Here's an overview of **containers and Docker**

Introduction to Containers

Containers are lightweight, portable, and efficient solutions for deploying applications along with their dependencies in isolated environments. They provide consistency across different computing environments by encapsulating everything needed to run an application.

Container Architecture

Containerization is based on:

- **Host OS**: Runs containerization software like Docker.
- **Container Engine**: Manages container lifecycles.
- **Images**: Read-only templates with application and dependencies.
- **Containers**: Instances of images running in isolated environments.
- **Volumes**: Persistent storage for containers.
- **Networking**: Communication between containers and external systems.

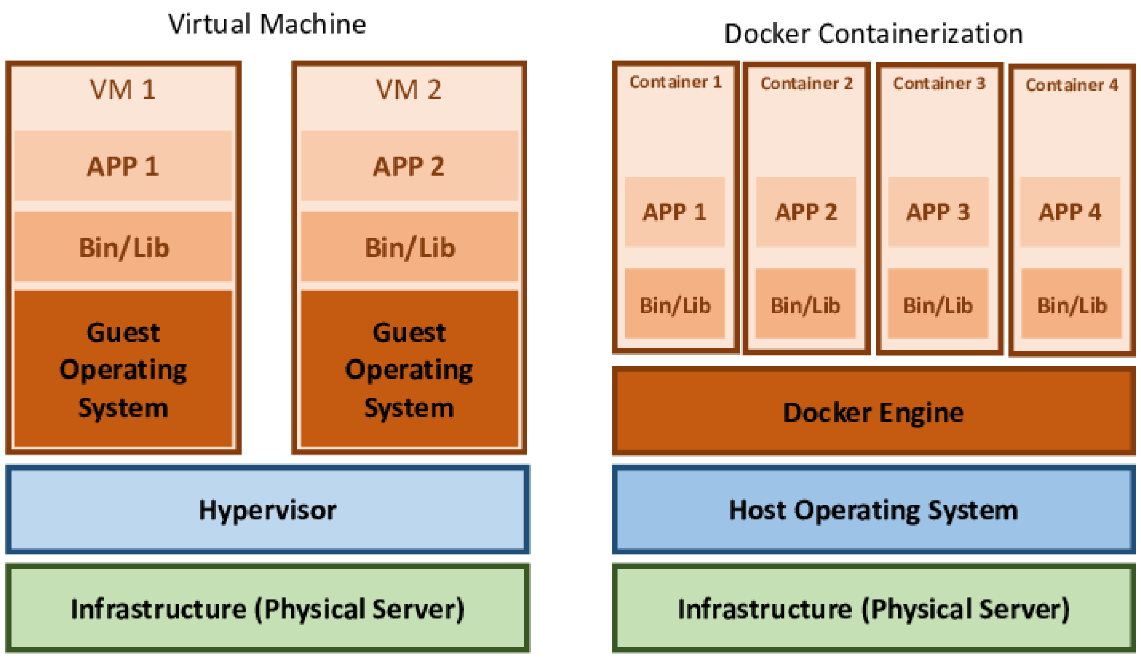
Overview of Docker

Docker is an open-source containerization platform that allows developers to build, deploy, and run applications in containers. It automates application deployment using container technology.

Key Docker Components:

- **Docker Engine**: Core runtime that manages containers.
- **Docker Client**: Command-line interface to interact with Docker.
- **Docker Hub**: Public repository for sharing container images.

- **Docker Daemon**: Background process managing containers.



Docker Architecture

Docker follows a **client-server** architecture:

- **Docker Client**: Issues commands like docker run to the Docker Daemon.
- **Docker Daemon**: Manages containers, images, and networking.
- **Docker Registry**: Stores Docker images.

Working with Containers

1. **Pulling an Image**:

   ```
   docker pull nginx
   ```

2. **Running a Container**:

```
docker run nginx
```

3. **Listing Images**:

```
docker images
```

4. **Running with a Name and Port Mapping**:

```
docker run --name mynginx -d -p 8080:80 nginx
```

5. **Listing Running Containers**:

```
docker ps
```

6. **Listing All Containers (Including Stopped)**:

```
docker ps -a
```

7. **Stopping a Container**:

```
docker stop mynginx
```

8. **Removing a Container**:

```
docker rm mynginx
```

9. **Removing an Image**:

```
docker rmi nginx
```

Dockerfile

A **Dockerfile** is a script containing a series of instructions to build a Docker image.

Step 1: Create a new Java project
We will create a simple Java application with a print statement inside it. Refer to the program below. Note that your file name and Main class name should exactly match each other.

```
class Sample{
    public static void main(String args[]){
        System.out.println("Welcome to Docker");
    }
}
```

2. Dockerfile for a Java Application

A **Dockerfile** for a Java application should use an OpenJDK image, copy the compiled JAR file, and define an entry point to run the Java application.

```
# Use OpenJDK as the base image

FROM openjdk:17

# Set the working directory inside the container

WORKDIR /app

# Copy the Java source file to the container

COPY Sample.java /app/

# Compile the Java program

RUN javac Sample.java

# Run the Java program when the container starts

ENTRYPOINT ["java", "Sample"]
```

Explanation of Commands

- **FROM openjdk:17** → Uses OpenJDK 17 as the base image.
- **WORKDIR /app** → Sets /app as the working directory inside the container.
- **COPY myapp.jar /app/myapp.jar** → Copies the JAR file from the local system to the container.
- **ENTRYPOINT ["java", "-jar", "myapp.jar"]** → Runs the Java application inside the container.

