# Docker

## SECTION-1

-> Docker is a container technology, a tool for creating and managing containers.A container is nothing but a package of code and dependencies to run that code.we can run exactly same application with same environment wherever that is.docker is a tool for building these containers.

-> support for containers is built into modern operating systems or at least there it's easy to get started with them, and Docker can be installed on all modern operating system to then work with it there. And Docker then in the end is a tool that simplifies the creation and management process off these containers. You wouldn't need it to create containers but it's the de facto standard for doing that since it makes that task so super simple.

-> with docker we can have same production and development environments(this ensures that it works exactly as tested).It should be easy to share common development environment.when we change project we don’t need to uninstall and re-install local dependencies and runtimes all the time.

-> Virtual machines:
Machines that run on our machine.virtual machine encapsulated with virtual os in their own shell independent from our host os.

-> we can solve the problem of having same production and development environment with virtual os.but having more virtual machines on host machine wastes a lot of space on hard drive and tends to be slow.so docker came into picture.

-> containers is the key concept , docker is just standard tool for creating and managing them.

-> we utilize built-in container support, which our operating system has or emulated container support, something Docker will take care about that this works. And then we run a tool called the Docker Engine on top of that. And that will all be set up by Docker when we install it, by the way. And then, based on that Docker Engine, which now runs on our system, which is just one tool, one lightweight small tool being installed there, we can spin up containers. And these containers contain our code and the crucial tools and runtimes our code needs, like Node.js, for example, but they don't container a bloated operating system, tons of extra tools or anything like that. They might have a small operating system layer inside of the container but even that will be a very lightweight version of an operating system, much smaller than anything you would install in a virtual machine. And you're going to see how containers work exactly and how you create them throughout this course, of course. Now, the other great thing about containers is that you can configure and describe them with a configuration file and you can then share that file with others

so that they can recreate the container or you can also build the container into something, which is called an image.

-> Docker setup:
Inorder to wrok with docker we need to install docker. Visit 'docker.com' and check for requirements.if matches then install docker desktop else install docker toolbox.

-> To install docker:
Go to docker.com => developers => Docs => download and install => Docker desktop for windows => system requirements(check in order to install it)
-> wsl2 is in the end is linux installation in windows installation

Commands:
-> - - help : to see all available options
-> docker build . => it builds image.here '.' location where Docker file is present.-t is used to add name:tag to image.ex: docker build -t goals:latest .

-> docker run -p portNumber:portNumber <image_id> => to run image.(First port number is local port to access application,second port number is internal docker container exposed port number)(creates a new container)
-p stands for publish and this allows us to tell docker under which local port on our machine ,internal docker specific port should be accessible.by default attached mode.we can use '-d' to run in detached mode.If we want to run in attach mode then we can use 'attach' or '-a'
-v volume_Name : path where volume should store in container.Ex: -v feedback:/app/feedback
- - network <Network_Name>

-> docker ps => list all running containers(ps:processes)
-> docker ps -a => list all containers including stopped ones.
-> docker stop <container_name> => to stop running container.
-> docker run -it <image_name> => '-it' is for interactive mode.It allows us to enter commands.'--rm ' is used to automatically remove a container when it is stopped.--name to name the container.
Images:
-> -t or -tag : can be tagged or name
-> docker images : can be used to list docker images.

-> docker image inspect : to analyze

-> docker rmi , docker prune : To remove images.

-> docker ps - - help : shows all available options for 'docker ps'

-> docker start <containerName> : instead of creating a new container,we can start existing container.by default detached mode.-a and -t to enter interactive mode.

-> docker attach <container> **:** Attaching to an already-running Container.

-> docker logs <container_name> **:** to know past logs of a container.

-> docker rm <containerName> **:** To remove a container.

-> docker rm <containerName1> <containerName2> <containerName3> **:** To remove multiple containers.

-> docker container prune **:** To remove all stopped containers at once.

-> docker image history <img_id> **:** shows size of each instruction in a docker file.

-> docker rmi <img_id> **:** to remove unused image.No container should us that image.

-> docker image prune **:** removes all unused images.

-> docker image inspect <img_id> **:** To inspect image

-> docker cp source <containerName>:destination **:** copying files to and from a container while it is running

EX: docker cp dummy/. boring_vaughan:/tests

-> docker push <img_name> : to push images

-> docker pull <img_name> : to pull images

-> docker tag oldName:tag newName:tag **:** To rename docker image.

