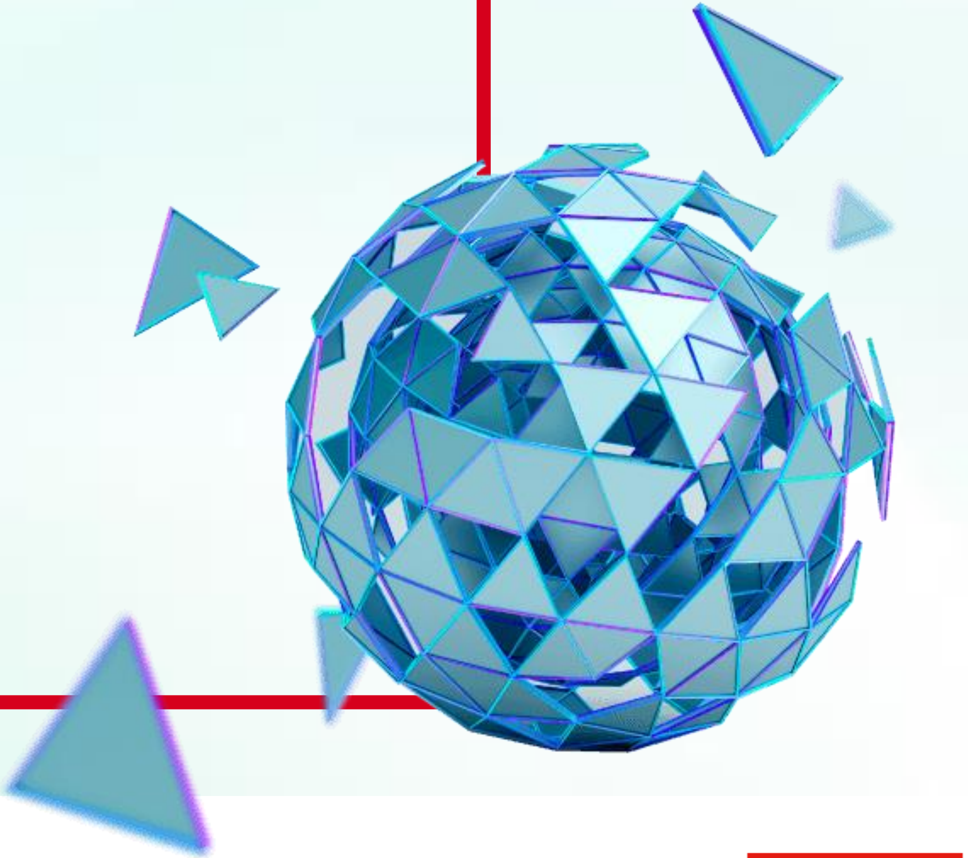
Docker Deep Dive



Nguyen Canh Hat

July 2023

Agenda

Introduction Docker

- What is Docker?
- Why Docker
- Docker Architecture
- Main Features of Docker
- Docker Installation

Docker Ecosystem

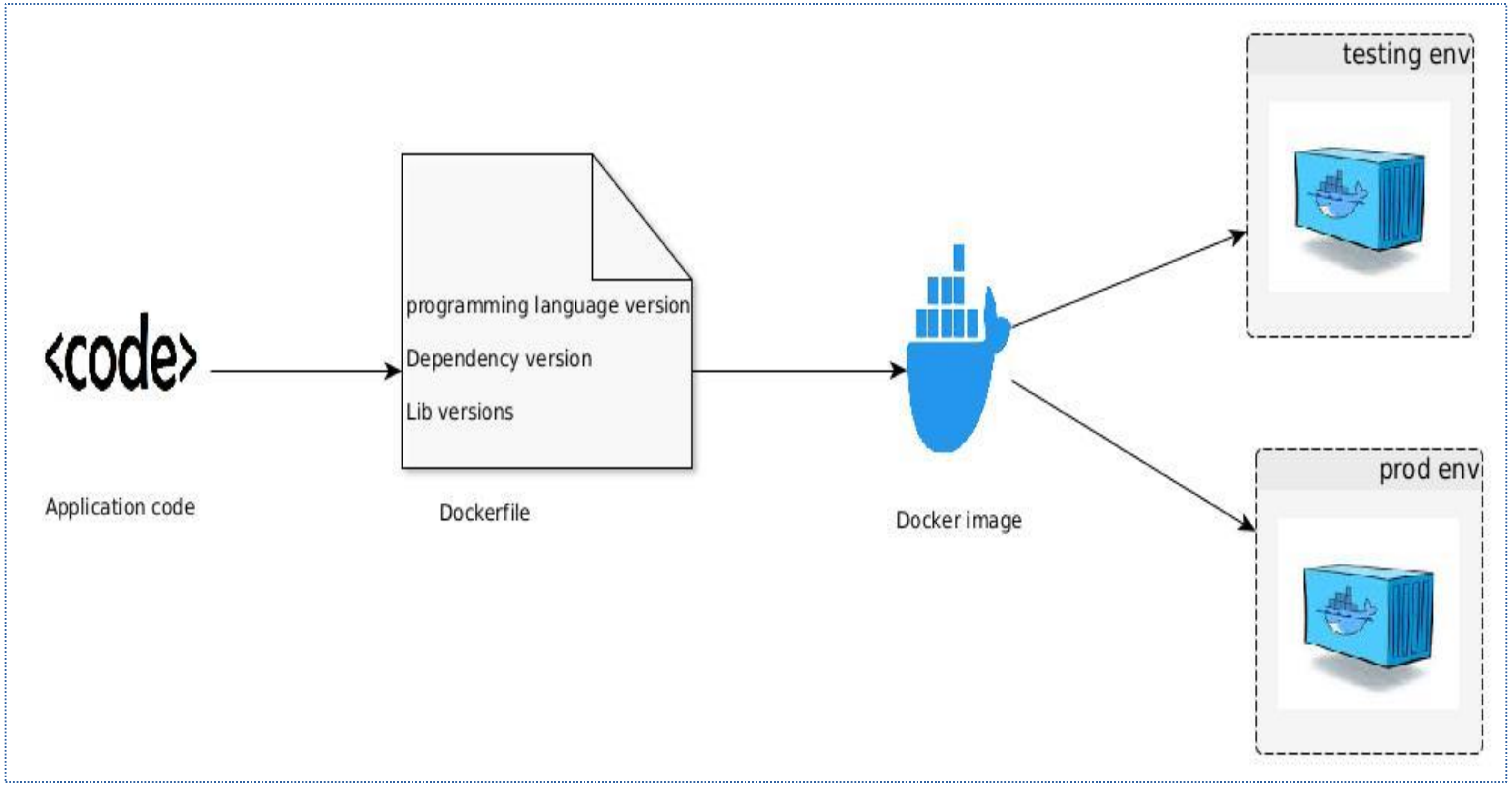
- Docker CLI
- Dockerfile
- Docker Registry
- Docker Volume
- Docker Security
- Docker Compose
- Docker Workflow

