SWE Pod 7 Workshop - Containerization (Docker & Docker Compose)

Containerization

Introduction

- Process in which an application is encapsulated including its dependencies, configuration files, OS libraries that are required to run the application. As the result of this process, we obtain a container (an application that can be run on any infrastructure).
- Containers are lightweight, portable, and required small amount of computing power. They are also isolated from the host machine.
- Solves the problem: "It works on my computer, but not on yours."

Benefits

- Create and deploy applications faster and more secure.
- Makes software development lifecycle (SDLC) more predictable.
- Less chance for variation, fewer code related issues.
- Consistent software environment during development, testing, and production.

User Cases

- Migration from monolithic applications to microservices.
- Migration from on-premise infrastructure to cloud infrastructure.
- IoT

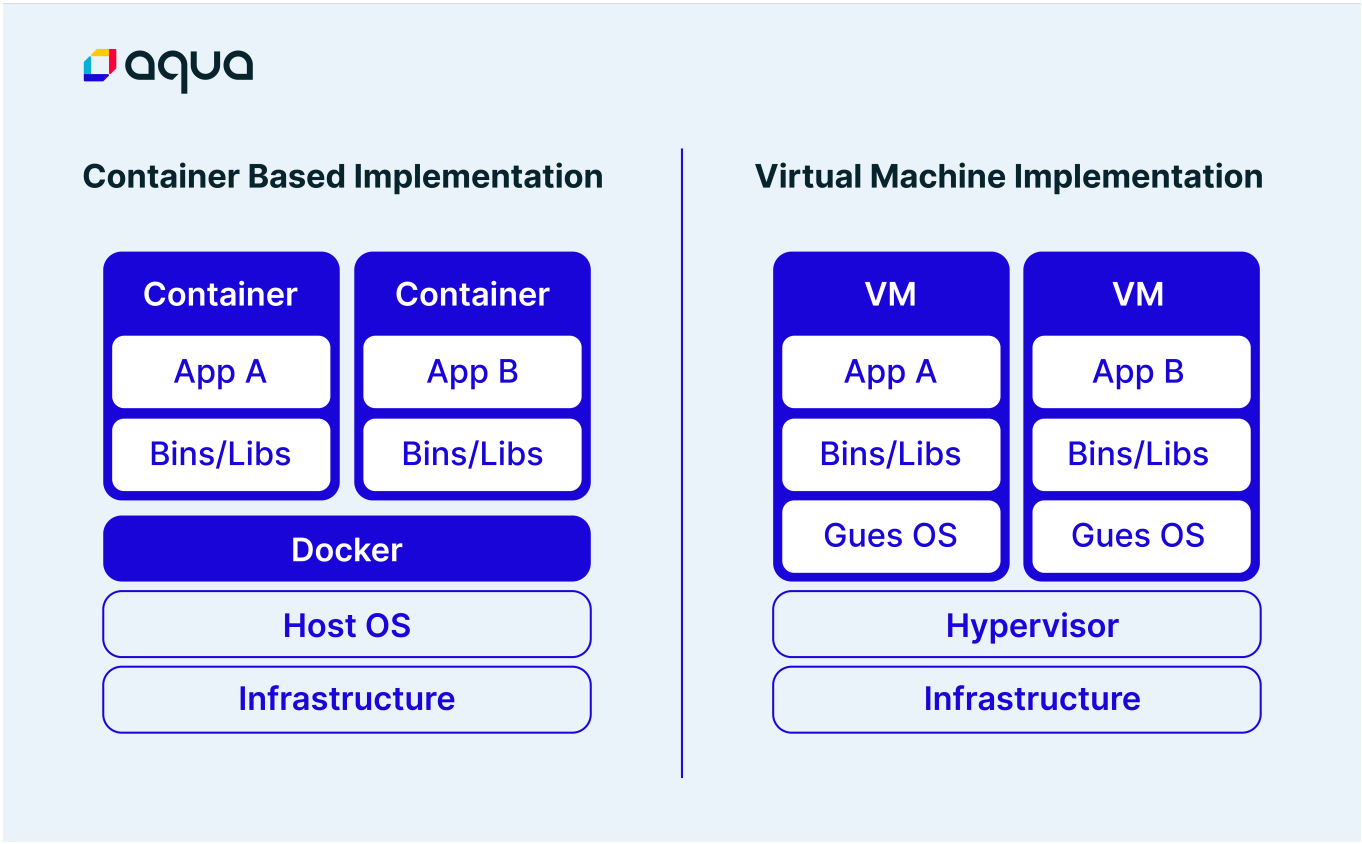
Virtualization (VMs) vs Containerization

