# Docker

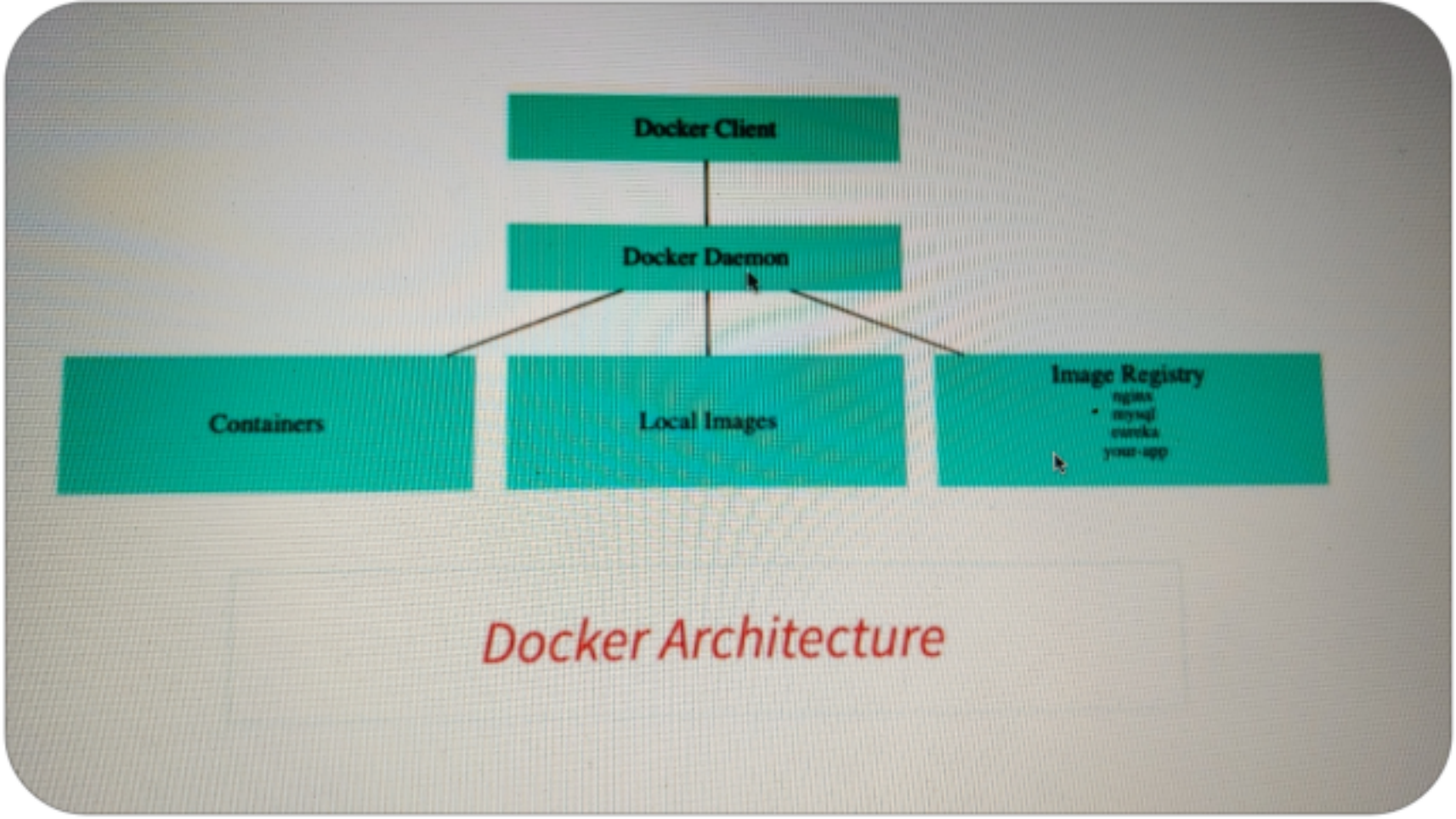
- Its an open platform for developers and system admin to build, ship and run distributed applications. Its a container management tool.
- It helps to run the same application without any issues on local developer machine, data center VMs and cloud, by using same deployment environment and same deployment procedure. When we install a docker in Windows or Mac we will get virtual linux os on top of existing os.
- **Traditional deployment**: Before docker, a development team needs to create one deployment document which contains information on required hardware

configurations, required OS, required softwares, application artifacts and provide it to deployment team. Now deployment team follow this document and try to deploy and run the application. This manual process was too complex and error prone. Docker helps in automating this process.

- **VM vs Docker**: A system has Host OS on top of which Hypervisor layer and on top of it many VM can be installed. Inside of each VM we install any guest OS with its own bins/libs. Finally we install the applications. In case of docker we don't have any hypervisor or guest OS layer. We just install docker engine on top of OS and run multiple docker images which runs seevices.

---

- **Docker architecture**: When we install docker it installs 2 piece of software in local machine. One is Docker client and other one ia docker server called docker daemon/engine. Docker daemon has local image container which has all locally created image or pulled image from docker registry. Docker daemon also has list of docker containers where each images are running. Also, docker daemon can either pull/ push image to remote docker registry called docker hub which can contains multiple docker repositories with multiple tags.



Docker Architecture

---

- **Dockerfile**: Its a text based document that contains all the command a usee could call on the command line to assemble an image. Docker can build images automatically by reading the instructions from this file.
- **Docker images and layers:** Docker images are the basis for containers. Any docker image always extends from a base image like ubuntu image. Each docker image references a list of read only layers. These layers are

