Mohammad Saeed Pourmazar

https://github.com/MohammadSaeedPourmazar

https://gitlab.com/MohammadSaeedPourmazar

https://medium.com/@MohammadSaeedPourmazar

https://dev.to/MohammadSaeedPourmazar

https://www.youtube.com/@MohammadSaeedPourmazar

https://www.instagram.com/MohammadSaeedPourmazar

https://www.facebook.com/MohammadSaeedPourmazar

https://www.linkedin.com/in/MohammadSaeedPourmazar/

https://orcid.org/0009-0008-9383-419X

Docker Tutorial

Introduction

What is Docker?

Docker is an open-source platform that automates the deployment, scaling, and management of applications by using containerization technology. It allows developers to package applications and their dependencies into containers, which are portable, consistent, and isolated environments.

Key Concepts:

Containerization

Containers are lightweight and efficient, containing everything needed to run an application, such as code, libraries, system tools, and settings. Containers run consistently across any environment, whether it's a developer's local machine, a testing environment, or a production server.

Docker Engine

The core component of Docker is the Docker Engine, a client-server application that enables the creation, running, and management of containers. It consists of:

- ***Docker Daemon (dockerd)**: The background service responsible for managing containers.*

- ***Docker CLI**: The command-line interface used to interact with the Docker daemon.*

Docker Images

An image is a read-only template that defines a container. It includes the application code, runtime, libraries, and dependencies. Images are portable and can be shared across systems.

Docker Containers

A container is a running instance of an image. It is isolated from the host and other containers but can share the operating system kernel.

Dockerfile

A Dockerfile is a script containing a series of instructions to build a Docker image. It defines the environment in which the application will run.

Docker Hub

Docker Hub is a cloud-based registry service for storing and sharing Docker images. It allows users to upload, download, and share images with others.

Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. With a simple YAML file, users can specify how to configure and start multiple containers that work together.

Docker Swarm

Docker Swarm is a native clustering and orchestration tool in Docker. It allows users to manage a group of Docker engines (called a swarm) as a single virtual system, making it easier to scale applications and maintain high availability.

Why Use Docker?

1. Consistency Across Environments:

- ***Eliminate "Works on My Machine" Issues**: Docker containers ensure that an application runs the same way across all environments—whether it's on a developer's laptop, a test server, or a production system. This consistency eliminates issues related to differing configurations or software versions.*

2. Portability:

- ***Cross-Platform Compatibility**: Docker containers can run on any platform that supports Docker, including Windows, Linux, and macOS. This portability makes it easy to move applications between various environments without worrying about compatibility.*

3. Isolation:

- ***Separation of Concerns**: Containers provide a lightweight form of isolation between applications. Each container has its own filesystem, libraries, and dependencies, so applications running in separate containers don't interfere with each other. This is particularly useful when running multiple applications or services on the same system.*

- ***Security**: Containers help isolate security vulnerabilities, reducing the risk of cross-application interference or attacks.*

4. Resource Efficiency:

- ***Lightweight and Fast**: Docker containers share the host system's kernel and are much more lightweight than traditional virtual machines. They start quickly and use fewer system resources, making them more efficient in terms of both memory and processing power.*

5. Scalability:

- ***Easy to Scale Applications**: Docker makes it easier to scale applications both vertically (adding more resources to a container) and horizontally (spinning up more containers). Tools like Docker Swarm and Kubernetes help automate the scaling process, making it simple to add or remove containers based on demand.*

6. Version Control and Rollback:

- ***Track Application Versions**: Docker images are versioned, which means you can easily roll back to a previous version of an application or update it to a new one. This provides version control at the container level, enabling easy updates, testing, and rollbacks.*

7. Simplified CI/CD Integration:

- ***Seamless Continuous Integration and Delivery (CI/CD)**: Docker fits perfectly into modern CI/CD pipelines. Developers can build and test their code in containers, which ensures that the same container can be used in the development, testing, and*

production stages. This speeds up delivery cycles and reduces errors related to environment mismatches.

8. Microservices Architecture:

- ***Support for Microservices**: Docker is ideal for applications designed with a microservices architecture, where each service runs in its own container. This enables services to be developed, deployed, and scaled independently, promoting faster development cycles and better fault isolation.*

9. Simplified Dependency Management:

- ***No Dependency Conflicts**: With Docker, you can bundle your application with all the necessary dependencies, libraries, and tools it needs to run. This reduces the risk of dependency conflicts and version mismatches.*

10. Faster Development and Deployment:

- ***Rapid Prototyping and Testing**: Since containers start quickly and can be spun up in seconds, developers can iterate rapidly, test features, and deploy applications faster. This accelerates the development cycle and improves time-to-market for new features.*

11. Community and Ecosystem:

- ***Extensive Support and Resources**: Docker has a large, active community and a vast ecosystem of tools, libraries, and pre-built images available in Docker Hub. This makes it easier to get started with Docker and find support when needed.*

12. Cloud Compatibility:

- ***Seamless Cloud Integration**: Docker works seamlessly with cloud platforms like AWS, Azure, and Google Cloud. You can deploy containers directly to the cloud or use cloud-based container orchestration services like Kubernetes to manage your applications.*

13. Environment Standardization:

- ***Unified Development Environment****: With Docker, developers can standardize their environments. The configuration and dependencies of the application are declared in the Dockerfile, ensuring consistency and reducing the number of environment-related bugs.*

14. Improved DevOps Collaboration:

- ***Better Collaboration Between Developers and Operations****: Docker makes it easier for developers to package their applications and for operations teams to deploy them. It bridges the gap between development and operations, improving collaboration and ensuring smoother workflows.*

Install Docker On Linux

System Requirements:

Before installing Docker, ensure your system meets these requirements:

- *Ubuntu, Debian, Fedora, or CentOS*
- *At least 4GB RAM*
- ***Root** or **sudo** privileges*

Install Docker:

1. Update your package index:

sudo apt update

This ensures your system knows about the latest versions of packages and their dependencies available from the configured repositories.

- ***sudo**: Runs the command with superuser (root) privileges.*
- ***apt**: The package manager used to handle software installation.*
- ***update**: Refreshes the list of available packages and their versions but does not install or upgrade any packages.*

2. Install dependencies:

sudo apt install -y ca-certificates curl gnupg

Installs three essential packages used for secure communications and software management

- ***sudo**: Runs with administrative privileges.*
- ***apt install**: Installs one or more packages.*
- ***-y**: Automatically answers "yes" to any prompts during installation.*

- ***ca-certificates**: Contains digital certificates used to verify HTTPS connections.*
- ***curl**: A command-line tool to transfer data from or to a server, often used with APIs or downloads.*
- ***gnupg**: GNU Privacy Guard, used for secure communication and data encryption, including verifying software signatures.*

3. Add Docker's official GPG key:

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

Used to create a directory for storing GPG keyrings used by APT (Advanced Package Tool) in a secure and standardized way.

- ***sudo**: Run with superuser privileges.*
- ***install**: A command used not only for copying files but also for creating directories with specific permissions.*
- ***-m 0755**: Sets the permissions to rwxr-xr-x:*
  - *Owner can read, write, and execute.*
  - *Group and others can read and execute.*
- ***-d**: Tells install to create a directory.*
- ***/etc/apt/keyrings**: The directory being created.*

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo tee /etc/apt/keyrings/docker.asc > /dev/null

Downloads Docker's GPG key and securely saves it to your APT keyrings directory.

