DevOps
AIDE501 – M2 – 3 Credits
2026/2027

Course Contents

- **Chapter 1: Cloud Computing**
- **Chapter 2: Containerization**
- **Chapter 3: Shell Scripting**
- **Chapter 4: DevOps**
- **Chapter 5: Orchestration**
- **Chapter 6: GitLab CI/CD**

Chapter 2:

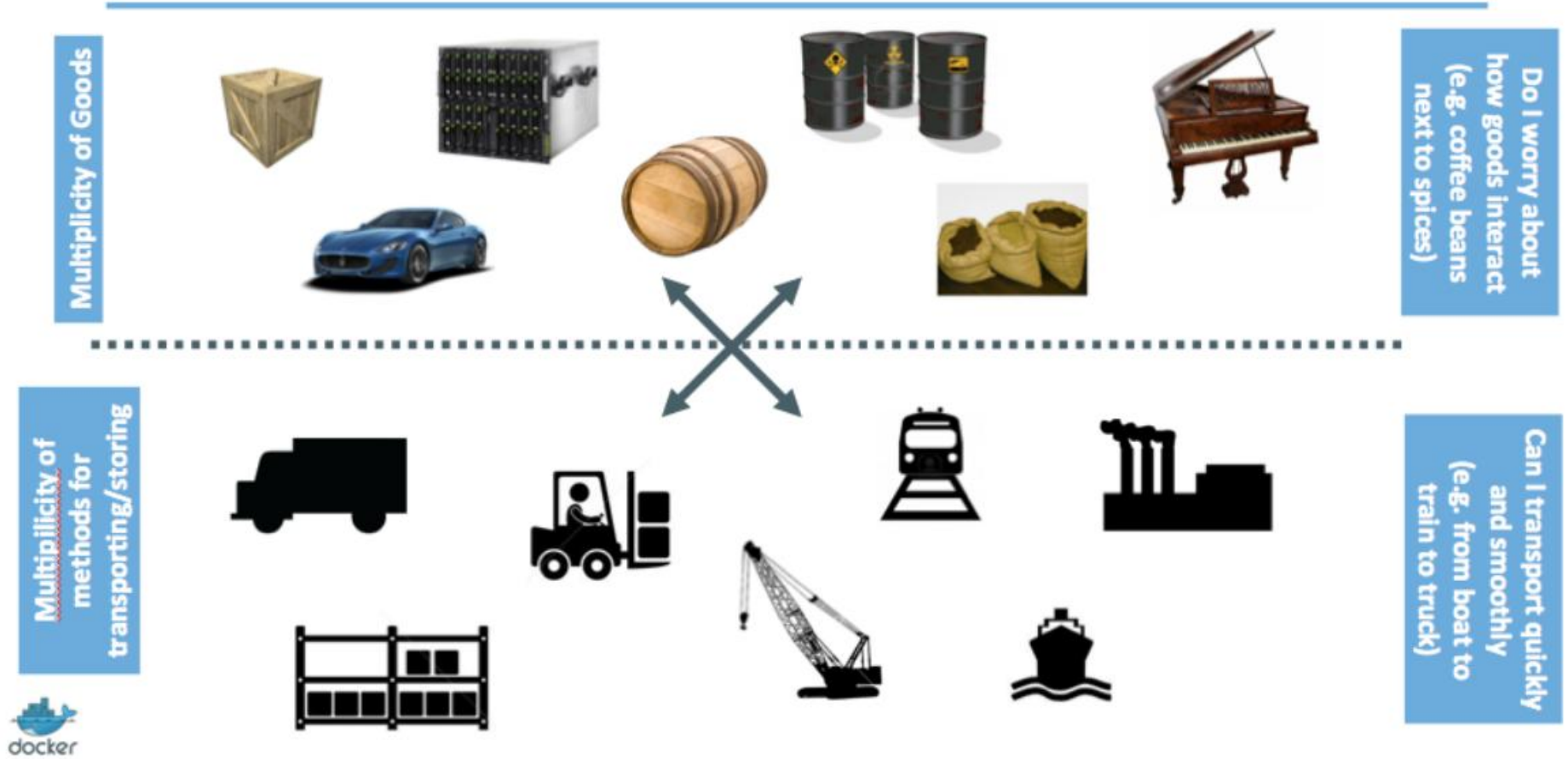
Containerization

Chap.1: Containerization

- Containerization basics (Docker, Podman)
- Registry, Docker Compose
- Managing and Securing Registries
- Identity and Access Management (IAM)
- Authentication, Authorization, API