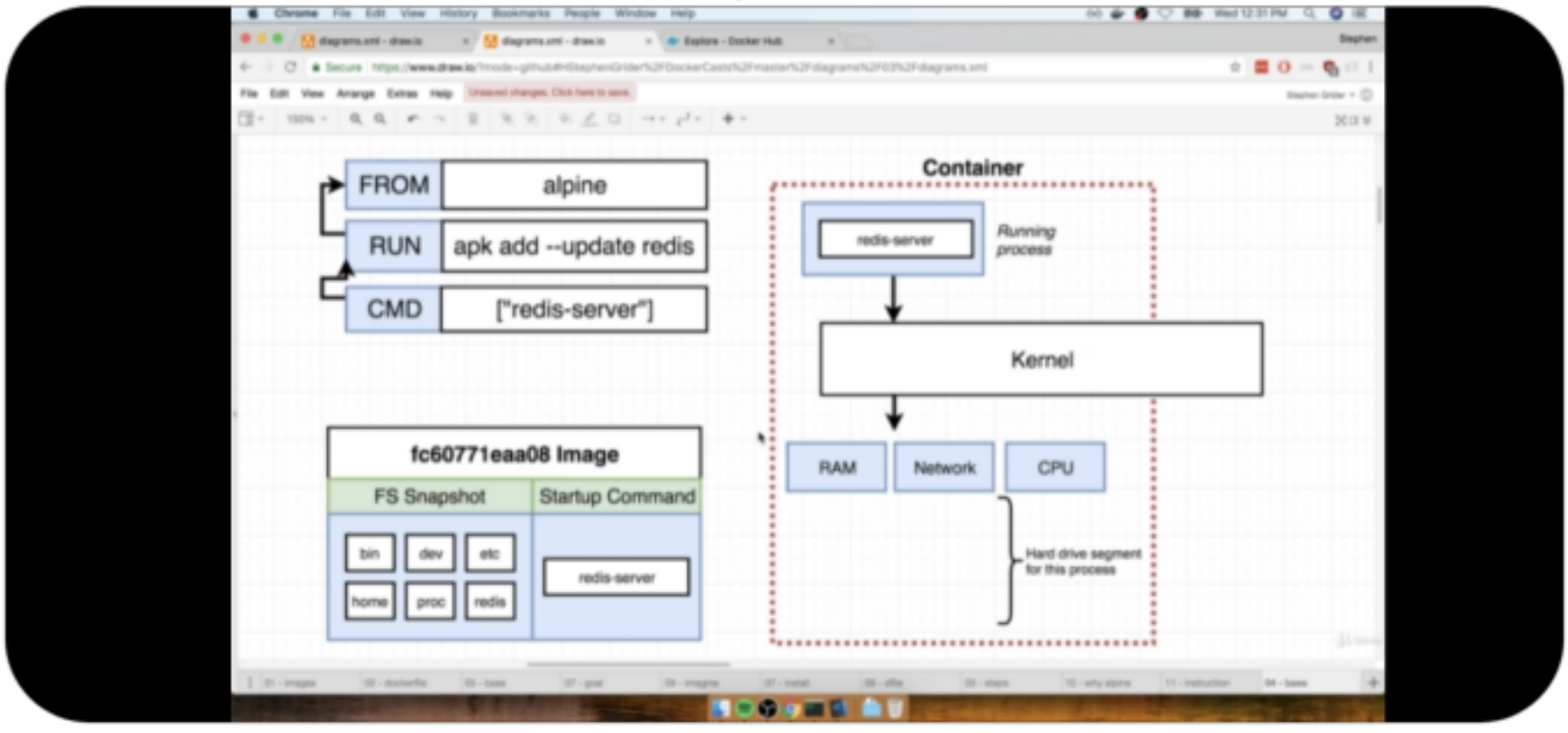
stacked together. When you create a new container from an image, new container layer will be get added on this image layer. This container layer can be read or writable by user. But the image layers are just read only. When a container is stopped all the changes made in container layer are lost. The underlying image layers remain intact when stopping a container. After modifying a container, you can use 'docker commit' to add the newly created thin R/W layer to the image.

- docker build command: The command to build an image from Dockerfile with a specific tag name for the image is 'docker build -t dockerId/ projectname:projectversion .' Run this command from the place

where we have the Dockerfile. Here -t means tag, dockerId means specific docker repository name, projectname will be the imageName and projectversion will be tag name.

- Life cycle of an image: build the image, run the image, download the image, cache the image, remove the image
- Docker Image Cache: Images always build from cache. If the same image already exists in the cache docker uses the same image to build new one. If we change the order or change anything in Dockerfile, image will rebuild from beginning. So always place changes as down as possible in Dockerfile. Images are built through multiple intermediate containers which will be removed once the final

image is ready.




- **Docker tag**: The same docker registry can hold different version of application through this tag. We can have same image name with different tag version related to each commit version of the application. Later we can provide the required tag version to run the image.

---

- **Docker container**: Docker containers are the running instances of a specific image. Run the image which will create and start docker container. Run =

create + start. Each containers will get isolated from one another and get its own CPU memory hard disk and networks. So each containers will have their own file system.

- docker run command: Once an image is built, container can be started from this image by running the command: 'docker run -d -p 5000:5000 --name=hello-workd in28min/hello-world-nodejs: 0.0.1.RELEASE'. Here we need to provide the repository name( I.e in28min/hello-world-nodejs)with its specific tag name. Also, we open the internal port 5000 to external port 5000 so the application can be accesses at port number 5000 If the image is not found in the local, docker daemon will pull this image from

docker hub and keep a copy of it in local and runs it. -p <host_port>:<container_port>

- Life cycle of a container: create a container, start a container, stop a container, remove a container

---

- **Docker network**: When we run a container, it is part of an internal docker network called bridge network. By default, all container run inside this bridge network. We can not access any of these container unless the port is exposed outside. Also, we can not access a container from another container through this bridge network.
- Accessing a container from another container: One option is to use docker link to connect micro services and another option to create custom network.

- Docker link: Start one container by providing a name like currency-exchange. Now while running another container currency-conversion through docker run command pass a new argument '--link' with value as previous container name which is 'currency-exchange'. Also, we need to set the required env variable value. For ex: '--env CURRENCY_EXVHANGE_SERVICE_HOST=http://currency-exchange.
- Docker custom Network: Create a new custom network, and then run both the services under this custom network, so that both services can communicate between each other. To create a new network use the command 'docker create network my-network'. Now while running both the service pass a new

argument '--network' with value as 'my-network'.

---

- **Docker container Data Management:** Containers are immutable. Once containers are deployed we can not change it, only we can re-deploy it. For ex: If we run nginx server of version 1.8 in a container, in order to upgrade it to 1.9 version we must stop the current container and start a new container with new image tag. If an application which is running inside a container produces some data, by default all these data are stored on a writable layer of container. This is called **persistent data problem**, because the data does not persist when we stop the container and it can be difficult to get the data out of container of any another process

needs it. Two options for resolving this issue to use Volumes or Bind Mounts.

- **Docker Volumes:** Volumes are the directory or part of the host file system where container will continuously persist its data. These Volumes are created and managed by container. Volumes can be created by Volume command in Dockerfile. So whenever we delete a container the corresponding volume will not be deleted from host machine and hence same volume can be used for the new container of the same image so that new container can be started from previous deleted container's state.
- We can define the Volume in Dockerfile as volumes:/var/lib/mysql. So this will create a new

folder mysql under the host machine path /var/lib. Now if we run a mysql image, the container will persists all of its data in mysql folder. So even after we stop this container the volume still be available. Note that since we dont mention any name for the volume here, container will define a random name for this volume.

- Another way to define a volume by providing our own custom name to it is, while running the docker run command pass new parameter named '--mount' with value as 'source=mysql-db,destination=/var/lib/mysql'. Now this command will create a new volume with name as 'mysql-db' and persistence location at '/var/lib/mysql'.

- Now while creating new container we can use the same Volume definition in Dockerfile or same --mount parameter in command.
- So docker will keep this mounted path even after stopping this container and make sure to provide the same mounted path to the same container once it is started again back. This will be helpful for production where it will **not allow to make any changes** to the created volumes.
- Example: docker run --name example -v /data/webapp:/webapp --t ubuntu:14.04. This command launches a Ubuntu container and mounts the directory /data/eyeball from host os into the container as /webapp. An application within the container can now use data from Host is that will be persisted even

when the container is stopped.