-> docker login: to login to docker hub.once loggedin we will stay logged till we logout.

-> docker volume ls: lists all volumes

-> docker network create <Network_name> : To create network

-> docker network ls : to list networks

## SECTION-2

-> we need to create image.containers are based on image.To create image , create a docker file

-> image is the blue print for creating container.image contains code+required tools /runtime.from the image containers are created.from single image multiple containers can be created.

-> image contains all the code and logic container needs.
-> image is template which contain code and application.containers are then running application.
-> containers are nothing but running instances of images.
-> two ways to create or getting an image from which we can create container.
1) use existing image or build-in image.(Docker hub)
2) create our own image.

-> Dockerfile contains instructions that we can execute when we create an image.

1) FROM image_name
This allows you to build your image up on another base image.image is either exists in our system or image form docker hub.

4) WORKDIR location:
EX: WORKDIR /app
It tells docker where commands should be run. RUN npm install is executed in root directory in container.

2) COPY . . or COPY . /app
Every image and container created using image have their own filesystem which is totally detached from filesystem on your system.first dot tells docker to copy all folders and subfolders present where this docker file is there into docker container.second dot is location where these files should be stored inside of container.
COPY . /app here all files from local system where docker file is present to /app folder in image or container./app folder is created if not present in container or image.

3) RUN command_name :
Ex: RUN npm install
RUN is to execute commands

* we run container in the end not image.image is just template for creating container.so we don't want to run server in image.all the code and dependencies are there in image but server should be run only when a server is running.so CMD is used to start the server.difference between CMD and RUN is that CMD is not exected when image is created, but executed when container is started based on image whereas RUN is executed when image is created.

6) EXPOSE Port_number:
EX: EXPOSE 80

When container is started, we want to expose certain port number to our local system(a machine which runs this container).

5) CMD [command_torunserver]:
EX: CMD [“npm”,”server.js”]
It is executed when the container is started.If CMD is not specified then CMD of base image is executed.with no base image and no CMD , we get an error.this should be at the end of the Dockerfile.

-> Images are read-only, we cannot edit images from outside (from source code). When any changes have made in source code, we need to build image again.changes will not reflect on image.bcz in image code is already copied.

-> Image layers:

whenever you build an image, Docker caches every instruction result, and when you then rebuild an image, it will use these cached results if there is no need to run an instruction again. And this is called a layer based architecture. Every instruction represents a layer in your Dockerfile. And an image is simply built up from multiple layers based on these different instructions.
All layers after changing layer will be build again.So keep “npm install” on top of “copy” command to avoid creating remaining layers all the time when image is created.but ensure that package.json file is copied(COPY package.json /app).

*-> containers are isolated from each other.
*-> when multiple containers are created then each container should have different port number.
-> In attached mode we can see console logs, but cannot enter commands.In detached mode we cannot see console logs but can enter commands.
-> If we want to run in attach mode then we can use ‘attach’

-> copying files to and from a container while it is running **:**
docker cp source <containerName>:destination
EX: docker cp dummy/. boring_vaughan:/tests
Here dummy is a folder and it contains some files in it.it will copy all files into running container in /tests folder.if it is not there then that folder will be created.

-> naming and tagging images and containers:
Naming containers “--name “ is used to name the containers.
Tagging the images is like

Name:tag => here name is also known as repository ,and tag can be used to give more information like version.suppose if more than one name is same then those names can be differ by tag.
Ex: docker build -t goals:latest .

-> sharing images**:**
everyone who has image can create containers.two ways to share.
1) share a docker file:
We can simply use build .
Docker file instructions might need surrounding files or folders.
2) share a build image:
Download image and run container based on it.

-> sharing images:
1) docker hub
official docker image registry
Pubic ,private and official images
2) private registry
1) docker hub
docker push <img_name>
docker pull <img_name>
For private repository need to include host:name.

