Docker Compose

# Key Concepts:

**Container:**

A container is a lightweight, standalone, and executable package that includes everything needed to run software, such as the code, runtime, libraries, and system tools. Containers are isolated from each other and the host system, which makes them portable and easy to manage.

**Image:**

An image is a read-only template from which containers are created. It contains the application code, libraries, and dependencies needed to run an application. Images can be stored in Docker registries, such as Docker Hub.

**Dockerfile:**

A Dockerfile is a text file that contains a set of instructions used to build a Docker image. It defines the base image, environment variables, commands, and any files that need to be copied into the image.

**Docker Engine:**

Docker Engine is the core service that runs and manages Docker containers. It is responsible for creating, running, and managing containers and images. It has both a server-side component (the Docker daemon) and a client-side component (the Docker CLI).

**Docker Hub:**

Docker Hub is a cloud-based registry service for sharing and managing Docker images. It allows users to discover, create, and publish Docker images. Users can pull images from Docker Hub to run containers locally.

**Volumes:**

Volumes are used for persistent data storage outside of containers. They allow data to persist even after a container is stopped or removed. Volumes can be shared between containers and are managed by Docker.

**Networks:**

Docker provides networking capabilities that allow containers to communicate with each other and with the outside world. Containers can be connected to different types of networks (e.g., bridge, host, or overlay) based on the use case.

**Docker Compose:**

Docker Compose is a tool for defining and running multi-container Docker applications using a simple YAML file (docker-compose.yml). It allows users to configure all containers, networks, and volumes in a single file and manage them as a cohesive unit.

**Registry:**

A Docker registry is a storage and distribution system for Docker images. Docker Hub is the default public registry. Users can also set up private registries to store their images securely.

**Tags:**

Tags are versions of images. They are used to differentiate multiple versions of the same image. For example, myimage:latest refers to the latest version, while myimage:v1.0 refers to version 1.0 of the image.

# Docker Compose:

Environment:

You can set environment variables directly in your container's environment with the environment attribute in your compose.yml.

It supports both list and mapping syntax:

**services:**

**webapp:**

**environment:**

**DEBUG: "true"**

**is equivalent to**

**services:**

**webapp:**

**environment:**

**- DEBUG=true**

You can choose not to set a value and pass the environment variables from your shell straight through to your containers. It works in the same way as **docker run -e VARIABLE ...:**

The order of precedence (highest to lowest) is as follows:

1. Set using docker compose run -e in the CLI.
2. Set with either the environment or env\_file attribute but with the value interpolated from your shell or an environment file. (either your default .env file, or with the --env-file argument in the CLI).
3. Set using just the environment attribute in the Compose file.
4. Use of the env\_file attribute in the Compose file.

Set in a container image in the ENV directive. Having any ARG or ENV setting in a Dockerfile evaluates only if there is no Docker Compose entry for environment, env\_file or run --env.

If you want to quickly switch between image tags to test multiple versions, or want to adjust a volume source to your local environment, you don't need to edit the Compose file each time, you can just set variables that insert values into your Compose file at run time.

Interpolation can also be used to insert values into your Compose file at run time, which is then used to pass variables into your container's environment

**Below is a simple example:**

**cat .env**

**TAG=v1.5**

**cat compose.yml**

**services:**

**web:**

**image: "webapp:${TAG}"**

# Profile:

Services are associated with profiles through the profiles attribute which takes an array of profile names:

services:

frontend:

image: frontend

profiles: [frontend]

phpmyadmin:

image: phpmyadmin

depends\_on: [db]

profiles: [debug]

backend:

image: backend

db:

image: mysql

Here the services frontend and phpmyadmin are assigned to the profiles frontend and debug respectively and as such are only started when their respective profiles are enabled.

Services without a profiles attribute are always enabled. In this case running docker compose up would only start backend and db.

To start a specific profile supply the --profile command-line option or use the COMPOSE\_PROFILES environment variable:

**COMPOSE\_PROFILES=debug docker compose up**

The above commands would both start your application with the debug profile enabled. In the example, compose.yml file above, this starts the services backend, db and phpmyadmin.

**Start multiple profiles:**

You can also enable multiple profiles, e.g. with docker compose --profile frontend --profile debug up the profiles frontend and debug will be enabled.

Multiple profiles can be specified by passing multiple --profile flags or a comma-separated list for the COMPOSE\_PROFILES environment variable:

**docker compose --profile frontend --profile debug up**

**COMPOSE\_PROFILES=frontend,debug docker compose up**

If you want to enable all profiles at the same time, you can **run docker compose --profile "\*".**

# Secrets:

A secret is any piece of data, such as a password, certificate, or API key, that shouldn’t be transmitted over a network or stored unencrypted in a Dockerfile or in your application’s source code.

Docker Compose provides a way for you to use secrets without having to use environment variables to store information. If you’re injecting passwords and API keys as environment variables, you risk unintentional information exposure. Services can only access secrets when explicitly granted by a secrets attribute within the services top-level element.

Use secrets

Getting a secret into a container is a two-step process. First, define the secret using the top-level secrets element in your Compose file. Next, update your service definitions to reference the secrets they require with the secrets attribute. Compose grants access to secrets on a per-service basis.

Examples

Simple

In the following example, the frontend service is given access to the my\_secret secret. In the container, /run/secrets/my\_secret is set to the contents of the file ./my\_secret.txt.

services:

myapp:

image: myapp:latest

secrets:

- my\_secret

secrets:

my\_secret:

file: ./my\_secret.txt

Advanced

services:

db:

image: mysql:latest

volumes:

- db\_data:/var/lib/mysql

environment:

MYSQL\_ROOT\_PASSWORD\_FILE: /run/secrets/db\_root\_password

MYSQL\_DATABASE: wordpress

MYSQL\_USER: wordpress

MYSQL\_PASSWORD\_FILE: /run/secrets/db\_password

secrets:

- db\_root\_password

- db\_password

wordpress:

depends\_on:

- db

image: wordpress:latest

ports:

- "8000:80"

environment:

WORDPRESS\_DB\_HOST: db:3306

WORDPRESS\_DB\_USER: wordpress

WORDPRESS\_DB\_PASSWORD\_FILE: /run/secrets/db\_password

secrets:

- db\_password

secrets:

db\_password:

file: db\_password.txt

db\_root\_password:

file: db\_root\_password.txt

volumes:

db\_data: