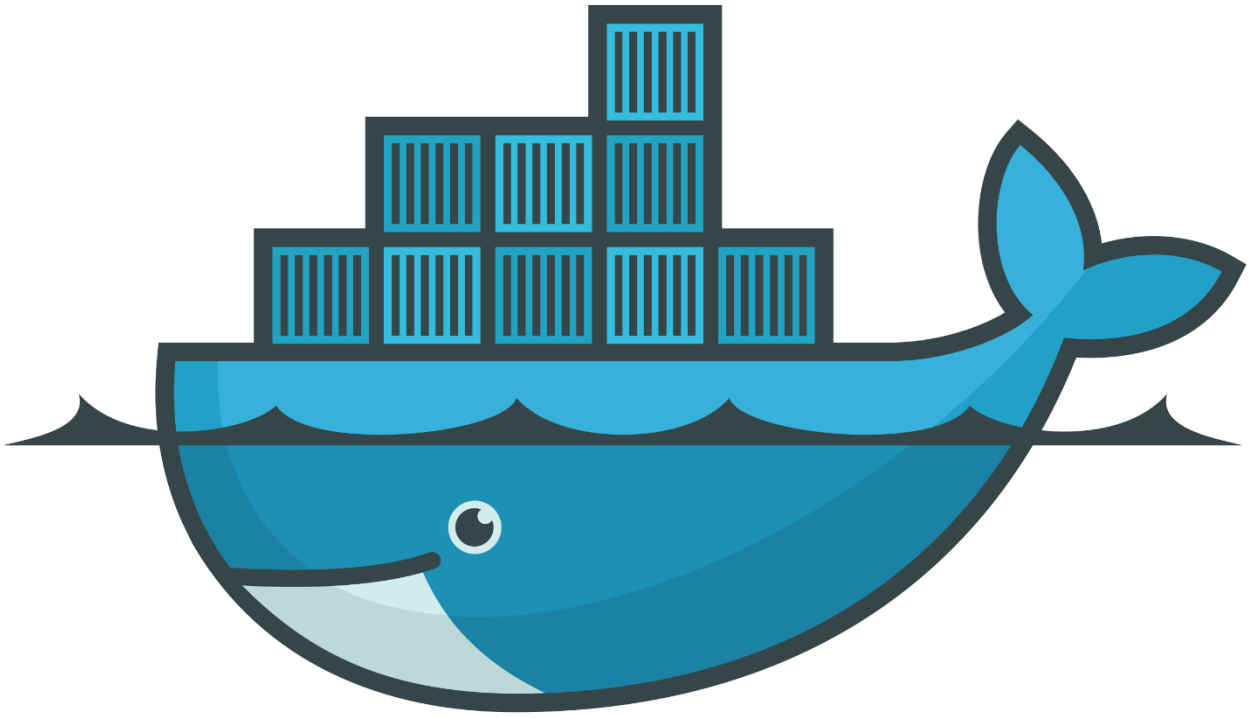


docker

What is Docker?

Docker is an open-source platform that allows you to automate the deployment, scaling, and management of applications using containerization.

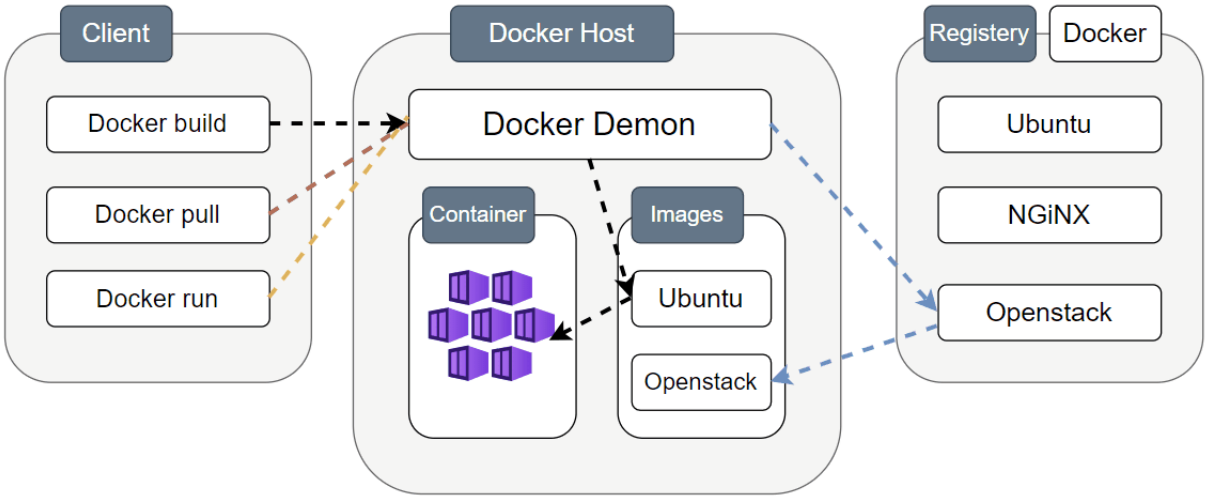
What is Containers?

