**Docker:**

**Dev: The application is working on my machine; something is wrong with your machine!**

**Qa : How dare you to blame me!**

**Here, dev machine may have different dependencies compared to testers, so while testing it won’t work properly as it is in the dev machine. To resolve this issue virtual machines (hypervisor) exists. Instead of sending only the code, we can send the image of OS along with the code. So, the tester can test the application in the environment with the same constraints & dependencies.**

**But, what if we are having multiple apps with different OS? It means that we are wasting resources. This is where container comes. All containers shares the same OS kernel. So, in our machine we install Docker but not hypervisor. In this docker we are having multiple containers, which runs multiple apps.**

* **Docker repositories - docker hub**
* **Docker engine**

**Container:**

* **baseImage is alpine (linux base image).**
* **topImage is application image.**
* **running environment of image**
* **5000 is the port binded (talk to application running inside a container)**
* **Virtual file system**

**Image:**

* **Images are files hosted in docker hub.**

**Container vs Image:**

**Image is like a class; Container is like instances.**

**i.e; We can have multiple containers running, which is like different versions of same application.**

**Before vs After Container:**

**Before: installation process is different on each OS, something could be gone wrong while installation.**

**After: own isolated environment, packaged with all needed configuration, one command to install app, can run same app with 2 different versions.**

**Application Deployement Before vs After:**

**Before: required jar files are shared to operations team along with textual guide of deployment to install on server.**

**Misunderstandings, back & forth communications, server configuration needed.**

**After: developers & operations work together to package the application, no environmental configuration is needed on server (except docker runtime)**

