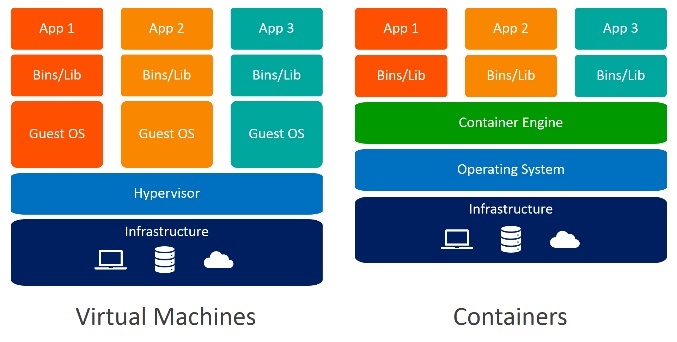
# Docker

**Introduction to Docker:**

* **Definition**: Docker ek containerization tool hai jo applications ko isolated environments me run karne ki facility deta hai. Yeh containers lightweight aur portable hote hain.
* **Use Case**: Docker application ke code aur uski dependencies ko ek saath package karta hai, taaki har system pe same environment me chal sake.

**Containers vs Virtual Machines:**

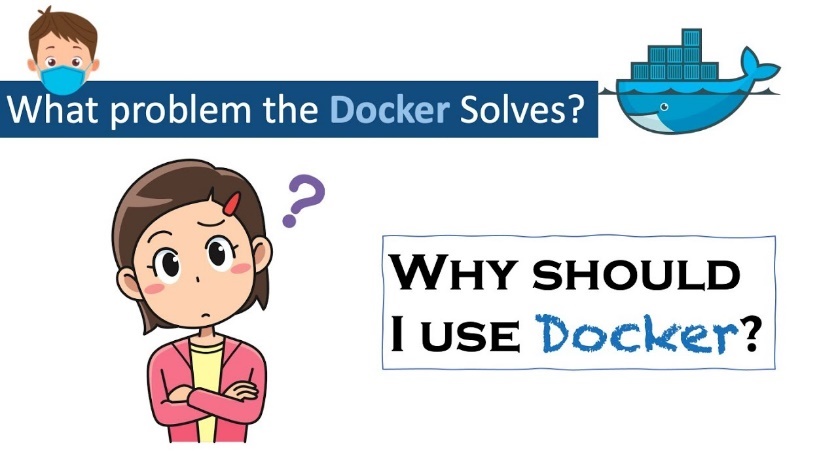


* **Containers**:
  + Lightweight hote hain.
  + Host OS ke kernel ko share karte hain.
  + Sirf application aur required libraries ko contain karte hain.
* **Virtual Machines**:
  + Full operating system ke saath aate hain.
  + Heavy hote hain aur zyada resources consume karte hain.
  + VM hypervisor ke through alag OS chalate hain.

**Key Docker Concepts:**

* **Docker Image**: Blueprint hota hai container banane ke liye. Isme application aur uski dependencies hoti hain.
* **Docker Container**: Ek running instance hota hai Docker image ka. Har container ek isolated process hota hai.
* **Dockerfile**: Ek text file jo instructions contain karti hai for creating Docker images. Example me likha hota hai ki kaunsa base image use karna hai, kya dependencies install karni hain, etc.
* **Docker Hub**: Ek public repository hai jaha se aap predefined Docker images pull kar sakte ho ya apne images push kar sakte ho.

**Why Use Docker?**



1. **Consistency Across Environments**:
   * Docker se **"It works on my machine"** problem solve ho jati hai. Ek baar code aur dependencies ko container me package kar do, wo har jagah same tarike se chalega (local machine, staging, production).
2. **Faster Deployment**:
   * Docker containers lightweight hote hain, isliye application ko quickly deploy kar sakte ho. No need to configure full VMs ya system environments.
3. **Resource Efficiency**:
   * Docker containers zyada resources consume nahi karte kyunki yeh **OS-level virtualization** ko use karte hain. Multiple containers ek hi machine pe chal sakte hain bina alag OS ke overhead ke.
4. **Portability**:
   * Docker containers portable hote hain. Matlab ek baar Docker image bana li, to aap usse **anywhere** (local system, cloud servers, different OS) run kar sakte ho bina modification ke.
5. **Isolation**:
   * Har container apne process, dependencies, aur network settings ke saath completely **isolated** hota hai. Iska matlab, ek container doosre ko affect nahi karega, chahe koi issue bhi ho jaye.
6. **Version Control for Applications**:
   * Docker images ka version control kar sakte ho. Har version ko easily manage aur deploy kar sakte ho. Yeh ek tarah se **Git** jaisa kaam karta hai, but for your application infrastructure.
7. **Easy to Scale**:
   * Docker ke saath **scalability** easy ho jati hai. Agar application ko zyada load handle karna hai, to multiple containers run kar ke scale kar sakte ho (horizontal scaling).