Containers are lightweight, isolated environments that encapsulate all the necessary dependencies, libraries, and configurations required to run an application.

Feature of Docker :

- Open-source platform
- Fast and efficient development life cycle
- Scalability
- Security
- Networking
- Image management

Architecture of Docker :



Docker utilizes a client-server architecture where the Docker customer speaks with the Docker daemon. It is a daemon that qualifies to create, manage and distribute containers. They could be ran on the same system or client can connect to daemon over REST API using UNIX socket / network.

Docker Daemon :

The Docker daemon manages services by communicating with other daemons and handles Docker objects like images, containers, networks, and volumes through API requests.

Docker Client :

The Docker client allows users to interact with Docker. It sends commands to the Docker daemon via the Docker API. Common commands include `` `docker build` ``, `` `docker pull` ``, and `` `docker run` ``.

Docker Host :

A Docker host runs containers and includes the Docker daemon, images, containers, networks, and storage.

Docker Registry :

Docker images are stored in a registry, with Docker Hub as the public option. Images can be pulled from or pushed to a registry using `` `docker pull` `` and `` `docker push` ``.

Docker Objects:

- **Images:** Read-only templates with instructions for creating containers.
- **Containers:** Run from images, containers are isolated environments where applications execute.
- **Storage:** Manages data within containers using storage drivers.

Docker Networking :

Provides isolation and enables containers to connect to multiple networks.

Types of Docker Networking :

- **Bridge:** Default network for containers on the same host.
- **Host**: No isolation between container and host.
- **None**: Disables networking.

Docker Basic Commands :

- `docker run`: **Creates and starts a new container from an image.**
  Example: `docker run image_name`
- `docker ps`: **Lists all running containers.**
  Example: `docker ps`
- `docker images`: **Lists all available Docker images on your system.**
  Example: `docker images`
- `docker stop`: **Stops a running container.**
  Example: `docker stop container_id`
- `docker start`: **Starts a stopped container.**
  Example: `docker start container_id`
- `docker restart`: **Stops and then starts a container.**
  Example: `docker restart container_id`
- `docker rm`: **Removes one or more containers.**
  Example: `docker rm container_id`
- `docker rmi`: **Removes one or more Docker images.**
  Example: `docker rmi image_name`
- `docker logs`: **Displays the logs of a container.**
  Example: ` docker logs container_id`
- `docker exec`: **Runs a command inside a running container.**
  Example: `docker exec container_id command `
- `docker pull`: **Fetches a Docker image from a registry.**
  Example: `docker pull image_name`
- `docker build`: **Builds a Docker image from a Dockerfile.**
  Example: `docker build -t image_name`

These commands are a great starting point for working with Docker. For more advanced options, refer to the Docker documentation.

What is Docker Compose?

Docker Compose is a tool that simplifies managing multi-container applications. It allows you to define services, networks, and volumes in a `docker-compose.yml` file, making it easy to run and manage your entire application as a single unit.

Key Points:

1. **Defining Services:** Specify the components of your application (e.g., web server, database) in a YAML file.

2. **Managing Dependencies:** Automatically handles service dependencies and the order of startup and shutdown.

3. **Configuration:** Configure services, environment variables, ports, and volumes within the YAML file for easy management.

4. **Service Discovery:** Creates a default network for services to communicate by their names, simplifying networking.

5. **Volume Management:** Define and manage data persistence between containers through volumes.

6. **Container Management:** Start, stop, and manage all containers with a single command.

7. **Overrides:** Use multiple Compose files to customize configurations for different environments or use cases.