**Docker vs Virtual Machine:**

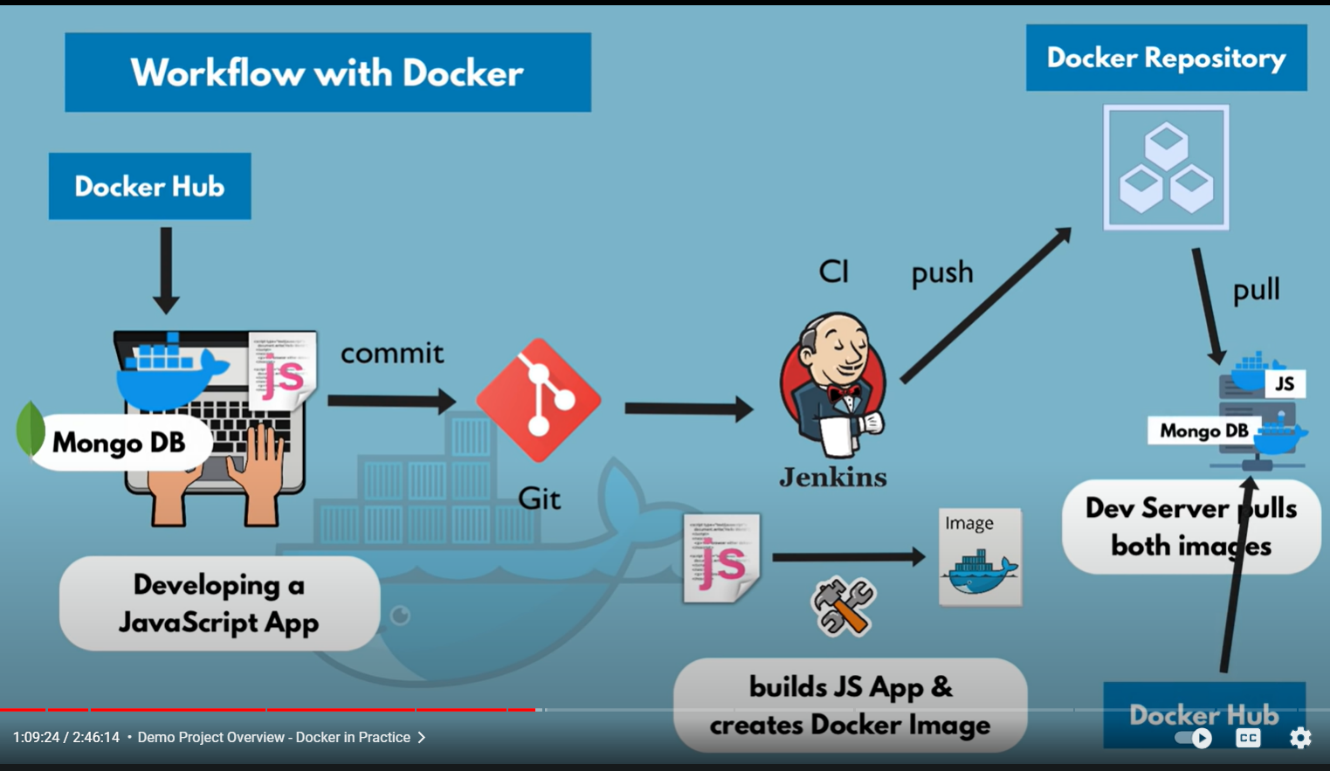
**Both are virtualization tools**

**Docker virtualises the application layer, which means OS kernel is not virtualised. So different applications uses same kernel, which is of host’s.**

**Virtual Machine virtualises both OS kernel & application layer, which means it virtualizes while OS.**

**Docker images are smaller in size.**

**Workflow with Docker:**

****

**Container Port vs Host Port:**

* **Multiple containers can run on our host machine**
* **Our laptop has only certain ports available**
* **Conflict arises when same port on host used by two/more containers**
* **We can solve this problem by port binding**

**Docker Architecture:**

**Docker engine has 3 sections.**

1. **Client : Docker CLI**
2. **REST API**
3. **Server : Docker daemon - performs clients operations**

**Docker Client CLI -> Docker Daemon -> Containerd (manages container lifecycle) -> runc (creates container)**

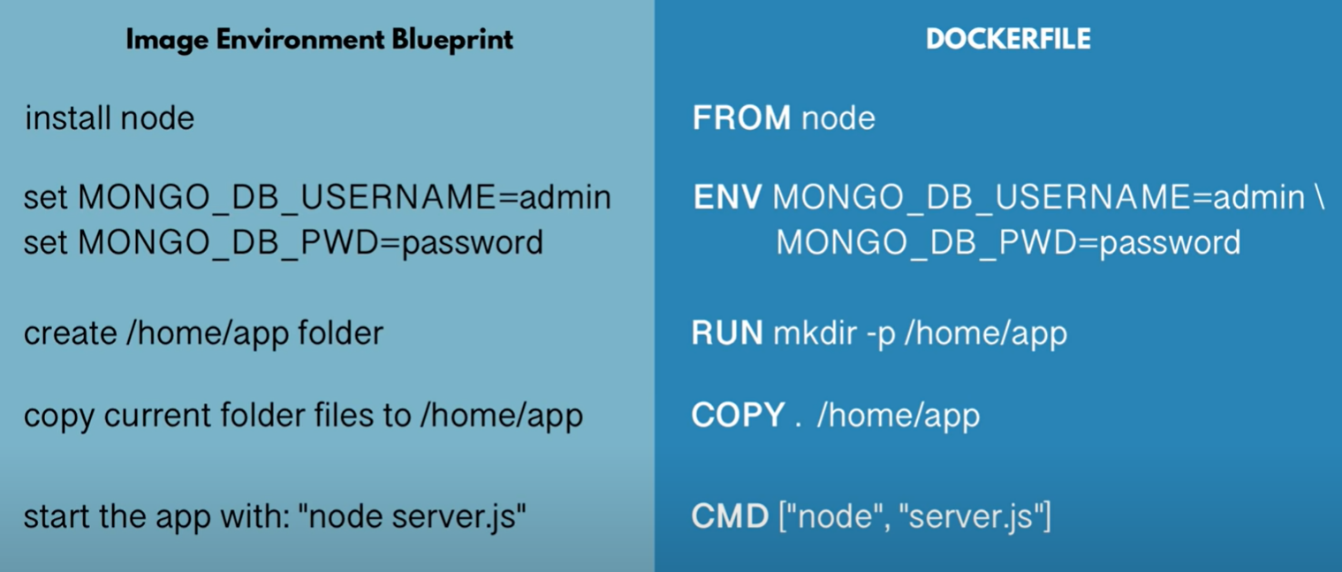
**Create Application on Docker:**

**There are 2 ways to create application on Docker.**

1. **Create project separately then dockerize/containerize.**
2. **Create project in container itself. (By downloading all required files in container)**

