependencies alone. In case of a Virtual Machine, we have the OS on the underlying hardware, then the Hypervisor like a ESX or virtualization of some kind and then the virtual machines. As you can see each virtual machine has its own OS inside it, then the dependencies and then the application.

This overhead causes higher utilization of underlying resources as there are multiple virtual operating systems and kernel running. The virtual machines also consume higher disk space as each VM is heavy and is usually in Giga Bytes in size, wereas docker containers are lightweight and are usually in Mega Bytes in size.

This allows docker containers to boot up faster, usually in a matter of seconds whereas VMs we know takes minutes to boot up as it needs to bootup the entire OS.

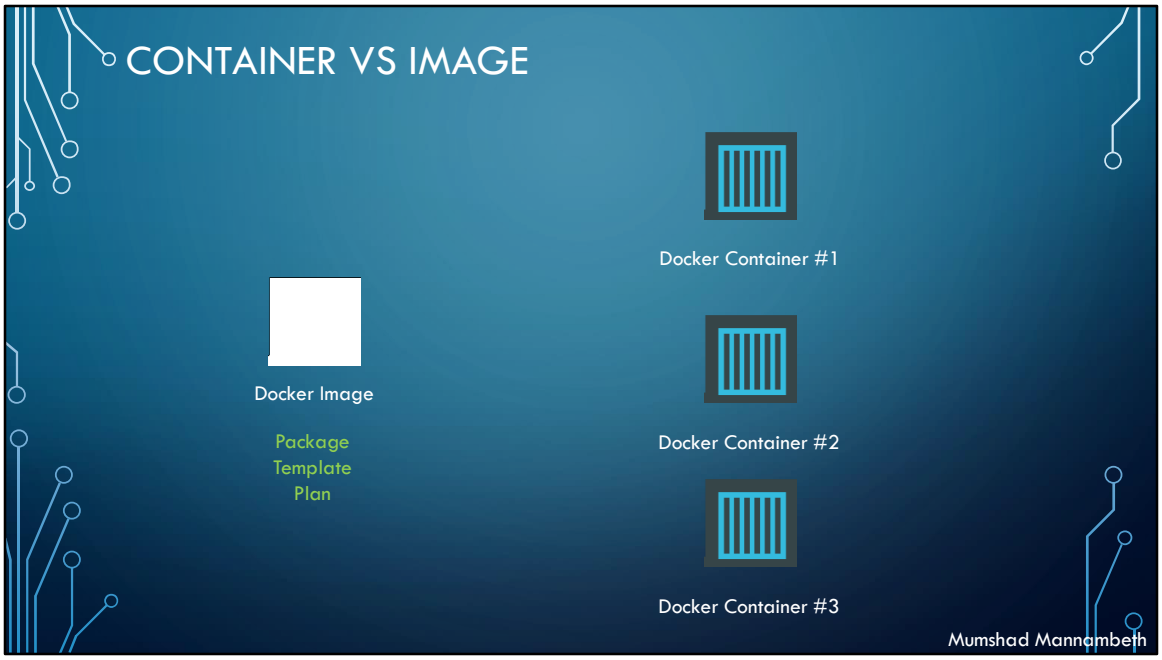
It is also important to note that, Docker has less isolation as more resources are shared between containers like the kernel etc. Whereas VMs have complete isolation from each other. Since VMs don't rely on the underlying OS or kernel, you can run different types of OS such as linux based or windows based on the same hypervisor.

So these are some differences between the two.



SO how is it done? There are a lot of containerized versions of applications readily available as of today. So most organizations have their products containerized and available in a public docker registry called dockerhub/or docker store already. <show dockerhub>. For example you can find images of most common operating systems, databases and other services and tools. Once you identify the images you need and you install Docker on your host..

bringing up an application stack, is as easy as running a docker run command with the name of the image. In this case running a docker run ansible command will run an instance of ansible on the docker host. Similarly run an instance of mongodb, redis and nodejs using the docker run command. And then when you run nodejs just point to the location of the code repository on the host. If we need to run multiple instances of the web service, simply add as many instances as you need, and configure a load balancer of some kind in the front. In case one of the instances was to fail, simply destroy that instance and launch a new instance. There are other solutions available for handling such cases, that we will look at later during this course.

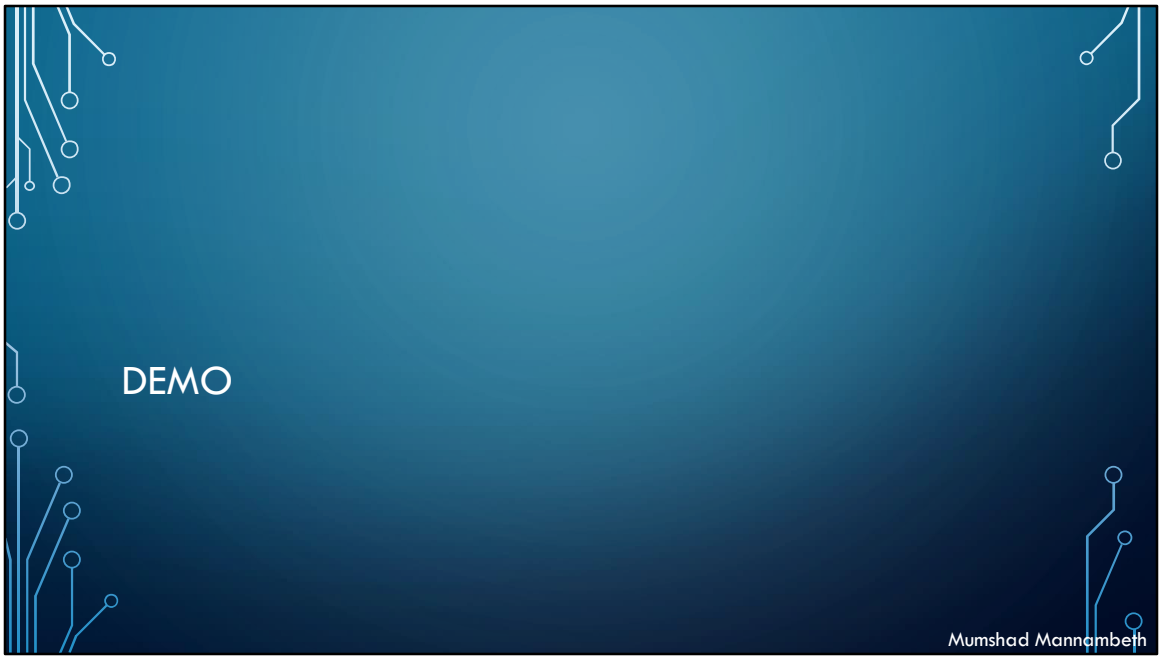


We have been talking about images and containers. Let's understand the difference between the two.

An image is a package or a template, just like a VM template that you might have worked with in the virtualization world. It is used to create one or more containers.

Containers are running instances off images that are isolated and have their own environments and set of processes

<show dockerhub> As we have seen before a lot of products have been dockerized already. In case you cannot find what you are looking for you could create an image yourself and push it to the Docker hub repository making it available for public.