Demo

Introduction Docker

What is Docker?

- Docker is a software platform that allows you to package and run applications in a standardized and portable way.



Why Docker?

Docker

-Lightweight and efficient: Share the host machine's operating system kernel.

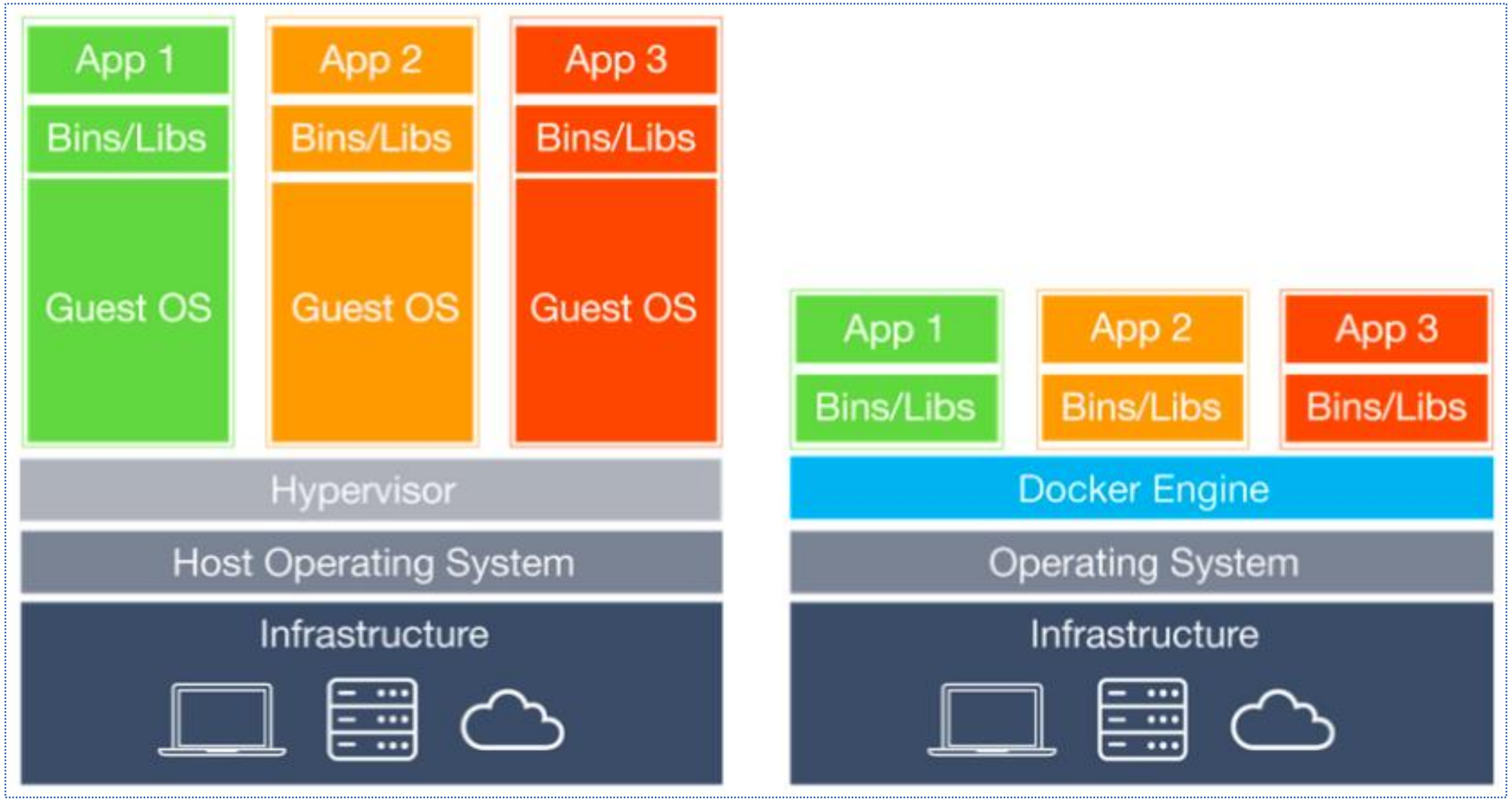
- Faster startup and deployment: Docker containers can start up within seconds.
- Isolation without performance overhead: Eeach container has its own isolated file system, processes, and resources.
- Portability and consistency: Docker provides a consistent runtime environment across different machines and platforms.