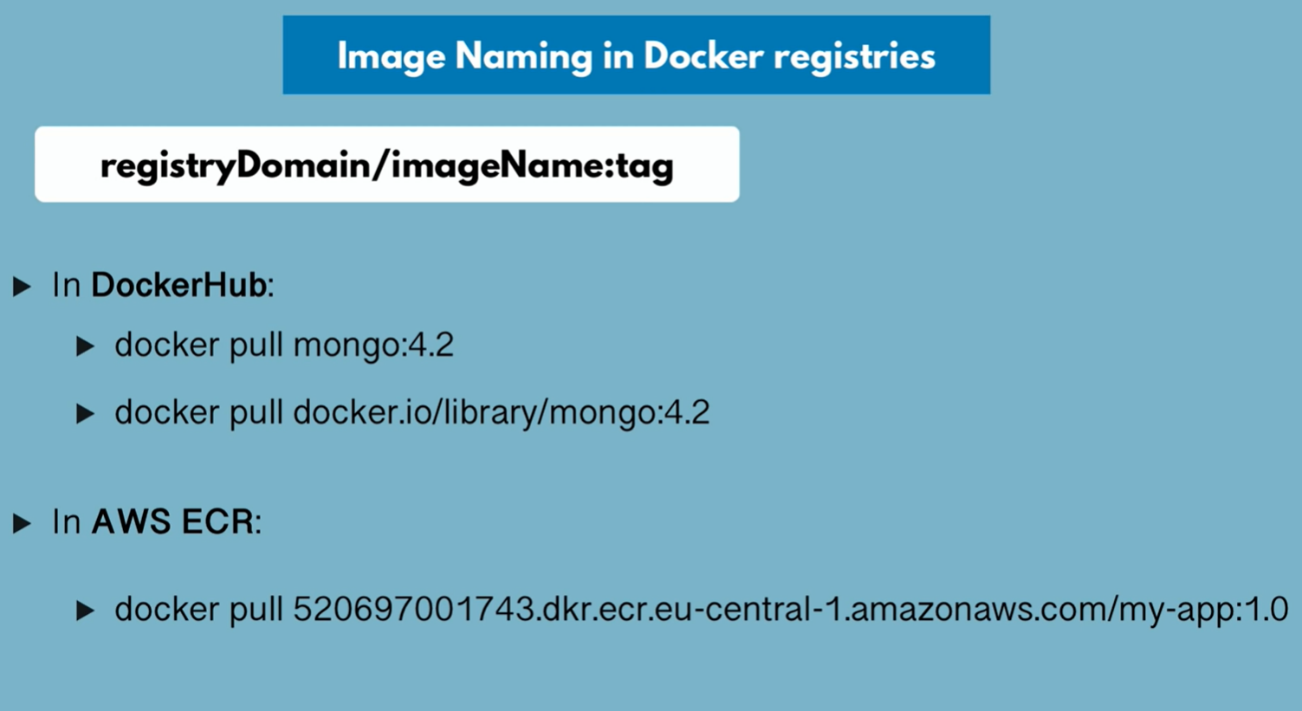
**Building our own Dokcer Image:**

**Docker file:**

* **It is bluprint for creating docker images**
* **It can be in 2 formats,**
  + **JSON (Dockerfile.json)**
  + **YAML (Dockerfile.yml)**
* **Whenever we adjust the docker file, we must rebuild the image**
* **Parts of docker file,**
* ****
  + **From - start by basing it on another image**
  + **ENV - to define environment variables**
  + **RUN - to execute any linux commands, the results will be happen inside the container**
  + **COPY - executes on host machine (. = source, /home/app = destination)**
  + **CMD - starts the app with file-name (starting point)**