- **Docker Bind Mounts:** It means a file or directory on the host machine which is mounted into a container. Here we are mapping the host files/folders into a container files/hosts. By using this, we can continuously deploy nee code inside a running container without restarting it.
- For that we can change the files/ folder content in the host machine. Then, the container which will continuously looking for any changes in this host machine will also change accordingly by copying/removing files/folder contents. This will help for developer for testing.
- Good use cases are sharing configuration files, sharing source code or build artifacts etc
- Draw back of this is we can not

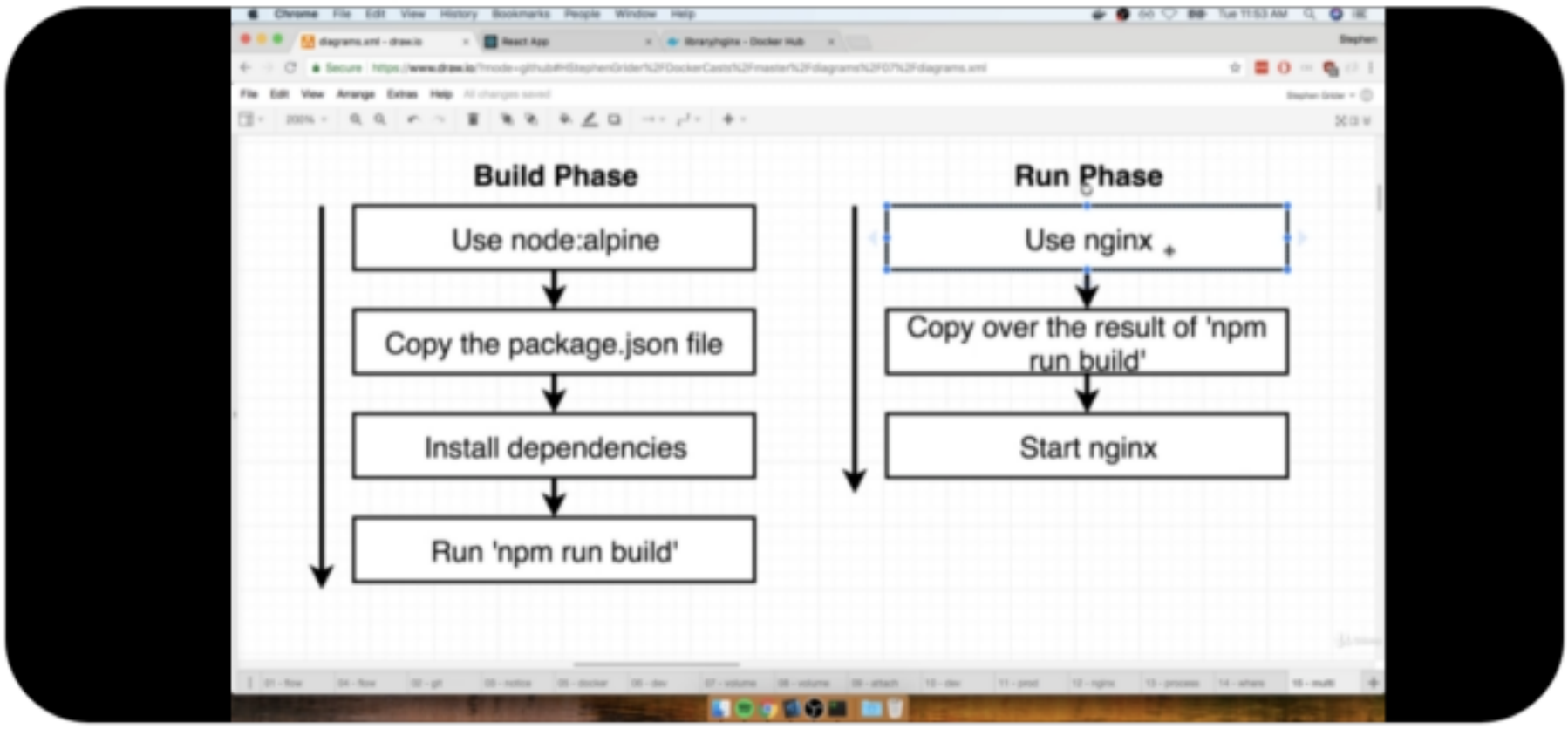
define Bind Mounts in Dockerfile as we can define Volumes in it.

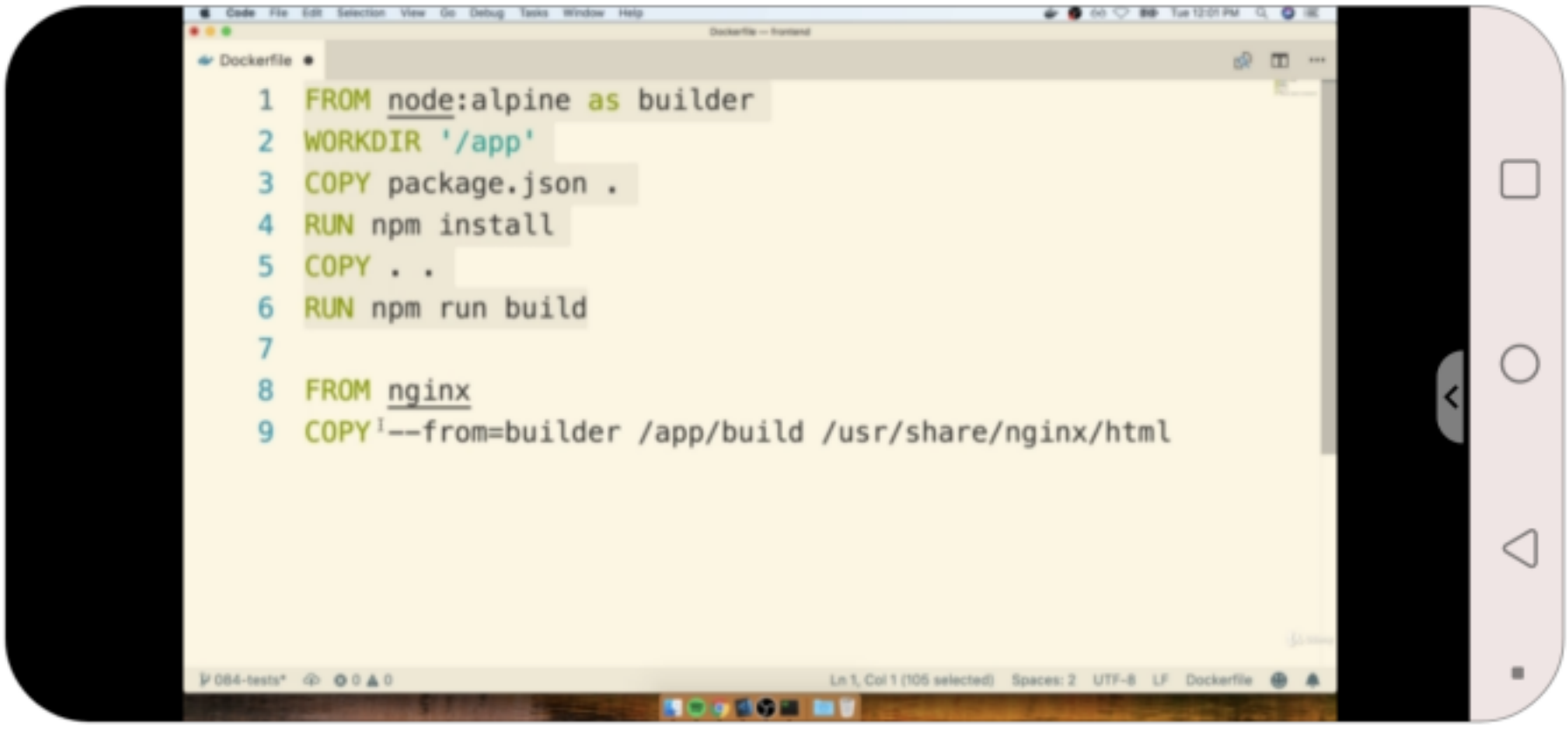
- So, this is used to map a folder of host machine to a folder in a container so container will holds a reference to this local folder instead of copying it inside container and whenever local folder content changes, container will identifies it immediately and react accordingly. Bind mounts will be helpful for development.
- Example: docker container run -d --name nginx --mount type=bind,source=$(pwd),target=/app nginx.

