What is Docker ?

Docker is a open source centralized containerization platform that allows to package an application with all its dependencies into one single entity as single container which can be easily deployed and run on any machine that supports docker. This makes it easier to develop , test , deploy applications in different environments.

Can You tell What is the Functionality of a Hypervisor?

A hypervisor is a virtualization software that helps in running multiple operating systems (Guest OS) on a single physical host system by providing an isolation between the virtual machines (VMs) and manages their resources.

Advantages of docker?

They consume less storage and memory. It reduces the gb applications to mb. Cost efficient and easy to scale. No down time.

Difference between Docker and Virtualization?

Docker and virtualization are both technologies used to create isolated environments for running applications, but they operate in fundamentally different ways:

Virtual Machines :

- VMs are hardware level Virtualization
- VMs are created using Hypervisor
- VMs are used to run the Operating Systems
- VMs will continue to execute even if there is no task
- VMs consume more space and time to start up.

Containers :

- Containers are hardware level Virtualization
- Containers are created using Container Engine
- Containers are used to run the the Application. NOT Operating Systems
- Containers will immediately go to EXIT state, if there is no task or active applications
- Containers consume less space and time to start up
- Using Containers we can reduce the no. of VM, But we cannot completely elimate VMs
- Container run in its own isolated address spac, the scope of the
  container application is within the container

Containerization: Containerization is a process of packaging the application along with its dependencies.

Virtualizaion: Virtualizaion is a process that allow more efficient utilization of the physical computer to cloud computing.

Dockerfile: Dockerfile contains set of instructions and it automates the process of image creation.

Name and Explain the Components of Docker?

Docker consists of the following as a docker components :

Docker Engine:

-- Docker engine is the runtime that executes containers.

Docker Images:

-- Docker images are lightweight, readable templates Containing executable packages that include the application with its dependencies. Container Images are composed of various Layers created using the Dockerfile Instructions.

Docker Containers:

-- Containers are the executable units of Container Images.Containers are used to run the applications defined in the Container Images .

Docker Compose:

-- Docker compose is a tool for defining and running multi-containered docker applications.

Container Registry:

-- It is used to save and version control the Container Images. Dockerhub is Container Registry to be used.

Container Repositories:

-- Container Repositories are the subset of Container Registry.

On What Circumstances Will You Lose Data Stored in a Container?

The Data in a container can be lost whenever the container is deleted, or if docker non-persistent storage (Ephemeral storage) is used without proper data management. To make the data persistent , it is recommended to use Docker volumes or volume binding (volume mounts) are recommended.

What is Docker Hub?

Docker Hub is container registry that serves as a centralized repository for Docker images. It built for developers and open source contributors to find, use , share and download container images.Docker Hub can be used either host public repos that can be used for free, or docker private repos for teams and enterprises.

What Command Can You Run to Export a Docker Image As an Archive?

You can use this following command to export a Docker image as an archive:

docker save -o <output_file_name>.tar <image_name>

eg:: docker save -o jaya.tar jaya/krishna:8o

Can a Paused Container Be Removed From Docker?

Yes, a paused container can be removed using the command with rm option:

docker rm <container_id>

How Do You get the Number Of Containers Running, Paused, and Stopped?

`docker ps ` for knowing the list of running containers and `docker ps -f "status=paused"` for paused ones. Stopped containers can be counted using `docker ps -f "status=exited"`.

{

docker ps -aq -f "status=exited":

Output: This command lists the container IDs of all exited containers.

Explanation: The -a option includes all containers (running and stopped), the -q option outputs only the container IDs, and the -f "status=exited" filter ensures only exited containers are listed.

docker ps -f "status=exited":

Output: This command lists detailed information about all exited containers.

The following command is used to know number of container are in running state:

docker ps -q | wc -l

The following command is used to know number of container are in paused state:

docker ps -aq -f "status=paused" | wc -l

}

Can You Tell the Difference Between CMD and ENTRYPOINT?

In Docker, both CMD and ENTRYPOINT are instructions used in a Dockerfile to specify what command should be run within a container.

CMD To set the default start-up command to the container. This Command can be changed at run-time.

ENTRYPOINT To set the default start-up command to the container. This Command cannot be changed at run-time.