**Private Docker Registry:**

* **Amazon ECR is one of the docker registry**
* **Login to docker before pushing the image**
  + **docker login -----**
* **Before login, AWS CLI needs to be installed & credentials must be configured**
* **Image naming in docker registries,**
  + **registryDomain/imageName:tag**

****

* **docker tag = rename the image**
* **docker tag <img-name> <img-name with address> - renames the image with address in repository**
* **docker push <img-name with address> - push the image to repository**
* **if any changes made to code, build the image again, change version (tag) & push it to repository (AWS ECR)**

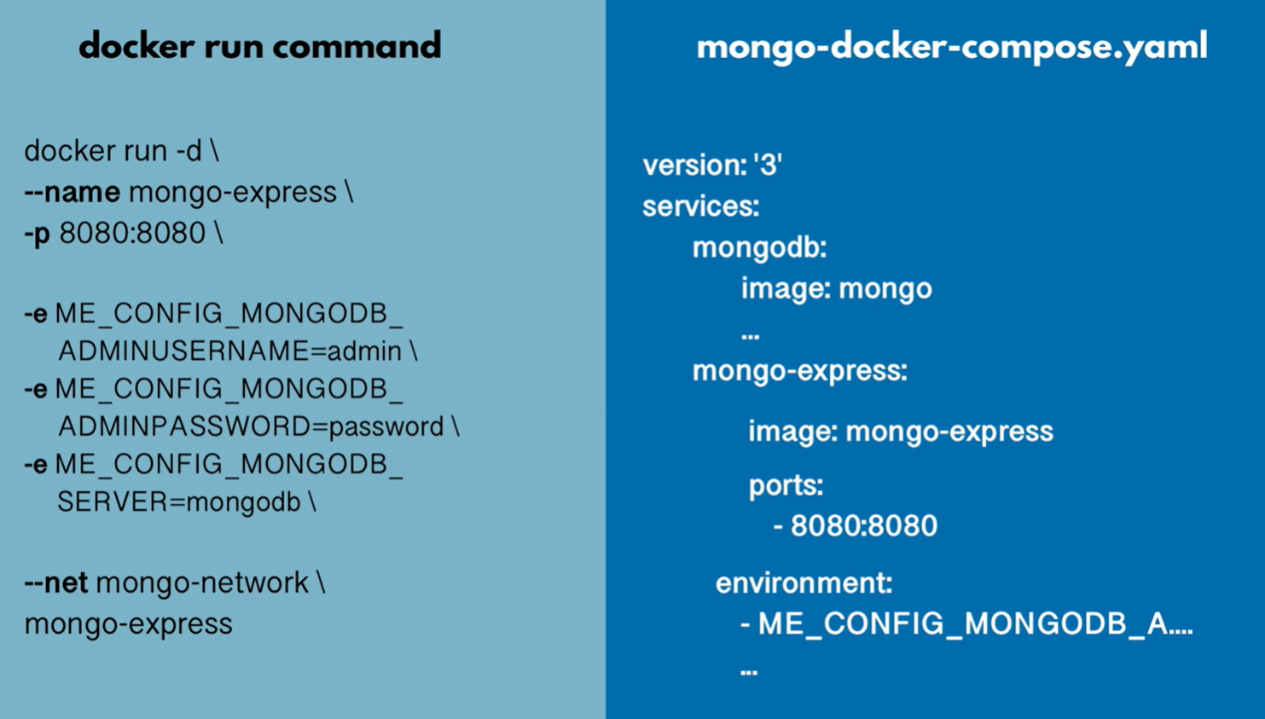
**Deploy containerized app:**

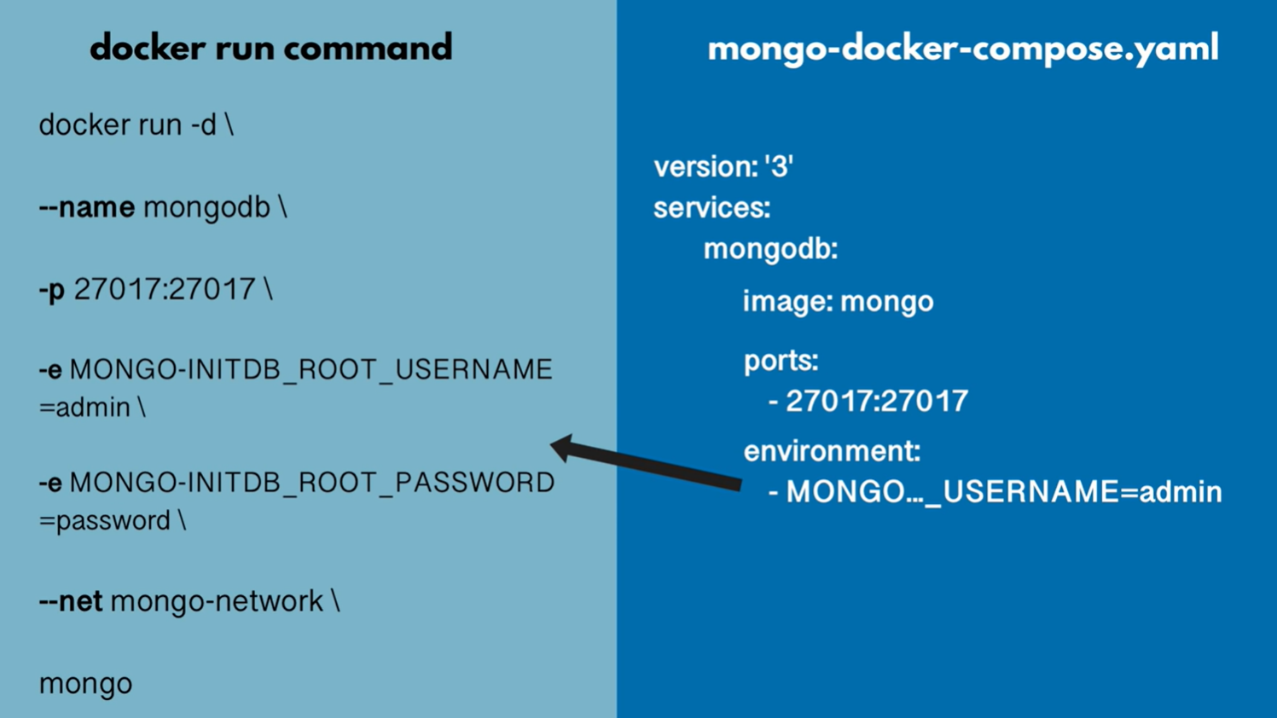
* **docker tag = rename the image**

**Docker Network:**

* **A Docker network is a virtual network that allows docker containers to communicate with each other, as well as with other network resources, such as the host machine or external services.**
* **Docker allows us to create user-defined networks.**
* **By leveraging docker networks, we can enable seamless communication between containers, facilitate microservices architectures**
* **When we create a docker container, it is assigned to a specific docker network. By default, docker creates 3 types of networks,**
  + **Bridge network:**
    - **It is default network created when docker is installed.**
    - **Containes on same bridge can communicate with each other using their IP addresses**
    - **We can expose specific ports on host machine to access services running inside the containers.**
  + **Host network:**
    - **When container is connected to host network, it shares the network namespace with host machine.**
    - **This means the container uses the same network interfaces as the host, making it easier to expose services without port mapping**
  + **None network:**
    - **Containers connected to the none network have no network connectivity. They can be used when you want to run a container without any network access**