---

- **Implementing multi step build through Dockerfile:** We can have multi step build to run applications in production. Example: In build phase we can create the required final js files

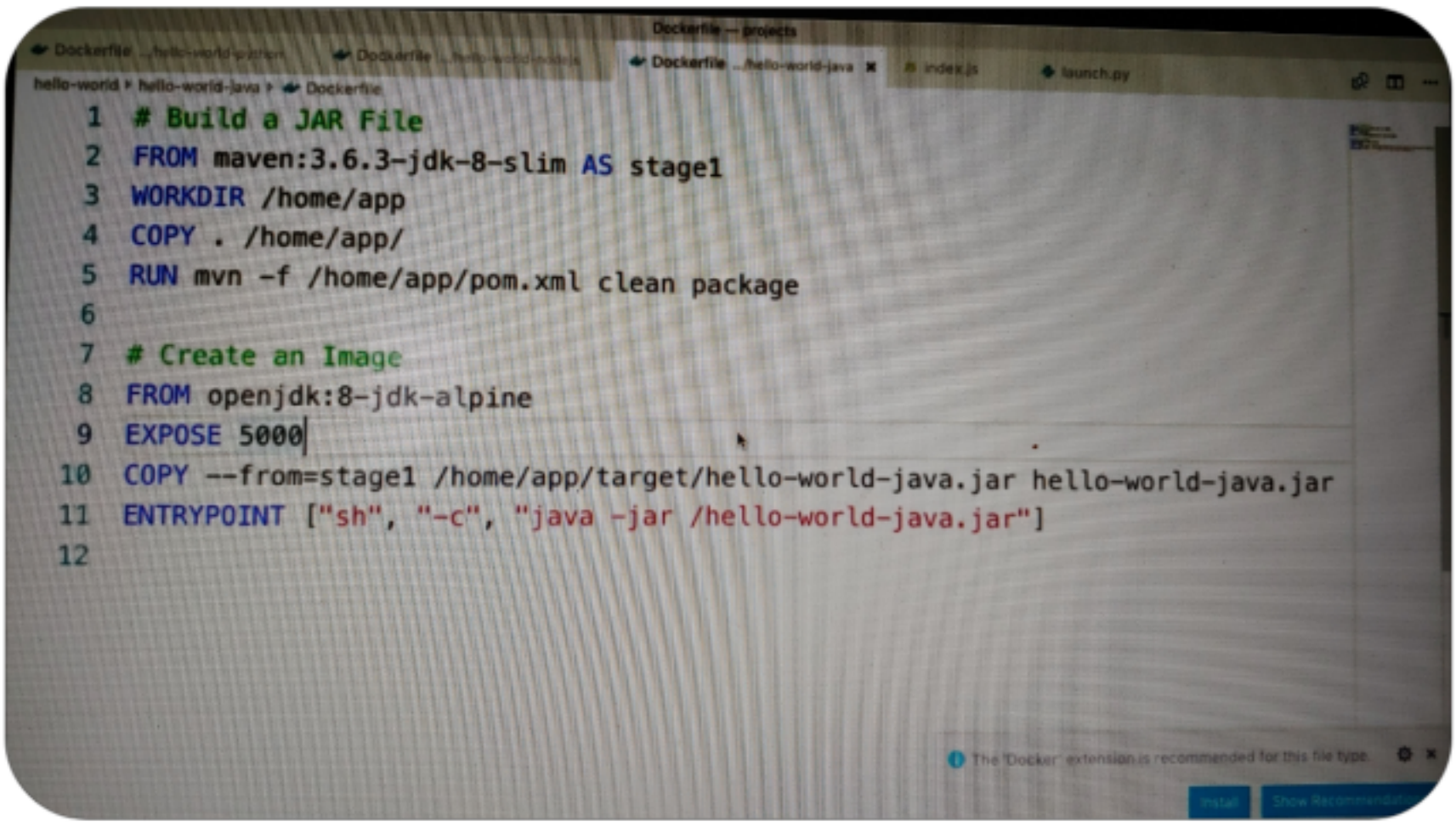
and then in next phase we can take these files and deploy them to nginx and run the server.




- **Running micro service as a docker container:** Same step we can use for java applications also. First to build the jar file and then create an image from it, so

that later we can run this image on jdk. If the jar application is built by using spring, then it runs the spring application on in a built tomcat server.

