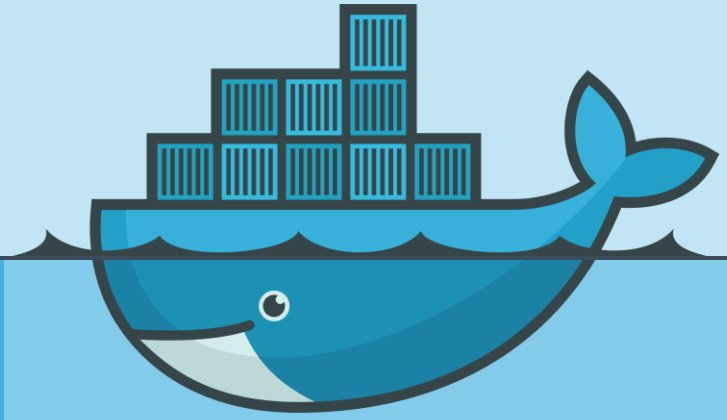

docker



Content

I. Introduction to Containerization
II. Introduction to docker
III. Port Mapping and Exposing
IV. Working with Volumes
V. Docker Compose
VI. Docker Networking
VII. Managing Environment Variables

Introduction To Containerization

Introduction To Containerization

- **What is Containerization?**
  - **Containerization is a lightweight form of virtualization that packages your application and all its dependencies into a single unit called a container, so it can run reliably in different computing environments.**
  - **Imagine sending a package. A container ensures everything the app needs (like libraries, binaries, runtime) is inside the box. No matter where it gets shipped (Windows, Linux, AWS, a laptop), it will work the same.**

Introduction To Containerization

- **Key Characteristics:**
  - **Everything needed to run the app is bundled together**
  - **Containers share the host OS kernel, unlike VMs**
  - **Fast startup, minimal overhead**

Containers VS Virtual Machines

Feature	Containers	Virtual Machines (VMs)
Virtualization	OS-level	Hardware-level (Hypervisor)
Startup Time	Seconds (fast)	Minutes (slow)
Size	Lightweight (MBs)	Heavy (GBs)
Isolation	Process-level isolation	Full OS isolation
Resource Usage	Efficient (shares OS kernel)	Inefficient (each VM has full OS)
Portability	Highly portable (Docker image)	Limited portability

Benefits Of Containerization

- **Lightweight:**
  - **No need to ship full OS**
  - **Just your app and its dependencies**
  - **Typically 10–100x smaller than VMs**
- **Fast Startup:**
  - **Containers start in seconds**
  - **Ideal for microservices and CI/CD pipelines**

Benefits Of Containerization

- **Isolated:**
  - **Each container runs in its own environment**
  - **Reduces risk of conflicts (e.g., library version mismatch)**
- **Reproducible & Consistent:**
  - **"Works on my machine" problem is solved**
  - **Runs the same in dev, test, staging, and production**

Benefits Of Containerization

- **Portable:**
  - **Can be run anywhere Docker is installed (local machine, cloud, CI/CD servers)**
- **Efficient Resource Usage:**
  - **Shares OS kernel, less RAM & CPU required compared to VMs**
- **Simplifies deployment**
- **Easy rollback (just switch to previous container image)**

Introduction To docker

Introduction To Docker

- What is Docker?
  - Docker is an open-source containerization platform that enables you to build, ship, and run applications inside containers.
  - It's the industry standard for containerization today and works on any machine with Docker installed, from your laptop to cloud servers.

Docker as a Containerization Platform

- Docker provides:
  - Tools to create containers (e.g. docker build, docker run)
  - Standards for packaging apps (via Dockerfiles and images)
  - A registry (Docker Hub) to share images
  - A runtime engine that runs containers on any OS
  - With Docker, your entire development and deployment process becomes portable, consistent, and automated.

Docker Engine

- Docker is made of two main components:

1. Docker Client (docker command)

    - What you interact with on the terminal.
    - Sends requests to the Docker daemon.