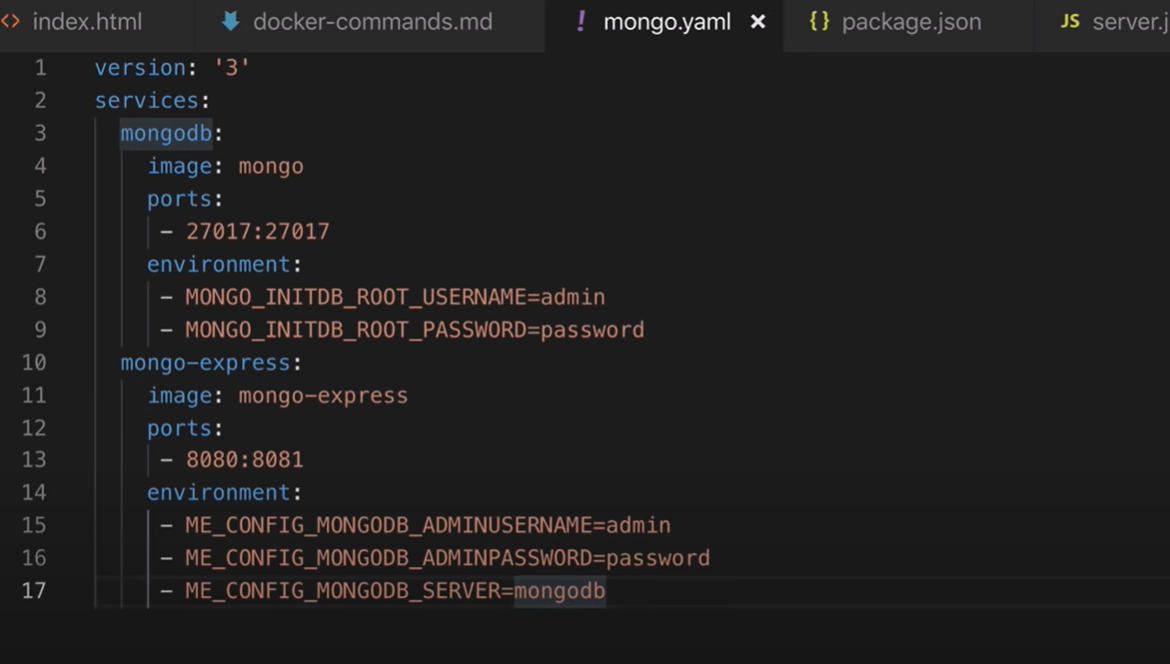
**Docker Compose:­­­­­**

* **It allows us to manage multi-container applications. (for running multiple docker containers)**
* **It enables us to define the services, network & volumes required for our application in a single YAML file called docker-compose.yml**
* **Using this we can define the entire application stack in a single configuration file**
* **Instead of separate docker run commands for each container, we can use a single docker-compose up to start all services defined in docker file**
* **Docker-compose down to stop & remove all containers, networks & volumes defined in docker file**
* **Components of docker-compose file,**
  + **Services:**
    - **Represesents a container in our application.**
    - **Here, we define the image to use, container configuration, env vars, ports to expose etc..**
  + **Networks:**
    - **We can define custom networks to control the communication between containers.**
    - **By default, docker compose creates a default network for our application**
  + **Volumes:**
    - **Used to persist data generated by containers, even after a container is removed/restarted data remains accessible.**
  + **Environment Variables:**
    - **Docker compose allows us to set env vars for each service**
    - **It is used to customize the behaviour of containers at runtime**
  + **Docker run command vs docker compose file:**