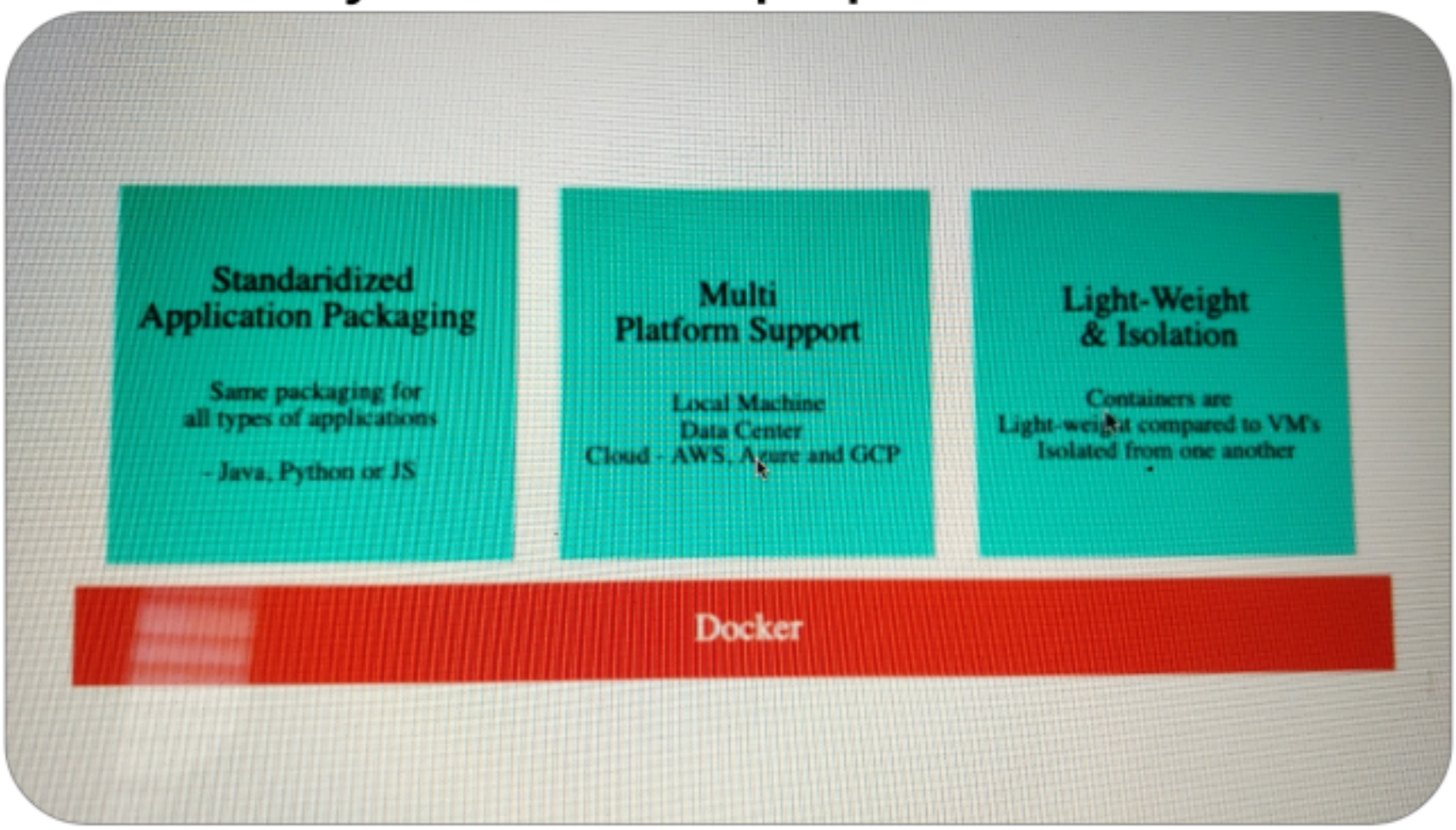



---

- CMD vs Entrypoint in DockerFile:
  If we use cmd option in Dockerfile then we can override this CMD value by passing new value as an arguments while running an image, but in entry point the argument of the EntryPoint option will be fixed as defined in the Dockerfile and which can not be

overridden while running image. If we want to pass dynamic value while running image use CMD option else use Entrypoint.

- Docker stop vs kill: stop gracefully shutdowns container, whereas kill shutdowns container immediately.
- Why docker is populer:




- Docker system prune vs docker container prune: system prune will delete all stopped containers, all unused networks, all unused images and build cache. Whereas container prune just deletes

stopped containers.

---

**Multiple docker commands:**

- To login to a running container: 'docker exec -it containerid/ containername /bin/bash or sh'
- To check container logs: 'docker logs -f container_id'
- To check container details: docker inspect <container_id>
- To create a new network: docker network create <network name>
- To list network: docker network ls
- To list images: docker images
- To list only running containers: docker container ls / docker ps
- To list all container including stopped containers: docker container ls -a / docker ps -all
- To stop a container: docker container stop container_id
- To remove a container: docker

container rm container_id

- ≡ ○ Pull an image: docker pull mysql
  If we don't specify a tag, then latest image will be pulled always.
- ≡ ○ Searching an image: docker search mysql
- ≡ ○ History of an image: docker image history <repository_name>:<tag_name>
- ≡ ○ More details of a image: docker image inspect <image_id>
- ≡ ○ To remove an image: docker image rm <image_id>
- ≡ ○ Pause/unpause a container: docker container pause/unpause <container_id>
- ≡ ○ To remove all stopped containers: docker container prune
- ≡ ○ To list volumes: docker volume ls
- ≡ ○ To inspect volume: docker inspect volume <volune_name>

- To check statistics at entire docker level: docker system df(disk space), docker system events, docker system info, docker system prune(remove unused data).
- To check CPU & memory utilization of a container: docker stats <container_id>

==========

## Docker Compose

- It is used to run multiple containers together as a single service. It provides relationship between multiple containers. For example we can start Mysql and Tomcat server with one YML file without starting each separately.
- **Steps involved in Docker Compose: First** define your

application's environment in a Dockerfile so it can be run anywhere. **Second**, to define all the required services that make up your application in a single file named docker-compose.yml so they all can be run together in an isolated environment. **Finally**, run a command 'docker-compose up -d' which will start entire app.

- docker-compose.yml file**:** In this file we will define list of services. For each service we provide container name, image information from which container needs to be started, mapping of host and container port details, restart policy in case container failed to start, and network name where this container should run. We can also define docker volumes, pass environment variable values if there are any,

mention depends_on option if the service depends on another service, new common network name which needs to be created etc.

```yaml
version: '3.7'
services:
  currency-exchange:
    image: in28min/currency-exchange:0.0.1-RELEASE
    ports:
      - "8000:8000"
    restart: always
    networks:
      - currency-compose-network

  currency-conversion:
    image: in28min/currency-conversion:0.0.1-RELEASE
    ports:
      - "8100:8100"
    restart: always
    environment:
      CURRENCY_EXCHANGE_SERVICE_HOST: http://currency-exchange
    depends_on:
      - currency-exchange
    networks:
      - currency-compose-network

# Networks to be created to facilitate communication between containers
networks:
  currency-compose-network:
```

- Restart Policy: If it is set as 'always' then docker will always restart the container automatically whenver it crashes
- links option in docker-compose.yml file: in this option we will mention list of those containers, such that if any of those container restarted, docker will restarts the current

container as well.

**docker-compose commands:**

- docker-compose up -d - to start all containers
- docker-compose -f xxx.yml up -d - If docker compose file name is something else than docker-compose.yml
- docker-compose down - to stop all containers
- dicker-compose events
- docker-compose config - to validate your yml file
- docker-compose images
- docker-compose ps
- dicker-compose stop

