Containers, Docker, Compose

Seminar Objectives

- Understand the concept of containers
- Learn the core functionality of Docker
- Explore docker compose for deploying microservices
- Gain practical skills with docker and containers

Contents

- Introduction to Containers
- Introduction to Docker
- Benefits of containers
- Working with Docker Images
- Managing Containers
- Advanced Docker concepts
- Best Practices

01

Containers

An introduction

Overview

- What are Containers?
- Containers vs Virtual Machines
- Benefits of containers
- Common Use Cases

What are containers?

- Environment:
  - Lightweight
  - Portable
  - Isolated
- Bundles App code with
  - Runtime
  - Dependencies
  - OS
- Consistent behavior
  - Independent of the host

Containers vs virtual machines

Containers	Virtual Machines (VMs)
Share the host operating system (OS) kernel	Each VM has its own OS and kernel
Lightweight, faster startup times	Heavier, slower startup times
Consume fewer system resources	Require more CPU, RAM, and storage
Ideal for microservices architecture	Suitable for monolithic applications

Benefits of using containers

- Consistency across deployments
- Portability (works anywhere)
- Efficiency (low resource usage)
- Isolation of processes

Common use cases

- SOA => Microservices
- DevOps and CI/CD
- Cloud and Edge deployments
- Cross-system development/compilation

Demo: Run on host

```
mkdir demo-app
cd demo-app
npm init -y
npm install express
npm install --save-dev typescript @types/node @types/express
npx tsc --init
```

Demo

src/index.ts

```
import express from 'express';

const app = express();
const port = process.env.PORT || 3000;

app.get('/', (req, res) => {
  res.send('Hello from the local environment!');
});

app.listen(port, () => {
   console.log(`Server is running on http://localhost:${port}`);
});
```

Demo

package.json

```
{
  "name": "demo-app",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    ”start": "node dist/index.js",
    "build": "tsc",
    "dev": "ts-node-dev src/index.ts"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "@types/express": "^4.17.21",
    "@types/node": "^22.5.4",
    "typescript": "^5.6.2"
  },
  “devDependencies”: {
    "express": "^4.21.0"
  }
}
```

Demo

```
npm run build
npm run start
```

Pitfalls

- Need to install Node/npm on the os
- Need to bring express… to my home/directory
- If other devs need to run this app, they need to install the right version of the OS, node, npm…
- If we repeat this many times over diverse software, the host machine will get cluttered with that software
- Even worse, we may have difficulty handling different versions of the same dependencies

Demo: containerize

Dockerfile

```
# Use an official Node.js runtime as a parent image
FROM node:16

# Set the working directory inside the container
WORKDIR /app

# Copy package.json and install dependencies
COPY package.json ./
RUN npm install

# Copy the rest of the application code
COPY . .

# Build the TypeScript code
RUN npm run build

# Expose the port that the app runs on
EXPOSE 3000

# Start the app
CMD ["npm", "start"]
```

Demo: containerize

build.sh

```
docker build -t demo-app .
docker run -p 3000:3000 demo-app
```

Demo recap

Host setup

- Requires install dependencies on host
  - Clutter
  - Version collisions
- Resource collisions:
  - Ports
  - Working directories
  - In general: shared resources on host

Container setup

- Container bundles everything
  - Host remains clean
  - Dependencies do not clash
- Only Docker is needed on host
  - No need for complex installations
- Multiple versions of app possible
  - Also of dependencies
  - Only Kernel is shared by all

Benefits of containers (recap)

Isolation

- Host remains clean
- Different deployments do not interfere with each other

Consistency

- Runs the same always:
- Container bundles always the same dependencies

Portability

- Can run anywhere

Quick setup

- Launching the container is fast
  - Like launching a process

02

Docker

An introduction

Overview

- What is Docker?
- Docker Architecture: Core Components
- Key Docker Concepts: Images, Containers, Registries
- Docker’s role in modern development
- Practical Demo: run a docker container

What is Docker?