****

**Docker compose takes care of creating a common network, so no need to provide network info in docker compose file.**

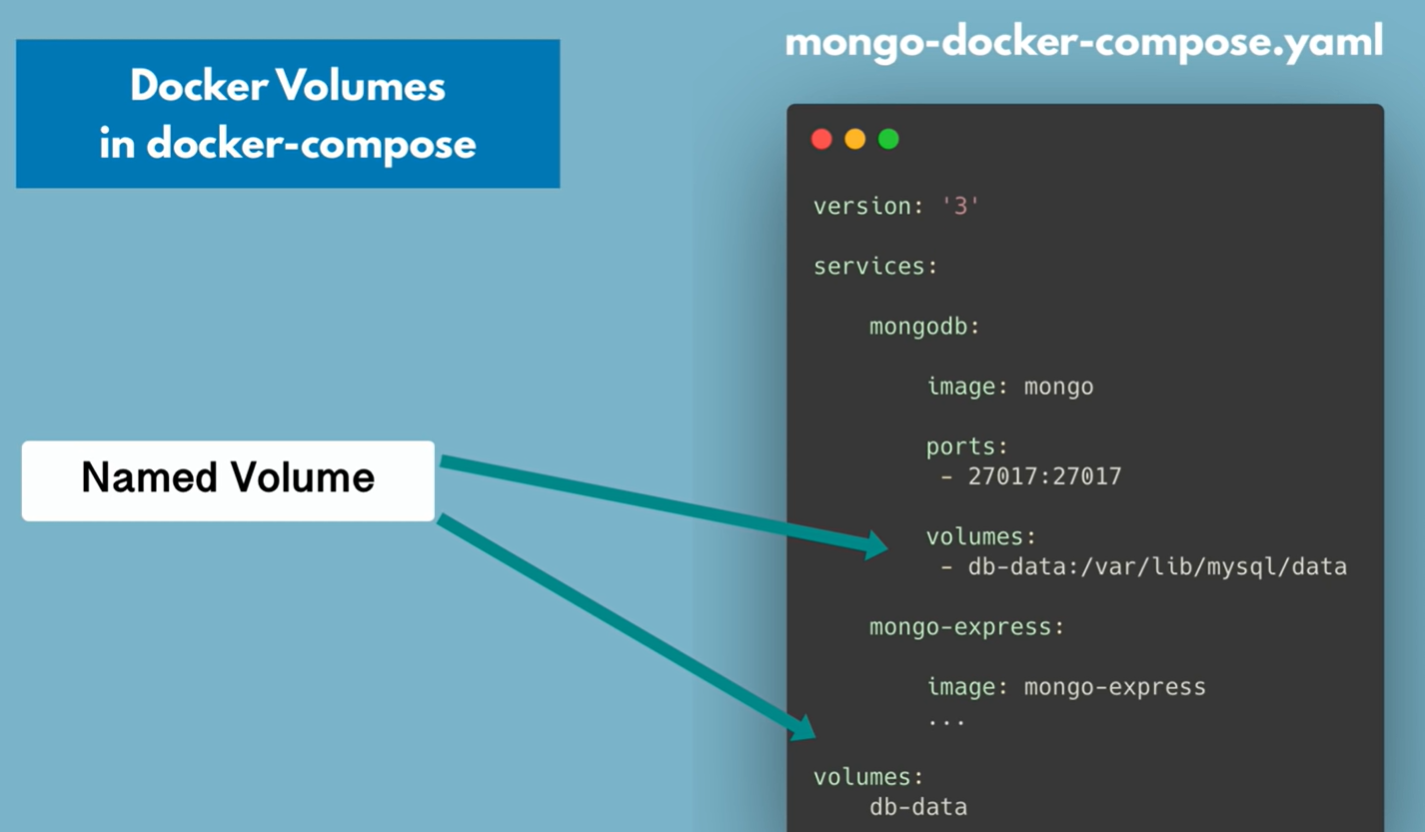
**Docker-compose file:**

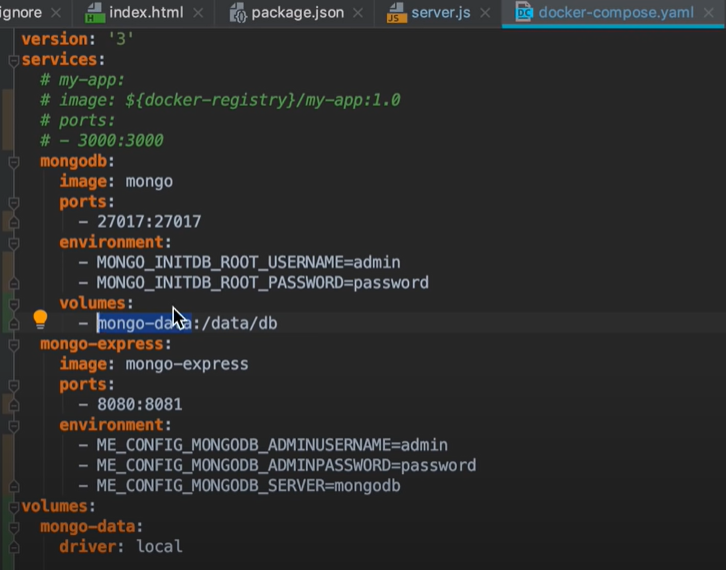
****

* + - **It is the configuration file based on which, container is created by docker engine.**
    - **To start docker compose from cmd,**
      * **docker-compose -f <file-name> up**