Virtualization enables you to run multiple operating systems on the hardware of a single physical server, while **containerization** enables you to deploy multiple applications using the same operating system on a single virtual machine or server.

Key Differences:

1. **Resource Overhead**. Container shares the same host system's operating system making them more lightweight. On the other hand, virtual machines requires their own OS, increasing the overhead, especially when many VMs are running on the same host system.

2. **Startup Time**. Container start up more quickly than VMs.
3. **Portability.** Both offer a great degree of portability, but containers have slight edge because they package the application and all its dependencies together and VMs are more dependent on the underlying hardware.
4. **Security Isolation.** Virtual machines have the advantage, because each vm is isolated from the host system, while containers still share the host system's OS.
5. **Scalability and Management.** The lightweight nature and rapid startup time offered by containers make them ideal for scaling applications quickly and efficiently.



Container Orchestration

Container orchestration is the process of managing, deploying, scaling, and networking containers in a production environment. Orchestration tools are essential for automating the operational aspects of containerized applications, especially when they are deployed as microservices across distributed environments. Here's a breakdown of the key aspects of container orchestration:

1. **Deployment and Scheduling**
   Orchestration tools allow containers to be deployed across a cluster of servers and ensure they are scheduled and started on appropriate hosts based on resource requirements. They manage the placement of containers to ensure efficient utilization of resources like CPU and memory.

2. **Scaling and Load Balancing**
   Containers can be scaled up or down depending on demand. Orchestration platforms monitor application performance and automatically adjust the number of running containers to meet current needs, often with built-in load balancing to ensure traffic is distributed evenly across containers.
3. **Fault Tolerance and Self-Healing**
   Orchestration tools detect if a container or node fails and can automatically restart containers or move them to other nodes within the cluster. This self-healing feature ensures high availability by replacing failed containers and minimizing downtime.
4. **Service Discovery and Networking**
   Orchestration platforms often provide built-in networking solutions, allowing containers to communicate with each other regardless of their host. They also manage service discovery, automatically assigning and managing IP addresses and DNS entries, so that containers can find and connect to each other.
5. **Storage Management**
   Containers are typically ephemeral, but orchestration platforms manage persistent storage to ensure that data is not lost if a container is stopped or restarted. They allow containers to attach to and detach from storage volumes as needed.
6. **Security and Configuration Management**
   Orchestration tools provide security features such as role-based access control (RBAC) to control who can deploy, modify, or view containers. They also handle secrets management to securely store sensitive information like API keys and passwords.

Container Orchestration Tools

Kubernetes: The most widely used container orchestrator, providing robust features for managing containerized applications across clusters.
Docker Swarm: An orchestration tool built into Docker, offering simpler, Docker-native orchestration capabilities.
Apache Mesos with Marathon: A cluster manager and orchestration tool for deploying applications and services across large-scale clusters.
Docker Compose: Docker Compose is a tool that simplifies the development, deployment, and management of multi-container Docker applications. It's especially useful when you have applications that require several services to work together, such as a web server, database, and caching layer. Use in small scale projects.

Docker

Introduction to Docker

- Open platform for developing, shipping, and running applications.

- Allows us to manage our infrastructure in the same ways we manage our application.
- Packages and runs applications in a loosely and isolated environment (**Container**).

Docker Architecture and Components

- Client-server architecture.
- **Docker Client:** Primary way that Docker users interact with Docker. Client use commands such as `docker run` to communicate with the Docker Daemon.
- **Docker Daemon:** Listens for Docker API request and manages Docker Objects such as images, containers, networks, and volumes.

The Docker Client talks to the Docker Daemon. They both (client and daemon) run on the same system. There are cases where you can connect your docker client to a remote docker daemon.