Virtual Machine

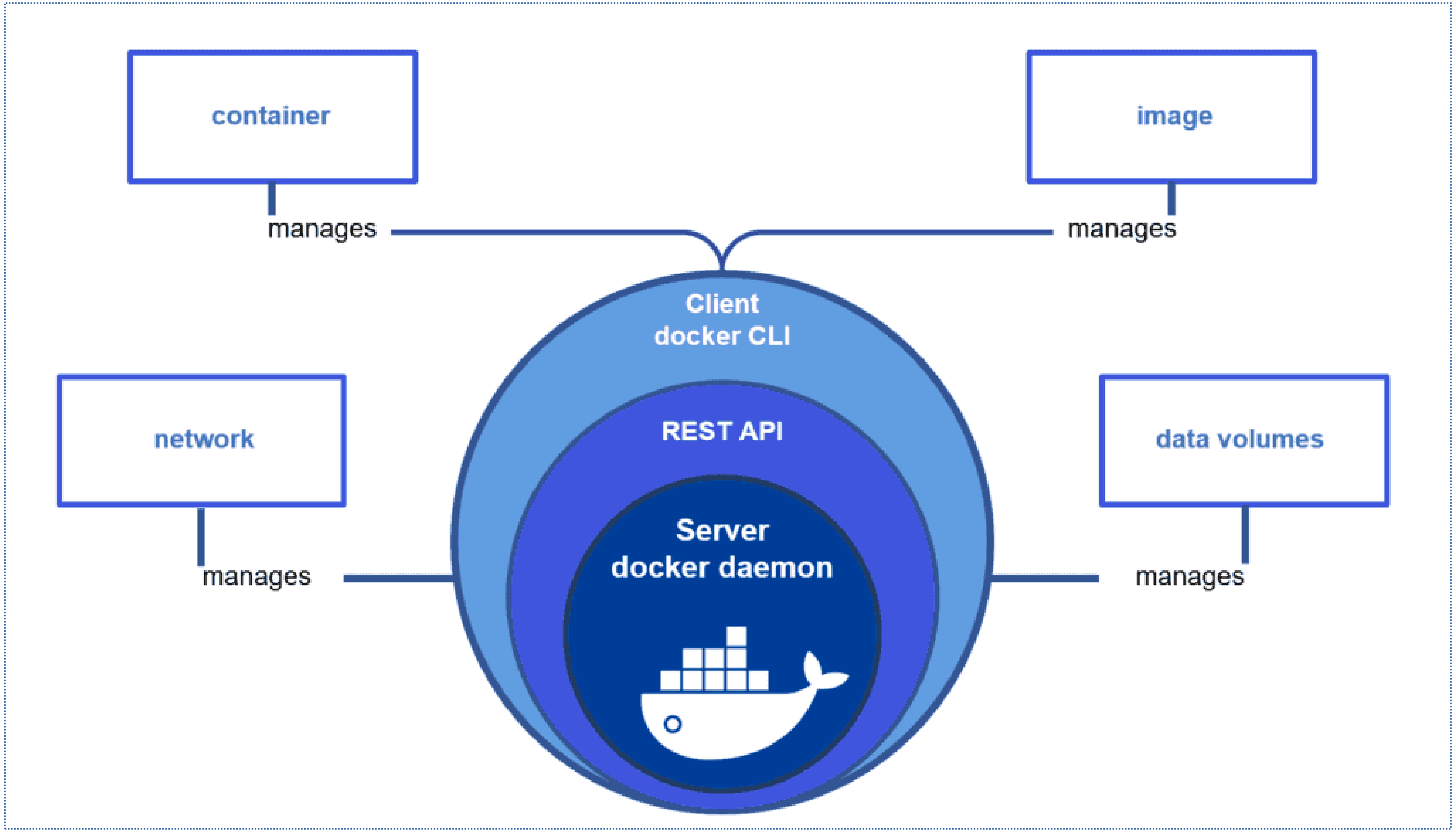
- Require a separate guest operating system for each instance..

- Take minutes to boot.
- Require a complete OS, resulting in more overhead and reduced performance.
- Requires compatibility with the underlying hypervisor or OS versions

Docker Architecture (1/2)



Docker Architecture (2/2)



Main Features of Docker (1/2)

Containerization: encapsulate applications and their dependencies, providing consistency, portability, and ease of deployment.

- Image-based packaging: Images are portable, and can be version-controlled. They capture the entire runtime environmen of an application.

- Efficient resource utilization: Multiple containers can run on a single host machine, sharing the host's operating system kernel.

- Rapid application deployment: Docker containers can be started and stopped quickly, enabling fast application deployment and scaling.

- Portability and compatibility: Eliminate compatibility issues and ensure consistent behavior across different systems, from development to production.

Main Features of Docker (2/2)

Isolation and security: Docker containers provide process-level isolation. Containers have their own file systems and resources, preventing conflicts between applications and reducing the attack surface.

- Versioning and rollbacks: Docker images can be versioned.

- Container networking: Docker provides networking capabilities that allow containers to communicate with each other and the external world.

- Orchestration and scalability: Docker can be integrated with orchestration tools like Docker Swarm and Kubernetes.

- Vibrant ecosystem: Docker has a large and active community.

Docker Installation (Ubuntu)

1) Update your system: ensure that your system is up to date by running the package manager update command.

```
sudo apt update
```

2) Install Docker:

```
sudo apt install docker.io
```

3) Start and enable Docker: Once Docker is installed, start the Docker service and enable it to start on boot. Run the following commands:

```
sudo systemctl start docker
sudo systemctl enable docker
```

4) Verify the installation:

```
docker --version
```

Docker Installation (Windows)

1) Download Docker Desktop: Visit the Docker website (https://www.docker.com/products/docker-desktop) and download the Docker Desktop installer for Windows.

2) Run the installer: Double-click the downloaded installer file and follow the on-screen instructions to install Docker Desktop. During the installation, you may be prompted to enable Hyper-V and/or Windows Subsystem for Linux (WSL) features if they are not already enabled. Enable them as necessary.

3) Start Docker: After the installation is complete, Docker Desktop should start automatically. You will see the Docker icon in the system tray.

