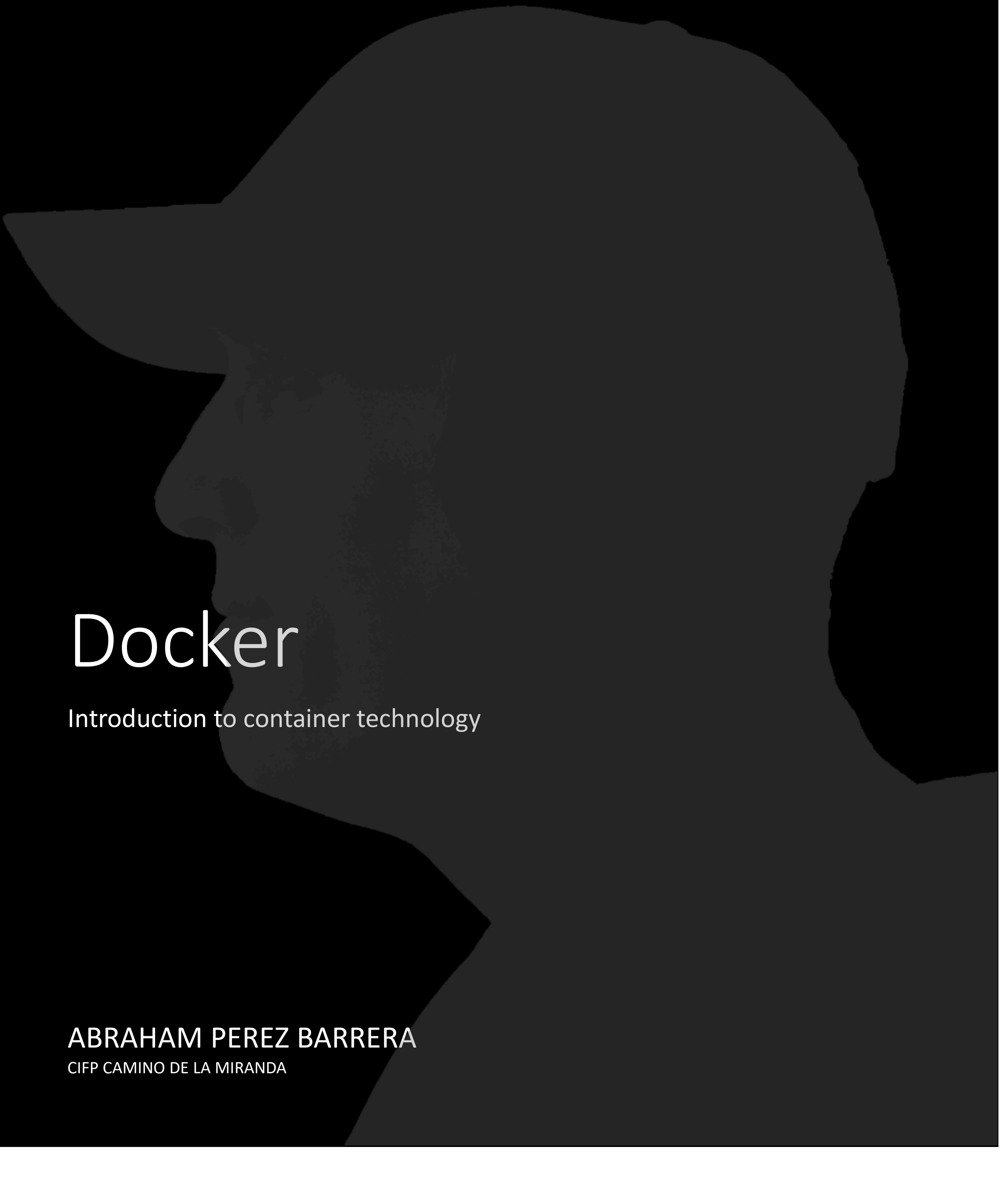


Docker

Introduction to container technology

ABRAHAM PEREZ BARRERA
CIFP CAMINO DE LA MIRANDA

Docker Concepts

Volumes

Volumes allow Docker containers to handle stateful data efficiently and consistently across container restarts.

