Intro to Docker

- Docker is built to manage containers, therefore it is important to understand the term `containers`

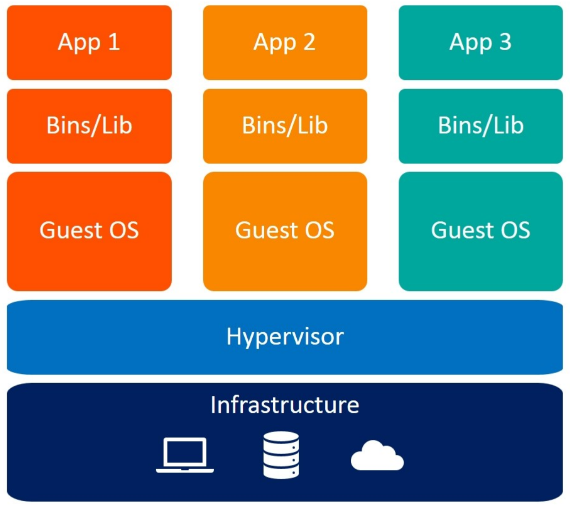
What is a container?

The Problem:

- Software development rate has skyrocketed but this has introduced discrepancies in terms of environment configuration, application isolation, and avoiding resource access conflicts.

Solution #1: Virtual Machines

- A Virtual Machine runs an instance of some Operating System and the host machine hardware is virtualized by a bare-metal hypervisor (a software responsible for the allocation of resources to virtual machines).



Virtual Machines

Solution #1: Virtual Machines (cont.)

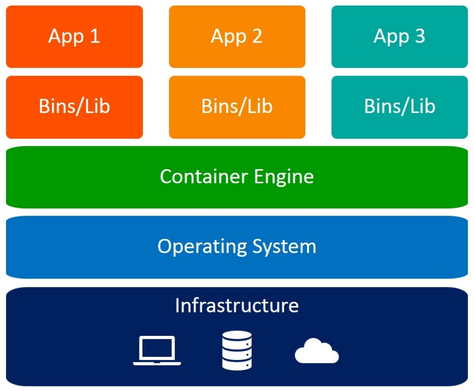
The reason VMs are not always required to run our applications is that:

- Creation of a VM takes time.
- Boot time of a VM is high.
- Most OS sizes are in GBs.
- Dependencies are needed to be installed from scratch.
- Inability to share a VM instance and recreate the exact environment.
- Hardware resource wastage.

Solution #2: Containers

- **Containers** were introduced as a lightweight alternative.
- They made it much easy to develop and deploy software applications.
- These containers are created via software that runs on the host machine serving certain OS.

Here, we do not virtualize hardware but we virtualize the Operating System



Containers

Solution #2: Containers (cont.)

- Containers do not run a complete OS, they operate on kernels.
- A kernel is a core program in any OS that has complete control over all processes.
- The size of these kernels is far less as compared to that of an entire OS. For instance, Ubuntu 18.04 LTS is 2.00 GB whereas Ubuntu docker image is 27.25 MB only.
- Containers allowed developers to recreate an environment free of all dependency conflicts.
- Containers run on-demand, they boot quickly and only use resources as and when needed.

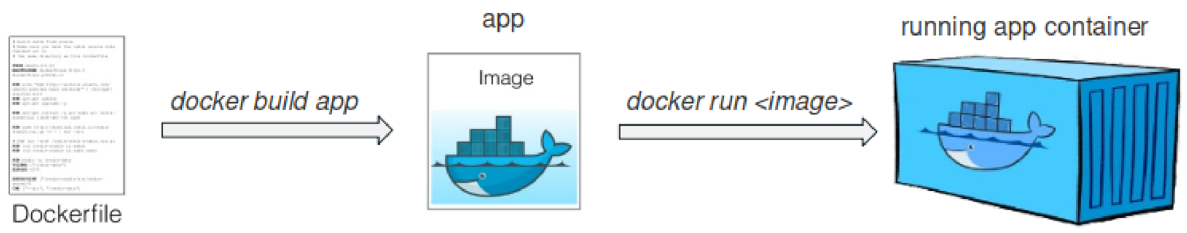
What is docker?