-> pushing an image to docker hub:
* create an account on docker hub
* create repository
* rename image with repository name
docker tag oldName:tag newName:tag ( this is to have local repository available)
* login to dockerhub from terminal
docker login
* docker push <repositoryName>
-> pulling image from docker hub:
docker pull <repository>
Docker pull will always pulls latest image.
you could also run Docker Run without running Docker Pull first. If Docker Run doesn't find an image locally on your machine, it will automatically reach out to the container history, your image name uses so here in this case, Docker Hub, and it will check for the image there. And if it finds an image of that name there it will use that and pull that automatically. What Docker Run will not do though, and that's important is automatically check for updates. If you have an image locally because you pulled or ran it before, then if

you run a container again based on an image, Docker Run will not check if the image you have locally on your system is the latest version of that image. So if in the meantime, you updated the image and pushed it again to Docker Hub, just using Docker Run there after will not give you that latest updated image. You instead manually need to run Docker Pull and then the image name first to ensure that you have that latest image version and you then execute Docker Run again. So it automatically pulls images but not necessarily the latest version if you already have that image locally on your system, because you pulled it before already.

## SECTION-3

-> images are read-only.
-> data stored in images is read-only.
-> data stored in containers is temporary
-> data in volumes is permanently stored,even after deletion of containers.
-> If we don't specify tag when creating image,by default it takes latest.
-> problem:
Containers are isolated,so data in one container does not effect data on another container.when containers are deleted, data will also be deleted.data in one container is not available in another container.
-> volumes:
Volumes persist data even after deleting container.volumes are folders on host machine hard drive which are mounted(available or mapped) into containers.with volumes we can connect local host machine folders with folders in containers.changes in either folder reflects in another one. Containers can read and write data to volumes.

-> two types of external data storage:
1) Volumes ( managed by docker) => named volumes ,anonymous volumes
Docker set up a folder or path on our host machine , exact location is unknown to developer.managed via docker volume commands4

-> anonymous volumes:
Anonymous volumes exists as long as our container exists.data disappears if we remove our container.bcz it re-creates volume whenever container is created.anonymous volumes are attached to specific container.
In docker file add:
VOLUME ["/app/feedback"] or
-v path

Here this path specifies path in container where volumes should be stored.

-> named volumes:

Named volumes persist data even after removal of containers.defined path in container is mapped to the created volume.we don't know the path where the volume on our host present. Named volumes will not be deleted by docker.named volumes are not attached to container.

When a container is created with volume and deleted, we can see same data by creating another container with same volume.we cannot edit named volumes since we dont know location.

-> We saw, that anonymous volumes are **removed automatically**, when a container is removed.

This happens when you start / run a container with the `--rm` option.

If you start a container **without that option**, the anonymous volume would **NOT be removed,** even if you remove the container (with `docker rm ...`).

Still, if you then re-create and re-run the container (i.e. you run `docker run ...` again), a n**ew anonymous volume will be created**. So even though the anonymous volume wasn't removed automatically, it'll also **not be helpful** because a different anonymous volume is attached the next time the container starts (i.e. you removed the old container and run a new one).

Now you just start **piling up a bunch of unused anonymous volumes** - you can **clear them** via `docker volume rm VOL_NAME` or `docker volume prune`.

2) Bind Mounts ( managed by us)

If we made any changes in local folders it will not reflect on folders in container.but with bind mounts it is possible.difference between bind mount and named volume is that , for named volumes developer does not know folder location in local machine whereas bind mounts we will create a folder in local machine and map it to folders in container.we can edit data in volumes.

Command:

-v absolute path from local machine:path in container.

Ex:

- v Users/udemy/docker-complete : /app

We need to make sure that docker has access to folder.

Shortcut for bind mounts: `-v "%cd%":/app`

mounts

-> Summary of anonymous,named and bind mount volumes:

-> anonymous volumes : -v location_in_container

Created specially for single container.servives container shutdown/restart until --rm is used.cannot be shared across containers.since it is anonymous it cannot be re-used (even on same image).

-> named volumes : -v name:location_in_container

Created in general, not tied to specific container.servives container shutdown/restart/removal.can be shared across container.

-> bind mount : -v absolute_path_local:path_in_container

Readonly volume:

After container internal path in bind mount syntax , we need to add ":ro"

Ex: -v users/admin/docker:app/:ro

in the container, if we specify a more specific sub-volume, so to say, then that sub-volume overrides the main volume, you could say, just like this anonymous volume ensures

-> Bind mount vs COPY:

at the time of development we use bind mount which provides all the code , so we think not to use COPY.but We don't use bind mount at the production.we only use at the time of development.

-> .dockerignore:

".dockerignore" file can allow us to avoid copying specific files into containers.whatever files we don't want to copy into container , we can include that files into ".dockerignore" file.