- **Volumes**: Ideal for persisting data in Docker containers. Docker manages the volume's lifecycle and location, and volumes are independent of the container.
- **Bind Mounts**: Mounts specific host filesystem directories or files into the container, providing direct access but can create dependencies on the host filesystem.
- **Tmpfs Mounts**: Volumes stored in memory for ephemeral data.

Volumes in Docker are a mechanism for persisting and managing data that is used by and shared between Docker containers. When working with Docker containers, you often need to store data that should persist beyond the container's lifecycle (since containers are ephemeral). Volumes are the recommended way to handle such persistent data.

Key Concepts:

1. **Volumes**:
    a. Volumes are directories on the host filesystem or managed by Docker (in Docker's storage driver) that are mounted into the container.
    b. Volumes are **persistent**, meaning they are not deleted when a container is stopped or removed (unlike data in containers' writable layers).
    c. Volumes are ideal for storing data such as databases, logs, configuration files, and other files that need to be persisted across container restarts.
2. **Bind Mounts**:
    a. **Bind mounts** link a file or directory from the host filesystem directly into a container. You specify a full path to the file or directory on the host.
    b. Changes made in the container are reflected in the host file system, and vice versa.
    c. Bind mounts can be useful for development or for accessing specific files on the host system, but they are less portable than volumes because they depend on the host file system.
3. **Tmpfs Mounts**:
    a. **Tmpfs mounts** are used to store data in memory. This data is ephemeral and disappears when the container is stopped.
    b. They are typically used for storing sensitive data or temporary files that don't need to persist.

Volumes in Docker: How They Work

Docker provides a specific command to manage volumes:

- **`docker volume create`**: Create a volume.

- **`docker volume ls`**: List all volumes.
- **`docker volume inspect`**: View details about a specific volume.
- **`docker volume rm`**: Remove a volume.

Volumes are stored in a special directory managed by Docker, usually on the host at `/var/lib/docker/volumes/` (Linux-based systems).

Key Points About Volumes:

1. **Persistence**: Data in volumes persists even when containers are removed, making them ideal for database data or configuration files that need to be preserved.
2. **Isolation**: Volumes can be shared between containers, allowing data to be exchanged or shared in a controlled way.
3. **Backup and Restore**: Volumes can be backed up and restored, making them a good choice for critical data.
4. **Performance**: Volumes generally offer better performance than bind mounts (especially on certain host filesystems) because they are managed by Docker.

Example of Inspecting a Volume:

You can inspect a volume to view detailed information about it:

```
docker volume inspect web_data
```

This would return information such as the volume's mount point on the host system and its configuration.

Networking

Docker networking allows containers to communicate with each other and the outside world, by offering flexible network modes tailored to various use cases (isolated, shared, or connected to external networks).

Docker provides several networking modes, each with a different scope and use case:

Key Docker Networking Types:

1. **Bridge Network (Default)**:
    a. The default network for containers when no other network is specified.
    b. Containers connected to the same bridge network can communicate with each other via their IP addresses.
    c. Typically used for containers running on a single host.
2. **Host Network**:
    a. Containers share the network namespace of the host system.
    b. The container's network interfaces are directly connected to the host's network, so it uses the host's IP address.

    c. This is useful for performance-sensitive applications that need direct access to the host’s network stack.
3. **Overlay Network**:
    a. Used to enable communication between containers across different Docker hosts, often used in Docker Swarm or Kubernetes.
    b. Containers on different machines can communicate as if they were on the same local network.
4. **None Network**:
    a. No networking is enabled for the container.
    b. This isolates the container from any network communication.
5. **Container Network**:
    a. One container can share the network stack of another container. This allows the containers to share the same IP address and network namespace.

Docker Networking Commands:

- `docker network ls`: List available networks.
- `docker network create`: Create a custom network.
- `docker network inspect`: View details of a network.

Here are examples illustrating the different types of Docker networks and how you might use them in different scenarios:

1. Bridge Network (Default)

- **Use Case**: Communication between containers on the same host.
- **Description**: The default network mode for containers. Containers on the same bridge network can communicate with each other using their container names or IP addresses. They can also communicate with the outside world through the host machine’s IP.

Example:

```
docker run -d --name web --network bridge nginx
docker run -d --name db --network bridge postgres
```

In this example, both the web and db containers are connected to the same `bridge` network and can communicate with each other (e.g., the web container can access the database container by its name db).