- **Docker Desktop:** An easy-to-install application for Linux, Mac and Windows that allows you to build and share containerized applications. It includes the Docker Daemon, the Docker Client, Docker Compose, and more components.
- **Docker Registries:** A storage and distribution system for Docker images. A docker registry is organized into Docker repositories. It allows users to pull/push images to/from the registry. It also works as a version control tool for the images. By default, docker interacts with **DockerHub** but there are other public registries such as Amazon Elastic Container Registry (ECR), Google container Registry (GCR), etc.
- **Docker Images**
- **Docker Containers**

Docker Images and Containers

Images

- An image is a read-only **template** (or "prototype/blueprints").
- Includes all of the files, env variables, binaries, libraries and configurations or create and run a container.
- Images can be based on another image with additional customization.
- Make use of a **Dockerfile** to build your own images. In this file we explicitly define the steps needed to create and run the image.

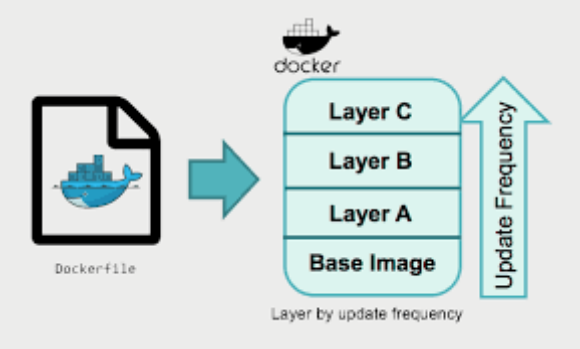
Two important principles:

1. **Immutable.** We are not allowed to modify an image once the image is created. We can only make a new image or add changes on top of it.

2. **Images are composed of layers.** Each layer represents modifications in the image.

Layers

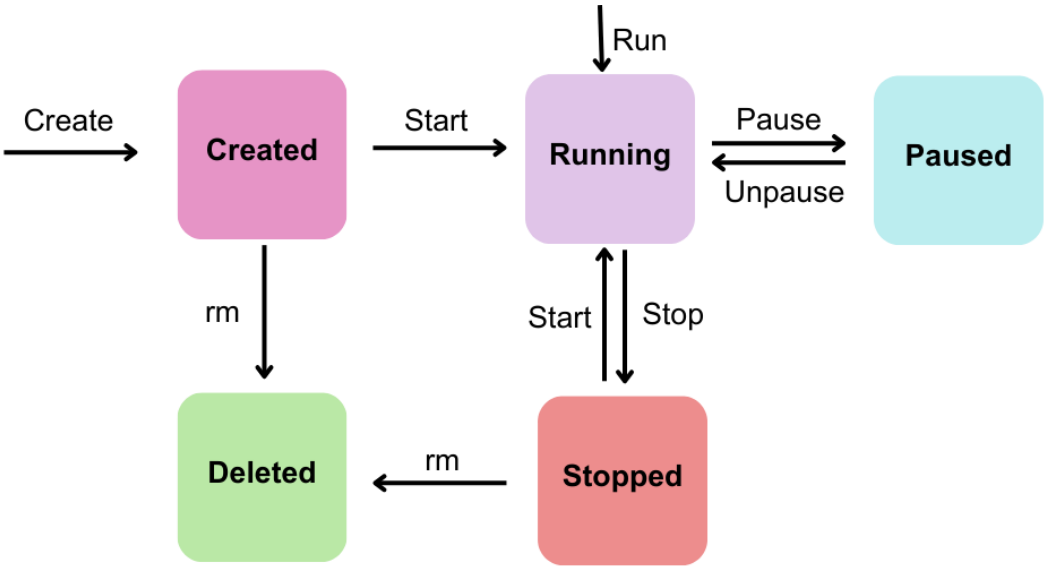
- Layers stacked on top of each other.
- Layers let you extend images of others by reusing their base layers, allowing you to add only the data that you application needs.