- Open-source platform
  - Set of elements: daemons, cli's, services
- Helps using Containers
  - Allows developers to package applications into a standardized, distributable unit known as an image
    - Including all of its dependencies
      - OS too (minus kernel)
  - Converts an image into a Container
    - Portably
    - Consistently
    - Reproducibly

Docker Elements

Docker Daemon

Runs on host. Responsible for handling containers and images

Docker Cli

CLI to interact with the daemon

Docker Image

Format to store the packaged contents of an application

Docker Containers

Running instance of an image
Similar to a process

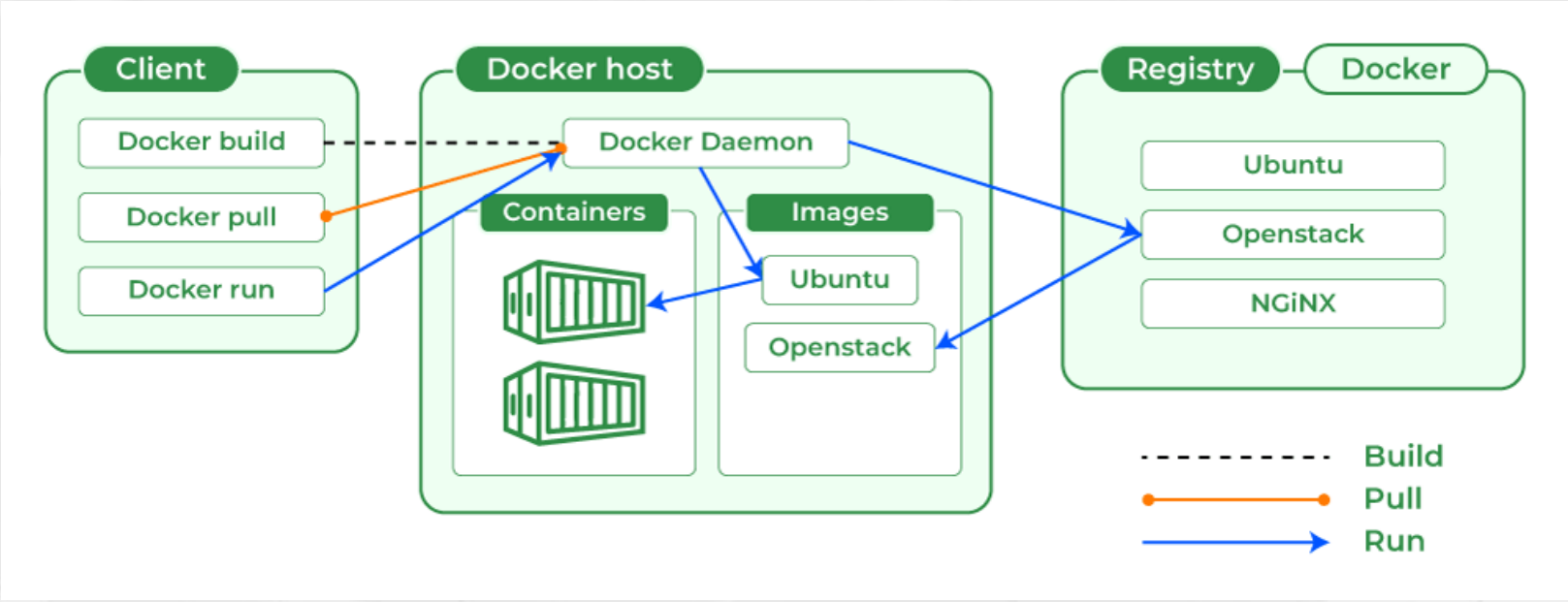
Docker Registry

Service capable to store and deliver docker images

Docker Hub

Docker Registry maintained by Docker

Architecture diagram



03

Docker Images

Overview

- Understanding Docker images and layers
- Build a Docker image with a Dockerfile
- Best practices to create images
- Managing Docker images
- Practical Demo: create a docker image