- ***curl**: Tool to transfer data from a URL.*
- ***-f**: Fail silently on server errors.*
- ***-s**: Silent mode (no progress bar).*
- ***-S**: Shows errors even in silent mode.*
- ***-L**: Follows redirects.*
- ***https://download.docker.com/linux/ubuntu/gpg**: The official Docker GPG key URL.*
- ***| sudo tee /etc/apt/keyrings/docker.asc**: Pipes the downloaded key and saves it with superuser privileges to /etc/apt/keyrings/docker.asc.*
- ***> /dev/null**: Suppresses the output of tee (i.e., hides the printed key from your terminal).*

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

Sets read permissions for all users on the Docker GPG key file you just saved.

- ***chmod**: Command to change file permissions.*
- ***a+r**: Grants read permission (r) to all users (a = user, group, others).*
- ***/etc/apt/keyrings/docker.asc**: Path to the Docker GPG key.*

4. Set up the repository:

echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

Adds the official Docker APT repository to your system so you can install Docker packages via apt.

- ***echo "deb [...] https://download.docker.com/linux/ubuntu ... stable"**: Constructs the repository entry.*
- ***arch=$(dpkg --print-architecture)**: Automatically sets your system's architecture (e.g., amd64, arm64).*
- ***signed-by=/etc/apt/keyrings/docker.asc**: Tells APT to use the Docker GPG key you previously added for verifying packages.*
- ***$(lsb_release -cs)**: Automatically inserts your Ubuntu codename (e.g., focal, jammy, etc.).*
- ***| sudo tee /etc/apt/sources.list.d/docker.list**: Saves the constructed repository line to a new Docker-specific list file.*
- ***> /dev/null**: Silences the output.*

5. Install Docker:

sudo apt update

This ensures your system knows about the latest versions of packages and their dependencies available from the configured repositories.

- ***sudo**: Runs the command with superuser (root) privileges.*

- ***apt***: *The package manager used to handle software installation.*
- ***update***: *Refreshes the list of available packages and their versions but does not install or upgrade any packages.*

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

Installs the Docker Engine and related tools from the official Docker APT repository.

- ***docker-ce***: *Docker Community Edition – the core Docker engine.*
- ***docker-ce-cli***: *Command Line Interface for Docker.*
- ***containerd.io***: *Core container runtime used by Docker.*
- ***docker-buildx-plugin***: *Plugin for advanced image building with* ***buildx*** *(multi-platform, caching, etc.).*
- ***docker-compose-plugin***: *Plugin for using docker compose (v2 syntax, replaces docker-compose binary).*

Verify installation:

docker --version

(Optional) Allow running Docker without sudo:

sudo usermod -aG docker $USER

Adds your current user to the docker group, allowing you to run Docker commands without using sudo.

- ***sudo***: *Run as superuser.*
- ***usermod***: *Modify a user account.*
- ***-aG***: *Append (-a) the user to the specified group (-G).*
- ***docker***: *The group that has permission to run Docker commands.*
- ***$USER***: *Environment variable for your current username.*

** This change won't take effect until you log out and log back in **

or you can run:

newgrp docker

To apply the new group permissions in your current session.

*you've switched your current shell session to include membership in the **docker** group.*

** This means you can now run Docker commands without log out and log back in **

Running and Managing Containers

A container is a lightweight, portable unit that runs an application and its dependencies in an isolated environment.

Running and managing containers in Docker involves starting, stopping, monitoring, and maintaining containerized applications.

1. Running Containers

Basic Syntax

docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run an Nginx Container

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

After running this, you can access the default Nginx web page by opening your browser

- ***docker run****: Starts a new container.*
- ***-d****: Runs the container in detached mode (in the background).*
- ***--name mynginx****: Names the container mynginx.*
- ***-p 8080:80****: Maps port 8080 on your host to port 80 on the container (port 80 is the default HTTP port used by the Nginx web server).*
- ***nginx****: Specifies the image to use, in this case the official Nginx image from Docker Hub.*

Interactive Mode

docker run -it ubuntu /bin/bash

This launches an interactive Bash shell in a new Ubuntu container. You'll be "inside" the container's terminal, where you can run Ubuntu commands like ***apt update****,* ***ls****,* ***pwd****, etc.*

- ***docker run****: Starts a new container.*
- ***-it****: Combines two options:*
- ***-i****: Keeps STDIN open (interactive).*
- ***-t****: Allocates a pseudo-TTY (terminal).*
- ***ubuntu****: Specifies the Docker image to use (official Ubuntu image from Docker Hub).*
- ***/bin/bash****: Runs the Bash shell inside the container.*

2. Starting, Stopping, and Restarting Containers

docker start <container_name or ID>

docker stop <container_name or ID>

docker restart <container_name or ID>

docker pause <container_name or ID>

docker unpause <container_name or ID>

- ***docker start****: Starts a stopped container.*
- ***docker stop****: Stops a running container.*
- ***docker restart****: Stops and then starts the specified container.*
- ***docker pause****: Suspends all processes in the specified container using the **cgroups freezer**.*
- ***<container_name or ID>****: The name or ID of the container you want to start (e.g., mynginx or d9b100f2f636).*

3. Inspecting and Viewing Containers

List Containers

docker ps # Running containers

docker ps -a # All containers (including stopped)

- ***docker ps:** Lists all running containers.*
- ***-a (or --all)****: Shows all containers, including those that are stopped.*

Key columns:

- ***CONTAINER ID**: Unique identifier for the container (first 12 characters).*
- ***IMAGE**: The image used to create the container (e.g., nginx).*
- ***COMMAND**: The command running inside the container.*
- ***CREATED**: How long ago the container was created.*
- ***STATUS**: Current state (e.g., "Up 2 hours" means the container is running).*
- ***PORTS**: Exposed ports, mapped between the container and the host.*
- ***NAMES**: The container's name (e.g., mynginx).*

View Logs

docker logs <container_name>

- ***docker logs**: Fetches the logs of a specified container.*
- *<container_name>: The name or ID of the container you want to view the logs for.*

Inspect Details

docker inspect <container_name>

- ***docker inspect**: Provides detailed information about a container, including its configuration, state, network settings, mounted volumes, and more.*
- *<container_name>: The name or ID of the container you want to inspect.*

Key columns:

- ***ID**: Container ID.*
- ***Name**: The container's name.*
- ***State**: Current state of the container (e.g., running, stopped).*
- ***Config**: Configuration settings like the image used, environment variables, and command.*
- ***Mounts**: Details of any mounted volumes or bind mounts.*
- ***NetworkSettings**: Network configurations, including IP addresses and port mappings.*

4. Executing Commands Inside Containers

docker exec -it <container_name> <command>

- ***docker exec****: Executes a command inside a running container.*
- ***-it****: Combines two options:*
- ***-i****: Keeps STDIN open (interactive).*
- ***-t****: Allocates a pseudo-TTY (terminal).*
- *<container_name>: The name or ID of the container where you want to run the command.*
- ***<command>****: The command you want to execute inside the container.*

5. Removing Containers

Stop and Remove

docker rm -f <container_name>

- ***docker rm****: Removes a stopped container from your system.*
- ***-f (or --force)****: Forces the removal of a running container. If the container is running, it will be stopped before being removed.*
- ***<container_name>****: The name or ID of the container you want to remove.*

Remove All Stopped Containers

docker container prune

- ***docker container prune****: Removes all stopped containers on your system to free up space.*
- *It will prompt you for confirmation before proceeding, asking if you're sure you want to delete the stopped containers.*