**Use Cases of Docker:**

1. **Microservices Architecture**:
   * Docker ko frequently use kiya jata hai **microservices** ko host karne ke liye. Har microservice ko alag container me run kiya jata hai, jo independent hota hai aur doosre containers ke saath communicate karta hai.
2. **Development and Testing**:
   * Developer ke liye Docker best hota hai, kyunki ek baar container setup kar do, fir sab developers wahi same environment me kaam kar sakte hain. Is se debugging aur testing easy ho jati hai.
3. **Continuous Integration/Continuous Deployment (CI/CD)**:
   * Docker ko CI/CD pipelines me use kar sakte ho for automated testing, building, aur deployment. Jenkins ke saath Docker integration common hai for this purpose.
4. **Multi-cloud and Hybrid Cloud Deployments**:
   * Docker se application ko **multi-cloud** (AWS, Azure, Google Cloud) ya **hybrid cloud** environments me deploy karna easy ho jata hai.

**Advantages of Docker Over Traditional Virtual Machines:**

1. **Faster Startup Time**:
   * Docker containers instantly start ho jate hain, whereas VMs ka boot-up time zyada hota hai.
2. **Lightweight**:
   * Docker containers OS ko share karte hain, isliye unka size chhota hota hai. VMs me full OS chalana padta hai jo heavy aur resource-consuming hota hai.
3. **Simplified Maintenance**:
   * Docker me maintenance easier hota hai, kyunki images aur containers easily update aur manage kiye ja sakte hain.

Example

Suppose tumhare paas ek e-commerce website hai jo microservices architecture pe based hai (shopping cart, user profile, payment gateway alag services hain). Har service ko alag Docker container me deploy kar sakte ho. Agar payment gateway me problem aati hai, to sirf us service ko restart ya scale karna hoga without affecting the rest of the application.

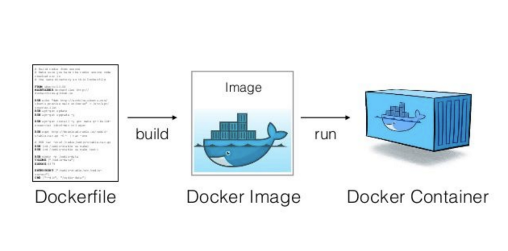
**Problems Solved by Docker:**



1. **"Works on My Machine" Problem**:
   * **Problem**: Ek developer ka code unki local machine pe perfectly kaam karta hai, lekin jab same code kisi aur environment (server ya doosre developer ke system) pe deploy hota hai to issues aate hain, jaise dependencies missing, version conflicts, etc.
   * **Docker Solution**: Docker container me aapka code aur uske dependencies bundled hote hain. Yeh bundle kisi bhi environment me (local, testing, production) same tarike se chalega, kyunki container apna isolated environment leke aata hai.
2. **Complex Environment Setup**:
   * **Problem**: Complex applications ke liye different services, dependencies, aur software versions ko manually configure karna tedious aur error-prone hota hai. Setup me time bhi zyada lagta hai.
   * **Docker Solution**: Dockerfile ke through aap easily apna environment automate kar sakte ho. Ek baar setup karne ke baad, us Dockerfile se same environment kisi bhi machine me build ho sakta hai, jo configuration errors ko avoid karta hai.
3. **Resource Inefficiency with Virtual Machines (VMs)**:
   * **Problem**: Virtual Machines (VMs) full OS ke saath aate hain jo zyada memory, CPU, aur storage consume karte hain, even if aap sirf ek chhoti si application run kar rahe ho.
   * **Docker Solution**: Docker containers lightweight hote hain aur sirf required dependencies ke saath aate hain, bina full OS ke. Isse resources ka efficient use hota hai aur multiple containers ko ek hi machine pe chalayenge to bhi kam resource lagega.
4. **Slow Boot Times in VMs**:
   * **Problem**: Virtual Machines (VMs) ko boot hone me kaafi time lagta hai, kyunki poora OS start hota hai.
   * **Docker Solution**: Docker containers instantly start hote hain, kyunki wo sirf application aur required processes ko run karte hain, full OS ko nahi.
5. **Application Isolation**:
   * **Problem**: Multiple applications ko ek hi system me run karne par conflicts ho sakte hain, jaise ek application ka dependency version doosre ke saath conflict kare. E.g., ek application ko Python 3.8 chahiye aur doosre ko Python 3.6.
   * **Docker Solution**: Docker ke through har application apne alag isolated container me run hoti hai. Iska matlab hai ki har application apne specific dependencies aur configurations ke saath safely run hoti hai, bina doosri application ko affect kiye.
6. **Scalability and Deployment Delays**:
   * **Problem**: Traditional methods me applications ko scale karna ya deploy karna slow aur complex hota hai. Agar aapko apne application ka ek instance badhane ki zarurat ho to manually zyada servers setup karne padte hain.
   * **Docker Solution**: Docker ke containers ko quickly spin-up ya scale kiya ja sakta hai. Aap easily multiple containers run kar sakte ho jab application load badh jata hai, aur jab load kam hota hai to extra containers ko stop kar sakte ho.
7. **CI/CD Pipeline Complexity**:
   * **Problem**: Continuous Integration (CI) aur Continuous Deployment (CD) processes me zyada dependencies aur configurations ki wajah se kaafi baar testing aur deployment fail ho jata hai.
   * **Docker Solution**: Docker CI/CD pipelines ko simplify karta hai, kyunki containers me sab kuch isolated aur predictable hota hai. Isse automated builds aur deployments fast aur reliable ho jate hain.