2. Docker Daemon (dockerd)

    - Background service that actually:
    - Builds images
    - Runs containers
    - Manages networking, volumes, etc.
    - Listens on a Unix socket or REST API

Docker Hub

- Docker Hub is a public cloud-based registry where developers store and share container images.
- It’s like GitHub but for Docker images.
- Features:

1. Pull official images like node, mongo, nginx
2. Publish your own images (free & private options)
3. Versioning via image tags (e.g., node:18-alpine)

Docker Summary

Concept	Description
Docker	A platform to create, run, and manage containers
Docker Engine	The runtime that builds/runs containers
Docker Client	CLI tool you interact with (docker run)
Docker Daemon	Background service that does the work
Docker Hub	Online image registry (like GitHub for containers)

Docker Images

- What Is a Docker Image?
  - A Docker Image is a read-only, layered template used to create Docker containers. It includes:
    - Your application code
    - All of its dependencies
    - System libraries
    - Configuration files
    - Environment setup
    - Instructions on how to start your application
  - It's like a snapshot of a fully configured machine environment, except it's much smaller and only includes exactly what’s needed.

Docker Images

- How to create Docker Images:
- You build images using a Dockerfile, which is a list of step-by-step instructions (e.g. install packages, copy files).
- Each command in the Dockerfile creates a new layer. These layers are cached for performance and re-used when possible.

Create your .dockerfile

```
FROM node:22
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
CMD ["node", "index.js"]

```

Then build your image

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

Docker Images

- Immutable: Once built, an image never changes. If you need changes, you rebuild it.

- Layered (Union File System):
  - Each instruction = new layer.
  - Layers are cached = fast rebuilds.

- Only final image is used when running.

- Portable:
  - Images can be pushed/pulled to/from Docker Hub or other registries.

- Tagged:
  - Images are versioned via tags, e.g., my-api:1.0

Docker Containers

- **Useful Images Terminal Commands:**

```
docker pull node:20-alpine      # Pull the Node.js Docker image
docker images                   # List all Docker images
docker rmi node:20-alpine       # Remove the Node.js Docker image
docker rmi -f node:20-alpine    # Force remove the Node.js Docker image
docker image prune -a           # Remove all unused Docker images
```

Docker Containers

- What Is a Docker Container?
  - A Docker Container is a live, running instance of a Docker image. It’s what actually executes your app.
  - When you run a docker image, Docker takes the image my-app, starts it, and runs it inside a container.
  - This container:
    - Has its own isolated filesystem
    - Shares the host OS kernel (but not filesystem or environment)
    - Runs as a lightweight process
    - Can be stopped, restarted, deleted, or duplicated

Docker Containers

- **Container Behavior:**
  - **Ephemeral by default: if you don’t use volumes, data is lost when the container stops.**
  - **Isolated: Containers can’t see each other unless you connect them via Docker networks.**
  - **Restartable: You can pause, stop, and start containers at will.**

Docker Containers

- **Useful Container Terminal Commands:**

```
docker run -p 3000:3000 my-app   # Start container
docker ps                        # List running containers
docker stop <container_id>       # Stop a running container
docker start <container_id>      # Start a stopped one
docker exec -it <id> bash        # Enter the running container
docker rm <container_id>         # Delete the container
```

Docker Containers

- **Container Lifecycle:**

  - **Created: Container is defined but not running.**
  - **Running: Your app is live.**
  - **Paused: Execution is temporarily frozen.**
  - **Stopped: Container is off but still exists.**
  - **Removed: Deleted permanently.**

Docker Containers

- **Inside a Container:**

  - **Once running, a container:**

    1. **Has its own file system (copied from the image layers)**
    2. **Has isolated processes (doesn’t see host’s processes)**
    3. **Has its own IP address in a Docker network**
    4. **May mount host volumes for persistent or shared data**

Docker Containers VS Images

- **Key Differences in Summary:**

  - **A Docker image is the template: the code, the dependencies, the config, etc.**
  - **A Docker container is the live thing: it's what actually runs your app, using the image as its foundation.**
  - **You can run many containers from the same image.**
  - **If an image is like a class, the containers are like the objects (instances) created from that class.**