6. Managing Data with Volumes

Run with a Volume

docker run -d --name myapp -v myvolume:/app/data ubuntu

This command will:

1. *Start an Ubuntu container named myapp.*
2. *Mount the volume myvolume from your host machine to the container's /app/data directory.*
3. *Allow you to persist data in myvolume, even if the container is removed.*

- ***docker run**: Starts a new container.*
- ***-d**: Runs the container in detached mode (in the background).*
- ***--name myapp**: Names the container myapp.*
- ***-v myvolume:/app/data**: Creates a volume mount.*
  - ***myvolume**: The name of the Docker volume on your host machine.*
  - ***/app/data**: The path inside the container where the volume will be mounted. Any data written to this path will be stored in the volume.*
- ***ubuntu**: Specifies the base image to use, in this case, the official Ubuntu image.*

List Volumes

docker volume ls

- ***docker volume ls**: Lists all Docker volumes on your system.*

Key columns:

- ***DRIVER**: The driver used for the volume. The default is local.*
- ***VOLUME NAME**: The name of the volume (e.g., myvolume).*

Inspect Volume

docker volume inspect myvolume

- ***docker volume inspect***: *Provides detailed information about a specific Docker volume.*
- ***myvolume***: *The name of the volume you want to inspect.*

7. Networking Containers

Default Bridge Network

docker network ls

- ***docker network ls***: *Lists all Docker networks on your system.*

Key columns:

- ***NETWORK ID***: *Unique identifier for the network.*
- ***NAME***: *The name of the network (e.g., bridge, host, none).*
- ***DRIVER***: *The network driver used. Common drivers are:*
- ***bridge***: *The default network driver, creates a private internal network on your host.*
- ***host***: *Uses the host's network stack directly.*
- ***none***: *Disables networking for the container.*
- ***SCOPE***: *The scope of the network (usually local for local networks).*

Create a Custom Network

docker network create mynet

This creates a new bridge network (the default type) called mynet. You can now use this network to connect containers to it.

- ***docker network create***: *Creates a new Docker network.*

- ***mynet***: *The name of the network you're creating.*

docker run -d --name app1 --network=mynet nginx

This command will:

- *Start a container named app1 using the nginx image.*
- *Connect app1 to the mynet network, meaning it will be able to communicate with other containers on that network.*
- ***docker run***: *Starts a new container.*
- ***-d***: *Runs the container in detached mode (in the background).*
- ***--name app1***: *Names the container app1.*
- ***--network=mynet***: *Connects the container to the mynet network you created earlier.*
- ***nginx***: *Specifies the Docker image to use (in this case, the official nginx image).*

docker run -d --name app2 --network=mynet busybox sleep 3600

- ***docker run***: *Starts a new container.*
- ***-d***: *Runs it in detached mode (background).*
- ***--name app2***: *Names the container app2.*
- ***--network=mynet***: *Connects the container to the custom Docker network mynet.*
- ***busybox***: *Uses the lightweight busybox image.*
- ***sleep 3600***: *Runs the sleep command for 3600 seconds (1 hour), keeping the container running.*

docker exec -it app2 ping app1

- ***docker exec***: *Executes a command in a running container.*
- ***-it***: *Runs interactively with a TTY (so you can see real-time output).*
- ***app2***: *The container where the command is being executed.*
- ***ping app1***: *Sends ICMP echo requests to app1, the name of another container.*

8. Docker Container Lifecycle

Command ***Description***

create	*Creates a container but does not start it*
start	*Starts a created/stopped container*
run	*Creates and starts a container*
stop	*Stops a running container*
kill	*Forcefully stops a container*
rm	*Removes a container*
pause/unpause	*Temporarily halts container processes*
exec	*Executes command in running container*
logs	*Displays log output*

Working with Docker Images

1. Understanding Docker Images

- *A **Docker image** is a read-only template with instructions for creating a container.*

- *Images are built from a **Dockerfile** and can be shared via **Docker Hub** or other registries.*

2. Building Docker Images

Using a Dockerfile

docker build -t myimage:latest .

- ***docker build**: Build a Docker image from a Dockerfile.*
- ***-t myimage:latest**: Tag the image with the name myimage and the tag latest.*
- ***.**: Use the current directory as the build context (it should contain the Dockerfile and any necessary files).*

3. Viewing Docker Images

docker images

The docker images command lists all the Docker images stored locally on your system.

Column Explanation:

- ***REPOSITORY**: Name of the image (e.g., myimage).*
- ***TAG**: Version tag (e.g., latest, v1, etc.).*
- ***IMAGE ID**: Unique identifier for the image.*
- ***CREATED**: When the image was built.*
- ***SIZE**: Disk size of the image.*

4. Removing Docker Images

docker rmi image_name

Remove a Docker image from your local system.

5. Tagging Docker Images

docker tag myimage myrepo/myimage:1.0

Create a new tag (name/version) for an existing Docker image.

- ***myimage*** *– The existing local Docker image (could be built or pulled).*
- ***myrepo/myimage:1.0*** *– The new name and tag you're assigning. This is often used to prepare an image for pushing to a Docker registry (like Docker Hub, AWS ECR, GitHub Container Registry, etc.).*

6. Pushing and Pulling Images

Push to Docker Hub

docker login

Authenticate your Docker CLI with a Docker registry, such as Docker Hub or a private registry.

docker push myrepo/myimage:1.0

Upload (push) a Docker image from your local machine to a Docker registry (e.g., Docker Hub, GitHub Container Registry, or a private registry).

Pull from Docker Hub

docker pull ubuntu:20.04

Download the ubuntu:20.04 image from the Docker registry (Docker Hub) to your local machine.

7. Running Containers from Images

docker run -d --name mycontainer myimage:latest

Start a Docker container in detached mode (-d), with the specified name and image.

- ***docker run**: Creates and starts a container.*
- ***-d**: Runs the container in detached mode, meaning it runs in the background and you won't see the container's logs in the terminal.*
- ***--name mycontainer**: Assigns the container the name mycontainer (you can reference it by this name in future commands).*
- ***myimage:latest**: Specifies the image (myimage) and the tag (latest) to use to create the container.*

8. Saving and Loading Images

Save image to tarball

docker save -o myimage.tar myimage

Export a Docker image to a .tar archive file, which you can store, transfer, or load later on another system.

- ***docker save**: Saves one or more Docker images to a tar archive.*
- ***-o myimage.tar**: Specifies the output file name (in this case, myimage.tar).*
- ***myimage**: The name (and optional tag) of the image you want to save.*

Load image from tarball

docker load -i myimage.tar

Import a Docker image from a .tar archive that was previously created using docker save.

- ***docker load***: *Loads an image from a tar archive into your local Docker image cache.*
- ***-i myimage.tar***: *Specifies the input tar file that contains the image.*

9. Exporting and Importing Containers

Export container (not image):

docker export container_name > container.tar

Export the filesystem of a Docker container (not an image) into a .tar archive.

- ***docker export***: *Exports the container's entire filesystem as a tarball.*
- ***container_name***: *The running or stopped container's name or ID.*
- ***> container.tar***: *Redirects the output into a file named container.tar.*

Import as image:

cat container.tar | docker import - mynewimage

This imports a tar archive (container.tar) into Docker and creates a new image named mynewimage.

10. Inspecting Images

docker inspect myimage

View detailed low-level information about a Docker image (or container, volume, etc.).