-> Arguments and environment variables:

Docker supports build-time arguments and runtime environments. Allow you to create more flexible images and containers.

-> ARG is available in docker file and not accessible in CMD and any other application code.set on image build via - - build - arg (image build).

-> ENV is available in docker file and in application code.set via ENV in docker file or via - -env on docker run.

-> If we specify "ENV PORT 80" in docker file then it is available in entire application environment.

On run command we can add - -env PORT=8000(- -env key:value )

-> why environment variables can be helpful. They help us run one and the same container based on one and the same image in different modes, in different configurations, you could say.

-> **Environment Variables & Security**

One important note about **environment variables and security**: Depending on which kind of data you're storing in your environment variables, you might not want to include the secure data directly in your `Dockerfile`.

Instead, go for a separate environment variables file which is then only used at runtime (i.e. when you run your container with `docker run`).

Otherwise, the values are "baked into the image" and everyone can read these values via `docker history <image>`.

For some values, this might not matter but for credentials, private keys etc. you definitely want to avoid that!

If you use a separate file, the values are not part of the image since you point at that file when you run `docker run`. But make sure you don't commit that separate file as part of your source control repository, if you're using source control.

-> ARG:

With those, we can actually plug in different values into our Dockerfile, or into our image when we build that image without having to hard code these values into the Dockerfile.

In docker file:

```
ARG DEFAULT_PORT=80
ENV PORT $DEFAULT_PORT
EXPOSE $PORT
```

With this we can build multiple images with same docker file without changing docker file.
In command prompt => “docker build feedback:dev - -build-arg DEFAULT_PORT=8000 .”

## SECTION-4

-> Networking:

-> our application in container communicates with www which is out of container without need to make any configurations.
-> container to host machine communication can be done through “host.docker.internal”(host.docker.internal acts as ip address of host machine) in the url.
Ex: mongodb://<host.docker.internal>:27017/swFavourites

-> container to container communication:

In order to make them communicate with each other we have to keep them in a same network.
For creating network :
docker network create <Network_name>
It is a docker internal network which we can use on docker containers.
When running container we can add - - network <Network_Name>

The containers communicate with each other in same network by using container name in the url.
Ex: mongodb://<container_Name>:27017/swFavourites

**-> Docker Network Drivers**

Docker Networks actually support different kinds of "**Drivers**" which influence the behavior of the Network.

The default driver is the "**bridge**" driver - it provides the behavior shown in this module (i.e. Containers can find each other by name if they are in the same Network).

The driver can be set when a Network is created, simply by adding the `--driver` option.

```
docker network create --driver bridge my-net
```

*Of course, if you want to use the "bridge" driver, you can simply omit the entire option since "bridge" is the default anyways.*

Docker also supports these alternative drivers - though you will use the "bridge" driver in most cases:

- **host**: For standalone containers, isolation between container and host system is removed (i.e. they share localhost as a network)
- **overlay**: Multiple Docker daemons (i.e. Docker running on different machines) are able to connect with each other. Only works in "Swarm" mode which is a dated / almost deprecated way of connecting multiple containers
- **macvlan**: You can set a custom MAC address to a container - this address can then be used for communication with that container
- **none**: All networking is disabled.
- **Third-party plugins**: You can install third-party plugins which then may add all kinds of behaviors and functionalities

As mentioned, the "**bridge**" driver makes most sense in the vast majority of scenarios.

# SECTION-5

-> Building multiple container applications with docker.
-> suppose if we have mongo db as database then we can dockerize db by just running official mogo db image.
-> backend can be dockerzed by creating a docker file.after creating and running docker image,we encounter an error , if we use localhost in connecting to db.we have to change it to host.docker.internal:27017.

-> same as backend frontend can also have docker file.it also expose port when running container.
-> ** when running container we use expose port.this is to  listen requests from backend.in backend code we write like
connect(mongodb://host.docker.internal:27017//courseGoals) this sends requests to db.

-> using network.
  -> create network using “docker create network network_name”
  -> build mongo image.
  -> run mongo image with network name.”docker run -d - -network network_name mongo”
  -> change backend code (request data to mongodb) mongodb(container name).
  -> run backend “docker run -d - - network network_name -p 80:80 image_name” here we provided network name and also published ports because frontend code works in browser.so it will not be the part of container network.in order to communicate with backend frontend need port which is exposed in backend.
 -> In frontend code (request data to localhost).when running frontend it does not require network.