Port Mapping and Exposing

Port Exposing

- **EXPOSE in Dockerfile:**

  - **Declares which port(s) the container intends to use.**
  - **It’s informational only: it does not actually publish the port to the host.**
  - **Used for documentation and by tools like Docker Compose.**
  - **This tells Docker: “My app listens on port 3000”, But you still need to map it when running the container.**

Port Mapping

- -p in docker run:
  - This is what actually makes the port available from host to container. It maps a port on your machine to a port in the container.
  - Syntax:

```
1  docker run -d -p 3000:3000 --name node-app-container my-node-app
```

    - Maps host port 8080 to container port 3000, If your Node app is listening on 3000, you can now open http://localhost:8080 on your browser.

Working With Volumes

Docker Volumes

- What Are Docker Volumes?

  - Docker volumes are persistent storage mechanisms managed by Docker that exist outside of the container’s writable layer.
  - They allow data to persist even after a container is removed.
  - Unlike the container’s file system, volumes are not deleted when the container is deleted (unless explicitly told to).

Docker Volumes

- **Why use Docker Volumes?**

  - **Containers are ephemeral, when a container dies or is deleted, everything inside it is lost, including any data generated or stored inside. Volumes solve this by decoupling data from the container's lifecycle.**

Docker Volumes

- There are mainly two types of docker volumes, Anonymous and Named
  1. Anonymous Volumes
     - Docker creates these automatically if you use -v without specifying a name:

```
1  docker run -v /app/data my-image
```

     - Stored with a random hash-like name.
     - Hard to manage because you don’t know the volume’s name.
     - Use Case: Temporary data storage without managing the volume manually.

Docker Volumes

2. **Named Volumes**
   - **You explicitly provide a name:**

```
1  docker run -v mydata:/app/data my-image
```

   - **Easier to manage and reuse.**
   - **Can be shared across containers.**
   - **Use Case: Database files, uploads, persistent configs.**

Docker Volumes

- **Mounting Volumes into Containers**
  - **Docker supports two main ways to mount volumes into containers:**
    1. **With -v flag**

```
1  docker run -v myvolume:/app/data my-image
```

    2. **With --mount flag**

```
1  docker run \
2    --mount type=volume,source=myvolume,target=/app/data \
3    my-image
```

Docker Volumes

- **Use Cases of Docker Volumes**
  1. **Persisting database data:**
     - **If you run a database like PostgreSQL, MySQL, or MongoDB inside a container, you'll lose all your data when the container dies, unless you use a volume.**

```
1  docker run -v pgdata:/var/lib/postgresql/data postgres
```

•This ensures:

1.Your data survives container restarts
2.You can upgrade the container image without losing data

Docker Volumes

- Use Cases of Docker Volumes
    2. Code hot-reloading in development
        - If you're developing a Node.js app, you'd want changes in your code on the host machine to reflect inside the container instantly.

```
1  docker run -v $(pwd):/app -w /app node:20-alpine npm run dev
```

        - $(pwd):/app mounts your current working directory into the container.
        - Changes you make to source code on your host immediately show inside the container.
        - Combine this with nodemon for real-time dev experience.

Docker Volumes

- Use Cases of Docker Volumes

    3. Persisting logs, uploads, and configs

```
1  docker run -v logs:/var/log/myapp my-logging-app
```

    - This allows logs to persist outside of the container and be accessed, shipped, or analyzed even after the container is destroyed.

Docker Volumes

- We can manage attaching docker volumes automatically using a docker compose file:

```yaml
services:
  app:
    image: my-node-app
    volumes:
      - myvolume:/app/data

volumes:
  myvolume:
```

- This creates and manages the volume automatically.

Docker Compose

Docker Compose

- What is docker compose?

  - Docker Compose is a tool used to define and run multi-container Docker applications, using a single YAML file called docker-compose.yml.

  - Instead of typing many long docker run commands manually for each container, you define all services, volumes, ports, etc., in a readable file and run everything together.

Docker Compose

- **Example of Docker compose:**

```yaml
services:
  backend:
    build: .                                    # Uses Dockerfile in the current directory
    ports:
      - "3000:3000"                             # Maps localhost:3000 → container:3000
    volumes:
      - .:/app                                  # Mount source code into container
      - ./docker-uploads:/app/docker-uploads    # Persist uploaded files
    restart: unless-stopped

    ## docker-compose up to start the service
    ## docker-compose down to stop the service and remove containers
    ## docker-compose logs to view logs
```

Docker Compose

Term	Description
services	Define your containers (e.g., app, db)
ports	Maps container port to host (3000:3000)
volumes	Mounts local directories into container
networks	Optional: connect services together securely

Docker Compose

- **Why Use Docker Compose?**

  - **Simplicity: One command to start everything**
  - **Reproducibility: Same config across teams/devices**
  - **Isolation: Each service runs in its own container**
  - **Scalability: Easily add new services or scale them**

Docker Networking

Docker Networking

- Docker uses networking to allow containers to:
  - Communicate with each other
  - Talk to the outside world (e.g., your browser)
- Docker provides built-in drivers for different use cases
  - Types:
    - bridge
    - host
    - none
    - overlay (for Swarm)
    - macvlan (for advanced cases)

Docker Networking

- **Bridge Network (Default)**
  - **Docker's default network type for containers**
  - **Each container gets a private IP address**
  - **A virtual switch connects containers to each other**
  - **Containers can talk to each other if on the same bridge**

```
1  docker run -d --name app1 --network my-bridge node
2  docker run -d --name app2 --network my-bridge mongo
```

Docker Networking

- **Host Network:**
  - **Shares the host's network namespace**
  - **The container uses the host machine's IP and ports**
  - **No port mapping (-p) is needed**

```
1  docker run --network host nginx
```

- **Low latency**
- **Useful for performance-critical services**
- **No network isolation**
- **Port conflicts possible**

Docker Networking

- **None Network:**
  - **Disables networking completely for the container**

```
docker run --network none alpine
```

  - **Use cases:**
    - **Total isolation (security/sandboxing)**
    - **Testing behavior without any network access**

Docker Networking

- **Creating Custom Networks:**
  - **Custom networks allow:**
    - **Service name-based container communication (via DNS)**
    - **Improved isolation and control**

```
1  docker network create my-network
2  docker run -d --name app --network my-network node
3  docker run -d --name db --network my-network mongo
```

Docker Networking

- **Best Practices**

  - **Use bridge or custom networks for app-to-app communication**
  - **Use host only for performance-critical apps (and avoid in Compose)**
  - **Use none for isolated, no-network scenarios**
  - **Use custom bridge networks to enable clean DNS-based container naming**

Managing Environment Variables

Environment Variables In Docker

Ways to Set Environment Variables in Docker

Method	Context	When It Applies
ENV in Dockerfile	Build & Run	Sets default env vars inside image
--env or environment in Compose	Runtime only	Overrides Dockerfile, more flexible
.env file	Compose only	Used to inject vars into docker-compose.yml

Environment Variables In Docker

- **Env in Dockerfile:**

```
1  ENV PORT=3000
2  ENV UPLOAD_DIR=docker-uploads
```

- **Used when: You want default values baked into the image**
- **Used by: process.env.PORT in Node.js**
- **Limitation: Hard to change unless you rebuild the image**

Environment Variables In Docker

- **Overriding with Compose (environment in compose):**

```yaml
backend:
  environment:
    - PORT=4000
    - UPLOAD_DIR=docker-uploads
```

- **Used when: You want to override Dockerfile ENV at runtime**

- **Best Practice: Put all configuration in Compose, keep Dockerfile generic**

Environment Variables In Docker

- **Using .env File with Compose:**

```
PORT=3000
UPLOAD_DIR=docker-uploads
```

```
ports:
  - "${PORT}:${PORT}"
environment:
  - PORT=${PORT}
  - UPLOAD_DIR=${UPLOAD_DIR}
```

- **.env is automatically loaded by Compose**
- **Place it next to your docker-compose.yml**
- **Keeps secrets/configs out of your code**

Any Questions?