Container

- A container is a runnable, portable and isolated instance of an image.
- They are defined by its image as well as any configuration option that we provide to it when we create or start it.
- Create, start, stop or delete a containers.
- When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Lifecycle



Build images

- Dockerfile - "the recipe" that docker uses to build images.
- Instructions are not case-sensitive but uppercase for convention.
- Starts with a base image - the foundation of our application
  `FROM <baseimage>:<version>`
- Run commands
  `RUN <command>`
- Set environment variables
  `ENV <name> <value>`
- Copy files from host to image
  `COPY <host-path> <image-path>`
- Set working directory where files be copied and commands will be executed.
  `WORKDIR <path>`
- Expose ports
  `EXPOSE <port-number>`
- Specify what command should be executed when we start the container
  `CMD <command or executable> arg` - easy to override - also used to provide default arguments for ENTRYPOINT.
  `ENTRYPOINT command arg` - hard to override - consider this when a container should run the same executable every time.

Container management

`docker run [OPTIONS] image [COMMAND] [ARGS...]`
`docker create [OPTIONS] image [COMMAND] [ARGS...]`
`docker start [OPTIONS] contianer`
`docker rm [OPTIONS] container`
`docker stop [OPTIONS] container`
`docker pause [OPTIONS] container`
`docker unpaused [OPTIONS] container`

Docker Compose

Docker Compose allows you to define, configure, and run multiple Docker containers as a single application using a YAML file. Here’s how it works and what it’s typically used for:

Key Features of Docker Compose

1. **Multi-Container Management**
   Docker Compose allows you to define and manage multiple containers that make up your application. You can start, stop, and restart all services with a single command, making it easier to work with complex applications.

2. **Configuration in a YAML File**
   With Docker Compose, you define your application's services, networks, and volumes in a single docker-compose.yml file. This file specifies the configuration details for each container, such as the image to use, environment variables, port mappings, and dependencies.
3. **Networking**
   Docker Compose automatically sets up networking between containers defined within the same docker-compose.yml file, allowing them to communicate easily by service name. This simplifies the setup process for containerized applications that rely on inter-container communication.
4. **Shared Volumes and Persistent Storage**
   You can define volumes in the docker-compose.yml file, allowing containers to share data or have persistent storage that persists even if containers are stopped or removed.
5. **Environment Management**
   Docker Compose allows you to manage environment-specific settings by supporting .env files, which help in setting environment variables separately. This is particularly useful for configuring credentials, API keys, or other sensitive data without hardcoding them into your configuration file.
6. **Scalability (to a Limited Extent)**
   Docker Compose lets you scale services by specifying the number of container instances. For example, you can scale a web service to run multiple container instances. However, Docker Compose isn't designed for high-scale orchestration and is more suitable for local development or small-scale applications.