Step 2: Build the Docker Image

Open **Command Prompt (cmd)** and navigate to **E:\Dockerjava**:

```
cd E:\Dockerjava
```

Run the following command to **build the Docker image**:

```
docker build -t my-java-app .
```

Step 3: Run the Docker Container

Once the image is built, run the container:

```
docker run my-java-app
```

This will **compile and run** Sample.java inside a Docker container.

Container Communication

Container Communication Using Docker Networks

Docker networks allow multiple containers to communicate with each other. By default, Docker provides a **bridge network**, but you can create a **custom network** for better control.

1. Create a Custom Network

```
docker network create mynetwork
```

This creates a **custom bridge network** named **mynetwork.**

2. Run Containers on the Custom Network

We will create two simple containers (app1 and app2) that communicate using the same network.

Example 1: Running Two Nginx Containers on the Same Network

```
docker run -d --network mynetwork --name app1 nginx
docker run -d --network mynetwork --name app2 nginx
```

- -d: Runs the container in detached mode.
- --network mynetwork: Connects the container to mynetwork.
- --name: Assigns a name to the container.

0. Verify Connectivity

To check if app1 can ping app2, access app1 and run a ping command:

Use a Different Container with ping

Since Nginx is mainly a web server, it **doesn't include network utilities** by default. Instead, you can create a lightweight test container using **BusyBox** or **Alpine**:

```
docker run -it --rm --network mynetwork busybox sh
```

Now test the connection:

```
ping app2
```

The **ping test was successful**, meaning that app1 and app2 are **correctly communicating** inside the mynetwork Docker network.

Now, your containers can talk to each other using their **container names** instead of IP addresses.

Docker Compose and Related Concepts

Docker is a powerful tool that allows developers to **create, deploy, and run applications in containers**. Let's break it down step by step.

---

1. What is Docker Compose?

Docker Compose is an **open-source tool** that helps you **define and manage multi-container Docker applications**. Instead of running containers one by one manually, **Compose lets you define all the services in a single docker-compose.yml file** and start everything with one command.

Example Use Case:

If you have an application with a frontend, backend, and database, you can use Docker Compose to define all three components and run them together easily.

---

2. What is a Container?

A **container** is a lightweight, standalone environment that includes everything needed to run an application—**code, dependencies, and system tools**.

Think of a container as a **"box"** where your application runs **consistently** on any machine without worrying about different environments.

Why use containers?

- Works the same on any system (Mac, Windows, Linux)
- Lightweight compared to Virtual Machines
- Starts quickly

---

3. What is an Image?

A **Docker Image** is a **blueprint** or **template** used to create a container. An image contains:

- The application code
- Required dependencies
- System configurations

Example:
A **node:18** image contains everything needed to run a Node.js application.

To create a container, you "build" an image and "run" it.

---

4. What is a Virtual Machine (VM)?

A **Virtual Machine (VM)** is a full **operating system (OS) running inside another OS** using virtualization.

Key Differences Between VMs and Containers:

Feature	Virtual Machine (VM)	Container
Size	Large (GBs)	Small (MBs)
Boot Time	Slow (minutes)	Fast (seconds)
Isolation	Strong (separate OS)	Process-level isolation
Performance	Uses more resources	Lightweight

Why Choose Containers Over VMs?

- Containers **share the host OS** instead of running a full OS like VMs.
- They are **faster, smaller, and more efficient**.

---

5. What is a Dockerfile?

A **Dockerfile** is a script that defines how to build a Docker image.
It contains instructions like:

- Which **base image** to use (FROM ubuntu)
- What **dependencies** to install (RUN apt-get install -y nodejs)
- Which **commands** to run (CMD ["node", "app.js"])

---

6. How Docker Compose Works

With **Docker Compose**, you define multiple containers in a docker-compose.yml file.
Instead of running docker run for each container, you just use:

docker-compose up

Example docker-compose.yml file for a web app:

```
version: "3.8"
services:
  web:
    image: nginx
    ports:
      - "80:80"
  database:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD: root
```

Section	Description
version: "3"	Specifies the **Docker Compose file format version** (version 3 is commonly used).
services:	Defines the **containers** (each service runs as a container).
web:	Defines a service/container named **web** running **Nginx**.
image: nginx	Uses the official **Nginx image** from Docker Hub.
ports:	Maps port **8080 on the host** to port **80 inside the**

	container (8080:80).
db:	Defines a service/container named **db** running **MySQL**.
image: mysql	Uses the official **MySQL image** from Docker Hub.
environment:	Sets environment variables for **MySQL**, like MYSQL_ROOT_PASSWORD.

This file tells Docker:

- **Run an nginx web server on port 80**
- **Run a MySQL database with a password**

To start everything, just run:

docker-compose up –d

docker ps

http://localhost:8080

docker exec -it mysql_server mysql -uroot –p

docker-compose down

Conclusion

Docker Compose helps manage multiple containers easily.
Containers are lightweight environments for running applications.
Images are blueprints used to create containers.
VMs are heavier and slower than containers.
Dockerfile defines how to build a custom image.