You can combine ENTRYPOINT and CMD to provide a default command with arguments that can be overridden:

Example:

ENTRYPOINT ["nginx"]

CMD ["-g", "daemon off;"]

In this setup, nginx is the main command, and -g daemon off; are the default arguments. You can override the arguments by providing new ones at runtime, but the nginx command will always be executed.

What Does the Docker Info Command Do?

docker info provides detailed information regarding the Docker system. It includes information such as the number of containers, images, storage driver that are used and much more. It's a valuable command for gaining details on overview of the Docker environment.

eg::docker info

Where are Docker Volumes Stored in Docker?

Docker volumes will be stored on the host machine in the directory /var/lib/docker/volumes . This ensures persistance of the data storage even if the container is removed.

what is home directory for docker?

The home directory for Docker, where it stores all its data including images, containers, volumes, and networks, is typically:

/var/lib/docker/

Can a Container Restart By Itself?

Yes, a container itself can restart automatically by setting up the --restart option during the creation period of time.

docker run --restart

What are Docker Object Labels?

Docker object labels are key-value mapping pair applied to the docker objects for better organizational and metadata purposes. For example, `docker run --label environment=production <image_name>` adds a label to a container.

Example Use Case::

If you have multiple containers running in different environments (e.g., development, staging, production), you can use labels to easily identify and manage them. For instance, you could filter all production containers using a command like:

docker ps --filter "label=environment=production"

Why is Docker System Prune Used? What Does It Do?

`docker system prune` is used for removal of unused data on inclusion of stopped containers, docker networks, and dangling images. It helps in freeing up the disk space on cleaning unnecessary resources.

How Do You Scale Docker Containers Horizontally with docker-compose?

Horizontal scaling is achieved through replicating the services across multiple nodes. Tools like Docker Compose or Docker Swarm facilitate this process. For example, using `docker-compose up --scale web=3`

How Do You Inspect the Metadata of a Docker Image?

By using the `docker inspect <image_name>` command , you can examine into detailed metadata about the Docker image. This contains the information regarding labels, layers, and the configuration settings.

why docker inspect command is used for?

The docker inspect <container_id_or_name> command is used to retrieve detailed, low-level information about Docker objects, such as containers, images, volumes, and networks.

How Do You Limit the CPU and Memory Usage of a Docker Container?

using the --cpus option you can set the CPU limits and with -m option you can set memory limits.

docker run --cpus=3 -m 1024M <image_name>

What are the Differences Between Docker Community Edition (CE) and Docker Enterprise Edition (EE)?

Usage of Docker Community Edition will be peferable for individuals and small-scale projects, It provides the essential features of containerization for free. On the other hand, Docker Enterprise Edition deals in providing the enterprise needs with advanced features and support for the large-scale projects in production environments.

What Is the Purpose of the "docker checkpoint" Command and docker commit?

The docker checkpoint and docker commit commands serve different purposes in Docker.

Docker Checkpoint:

Purpose: Freezes a running container and saves its state, allowing it to be restored later.

Use Cases: Useful for restarting containers without losing their state, forensic debugging, and migrating containers between hosts.

docker checkpoint create my_container checkpoint_name

Docker Commit:

Purpose: Creates a new image from the current state of a container’s filesystem.

Use Cases: Useful for saving changes made to a container’s filesystem, creating a new image with those changes.

Can We Use JSON Instead Of YAML While Developing a Docker-Compose File in Docker?

Yes, Docker Compose has support for both YAML and JSON formats for defining the configuration of services. While YAML is more commonly used due to its readability and clearness , you

can also use JSON as an alternative. The choice between of two will depends on personal preference on requirements of the projects. To use JSON, simply try on creating a `docker-compose.json` file instead of a `docker-compose.yml` file, and define your services in JSON format.

How Will You Ensure Container 2 Runs Before Container 1 While Using docker-compose?

In Docker Compose, the order of the services startup is determined by their dependencies. By specifying container dependencies with the "depends_on" key in the docker-compose.yml file, you can ensure the desired startup order.

```
services:
  container1:
    depends_on:
     - container2
   ...
  container2:
  ....
```

How to identify vulnerabilities in docker image?