Docker Volumes

1. What is a Docker Volume?

*A **volume** is a **persistent storage mechanism** in Docker that exists **outside of the container's writable layer**, and is managed by Docker itself.*

Why use Volumes?

- ***Persistent storage** even if the container is deleted.*
- ***Data sharing** between containers.*
- ***Better performance** and **security** than bind mounts.*
- *Easier **backup and restore**.*

2. Creating and Using Volumes

Create a volume:

docker volume create myvolume

Create a named Docker volume called myvolume.

Run container using volume:

docker run -d -v myvolume:/app/data --name mycontainer nginx

The docker run command you provided runs an NGINX container in detached mode (-d), mounts a Docker volume myvolume to the container's /app/data directory, and names the container mycontainer.

- ***docker run****: Starts a new container.*
- ***-d****: Runs the container in detached mode (in the background).*
- ***-v myvolume:/app/data****: Mounts a volume named myvolume to the /app/data directory inside the container.*
- ***--name mycontainer****: Assigns the name mycontainer to the running container.*
- ***nginx****: Specifies the image to run, in this case, the official NGINX image.*

3. Inspecting Volumes

docker volume ls **# List all volumes**

List all the volumes in your Docker environment. Volumes are used to persist data generated and used by Docker containers.

- ***DRIVER****: The driver used for the volume (usually local).*
- ***VOLUME NAME****: The name of the volume.*

docker volume inspect myvolume **# Get detailed info**

Retrieve detailed information about a specific Docker volume. It provides metadata such as the volume's mount point, driver, labels, and more.

4. Removing Volumes

docker volume rm myvolume **# Remove a volume**

Remove a specific Docker volume. Once a volume is removed, the data stored in it will be lost permanently, so be sure you no longer need the data before removing it.

docker volume prune **# Remove all unused volumes**

Remove all unused or "dangling" Docker volumes. These are volumes that are not currently being used by any containers.

Volumes in use by a container cannot be removed.

5. Bind Mounts vs Volumes

Feature	***Volumes***	***Bind Mounts***
Managed by	*Docker*	*Host OS*
Use case	*Persistent, shared data*	*Config, source code mounting*
Path defined by	*Docker*	*User-defined*
Backup support	*Yes*	*Manual*

6. Named vs Anonymous Volumes

Named Volume:

docker run -v myvolume:/app/data nginx

Run a Docker container with a volume mounted inside it.

- ***docker run****: This command is used to create and start a new container from a specified image.*
- ***-v myvolume:/app/data****: This flag specifies that a volume is being mounted into the container.*
  - ***myvolume****: This is the name of the volume on the host system that you want to mount. If the volume doesn't exist, Docker will create it automatically.*
  - ***/app/data****: This is the path inside the container where the volume will be mounted. The data stored in myvolume will appear at this path inside the container.*
- ***nginx****: This is the image from which the container is created. In this case, the image is nginx, which is a popular web server. By default, Docker will run the Nginx container in the foreground.*

Anonymous Volume:

docker run -v /app/data nginx

- ***docker run****: This command is used to create and start a new container from a specified image, in this case, the nginx image.*
- ***-v /app/data****:*
  - *This flag tells Docker to mount a volume into the container. However, the syntax provided is incomplete because it specifies only a path without specifying a host volume or container path.*
  - *In this case, Docker assumes the /app/data directory is on the host machine (because no volume name is provided before the colon). If the directory doesn't exist on the host, Docker will create it automatically and mount it to the container's /app/data directory.*
- ***nginx****: This is the image from which the container is being created. The container will run Nginx (a web server) and mount the /app/data directory from the host to /app/data in the container.*

7. Sharing Volumes Between Containers

docker run -d -v sharedvol:/shared --name container1 busybox

Create and run a Docker container in detached mode with a volume mounted inside it

- ***docker run****: This command is used to create and run a Docker container from a specified image (busybox in this case).*
- ***-d****: The -d flag stands for "detached mode." This runs the container in the background, allowing you to use the terminal for other commands while the container continues to run.*
- ***-v sharedvol:/shared****:*
  - *The **-v** flag is used to mount a volume into the container.*
  - ***sharedvol****: This is the name of the volume on the host. If the volume doesn't already exist, Docker will automatically create it.*

    - ***/shared***: *This is the directory inside the container where the volume will be mounted. Any changes made to the /shared directory inside the container will be reflected in the sharedvol volume on the host, and vice versa.*
- ***--name container1***: *This flag assigns a custom name (container1) to the container. By default, Docker assigns a random name to containers, but with this flag, you can give it a more meaningful or human-readable name.*
- ***busybox***: *This is the image that the container will be created from. busybox is a minimal Docker image, commonly used for testing or when you want a small container to run commands.*

docker run -d --volumes-from container1 --name container2 busybox

start a new container (container2) from the busybox image and mount the volumes from an existing container (container1)

- ***docker run***: *This command creates and runs a Docker container from the specified image (busybox in this case).*
- ***-d***: *The -d flag stands for "detached mode," which means the container will run in the background, freeing up your terminal for other commands.*
- ***--volumes-from container1***:
    - *This flag mounts the volumes from the existing container (container1) into the new container (container2).*
    - *Docker will mount all the volumes that are mounted in container1 into container2. This is useful for sharing data between containers that are using the same volumes.*
    - *In this case, any volume mounted in /shared inside container1 (from the previous command you ran) will also be mounted in the same location (/shared) inside container2.*
- ***--name container2***: *This assigns the name container2 to the newly created container. By default, Docker assigns a random name, but this flag allows you to specify a custom name for easier reference.*
- ***busybox***: *This is the image from which the new container will be created. busybox is a lightweight image, often used for testing and running basic Linux utilities in containers.*

8. Volume with Docker Compose

sudo nano docker-compose.yml

version: "3.8"

services:

web:

image: nginx

volumes:

- mydata:/usr/share/nginx/html

volumes:

mydata:

bash

Copy code

docker-compose up -d

start up your Docker containers in the background, based on the configuration specified in your ***docker-compose.yml*** *file.*

- ***docker-compose****: docker-compose is a tool used to define and manage multi-container Docker applications. It allows you to define your application's services (containers), networks, and volumes in a docker-compose.yml file and then run them with simple commands.*
- ***up****: The up command is used to build, create, and start the containers specified in your docker-compose.yml file. If the containers are already created, it will just start them.*
- ***-d****: The -d flag stands for detached mode, which means the containers will run in the background. Without -d, the containers would run in the foreground, and you would see their logs and output directly in your terminal. With -d, the command will exit, and the containers will continue running in the background.*

Docker Networking

Docker networking allows containers to communicate with each other, the host system, and external networks. It provides different networking options to control how containers interact.