2. Host Network

- **Use Case**: When you need the container to use the host’s network stack, for high-performance applications or when you want to avoid network isolation between the host and container.
- **Description**: The container shares the network namespace of the host machine, meaning it directly uses the host’s IP and network interfaces. This can be beneficial for certain applications that need direct access to the host's resources.

Example:

```
docker run -d --name web --network host nginx
```

In this case, the `web` container will use the host's network, meaning the container will directly bind to the host's IP address and network interfaces (e.g., `localhost:80`).

3. Overlay Network

- **Use Case**: Communication between containers across different Docker hosts, often used in Docker Swarm or Kubernetes for multi-host container orchestration.
- **Description**: This type of network allows containers on different Docker hosts to communicate with each other as if they were on the same local network. It's typically used in clustered environments.

Example (Docker Swarm): First, initialize a Docker Swarm (if you haven't already):

```
docker swarm init
```

Create an overlay network:

```
docker network create -d overlay my_overlay_network
```

Deploy a service that uses the overlay network:

```
docker service create --name web --network my_overlay_network nginx
```

This creates an `nginx` container in a Docker Swarm cluster, and the container can communicate with other containers connected to the same `my_overlay_network`, even if those containers are running on different machines in the swarm.

4. None Network

- **Use Case**: When you want the container to be completely isolated from any network, essentially preventing it from accessing any external network resources.
- **Description**: The container has no network connectivity. It's useful when you want to isolate a container completely or when it's not needed to access the network at all.

Example:

```
docker run -d --name isolated-container --network none nginx
```

In this case, the `isolated-container` will have no network access, meaning it cannot communicate with other containers or the host machine.

5. Container Network

- **Use Case**: When you want two containers to share the same network stack (IP address, ports, etc.).
- **Description**: A container can use another container's network namespace. This means they share the same IP address and network configuration, which is useful if you want closely coupled applications to share resources directly.

Example:

```
docker run -d --name app1 nginx
docker run -d --name app2 --network container:app1 nginx
```

In this example, `app2` will use the network stack of `app1`, meaning both containers share the same IP address. They can directly communicate over `localhost` using the same ports, as if they were running in the same container.

Summary of Use Cases:

- **Bridge**: Ideal for container-to-container communication on the same host. Default network mode for single-host applications.
- **Host**: Best for applications that need high performance or need to use the host machine's network stack directly.
- **Overlay**: Perfect for multi-host communication in Docker Swarm or Kubernetes for distributed applications.
- **None**: Used for isolating containers that don't need network access.
- **Container**: Useful for containers that need to share the same network stack, typically for tightly coupled applications.

These network types give you flexibility in how you manage and isolate communication between containers, either within a single host or across multiple machines in a Docker Swarm or Kubernetes cluster.

Docker Engine Set UP

To install Docker, we will follow the steps on the official Docker website. Using a repository:

https://docs.docker.com/engine/install/ubuntu/

Install using the apt repository

Before you install Docker Engine for the first time on a new host machine, you need to set up the Docker repository. Afterward, you can install and update Docker from the repository.

Set up Docker's apt repository.

content_copy

# Add Docker's official GPG key:

sudo apt-get update

sudo apt-get install ca-certificates curl gnupg

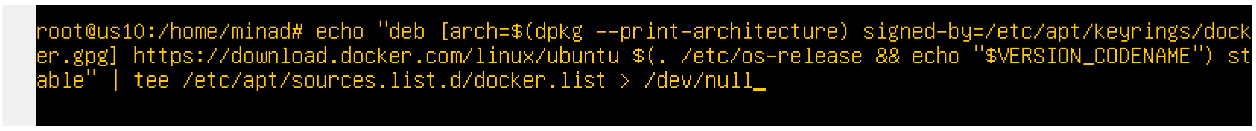
sudo install -m 0755 -d /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Add the repository to Apt sources:

echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo "$ UBUNTU_CODENAME") stable"



```
root@us10:/home/minad# echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/dock
er.gpg] https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo "$VERSION_CODENAME") st
able" | tee /etc/apt/sources.list.d/docker.list > /dev/null_
```

sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

Install the Docker packages.