Demo of setting up Docker

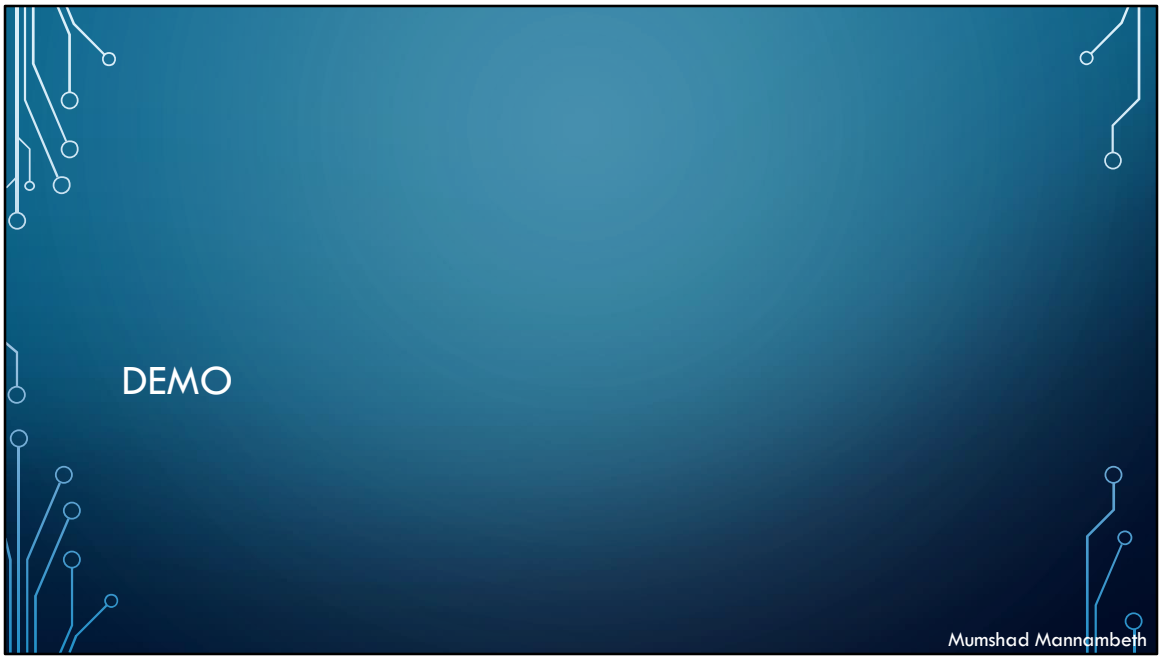
QUIZ

- Differences between Docker and VMs

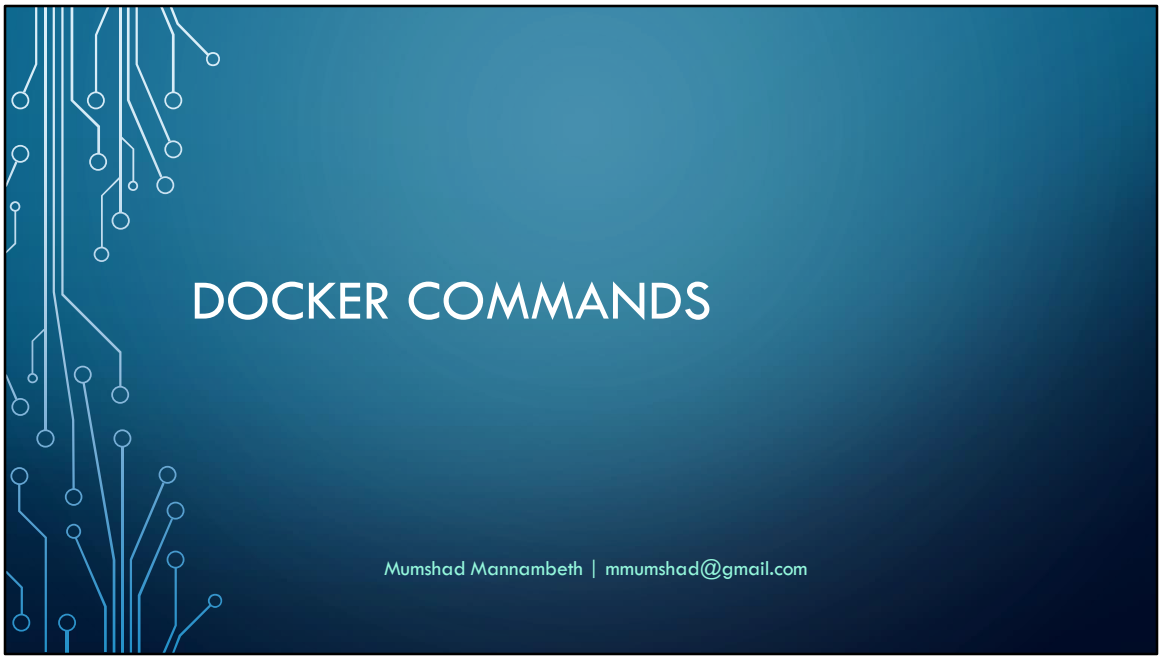
Mumshad Mannambeth

CODING EXERCISES

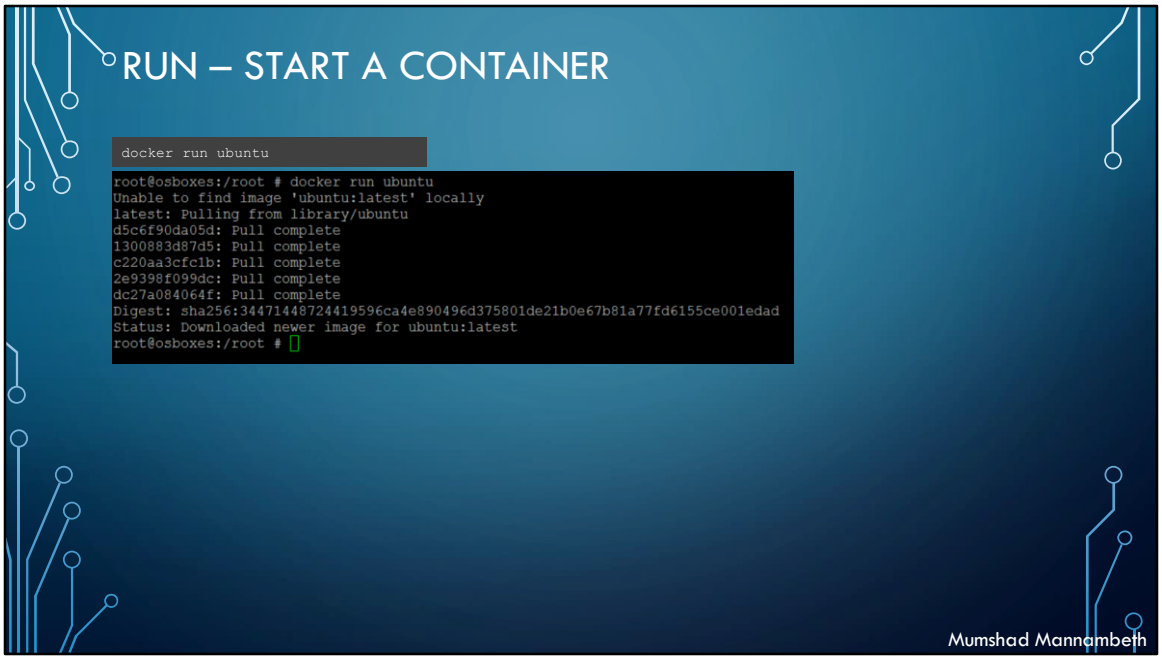
Mumshad Mannambeth



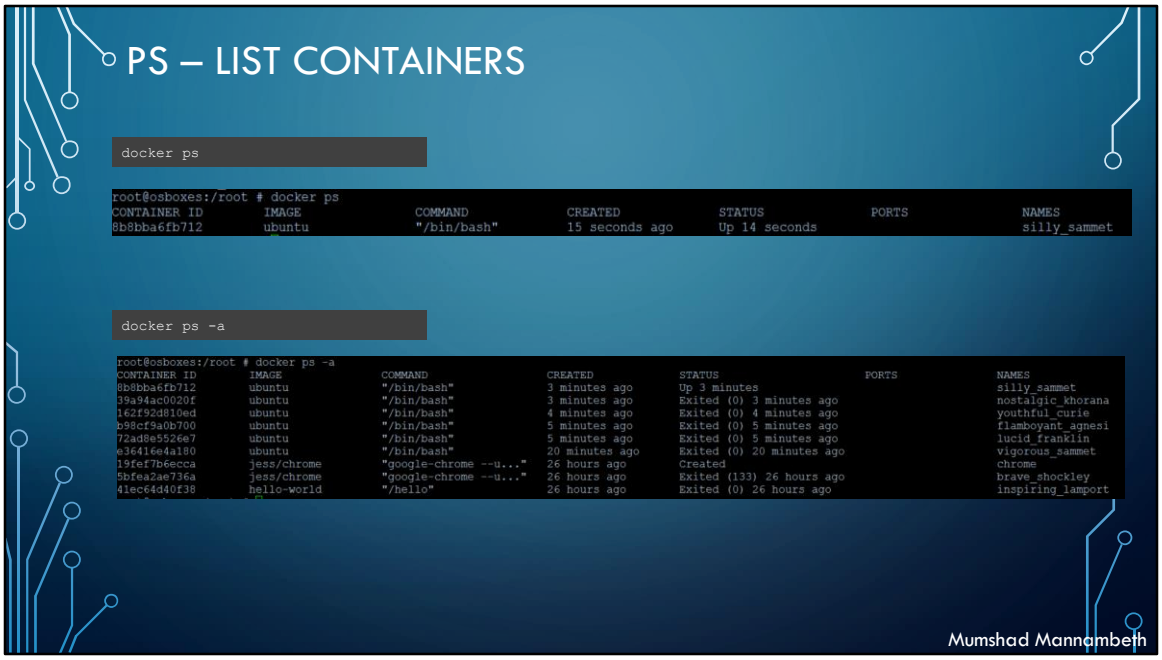
Demo of Coding Exercises



Hello and welcome to this lecture on Docker Commands. My name is Mumshad Mannambeth and we are learning Docker Fundamentals.

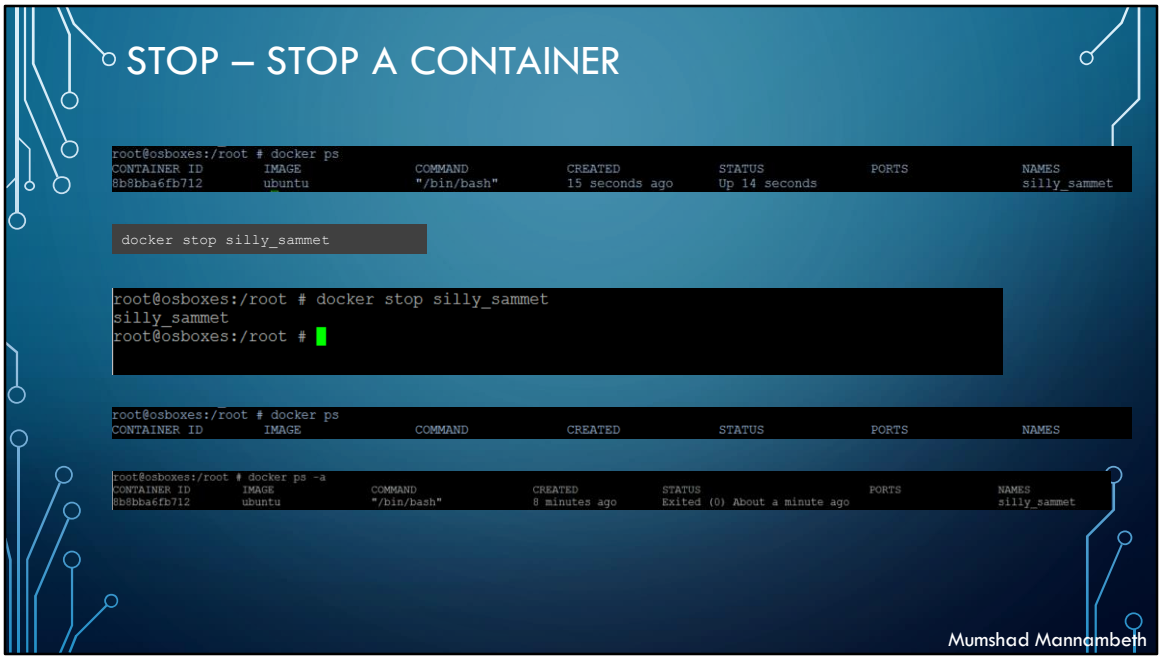


Let's start by looking at the Docker Run command. The Docker run command is used to run a container from an image. Running the docker run Ubuntu command will run an instance of the Ubuntu image on the Docker host if it already exists. If it doesn't exist it will go out to docker hub and pull the image down, but this is only done the first time. For subsequent executions the same image will be used.



The docker ps command lists all running containers and some basic information regarding them. Such as the container ID, the name of the image we used to run the containers, the current status and a random funny name assigned to it by Docker which in this case is silly_sammet.

To see all containers running or not, use the –a option. This outputs all running as well as previously exited containers.



To stop a running container, use the stop command. But you must provide either the container ID or the container name in the stop command. If you are not sure of the name run the docker ps command to get it. On success you will see the name printed out and running docker ps again will show no running containers. Running docker ps –a however shows the container silly_sammet and that it exited about a minute ago. But what if we don't want this container lying around consuming space? What if we want to get rid of it for good?

RM – REMOVE A CONTAINER

```
docker rm silly_sammet
```

```
root@osboxes:/root # docker rm silly_sammet
silly_sammet
root@osboxes:/root # docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
```

Mumshad Mannambeth

Let us try to clean up a bit. Use docker rm command to remove a stopped or exited container permanently. If it prints the name back we are good. Run the docker ps command again to verify nothing is listed. Good, but what about the Ubuntu image that was downloaded at first. We are not using that anymore, how do we get rid of it?

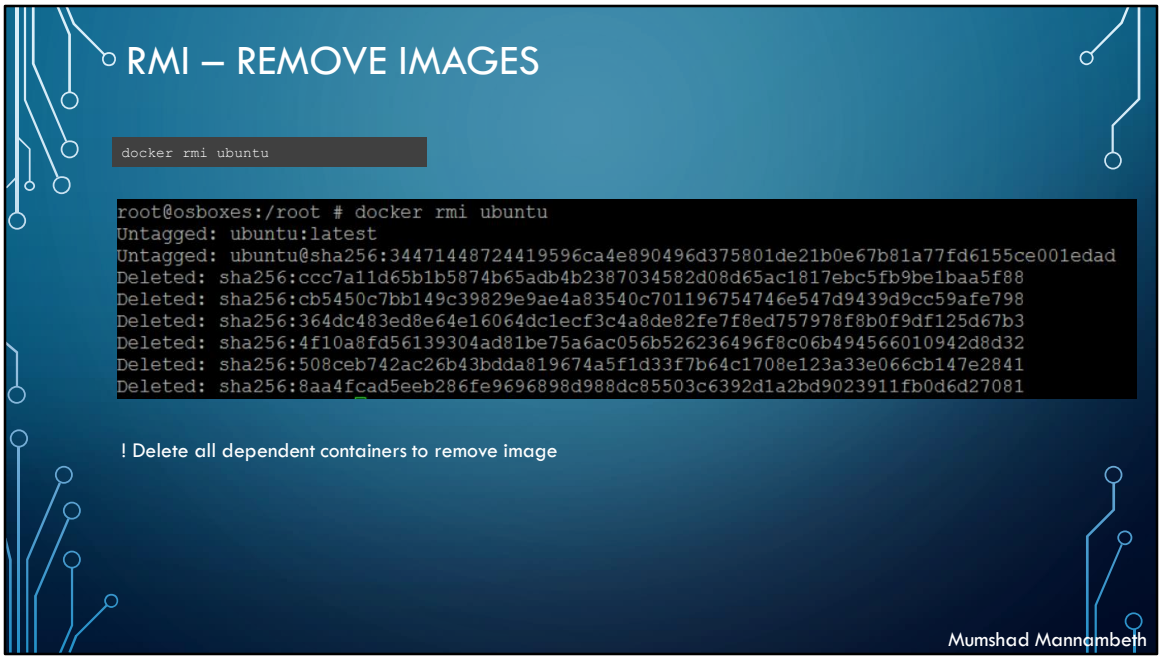
IMAGES – LIST IMAGES

```
docker images
```

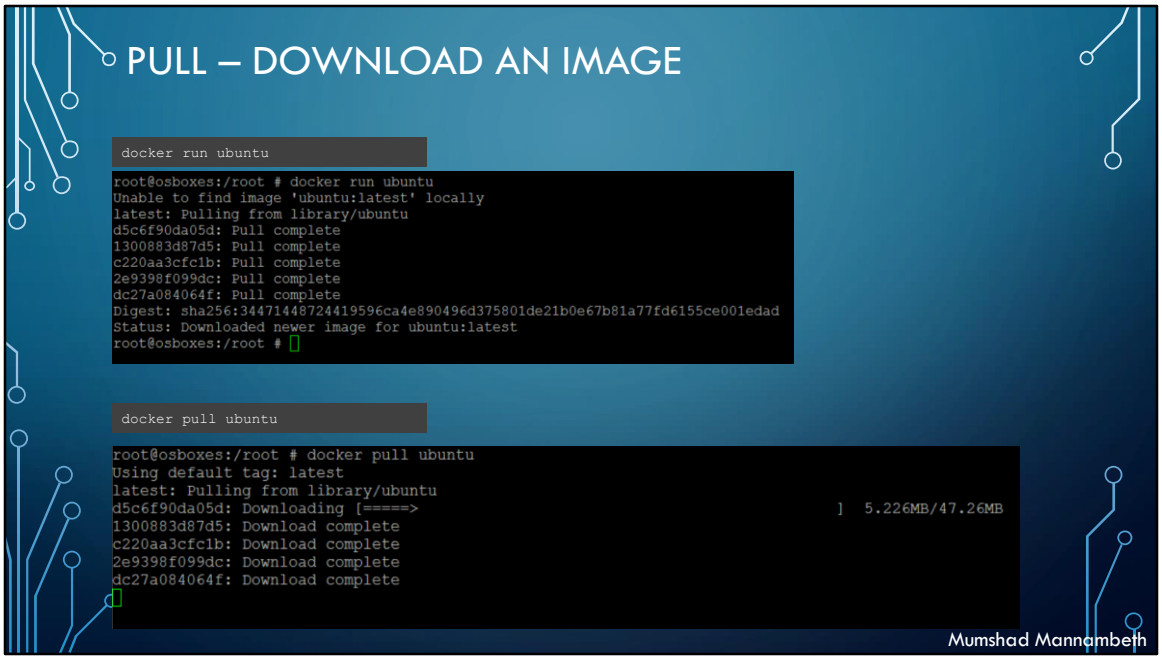
```
root@osboxes:/root # docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              ccc7a11d65b1        10 days ago         120MB
```

Mumshad Mannambeth

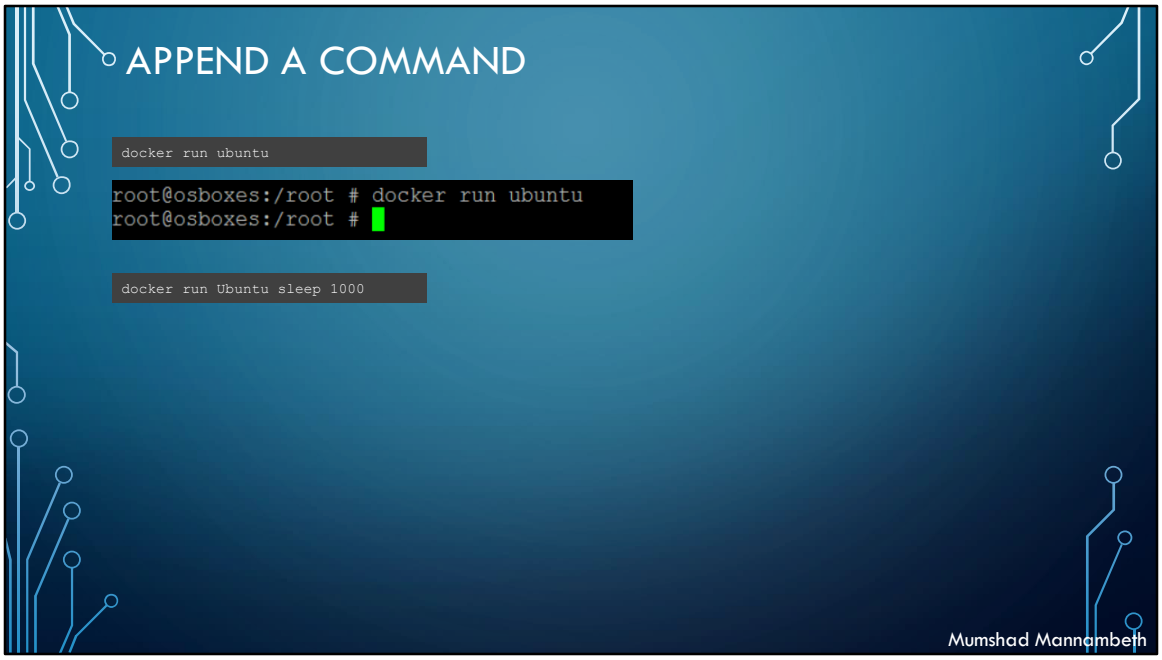
Run the images command to see a list of available images.