Key Docker Networking Concepts and Modes:

1. Bridge Network (Default Network)

- ***What it is**: The default network driver used when you create a container without specifying a network.*

- ***How it works**: Containers in the bridge network are isolated from the host and other networks. They can communicate with each other, but not with the host system or other networks unless explicitly configured.*

- ***Use case**: For isolated applications that don't need to communicate with the host or other external services directly.*

Example:

```
docker network create --driver bridge my_bridge_network
```

- ***docker network create**: This is the Docker command used to create a new network.*
- ***--driver bridge**: Specifies the network driver to use. The bridge driver creates a private internal network on the host, allowing containers connected to it to communicate with each other.*
- ***my_bridge_network**: This is the custom name you're assigning to the network.*

2. Host Network

- ***What it is**: The container shares the host's network stack. It does not get its own IP address; instead, it uses the host's IP address.*

- ***How it works***: *Containers running on the host network can access the host's network interfaces and resources directly.*

- ***Use case***: *For containers that need to access network resources on the host without isolation.*

Example:

docker run --network host my_container

- ***docker run***: *Starts a new container.*
- ***--network host***: *Uses the host network driver, meaning the container shares the host's network stack directly.*
- ***my_container***: *This is the image or name of the container you want to run*

Key Points about ***--network host****:*

- ***No network isolation***: *The container will not get its own IP address; it will share the host's IP.*
- ***No port mapping needed***: *Ports opened in the container are immediately available on the host.*
- ***Better performance***: *Slightly better network performance due to no NAT layer.*
- ***Linux only***: *The host network mode works only on Linux, not on Docker Desktop for Mac/Windows.*

3. None Network

- ***What it is***: *The container has no network connectivity at all.*

- ***How it works***: *This is used when you don't need network access inside the container, such as for tasks like batch processing or standalone applications.*

- ***Use case***: *For containers that don't require networking.*

Example:

docker run --network none my_container

- ***docker run****: Starts a new container.*
- ***--network none****: Disables all networking for the container.*
- ***my_container****: The name or image of the container you're starting.*

Key Points about ***--network none****:*

- *The container* ***has no access to any network****, including the internet, the host, or other containers.*
- ***No IP address*** *is assigned inside the container.*
- *This is useful for:*
  - *Security isolation*
  - *Testing apps without network*
  - *Sandboxing environments*

4. Overlay Network

- ***What it is****: This network type allows containers running on different Docker hosts to communicate with each other. It is typically used in Docker Swarm or Kubernetes environments.*

- ***How it works****: Overlay networks use an underlying key-value store (e.g., Consul, etcd) to manage network configuration and ensure connectivity across multiple Docker hosts.*

- ***Use case****: For multi-host communication, commonly used in Swarm services or Kubernetes.*

Example:

docker network create --driver overlay my_overlay_network

- ***docker network create***: *Creates a new Docker network.*
- ***--driver overlay***: *Specifies the overlay network driver, used for multi-host container communication.*
- ***my_overlay_network***: *The name you're giving to this overlay network.*

5. Macvlan Network

- ***What it is***: *This network driver gives containers their own MAC addresses, making them appear as physical devices on the network.*

- ***How it works***: *Containers can communicate with the host and external networks as if they were physical devices, allowing advanced networking configurations.*

- ***Use case***: *For containers that need to be fully integrated into the physical network.*

Example:

docker network create -d macvlan --subnet=192.168.1.0/24 --gateway=192.168.1.1 -o parent=eth0 my_macvlan_network

- ***docker network create***: *Creates a new Docker network.*
- ***-d macvlan***: *Specifies the macvlan driver, which allows containers to appear as physical devices on the local network.*
- ***--subnet***: *Defines the IP range the network can use.*
- ***--gateway***: *Sets the default gateway for the containers in the network.*
- ***-o parent=eth0***: *Tells Docker to bind the macvlan to the physical interface eth0 on the host.*
- ***my_macvlan_network***: *The name of the new network.*

6. Container Network

- ***What it is***: *A network that allows one container to directly connect to another container. It shares the same network namespace and allows full access to each other's interfaces.*

- ***How it works****: This mode allows direct communication between containers without any intermediary.*

- ***Use case****: When you want containers to interact closely and share resources without needing to expose them to the host.*

Example:

docker run --network container:my_container_name my_new_container

- ***docker run****: Runs a new container.*
- ***--network container:my_container_name****: Shares the network namespace of an existing container named my_container_name.*
- ***my_new_container****: The image for the new container you're starting.*

Key Networking Commands

List networks:

docker network ls

- *Lists all Docker networks currently available on your system.*
- *Shows both default and user-defined networks.*

Inspect a network:

docker network inspect my_network

- *Displays detailed JSON-formatted information about the Docker network named* ***my_network****.*

Connect a container to a network:

docker network connect my_network my_container

- ***Purpose**: Connects an existing container **(my_container)** to an **additional Docker network (my_network)**.*
- *After running this, the container will be attached to **both its original network and my_network**.*

Disconnect a container from a network:

docker network disconnect my_network my_container

- ***Purpose**: Disconnects the container my_container from the Docker network my_network.*
- *After this, the container will no longer be able to communicate with other containers on **my_network**.*

Docker Compose Networking

*When using Docker Compose, containers defined in the same **docker-compose.yml** file are automatically placed in a single network. You can specify the network settings as follows:*

sudo nano docker-compose.yml

```
version: "3"
services:
  web:
    image: nginx
    networks:
      - webnet
  db:
    image: mysql
```

```
    networks:
      - webnet
networks:
  webnet:
```

Key Networking Considerations:

- ***Ports***: *Containers can expose ports to communicate with the external world. You can map container ports to host ports using the* ***-p*** *flag (e.g.,* ***docker run -p 8080:80****).*

- ***DNS***: *Docker provides a built-in DNS for containers to resolve the names of other containers on the same network.*

- ***Isolation***: *Networks can be used to isolate containers from each other for security or organizational reasons.*

Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to define a multi-container environment in a single YAML file, and with a single command, you can spin up all the containers required for your application.

Key Concepts of Docker Compose:

docker-compose.yml

This YAML file is where you define all the services, networks, and volumes for your multi-container application.

Services

These are the different containers that make up your application. Each service has its own configuration, including the image to use, build context, ports, volumes, and environment variables.

Volumes

Docker Compose allows you to define persistent storage that can be shared between containers.

Networks

You can define how containers communicate with each other, using both default and custom networks.

Commands

With Docker Compose, you can start, stop, build, and manage multiple containers using a set of easy-to-remember commands.

Example docker-compose.yml file:

sudo nano docker-compose.yml

```
version: '3'
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
    networks:
      - webnet

  db:
    image: postgres:latest
    environment:
      POSTGRES_PASSWORD: example
    networks:
      - webnet

networks:
  webnet:
    driver: bridge
```

Explanation:

1. ***Version**: The version of Docker Compose syntax used. Version '3' is the most widely used for newer versions.*

2. ***Services**:*

   - ***web**: A service running Nginx with a port mapping from the host to the container.*

   - ***db**: A PostgreSQL container with a password environment variable for setting the POSTGRES_PASSWORD.*

3. ***Networks***: *Both services are connected to a custom network webnet.*

Basic Commands:

Start your application

docker-compose up

Start and run a multi-container Docker application defined in a ***docker-compose.yml*** *file. It reads the configurations from this file, builds the images (if necessary), and starts the containers accordingly.*

- ***docker-compose up***: *Starts the containers defined in the docker-compose.yml file.*

Start in detached mode

docker-compose up -d

Start the containers defined in your ***docker-compose.yml*** *file in detached mode, meaning the containers will run in the background without occupying your terminal session.*

- ***docker-compose up***: *Starts the containers defined in the docker-compose.yml file.*
- ***-d (detached mode)***: *Runs the containers in the background, freeing up the terminal.*

Stop your application

docker-compose down

Stop and remove all containers, networks, and volumes that were created by ***docker-compose up***. *This command is typically used to clean up after running the containers.*

Build images

docker-compose build

*build (or rebuild) the Docker images defined in the **docker-compose.yml** file. This command reads the configuration and builds the images as specified, often pulling from a Dockerfile if one is defined.*

View logs

docker-compose logs

*view the logs from the containers managed by Docker Compose. It allows you to see the output (including errors, warnings, and other log data) from the services defined in your **docker-compose.yml** file.*

List containers

docker-compose ps

*list the status of the containers managed by Docker Compose. It provides details about the containers, including their names, status, ports, and other information based on the services defined in the **docker-compose.yml** file.*

Use Cases for Docker Compose:

- ***Multi-container applications**: Like an app with a web front-end, database, and caching layer.*

- ***Development environments**: Easily spin up an environment with required dependencies for testing or development.*

- ***CI/CD pipelines****: Integrate multi-container setups into your CI pipeline, where you can spin up and tear down services.*

Environment Variables

Environment variables in Docker are used to pass configuration values and information to containers at runtime. They can be set during the container creation and help configure the behavior of applications inside containers.