Latest Specific version

To install the latest version, run:

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

To check if Docker is installed correctly

Verify that the Docker Engine installation is successful by running the hello-world image.

sudo docker run hello-world

This command downloads a test image and runs it in a container. When the container runs, it prints a confirmation message and exits.

You have now successfully installed and started Docker Engine.

Once we have installed Docker, we can check that it works by running the HelloWord image:

>>sudo docker run hello-world.

```
docker run -it --name myContainer_name -v /hostvolume:/containervolume
image_name

-v [host directory]:[container directory]
```

DockerFile vs Docker Compose

- **Dockerfile**: Defines how to build a Docker image.

- **Docker Compose**: Defines how to run and manage multi-container applications.

Dockerfile

A Dockerfile is essentially a script for creating Docker images. It contains a series of instructions that Docker uses to build an image. Think of it like a recipe:

- **Instructions** Each line in a Dockerfile provides an instruction, such as which base image to use, what software to install, and what commands to run.

When you build an image, Docker reads the Dockerfile, executes the instructions, and produces a new image. This image can then be run as a container.

Example of a Dockerfile:

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim-buster

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80
```

~~~~
# Run app.py when the container launches
CMD ["python", "app.py"]
```
~~~~

Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services:

- **Services**: In Compose, each containerized component of your application is defined as a service.

- **Usage**: You describe the configuration of your application in a `docker-compose.yml` file. This includes details like how many instances of each service to run, networking options, and volumes. Running `docker-compose up` then starts all the services as per your configuration.

Example of a docker-compose.yml file:

```
version: '3'
services:
  web:
    # Use the Dockerfile in the current directory to build the image
    build: .
    ports:
      - "5000:80"
  redis:
    # Use the official Redis image from the Docker Hub
    image: "redis:alpine"
```

Comparison

- **Scope**: Dockerfile is about building a single container image, whereas Compose is about running multiple containerized services together.

- **Configuration**: Dockerfile uses a simple, script-like syntax to define how an image is built. Compose uses YAML to define how different services interact.

- **Usage**: Dockerfiles are used during the build stage (`docker build`), and Compose is used during the deployment stage (`docker-compose up`).

DockerFile

A **Dockerfile** is a text file that contains instructions on how to build a Docker image. Below is a simple breakdown of the main parts of a Dockerfile:

Simple Example Breakdown

```
FROM ubuntu:20.04
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y bind9
```

```
COPY ./named.conf /etc/bind/named.conf
EXPOSE 53/tcp 53/udp
CMD ["named", "-g", "-c", "/etc/bind/named.conf"]
```

- **FROM**: Start from the ubuntu:20.04 image.
- **ENV**: Set an environment variable to avoid interactive prompts during installation.
- **RUN**: Update the package list and install bind9.
- **COPY**: Copy the custom named.conf configuration file into the container.
- **EXPOSE**: Indicate that the container will listen on port 53 for DNS traffic (both TCP and UDP).
- **CMD**: When the container starts, run named (BIND9) with the custom configuration.

1. FROM

- **Purpose**: Specifies the base image to use for your Docker image.
- **Example**: `FROM ubuntu:20.04`
- **Explanation**: This tells Docker to start with the ubuntu:20.04 image (a minimal installation of Ubuntu version 20.04).

2. ENV

- **Purpose**: Sets environment variables inside the container.
- **Example**: `ENV DEBIAN_FRONTEND=noninteractive`
- **Explanation**: This prevents prompts during package installations (commonly used in Debian/Ubuntu-based images to prevent interactive prompts).

3. RUN

- **Purpose**: Executes commands inside the container during the image build process. The **`RUN`** instruction allows you to perform operations like installing software or updating the system. (The **`RUN`** instruction runs commands in the container's shell and commits the resulting changes to the image.)
- **Example**: `RUN apt-get update && apt-get install -y bind9`
- **Explanation**: This installs the bind9 package.

4. COPY

- **Purpose**: Copies files or directories from the **host machine** (where the Docker build is being run) into the **container** at the specified location. It is used to bring external files into the image, like application code, configuration files, or assets.
- **Example**: `COPY ./named.conf /etc/bind/named.conf`
- **Explanation**: This copies the named.conf file from your local machine into the container's /etc/bind/ directory.