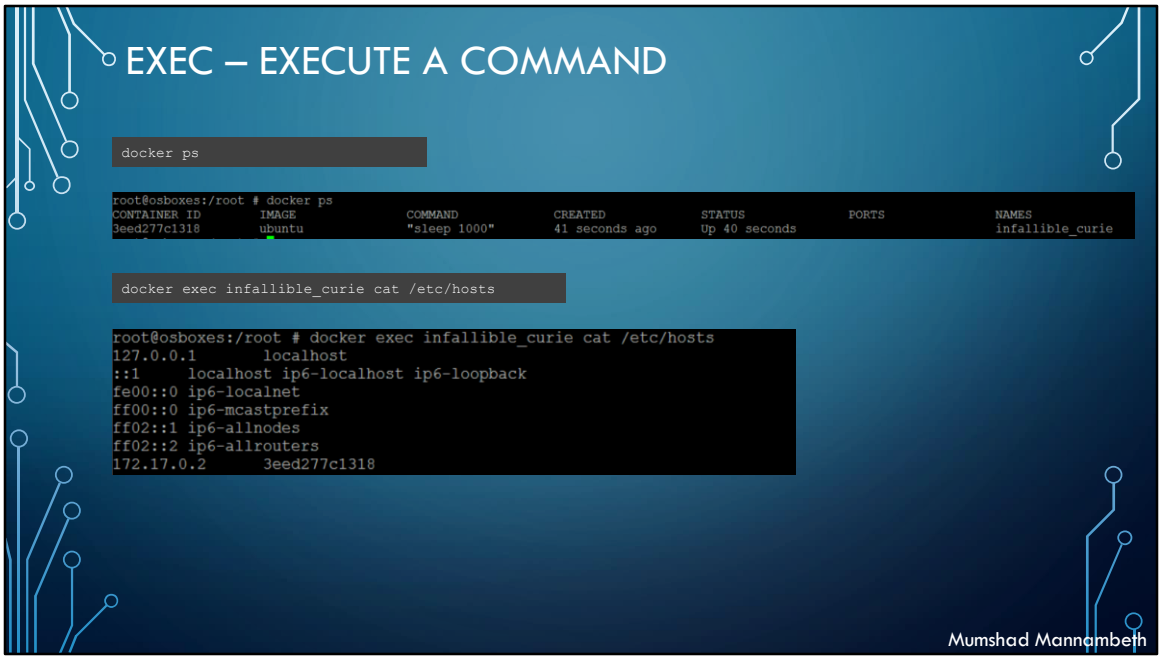
Run the docker rmi command to remove images. But you must ensure no containers are running off of it. You must stop and delete all dependent containers to be able to delete the image.



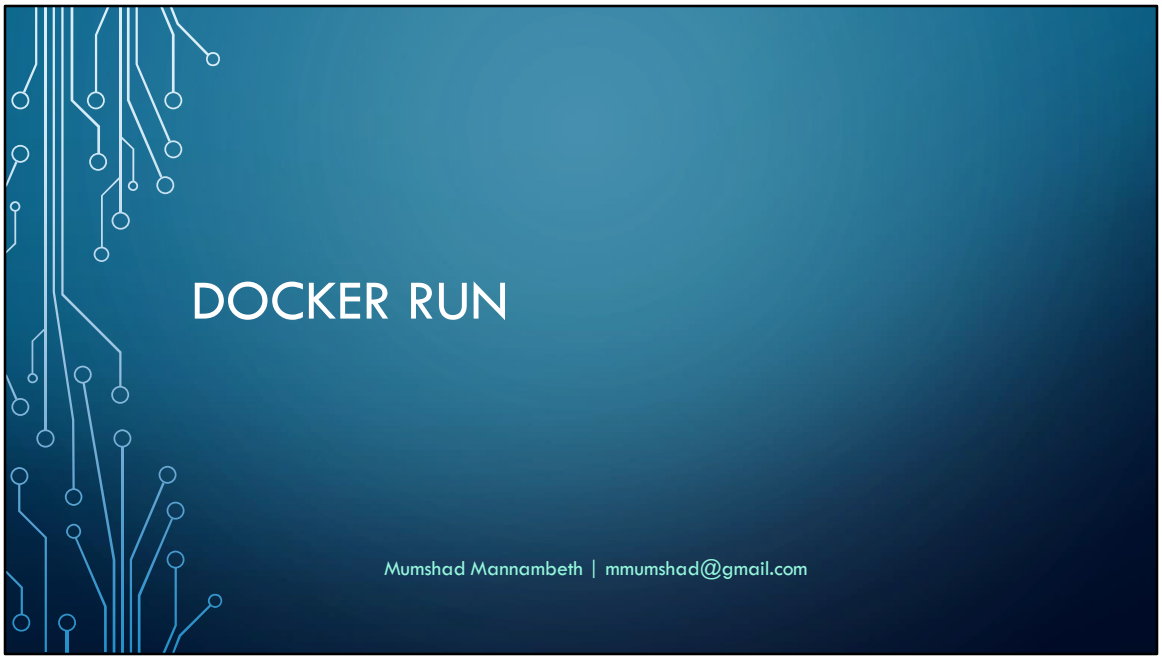
When we ran the docker run command earlier, it downloaded the Ubuntu image as it couldn't find one locally. What if we simply want to download the image and keep so when we use run command we don't have to wait for it to download? Use the docker pull Ubuntu command to only pull the image.



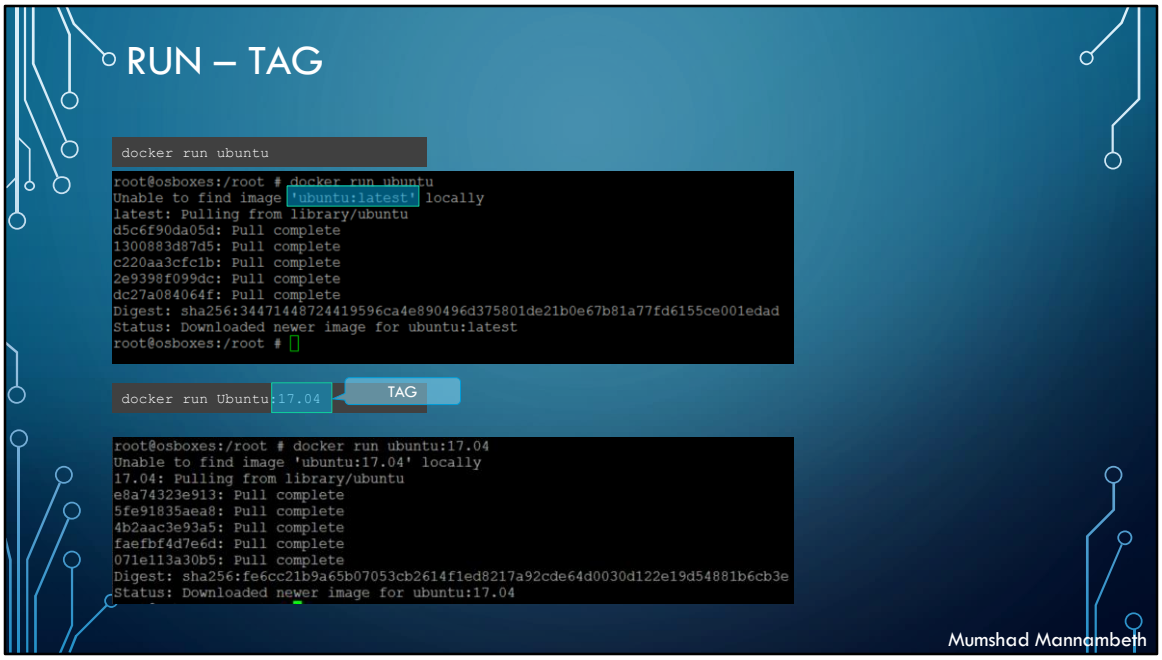
When you run the docker run command with Ubuntu, you will notice that the container doesn't actually stay alive. This is because there is nothing for the container to do and it would exit immediately after starting. Docker containers are meant to run services or applications. If there isn't anything running docker stops the container immediately. If the image isn't running any services as is the case with Ubuntu, you could execute something with the docker run command, for example a sleep command with a duration of 1000 seconds. When the container starts it runs the sleep command and goes into sleep for 1000 seconds.



What we just saw was executing a command when we run the container. But what if we would like to execute a command on a running container. For example, when I run the docker ps command I can see that there is a running container. Let's say I would like to see the contents of a file inside the running container. I could use the docker exec command to execute a command on my docker container.

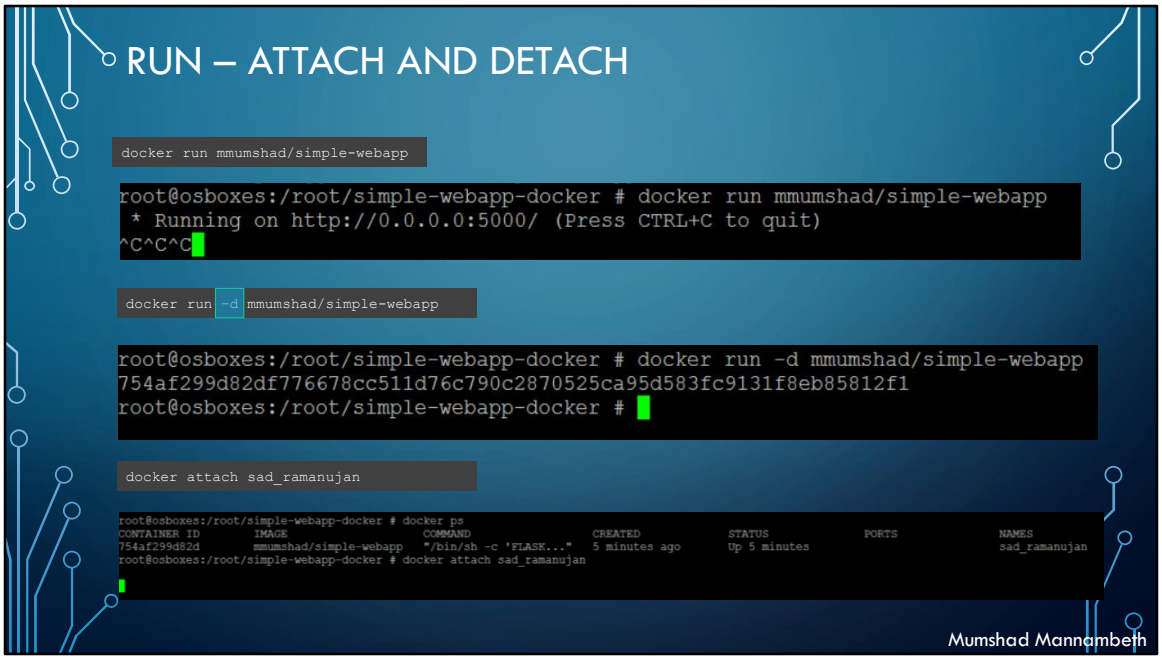


Hello and welcome to this lecture on Docker RUN Command. My name is Mumshad Mannambeth and we are learning Docker Fundamentals.



Let's take a deeper look at Docker RUN command. We learned that we could use the docker run Ubuntu command to run a container. In this case the latest version of Ubuntu. But what if we want to run another version of Ubuntu? Like for example the version 17.04. Then you specify the version separated by a colon. This is called a tag.