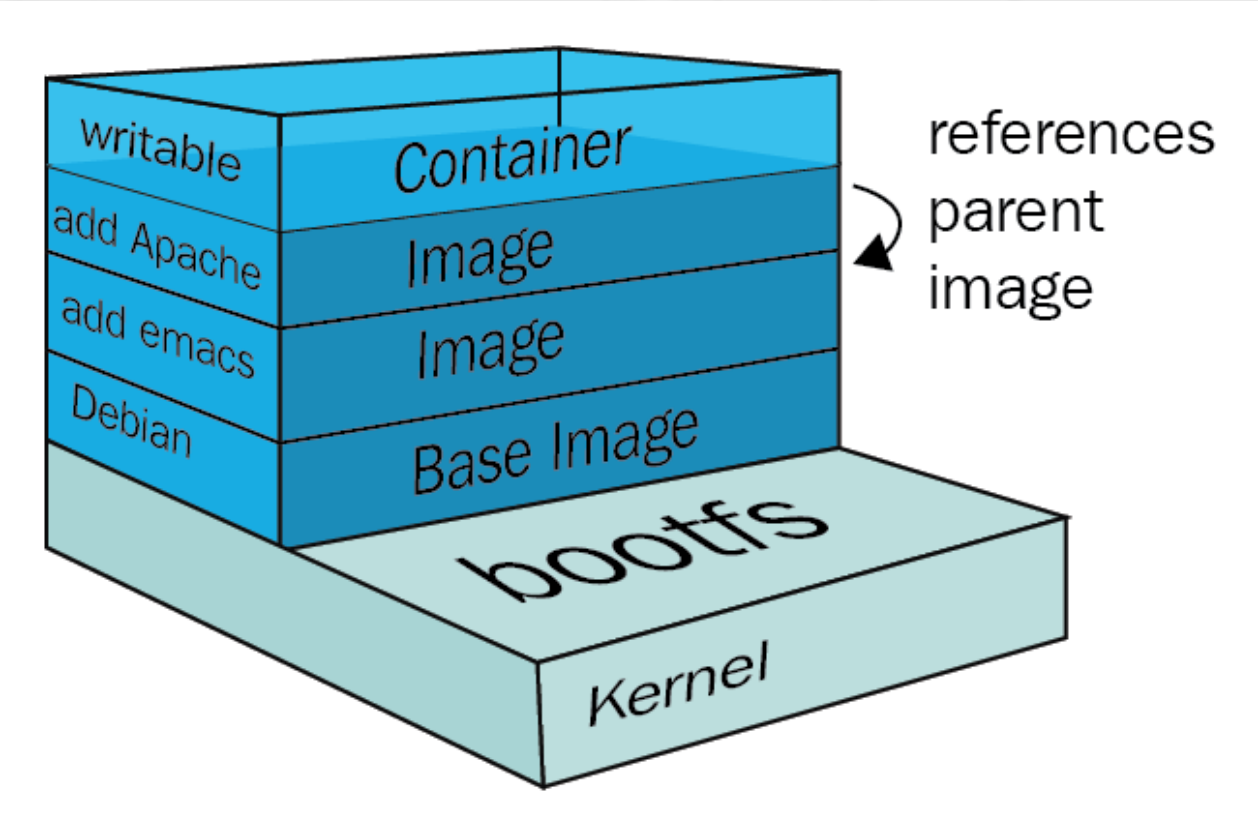
Docker Images

- Snapshot of an application (or set of them)
- Think of it as a complete filesystem
  - Like the disk of a computer
    - Minus the boot partition (that contains kernel code)
  - Contains all software needed to run an application
- Contains metadata
  - Indicates what should execute when it is run
- Can be packaged in an archive file (.tgz)
  - There is a standard definition of the format of such an archive
- Template to create a container

Layers

- Images are built in layers
  - Usually, the base layer is an operating system
  - Another layer may add system dependencies
  - Finally, another layer may actually copy application
- Layers can be shared among different images
  - Two images can share a bottom part
- Think of each layer as its own archive file (.tgz)
  - Starting with an empty disk
    - Unarchive each layer archive file at the root
  - At the end we have a “disk” with the whole system

Layers


writable
Container
add Apache
Image
add emacs
Image
Debian
Base Image
bootfs
Kernel
references
parent
image

Dockerfiles

- Script-like text file
- Contains instructions to build an image
- Indicates the image we start from
  - Typically images are built on top of other images
    - By adding extra layers
- Typically installs software from a base OS
- Additionally adds files accompanying the Dockerfile
- Each instruction in a Dockerfile creates a new layer in the resulting image