5. EXPOSE

- **Purpose**: Tells Docker which ports the container will listen on.
- **Example**:

```
EXPOSE 53/tcp
EXPOSE 53/udp
```

- **Explanation**: This indicates that the container will use port 53 for both TCP and UDP (common for DNS servers).

6. CMD

- **Purpose**: Defines the default command to run when the container starts.
- **Example**: `CMD ["named", "-g", "-c", "/etc/bind/named.conf"]`
- **Explanation**: This sets the command to run BIND9 with specific options when the container starts. The command in CMD is executed when you run the container.

7. WORKDIR (Optional)

- **Purpose**: Sets the working directory for subsequent instructions.
- **Example**: `WORKDIR /usr/local/bin`
- **Explanation**: Any commands (like RUN or CMD) following this instruction will be executed in the specified directory.

Summary of Common Instructions

1. **FROM**: Base image to start from.
2. **RUN**: Execute commands during image build.
3. **COPY**: Copy files from the host machine into the container.
4. **CMD**: Default command to run when the container starts.
5. **ENV**: Set environment variables.
6. **EXPOSE**: Indicate which ports the container listens on.

RUN vs COPY

In essence, `RUN` is for executing commands that alter the image (like installing software or modifying files), while `COPY` is for adding static files or directories from the host machine into the container.

Feature	`RUN`	`COPY`
Purpose	Executes commands inside the container	Copies files/directories from the host to the container
Effect	Changes the image (installs packages, updates, etc.)	Adds files or directories to the image
Usage	Used for installing software, running scripts, or modifying the image	Used for including application files, configs, or other assets in the container
Syntax	`RUN <command>`	`COPY <src> <dest>`

Feature	*RUN*	*COPY*
Example	`RUN apt-get install -y curl`	`COPY ./app /usr/src/app`
Build Process	Creates layers by executing commands	Creates layers by copying files

Example Dockerfile

Tomcat Image

```
FROM tomcat:8.0-alpine
LABEL maintainer=”deepak@softwareyoga.com”
ADD sample.war /usr/local/tomcat/webapps/
EXPOSE 8080
CMD [“catalina.sh”, “run”]
```

For this example we will need an app call sample.war; take a look at the practice.

Bind9 Image

Basic Dockerfile to create a DNS (such as for BIND9) image using custom configuration files.

```
# Use an official base image with BIND9 (or other DNS server) as the
starting point
FROM ubuntu:20.04

# Set environment variables for non-interactive installations
ENV DEBIAN_FRONTEND=noninteractive

# Update and install necessary packages (e.g., BIND9 for DNS server)
RUN apt-get update && \
    apt-get install -y bind9 bind9utils bind9-doc && \
    rm -rf /var/lib/apt/lists/*

# Create directories for configuration files (this is where your custom
files will go)
RUN mkdir -p /etc/bind

# Copy your custom configuration files into the image
# Assuming you have 'named.conf' and zone files on the host
COPY ./named.conf /etc/bind/named.conf
COPY ./zones/ /etc/bind/zones/

# Expose DNS ports
EXPOSE 53/udp
EXPOSE 53/tcp

# Set the command to start the DNS service (for example, BIND9)
CMD ["named", "-g", "-c", "/etc/bind/named.conf"]
```

Steps to Build and Use the Image

1. Prepare your configuration files:

- named.conf: Is the main configuration file for BIND9.

- Zone files: Place them inside the `zones/` directory (adjust as per your requirements).

2. Directory Structure:

Your project structure could look like this:

```
   my-dns-server/
   ├── Dockerfile
```

```
    ├── named.conf
    └── zones/
        └── db.example.com
        └── db.127
```

3. Build the Docker image:

```
docker build -t my-dns-server .
```

4. Run the container:

```
    docker run -d --name my-dns-container -p 53:53/udp -p
```

This setup uses BIND9 as the DNS server, but the same principles can apply to other DNS server software if you're using a different one.

FROM tomcat:8.0-alpine

LABEL maintainer="deepak@softwareyoga.com"

ADD sample.war /usr/local/tomcat/webapps/

EXPOSE 8080

CMD [“catalina.sh”, “run”]

Environment variables

Environment variables in Docker help you configure containers flexibly and securely. They can be set in the Dockerfile, passed at runtime using docker run, or loaded from .env files.