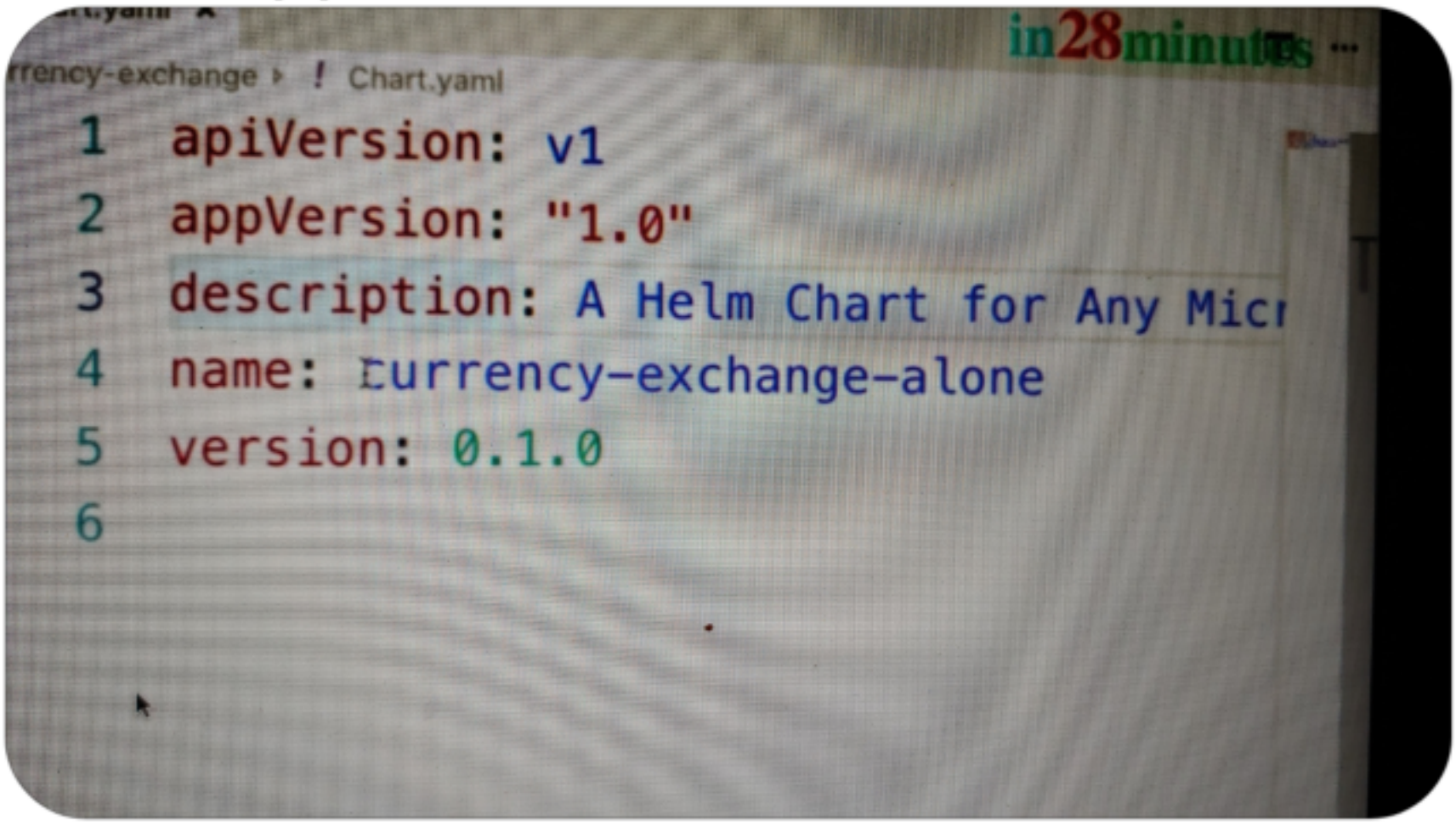
==========

## Helm

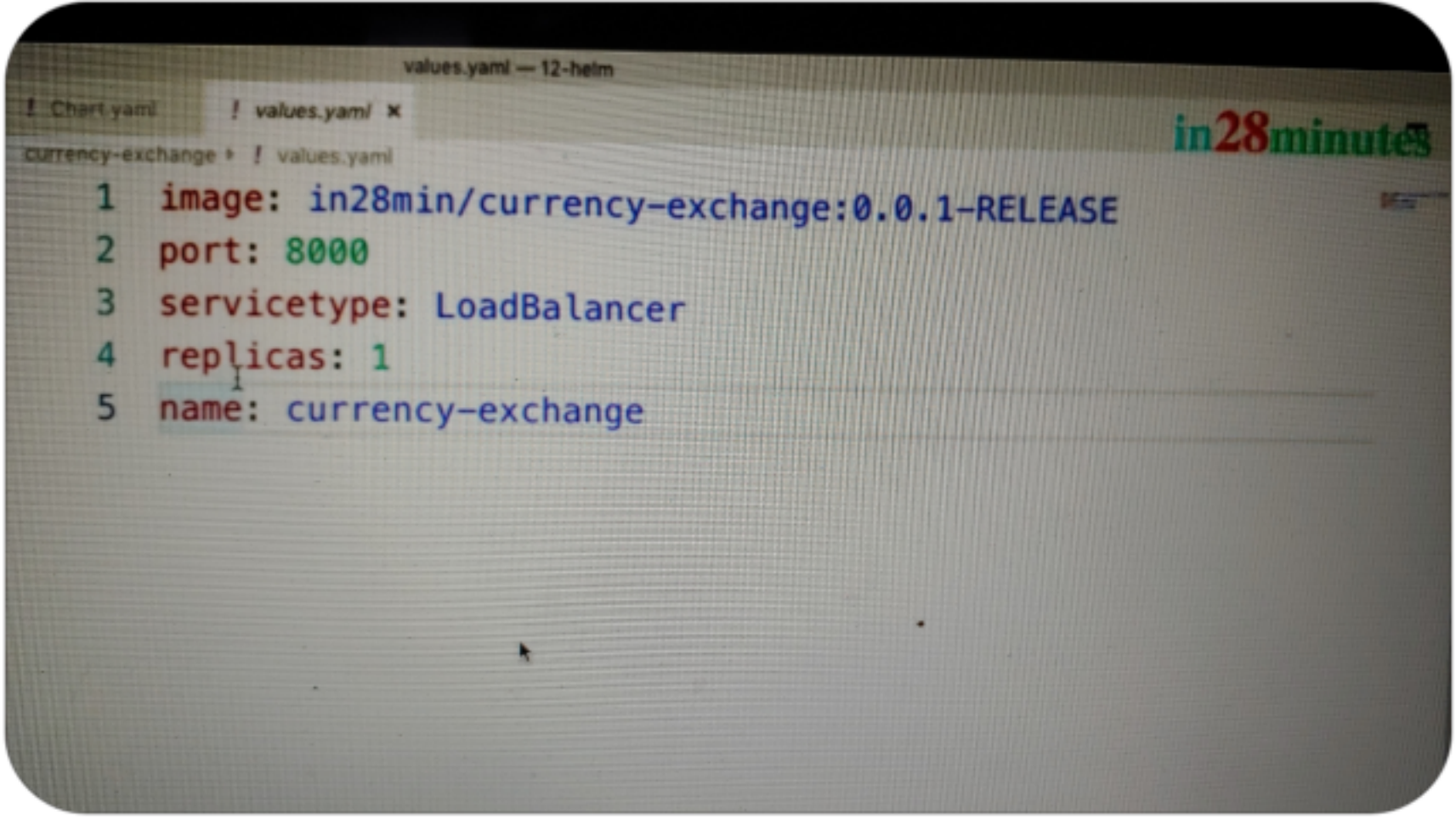
- Package manager for k8s
- For each micro service specific we can have helm project
- Helm has concepts of creating helm **chart**, publish it to helm **repository** and **release** which is a installation of a specific version of chart to any k8s cluster on cloud.
- Helm has a client side component called Helm Client and a server side component called Helm Tiller. We need to install Tiller on k8s cluster and client on to local machine. By default cloud shell also provides helm client which we can use.
- Create a new helm project by a command - 'helm create hello-world'. This will create a new helm project with the name 'hello-world' and by default creates Chart.yml and values.yml

files along with templates folder.