-> if we dont use volumes when running mongodb then we loss data.in order to persist data on container removal use “-v data:/data/db” when running mongodb image.
-> to provide security to database , use “ -e MONGO_INITDB_ROOT_USERNAME=max  -e MONGO_INITDB_ROOT_PASSWORD=secret” for this to work we need to change backend connection code to
“ max:secret@mongodb:27017” and at the end of connection string add “?authSource=admin”
Here max is username and secret is password.

-> Longer container internal paths have precedents and overwrite shorter paths.so shorter paths should be written at right side of longer paths.

-> when we use bind mount to get changes directly to container when something changes locally ,then we have to use named volume right side of bind mount to persist nodemodules .this avoids overwriting node modules when something change locally.this tells to persist nodemodules even there is bind mount.

-> So, if the code changes their author,
this has no impact on the already running node server.
And that's of course not what we want here.
We want the node server to restart
whenever the code changes.
Yes, we could stop and restart the container,
that would do the trick, but even better than that,
we can add the extra dependency to this project,
which then actually will restart the server
automatically for us when the code changes.
For this, I'll delete package.JSON,
and here in packaged.js, I'll add a new devDependencies node
below dependencies, and add nodemon here as a dependency,
and I'll pick version 2.0.4.
Nodemon is an extra tool,
which I already covered early in the course,
which watches the project folder for file changes
in JavaScript files,
and if such a JavaScript file does change,
it restarts the node server.
We just have to ensure that to utilize nodemon,
we can do that by adding a start script here
to the scripts section.
And there, I wanna run nodemon app.js
to use nodemon to run app.js,
and under the hood nodemon uses node,
but it restarts the node server whenever any JavaScript file
in the project folder,
so in this case, in the backend folder changes.
So these are changes we need to make to package.js,
in the docker file of the backend project,
we now need to run this start script,
and can start like this to utilize nodemon.
And that of course now means that we need to stop
our goals backend container,

so there's node application container we started,
and then restart it, but based on a rebuilt image
because we changed the docker file.

-> in Dockerfile
ENV MONGODB_USERNAME=root
ENV MONGODB_PASSWORD=secret

In connection string use `` write string between backticks and use
${process.env.MONGODB_USERNAME }:${process.env.MONGODB_PASSWORD}

When running container as usual.

-> .dockerignore
node_modules
Dockerfile
.git

## SECTION-6

-> Docker compose:
Docker Compose is a tool that allows you to replace
Docker build and Docker run commands
and they're not just one Docker build
and one Docker run command,
but potentially multiple Docker build
and Docker run commands with just one configuration file
and then a set of orchestration commands
to start all those services, all these containers at once
and build all necessary images, if it should be required
and you then also can use one command to stop everything

-> docker compose will not replace docker files for custom images.docker compose does not replace docker images and containers.docker compose does not suit for multiple containers running on different hosts(machines). Containers should be on same host.

-> using docker compose we can describe multicontainer environment.

we specify a version here
and with version I don't mean a version
for our app or our file here.
Instead, we specify the version
of the docker compose specification we wanna use
and the version we defined here
has an effect on the features
we can use in this compose file
because the docker compose specification
and therefore the Syntax we have
to use in this docker compose file
is under active development
and may change over time.
So you can lock in a certain version up here
so the docker knows which features are
and are not available.
And therefore it's able
to execute your docker compose file correctly.
Now you can visit docs.docker.com/compose/composed-file.
And you will find a complete reference
of all the configuration options
you can set up in a composed file,
and you also find the different
compose specification versions,
which are available.
And obviously that will change over time.

-> .yaml or .yml :
1) version: “3.8”
( In a Docker Compose file, the `version` field specifies the version of the Docker Compose file format that you are using. It helps define the syntax and features available in the Compose file. The version is specified at the beginning of the Compose file and is required for the file to be interpreted correctly by Docker.)