Are used to define dynamic values that can configure or influence the behavior of containers. They allow you to customize the execution of applications inside the container and provide a flexible way to manage settings that might vary across different environments (e.g., development, testing, production).

Common Uses of Environment Variables in Docker:

1. **Configuration and Customization**:
   - Environment variables can be used to configure settings for applications inside the container (like database URLs, API keys, etc.).
   - Example: Setting a database connection URL:

```
ENV DATABASE_URL="mysql://user:password@localhost/dbname"
```

2. **Avoid Hardcoding Sensitive Information**:
   - Environment variables are often used to pass sensitive information such as API keys, passwords, or tokens to containers, avoiding the need to hardcode them in source code or Dockerfiles.
     - Example: `ENV API_KEY="your_api_key_here"`
3. **Making Applications Portable**:
   - By using environment variables, the same container image can be reused in different environments, such as development, testing, or production, without changing the configuration.
   - Example: `ENV APP_MODE="production"`
4. **Runtime Configuration (with docker run)**:
   - You can pass environment variables at runtime when starting a container with the docker run command using the -e or --env flag.
   - Example: `docker run -e "APP_SECRET_KEY=your_secret_key" my-app`
5. **Setting Default Values in Dockerfiles**:
   - In a Dockerfile, you can use ENV to set default values for environment variables that the container will use when it starts.
   - Example: `ENV LOG_LEVEL="info"`
6. **Container Behavior and Control**:
   - Environment variables can help define container behavior, such as whether a service should run in foreground or background, the port to listen on, etc.

    - Example: `ENV APP_PORT=8080`

Setting Environment Variables in Docker:

1. **In a Dockerfile**: You can set environment variables using the ENV instruction in the Dockerfile. `ENV MY_VAR="some_value"`

This sets MY_VAR to "some_value" inside the container, and it will be available to any process running inside the container.

2. **At Runtime**: You can pass environment variables when running a container using the -e or --env flag with docker run.

```
docker run -e "MY_VAR=some_value" my-container
```

This will set MY_VAR to "some_value" inside the running container.

3. **Using .env Files**: You can also use an .env file to load environment variables into a Docker container. Docker Compose, for example, can load environment variables from a .env file.

Example .env file:

```
MY_VAR=some_value
DATABASE_URL=mysql://user:password@localhost/dbname
```

Then, in your Docker Compose file or when running docker run, you can use the --env-file flag to load those variables:

```
docker run --env-file .env my-container
```

Example of Using Environment Variables:

Dockerfile:

```
FROM node:14

# Set environment variables
ENV NODE_ENV=production
ENV APP_PORT=3000

# Install dependencies
WORKDIR /app
COPY . .
RUN npm install

# Run the app
CMD ["node", "app.js"]
```

Running the Container:

```
docker run -e "NODE_ENV=development" -e "APP_PORT=4000" my-node-app
```

In this example:

- NODE_ENV is set to "development", overriding the default "production" value in the Dockerfile.
- APP_PORT is set to "4000", changing the port on which the app listens.

Key Benefits of Environment Variables:

1. **Portability**: Makes it easy to change configurations without modifying the Dockerfile or code.

2. **Security**: Allows you to keep sensitive information like passwords, keys, etc., out of your code and version control.

3. **Flexibility**: Enables dynamic configuration depending on the environment (e.g., using different settings for development vs. production).

Common Docker Environment Variables:

- PATH: The system's search path for executable programs.
- HOME: The home directory for the current user.
- USER: The current logged-in user.
- PWD: The current working directory.
- HOSTNAME: The name of the container or host machine.

Docker Compose

Docker Compose is a tool used for defining and running multi-container Docker applications. It allows you to configure the services your application needs (like databases, caches, and web servers) in a single file, typically named docker-compose.yml, and then spin them up with a single command. This is particularly useful for applications that require multiple services to work together.

