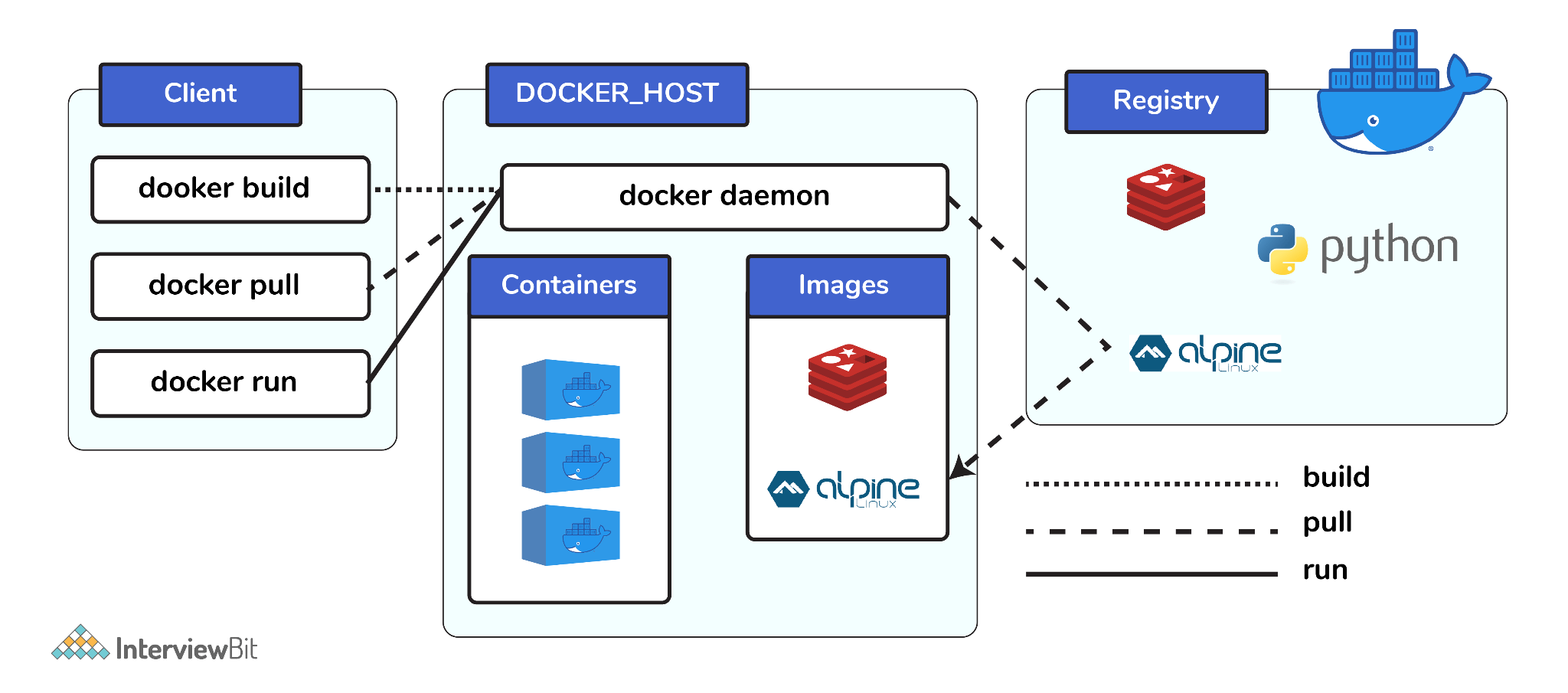
* **Reference video:** [Youtube](https://youtu.be/3c-iBn73dDE)
* **Why docker ?**
  + To make easy deployment of the applications once they are built up.
  + We need to take care of many things like installing packages, dependencies, environment setup on the machine where we have to deploy our application.
  + By using docker we can do these in a faster and more efficient way by making use of docker containers.
  + Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.
  + It also helps when we want to run our application in the same language but in different versions, at this stage we have to install dependencies based on the requirement of both the versions which may be not possible or may arise some conflicts, but it can be solved using dockers.
  + Docker will give an isolation in the form of containers to the application and their infrastructure (dependencies) needed to run it independently.
  + It also helps in running databases very easily just using one single command.
* **What is docker?**
  + Docker is a containerization platform which helps us in developing, shipping, and running applications in a faster way.
  + Docker is a containerization platform that combines all the dependencies of our applications in a package so that we can ship this package in any environment and run our application seamlessly.
  + Docker uses the OS kernel of the host machine where the docker container is running eg: our server.
  + Docker virtualizes the applications not the OS.
  + If the docker image is not compatible with our local machine OS versions then use docker toolbox to settle up the conflicts of the OS.
* **What do we mean by containerization ?**
  + Containerization is the packaging together of software code with all its necessary components like libraries, frameworks, and other dependencies so that they are isolated in their own "container".
* **Components of Docker Architecture ?**
  + *Docker Client:* This component executes build and run operations to communicate with the Docker Host.
  + *Docker Host:* This component holds the Docker Daemon, Docker images, and Docker containers. The daemon sets up a connection to the Docker Registry.
  + *Docker registry:* It is the place where we store the pre-built docker images which can be directly downloaded as and when needed. EG: docker Hub, Azure Container Registry.  
    