4) Verify the installation: Once Docker is running, open a command prompt or PowerShell window and run the following command:

```
docker --version
```

02

Docker Ecosystem

Docker CLI (1/2)

Manage images:

- - List available Docker images:
  docker images
- - Download an image from a registry:
  docker pull <image_name>
- - Build a Docker image from a Dockerfile:
  docker build -t <image_name> <path_to_dockerfile>
- - Push an image to a registry:
  docker push <image_name>
- - Remove an image:
  docker rmi <image_name>

Manage containers:

- - Create and start a new container from an image:
  docker run <image_name>
- - List running containers:
  docker ps
- - Stop a running container:
  docker stop <container_id>
- - Start a stopped container:
  docker start <container_id>
- - Restart a running container:
  docker restart <container_id>
- - Run a command inside a running container:
  docker exec <container_id> <command>

Docker CLI (2/2)

Manage volumes:

- - List Docker volumes:
  docker volume ls
- - Create a new Docker volume:
  docker volume create <volume_name>
- - Inspect details of a Docker volume:
  docker volume inspect <volume_name>
- - Remove a Docker volume:
  docker volume rm <volume_name>

Manage networks:

- List Docker networks:
docker network ls

- Create a new Docker network:
docker network create <network_name>

- Connect a container to a network:
docker network connect <network_name> <container_id>

- Disconnect a container from a network:
docker network disconnect <network_name> <container_id>

Dockerfile (1/5)

A file that describes how to package and configure everything needed for an application to run within a Docker container.

```
# Use an official Python runtime as the base image
FROM python:3.9-slim

# Set the value of an environment variable
ENV APP_PORT=5000

# Set the working directory in the container
WORKDIR /app

# Copy the requirements file to the container
COPY requirements.txt .

# Install the application dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy the application code to the container
COPY app.py .

# Expose a port for the web application to listen on
EXPOSE $APP_PORT

# Set the command to run when the container starts
CMD ["python", "app.py"]
```

Dockerfile (2/5)

COPY compared to **ADD**: used to copy files and directories from the host machine into the Docker image

COPY	ADD
- Keep the original file timestamps	- The timestamps of the copied files will be reset
- NOT support extracting tar files automatically	- Support extracting local tar files automatically
	- Fetch and extract remote URLs, including tar files and compressed archives, into the image
# Example: # Copy a single file *COPY app.js /app/app.js*	# Example: # Add a tar file and extract its contents *ADD archive.tar.gz /app/*
# Copy a directory COPY src/ /app/src/	# Add a remote URL and extract its contents *ADD http://example.com/archive.tar.gz /app/*

Dockerfile (3/5)

CMD compared to ENTRYPOINT: instructions that define the command to be executed when a container is run.

CMD	ENTRYPOINT
- Set the default command and/or parameters for the container.	- Specify the command that will be run as the main process within the container.
- Can be overridden by providing a command when running the container	- Can NOT be overridden by providing a command when running the container (can append params)
- There are 3 forms:	- There are 3 forms:
+ Shell form: *CMD echo "Hello, World!"*	+ Shell form: *ENTRYPOINT echo "Hello, World!"*
+ Exec form: *CMD ["echo", "Hello, World!"]*	+ Exec form: *ENTRYPOINT ["echo", "Hello, World!"]*
+ JSON array form: *CMD ["/bin/sh", "-c", "echo", "Hello, World!"]*	+ JSON array form: *ENTRYPOINT ["/bin/sh", "-c", "echo", "Hello, World!"]*

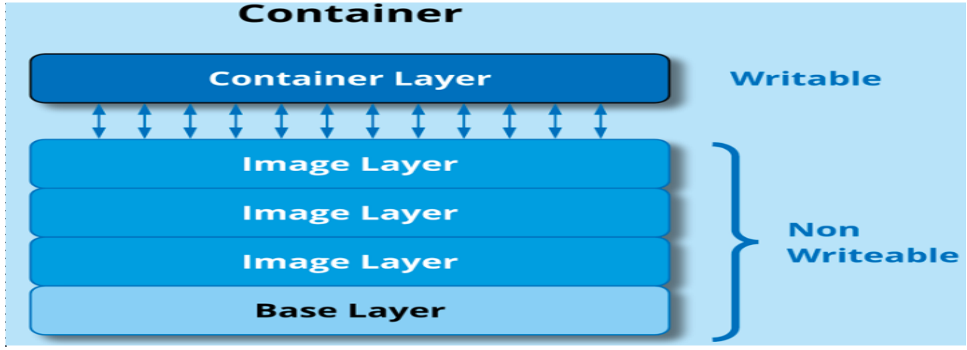
CMD instruction can provide default arguments or options to the main command specified by ENTRYPOINT.

ENTRYPOINT ["python"]

CMD ["app.py"]

Dockerfile (4/5)

Docker image layers: a Docker image is built using a layered file system. Each layer represents a specific change or modification made to the previous layer, resulting in a stack of read-only layers. These layers collectively make up the Docker image.



Benefits:
- Efficient storage utilization: Since layers are read-only and reusable, Docker images can share common layers across multiple images.

- Faster builds and deployments: Docker can reuse the cached layers that haven't changed, avoiding the need to rebuild them.