Docker Scan:

Docker has built-in support for security scanning through its integration with Snyk. You can use the docker scan command to scan your images for vulnerabilities.

Example:

docker scan <image_name>

How do the Docker Daemon and the Docker Client Communicate With Each Other?

The Docker daemon and client communicate on using REST APIs. The Docker client will send the commands to the daemon using the API, and the daemon will execute those commands on managing containers, images, and other Docker objects.

Is it a Good Practice to Run Stateful Applications on Docker?

Docker is primarily designed for the stateless applications. On using Docker volumes or persistent storage stateful applications can be runnable but it's crucial to carefully manage data persistence and backup to avoid data loss.

Docker Secrets?

Docker Secrets allows you to securely store and manage sensitive data such as passwords, SSH keys, and API tokens in Docker swarm. These secrets are encrypted both in transit and at rest.

docker secret create db_password mysecretpassword

How Do You Create a Multi-stage Build in Docker?

A multi-stage build in Docker involves with using multiple "FROM" instructions in a Dockerfile. Each "FROM" instruction will begin a new stage, allowing you to build and copy the artifacts from previous stages for reducing the final image size.

Example of Dockerfile with multi-stage build:

```
FROM builder as build
# Build stage
FROM alpine
# Final stage
COPY --from=build /app /app
```

How Do You Update a Docker Container Without Losing Data?

To update a Docker container without losing data, you can try on using a combination of Docker volumes or bind mounts to make the data persistant outside the container. When updating, create a new container with the updated image and then link it to the existing data volume.

How Do You Manage Network Connectivity Between Docker Containers And the Host Machine?

Docker provides several ways to manage network connectivity between containers and the host machine.

Bridge Networks: Bridge networks are the default networks , these are created when a Docker daemon starts. Through this network containers on the same bridge network can communicate with each other.

Host Networks: In this containers will share the host's network namespace. so that containers can directly use the host machine's network interfaces.

Overlay Networks (Swarm Mode): Overlay networks are used in Docker Swarm mode for communication between the services that running on different nodes. They provide multi-host networking for orchestration of containers.

How Do You Debug Issues in a Docker Container?

Debugging techniques will provide a comprehensive approach for troubleshoot and to the resolve issues within Docker containers. Depending on the nature of the problem on following these guided commands you can understands the details of the container's behavior.

Container Logs: On running this `docker logs <container_id>` command you can view the standard output and error logs.

docker logs <container_id>

Inspect Container Details: helps in retrieving the detailed information about the container.

docker inspect <container_id>

Interactive Shell: Helps in accessing the interactive shell inside the container for detailing.

docker exec -it <container_id> /bin/bash

How Do You Share Data Between Containers in Docker?

In Docker, you can share data between the containers on using volumes or by utilizing the `--volumes-from` option. Volumes will provide a persistent and the shared storage mechanism, allowing the data to be accessed and modified by multiple containers.

How Do You Perform a Live Migration Of Docker Containers Between Hosts?

For performing a live migration of docker containers between hosts can be achieved through using container orchestration tools like Docker Swarm or Kubernetes. These tools will manage the seamless containers movements across the hosts on ensuring minimal downtime.

docker exec: It helps in execute the commands in a running container.

Container Execution Modes ::

- **Foreground / Attached Mode** # Default Mode

  docker run <image_name>

- **Background / Detached Mode**

  docker run -d <image_name>

- **Interactive Mode**

  docker run -it <image_name>

Port Mapping / Port Binding ::::

- It is used to expose the container to access through internet.
- It is a process of mapping the container port with the host port.

docker run -it -p <host_Port>:<container_Port> tomcat:8.0

Container :::

- Runs in an isolated address space.

Container

Stateless Applications

- Application that will not have any state of execution
- Applications that will not create any output/need any input.

Stateful Applications ::

- Application that will have state of execution
- Applications that requires some input and generate some output

docker volume :::

- Is used to create permanent Volumes that can be attached to any container to maintain the persistant data accross the Applications running through containers.

docker volume create <vol_name>

docker volume inspect <vol_name>

docker run -it centos bash

docker run -it --mount source=sa-wd-devops-vol1,destination=/sa-wd-devops-vol1 centos bash

1. docker run: This command is used to create and start a new container from a specified image.
2. -it: These are two options combined:

- -i (interactive): Keeps STDIN open even if not attached.
- -t (tty): Allocates a pseudo-TTY, which allows you to interact with the container via the terminal.