2) services:
frontend:
build : ./frontend
ports :
-”3000:3000”
stdin_open: true ( to let know docker that this service needs open connection input)
tty : true ( the TTY option,which we should also set to true because in the end the -it flag is just a combination of the dash input flag for opening standard input. And the TTY flag for attaching this terminal, so to say.)

backend:
build : ./backend ( specifying path where docker file for backend is present to build image)
ports:
-‘80:80'
volumes:
- logs:/app/logs
- ./backend:/app (bind mount in normal dockerfile need absolute path.here it requires relativepath)
depends_on:
- mongodb (backend should work when mongodb is up and running)
container_name: mongodb
mongodb:
image: ‘mongo’ (image from local or custom image or from repository)
volume:
- data:/data/db : ro (ro means readonly.we can specify as many volumes as we want by using ‘-’)
environment: (or)
MONGO_INITDB_ROOT_USERNAME: max (or)
- MONGO_INITDB_ROOT_USERNAME=max
env_file:
- ./env/mongo.env
volumes:
data:
Services have child elements.which are nothing but containers.the number of containers we want we have to define under services.no need to specify “- - rm” bcz by default when services are down, containers will be removed automatically.and no need to specify ‘- d’.no need to specify network.by default docker compose creates a new network for all containers.

We need to specify named volumes used above under volumes at the end.anonymous and bind mounts not needed.if you then use the same volume name in different services, the volume will be shared so different containers can use the same volume, the same folder on your hosting machine. That is something which is possible. So named volumes should be specified.

- > docker compose up . to start all services defined in compose file.
-> docker-compose down to stop and remove all services.
-> docker-compose up -d to start in detached mode.
-> docker-compose down -v to delete volumes as well.
-> the service names are the names you can use in your code to send requests and to leverage the network Docker Compose creates for you
-> docker-compose build to build image when we change anything in dockerfile.if we dont build then it uses previous image.

-> Yes, with the configuration you provided in your `docker-compose.yml` file, accessing the application in your web browser through `localhost:8000` is absolutely correct.In your `docker-compose.yml`, you have mapped the port 8000 from the container to the host machine using the `ports` directive:

**ports:**
**- 8000:8000**

This means that any traffic coming to the host machine's port 8000 will be forwarded to the container's port 8000. By running the Django development server with the command:

**command: python3 manage.py runserver 0.0.0.0:8000**

you ensure that the Django server is bound to all available network interfaces within the container, making it accessible from the host machine.So, when you access `localhost:8000` in your web browser, the request is forwarded to the Django application running inside the container, and you should see your Django app in the browser just as if it were running directly on your host machine.Everything seems to be set up correctly, and you're accessing the application correctly through `localhost:8000`.

# SECTION-7

-> Utility containers:
they don't start an application when you run them, but instead you run them in conjunction with some command specified by you to then execute a certain task.with utility containers we can write project in container.instead of creating project at beginning and making it available in container, we can write project in container.

-> The Docker exec command allows you to execute certain commands inside of a running container besides the default command, this container executes.
-> command:
docker exec -it container_name npm init
docker exec -it node npm init
-> process:
1) write a docker file.
FROM node: 14-alpine
WORKDIR /app

the entrypoint instruction.
It is quite similar to the command instruction,
but it has one key difference.
If we add a command after the image name on docker run,
then this command, in this case npm install
overwrites the command specified in a Dockerfile.
If there is one.
We didn't have one thus far.
But if we had one, this command in the Dockerfile
would be overwritten by this command
after the image name on docker run.
With the entrypoint that's different.
For the entrypoint, whatever you enter here
after your image name on docker run
is appended after the entry point.
-> with CMD command when we run in terminal it overrides CMD command.but it is not happened with ENTRYPOINT.with ENTRYPOINT it will be appended.

# SECTION-8 (Deploying docker containers)

-> install docker on remote machine(eg: via SSH).push ,pull images and run container on remote machine.

-> steps:
1) create and launch EC2 instance ,VPC (virtual private cloud) and security group.
2) configure security group to expose all required ports to www.
3) connect to instance( via SSH) install docker and run container.
4) build image in host machine and push it to docker hub.and pull that image on remote machine and run container.
   -> build image locally.
   -> create repository in docker hub.
   -> rename image with tag name present in docker hub. Ex: 19210553/docker_demo_deployment
   -> login to docker hub in terminal with “docker login “ command.
   -> docker push 19210553/docker_demo_deployment:latest.
5) on remote machine run docker container like
  * sudo docker run -d -p 8000:8000 19210553/docker_demo_deployment *

-> create EC2 and connect to terminal.there install docker with commands
Sudo ensures the command is executed as a root user / with sufficient permissions.
   * sudo yum update -y
   * sudo yum -y install docker
   * sudo service docker start
   * docker run

-> In EC2 there are inbound and outbound rules.Inbound rules means various machines from different ports trying to access the EC2.outbound rules means EC2 trying to access various other machines.outbound rules controls which traffic is allowed from the instance queue somewhere else
EX for outbound :
so Docker on this remote machine, first of all had to download that image from Docker Hub.And that worked without any issues because all outbound traffic is allowed,

-> we mention ports in security groups.we specify them to give access to some machines try to connect with some port numbers.