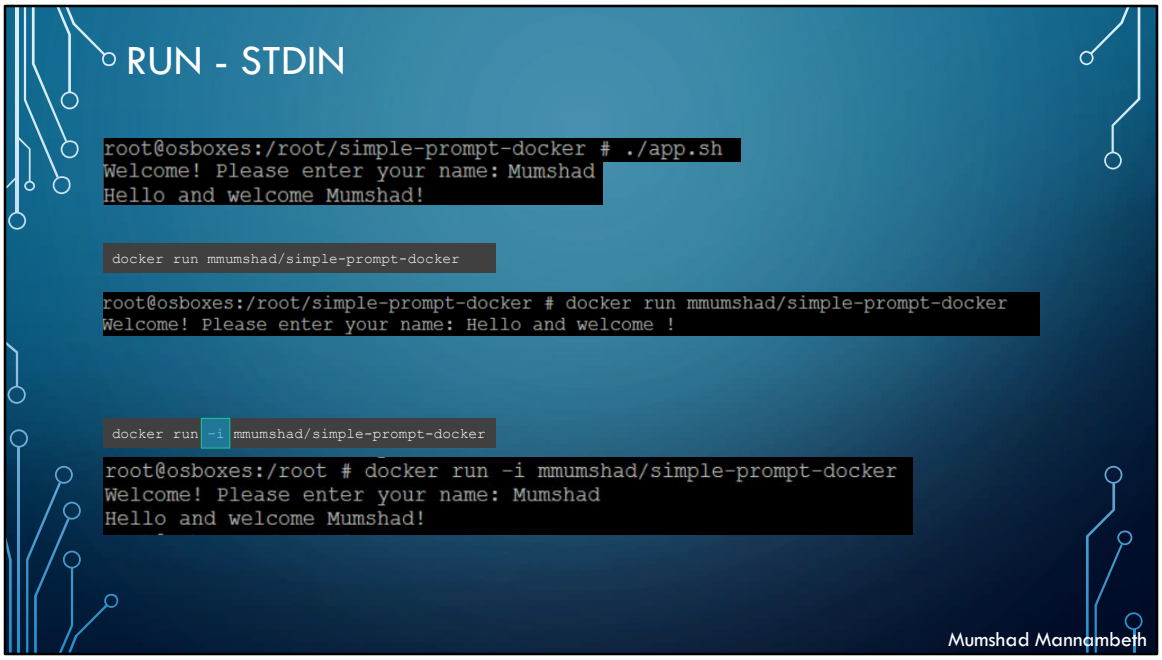
Also, notice that if you don't specify any tag as in the previous command, docker will consider the default tag which is "latest"



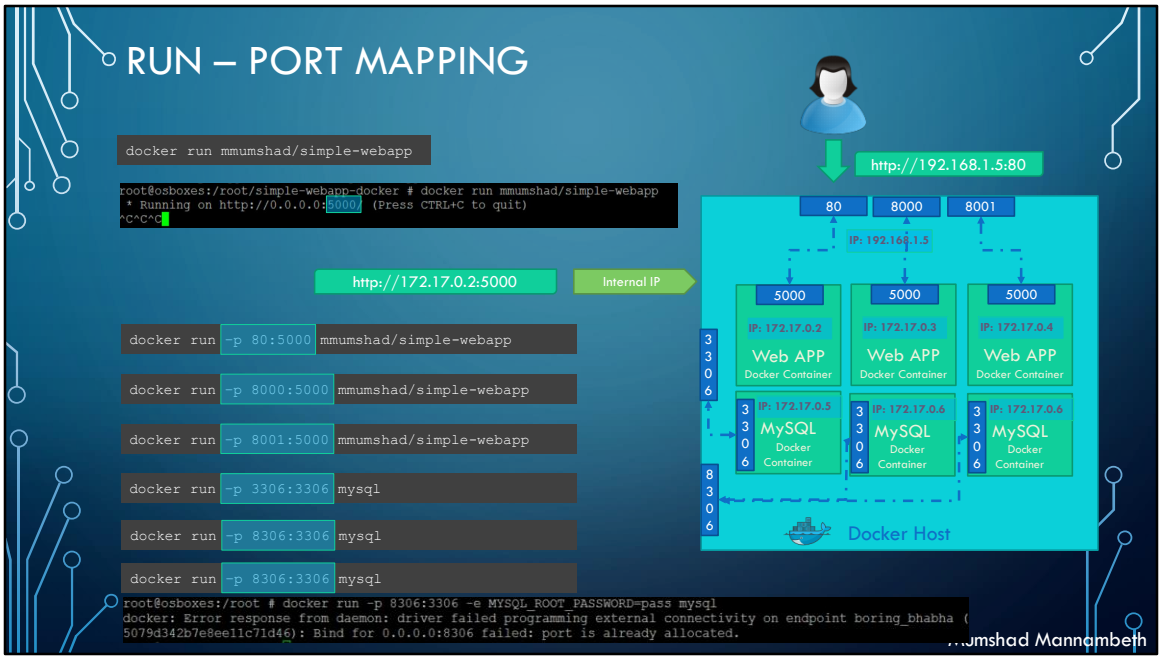
I am now going to run a Docker image I developed for a simple web application. The repository name is mmumshad/simple-webapp. It runs a simple web server that listens on port 5000. When you run a docker run command like this, it runs in a background or in an attached mode, meaning you will be attached to the console of the docker container and you will see the output of the web service on your screen. You won't be able to do anything else on this console other than view the output until this docker container stops. It won't respond to yours inputs and you won't be able to do anything else on this terminal. If you need to stop the docker container open another terminal and stop the container using the docker stop command.

To get around this, run the docker container in the detached mode by providing the –d option. This will run the docker container in the detached mode and you will be back to your prompt.

If you would like to attach back to the running container in the foreground, run the docker attach command and specify the name of the docker container in this case sad_ramanujan



I have a simple prompt application that when run asks for my name. And on entering my name prints a welcome message. If I were to dockerize this application and run it as a docker container like this, it wouldn't wait for the prompt. That is because by default the docker container doesn't listen to a Starndard Input. You must map the standard input of your host to docker container using the –I parameter.



Let's go back to the example were we run a simple web application in a docker container on my docker host. Remember the underlying host were docker is installed is called Docker Host or Docker Engine. When we run a containerized web application, it runs and we are able to see that the server is running, but how does a user access my application? As you can see my application is listening on port 5000. So I could access my application by using port 5000. But what IP do I use to access it from a web browser?

There are two options available. One is to use the IP of the docker container. Every docker container gets an IP assigned by default. In this case it is 172.17.0.2. But remember that this is an internal IP and is only accessible within the docker host. So if you open a browser from within the docker host, you can go to http://172.17.0.1:5000 to access the IP address.

But since this is an internal IP users outside of the Docker Host cannot access it using this IP. For this we could use the IP of Docker Host, which is 192.168.1.5 in this case. But for that to work, you must have mapped the port inside the docker container to a free port on the Docker host. For example, if I want the users to access my

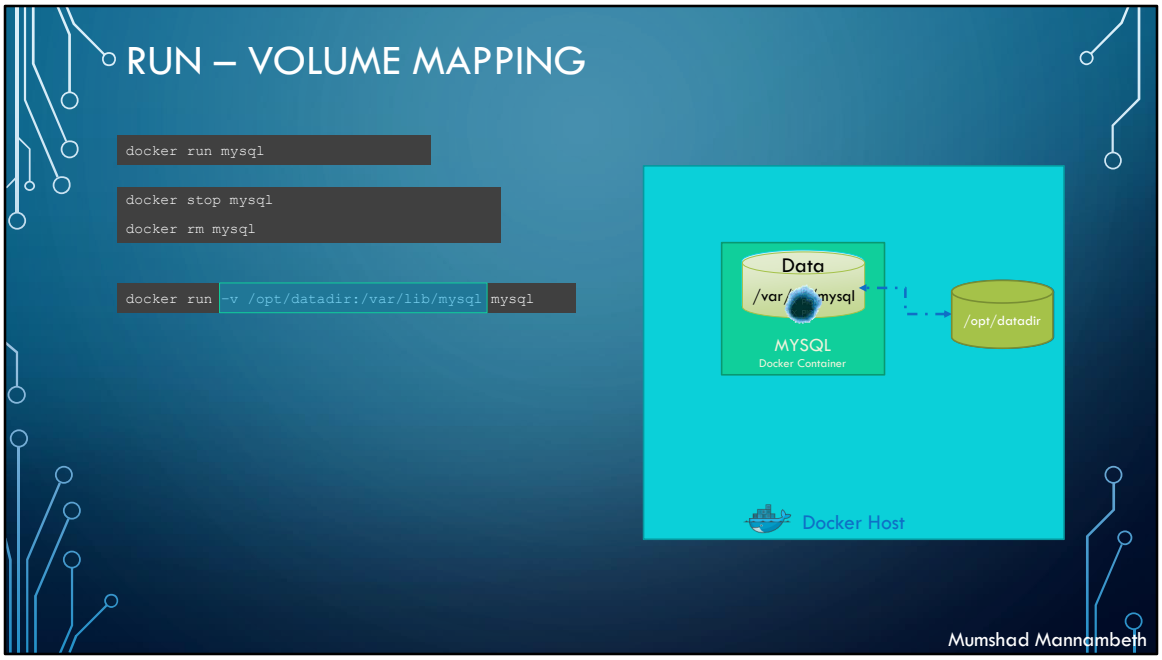
application through port 80 on my Docker host, I could map the port 80 of local host to port 5000 on the docker container using the –p parameter in my run command like this . And so the user can access my application by going to URL http://192.168.1.5:80 and all traffic on port 80 on my Docker host will get routed to port 5000 inside the docker container.

This way you can run multiple instances of your application and map them to different ports on the docker host.

Or run instances of different applications on different ports. For example, in this case I am running an instance of mysql that runs a database on my and listens on the default mysql port 3306 or another instance of mysql on another port 8306.

So you can run as many applications like this and map them to as many ports as you want. And of course you cannot map to the same port on the Docker host more than once.

We will discuss more about port mapping and networking of containers in the Network lecture later on.



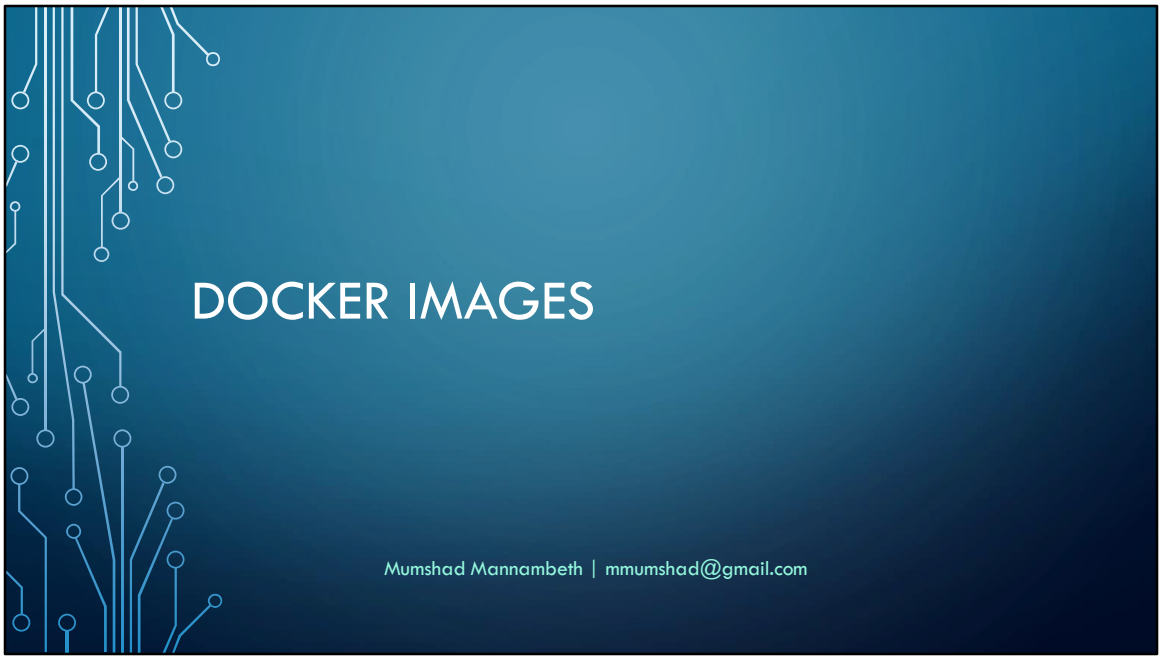
Let's now look at how data is persisted in a Docker container. For example, let's say you were to run a mysql container. When databases and tables are created, the data files are stored in location /var/lib/mysql inside the docker container. Remember, the docker container has its own isolated file system and any changes to any files happen within the container.

Let's assume you dump a lot of data in the database. What happens if you were to delete the mysql container and remove it?
As soon as you do that, the container along with all the data inside it gets blown away. Meaning all your data is gone. If you would like to persist data you would want to map a directory outside the container on the Docker host to a directory inside the container. In this case I create a directory called /opt/datadir and map that to /var/lib/mysql inside the docker container using the –v option and specifying the directory on the docker host followed by a colon and the directory inside the docker container. This way when docker container runs it will implicitly mount the external directory to a folder inside the docker container. This way all your data will now be stored in the external volume at /opt/datadir and thus will remain even if you delete the docker container.