* **What is Dockerfile ?**
  + Dockerfile is a special file which contains the set of instructions, dependencies need to be installed, port to be exposed, environment variables setup, and other configurations.
  + It also contains the command to be executed to run the application in “CMD” or “ENTRYPOINT” section.
  + It defines the steps and configurations needed to create a reproducible and standardized environment for running a specific application or service. Example of Dockerfile
  + FROM python:3.10-alpine
  + WORKDIR /deployer
  + COPY . /deployer
  + RUN pip install -r /deployer/requirements.txt
  + ENTRYPOINT ["python3", "/deployer/main.py"]
* **Difference between CMD and ENTRYPOINT**
  + **ENTRYPOINT**
    - The ENTRYPOINT instruction allows you to configure a container that will run as an executable.
    - It sets the main command and parameters that will be executed when the container starts.
    - ENTRYPOINT ["echo", "Hello"]
    - When you run a container using this image (docker run <image>), it will output Hello.
  + **CMD**
    - The CMD instruction provides default arguments for the executable defined by ENTRYPOINT. It can be overridden by specifying command-line arguments when running the container.
    - If a Docker image has both CMD and ENTRYPOINT, the CMD arguments are appended to the ENTRYPOINT command when the container is run.
  + **Combining ENTRYPOINT and CMD** 
    - You can use both ENTRYPOINT and CMD in a Dockerfile. The CMD instruction provides default arguments for the ENTRYPOINT command.
    - ENTRYPOINT ["echo"]
    - CMD ["Hello", "world"]
    - When you run a container using this image (docker run <image>), it will output Hello world.
  + In summary, ENTRYPOINT is used to set the main command and parameters, and CMD is used to provide default arguments for that command.
* **What is Docker image ?**
  + A Docker image is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and system tools.
  + Image is the actual package that contains all the dependencies, environment, configurations, application code needed to run the image.
* **What is Docker Container ?**
  + Docker container is a running form of images.
  + The upper most image will be our own application running in the container.
  + Container gives the environment to the image to actually run.
* **Difference between normal and alpine images ?**
  + Primary difference is the size of the docker images.
  + In alpine images packer manager used is basic like **apk,** shell used in **ash** instead of **bash** etc.
  + So basically alpine images are less feature giving and flexible for developers in terms of shell n all but very good in terms of fast deployment, security and less image sizes.
  + Ideal for scenarios where a minimal and lightweight image is crucial, such as in microservices, container orchestration, and environments with resource constraints.
* **Can a container restart itself ?**
  + Yes, it is possible only while using certain docker-defined policies while using the docker run command. Following are the available policies:
  + **no:** The default behaviour is to not start containers automatically.
  + **on-failure:** Restart the container if it exited with a non-zero exit code or if the docker daemon restarts.
  + **unless-stopped:** Restart the container unless the container was in stopped state before the Docker daemon was stopped (explained later).
  + **always:** Always restart a stopped container unless the container was stopped explicitly.
  + These policies can be used as:
  + docker run -dit — restart [restart-policy-value] [container\_name]
* **Life cycle of Docker Container ?**
  + Docker containers go through the following stages:
    - Create a container
    - Run the container
    - Pause the container (optional)
    - Un-pause the container (optional)
    - Start the container
    - Stop the container
    - Restart the container
    - Kill the container
    - Destroy the container
* **How does docker networking work ?**
  + Docker provides several networking options, and the default is the bridge network.
  + **Bridge Network**: When you install Docker, it creates a default bridge network named bridge. Containers attached to this network can communicate with each other using container names as hostnames. Docker assigns IP addresses from the 172.17.0.0/16 address range by default, but you can customize this.
  + **Host Network**: You can also use the host network mode, where the container shares the network namespace with the host. In this mode, the container uses the host's network interfaces directly, so it doesn't get its own isolated network stack.
* How is the IP address assigned to the docker container ?
  + **Automatically Assigned IP**: When you create a container, Docker automatically assigns an IP address from the subnet associated with the network to the container.
  + **Container Naming**: Containers can communicate with each other using their names as hostnames. Docker automatically sets up the DNS resolution for containers within the same network.
  + **User-Defined Networks**: If you create a custom bridge network, you can specify the IP address range and manually assign IP addresses to containers.
* How does port mapping works in docker container ?
  + **Publishing Ports with -p option:**
    - docker run -p <host-port>:<container-port> my-web-app
    - docker run -p 8080:80 my-web-app
    - In this example, the container's port 80 is mapped to the host's port 8080.
  + **Exposing Ports with -e option:**
    - You can use the -e option to expose a port without publishing it to the host. This allows containers to communicate with each other on the same Docker network.
    - docker run --expose=8080 my-web-app
    - This doesn't make the port accessible from outside the Docker network, but it allows other containers on the same network to communicate with the container on port 8080.
  + **Dynamic Port Allocation with -P option:**
    - If you don't specify a particular port, you can use the -P option to dynamically allocate a port on the host machine.
    - docker run -P my-web-app
    - Docker will select a random available port on the host and map it to the container's exposed ports.