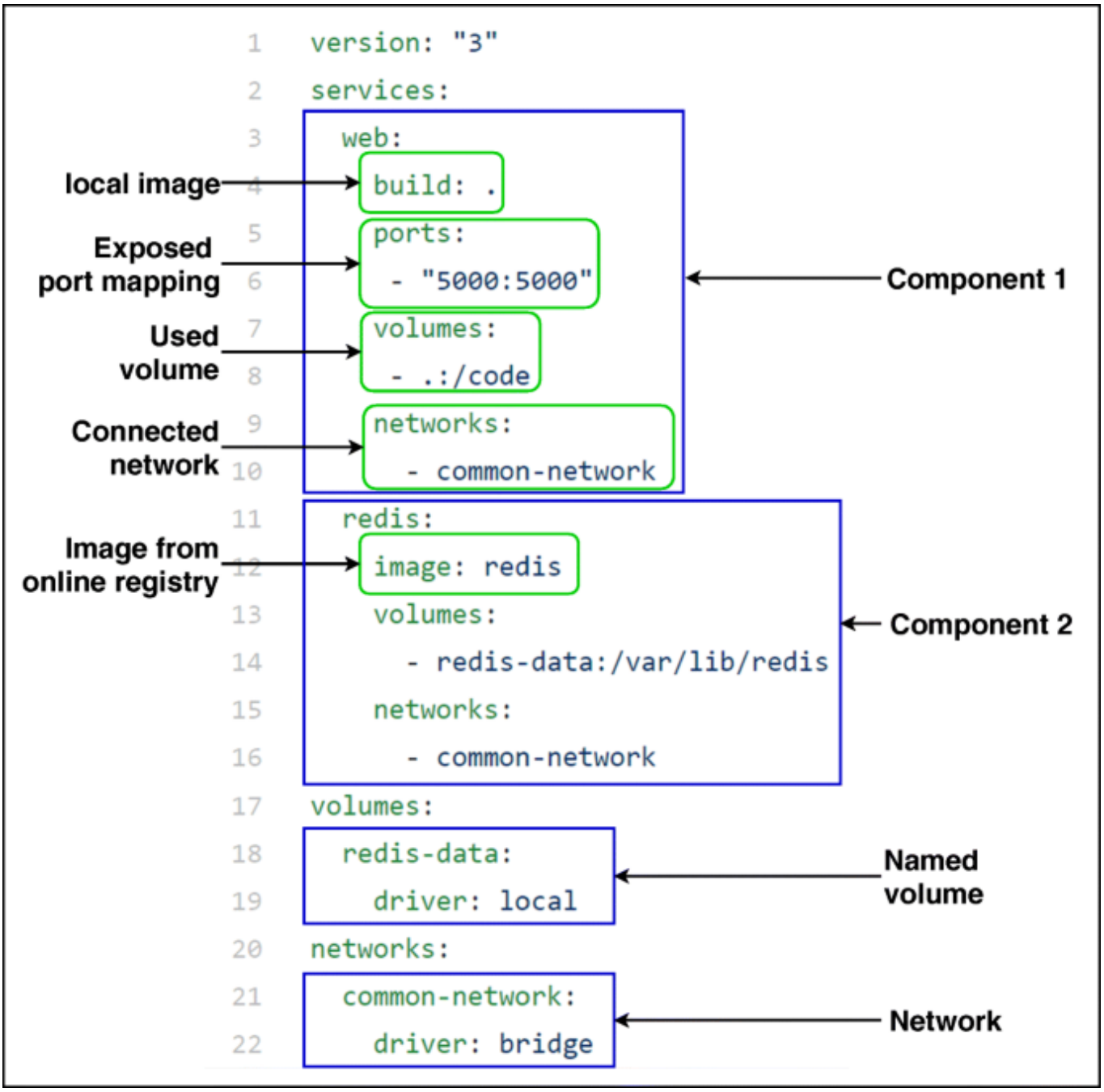
Typical Use Cases for Docker Compose

1. **Local Development and Testing**
   Docker Compose is commonly used for local development, as it lets you create an environment similar to production. Developers can define the services and configurations needed for the application and run it locally with a single command.
2. **Testing Microservices**
   In microservices architectures, Docker Compose allows you to spin up and test all the interconnected services locally before deploying them. You can create a setup for your app's entire stack, test integration between services, and perform end-to-end testing in an isolated environment.
3. **Simplifying CI/CD Pipelines**
   Many CI/CD pipelines use Docker Compose to test multi-container applications. It can spin up the necessary services in an isolated environment, run tests, and then tear everything down afterward, ensuring that each build has a clean environment.

4. **Small-Scale Deployments**
   While Docker Compose is typically used for development, it can also handle simple production deployments. For example, small applications or prototypes that don’t require high availability or advanced orchestration can be managed with Docker Compose.

Compose Application Model (Levels)

- **Service:** An abstract concept of a container that you want to run as a part of your application. Each service corresponds to a containerized application and its configuration.
- **Networks:** Services make use of networks to communicate with each other. Capable of establishing an IP route between containers within services connected together.
- **Volumes:** Services store and share persistent data in volumes.
- **Configs:** Configuration of data that is dependent on the runtime or platform.
- **Secret:** Configuration of sensitive data.


version: "3"
services:
web:
build: .
ports:
- "5000:5000"
volumes:
- .:/code
networks:
- common-network
redis:
image: redis
volumes:
- redis-data:/var/lib/redis
networks:
- common-network
volumes:
redis-data:
driver: local
networks:
common-network:
driver: bridge
local image
Exposed port mapping
Used volume
Connected network
Image from online registry
Component 1
Component 2
Named volume
Network