-> this approach have some disadvantages.when we build EC2 instance we need to maintain everything like traffic , security etc.If we have skills to manage everything then it’s fine to use this method.but most of the situations it’s not fine.

-> the solution is to use ECS (Amazon Elastic Container Service).creation ,management ,updating is handled automatically ,monitoring and scaling is simplified.

-> AWS ECS thinks in four categories
  * clusters, *containers, *tasks, *services.
-> when creating ECS there are some options like container name and image name.there we have to give image name as name given in docker hub.if we are using any other registry then full address need to be mentioned.
-> the configuration will say the blueprint for your application.Here, you can tell AWS how it should launch your container.So not how it should execute docker run,but how the server on which it runs this should be configured, you could say.

-> Now here, we define our service.And a service, in the end, now controls how this task,so this configured application and the container that belongs to it, should be executed.And it's here where you, for example, could add a load balancer, and it manages all the heavy lifting of redirecting the incoming requests, queue the up and running containers, and all of that behind the scenes. Now, I'm not going to use a load balancer here, but we will add one later in this module. To keep things simple for now, we'll start without one. So you could say every task is executed by a service. So you have one service per task. And these are just the terms in which AWS thinks.

-> when we change anything in code we have to again create image and push it to docker hub.how ECS is going to know updated image is
Go to clusters -> default -> tasks -> task definitions -> create new revision.
When create new task it automaticallypulls latest image.
Once task is created , from actions dropdown select update service

# SECTION-11

-> To learn Kubernetes visit “kubernetes.io”
-> Kubernetes is an open-source system for automating the deployment,scaling, and management of containerized applications.

-> why Kubernetes?
The problems:

1. Manual deployment of containers is hard to maintain, error-prone and annoying.
2. Containers might crash/go down and need to be replaced. When manually deploying on the ec2 instance, we have to manually monitor and replace them with new containers when they crash.
3. We need more container instances when there is more traffic.
4. Incoming traffic should distributed equally.

The solution :

1. Using ECS checks and automatically re-deploy when container crashes.
2. Autoscaling
4. Load balancer.

But disadvantage is that when we use AWS ECS we lock in that particular service.when we want to switch service provider we cannot use same configuration file and need to learn that service.
So we can use kubernetes.

-> What is kubernetes exactly:
we have a way of defining our deployments, our scaling of containers and how containers should be monitored and replaced if they fail. We have a way of defining all of that independent from the cloud service we're using.It can help us with automatic deployment,scaling ,loadbalancing and management.

We write configuration (i.e desired architecture number of running containers etc) we can actually pass that configuration with certain tools to any cloud provider or actually any machine owned by us which is configured correctly.If cloud provider does not support config file then we can install some kubernetes software.some providers require some configurations then that particular configurations can be written in same configuration file. When using different cloud provider that particular configuration need to be removed.

-> kubernetes is not alternative to AWS or azure.it is not a cloud service provider.It is an open source project.it is collection of software and collection of concepts.which can be together used with any cloud provider.It can be used with any provider.not restricted to only one provider.Its not

alternativeto docker.It works with docker container.Its not a paid service.So kubernetes is a collection of concepts and tools which helps us with deploying containers anywhere.

-> pod is a smallest unit that holds container.which is responsible for running this container.
Now this pod with the container inside of it, then itself, runs on a so-called worker node. So a worker node is the thing in the Kubernetes world which runs your containers in the end. And you can think of worker nodes as your machines,each worker node can have more than one pod.Kubernetes also needs a proxy which in the end just is another tool Kubernetes sets up for you in the end on such a worker node to control the network traffic of the pods on that worker node. So basically to control whether these pods can reach the internet.kubernetes provides autoscaling by distributing pods to available worker nodes.master node interacts with worker nde and controls them.master node controls your deployment(i.e all worker nodes).

-> worker nodes and master node form a cluster and therefore one network.