3. --mount source=sa-wd-devops-vol1,destination=/sa-wd-devops-vol1: This option mounts a volume into the container. Here's the breakdown:
   - source=sa-wd-devops-vol1: Specifies the name of the Docker volume to mount.
   - destination=/sa-wd-devops-vol1: Specifies the path inside the container where the volume will be mounted.
4. centos: This is the name of the Docker image to use for the container. In this case, it's the CentOS image.
5. bash: This specifies the command to run inside the container. Here, it starts a Bash shell.

docker volume ls

docker run -itd --name cont_name -v vol_name:path image_id

docker run -itd --name cont2 --privileged --volumes-from cont1 image_name

--mount: This is a more explicit and flexible way to mount volumes. The --mount flag has a more detailed syntax and supports additional options.

-v vol_name:path: This is a shorthand way to mount volumes. The -v flag is simpler but less flexible compared to --mount

docker commit ::

- To Create a New Container Image based on the properties of existing Container.

Syntax :

docker commit <exist_con_id> <repo_name>/Image_Name>:<tag>

Eg.:

docker commit 6070980df5f2 loksaieta/sa-javaappbuildimg:v1.0

docker build ::

- To create a new Container Image based on the Dockerfile reference.
- Dockerfile composed of Instructions to Create Docker Container Images
- Application Developers create the Dockerfile and update in the Source Code Repository.

eg:: docker build -t image_name .

Docker File Instructions :::

FROM	# To Identify the Base Image
RUN	# To run the package manager
COPY	# To Copy the file from host volume to container volume
CP	# To Copy the file within the container volumes

ADD # To Copy the file from Host Volume as well as from URL.

ENV # To define the Environment Variable.

ARG # To pass Arguements to the Steps in Dockerfile

EXPOSE # To Define the Container Port.

WORKDIR # To set the current working directory within the Container.

CMD # To set the default start-up command to the Container.
This Command can be changed at run-time.

ENTRYPOINT # To set the default start-up command to the Container.
This Command cannot be changed at run-time.

Docker Compose

- Is an extension/Plugin to Docker Engine.
- Is used to run multiple containers as a Service!
- Docker Compose uses the Yaml file to create the Service definitions

Yaml Files are based on Keys & Values -- key:value Pairs

```
version: '3'
services:
  webserv1:
    image: "tomcat:8.0"
    ports:
```

```
    - 8098:8080
  dbserv1:
    image: "redis:alpine"
```

docker-compose up -d : To create n number of containers

docker-compose start : To start n number of containers

docker-compose stop : To stop n number of containers

docker-compose down : To delete n number of containers

Docker Swarm ::::

Container Orchestration Tool ::::

- Docker Swarm is one the Container Orchestration Tools.
- It is meant only for Docker Containers.
- Used to Ensure High Availability of Containers by creating Replicas of Containers and load balancing.
- We cannot Do Auto-Scaling

To generate a Docker Swarm join command, you first need to initialize a Swarm on the manager node.

docker swarm init --advertise-addr <manager_ip>

docker node ls

Lists all the nodes in the Docker Swarm.

docker service create --name service_name -p 8082:80 --replicas 10 image

Creates a new service named service_name with 10 replicas, mapping port 8082 on the host to port 80 in the container.

docker service ps service_name

Lists the tasks (containers) of the specified service.

docker node update --availability drain <node ip address>

To prevent the node from receiving new tasks and to safely remove.

docker node update --availability active <node ip address>

Sets the specified node back to “active” mode.

docker service scale service_name=10

Scales the specified service to 10 replicas.

docker node rm <node ip address>

Removes the specified node from the Swarm.

docker swarm leave --force

Forces the current node to leave the Swarm.

- docker node rm is used from a manager node to remove another node from the Swarm.
- docker swarm leave --force is used on the node itself to leave the Swarm.