Structure of Dockerfiles: FROM

- Used to indicate the base image
- Starting point of a Docker file
- All subsequent instructions add files (one way or another) on top of this image

```
FROM node:16
```

…Dockerfiles: WORKDIR

- Sets the working directory where the rest of the commands are going to be referenced to
- If the directory does not exist, Docker creates it

```
FROM node:16

WORKDIR /app
```

…Dockerfiles: COPY

- Copies files and directories from the host to the image
  - Files are within the directory where the Dockerfile is interpreted
- Used to copy application artifacts

```
FROM node:16
WORKDIR /app
COPY package.json ./
```

- Copies package.json from the current host dir to the image WORKDIR (/app)

…Dockerfiles: RUN

- **Executes a command within the image.**
- **The command is expected to modify the image ”disk” by adding one more layer**

```
FROM node:16
WORKDIR /app
COPY package.json ./
RUN npm install
```

…Dockerfiles: EXPOSE

- Informs Docker of ports that the software will bind to when run
  - Just metadata
  - The port is not exposed outside of the container instantiated from this image
  - Not necessary to actually map to ports

```
FROM node:16
WORKDIR /app
COPY package.json ./
RUN npm install
EXPOSE 3000
```

…Dockerfiles: ENTRYPOINT

- Defines the program that will be run when Docker creates a container from the image
  - Docker executes the file when creating the container
- If not provided, the ENTRYPOINT of the image in the FROM statement is used.

```
FROM node:16
WORKDIR /app
COPY package.json ./
RUN npm install
EXPOSE 3000
ENTRYPOINT [”/bin/sh", ”-c"]
```

…Dockerfiles: CMD

- **Defines the list of arguments provided to the ENTRYPOINT**
- **Typically defines the main process to be run when the ENTRYPOINT is generic (e.g., /bin/sh)**

```
FROM node:16
WORKDIR /app
COPY package.json ./
RUN npm install
EXPOSE 3000
ENTRYPOINT [”/bin/sh", ”-c"]
CMD ["npm", "start"]
```

Building images

- When a Dockerfile is used, docker behaves as follows:
  - It creates a container with the FROM image.
    - It creates a top layer that can be written
- Executes each RUN command within the current container,
  - Writes on the top layer
  - Closes it
  - Adds a new writable top payer
- Executes each COPY (also ADD) command by
  - Copying the files to the top writable layer
  - Closing the layer
  - Adding a new writable layer on top

Building/running images

```
docker build -t demo-app .
```

- When executed where the Dockerfile is, it builds an image
- It names the image “demo-app”

```
docker run -p 3000:3000 demo-app
```

- Creates a container out of the image “demo-app”
- Executes its ENTRYPOINT, passing its CMD as args
  - Creates an additional read/write layer to the container

Images: best practices

- Use small base images.
  - To keep the resulting image small
- Keep in mind caching
  - Docker caches image layers
  - Copy and install the dependencies first in the Dockerfile
    - The layers created can be shared by other images using the same dependencies
- Use .dockerignore to avoid copying too many files to the image

Images: best practices

- **Minimize layers:**
  - **Each instruction creates a layer…**
  - **Layer add overhead when containers are run**
    - **(Union file systems)**

```
RUN apt-get update && apt-get install -y \
  curl \
  vim \
  && rm -rf /var/lib/apt/lists/*
```

1 layer

Instead of

```
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get install -y vim
RUN rm -rf /var/lib/apt/lists/*
```

4 layers

Summing up

- Dockerfiles declare how to build an image
- Main mechanism to build an image
  - Not the only one
    - We can run a container (docker run…)
      - And run commands within it (similar to RUN)
      - And use docker client to add files (similar to COPY)

04

Managing Images

Overview

- Local management
  - Tagging/deleting/listing/pruning
- Registries
  - Push/Pull/login

Local management: Listing

- Lets us see all the images stored within our host
  - Managed by the docker daemon on our host
  - Provides both image IDs and all their tags

```
docker images
```

Local management: Deleting

- Removing a concrete image
  - By ID
    - Removes all tags (must be forced if multiple)
  - By Tag
    - Removes the tag only

```
docker image rm …
```