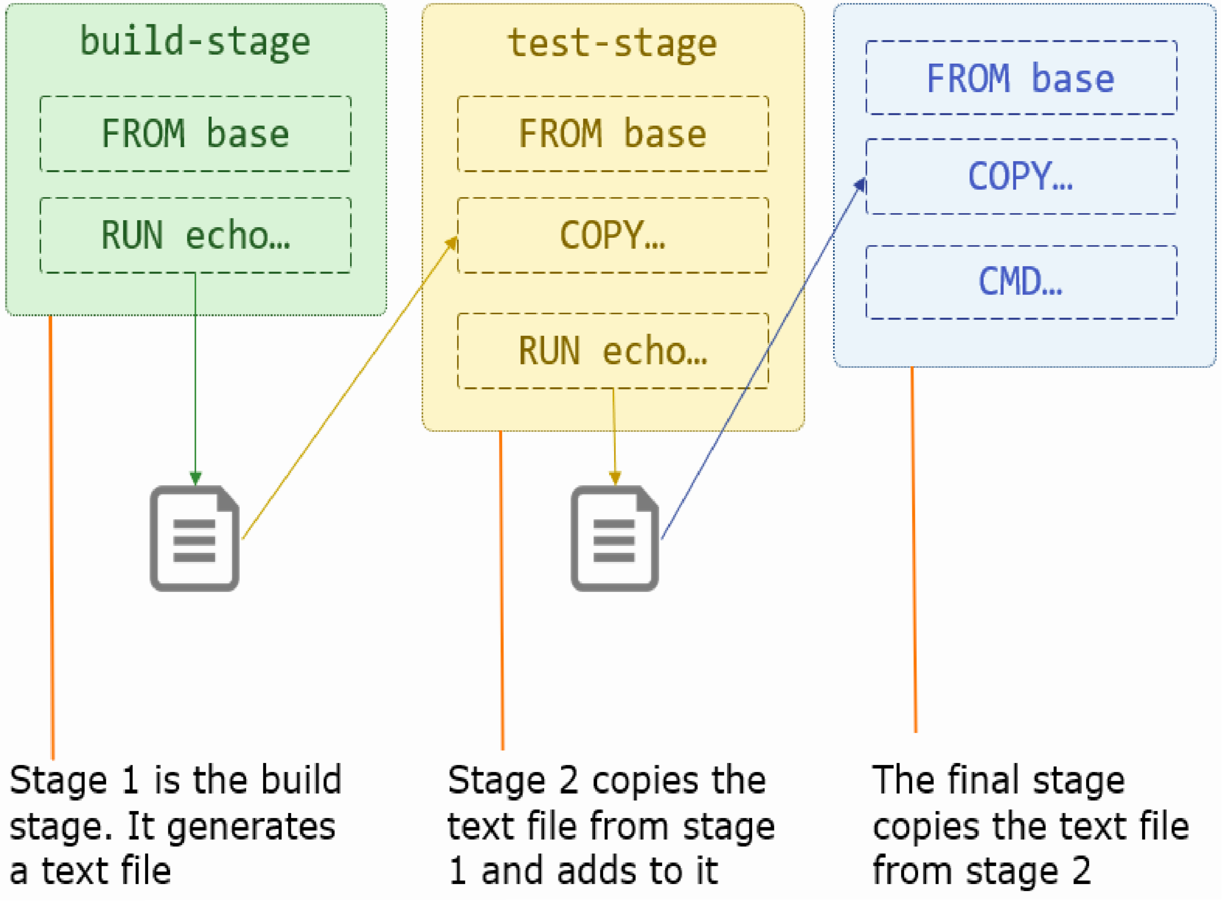
- Better caching and reusability: If a layer has not changed, it can be reused across different images or builds. This caching mechanism reduces the need to re-download or rebuild identical layers, making subsequent builds faster and more efficient.

Example:

FROM python:3.9-slim	*Layer 0 (base image)*
ENV APP_PORT=5000	*Layer 1*
WORKDIR /app	*Layer 2*
COPY requirements.txt .	*Layer 3*
RUN pip install --no-cache-dir -r requirements.txt	*Layer 4*
COPY app.py .	*Layer 5*
EXPOSE $APP_PORT	*Layer 6*
CMD ["python", "app.py"]	*Layer 7*

Dockerfile (5/5)

Docker image Multi-stage builds: allow us to build a final image using multiple stages or phases, each with its own set of instructions. The primary purpose is to help reduce the size of the final image and improve the overall efficiency of the Docker build process.



Stage 1 is the build stage. It generates a text file

Stage 2 copies the text file from stage 1 and adds to it

The final stage copies the text file from stage 2

Example:

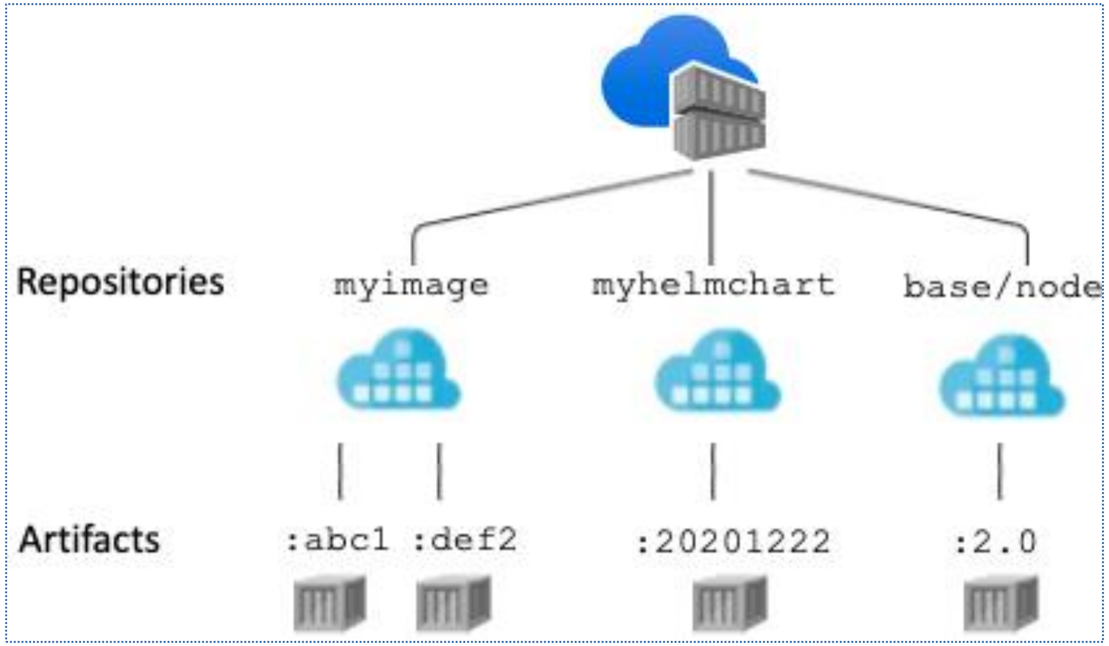
```
# Stage 1: Build stage
FROM node:14 as build-stage
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Production stage
FROM nginx:alpine as production-stage
COPY --from=build-stage /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Docker Registry

- Definition: A service that stores and distributes Docker images.

- Docker registry kinds: Public, Private

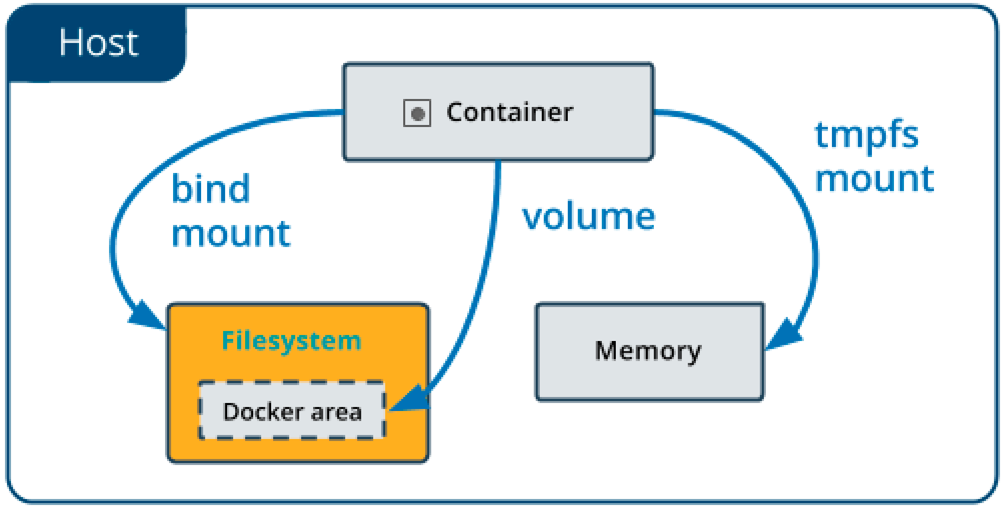
- Main components: Registry server, Registry client



Docker Volume (1/2)

- A volume mount refers to the process of attaching a directory or file from the host machine to a specific location withi container. It allows you to share data between the host and the container, or between multiple containers.

- Why do we need it?
  + Persistent Data
  + Configuration Files
  + Sharing Data Between Containers
  + Development and Debugging



Docker Volume (2/2)

A few examples:

- Docker volume mounts (default folder: /var/lib/docker/volumes/):

```
  docker run -v myvolume:/data

  docker run -v sharedvolume:/data
  docker run -v sharedvolume:/data myimage2

  docker volume create mydata
  docker run -v mydata:/data myimage1
  docker run -v mydata:/data myimage2
```

- Docker bind mount

```
  docker run -d -v /path/on/host:/var/lib/postgresql/data --name postgres \
     -e POSTGRES_PASSWORD=mysecretpassword postgres:latest

  docker run -d -v /path/on/host/config.ini:/app/config.ini --name myapp  myappimage:latest