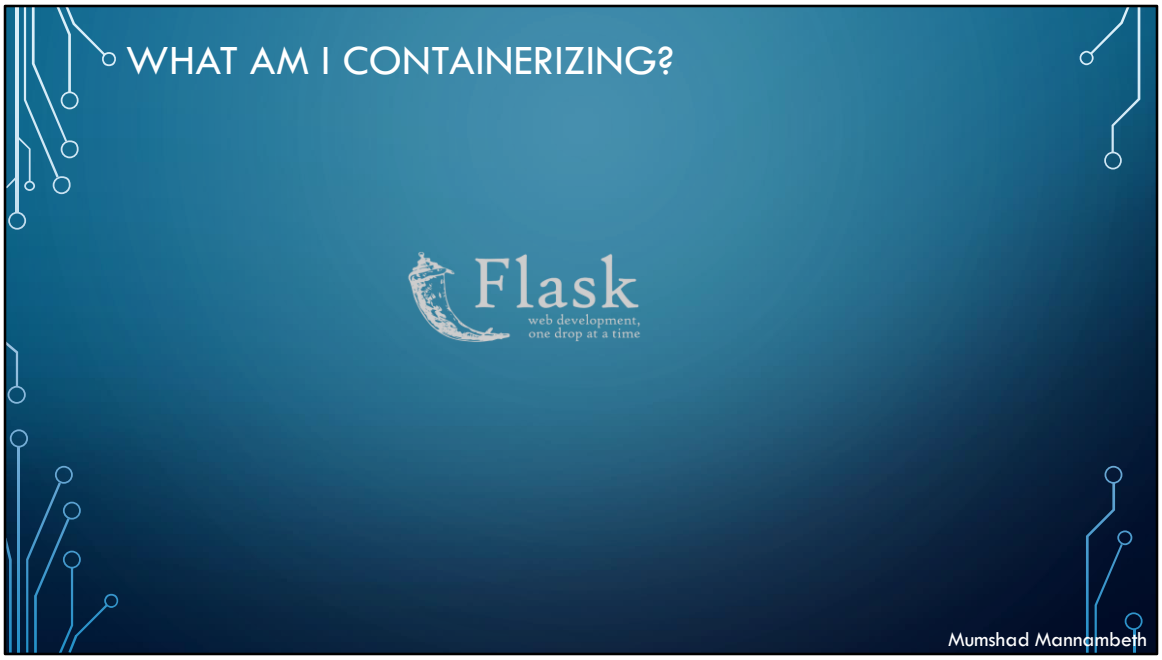
EXERCISES

- Tags
- Attach
- STDIN
- Port Mapping
- Volume Mapping

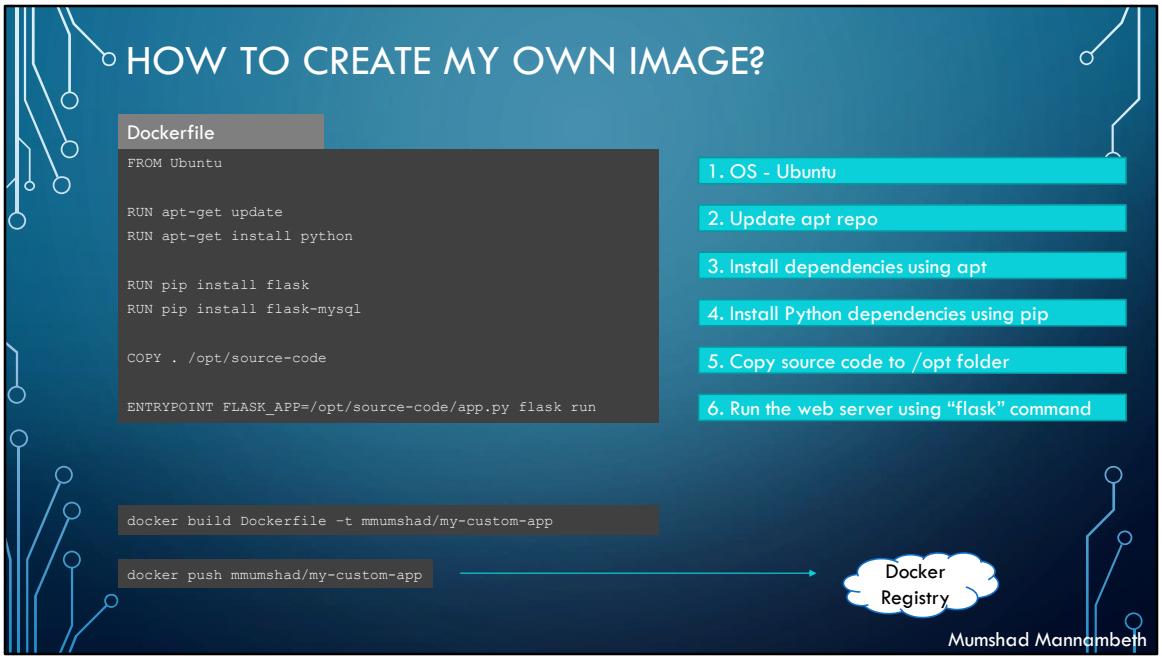
Mumshad Mannambeth



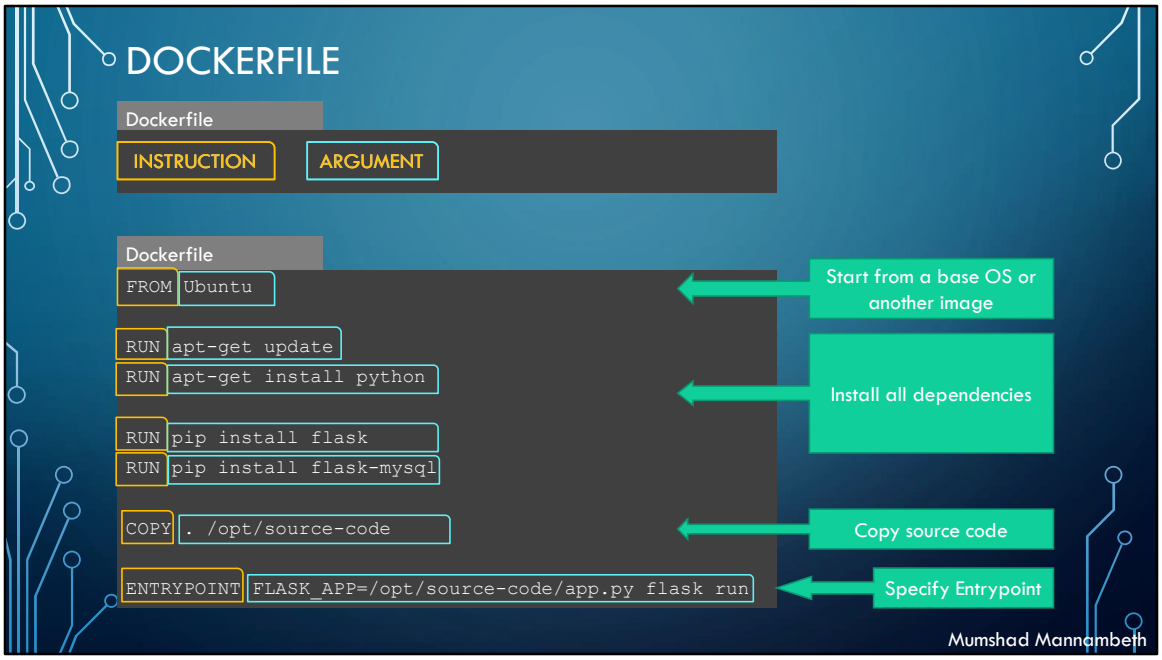
Hello and welcome to this lecture on Docker Images. My name is Mumshad Mannambeth and we are learning Docker Fundamentals.



Why would you need to create your own image? It could either be because you cannot find a component/service that you want to use as part of your application. Or you and your team decided that the application you are developing will be dockerized for ease of shipping and deployment.



First we need to understand what we are containerizing or what application we are creating an image for and how the application is built. So start by thinking what you might do if you want to deploy the application manually. We write down the steps required in the right order. I am creating an image for a simple web application. If I were to set it up manually, I would start with an OS like Ubuntu, then update source repositories using apt command, then install dependencies using apt command. Then install the python dependencies using pip and then COPY the source code of my application to a location like /opt. And finally run the web server using flask command. Now that I have the instructions, create a Dockerfile using these. Here is a quick overview of the process of creating your own image. Create a Dockerfile named "Dockerfile" and write down instructions for setting up your application in it. Such as installing dependencies, were to copy the source code from and what the entrypoint of the application is etc. Once done, build your image using docker build command and specify the Dockerfile as input as well as a tag name for the image. This will create an image locally on your system. To make it available on the public Docker registry, run the docker push command and specify the name of the image you just created.



Now let's take a closer look at that Dockerfile. Dockerfile is a text file written in a specific format that Docker can understand. It's in an Instruction and arguments format.

For example, in this Dockerfile everything on the left in CAPs is an instruction. In this case FROM, RUN, COPY and ENTRYPOINT are all instructions. Each of these instruct Docker to perform a specific action while creating the image. Everything on the right is argument to those instructions.

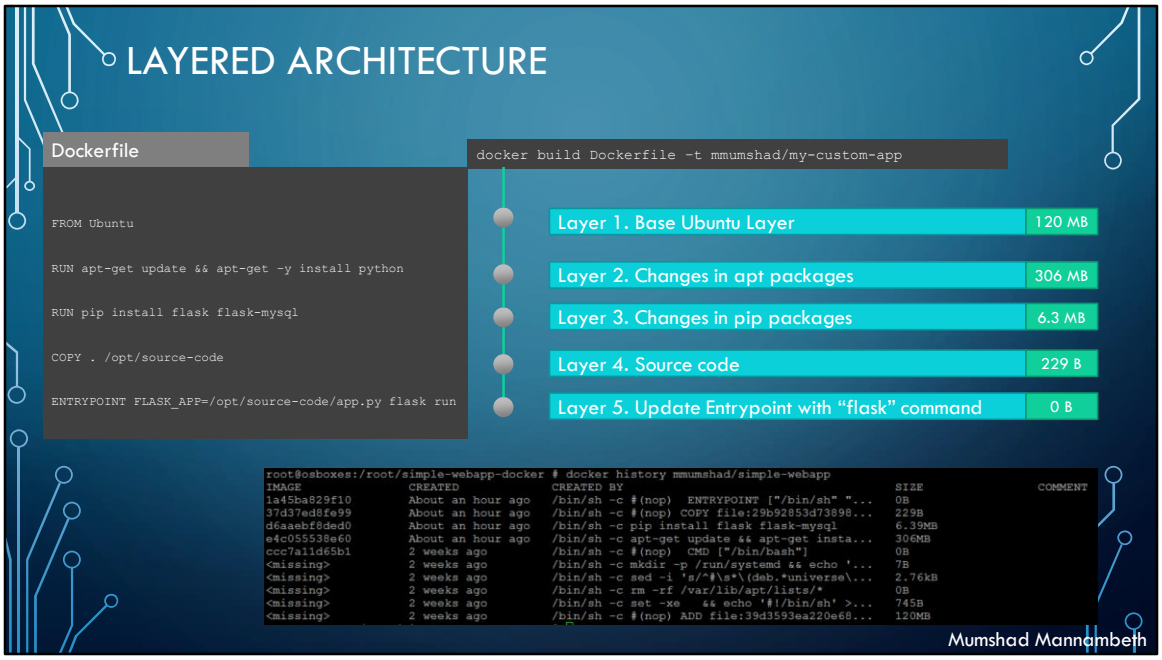
The first line, From Ubuntu, defines what the base OS should be for this container. Every docker image must be based of another image. Either an OS or another image that was created before based on an OS. You can find official releases of all Operating Systems on docker hub. Note that all Dockerfiles must start with a FROM instruction.

The RUN instruction instructs Docker to run the command on those base images. So at this point Docker runs the apt-get update command to fetch updated packages and installs required dependencies on the image.

Then the COPY instruction copies files from local system into the Docker image. In this case the source code of our application is in the current folder and I will be

copying it over to location /opt/source-code inside the docker image.

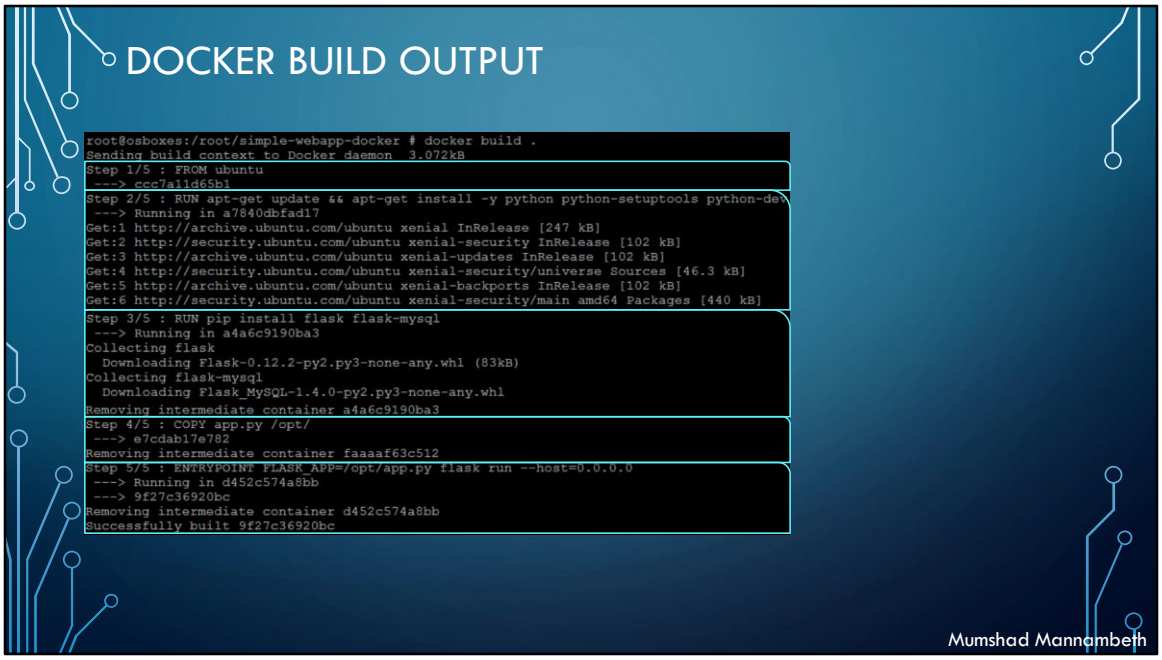
And finally ENTRYPOINT allows us to specify a command that will be run when the image is run as a container.