Local management: Tagging

- Important for version control, clarity
- Also used to determine the push registry
- Either by ID or by existing tag
- “tag” === “name”

```
docker image tag <current-name-or-id> <another name>
```

Local management: Pruning

- Important to remove unused images & layers
- “unused”
  - Not tagged &
  - Not used in a container
- Optionally removes all images not used in containers
- Important to clean up unused space

```
docker image prune
```

Registries

- A docker registry is a Cloud service that stores different Docker images
- Images are organized in image repositories
  - Set of images with identical prefix tags
    - The suffix ending in “:” followed by a string
      - The suffix often used to specify version
      - Also, suffix also used to indicate which OS the image is based upon
- Most well known public registry is the ***Docker Hub***
  - Alse the default registry for docker

Registries: pull

- **Images on registries can be used to launch containers**
  - **Before being able to launch a container based on an image at a registry, it is necessary to pull the image**

```
docker pull <image-name>
```

- **After it**
  - <image-name> **will be included in the listing of**

```
docker images
```

Registries: push

- **Local images can be registered on the default registry doing:**

```
docker push <image-name>
```

- **Before it, the user must be logged in the registry**

```
docker login
```

Registries: login

- **The registry can be specified by its domain/url:**

```
docker login myregistry.example.com
```

- **With user/password**

```
docker login -u myusername -p mypassword myregistry.example.com
```

- **Better echoing the password**

```
echo “mypassword” | docker login -u myusername –password-stdin myregistry.example.com
```

- **Credentials are stored in…**

```
$HOME/.docker/config.json
```

05

Managing Containers

Overview

- **Listing and inspecting containers**
- **Starting, Stopping, and removing containers**
- **Container logs**
- **Managing container resources**
- Container Lifecycle

Listing & Inspecting

- **Listing active containers (optionally all, even stopped)**

```
docker ps [-a]
```

- **Inspecting containers**

```
docker inspect <container-id>
```

- **Result is a JSON structure**
  - **Can be queried**
    - **Directly in docker inspect command**
    - **Using external tools (e.g. jq)**

Inspecting examples (directly)

- **Get IP address of a container**

```
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' container_name_or_id
```

- **Get the image name of a container**

```
docker inspect --format='{{.Config.Image}}' container_name_or_id
```

- **List all port bindings**

```
docker inspect --format='{{range $p, $conf := .NetworkSettings.Ports}}{{$p}} -> {{(index $conf 0).HostPort}} {{end}}'
container_name_or_id
```

Inspecting examples (with jq)

- **Get IP address of a container**

```
docker inspect container_name_or_id | jq -r '.[0].NetworkSettings.Networks[].IPAddress'
```

- **Get the image name of a container**

```
docker inspect container_name_or_id | jq -r '.[0].Config.Image'
```

- **List all port bindings**

```
docker inspect container_name_or_id | jq -r '.[0].NetworkSettings.Ports | to_entries[] | "\(.key) -> \(.value[0].HostPort)"'
```

Creating a container

- **In the background**
  - **If cmd is provided it overrides the one in the image**

```
docker run –d  image_name_or_id [cmd_string]
```

- **Interactive**

```
docker run -it image_name_or_id
```

- **Naming it**

```
docker run –name name-given-by-me image_name_or_id
```

- **Many more options**
  - **It pulls the image if not found locally**

Starting/Stopping/Removing

- Stop a container
  - Gracefullv shuts down the container

```
docker stop container_name_or_id
```

- Start an existing (paused/stopped) container

```
docker start container_name_or_id
```

- Remove a container if stoppend (force optionally)

```
docker rm [-f] container_name_or_id
```

Execute command in container

- The provided command must be available within the container
  - It is run within the container

```
docker exec container_name_or_id command_name_or_path
```

- A frequently used example
  - Handy for debugging problems

```
docker exec container_name_or_id /bin/bash
```

Managing container resources

- **CPU and memory limits**

```
docker run –memory=“256m” –cpus=“2” image_name_or_id
```

- **Mapping host directories and files**

```
docker run –v host_path:container_path container_name_or_id
```

- **Freeing unused resources**

```
docker container prune
```

```
docker system prune
```