  docker run -it -v /path/to/source/code:/app --name myapp myappimage:latest
```

Docker Security (1/3)

Here are 2 key aspects of Docker security:

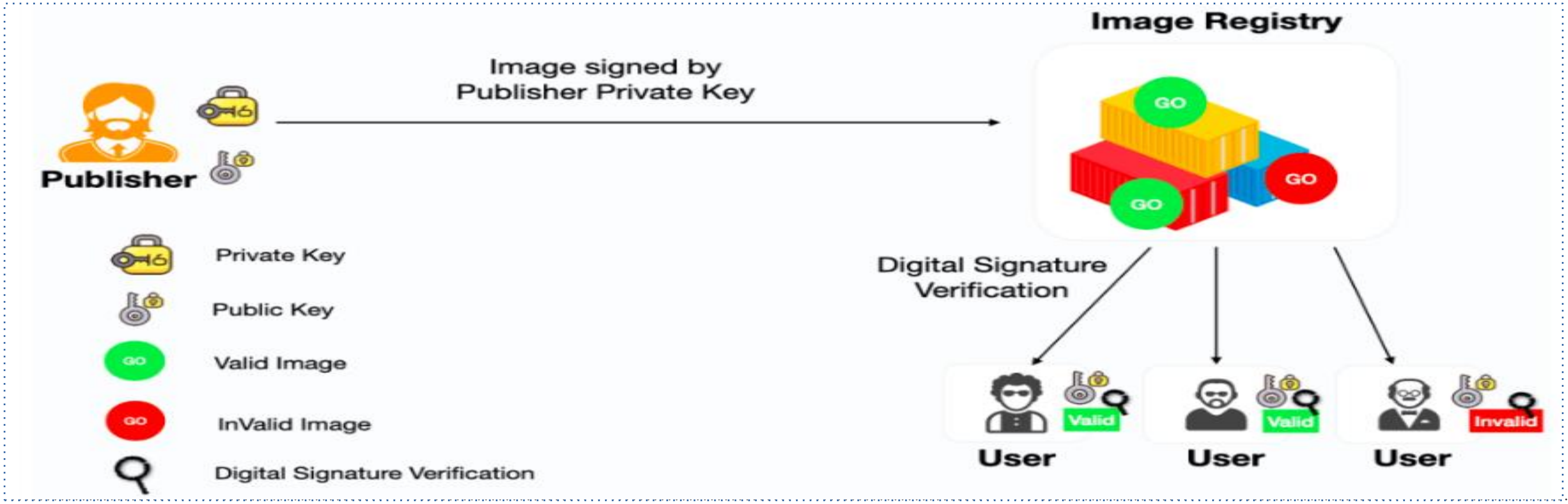
- Image security: Docker images should be built from trusted and verified sources to minimize the risk of including malicious code or vulnerabilities.

- Image scanning: Use container image scanning tools to identify vulnerabilities or insecure configurations in your Docker images.

Docker Security (2/3)

- Why do we need Docker Content Trust:
  + Image Authenticity: Make sure that the images they are pulling and running come from a trusted source.
  + Image Integrity: DCT verifies that the content of an image has not been altered since it was signed.

- What DCT is: a feature specific to Docker that enables the verification of the authenticity and integrity of Docker imag



Docker Security (3/3)

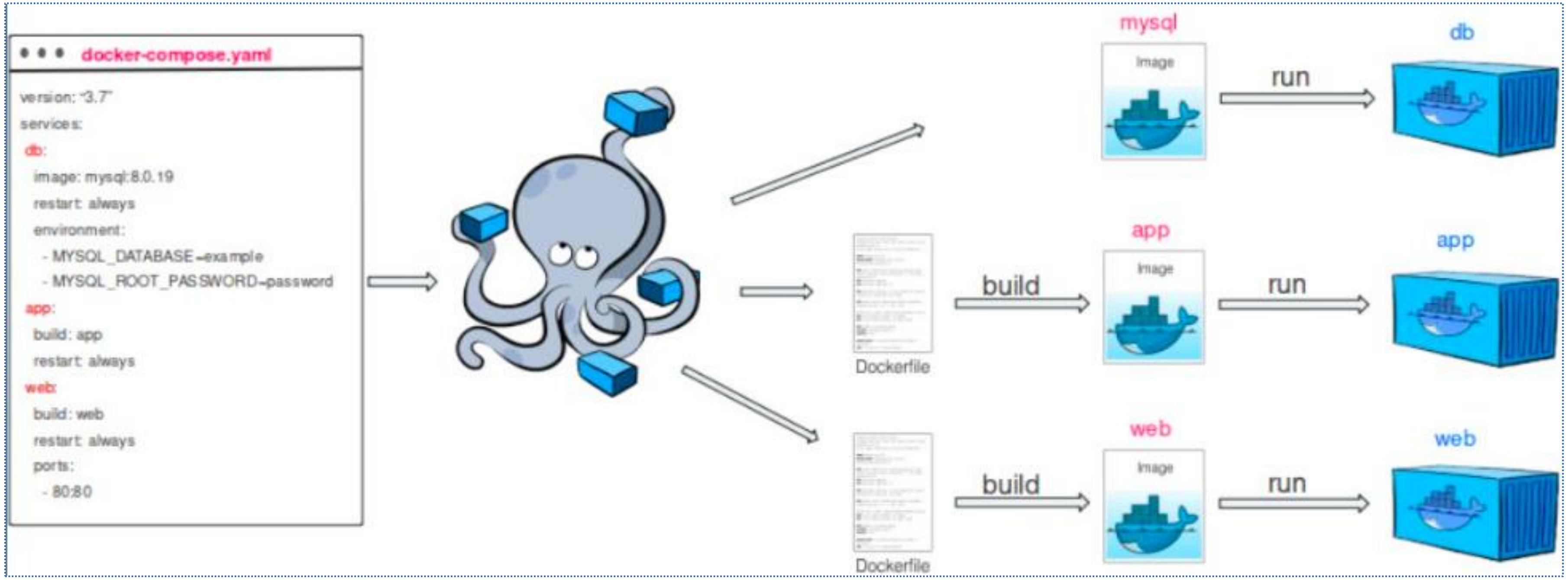
- Docker image vulnerability: Refer to security issues or weaknesses present in Docker images that could potentially be exploited by attackers.

- some popular tools used for Docker image vulnerability scanning:
  + Docker Security Scanning
  + Snyk
  + Tenable.io
  + Trivy

Docker Compose (1/2)

- Docker Compose: manage multi-container Docker applications.

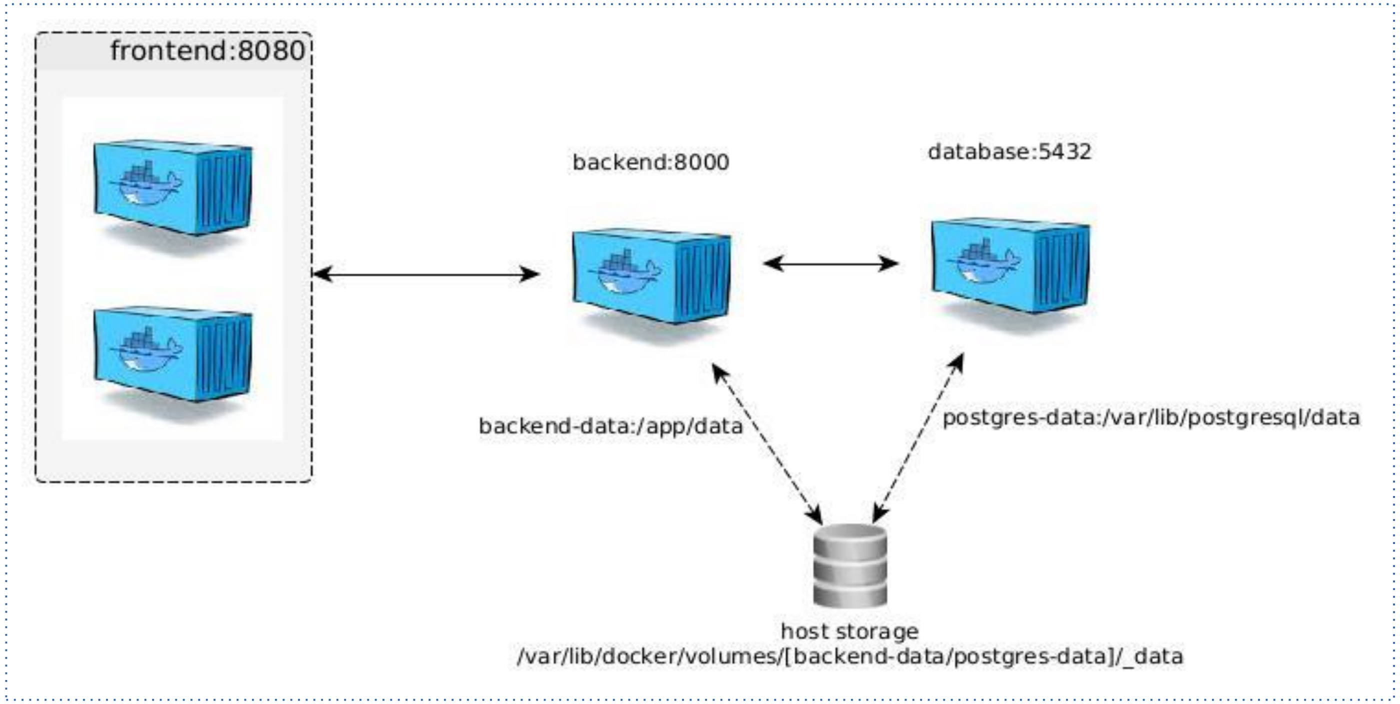
- Benefits:
  + Orchestration of Multiple Containers
  + Declarative Configuration
  + Service Dependencies



docker-compose.yaml