- Before diving into docker commands and deploying docker containers, let us first understand `What is docker?`
- Docker started as an open-source project in 2013 and to put it in the simplest terms, Docker is nothing but software that helps a developer create and manage containers.

Docker components

- **Dockerfile:** A Dockerfile is a simple human-readable text file that instructs docker how to build a docker image. Each line in this file is a command in the form `[INSTRUCTION] [ARGUMENT]`.
- **Docker image:** Docker images are a blueprint of docker containers. It contains information about which software the container will run and how. Docker images are built using Dockerfile.
- **Docker container:** If the docker image is a blueprint then the docker container is an instance of these blueprints. Docker container takes the specifications from docker image and executes them to create an environment ready to run your application.

Docker components (cont.)



- Docker Installation Instructions

Dockerfile

- Dockerfile is a human-readable text file which tells docker how to build a docker image.
- Most of the time is named as `Dockerfile` but one can name it anything.

Dockerfile : Common commands

```
FROM NAME[:TAG|@DIGEST]
```

The `FROM` instruction tells docker-engine which base image is to be used.

Example:

- `FROM ubuntu:latest`
- `FROM redis:community-7.0.0-beta`

Dockerfile : Common commands (cont.)

```
- RUN <COMMAND> (shell form)
- RUN ["EXECUTABLE", "PARAM1", "PARAM2"] (exec form)
```

- The `RUN` instruction executes a command in docker container's shell.
- Default executable for linux is /bin/sh.
- In shell form, the entire command must be specified after the word `RUN`, whereas in exec format, the same command is specified as a JSON array.

Example:

- `RUN apt update`
- `RUN ["npm", "install"]`

Dockerfile : Common commands (cont.)

```
EXPOSE <PORT> [<PORT>/PROTOCOL]
```

`EXPOSE` informs docker that the container will be listening on the specified port.

- `EXPOSE` does not open the port to the host machine, it just serves as a means of documentation to the developer(s).

Example:

- `EXPOSE 5000`
- `EXPOSE 6622/udp`

Dockerfile : Common commands (cont.)

```
ENV <KEY>=<VALUE> [<KEY>=<VALUE>...]
```

Use the `ENV` command to set environment variables.

- Environment variables are a great alternative to explicit variable assignment which includes configuration data and sensitive information. - `=` sign in the instruction can be omitted and replaced by space.

Example:

- `ENV DB=Mongodb`
- `ENV WORKERS 10`

Dockerfile : Common commands (cont.)

```
- CMD <COMMAND> (shell form)
- CMD ["EXECUTABLE", "PARAM1", "PARAM2"] (exec form)
- CMD ["PARAM1", "PARAM2"] (entrypoint default args)
```

- `CMD` command is used to provide default arguments for `ENTRYPOINT` instruction.
- It is used by docker only if no additional arguments are provided while running the docker container.
- The main reason `CMD` is used, is to provide some default arguments in the case where user does not specify any.
- If a user specifies multiple CMD instructions, only the last instruction will be executed by docker.
  Example:
- `CMD ["echo", "linux is the best"]` (executes a command)
- [illegible]

Dockerfile : Common commands (cont.)

```
- ENTRYPOINT <COMMAND> (shell form)
- ENTRYPOINT ["EXECUTABLE", "PARAM1", "PARAM2"] (exec form)
```

When a container is used as an executable there must be one `ENTRYPOINT` command in the dockerfile. ENTRYPOINT arguments can not be over-ridden by providing command-line arguments to `docker run`.

Example:

- `ENTRYPOINT ["ps", "aux"]` (executes a command)
- `ENTRYPOINT ["gunicorn", "--workers", "2", "test:app"]` (start the app and use the container as an executable)

Dockerfile : Common commands (cont.)

```
# Use ubuntu base image
FROM ubuntu:latest
# run apt commands
RUN apt -y update && apt -y install software-properties-commo
# add repository and install python
RUN add-apt-repository ppa:deadsnakes/ppa && apt -y update
RUN apt install python3.7
# make a directory named code
RUN mkdir /code
# copy files from local machine's pwd to container storage
COPY . /code/
# change pwd inside container
WORKDIR /code
# install python dependencies
RUN pip install -r requirements.txt
# inform docker that our container will listen to port 5000
EXPOSE 5000
# start the API application
CMD ["uvicorn", "app:app","--host","0.0.0.0","--port","5000"]
```

Docker Images

Image creation flow

1. The Dockerfile is parsed line by line.
2. For each command, the docker-engine builds a separate layer.
3. The number of layers would be equal to the number of commands specified in Dockerfile.
4. Every time the image is rebuilt docker just has to pull the already created layers, that way recreation of an image is a fast operation.
5. Only the following layers will be rebuilt in case of any change:
    - The layer with changes.
    - All the layers below the changed layer.