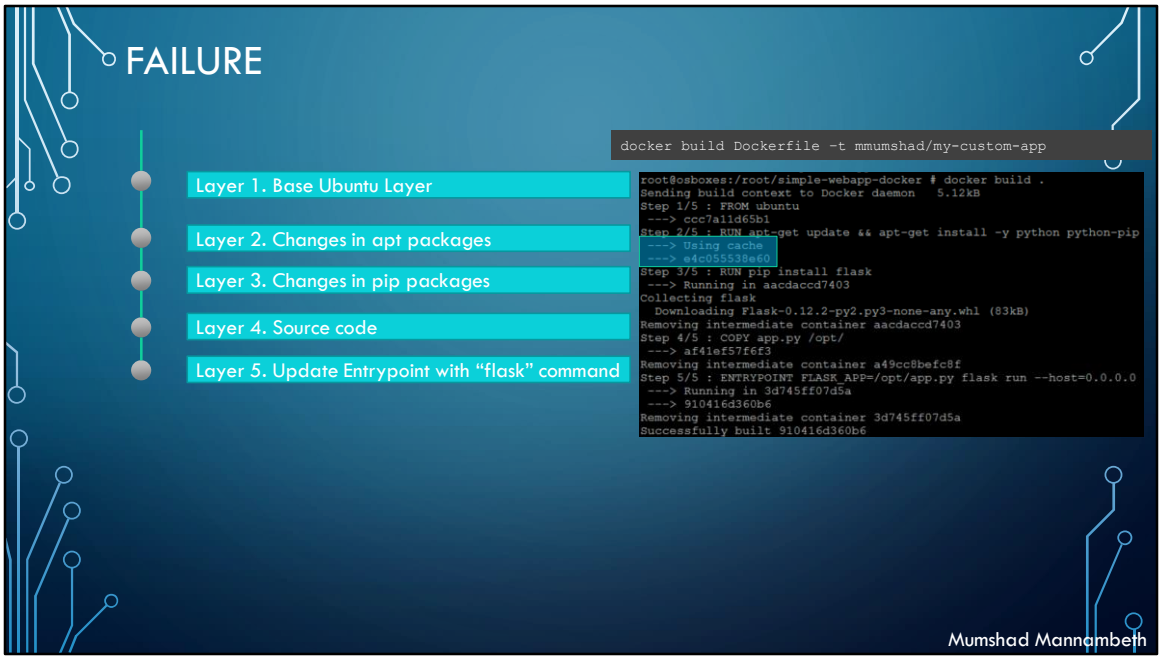
When docker builds the images, it builds these in a layered architecture. Each line of instruction creates a new layer in the Docker image with just the changes from the previous layer. For example, the first layer is a base Ubuntu OS, followed by the second instruction that creates a second layer which installs all the apt packages, and then the third instruction creates a third layer with the python packages, followed by the 4th layer that copies the source code over and the final fifth layer that updates the entrypoint of the image.

Since each layer only stores the changes from the previous layer, it is reflected in the size as well. If you look at the base Ubuntu image, it is around 120 MB in size. The apt packages that I install is around 300 MB, and the remaining layers are small.

You could see this information if you run the docker history command followed by the image name.



When you run the docker build command you could see the various steps involved and the result of each task. All the layers built are cached so the layered architecture helps you restart Docker build from that particular step in case it fails or if you were to add new steps in the build process, so you don't have to start all over again.

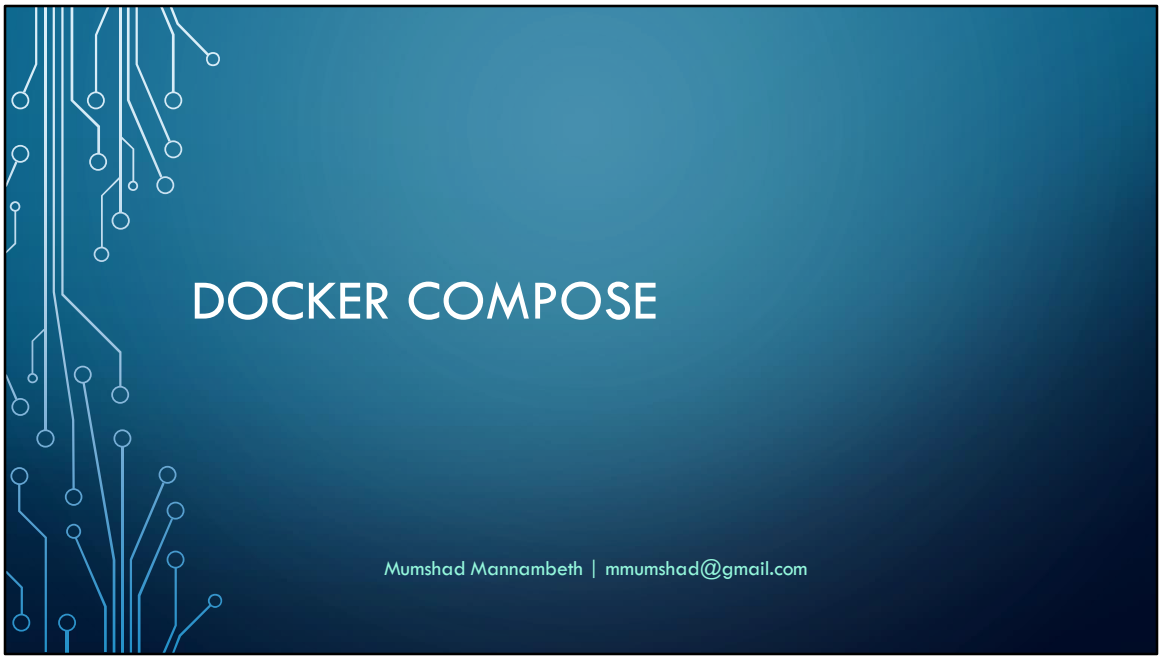


All the layers built are cached by Docker. So, in case a particular step was to fail, for example in this case Step 3 failed and you were to fix the issue and re-run docker build, it will re-use the previous layers from cache and continue to build the remaining layers. The same is true if you were to add additional steps in the Dockerfile. This way rebuilding your image is faster and you don't have to wait for Docker to rebuilt the entire image each time. This is helpful especially when you update source code of your application as it may change more frequently. Only the layers above the updated layers needs to be rebuild.

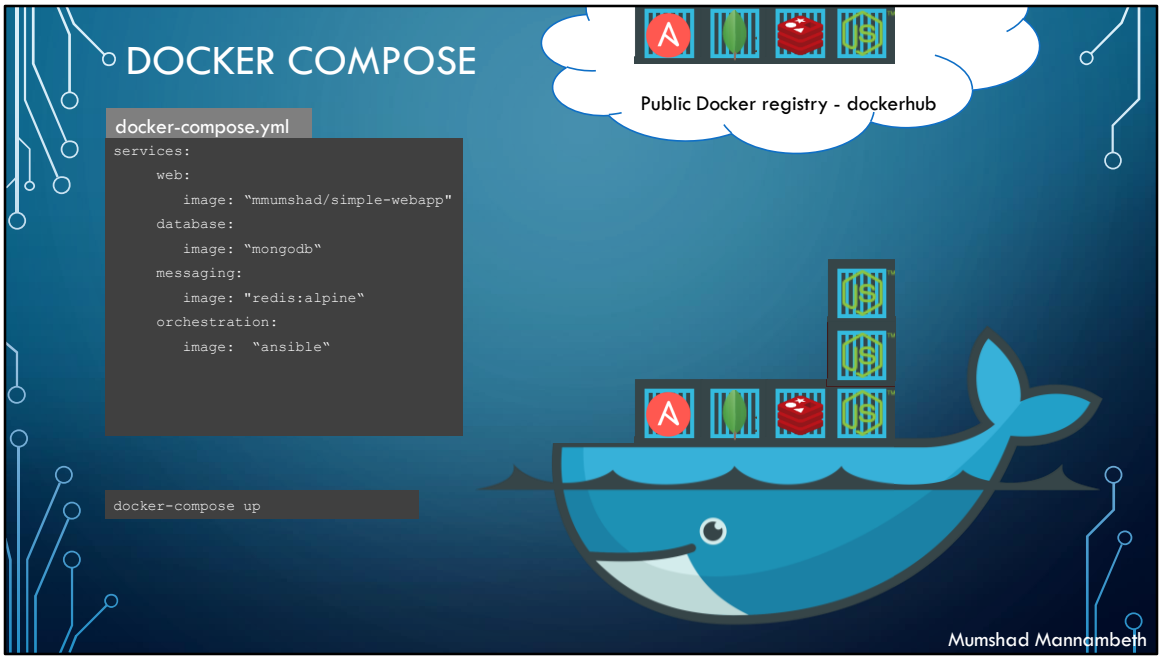
We just saw a number of products containerized. But that's just not it. You can containerize almost all of the applications, even simple ones like a browsers, or utilities like curl, applications like Spotify, skype etc. Basically you can containerize everything. And going forward I see that's how everyone is going to run applications. Nobody is going to install anything anymore going forward. Instead they are just going to run it. And when they don't need it anymore get rid of it.

EXERCISES

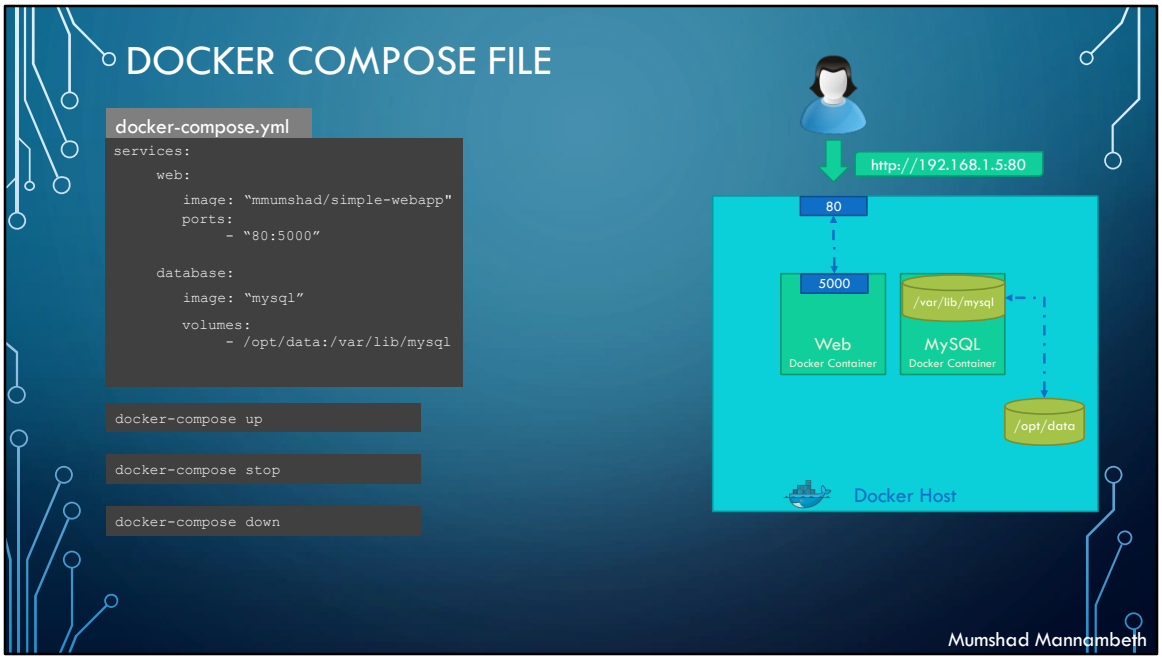
- Create your own Dockerfile



Hello and welcome to this lecture on Docker Compose. My name is Mumshad Mannambeth and we are learning Docker Fundamentals.



Earlier we saw how to deploy a stack using docker run command. You could use the docker run command if you wanted to deploy a test container of some kind. Instead of running separate docker run commands a better way to do this is to define your configuration in a docker-compose file. A docker compose file is a file in YAML format were you define the different services involved in your application such as web, database, messaging, orchestration etc. Once the file is defined, running the docker-compose up command will bring up the stack.



In case you haven't worked with YAML files before, please checkout the introduction YAML module at the end of this course in the APPENDIX section. That module provides a brief overview of YAML and some practice coding exercises on getting started with YAML. Go through that lecture first and come back here. Else, if you are good with YAML, please continue this lecture.

Let's take a closer look at the Docker Compose file. The docker compose file is a YAML file with a dictionary named services. It has a list of services defined in a key and value format. The key is the name of the service. This could be anything you want to name your service. In my case I have a 2 tier application - a web and database tier, so I named my services accordingly. The value for each service must be a dictionary with a minimum specification of an image. The image could be an image previously built or available on docker hub. My web image is a custom image I built with the repository named mmumshad/simple-webapp. And my database tier uses mysql image. Running the docker-compose up command will bring up the two containers as I specified.