1. Setting Environment Variables in the Dockerfile

You can define environment variables directly in the Dockerfile using the ENV instruction. This is useful for setting default values that will be available to the container when it runs.

Example:

```
FROM ubuntu:latest
ENV APP_ENV=production
ENV DB_HOST=localhost
```

*When the container is run, these environment variables (**APP_ENV** and **DB_HOST**) will be available to any application inside the container.*

2. Setting Environment Variables at Runtime with docker run

*You can also set environment variables dynamically when starting a container using the **-e** or **--env** flag in the **docker run** command.*

Example:

```
docker run -e APP_ENV=development -e DB_HOST=127.0.0.1 my_image
```

Run a Docker container with environment variables defined.

- ***docker run**: This starts a new container.*
- ***-e APP_ENV=development**: Sets the environment variable **APP_ENV** to **development** inside the container.*

- ***-e DB_HOST**=127.0.0.1: Sets the environment variable **DB_HOST** to **127.0.0.1** (localhost) inside the container.*
- ***my_image**: Specifies the Docker image to use for the container.*

*In this case, the container will receive **APP_ENV=development** and **DB_HOST=127.0.0.1** as environment variables.*

3. Using .env File with Docker Compose

*If you're using Docker Compose, you can define environment variables in a **.env** file. This makes it easier to manage configuration across different environments and setups.*

Example .env file:

```
APP_ENV=production
DB_HOST=localhost
```

*Docker Compose will automatically load this file and use the values in the services defined in your **docker-compose.yml** file.*

Example docker-compose.yml:

```
version: '3'
services:
  app:
    image: my_image
    environment:
      - APP_ENV=${APP_ENV}
      - DB_HOST=${DB_HOST}
```

*The environment variables from the **.env** file will be injected into the container at runtime.*

4. Accessing Environment Variables in a Running Container

*Once the container is running, you can access the environment variables from inside the container using the **env** or **printenv** command.*

Example:

docker exec -it my_container env

*Will execute the **env** command inside a running Docker container (**my_container**), and it will print out all the environment variables set inside that container.*

- ***docker exec**: Executes a command inside a running container.*
- ***-it**: Combines two flags:*
  - ***-i** (interactive) keeps the session open.*
  - ***-t** allocates a pseudo-TTY, which allows for better terminal interaction.*
- ***my_container**: The name or ID of the running container you want to execute the command inside.*
- ***env**: The command to display all environment variables inside the container.*

5. Overriding Environment Variables

*If you want to override an environment variable that was set in the **Dockerfile** or by a **.env** file, you can use the **-e** flag during **docker run**.*

Example:

docker run -e APP_ENV=staging my_image

Runs a Docker container from the image my_image with a single environment variable set:

- ***APP_ENV**=staging: This sets the **APP_ENV** variable inside the container to **"staging".***

*In this case, the **APP_ENV** will be overridden to **staging** instead of the default **production** value from the **Dockerfile**.*

6. Default Values for Environment Variables in Docker Compose

*You can set default values for environment variables in Docker Compose if they are not specified in the **.env** file.*

Example:

```
version: '3'
services:
  app:
    image: my_image
    environment:
      - APP_ENV=${APP_ENV:-development}  # Default value if not set
```

*If **APP_ENV** is not set in the **.env** file, it will default to **development**.*

Use Cases:

- ***Configuration Management**: Use environment variables for storing sensitive information like database passwords or API keys.*

- ***Cross-Environment Setup**: Manage different configurations for development, staging, and production environments.*

- ***Security**: Avoid hardcoding sensitive information directly in your Dockerfiles or code.*

Docker Swarm

Docker Swarm is a container orchestration tool built into Docker that allows you to manage a cluster of Docker nodes as a single virtual host. It simplifies the deployment and scaling of containerized applications across a cluster.

Key Concepts:

Swarm Mode

This is Docker's native clustering and orchestration mode. When you enable Swarm mode, you can manage multiple Docker hosts as a single unit.

Manager Node

The manager node is responsible for managing the swarm cluster, scheduling tasks, and maintaining the cluster's state. You can have multiple manager nodes, but only one will be the leader at any given time.

Worker Node

Worker nodes execute the tasks (containers) assigned to them by the manager. They do not make any decisions about cluster management.

Service

A service is a task or a set of tasks (containers) running in the Swarm cluster, and it defines the desired state for your application.

Task

A task is a running container in the Swarm, which is assigned by the manager to the worker nodes.

Replica

A set number of instances of a service that you want running at any given time.

Basic Commands:

Initialize a Swarm:

docker swarm init

This command sets up your current machine as a manager node.

Add a Worker Node to the Swarm:

On the manager node, get the join token:

docker swarm join-token worker

Then run the output command on the worker node to join the swarm.

List Nodes in the Swarm:

docker node ls

This shows all the nodes in the swarm (both manager and worker nodes).

Create a Service:

You can create a service with a specified image and the desired number of replicas:

docker service create --name my_service --replicas 3 nginx

This will run 3 replicas of the NGINX container across your cluster.

Scaling a Service:

You can scale a service up or down by changing the number of replicas:

docker service scale my_service=5

This command will scale my_service to 5 replicas.

Inspect a Service:

To get detailed information about a service:

docker service inspect my_service

Updating a Service:

You can update the service to use a new image version:

docker service update --image nginx:latest my_service

Remove a Service:

docker service rm my_service

Example Workflow:

1. Set up Swarm:

- *Initialize the manager node:* ***docker swarm init****.*
- *Add worker nodes using the join token.*

2. Create a Service:

- *Deploy a multi-container service, e.g.,* ***docker service create --name webapp --replicas 3 nginx****.*

3. Scale the Service:

- *Scale up or down the service based on the load.*

4. Update the Service:

- *Push new images or configurations to the service and have it update accordingly.*

5. Monitor and Maintain:

- *Monitor the service status and inspect logs:* ***docker service ps webapp****.*

Advantages of Docker Swarm:

- ***Simplicity****: It's easy to set up, as Docker Swarm is built into Docker, requiring no external tools.*
- ***Scalability****: You can easily scale your application up and down by modifying the replica count.*
- ***Load Balancing****: Swarm automatically balances the load across nodes, making your application more resilient.*
- ***High Availability****: Docker Swarm has built-in features for managing failures, ensuring that your services are always available.*

Monitoring Containers

Monitoring containers in Docker is essential to ensure that your applications run smoothly and to catch any issues early. Docker provides various tools and techniques for container monitoring

1. Using docker stats Command

*The **docker stats** command provides a real-time view of container resource usage (CPU, memory, network I/O, disk I/O, etc.).*

docker stats

docker stats <container_name>

This commands shows statistics for all running containers.