| **Flag value** | **Description** |
| --- | --- |
| -p 8080:80 | Map port 8080 on the Docker host to TCP port 80 in the container. |
| -p 192.168.1.100:8080:80 | Map port 8080 on the Docker host IP 192.168.1.100 to TCP port 80 in the container. |
| -p 8080:80/udp | Map port 8080 on the Docker host to UDP port 80 in the container. |
| -p 8080:80/tcp -p 8080:80/udp | Map TCP port 8080 on the Docker host to TCP port 80 in the container, and map UDP port 8080 on the Docker host to UDP port 80 in the container. |
| docker run -p 127.0.0.1:8080:80 nginx | If you include the localhost IP address (127.0.0.1) with the publish flag, only the Docker host can access the published container port. Outside world cannot access it. |

* **What is docker daemon ?**
  + Docker daemon is a background process that manages Docker containers on a system.
  + **Management of Containers:**
    - The Docker daemon is responsible for creating, running, stopping, and deleting containers.
    - It manages the entire container lifecycle, ensuring that containers are started and stopped as needed.
  + **Communication with Docker CLI:**
    - The Docker daemon listens for Docker commands from the Docker CLI (Command Line Interface) or other Docker clients.
    - When you run a Docker command, such as docker run or docker build, the Docker CLI communicates with the Docker daemon to execute the requested operation.
* **Difference between VM and container?**
  + VM virtualizes both OS and application, but container virtualizes the app running on it and container will use the OS enviorment of the host\_machine itself(host machine can also be one VM running on our laptop or running on the AWS cloud i.e EC2 instance).
* **Advantage of the containers:**
  + Let's say our application uses 2 diff types of mySQL versions. Then I can make one container running MYSQL V1 and another container running MYSQL V2.
* **Difference between container and image**
  + Container is the running environment for image
  + Means container will give the environment, file system needed to store some data, etc to the image(our application or predefined some image downloaded from the docker hub).
  + Container will bind one port to the image running on the container.
* **What do we mean by port mapping of containers?**
  + On 1 host machine/ VM more than 1 docker container can run parallely.
  + 2 containers can run on the same port in the same host machine.
  + Now question arises that how will the API call bifurcated between the

2 containers(basically 2 images running on different containers).

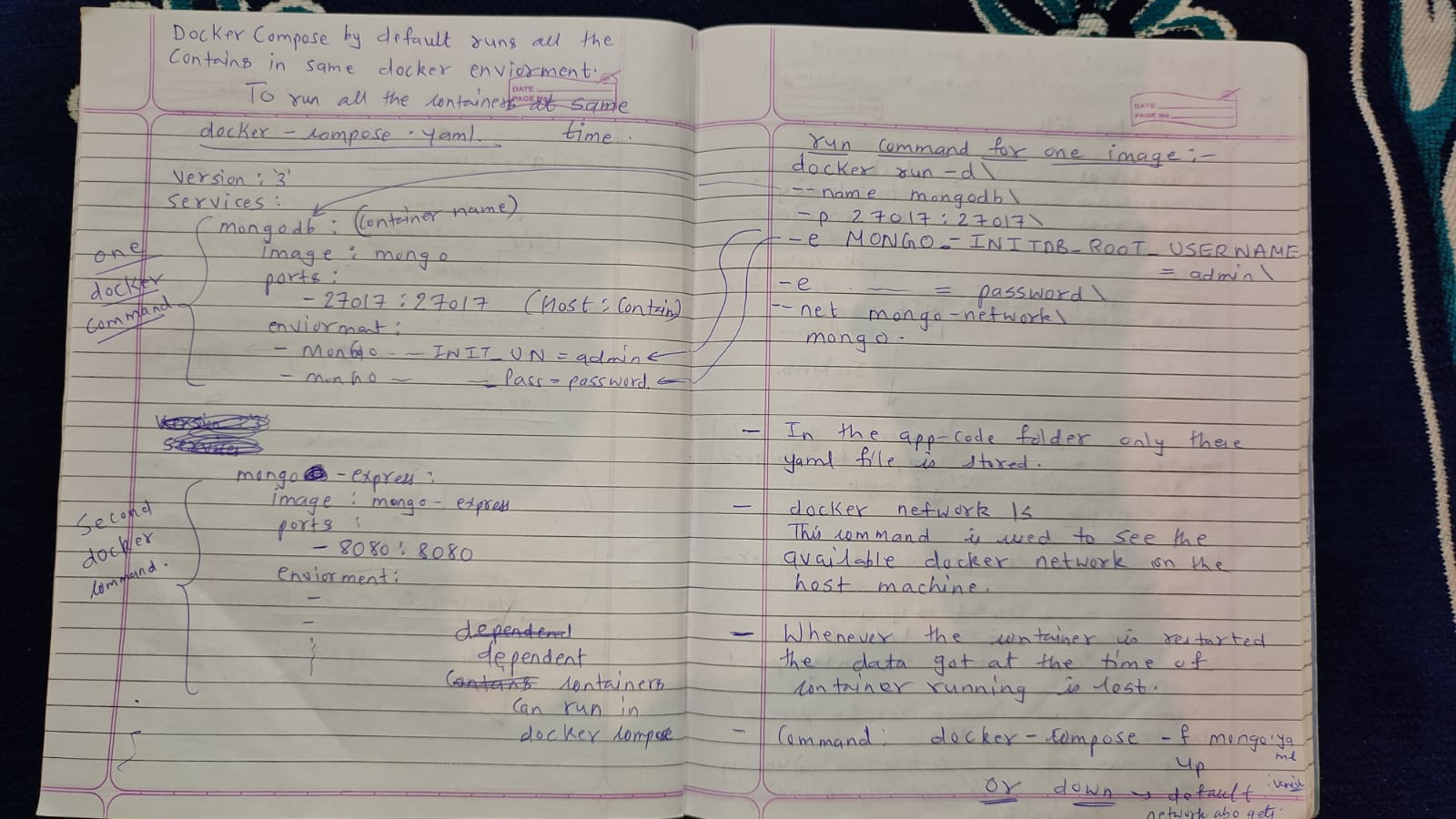
* + Now let's say there are 2 containers running on the same port 3000.
  + Here the concept of binding the host port to the container port comes in.
  + Whenever a container starts running on a VM/host machine, the machine will assign one port to that container(eg: 5000). Now, this 5000 port will be mapped with one of the ports of the container(eg:3000).
  + Now for the other container having the same port(eg: 3000) will be binded to another port number assigned by the machine(eg: 6000).