However we have a problem. As discussed previously, an end user cannot access the web application if you don't map its ports to the Docker host. We know that we could

do this in command line, if we were to run the container using the docker run command like this. But in this case we are not using the docker run command instead we use the Docker compose command.

To map ports of a container in docker compose file, specify a ports property in the service properties and add the port mappings you need for that container.

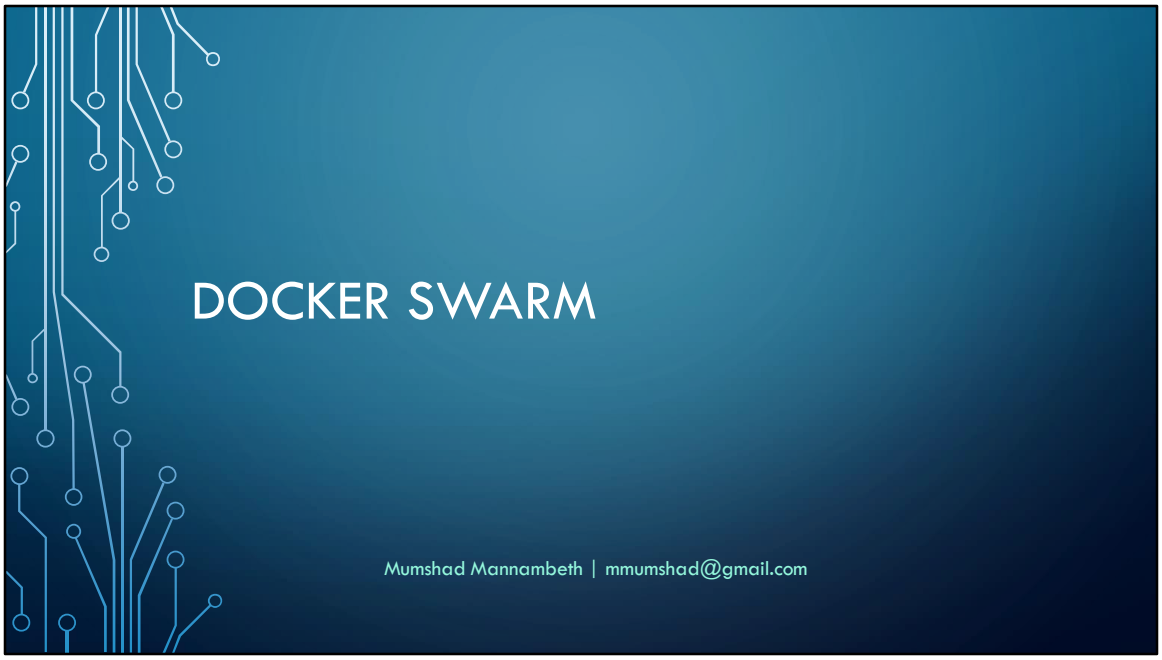
Similarly to map volumes in case of mysql server, specify a volumes property and add a list of volume maps from docker host to docker container.

Finally use docker-compose stop to stop the containers and docker-compose down to bring everything down and remove the containers entirely.

EXERCISES

- Create your own Docker Compose

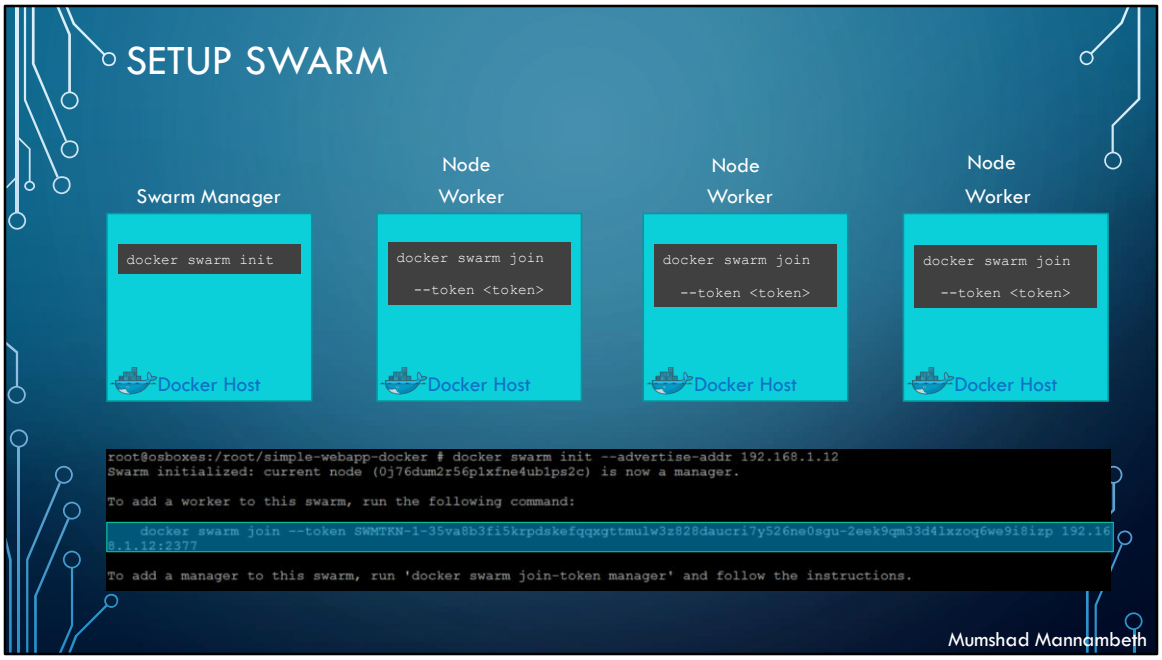
Mumshad Mannambeth



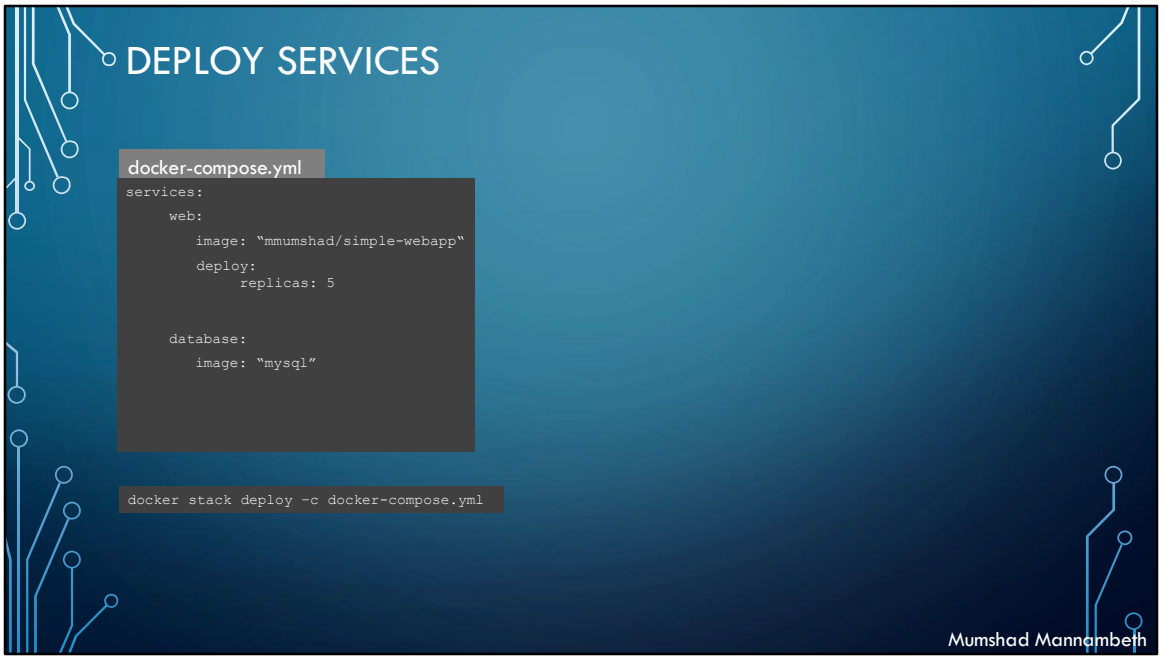
Hello and welcome to this lecture on Docker swarm. My name is Mumshad Mannambeth and we are learning Docker Fundamentals. This is an advanced topic and is out of scope for this beginners guide, but we will just go through and understand it at a high level.

Up until now, we have been working with a single Docker host and running containers on it. This is good for dev/test purposes, but we wouldn't want to use this configuration in production because this is a single point of failure. If the underlying host fails, we lose all the containers and our application goes down.

This is were Docker Swarm comes into play. With Docker swarm you could now manage multiple Docker Machines together as a single cluster. Docker swarm will take care of placing your services into separate hosts for high availability.

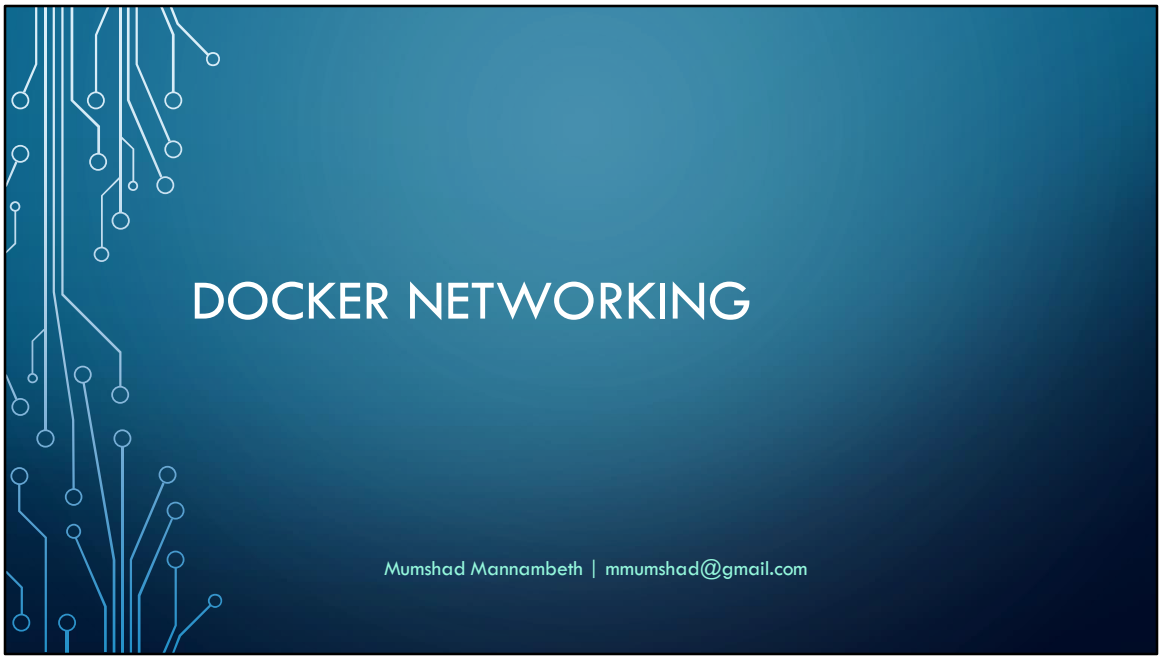


Setting up Docker swarm is easy. First you must have hosts with Docker ready. You must designate one host to be the master or the Swarm Manager and others as slaves or workers. When you are ready run the docker swarm init command on the swarm manager and that will initialize the swarm manager and provide the command to be run on the workers. Copy the command and run it on the worker nodes to join the manager. After joining the swarm, the workers are also referred to as nodes. You are now ready to create services and deploy them on the swarm cluster

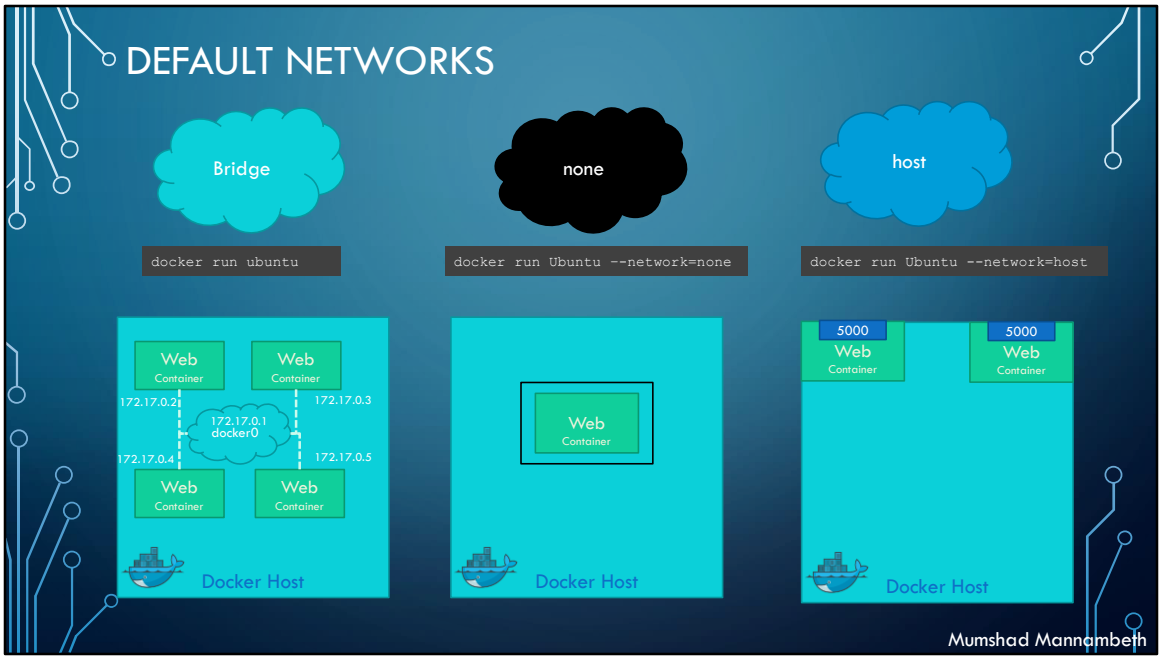


Let us start be using the same docker compose file we used earlier. Now to deploy multiple instances of a service across docker hosts using swarm add a new property to the image called deploy and specify the number of replicas required in it. In this case 5.