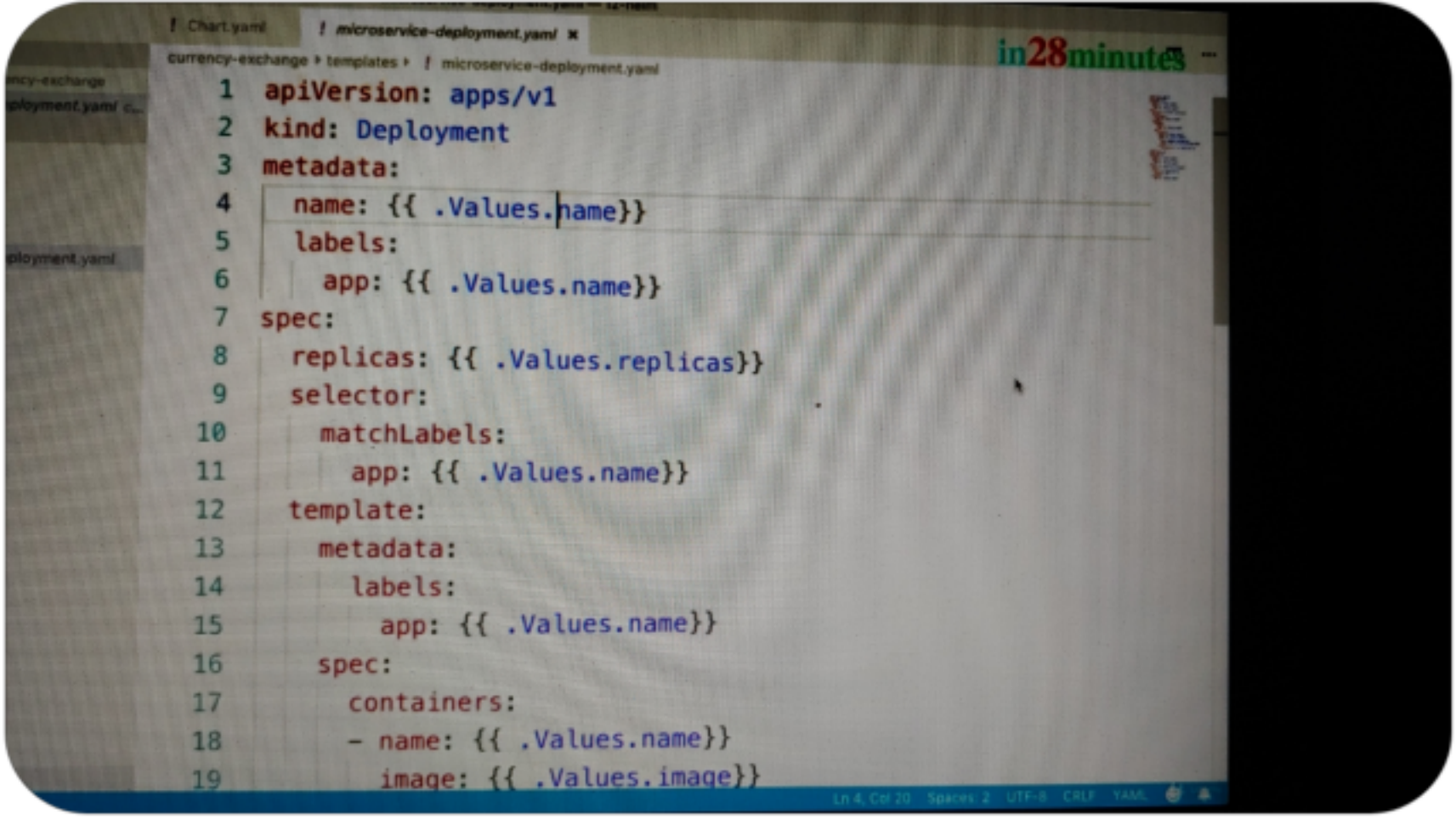
- **Helm chart.yml**: This file contains meta informations of the helm project like name, version, description, apiVersion and appVersion.




- **Helm Values.yml:** In this file we define set of variables along with their values which are used in our micro service applications.

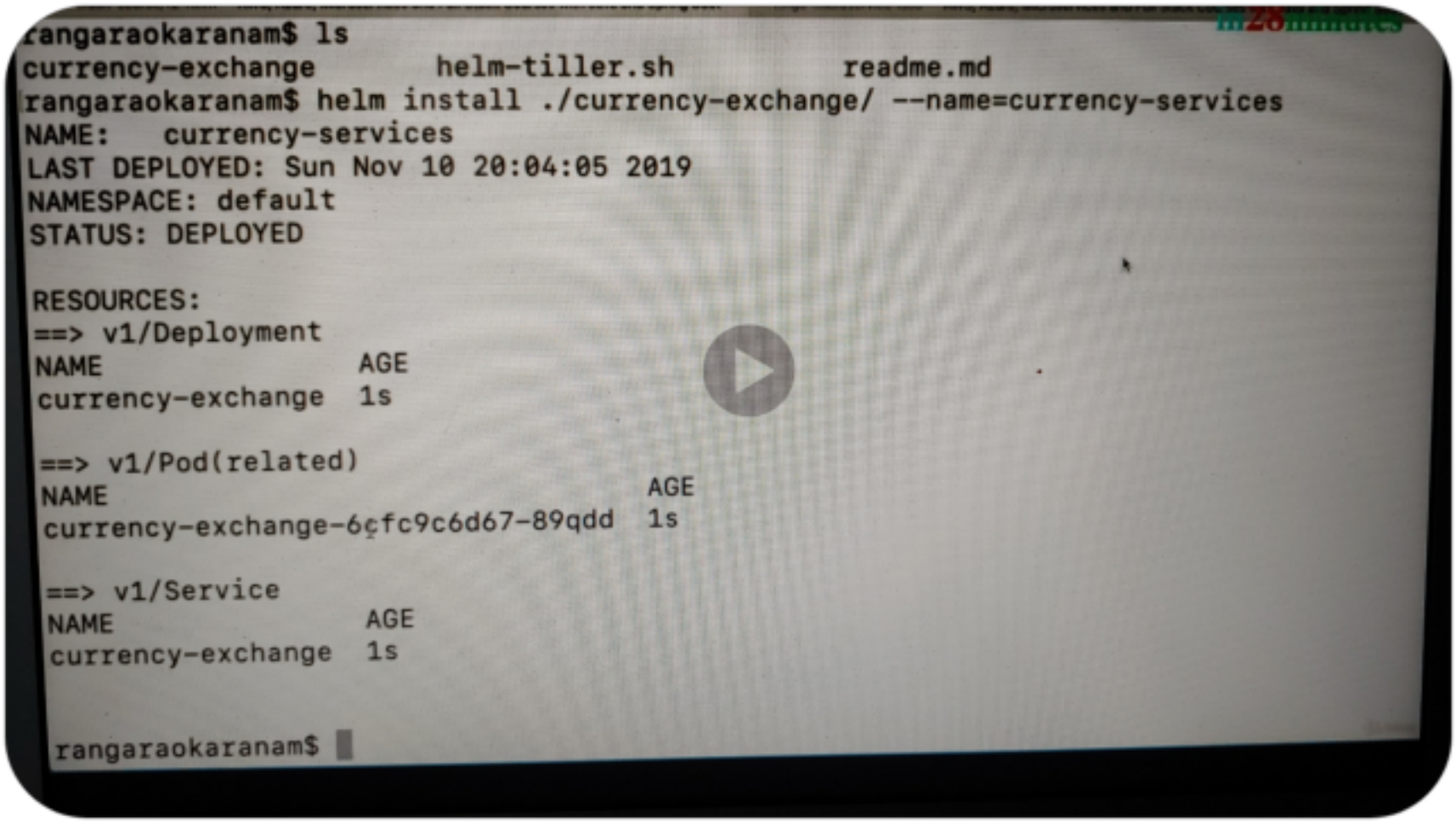



- **Helm template folder:** This folder contains all the k8s resource specific yml files like deployment, replica set, services, ingress etc. In these yml files we use variables which are defined in the Values.yml file. To refer the variable we use Go template syntax like {{ .Values.name}}




- Command 'helm lint <complete path of helm project root>' for testing the format of helm chart
- 'heml template <complete path of helm project root>' - generates all templates with variables actual values from values.yml without Tiller for quick feedback with o/p
- **To deploy a micro service to K8S cluster:** Type the command - 'helm install --name <release name> <full helm project path>. Ex: helm install ./hello-world/ --name=hello-world. This will creates all mentioned K8S

resources like deployment, pods, services, load balancer etc.