**Docker Volumes:­­­­­**

* **Used for data persistence in docker**
* **Data is gone when container is restarted/removed**
* **Date gets automatically replicated**
  + **Changes made locally reflected in virtual**
* **Volume types,**
  + **Host volumes - we decide where on the host file system the reference is made**
  + **Anonymous Volumes - for each container a folder is generated that gets mounted**
  + **Named volumes - we can reference the volume by name, should be used in production**

****

****

* **Docker volume locations,**
  + **Windows - c:/programData/docker/volumes**
  + **Linux - /var/lib/docker/volumes**
  + **Mac - /var/lib/docker/volumes**

**Commands:­­­­­**

**Basic docker commands:**

**docker pull redis - pulls image from docker hub**

**docker ps - lists running containers**

**docker ps -a - lists all containers (both running & stopped)**

**docker ps -a | grep <img-name> - lists all containers with img-name (both running & stopped)**

**docker image ls (docker images) - lists images on our machine**

**docker rmi <image-id> - removes image (before removing image remove container)**

**docker rm <container-id> - removes container**

**docker image pull ubuntu:latest - pulls latest ubuntu image from docker hub (repository)**

**docker run -it <ubuntu-id> /bin/bash - convert console into ubuntu terminal**

**docker run <image> - runs image (if image not exists, it pulls from dokcer hub)**

**docker run postgres:9.6 -**

**docker run -d redis - runs container in detached mode. (i.e; we can use same terminal again)**

**docker run -p6000:6379 - laptop port 6000 is binded with container port 6379**

**docker stop <container-id> - stops the container**

**docker start <container-id> - starts the stopped container**

**docker run –name < name> redis - starts container with custom name**

**Network commands:**

**docker network ls - lists all docker networks**

**docker network create <name> - creates network**

**docker run -d - runs docker image in network**

**-p 27017:27017**

**-e <env-var1>**

**-e <env-var2>**

**--name <custom-name>**

**--net <network-name>**

**<image-name>**

**Debugging of Containers commands:**

**docker logs <container-id> - fetch logs of container**

**docker logs <container-id> | tail - fetch last part of logs**

**docker logs <container-id> -f - used to stream the logs**

**docker logs <container-name> - fetch logs of container**

**docker exec -it <container-id> /bin/bash - gets us into redis terminal**

**docker exec -it <container-id> /bin/sh - gets us into redis terminal**

**Docker Compose commands:**

**docker-compose -f <file-name> up - to run docker compose file from cmd (creates containers & network)**

**docker-compose -f <file-name> down - to stop docker compose file from cmd (stops all docker containers & network)**

**Docker Image Build commands:**

**docker build -t img-name:1.0 . - to run docker compose file from cmd (creates containers & network)**

**-t - tag of image**

**Img-name:1.0 - name of our image with tag 1.0**

**. - location of docker file (. means file is in current directory)**

**Pushing docker image to private repository:**

**docker tag <img-name> <img-name with address> - renames the image with address in repository**

**docker push <img-name with address> - push the image to repository**

**Note:**

* **Using volumes, we can store data.**
* **For one image, we can create multiple containers.**
* **Docker file is configuration file based on which the container is created by docker engine.**
* **Docker compose provides lot of features like scheduling container, storing etc...**
* **Create docker network & start running containers in it**
* **Docker compose can be version controlled along with source code**
* **If container is restarted & there is no volumes used then data will be lost**
* **We can have multiple RUN commands in docker file**
* **Whenever we adjust the docker file, we must rebuild the image**