2. Docker Logs

Docker allows you to view the logs of running containers. You can check logs to troubleshoot issues with containers.

docker logs <container_name>

The docker logs command allows you to view the logs of a running or stopped container

For continuous log streaming:

docker logs -f <container_name>

follow the logs of a container in real time, meaning it will continuously show new log entries as they are written

*You can also view logs for a specific time range by using **--since** and **--until** options.*

3. Using Docker API

You can access the Docker API to gather metrics and container information programmatically. It gives more flexibility if you're building a custom monitoring solution.

GET /containers/(id)/stats

The GET /containers/(id)/stats endpoint is part of the Docker Remote API, and it provides resource usage statistics for a specific container. These stats include CPU, memory, network I/O, and disk I/O usage, which are useful for monitoring and analyzing the container's performance.

4. Using Third-Party Tools

There are several third-party monitoring tools that integrate well with Docker:

- ***Prometheus & Grafana**: Prometheus is often used to collect container metrics, and Grafana is used for visualizing them.*
    - *Set up a Prometheus server to scrape metrics from Docker containers.*
    - *Use cAdvisor or the Docker Prometheus exporter for metrics collection.*

Prometheus Docker monitoring setup example:

```
scrape_configs:
  - job_name: 'docker'
    static_configs:
      - targets: ['<docker_host>:<metrics_port>']
```

- ***cAdvisor**: Google's cAdvisor is a tool that provides container metrics, and it is commonly used with Prometheus and Grafana.*

```
docker run -d --name=cadvisor \
 -p 8080:8080 \
 --volume=/var/run/docker.sock:/var/run/docker.sock \
 google/cadvisor:latest
```

- ***docker run -d****: Run the container in detached mode (in the background).*
- ***--name=cadvisor****: Name the container* ***cadvisor****.*
- ***-p 8080:8080****: Map port* ***8080*** *of the host to port* ***8080*** *of the container, which is where cAdvisor serves its web UI and metrics.*
- ***--volume=/var/run/docker.sock****:/var/run/docker.sock: Mount the Docker socket so cAdvisor can access Docker runtime information.*
- ***google/cadvisor:latest****: Use the latest image of cAdvisor from Google's Docker registry.*

- ***Datadog****: A cloud-based monitoring service that integrates with Docker and provides real-time monitoring, alerting, and dashboards.*

- ***ELK Stack (Elasticsearch, Logstash, Kibana)****: For logging and monitoring with centralized log collection, search, and visualization.*

5. Using Docker Events

Docker emits events when something happens with containers, such as starting, stopping, or failing. You can listen for Docker events to track the status of your containers.

```
docker events
```

This command will start streaming events to the terminal in real-time.

You can filter events for a specific container:

```
docker events --filter 'event=start' --filter 'event=die' <container_name>
```

Listen for specific Docker events related to a particular container, filtering for both start and die events.

6. Docker Monitoring with Metrics

*You can enable Docker to expose its metrics for monitoring purposes using the **Docker Stats API**. Here is an example of exposing Docker container metrics to a monitoring tool:*

1. *Install the Docker Stats Exporter:*
   - *It collects metrics from the Docker API and makes them available for Prometheus.*

```
docker run -d \
  --name=docker-stats-exporter \
  -p 9104:9104 \
  -v /var/run/docker.sock:/var/run/docker.sock \
  quay.io/prometheuscommunity/docker-exporter
```

starts a container running the Docker Stats Exporter for Prometheus. This exporter collects Docker container statistics (such as CPU, memory, network, and disk usage) and exposes them at a specified HTTP endpoint, which Prometheus can scrape.

- ***docker run -d**: The -d flag runs the container in detached mode, meaning it runs in the background.*
- ***--name=docker-stats-exporter**: This sets the container's name to docker-stats-exporter for easy reference later.*
- ***-p 9104:9104**: This binds port 9104 on your host machine to port 9104 in the container. The exporter exposes metrics at this port.*
- ***-v /var/run/docker.sock:/var/run/docker.sock**: This mounts the Docker socket from the host (/var/run/docker.sock) into the container. This allows the exporter inside the container to access Docker's internal stats about running containers.*
- ***quay.io/prometheuscommunity/docker-exporter**: This specifies the image to use for the container. In this case, it's the Docker Stats Exporter image from the Prometheus Community's repository on Quay.io. The image exposes container metrics compatible with Prometheus.*

2. *Prometheus Configuration:*
    - *Add the stats exporter as a target in your Prometheus configuration.*

```
scrape_configs:
  - job_name: 'docker'
    static_configs:
      - targets: ['localhost:9104']
```

7. Container Resource Limits

You can set resource limits for Docker containers to prevent them from consuming too many resources.

For example:

```
docker run -d --memory=1g --cpus=0.5 --name=my_container my_image
```

Launches a Docker container with resource limits applied.

- ***-d**: Runs the container in detached mode (in the background).*
- ***--memory=1g**: Limits the maximum memory usage of the container to 1 gigabyte.*
- ***--cpus=0.5**: Restricts the container to use at most 50% of a single CPU core.*
- ***--name=my_container**: Assigns the name my_container to the container.*
- ***my_image**: Specifies the image to use when creating the container.*

8. Using Docker Compose for Monitoring

*If you're using Docker Compose to manage multi-container applications, you can integrate monitoring tools into your **docker-compose.yml** file. For example, adding a Prometheus container:*

```
version: '3'
services:
  prometheus:
```

```
    image: prom/prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
  cAdvisor:
    image: google/cadvisor
    ports:
      - "8080:8080"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

9. Setting Up Alerts

You can configure alerts based on container metrics (such as CPU or memory usage). Prometheus, combined with Alertmanager, can help trigger alerts if a container exceeds certain thresholds.

Example alert rule in Prometheus:

```
groups:
  - name: container_alerts
    rules:
      - alert: HighCPUUsage
        expr: rate(container_cpu_usage_seconds_total{job="docker"}[1m]) > 0.9
        for: 5m
        labels:
          severity: critical
        annotations:
          description: "CPU usage is over 90% for the last 5 minutes."
```

Secrets Management

Secrets management in Docker is the process of securely storing and accessing sensitive data such as passwords, API keys, SSH keys, TLS certificates, and database credentials.

1. Secrets in Docker Swarm (Native Secrets Management)

Docker Swarm has built-in support for managing secrets.

How It Works

- *Secrets are encrypted and stored in the Raft log.*
- *Only services running in Swarm mode can access secrets.*
- *Secrets are mounted as in-memory files inside containers.*

Step-by-Step Example

Step 1: Initialize Docker Swarm (if not already)

docker swarm init

*Initialize a **Docker Swarm**, which enables clustering of Docker engines. When you run this command, your Docker host becomes the **manager node** of the swarm.*

Step 2: Create a secret

echo "my_db_password" | docker secret create db_password –

*Create a secret in Docker Swarm called **db_password** with the value **my_db_password**.*

- ***echo "my_db_password"****: Outputs the string to standard output.*
- *The | pipe sends that output to the next command.*
- ***docker secret create db_password -:***
  - ***db_password****: Name of the secret.*
  - ***-****: Tells Docker to read the secret content from standard input (stdin), which is coming from* ***echo****.*

Step 3: Deploy a service with the secret

```
docker service create --name my_service \
 --secret db_password \
 nginx
```

- ***Creates a service*** *(my_service) in the Swarm.*
- ***Runs the nginx*** *container as a replicated service (default: 1 replica).*
- ***Mounts the secret*** *db_password into the container at runtime.*

Step 4: Access the secret inside the container