  + So, whenever the request comes it will come on the port of the host machine/ VM not directly to the container.
  + According to the mapping done between the machine port and container port(basically image is being binded to the host port) the request will be forwarded to that specific container(image-our sample app) by the host machine/ VM.
  + It means the API calls will be made on the port number of the host machine/ VM and not the actual container.
* **What is stored in the docker container?**
  + Docker uses storage drivers to store image layers, and to store data in the writable layer of a container.
  + The container's writable layer does not persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime.
* **General Points**
  + Whenever we are deploying the app the database used by that app will run on different containers.
  + It simply means that let's say we have one java-script app and that app uses mongodb as the data-base for the storage.
  + We will use the image from the docker hub of the mongo db directly. For our app code one separate custom docker image will be formed which will be stored into some private docker repository of the organisation.
  + Now when we want to run both the images, server/host\_machine will pull both the images mongo-db from the public docker hub and app-code image from the private docker repository and each image will be deployed on individual separate containers running parallely.
  + There will be some intermediator called jenkins which will make the docker image of our app-code based on the given artifacts of the app-code. (artifacts like the environment info, version info of the language used by the application, etc)

**DOCKER COMMANDS TABLE**

| **When to use** | **Command** |
| --- | --- |
| build image | docker build -t <image\_name>:<tag> <path\_to\_Dockerfile> |
| run image | docker run -p <host\_port>:<app\_port> <image\_name>:<tag> |
| list running containers | docker ps |
| list all containers | docker ps -a |
| start container | docker start <container\_name/id> |
| stop container | docker stop <container\_name/id> |
| remove container | docker rm <container\_name/id> |
| remove image | docker rmi <image\_name>:<tag> |
| remove all containers | docker rm -f $(docker ps -aq) |
| remove all images | docker rmi -f $(docker images -aq) |
| get container id | docker container ls –quiet –filter name=^<container\_name>$ |
| get all info of container | docker inspect <container\_name/id> |
| check container status | docker inspect -f ‘{{.State.Status}}’ <container\_name/id> |
| live logs of container | docker logs <container\_name/id> -f |
| go inside file-system of container | docker exec -it <container\_name/id> /bin/sh |
| Expose port | docker run --expose=8080 my-web-app |
| Docker currently used port | docker port <container-id or container-name> |

**Reference:** [Docker commands official](https://docs.docker.com/engine/reference/commandline/docker/)

**Additional Images**

****