Commands (cont.):

List all images:

```
- docker image ls [OPTIONS]
- docker images [OPTIONS]
```

- List all images on your computer. Use the `-a` flag to include intermediate images.

Building a docker image:

```
- docker image build [OPTIONS] PATH
- docker build [OPTIONS] PATH
```

Examples:

- `docker image build .` - Dockerfile in the same directory with name `Dockerfile`
- `docker image build -t my_image:latest code/mydockerfile.txt`

Commands (cont.)

Deleting a docker image:

```
- docker image rm [OPTIONS] IMAGE [IMAGES..]
- docker rmi [OPTIONS] IMAGE [IMAGES..]
```

- Delete single or multiple images. Use the `-f` flag to force delete an image.

Examples:

- `docker image rm -f nginx:1.1`
- `docker image rm Redis:latest python:slim node:old`

Commands (cont.)

Remove unused images:

```
docker image prune
```

- Deletes all unused images, after confirmation. Unused images are the ones that are not tagged. As these images are not used in any container it is the same to delete them and reclaim the disk space.

Commands (cont.)

Show history of an image:

```
- docker image history [OPTIONS] IMAGE
- docker history [OPTIONS] IMAGE
```

- Prints history of the image including layer creation history with time.

Examples:

- `docker image history object-detection:pytorch`
- `docker image history`

Commands (cont.)

Inspect image:

```
- docker image inspect [OPTIONS] IMAGE [IMAGES..]
- docker inspect [OPTIONS] IMAGE [IMAGES..]
```

- Prints all information single or multiple images in a JSON format. Information includes network info, image metadata, environment variables, layers, architecture type, size, etc.

Examples:

- `docker image inspect object-detection:pytorch`

Commands (cont.)

Pull image:

```
- docker image pull [OPTIONS] NAME[:TAG|@DIGEST]
- docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

- Pull images from docker hub. Docker hub is an online repository of docker images that are published by the community members and the official team.

Examples:

- `docker image pull postgres:latest`
- `docker pull postgres@sha256:d9b06fa8350e5ec95da29e2ef99be89c7599ef89646257b9a0fe0cee74428415`

Commands (cont.)

Push image:

```
- docker image push [OPTIONS] NAME[:TAG]
- docker push [OPTIONS] NAME[:TAG]
```

- Push images to docker hub. Any user can push his custom image for sharing or for future use. Users can push frequently used images and pull them whenever required to avoid image recreation.

Examples:

- `docker image push my_custom_go_image:development`

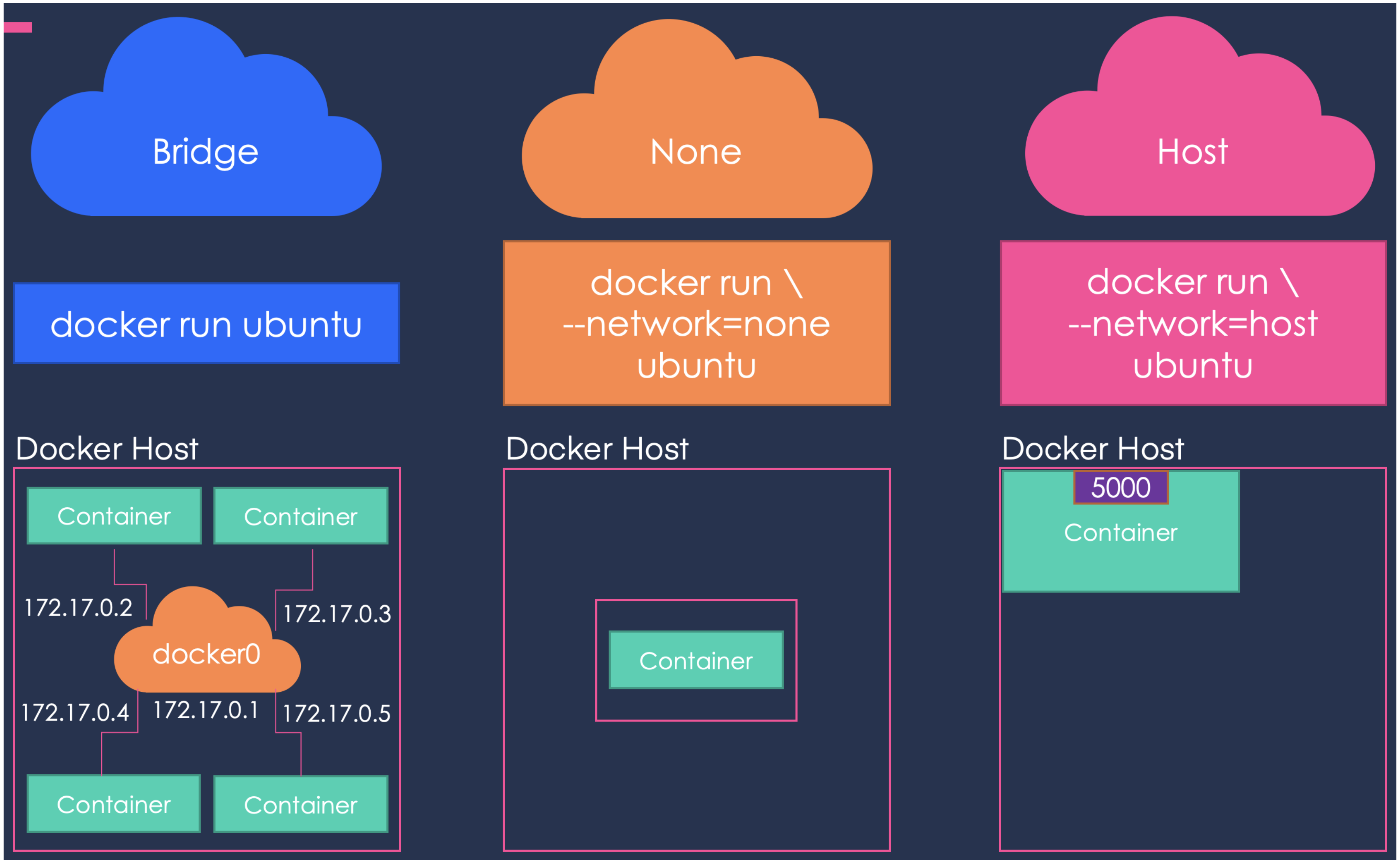
Networking in Docker

- To facilitate communication between different containers on the host machine, docker provides support for multiple types of networks, these include:
- **host:** In this configuration, the container attaches itself to the host port,therefore this port now can not be used by any other process on the host.

Networking in Docker (cont.)

- **bridge:** This is the default network configuration for every container. Docker creates a network of itself with IP addresses in the subnet 172.17.0.0. In bridge mode, the user can map the container port to a port on the host machine.
- **none:** Disable networking.
- **user-defined:** Using `docker network create`, users can create their subnets. It is a good option instead of using the default bridge as it can control which applications are allowed to communicate and which are not.

Networking in Docker (cont.)



Starting a Docker container

- Recall:

  Docker images are blueprints of docker containers

- To bring these blueprints to life is to create a docker container using the `docker run` command. For an instance let's consider we have an image with name `ubuntu:latest` already built onto our system.

The command:

```
docker run [OPTIONS] IMAGE_NAME[:TAG|@DIGEST] [COMMAND] [ARGS
```

Examples:

- `docker run ubuntu:latest`
- `docker run -it -p 5000:80 -e "MODE=DEBUG" my_mongo_image`

Starting a Docker container (cont.)

Options used in docker run:

- There are a plethora of options available for `docker run`. Let's go to a few key ones.

```
-p HOST_PORT:CONTAINER_PORT[/PROTOCOL]
```

The `-p` option publishes the specified container port to the specified host machine port.

Example:

- `docker run -p 5000:80/tcp my_mongo_image`

Starting a Docker container (cont.)