**Docker Image:**

**Overview**



**Docker image ek lightweight, standalone, executable package hoti hai jisme application ka code aur uske saath us application ko run karne ke liye zaroori sab kuch hota hai (dependencies, libraries, etc.).**

**Ye image ek blueprint hoti hai jo containers banane ke kaam aati hai.**

**Docker Image Basics:**

1. **Image vs Container**:
   * **Image**: Blueprint hoti hai jisme ek application ke run hone ke liye zaroori saari cheezein included hoti hain.
   * **Container**: Ek running instance hota hai Docker image ka.
2. **Layers**: Docker images multiple layers se banti hain. Har layer ek change ko represent karti hai (jaise software install karna, files add karna, etc.). Layers ko reuse karke time aur storage save kiya jata hai.
3. **Read-only**: Docker images read-only hoti hain. Jab container ko image se run kiya jata hai, ek writable layer image ke upar add hoti hai jisme container ki changes hoti hain.

**Docker Image Operations:**

**1. Pulling Docker Images (Download from Docker Hub):**

Docker images ko Docker Hub se pull kar sakte hain, jo ek public registry hai jahan se aap images download karte ho.

Command:

docker pull [image\_name]

docker pull nginx

**2. Listing Docker Images:**

Aapke system pe jo bhi images stored hain, unhe dekhne ke liye:

docker images

**Creating a Docker Image:**

Khud ki image banane ke liye aapko ek **Dockerfile** create karna padta hai. Dockerfile ek plain text file hoti hai jo instructions define karti hai kaise ek image banani hai.

Example :

1.Create a Dockerfile: Dockerfile ke andar application setup ka process likha jata hai.

**Example Dockerfile**:

# Base image

FROM ubuntu:latest

# Author

MAINTAINER your\_name

# Update the image and install packages

RUN apt-get update && apt-get install -y nginx

# Expose the port on which the app will run

EXPOSE 80

# Command to run when container starts

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

2. **Build the Docker Image**: Dockerfile ko use karke image build karte hain.

docker build -t my\_nginx\_image .

Is command se Dockerfile ke instructions follow karke ek new image banegi with the name my\_nginx\_image

**3.Run the Docker Image**: Is image se container run kar sakte hain.

docker run -d -p 8080:80 my\_nginx\_image

* + -d: Detached mode (background me run karega).
  + -p 8080:80: Port mapping (localhost:8080 will be mapped to container’s port 80).

**-p (publish):** Tumhare host machine aur container ke beech ports map karne ke liye.

**8080:80**: Iska matlab tumhare host machine ka port 80 map hoga container ke port 80 se

**Docker Image Commands**:

* 1. **View Specific Image Details**: Image ke detailed information dekhne ke liye:
     1. docker inspect [image\_name]
     2. docker inspect my\_nginx
  2. **Tagging a Docker Image**: Image ko tag kar sakte hain, jo versioning ke kaam aata hai.

docker tag [source\_image] [repository\_name]:[tag]

docker tag my\_nginx\_image myrepo/nginx:v1

* 1. **Pushing Docker Images to Docker Hub**: Agar aapne apni khud ki image banayi hai

aur Docker Hub pe push karna chahte ho, to

docker push [repository\_name]:[tag]

docker push myrepo/nginx:v1

* 1. **Removing Docker Images**: Agar aapko koi image remove karna ho:

docker rmi [image\_name]

**Docker Image Structure:**

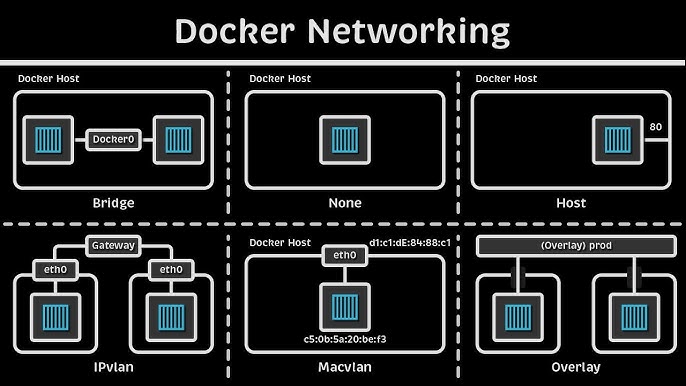
* **FROM**: Base image ka specification, jo aapke image ka base hota hai. Har Dockerfile ek base image se start hoti hai.
* **RUN**: Commands jo aapke base image pe run hote hain (jaise packages install karna, files copy karna, etc.).
* **CMD**: Command jo container start hone par chalegi.
* **EXPOSE**: Port jo container expose karega.
* **COPY/ADD**: Local file ya directory ko image me copy karne ke liye.
* **WORKDIR**: Working directory set karne ke liye container ke andar.

**Docker Networking Basics**

Jab aap Docker containers ko alag-alag services ke liye use karte ho, to unhe ek doosre se communicate karna hota hai. Networking ke bina containers ko ek doosre se ya external world se connect karna mushkil hota hai

Docker mein, jab bhi tum container run karte ho, unhe ek network environment milta hai jo unke communication aur isolation ko handle karta hai. Docker different network drivers provide karta hai jo tumhare container ke networking behavior ko define karte hain.

Docker ke paas kuch predefined networking options hote hain jo aapke containers ko connect karne me help karte hain.



**Types of Docker Networks:**

1. **Bridge Network** (Default network):
   * **What it is**: Ye ek default network hoti hai jo har container ko isolated environment me connect karti hai.
   * **Use Case**: Jab aapko containers ke beech limited communication chahiye, tab ye use hoti hai. Default bridge network ka naam hota hai bridge
   * **Example**: Jab aap command run karte ho docker run bina kisi network specify kiye, to container bridge network pe attach hota hai.
   * docker network inspect bridge
2. **Host Network**:
   * **What it is**: Isme container host machine ke network ko directly use karta hai. Container ke andar ka network host machine ke saath merge ho jata hai.
   * **Use Case**: Jab aapko performance ki zarurat ho aur alag IP ya ports assign karne ki need na ho.
   * **Example**: Agar container ko directly host system ke ports ko access karna ho.
   * docker run --network host [image\_name]
3. **None Network:**

* **What it is:** Is network type me container ko koi network assign nahi hota. Container fully isolated hota hai, bina kisi network interface ke**.**
* **Use Case:** Jab container ko totally isolated environment me run karna ho aur networking ki zarurat na ho
* **docker run --network none [image\_name]**

1. **Custom User-defined Bridge Networks**:

* **What it is**: Ye bridge network ka advanced version hota hai jisme aap khud ke bridge networks define kar sakte ho. Isme containers ko aapas me easily communicate karne ka fayda hota hai bina explicit port mapping ke.
* **Use Case**: Jab aapko multiple containers ko ek shared network pe connect karna ho taaki wo aapas me communicate kar sakein, jaise microservices architecture me
* docker network create my\_custom\_bridge
* docker run --network my\_custom\_bridge [image\_name]

1. **Overlay Network**:

* **What it is**: Ye advanced type ka network hota hai jo Docker Swarm ya Kubernetes clusters ke liye use hota hai. Overlay network alag hosts pe chal rahe containers ko ek shared network pe connect karta hai.
* **Use Case**: Jab aapko distributed architecture banana ho, jisme multiple Docker daemons (hosts) pe running containers ko ek shared network pe communicate karna ho.

**Key Docker Networking Commands:**

1. **Inspecting Networks**:
   * docker network ls: Available networks ki list show karta hai.
   * docker network ls
2. **Creating a Network**:

* docker network create [network\_name]: Custom Docker network banane ke liye.
* docker network create my\_custom\_network

1. **Connecting a Container to a Network**:

* docker network connect [network\_name] [container\_name]: Ek running container ko kisi specific network pe connect karta hai.
* docker network connect my\_custom\_network my\_container

1. **Disconnecting a Container from a Network**:

* docker network disconnect [network\_name] [container\_name]: Container ko network se disconnect karta hai
* docker network disconnect my\_custom\_network my\_container

1. **Inspecting a Network**:

* docker network inspect [network\_name]: Yeh command network ki details dikhata hai, jaise kaun kaun containers us network se connected hain.