Kubernetes ::::

- It is a Open-Source Container Orchestration Tool
- Kubernetes is used to Deploy any type of Containers.
- It is used to ensure high availability of the services running through Containers.
- Used to Ensure High Availability and High Scalability of Containers by creating Replicas, Auto-Scaling & Load Balancing.

Working with Kubernetes :::

- Used to deploy the Containerized Application Services to the Target Environments

Continous Integration/Continous Deployment Workflow :::

1. Create Source Code
2. Application Build
3. Application Image Build using Dockerfile
4. Publish the Application Image to Container Registry
5. Deploy the Application Images to the Target Environments using Kubernetes

If your web application is running on port 80 inside the container, you can map it to port 8080 on the host machine using the -p flag in the docker run command:

docker run -d -p 8080:80 loksaieta/sa-webappimg:v1.0

After running the above command, you can access your web application through the host machine's IP address or domain name on port 8080. For example, if your host machine's IP address is 192.168.1.100, you can access the application by navigating to:

http://192.168.1.100:8080

Publish/Push the Application Images to Container Registry.

Login to DockerHub using Docker CLI.

- DockerHub LoginID
- DockerHub Access Token
    - Create Access Token in DockerHub

docker login -u loasxsdksaieta

Password: dcasdfasdfasdfasdfasdfasdfasdf

docker push loksaieta/sa-webappimg:v1.0

docker pull loksaieta/sa-webappimg:v1.0

docker-compose components:

version: version is used to maintain the version of docker compose files and its mandatory.

services: It is a service to create a different containers at a time.

volume: It is used to allocate for the backup purpose to the container.

network: It is used to allocate the network (port numbers to the container).

General Commands

docker compose version:

Description: Displays the version information of Docker Compose.

Usage: docker compose version

docker compose config:

Description: Validates and views the Compose file in canonical format.

Usage: docker compose config

1. **Start All Services**:
    - To start all services defined in the docker-compose.yml file, simply run:
    - docker compose start

2. **Start Specific Services**:
    - To start specific services, you can specify the service names. For example, to start only the web service:
    - docker compose start web
    - If you want to start both web and db services, you can list them:

docker compose start web db

Service Management Commands:

docker compose start:

Description: Starts existing containers for a service.

Usage: docker compose start [SERVICE...]

docker compose stop:

Description: Stops running containers without removing them.

Usage: docker compose stop [SERVICE...]

Options:

-t, --timeout: Specify a shutdown timeout in seconds.

docker compose restart:

Description: Restarts running containers.

Usage: docker compose restart [SERVICE...]

docker compose create:

Description: Creates containers for a service.

Usage: docker compose create [SERVICE...]

- The containers are created but remain in a stopped state.
- Useful for preparing the environment without starting the services immediately

docker compose up:

Creates the containers if they do not already exist.

Starts the containers, attaching to their output.

If the containers already exist, it will start them without recreating unless the configuration has changed.

docker compose pause:

Description: Pauses running containers of a service.

Usage: docker compose pause [SERVICE...]

docker compose unpause:

Description: Unpauses paused containers of a service.

Usage: docker compose unpause [SERVICE...]

docker compose wait:

Description: Blocks until a container stops, then prints the exit code.

Usage: docker compose wait [SERVICE...]

Lifecycle Commands

docker compose up:

Description: Builds, (re)creates, starts, and attaches to containers for a service.

Usage: docker compose up [OPTIONS] [SERVICE...]

Options:

-d, --detach: Run containers in the background.

--build: Build images before starting containers.

docker compose down:

Description: Stops and removes containers, networks, images, and volumes.

Usage: docker compose down [OPTIONS]

Options:

-v, --volumes: Remove named volumes declared in the volumes section of the Compose file.

Monitoring Commands

docker compose ps:

Description: Lists containers.

Usage: docker compose ps [OPTIONS] [SERVICE...]

Options:

-q, --quiet: Only display IDs.

docker compose logs:

Description: Views output from containers.

Usage: docker compose logs [OPTIONS] [SERVICE...]

Options:

-f, --follow: Follow log output.

Image Management Commands

docker compose images:

Description: Lists images used by the services.

Usage: docker compose images [OPTIONS] [SERVICE...]

docker compose push:

Description: Pushes service images to a registry.

Usage: docker compose push [SERVICE...]