```
-i -t
```

- By default the processes running inside a docker container can not read from the host machine standard input or STDIN. To over come this issue and provide input to the container processes from host machine use the `-i` flag.
- The `-t` flag is used to create a pseudo terminal for our app inside the container, now user can see every prompt from the app straight to the host machine terminal. Both options are used together as `-it`.

Example:

- `docker run -it ubuntu:latest`

Starting a Docker container (cont.)

```
-e VAR=VALUE
```

- The `-e` option sets environment variables in the container. If the variable in this option are already specified in Dockerfile then ones in dockerfile be overridden. Specifying a variable without any value will use the values from host machine.

  Example:

- `docker run -e USER=ADMIN -e PASSWD=hd7632b alpine_image:jimjam`

Starting a Docker container (cont.)

```
--name CONTAINER_NAME
```

- Every container is assigned an internal IP address which does not coincide with the host IP space.
- If multiple containers want to communicate then they must contact via the assigned IP address but here we can not guarantee that a container will be assigned the same IP every time, the workaround is to contact a container via its name.
- Docker assigns a random name to container, unless specified. Providing a name explicitly, creates an entry for the conatiner IP and the name inside docker's embedded DNS.

Example:

- `docker run --name logger_container alpine_image:jimjam`

Starting a Docker container (cont.)

```
--network NETWORK_NAME
```

- Communication between containers can be managed seemlessly using --network. Docker has 4 types of network to choose from these include host, bridge, none, and user-defined. Default is bridge.

Example:

- `docker run --network host alpine_image:jimjam`
- `docker run --network custom_subnet1 alpine_image:jimjam`
- `docker run --network none alpine_image:jimjam`

Starting a Docker container (cont.)

```
-v VOLUME_NAME:CONTAINER_PATH
```

- The `-v` flag is used to mount a volume in the docker container. Value for `-v` is the volume name and path inside container seperated by a `:`

Example:

- `docker run -v vol2:/code/app alpine_image:jimjam`

Data Storage in Docker

The data layers:

- Every docker container has 2 layers, the first layer is a Read-only layer aka the docker image and the second layer is a Writeable layer aka the container storage.

Container data management:

- When a container is stoppped all of the data written inside the container is stored on the host machine disk. On restarting the container this data can be accessed again. But when a container is removed the data created by the container is also deleted permanently. Therefore, data does not persist on the host machine.

Achieving data persistence:

- To save data on the host disk, the container can be mounted to a location on the host machine. There are 3 types of mounts in docker:
- **bind mount:** A bind mount can save data from a docker container to any location on the host file system.
- **volume:** Easiest option available. Data written into docker volumes is stored in `/var/lib/docker/volumes/` (on Linux). Volumes can be mounted into multiple containers.
- **tmpfs:** tmpfs is a temporary file storage system which stores data in a volatile memory i.e. RAM. Therefore, this option can not be counted as true persistence.

Docker volumes:

Create a volume:

```
docker volume create custom_vol0
```

Creates a volume in `/var/lib/docker/volumes`

Delete a volume:

```
docker volume rm custom_vol0
```

Deletes the volume from `/var/lib/docker/volumes`

Inspect a volume:

```
docker volume inspect custom_vol0
```

Shows in-depth information about the volume in a JSON array format.

Mounting a volume

- A volume can be mounted via docker-cli or while creating the docker image by including the volume information in Dockerfile.

1. Via dockerfile: By the using `VOLUME` command in a dockerfile, we tell docker to mount the volume everytime its container is run.

Mounting a volume (cont.)

2. Thorugh docker-cli: Including the `-v` flag with other mounting information in `docker run` command, volumes can be mounted at run-time. An alternative to `-v` is to use it's verbose version `--mount`. The only difference between them is that `--mount` argument names are more verbose and easy to understand.

Example:

- `docker run --name devtest -v myvol2:/app nginx:latest`
- `docker run --name devtest --mount source=myvol2,target=/app nginx:latest`

Lab

- Docker Getting Started

Resources

- (repo)Docker guide
- (docs)Dockerfile reference
- (article)Intro Guide to Dockerfile Best Practices
- (repo)Docker for Java Developers
- (article)Containers vs. VMs: What’s the Difference?by IBM
- (article)Containers vs. virtual machines by Microsoft