**Use Case Example**:

**Scenario**: Tumhare paas ek web application hai jo backend ke liye ek MySQL database use karta hai. Tum in dono containers ko ek custom bridge network pe run karte ho taaki backend (database) ko sirf web application access kar sake aur koi external request database tak directly na pahunch sake.

**Solution**:

1. Custom network create karo:

* docker network create web\_app\_network

1. Web application aur database ko us network pe run karo:

* docker run -d --network web\_app\_network --name mysql\_container mysql
* docker run -d --network web\_app\_network --name web\_app\_container my\_web\_app

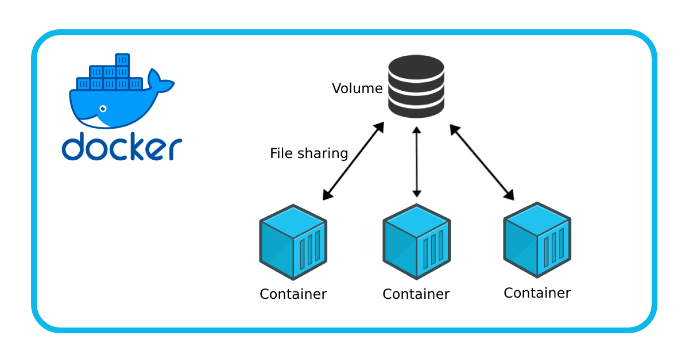
1. Is setup me web application container database container se directly communicate karega, bina external world se expose hue

**Container Communication:**

* **Within Same Network**: Agar containers ek hi network mein hain (like a bridge network), to wo directly communicate kar sakte hain. For example, agar tumhare do containers hain, app\_container aur db\_container, aur wo same network pe hain, to app\_container ko db\_container ke hostname ke through access mil jayega.
* **Between Different Networks**: Agar containers different networks pe hain, to tumhe manually setup karna padega, jaise port mapping or service discovery ke through.

**Docker Volume**

**1. Docker Volume kya hota hai?**



Docker container ka ek default behavior hota hai ki jab bhi tum container ko stop ya delete karte ho, uske saath container ke andar ka saara data bhi delete ho jata hai. **Docker volume** is issue ka solution hai.

Volumes ko tumhare host system pe store kiya jaata hai, aur yeh multiple containers ke beech share kiya ja sakta hai. Yeh isliye zaroori hota hai kyunki container ka internal storage to temporary hota hai (container delete hone par storage bhi delete ho jata hai).

**Docker volume** ek mechanism hai jo tumhare container ke data ko **persistent** banata hai, taaki container ko delete karne ke baad bhi tumhara data safe rahe aur accessible ho.

* **Example**: Tumne ek database container run kiya aur usme kuch data store kiya. Agar tum volume use nahi karte ho, to container ko delete karne par wo data bhi chale jayega. Lekin agar volume use karte ho, to wo data ek external location par store ho jata hai jo container ke delete hone par bhi safe rahega.

**2. Volume Mounting kya hoti hai?**

**Mounting** ka matlab hota hai container ke andar ek directory ko host system ki kisi directory ya volume se **link** karna.

* **Bind Mount**: Host system ki koi specific directory ko container ke andar link karna.
* **Volumes**: Docker manage ki hui storage jaha tum container ka data rakh sakte ho. Ye zyada flexible aur isolated hota hai.

**2. Docker Volume Types**

**Mainly, Docker me 3 tarah ke volumes hote hain:**

1. Volumes (Managed by Docker):
   * Yeh Docker ke under manage hote hain.
   * Tumhe manually host system ka path dene ki zaroorat nahi hoti.
   * Data automatically /var/lib/docker/volumes/ folder me store ho jata hai.
   * Yeh commonly used hota hai jab tum data ko persist rakhna chahte ho bina host path specify kiye.
2. Bind Mounts:
   * Bind mounts me tum specific folder ya file ko host ke filesystem se container ke andar mount karte ho.
   * Example: Agar tumhare host system pe ek folder /host/data/ hai aur tum isko container ke /container/data/ folder me access karna chahte ho, to bind mount ka use kar sakte ho.
   * Tumhe manually host path define karna padta hai**.**
3. tmpfs Mounts:
   * Yeh purely in-memory mount hote hain. Matlab isme data sirf RAM me store hota hai, aur container ke restart ya stop hone pe yeh data delete ho jata hai.
   * Mainly sensitive data ya performance-critical tasks ke liye use hota hai.