Containerization

Containerization



Intermodal shipping containers



A new shipping ecosystem





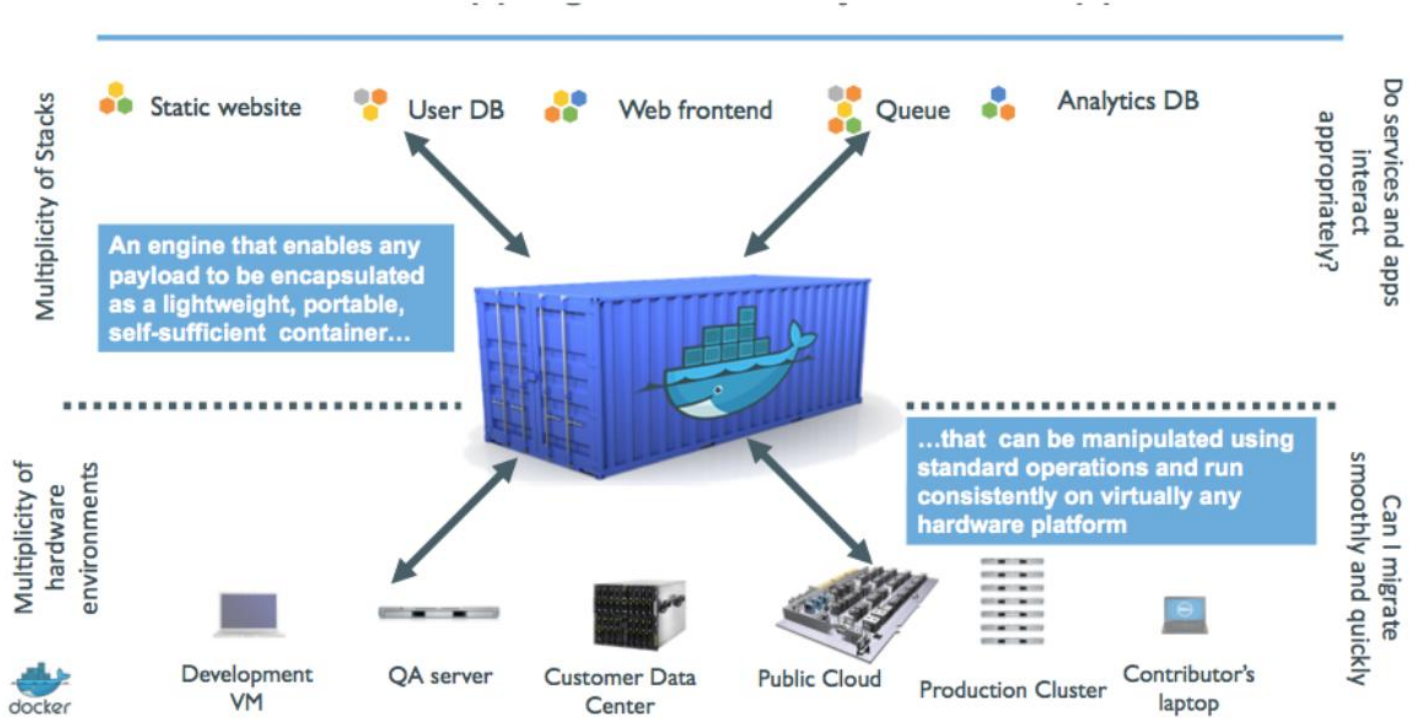
- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)