```
docker exec -it $(docker ps -q -f name=my_service) /bin/sh
```

- ***docker exe****: Run a command in a running container.*
- ***-it****: -i: interactive, -t: TTY – needed for shell access.*
- ***$(...)****: Command substitution – injects output of docker ps ....*
- ***docker ps -q -f name=my_service****: Gets the container ID of the first replica of my_service.*
- ***/bin/sh****: Starts a shell session inside the container.*

```
cat /run/secrets/db_password
```

Inside a container *in Docker Swarm to read the value of a secret that was mounted via the* ***--secret*** *flag during service creation.*

2. Using Docker Compose with Secrets (Swarm Mode)

*Secrets can be used in **docker-compose.yml** when deploying stacks to Docker Swarm.*

Example:

```
version: '3.8'

services:
  web:
    image: nginx
    secrets:
      - db_password

secrets:
  db_password:
    file: ./db_password.txt
```

Deploy with:

```
docker stack deploy -c docker-compose.yml mystack
```

*Deploy a multi-service application (stack) to a Docker Swarm cluster using a **docker-compose.yml** file.*

- ***docker stack deploy**: Deploys a full stack of services (like Docker Compose but for Swarm).*
- ***-c docker-compose.yml**: Specifies the Compose file to use.*
- ***Mystack**: The name of the stack. All services will be prefixed with this (e.g., mystack_web).*

3. Secrets in Docker (Non-Swarm / Standalone Mode)

*Docker **does not natively support secrets** outside of Swarm. But you can use workarounds:*

A) Environment Variables (Not Secure)

docker run -e DB_PASSWORD=mysecretpassword myapp

*Start a container from the **myapp** image and pass an environment variable **DB_PASSWORD** with the value **mysecretpassword** to the container.*

- ***docker run**: Starts a new container.*
- ***-e DB_PASSWORD=mysecretpassword**: Sets an environment variable inside the container.*
- ***Myapp**: The image to use for the container.*

*Not recommended for sensitive data (viewable via **docker inspect**, **ps**, or logs).*

B) Bind Mounting Secrets

docker run -v /host/path/secret.txt:/run/secrets/db_password myapp

***Mounts a file** from your host into the container at the path where Docker secrets are typically accessed.*

- ***docker run**: Start a new container.*
- ***-v /host/path/secret.txt:/run/secrets/db_password**: Mount the file secret.txt from your host into the container at /run/secrets/db_password.*
- ***Myapp**: The image to use for the container.*

*Inside the container, the app reads from **/run/secrets/db_password**.*

C) Docker Secrets CLI Tools (3rd Party)

Use tools like:

- ***Vault by HashiCorp***
- ***AWS Secrets Manager***
- ***Azure Key Vault***
- ***doppler***

Debugging Containers

Debugging containers in Docker involves identifying and resolving issues with the container's performance, networking, or other aspects.

1. Check Container Logs

docker logs <container_name_or_id>

View the logs (output) of a running or stopped Docker container.

- ***<container_name_or_id>**: Replace this with the actual name or ID of your Docker container.*

Tips:

- *To follow the logs in real time: **docker logs -f <container_name_or_id>***

- *If your container runs with logging to files, you may need to access those files inside the container.*

2. Access the Running Container's Shell

docker exec -it <container_name_or_id> /bin/bash

Start an interactive bash shell inside a running Docker container.

- ***docker exec**: Runs a command in a running container.*
- ***-i**: Interactive (keeps STDIN open).*
- ***-t**: Allocates a pseudo-TTY (makes it act like a terminal).*
- *<container_name_or_id>: The name or ID of the running container.*
- ***/bin/bash**: The command you want to run (here, it's Bash shell).*

If the container doesn't have bash, use:

docker exec -it <container_name_or_id> /bin/sh

Open an interactive shell (sh) inside a running container *— particularly useful for lightweight containers like Alpine, BusyBox, or scratch-based images that don't include bash.*

3. Inspect Container Status and Details

docker inspect <container_name_or_id>

Display detailed, low-level information about a Docker container (or image, volume, network, etc.) in JSON format.

4. Check Docker Events

docker events

Stream real-time event logs from the Docker daemon.

5. Network Debugging

Check Networking Issues:

- *Verify the container's network settings using **docker inspect <container_name_or_id>**.*

- *Use **docker network inspect <network_name>** to inspect the Docker network configuration.*

Ping Other Containers:

docker exec -it <container_name_or_id> ping <other_container_name_or_ip>

Ping another container or IP address from inside a running Docker container, which is very helpful for testing network connectivity between containers.

Check Port Bindings:

docker ps

List all running Docker containers on your system.

6. Check Resource Usage

docker stats

Display a live stream of resource usage statistics for running Docker containers.

7. Rebuild or Restart the Container

Sometimes, rebuilding or restarting the container might fix issues:

Restart:

docker restart <container_name_or_id>

Stop and then start a Docker container, effectively restarting it.

Rebuild (if using a Dockerfile):

docker build -t <image_name> .

*Build a Docker image from a **Dockerfile** in the current directory (.) and tag it with the specified name.*

- ***docker build**: Command to build an image.*
- ***-t <image_name>**: Tags the image with a name (e.g., **myapp**, **myapp:latest**, **myrepo/myapp:v1**).*
- ***.**: The build context — typically the current directory containing the Dockerfile and required app files.*

docker run <image_name>

Create and start a container from a Docker image. It executes the image as a container in the background or interactively, depending on the options provided.

- *<image_name>: The name (and optionally, the tag) of the image to run, such as myapp, nginx, or ubuntu:latest.*

8. Check Docker Daemon Logs

Docker daemon logs can be useful for diagnosing issues with Docker itself.

sudo tail -f /var/log/upstart/docker.log

Monitor Docker-related logs in real-time from the /var/log/upstart/docker.log file. Specifically, it follows (using -f) the log file as new entries are added.

- ***sudo**: Runs the command with elevated (administrator) privileges. This is required because accessing /var/log/ typically requires root permissions.*
- ***tail -f**: Displays the last few lines of the file and then keeps the terminal open to display new logs as they are appended.*

- ***/var/log/upstart/docker.log**: This is a log file created by Upstart, which is an init system used to manage services. It logs Docker service-related information on systems using Upstart.*

9. Check for Image Issues

- *If there are issues with the container image, such as missing dependencies or incorrect configurations, it could be the root cause of the problem.*

- *Rebuilding the image or checking the Dockerfile for issues (e.g., wrong base image, missing steps) can help resolve this.*

10. Use Docker Compose (for multi-container setups)

If using Docker Compose, you can check the logs of all containers at once:

```
docker-compose logs
```

View the logs of all containers in a Docker Compose setup. It collects and displays the logs for all the services defined in your docker-compose.yml file.

11. Check Container Exit Codes

Containers that fail might provide a non-zero exit code, which can help in debugging:

```
docker inspect --format '{{.State.ExitCode}}' <container_name_or_id>
```

Retrieve the exit code of a specific container. This can help determine whether a container stopped successfully or encountered an error.

- ***docker inspect**: Retrieves detailed information about a container (in JSON format).*
- ***--format '{{.State.ExitCode}}'**: Uses Go templating to extract just the exit code of the container's last run.*
- ***Exit codes:***
  - ***0**: The container exited successfully.*

- ***Non-zero values***: *Indicates an error or failure.*

- *<container_name_or_id>: Replace with the actual container name or ID you want to inspect.*