#### **3. Volume kaise banate hain?**

Volumes ko manually create karna kaafi simple hota hai:

docker volume create my\_volume

Isse ek volume my\_volume naam ka create ho jayega jo tumhare containers ke liye data store karega

**4. Volume ko container ke saath kaise use karte hain?**

Tum volume ko container ke andar use kar sakte ho **-v** flag ke through:

docker run -d -v my\_volume:/data my\_container

Is command me my\_volume tumhare container ke /data folder me mount hoga.

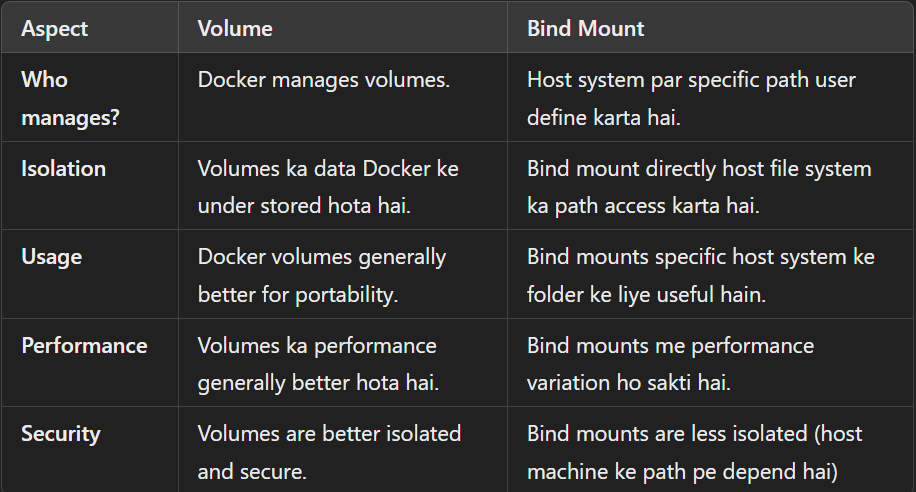
**5. Bind Mount Example**

Bind mount me tum host ke kisi directory ko container ke andar mount karte ho:

docker run -d -v /path/on/host:/path/in/container my\_container

**6. Bind Mount aur Volume me farq kya hai?**

* **Volume** Docker ke through managed hota hai, aur safer hota hai long-term use ke liye.
* **Bind Mounts** specific host path ko link karta hai, but zyada risky hote hain kyunki host ke path me galti se changes ho sakte hain.



**8. Why Use Volumes?**

* **Data Persistence**: Data lost nahi hota jab container delete ho jaye.
* **Data Sharing**: Multiple containers same volume ko access kar sakte hain.
* **Backup & Restore**: Volumes ka backup lena easy hota hai.

**3. Q use karte hain Docker volumes?**

Docker volumes use karne ke kuch important reasons hain:

* **Persistent Data**: Tumhare container ke data ko container ke lifecycle se independent banana. Container delete ho jaye to bhi data rahe.
* **Data Sharing**: Tum multiple containers ke beech ek shared volume create kar sakte ho. Agar tumhe ek database ko multiple microservices ke saath share karna ho, to tum shared volume bana sakte ho.
* **Performance**: Docker volumes ka performance generally better hota hai bind mounts ke comparison me, specially Docker-specific optimizations ke wajah se.
* **Backup & Restore**: Tum easily Docker volumes ka backup le sakte ho aur kisi aur container ya machine par restore kar sakte ho.

### **4. Volume aur Mounting kaise karte hain?**

#### **Example 1: Volume Create Karna**

* 1. Volume create karna:

docker run -d -v /path/on/host:/path/in/container mycontainerimage

* 1. Volume ko container me mount karna

docker run -d -v $(pwd):/usr/share/nginx/html nginx

Isme:

* -v my\_volume:/app/data: Ye tumhare Docker volume my\_volume ko container ke andar /app/data directory se mount karta hai.
* Tum jo bhi data /app/data me store karoge, wo volume me rahega, aur container ko delete karne ke baad bhi available hoga

**Example 2: Bind Mount Karna**

Agar tum chahte ho ki host machine ka koi specific folder container ke andar available ho, to tum **bind mount** use kar sakte ho:

1. Bind mount command:

docker run -d -v /path/on/host:/path/in/container mycontainerimage

docker run -d -v $(pwd):/usr/share/nginx/html nginx

Isme:

* $(pwd) se tum host machine ka current working directory mount kar rahe ho.
* /usr/share/nginx/html: NGINX container ke andar jo folder hai, usko tum mount kar rahe ho.