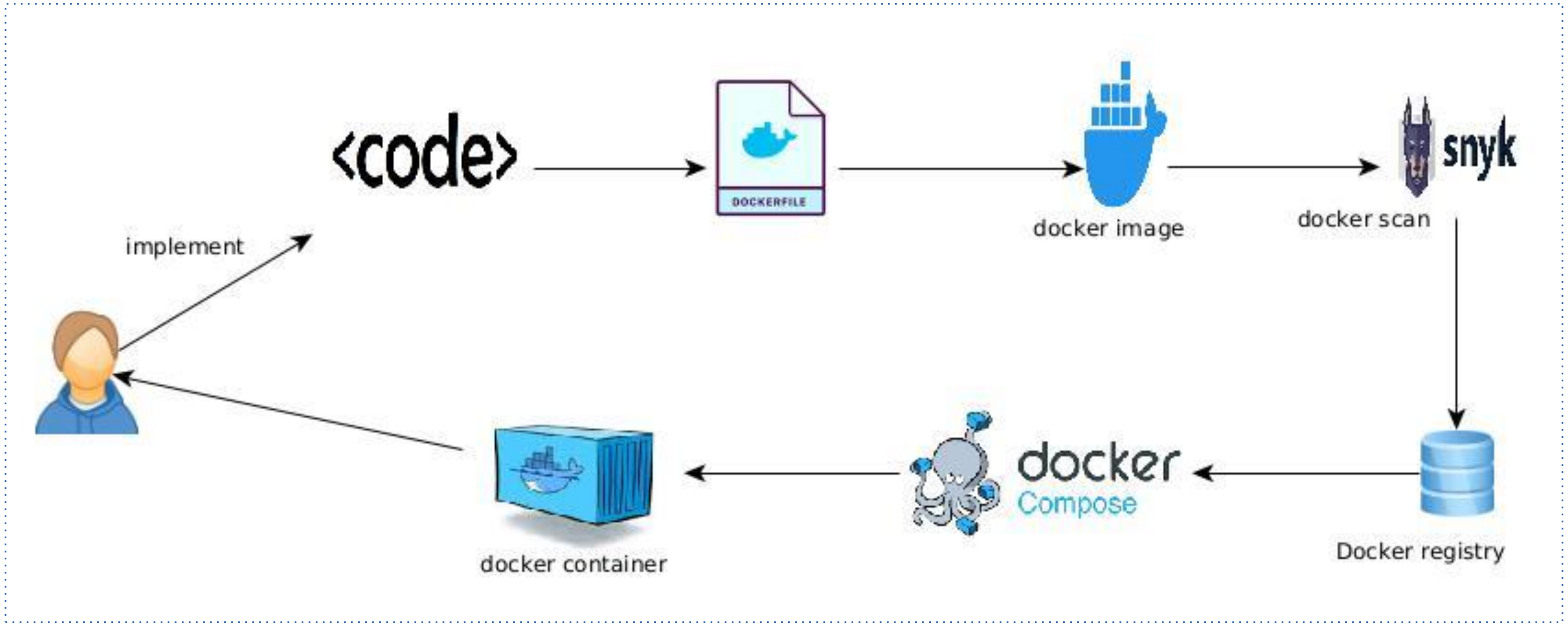
```
version: “3.7”
services:
 db:
  image: mysql:8.0.19
  restart: always
  environment:
   - MYSQL_DATABASE=example
   - MYSQL_ROOT_PASSWORD=password
 app:
  build: app
  restart: always
 web:
  build: web
  restart: always
  ports:
   - 80:80
```

Docker Compose (2/2)


frontend:8080
backend:8000
database:5432
backend-data:/app/data
postgres-data:/var/lib/postgresql/data
host storage
/var/lib/docker/volumes/[backend-data/postgres-data]/_data

```
version: '3'
services:
  frontend:
    image: frontend-image:tag
    ports:
      - 8080:80
    deploy:
      replicas: 2
    environment:
      - FRONTEND_VARIABLE=value
    networks:
      - mynetwork

  backend:
    image: backend-image:tag
    depends_on:
      - database
    environment:
      - BACKEND_VARIABLE=value
    volumes:
      - backend-data:/app/data
    ports:
      - 8000:8000
    networks:
      - mynetwork

  database:
    image: postgres:latest
    environment:
      - POSTGRES_USER=myuser
      - POSTGRES_PASSWORD=mypassword
      - POSTGRES_DB=mydatabase
    volumes:
      - postgres-data:/var/lib/postgresql/data
    ports:
      - 5432:5432
    networks:
      - mynetwork

volumes:
  postgres-data:
  backend-data:

networks:
  mynetwork:
    driver: bridge
```

Docker Workflow



03

Demo

Demo

- Refer to azure-devops-ci-cd/docker-demo/documents/Application_deployed_Docker.pdf at main · nashtech-garage/azure-devops-ci-cd (github.com)

Thank you

Nash
Tech.