- **To deploy another micro service to K8S cluster:** Create a new helm project by helm create command and in the Values.yml file mention the image repository for this micro service from where helm can download and install. Ex: image: in28min/ currency'concmbersion: 0.0.1-RELEASE. Now run the same helm install command to deploy this to K8S cluster.
- We can also run the helm install

in debug mode by appending '--debug --dry-run', which will output what will be installed.

- ≡ ○ Use command 'helm history <helm release name>' to show history of this release.
- ≡ ○ **Helm upgrading a release:** Make a change in Values.yml for example we can change replicaSet value from 1 to 2. Now in order to release this change we can use the command: 'helm upgrade <release name> <chart full path>'. Ex: 'helm upgrade currency-service-1 ./ currency-conversion/'. Now this will create another new pod for the additional replicaSet. Now 'helm history' will show 2 output for Revision 1 and 2 now.
- ≡ ○ helm ls --all - lists current latest release
- ≡ ○ **Helm rollback to previous**

**release:** If we found a big in latest release and want to deploy previously working release then we can type the command 'helm rollback <release-name> <release-version>'. Ex: 'helm rollback currency-services-1 1'. Now there will be only one pod for the previous replica set 1.

- We can also create multiple K8S resource yml files under templates folder and deploy each of them in one helm install command.
- **Helm delete a release**: type a command 'helm delete --purge <release name>'

==========

## Istio

- Service mesh which can provide

multiple common functionality

- ≡ ○ Run as a separate container inside every pods to handle all these common functionalities, we calls it as sidecar pattern
- ≡ ○ Istio gateway and istio virtual service
- ≡ ○ Deployment strategies: rolling update, blue green with mirroring (only one version will be active, mirroring each request to another version for testing), canary deployment (imcreasingly moving from old service to new service percentage wise)

==========

## Vagrant

Tool to provision local development environment

==========

# Terraform

Resource management system or server provisioning system to support cloud infrastructure.
.tf files

- Provider with region and version
- Number of resource options
- There are plenty of providers in terraform
- Create a new Terraform configuration file main.tf where we define the provider details. After that we need to run 'terraform init' command which will download and install provider plugins to local system
- set up terraform env variables for aws keys and secret keys
- Terraform plan, apply,

--refresh=false, terraform fmt, validate, show

- ≡ ○ Resource for s3 buckets, resource for ec2 instance
- ≡ ○ TerraformTerraform output, variables, aws data providers for subnets.
- ≡ ○ Terraform destroy to delete all the resources
- ≡ ○ Terraform always creates immutable resources in the cloud
- ≡ ○ Terraform workspace for dev, qa, and other environment
- ≡ ○ Terraform mainly have 3 states: Desired, Known and Actual
- ≡ ○ Desired is the required changes, Known is the previois ran state which actually saved in terraform.tfstate file and Actual is the current changes done in instance through UI.
- ≡ ○ Standard way is to save the

terraform.tfstate and terraform.tfstate.backup files in the aws cloud SB3 bucket by using aws Dynamo DB to provide the access lock by multiple user. Then we can set up the local tertagorm to use this remote backend

- ○ All the hardcoded values can be removed and use it through env variables by using aws data providers.

==========

## AWS

- ○ IAM users
- ○ S3 Buckets (Simple Storage Service)
- ○ EC2 Instance - Virtual server in the cloud (Elastic Compute Cloud)

- 1. Software
- 2. Hardware
- 3. Networks - VPC(Virtual Private Cloud), Private and Public Networks. All of these networks will be created by aws by default with specific up addresses. For each region there will be single vpc network and for each zone there will be multiple subnets.
- 4. Security Groups - we need to define a security group for each of the resources. Usually this contaims ingress details for any protocols details (http,https,ssh), egress for any external ip address, vpc network etc
- 5. EC2 keypair required to access from terraform
- With all these above informatioms we can create EC2 instace in aws from terraform.
- We can deploy any server like

http in this ec2 instance through terrafor and write any message in index.html and access it through public up address.

- We can also deploy multiple http server instances in ec2
- We can install load balancer for these multiple instances through terraform

==========

## Jenkins

- Run jenkins locally in some docker container.
- Once it runs login to it through admin credentials and install required software like maven, graddle, docker etc.
- Then configure scm system by pointing it to the git repository where the codes are committed

by individual developers and points to some branch like master branch

- So whenver developer commits code jenkins build will start which basically checks the Jenkinsfile which present the home directory of the project and start running script one by one.
- **Jenkinsfile**:
- https://github.com/in28minutes/jenkin-devops-microservice/blob/master/Jenkinsfile
- In this Jenkinsfile basically we can run different stages one by one like- compile,code quality, unit test, integration test, package, image build and push image to docker hub.

==========

## Ansible

Configuration management tool.
Tool to install software in a server and configure them

- ansible.conf file: contains main configurations and user details
- ansible_hosts: contains details of all remote server addresses and custom group details
- Playbooks folder contains one or multiple playbooks
- Each playbook contains single or mutiple plays which has single or multiple tasks with their corresponding actions
- We can run playbooks against a single host or multiple hosts
- Can have ansible variables
- Ansible facts which contains entire deatils of the host or hosts
- Ansible conditional statements or loops

- Creating clear roadmap map

==========

## Learn efficiently

- Chunk the bigger subject into smaller and have specific learning timeline for the same
- 25 minutes of complete focus and 5 min of break. Can repeat the same for 2 or 3 times a day
- Spaced repetition of any subject for 1, 2, 4 and 8 days
- Learning actively instead of passive (practical instead of theory)
- Refer and learn from multiple resources
- Figure out 20% of the tasks which can complete 80% of the work

- ≡ ○ Be a part of learning community
- ≡ ○ Make a new habit: trigger, routine, reward and belief. Obvious, easy, attractive and satisfying. Dont break the change. Do it over and over
- ≡ ○ Take down note for every learning which can be refferred later for every repitation
- ≡ ○ Use visual memory to remember things by making a story, images, diagrams
- ≡ ○ Deep work. Full concentarion , define end time, sleep well, execise everyday, feel good always, easy starting seequnce, power of habit, use power of locci, visualize and sense, active learn, feel good
- ≡ ○ Domt try to become master, try to learn most important things
- ≡ ○ Test urself always after learning something