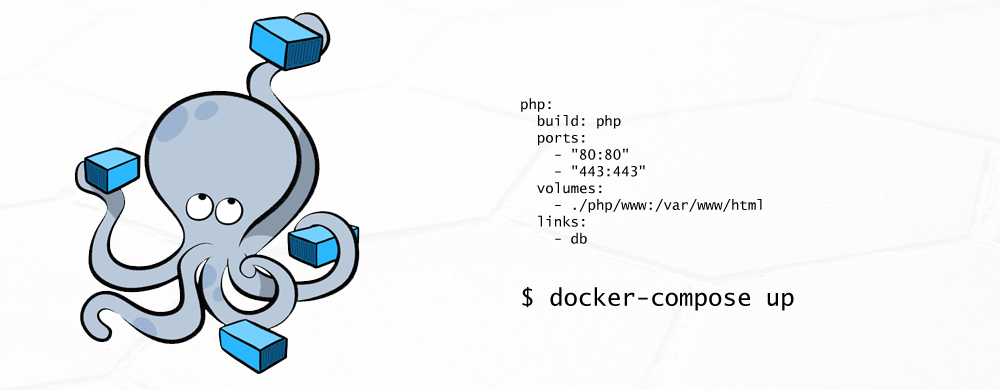
**Example 3: Anonymous Volume**

Tum volume specify nahi karte, Docker automatically ek anonymous volume create kar deta hai:

docker run -d -v /app/data mycontainerimage

Is case me Docker apne aap ek volume banata hai jo tumhare container ke andar /app/data directory ko persist karega.

**Docker Compose**



Docker Compose ek tool hai jo tumhe multi-container Docker applications ko easily define aur manage karne ki capability deta hai. **Compose** tumhe ek single YAML file mein multiple containers ki configuration likhne ka option deta hai, taki tum ek hi command ke through apne poore environment ko launch aur manage kar pao.

**Kya hai Docker Compose?** Docker Compose ek tool hai jo multiple Docker containers ko ek saath run karne ke liye use hota hai. Tumhe manually containers ko start/stop ya configure karne ki zarurat nahi padti; tum bas ek docker-compose.yml file ke through apne environment ko easily manage kar sakte ho.

**Q hai ye important?** Imagine karo ki tum ek complex application pe kaam kar rahe ho jo multiple services ka use karta hai jaise ek web server, ek database, aur ek caching layer. Manually in sab containers ko manage karna mushkil ho sakta hai. Docker Compose is problem ko solve karta hai by providing a single configuration file jisme tum apne poore environment ko define karte ho, aur sirf ek command ke through use launch kar sakte ho.

**Problems Docker Compose Solves:**

1. **Multi-Container Management**: Agar tumhe multiple containers chalane hain jo ek doosre ke saath communicate karte hain (e.g., a web server + database), to Docker Compose tumhe in sab services ko ek hi YAML file mein define karne ka option deta hai. Tum sirf ek command se sab services ko launch ya stop kar sakte ho.
2. **Simplified Configuration**: Har container ko individually configure karne ke bajaye, Docker Compose ek simple file format (YAML) provide karta hai jisme sab containers ki settings likh sakte ho, jaise ports, volumes, environment variables, etc.
3. **Networking Simplification**: Compose automatically ek default network create karta hai jisme sab containers connect hote hain, so tumhe manually container networking setup nahi karni padti.
4. **Reusability**: Ek bar docker-compose.yml file create karne ke baad, tum usse kisi bhi environment mein reuse kar sakte ho, jaise development, testing, aur production.

**Docker Compose Kaise Kaam Karta Hai?**

* **Step 1**: docker-compose.yml file create karo jisme tum apne containers ki configuration likhoge.
* **Step 2**: Ek single command run karo jo sab containers ko launch kar dega aur unko apne network mein connect kar dega.