Utility Commands

docker compose cp:

Description: Copies files/folders between a service container and the local filesystem.

Usage: docker compose cp [OPTIONS] SERVICE:SRC_PATH DEST_PATH|SERVICE:DEST_PATH

docker compose exec:

Description: Executes a command in a running container.

Usage: docker compose exec [OPTIONS] SERVICE COMMAND [ARGS...]

Eg:: docker compose exec web sh

Eg:: docker compose exec db ls /var/lib/postgresql/data

Dockerfile Commands:

1. FROM

Definition: Specifies the base image to use for creating the Docker image. Example:

Dockerfile

FROM python:3.8-slim

2. WORKDIR

Definition: Sets the working directory inside the Docker container. Example:

Dockerfile

WORKDIR /usr/src/app

3. COPY

Definition: Copies files from the host machine to the Docker container. Example:

Dockerfile

COPY . .

4. RUN

Definition: Executes commands inside the Docker container during the image build process. Example:

Dockerfile

RUN apt-get update && apt-get install -y libpq-dev

5. CMD

Definition: Specifies the command to run when the Docker container starts. Example:

Dockerfile

CMD ["python", "app.py"]

6. EXPOSE

Definition: Informs Docker that the container listens on the specified network ports at runtime. Example:

The EXPOSE command does not actually publish the port. To actually publish the port when running the container, use the -p flagon docker run to publish and map one or more ports.

Dockerfile

EXPOSE 5000

7. ENV

Definition: Sets environment variables. Example:

ENV

ENV <key> <value>

This command used to set the environment variables that is required to run the project.

ENV sets the environment variables, which can be used in the Dockerfile and any scripts that it calls. These are persistent with the container too and they can be referenced at any time.

ENV HTTP_PORT="9000"

Dockerfile

ENV DEBUG True

(or)

ENV NGINX_VERSION 1.19.0

RUN apt-get update && \

apt-get install -y nginx=$NGINX_VERSION

8. ENTRYPOINT

Definition: Configures a container that will run as an executable. Example:

Dockerfile

ENTRYPOINT ["./start.sh"]

9. ARG

Definition: Defines a variable that users can pass at build-time to the builder with the docker build command. Example:

Dockerfile

ARG VERSION=latest

```
RUN echo "Building version $VERSION"
```

```
FROM node:14

# Define build-time argument
ARG NODE_ENV=production

# Use the argument in a RUN command
RUN if [ "$NODE_ENV" = "development" ]; \
    then npm install; \
    else npm install --only=production; \
    fi
```

Here, ARG NODE_ENV=production sets a default value for NODE_ENV. You can override this value when building the image:

shell

```
docker build --build-arg NODE_ENV=development -t my-node-app .
```

10. LABEL

Definition: Adds metadata to an image. Example:

Dockerfile

LABEL version="1.0"

11. VOLUME

Definition: a volume is a persistent storage location that exists outside of the container. Volumes are useful for storing data that needs to persist even if the container is stopped or removed.

Dockerfile

VOLUME ["/data"]

12. MAINTAINER

This statement is a kind of documentation, which defines the author who is creating this Dockerfile or who should you contact if it has bugs.

Example:

MAINTAINER Firstname Lastname <example@gmail.com>

13. ADD

The ADD command is used to add one or many local files or folders from the source and adds them to the filesystem of the containers at the destination path.

It is Similar to COPY command but it has some additional features:

If the source is a local tar archive in a recognized compression format, then it is automatically unpacked as a directory into the Docker image.

If the source is a URL, then it will download and copy the file into the destination within the Docker image. However, Docker discourages using ADD for this purpose.

ADD rootfs.tar.xz /

ADD http://example.com/big.tar.xz /usr/src/things/

ENTRYPOINT vs CMD in Dockerfile

Both ENTRYPOINT and CMD define the command that runs when a container starts. Here's when you use each:

ENTRYPOINT

Use ENTRYPOINT to specify a command that will not be overridden by docker run command line arguments. It's ideal for setting up the primary command for your container. Example:

Dockerfile

ENTRYPOINT ["python", "app.py"]

This ensures app.py always runs when the container starts.