→ massive globalization

- 5000 ships deliver 200M containers per year

docker

A shipping container system for apps



Containerization basics

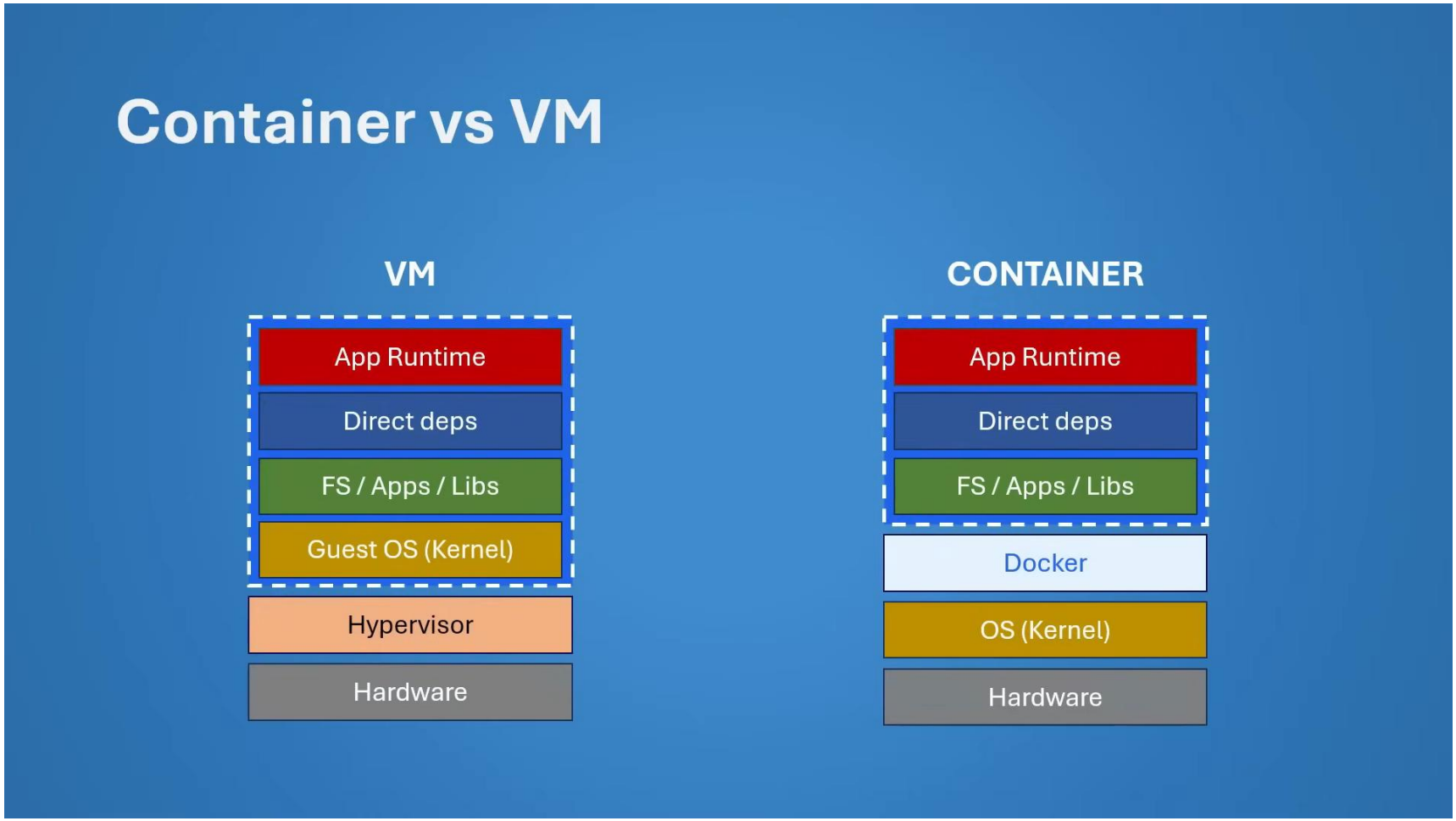
- **Containerization:**

  A lightweight virtualization technology that packages an application and its dependencies into a self-contained unit called a container. This ensures consistency across different environments.

  It is the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a container—that runs consistently on any infrastructure.

  So, no matter where you run the container (your laptop, a cloud server, or a PaaS platform), it will behave the same way.