Basic Docker Compose Commands:

- `docker-compose up`: Starts the containers defined in the Compose file.
  Example: `docker-compose up`
- `docker-compose down`: Stops and removes the containers defined in the Compose file.
  Example: `docker-compose down`
- `docker-compose build`: Builds or rebuilds the images defined in the Compose file.
  Example: `docker-compose build`
- `docker-compose start`: Starts the containers without creating them.
  Example: `docker-compose start`
- `docker-compose stop`: Stops the running containers.
  Example: `docker-compose stop`
- `docker-compose restart`: Stops and then starts the containers.
  Example: `docker-compose restart`
- `docker-compose logs`: Displays the logs of the running containers.
  Example: `docker-compose logs`
- `docker-compose ps`: Lists the status of the containers.
  Example: `docker-compose ps`
- `docker-compose exec`: Runs a command inside a running container.
  Example: `docker-compose exec service_name command`

- `` `docker-compose down -volumes` ``: **Stops and removes containers and volumes.**
  Example: `` `docker-compose down -volumes` ``

These commands are essential for managing multi-container applications with Docker Compose. For more advanced usage, consult the Docker documentation.

Sample Dockerfile :

Here's a sample Dockerfile that demonstrates the basic structure and commonly used commands:

```
# Use an official Node.js runtime as a parent image
FROM node:20-alpine

# Set the working directory
WORKDIR /src

# Copy package.json and package-lock.json to the working directory
COPY package.json package-lock.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code to the working directory
COPY . .

# Build the application
RUN npm run build

# Expose the application port
EXPOSE 3000

# Start the application
CMD ["npm", "run", "start"]
```

Explanation :

- `` `FROM node:20-alpine` ``: **Uses the official Node.js version 20 image based on Alpine Linux as the base image.**
- `` `WORKDIR /src` ``: **Sets the working directory to `/src` within the container.**
- `` `COPY package.json package-lock.json ./` ``: **Copies the `package.json` and `package-lock.json` files to the `/src` directory inside the container.**
- `` `RUN npm install` ``: **Installs the Node.js dependencies specified in `package.json`.**
- `` `COPY . .` ``: **Copies all the files from the current directory on the host machine to the `/src` directory inside the container.**
- `` `RUN npm run build` ``: **Executes the build command to compile or bundle the application.**
- `` `EXPOSE 3000` ``: **Exposes port 3000 so that the application can be accessed from outside the container.**
- `` `CMD ["npm", "run", "start"]` ``: **Specifies the command to start the application when the container is run.**

This Dockerfile provides a clear, step-by-step process for building and running a Node.js application in a Docker container.

Sample Docker Compose file :

Here's a sample Docker Compose file (docker-compose.yml) that demonstrates a basic configuration for a multi-service application:

```
1. 1 version: '3.8'  # Specify the Docker Compose version

2. 2 services:
3.  3  app:
4. image: node:20-alpine  # Use the official Node.js image
5. working_dir: /src  # Set the working directory inside the container
6. volumes:
      -  .:/src  # Mount the current directory to /src inside the
         container
7. ports:
      -  "3000:3000"  # Map port 3000 on the host to port 3000 in the
         container
8. command: npm run start  # Command to start the Node.js application
9. depends_on:
      -  db  # Ensure the app service starts after the db service

10.db:
11.image: postgres:15-alpine  # Use the official PostgreSQL image
12.environment:
13.POSTGRES_USER: example_user  # Set the database username
14.POSTGRES_PASSWORD: example_pass  # Set the database password
15.POSTGRES_DB: example_db  # Set the database name
16.volumes:
      -  db_data:/var/lib/postgresql/data  # Persist database data

17.volumes:
18.db_data:  # Define a named volume for database persistence
```

In this example:

- **The Docker Compose file uses version 3.8 syntax.**
- **There are two services defined: ` app ` and ` db `.**
- **The ` app ` service uses the ` node:20-alpine ` image from Docker Hub.**
    - **Port mapping is defined to map port 3000 on the host to port 3000 in the container. This allows access to the Node.js application running inside the container from the host machine.**
    - **A volume is created to mount the current directory (` . `) on the host to the ` /src ` directory inside the container. This ensures that any changes made to the code on the host are immediately reflected in the container, facilitating live updates during development.**
    - **The ` npm run start ` command is executed to start the Node.js application. This command runs the application inside the container.**
    - **The ` app ` service depends on the ` db ` service. This dependency ensures that the database service is fully up and running before the application service starts, avoiding potential startup issues.**
- **The ` db ` service uses the ` postgres:15-alpine ` pre-built image from Docker Hub.**
    - **Environment variables are set to configure the PostgreSQL database. These include the database username, password, and database name, ensuring that the database is properly initialized with the desired credentials.**
    - **A volume named ` db_data ` is created to persist the database data. This allows the database data to be stored on the host, ensuring that it is not lost when the container is stopped or removed.**