06

Docker Compose

Overview

- **What is docker compose**
- **The docker compose file format**
- **Building a multi-container application**
- **Scaling services with docker-compose**
- Docker Compose networking
- Practical Demo: Define and run a multi-container app

What is Docker Compose

- Tool that allows defining multi-container applications
- Useful to launch several microservices running together
- Easy to use:
  - YAML format, versionable
- Declarative configuration:
  - Services (as containers), networks, and volumes
  - All in a file, as a unit.
  - Automatic networking between the services
  - Automatic DNS resolution within the created network

Docker compose YAML format

```yaml
version: "3"
services:
  web:
    image: my_web_app_image
    ports:
    - "8080:80"
    environment:
    - APP_ENV=production
    depends_on:
    - db
    - redis

        db:
    image: postgres:13
    volumes:
    - db_data:/var/lib/postgresql/data

        redis:
    image: redis:alpine

      volumes:
  db_data:
```

version: Version of file format

services: Specifies how each container has to be run

- **image:** The image to use for the container
- **ports:** mapping from host to container
- **environment**: pass environment variables to processes within container
- **depends_on**: dependencies between services used to wait

volumes: Defines persistent storage volumes
In this example, db_data is used to store the database files outside of the db container

Build a multi-container app: step 1

Create a **docker-compose.yml** File

```yaml
version: "3"

services:
  web:
    image: my_web_app_image
    build: ./webapp
    ports:
    - "8080:80"
    depends_on:
    - db
    - redis
    environment:
    - DATABASE_URL=postgres://user:password@db:5432/mydatabase
    - REDIS_URL=redis://redis:6379

  db:
    image: postgres:13
    environment:
    - POSTGRES_USER=user
    - POSTGRES_PASSWORD=password
    - POSTGRES_DB=mydatabase
    volumes:
    - db_data:/var/lib/postgresql/data

  redis:
    image: redis:alpine

volumes:
  db_data:
```

Build & Run

- **Basic, building the image**

```
docker compose up --build
```

- **Run as a daemon**

```
docker compose up –d  --build
```

Scaling & auto service restart

- Any service in the compose can be scaled

```
docker compose up –scale web=3
```

- Docker compose allows specifying restart policies for a service
  - Add a field to the service spec in the compose file:

```
services:
  myservice:
    restart: always
...
```

Docker compose simple networking

- Internal communication:
  - All containers are in the same subnet
    - Can communicate with each other
  - A container can resolve the name of the other containers in the compose
    - Gets its IP in the internal network
- External access
  - By forwarding ports

```
services:
  web:
    ports:
    - “8080:80”
```

Demo: define and run compose

```yaml
version: "3"
services:
  web:
    image: my_web_app_image
    ports:
    - "8080:80"
    depends_on:
    - db
    - redis

  db:
    image: postgres:13
    environment:
    - POSTGRES_USER=user
    - POSTGRES_PASSWORD=password

  redis:
    image: redis:alpine
```

Start the application: `docker compose up`

Scale the application: `docker compose up –scale web=3`

07

Advanced Docker

Overview

- **Multi stage builds**
- **Configuration and Environments**
- **Advanced networking**
- **Docker Swarm**
- Kubernetes

Multi stage builds

- It is common to end up with large images
  - When most of their content is not being used
- Typical when a compilation phase must be carried out
  - To obtain the executable that we will run
    - The compiler chain is useless afterwards
- Multi-stage builds let us build in one container
  - Generating an image without unnecessary content
  - In the Dockerfile:
    - A set of instructions build
    - Another set of instructions package

Multi stage builds: Example

```
# Stage 1: Build the application
FROM node:16 AS builder
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Create the final image
FROM node:16-alpine
WORKDIR /app
COPY --from=builder /app/dist /app
CMD ["node", "index.js"]
```

- **Stage 1**
  - **Contains all tools and dependencies needed to build the application**
- **Stage 2**
  - **Contains only the build executable and the needed runtime (if any)**

Multi stage builds: advantages

- Smaller images
  - Only the necessary files for running are left within the image
- Security
  - By leaving out build tools and unnecessary dependencies, the attack surface si reduced, increasing the security
- Simplified CI/CD pipelines
  - Multi-stage builds simplify constructing pipelines
    - The same file contains the build and the run