Container Vs VM



All containers are run by a single operating-system kernel and are thus more lightweight than virtual machines

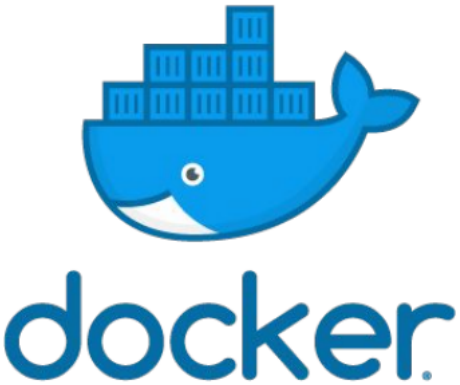
From Containers to Docker

Docker

- Extremely popular container implementation

- Easy to use authoring tools
  - Container images are created from recipe-like files
  - Images can be named, tagged and built on top of other images

- Cloud-based image distribution strategy
  - Several remote registries available (e.g. Docker Hub)
  - Client includes facilities to authenticate, push and pull images



What is Docker in PaaS?



Docker is an open-source platform that allows developers to build, package, and run applications in containers.

Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers.

Docker Benefits

Benefit	Explanation
Packaging Applications	Developers package their app and dependencies into a Docker image. PaaS providers (like Heroku, Azure App Service, or Google App Engine) can then easily deploy that image without worrying about setup or configuration.
Consistency	The same container image can run on any system or cloud.
Efficiency	Containers share the same OS kernel, so they start fast and use fewer resources.
Portability	Docker containers make it easy to move apps between environments — from development → testing → production — or even between different cloud platforms.
Isolation	Each container runs independently, so multiple apps can share the same PaaS infrastructure without interfering with each other.

Docker Benefits

Benefit	Explanation
Scalability	PaaS platforms can run multiple instances of a Docker container and scale them automatically based on load.
Automation	Fits easily into DevOps CI/CD pipelines.
Flexibility	Any language, framework, or runtime can be deployed if it’s inside a Docker container.
Microservices Ready	Ideal for running small, modular services that scale independently.

Docker Images and Containers

- **Docker Image:** A read-only (static) template containing the application code, runtime, libraries, and dependencies, Configuration files.
- It is stored on disk (e.g., Docker Hub or local).
- It is used to create containers.

- **Docker Container:** A runnable (Dynamic) instance of a Docker image.
- It exists in memory while running.
- It is created from an image.
- It is Mutable (can change during runtime) and Isolated from other containers and the host system.

Docker Images and Containers

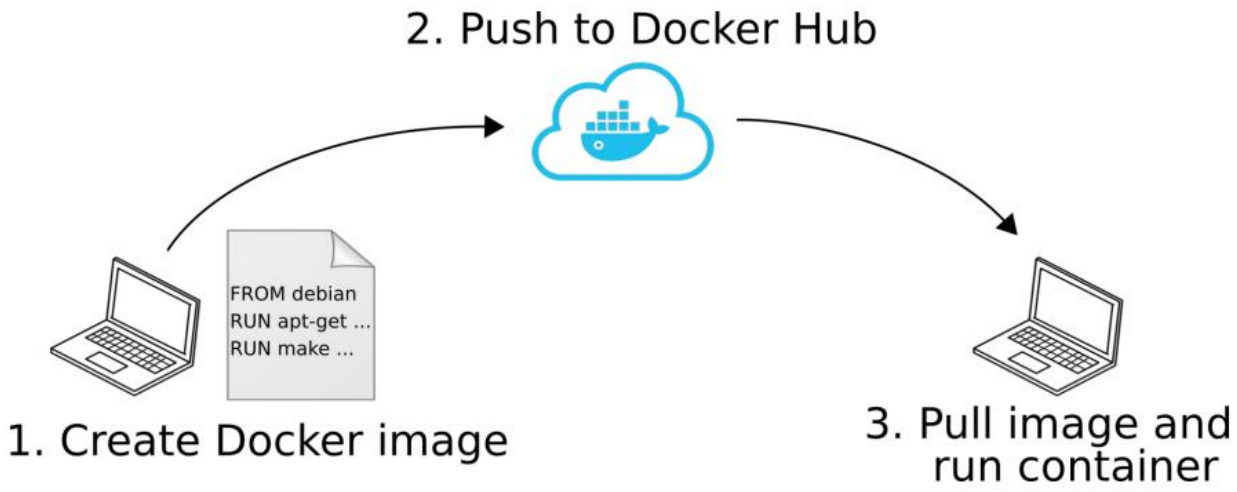
- When you start a container, Docker uses the image as a base and adds a thin read/write layer on top where data can change.