**Docker Compose Commands:**

1. **Compose Up**: Ye command tumhare docker-compose.yml file ke hisaab se sab services (containers) ko start karta hai.

docker-compose up

1. **Compose Down**: Is command se tum apne sab containers ko stop aur remove kar sakte ho jo docker-compose.yml ke through run ho rahe hain.

docker-compose down

1. **Compose Build**: Agar tumhe Docker images ko rebuild karna hai (especially jab tum apne Dockerfile mein changes karte ho), to ye command use hota hai.

docker-compose build

1. **Compose Logs**: Tumhare containers ke logs dekhne ke liye:

docker-compose logs

1. **Compose Scale**: Agar tumhe ek service ka multiple instances chalana hai:

docker-compose up --scale [service\_name]=[count]

**Docker Compose YAML File: Basic Structure**

docker-compose.yml file mein tum multiple services define karte ho jo ek saath run karenge. Yeh file kuch aise dikhti hai:

version: '3'

services:

web:

image: nginx

ports:

- "8080:80"

volumes:

- ./web:/usr/share/nginx/html

networks:

- app-network

db:

image: mysql

environment:

MYSQL\_ROOT\_PASSWORD: password

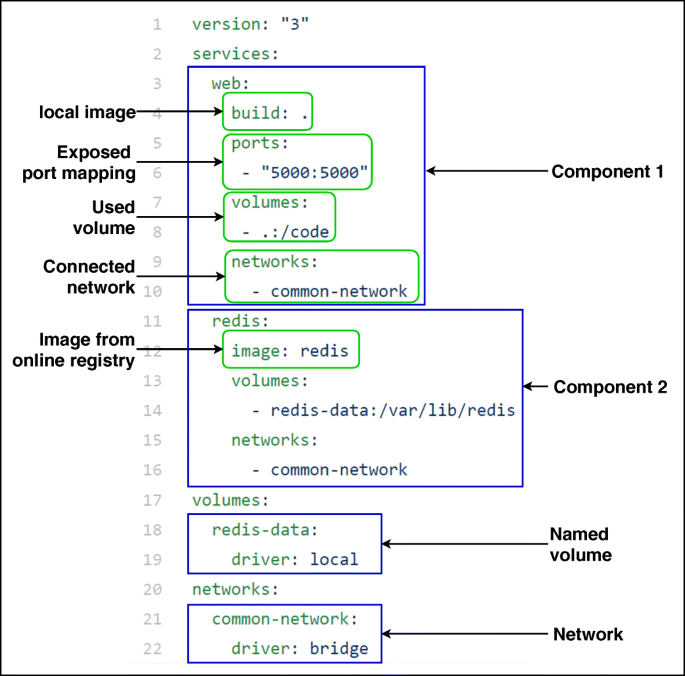
networks:

- app-network

networks:

app-network:

driver: bridge



**Breaking Down the YAML File:**

* **Version**: YAML file ki version define karta hai. Latest Docker Compose mein version 3 use hota hai.
* **Services**: Yahaan pe tum define karte ho ki kaunse containers tumhe run karne hain (jaise web server, database, etc.).
  + **Image**: Jo Docker image use karni hai (e.g., nginx, mysql).
  + **Ports**: Host aur container ports ko map karte ho (e.g., 8080:80).
  + **Volumes**: Host ke files ko container ke andar mount karte ho.
  + **Networks**: Ek network define karte ho jisme containers ko connect karna hai.

**Real-World Example: Web Application with Docker Compose**

Imagine karo tum ek e-commerce website develop kar rahe ho jisme teen main components hain:

1. **Web Server**: NGINX
2. **Database**: MySQL
3. **Caching**: Redis

In sab containers ko manage karna manually tedious hoga. Is liye, Docker Compose use karke tum ek single docker-compose.yml file likhoge aur sab services ko ek saath manage karoge.

YML :

version: '3'

services:

web:

image: nginx

ports:

- "8080:80"

volumes:

- ./web:/usr/share/nginx/html

networks:

- frontend

- backend

db:

image: mysql

environment:

MYSQL\_ROOT\_PASSWORD: password

networks:

- backend

cache:

image: redis

networks:

- backend

networks:

frontend:

backend:

**Explanation**:

* **Web server**: Ye NGINX container hai jo 8080 port pe externally accessible hai.
* **Database**: MySQL container hai jo internal backend network pe run ho raha hai.
* **Cache**: Redis container hai jo backend network mein connected hai.
* **Networks**: Do networks hain — ek frontend jisme web server hai, aur ek backend jisme DB aur Redis hain.

**Benefits of Docker Compose in Real-World Projects:**

1. **Easy to Setup Development Environment**: Multiple containers ko easily setup kar sakte ho ek docker-compose.yml ke through, jo development team ke liye consistent environment ensure karta hai.
2. **Streamlined Testing and CI/CD Pipelines**: Continuous integration/testing ke liye Docker Compose ko use karna easy hota hai kyunki tumhare test environment ke multiple services ko ek saath bring up karna aur test run karna seamless hota hai.
3. **Microservices Architecture**: Agar tum multiple microservices ke saath kaam kar rahe ho jo different containers mein chal rahe hain, Docker Compose un sabko manage karne ka simple way provide karta hai.