Configuration and Environments

- Applications need to be configured differently
  - Depending on how they need to be run
- E.g.
  - Production vs Development
  - Credentials to access other services
  - Resources available
- Typically
  - Through environment variables
  - Through mapped configuration files (from host)

Environment variables

- Seen in docker compose
- Available also in docker run

```
docker run -e APP_ENV=production -e DB_USER=admin -e DB_PASS=password myappImage
```

- Can also use an .env file

```
APP_ENV=production
DB_USER=admin
DB_PASS=password
```

```
docker run –env-file .env myappImage
```

- Easily switch configuration by changing environment variables
- Keep configuration out of Dockerfiles.

Advanced networking

- Bridge networking (default)
  - Containers on the same host communicate through the same private local network domain
  - Useful for development
- Host networking
  - The container shares the host network namespace
  - Same stack (interfaces, routing, firewall) as host
- Overlay
  - Used in multi-host deployments (swarm, Kube)
  - “overlay” network over the hosts shared network
    - Local domain network, containers on it.

Docker orchestration

- Docker compose is good for just one host
- There are many use cases that need to orchestrate container deployments over multiple hosts
- Several tools exist for this.
  - Docker swarm
  - Kubernetes
  - . . .

Docker swarm

- Native to docker: Swarm mode is built into Docker
  - Easy to setup and manage clusters of docker hosts
- Declarative service model
  - Declare the goal state of a set of services (based on containers)
    - Swarm ensures the desired state is reached.
- Simpler than kubernetes

```
docker swarm init
docker service create --name web --replicas 3 -p 80:80 nginx
```

Kubernetes

- Advanced Orchestration
  - Powerful extensible orchestration
  - Can manage thousands of containers across hundreds of hosts
- Fine grained control
  - Detailed separate control over.
    - Scaling, networking, security, service discovery
- Popular for production
  - Standard for orchestrating containerized apps at scale
  - Used by large orgs for production workloads

Kubernetes vs Swarm

- Ease of use
  - Swarm, for simple scenarios
  - Kubernetes has a steeper learning curve
    - More complex, easier to get it wrong
- Scaling
  - Kubernetes is preferred for large scale operations
  - Swarm is simpler in smaller scenarios.

08

Best Practices

Overview

- Security best practices
- Optimizing Dockerfiles for smaller images
- Managing Logs and scaling in production
- Dealing with container sprawl and performance issues

Security best practices

- Run containers as non-root users
  - By default they run as root user
  - Risk if container is compromised

Example Dockerfile:

```
RUN useradd –m myuser
USER myuser
```

Security best practices

- Limit container capabilities
  - It is possible to provide containers with extra capabilities to exercise system calls.
  - It is best to remove all such capabilities
    - To avoid breaking isolation

```
docker run --cap-drop=ALL --cap-add=NET_ADMIN myimage
```

- Only the NET_ADMIN capability is given
- All other are dropped

Security best practices

- Use official & minimal base images
  - From trusted sources, like docker hub
  - Risk if container is compromised
- Regularly update images
  - Pick up security patches
- Scan images for vulnerabilities
  - Tools available:
    - Clair,
    - Trivy
    - docker scan

```
docker scan myimage
```

Optimize for smaller images

- Use multi-stage builds (seen before)
- Minimize layer creation
  - Mix multiple “RUN” statements into one
- Use .dockerignore file
  - To keep unwanted files out of the image

Manage logs in production

- **Centralized logging**
  - **Log drivers:** json-file, syslog, journald, fluentd, gelf

```
docker run  --log-driver syslog myappImage
```

- **Log rotation**
  - **Ensure it is set up to prevent consuming too much disk**
  - **Configure docker in daemon.json:**

```
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  }
}
```

Dealing with Performance issues

- In large deployments, *container sprawl* can become an issue
- Monitor resource usage
  - Tools: *Prometheus*, *Grafana*
    - Help track CPU and memory usage across all containers
- Use docker prune commands
  - To remove unused objects
    - Images, containers, networks, volumes,…
- Set resource limits (CPU, memory)

Q & A