- You build an image using a **Dockerfile**.
- You run that image to create one or more containers.
- Containers share the image's base layers but have their own writable layer.

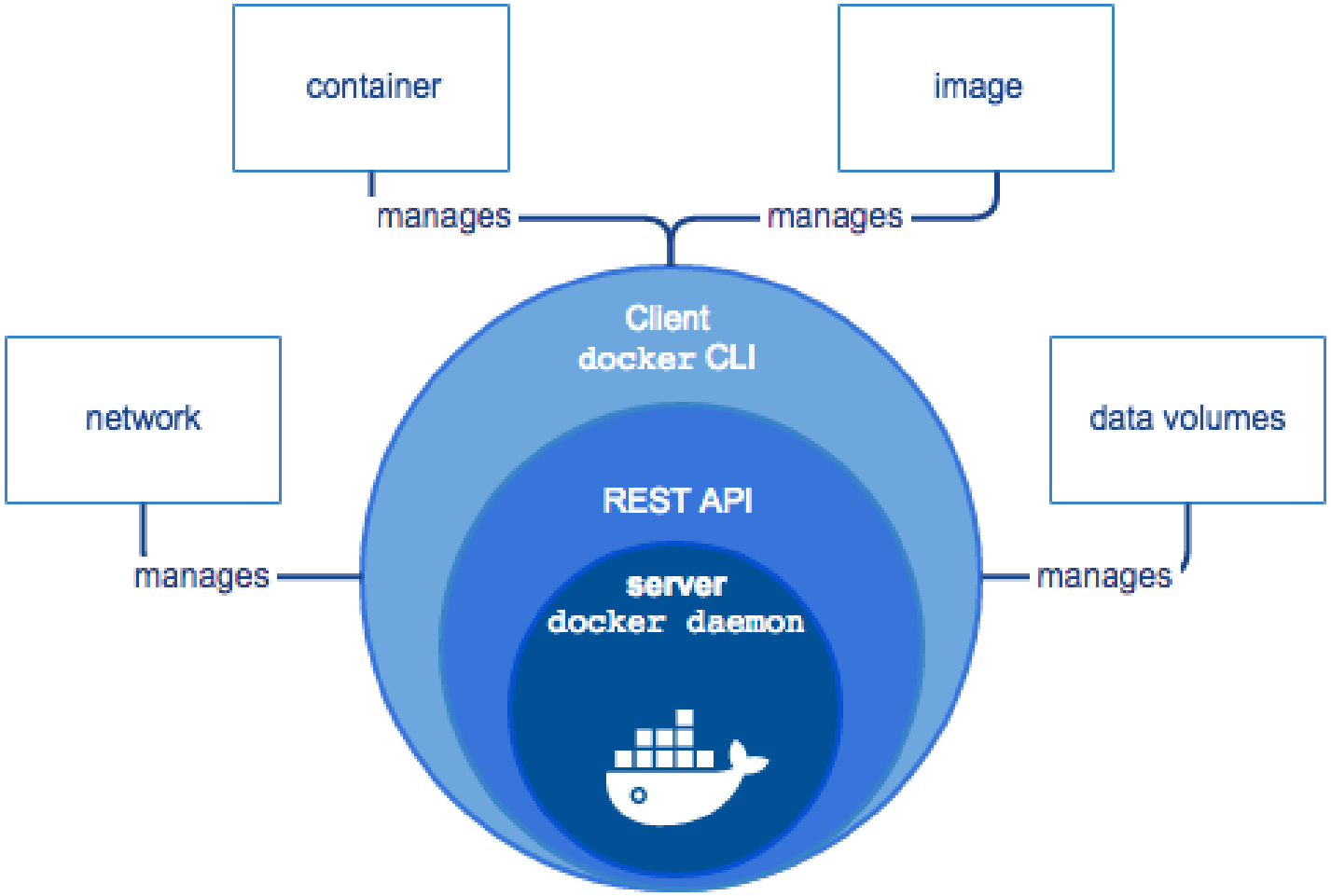
Docker Workflow

Docker workflow

1. An image is created locally from a Dockerfile
2. Push (i.e. upload) the image to a remote registry
   DockerHub is the public registry maintained by the Docker company
3. Pull (i.e. download) the image on a target machine and run the container