Key Features of Docker Compose:

1. **Multi-container setup**: You can define multiple containers that your application needs (e.g., frontend, backend, database, cache, etc.) in one YAML file.
2. **Configuration**: You can specify how containers should be built, networks they should connect to, and volumes they should mount, all in the `docker-compose.yml` file.
3. **Networking**: Docker Compose automatically sets up a network for your containers, allowing them to communicate with each other.
4. **Environment variables**: You can define environment variables and configurations that the services need.
5. **Scaling services**: Compose allows you to scale services (e.g., running multiple instances of a service) with a single command.
6. **Development, testing, and production environments**: It helps create consistent environments across different stages of development.

Basic Workflow:

1. **Create a *`docker-compose.yml`* file**: Define all your application services, networks, and volumes.
2. **Run *`docker-compose up`***: This command builds, starts, and links all the defined services.

3. **Run *`docker-compose down`***: Stops and removes all containers, networks, and volumes defined in the `docker-compose.yml` file.

Example of a `docker-compose.yml`:

```
version: "3.8"
services:
  web:
    image: nginx
    ports:
      - "80:80"
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: example
```

```
In this example, you have two services: a web service running Nginx and a
database service using PostgreSQL. Docker Compose will manage both
services together and ensure they run in the same network, making it easy
to coordinate their interaction.
```

Common Commands:

- `docker-compose up`: Starts the containers.
- `docker-compose down`: Stops and removes containers, networks, and volumes.
- `docker-compose logs`: View the logs of all services.
- `docker-compose ps`: List the running containers.

Docker Compose simplifies the management of complex applications, especially when dealing with multiple interconnected services.

Another Example

```
version: "3.8"
services:
 bind9:
  image: internetsystemsconsortium/bind9:9.16
  container_name: bind9-server
  volumes:
   - ./named.conf:/etc/bind/named.conf
   - ./zones:/var/cache/bind
  ports:
   - "53:53"
   - "53:53/udp"
  environment:
   - ALLOW_QUERY=any
  restart: always
```

Explanation:

- **`image`**: The official BIND9 image is being used (`internetsystemsconsortium/bind9:9.16`).

- **`container_name`**: This gives the container a name (`bind9-server`) for easier reference.
- **`volumes`**:
  - `./named.conf:/etc/bind/named.conf`: This mounts a local `named.conf` file to configure BIND9.
  - `./zones:/var/cache/bind`: This mounts a directory with zone files for BIND9.
- **`ports`**: BIND9 listens on port 53 for DNS queries. Both TCP and UDP traffic on port 53 are forwarded to the container.
- **`environment`**: The `ALLOW_QUERY=any` environment variable allows DNS queries from any source (you can adjust this based on your use case).
- **`restart`**: Ensures the container restarts automatically if it fails.

Notes:

1. **Configuration Files**: The `named.conf` file is the main configuration file for BIND9. You'll need to create this file and any DNS zone files in the local directory where you're running Docker Compose (or point to an existing configuration if you have one).
2. **DNS Zone Files**: The `zones` directory should contain DNS zone files (e.g., `db.example.com`).

Using Volumes in `docker-compose.yml`

In Docker Compose, volumes are defined within the `volumes` section. You can either define named volumes or use bind mounts. Here's an example:

Example with Volumes

```
version: "3.8"
services:
  web:
    image: nginx
    volumes:
      - web_data:/usr/share/nginx/html
    ports:
      - "80:80"

  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: example
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  web_data:
  db_data:
```

Explanation of Volumes in This Example:

1. **web Service**:
   a. **`volumes`**: The web service uses a named volume `web_data` to persist the web content. The volume is mounted at `/usr/share/nginx/html` in the container, which is where Nginx serves the web content.
   b. **`volumes` declaration**: The `volumes` section at the bottom creates the named volume `web_data`.
2. **db Service**:
   a. **`volumes`**: The db service uses the named volume `db_data` to persist the PostgreSQL database data. The volume is mounted at `/var/lib/postgresql/data`, which is where PostgreSQL stores its data.
   b. This ensures that even if the container is removed or restarted, the database data will persist.
3. **`volumes` Section**:
   a. This section defines the named volumes `web_data` and `db_data`, which are managed by Docker.

Bind Mount Example

If you wanted to use a bind mount instead of a named volume (for example, to mount a specific directory from your host machine into a container), the configuration would look like this:

```
version: "3.8"
services:
  web:
    image: nginx
    volumes:
      - ./my_local_web_content:/usr/share/nginx/html
    ports:
      - "80:80"
```