To run the application, execute the docker stack deploy command and specify the docker-compose file name. This will deploy 5 instances of application across Docker hosts. There are some additional steps required to configure load balancing, but those are out of scope for this basic course.



Hello and welcome to this lecture on Docker Networking. My name is Mumshad Mannambeth and we are learning Docker Fundamentals.

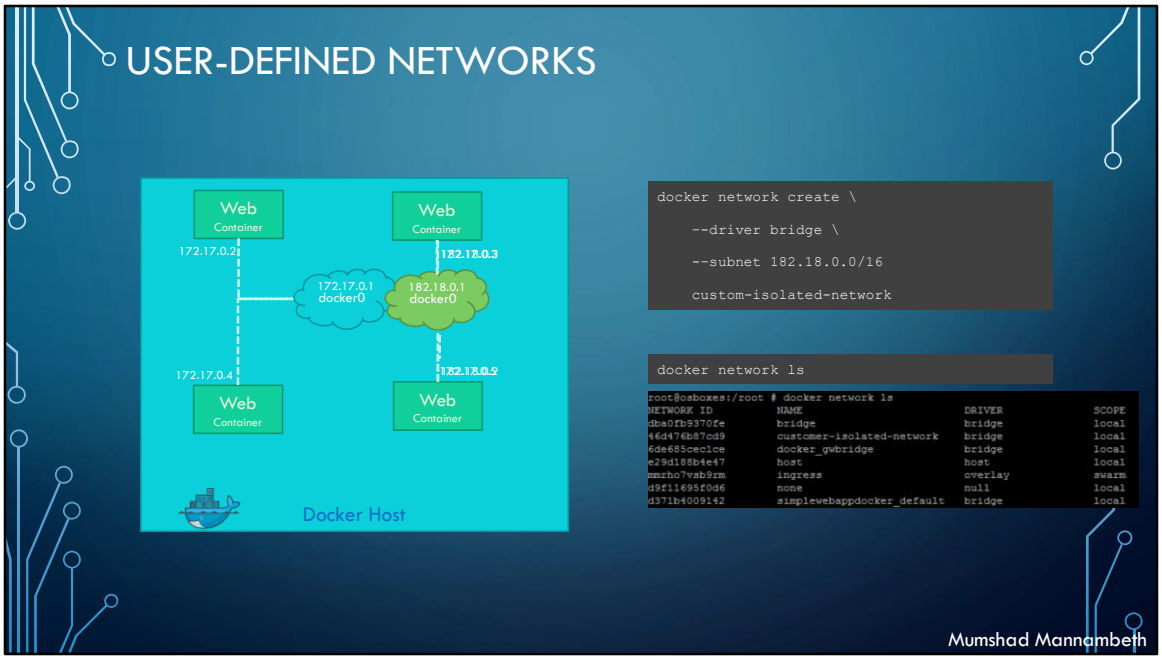


When you install Docker, it creates three networks automatically – Bridge, Null and Host. Bridge is the default network a container gets attached to. If you would like to associate the container with any other network specify the network information using the network command line parameter like this.

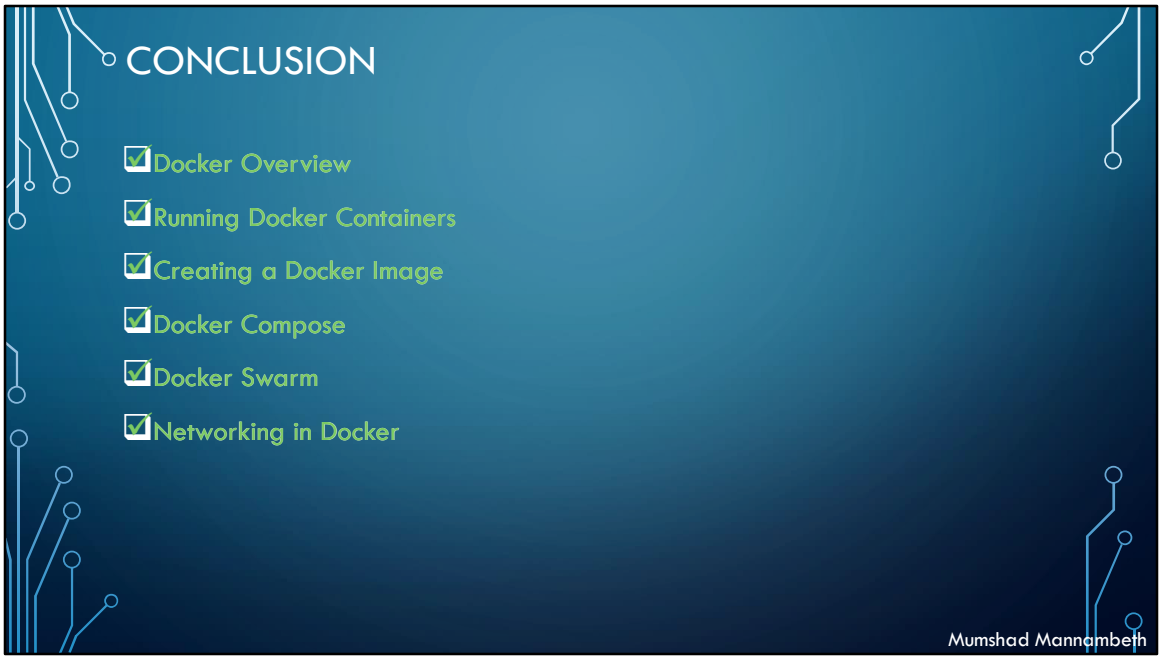
The bridge network is a private internal network created by Docker on the host. All containers attach to this network by default and they get an internal IP address usually in the range 172.17 series. The containers can access each other using this internal IP if required. To access any of these containers from the outside world is to map ports of these containers to ports on the Docker host as we have seen before.

Another way to access the containers externally is to associate the container to the host network. This takes out any network isolation between the docker host and the docker container. Meaning if you were to run a web server on port 5000 in a web app container it is automatically accessible on the same port externally without requiring any port mapping as the web container uses the host network. This would also mean that unlike before you will now not be able to run multiple web containers on the same host on the same port as the ports are now common to all containers in the host network.

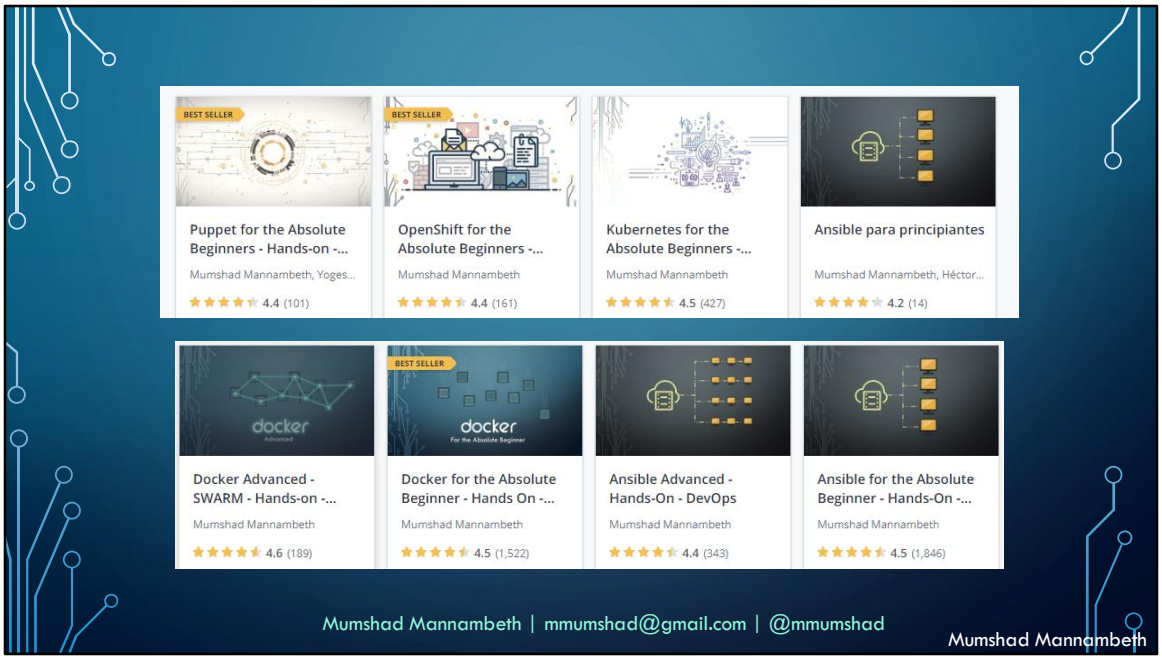
With the none network the containers are not attached to any network and doesn't have any access to the external network or other containers.



So we just saw the default bridged network, with the network ID 172.17.0.1. So all containers associated to this default network will be able to communicate to each other. But what if we wish to isolate the containers within the Docker host. For example the first two web containers on internal network 172 and the second two containers on a different internal network 182? By default Docker only creates one internal bridge network. For this we could create our own internal network using the command docker network create and specify the driver which is bridge in this case and the subnet for that network followed by the custom-isolated network name.
 Run the docker network ls command to list all networks.

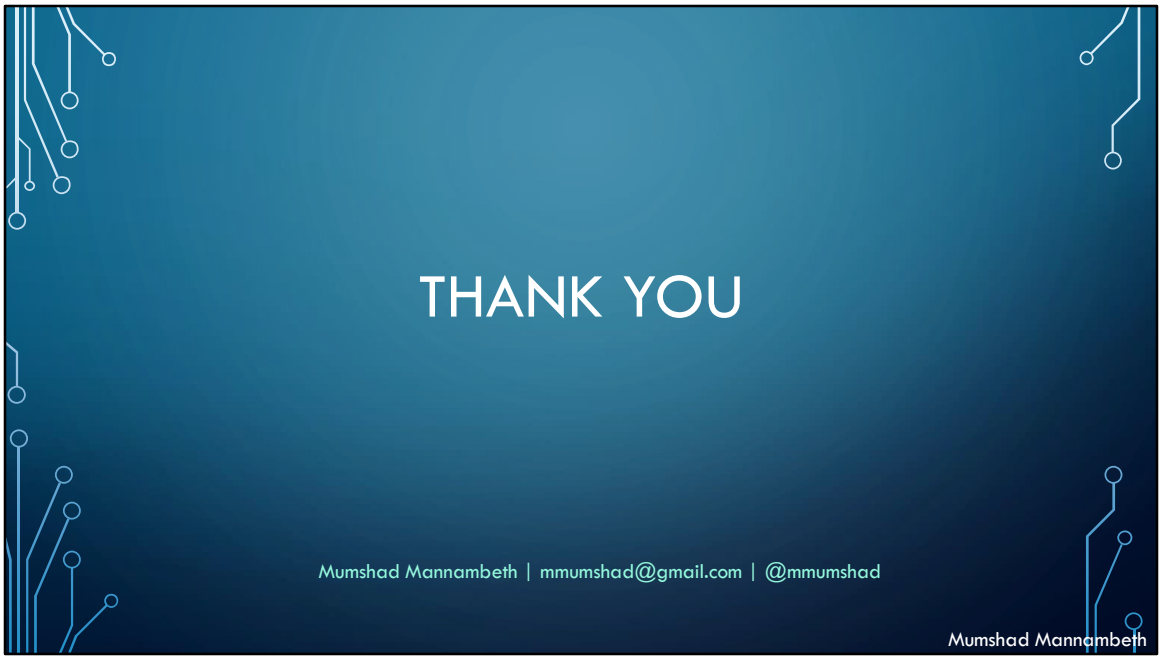


That concludes the Docker beginners course. We covered the real basics of Docker, understood the various concepts such as the Architecture, what containers and images are. We saw how to install and get started with Docker and how to run containers in different ways. We also looked at how to create your own image and practiced developing some Dockerfiles. We also went at a high level on Docker Compose and Docker Swarm. And finally we looked at various types of Networks in Docker. That's it for this begineers course and I will hopefully develop and Advance course on Docker covering some advanced topics and looking deeper at Docker Compose and Docker Swarm. I hope this was a good learning and you have enough to get started on your Docker journey. So Thank you very much for your time, please leave a review and share this course with your friends eager to learn Docker. Also feel free to check out the other courses in the DevOps series on Ansible. Until next time, Happy learning and happy containerizing.



Check out my other courses.

Until next time take care and happy learning!



Thank you very much for attending the Docker beginners course and I hope you enjoyed learning. As always I put my best teaching techniques in place and I always aim for a 5 star rating. In case you feel there is any area that I can improve upon or you feel something is missing from the course feel free to reach out to me. I am currently working on an Advanced course on Docker covering other topics in-depth, more demos and more coding exercises. And of course I will send out an announcement once the course is published. So kindly watch out for it. In case you think I should include any special topics in the advanced course, please please reach out to me by sending me a direct message on Udemy or reach me by email at mmumshad@gmail.com or on twitter @mmumshad.