CMD

Use CMD to provide default arguments for the ENTRYPOINT command or to specify a command that can be overridden. Example:

Dockerfile

```
CMD ["--help"]
```

If paired with ENTRYPOINT, CMD provides arguments:

Dockerfile

```
ENTRYPOINT ["python", "app.py"]
CMD ["--debug"]
```

This runs python app.py --debug, but you can override --debug when you run the container:

Shell

Combining ENTRYPOINT and CMD

Best used together when you need a default command with optional arguments:

Dockerfile

```
ENTRYPOINT ["python", "app.py"]
CMD ["--default-option"]
```

Example 1: Basic Web App

Dockerfile

```
FROM nginx:latest

ENV NGINX_PORT=80

WORKDIR /usr/share/nginx/html

COPY . .

EXPOSE 80

ENTRYPOINT ["nginx", "-g", "daemon off;"]

CMD ["-g", "daemon off;"]
```

Example 2: Python Flask Application

Dockerfile

```
FROM python:3.8-slim

ENV FLASK_APP=app.py
```

```
ENV FLASK_RUN_HOST=0.0.0.0

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt

COPY . .

EXPOSE 5000

ENTRYPOINT ["flask"]

CMD ["run"]
```

Example 3: Node.js Application

Dockerfile

```
FROM node:14

ENV NODE_ENV=production

WORKDIR /usr/src/app
```

```
COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 8080

ENTRYPOINT ["node"]

CMD ["server.js"]
```

Example 4: Java Application
Dockerfile

```
FROM openjdk:11-jre-slim

ENV APP_HOME=/usr/app

WORKDIR $APP_HOME

COPY . .

RUN ./mvnw install
```

```
EXPOSE 8080

ENTRYPOINT ["java", "-jar"]

CMD ["app.jar"]
```

Example 5: Go Application
Dockerfile

```
FROM golang:1.17

ENV GO111MODULE=on

WORKDIR /go/src/app

COPY . .

RUN go build -o main .

EXPOSE 8080

ENTRYPOINT ["./main"]
```

CMD []

---

docker compose:

Docker Compose YAML File Structure

A Docker Compose YAML file allows you to define and run multi-container Docker applications. Here's a detailed breakdown of its structure:

1. Version

Defines the version of the Compose file format.

yaml

```
version: '3.8'
```

Different versions support different features. Always choose the version compatible with your Docker Engine.

2. Services

Specifies the containers to run. Each service is like a single container configuration.

yaml

```
services:
```

```
  web:
    image: nginx:latest
    ports:
      - "8080:80"
  database:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
```

3. Service Configuration Options

image: Specifies the Docker image to use.
yaml

```
image: nginx:latest
```

build: Defines build instructions for creating an image.
yaml

```
build:
  context: ./path/to/build
  dockerfile: Dockerfile
```

ports: Exposes and maps ports.
yaml

ports:

```
- "8080:80"
```

volumes: Mounts host paths or volumes.

yaml

volumes:

```
- /host/path:/container/path
```

environment: Sets environment variables.

yaml

environment:

```
- MYSQL_ROOT_PASSWORD=example
```

depends_on: Specifies dependencies.

yaml

depends_on:

```
- database
```

4. Networks

Defines networks for communication between services.

yaml

```
networks:
  my-network:
```

5. Volumes

Defines named volumes for persistent storage.

yaml

```
volumes:
  my-volume:
```

Full Example

yaml

```
version: '3.8'

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html
    environment:
      - NGINX_PORT=80
    depends_on:
```

```
    - database

  database:
    image: mysql:5.7
    environment:
      - MYSQL_ROOT_PASSWORD=example
    volumes:
      - db-data:/var/lib/mysql          # No space b/w host and cont

volumes:
  db-data:

networks:
  my-network:
```

In this example:

web service uses the nginx image, maps port 8080 to 80, mounts a volume, sets an environment variable, and depends on the database service.

database service uses the mysql image, sets an environment variable, and mounts a volume for data persistence.

volumes defines a named volume db-data.

networks defines a custom network my-network.

```
 version: '3.8'

services:
  myapp:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: my_custom_container_name
    ports:
      - "8080:80"
    volumes:
      - ./app:/usr/src/app
    environment:
      - NODE_ENV=production
    command: node server.js
```