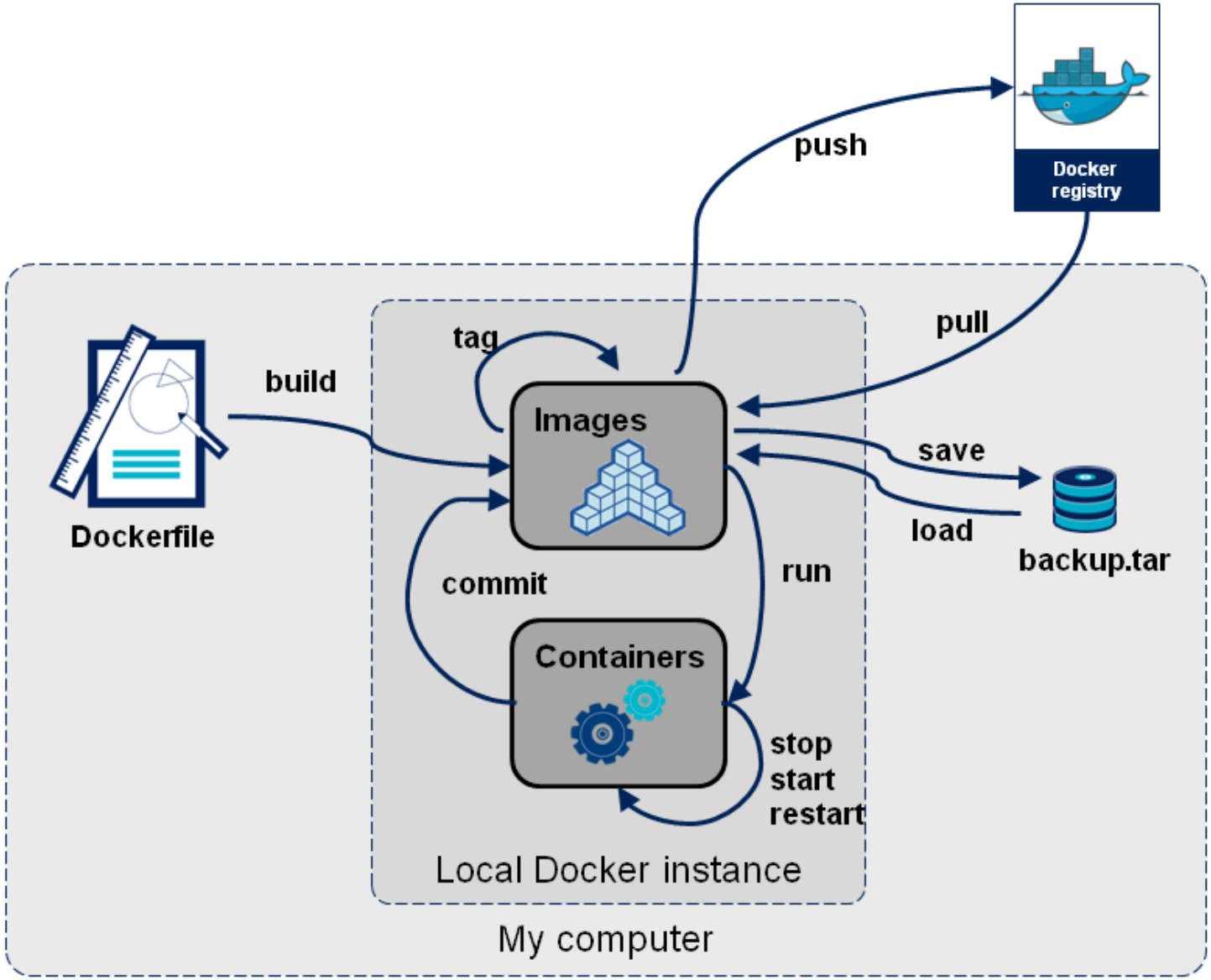
Doker Architecture



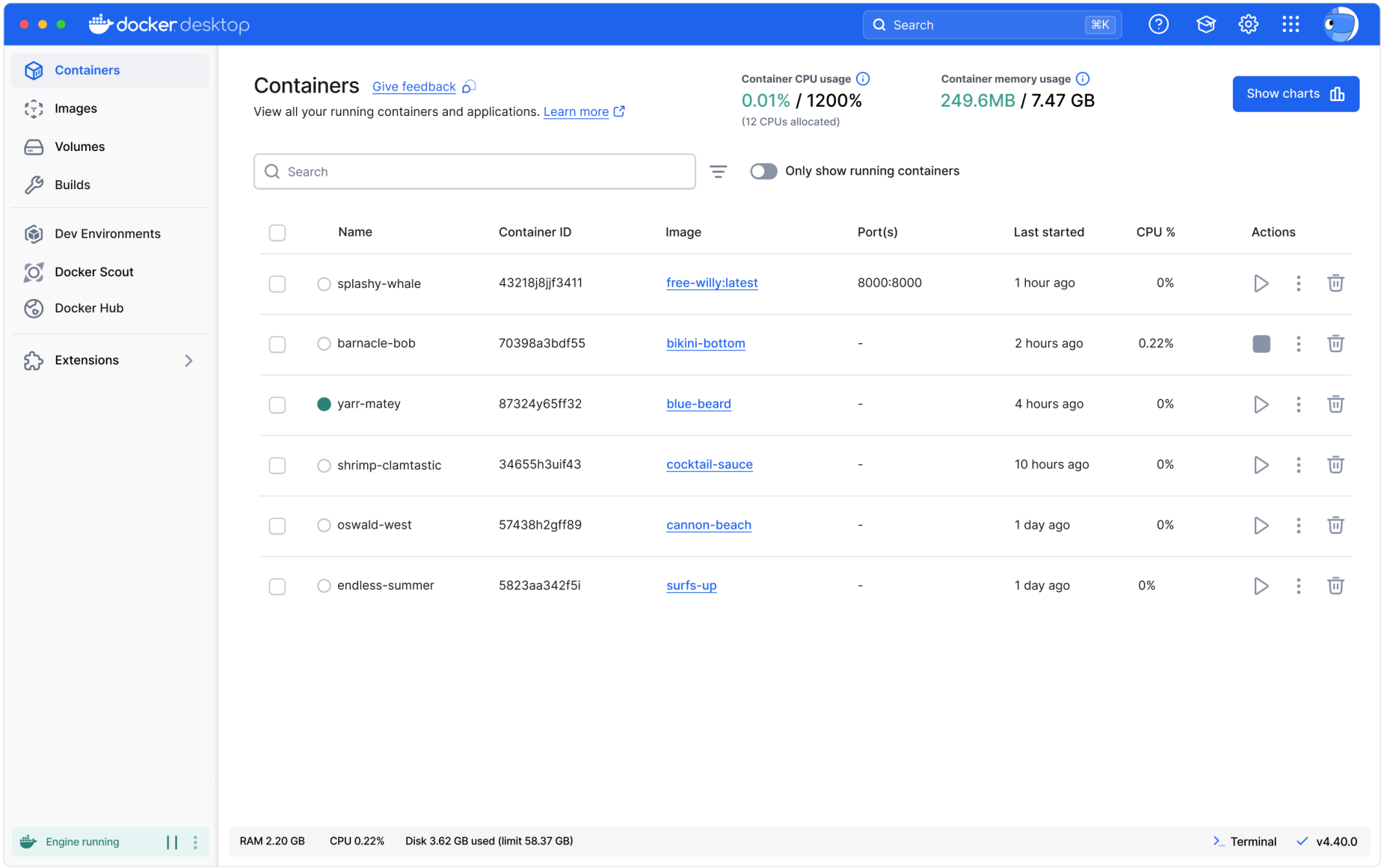
Docker Daemon

- Docker Daemon: A persistent background process that manages Docker images, containers, networks, and storage volumes.
- The Docker daemon constantly listens for Docker API requests and processes them

Docker CLi


Docker registry
push
pull
tag
build
Images
save
load
backup.tar
Dockerfile
commit
run
Containers
stop
start
restart
Local Docker instance
My computer

Docker desktop



Dockerfile & Commands

Once docker desktop has been installed the first step is creating a *dockerfile* in your project directory before it can be converted into an image.

This dockerfile contains key terms which include base image, environment setup, working directory, copy files, run commands, expose ports and startup command.

Base Image: Specifies the base image in which your custom image will be built. Can be ubuntu: 20.04, or python 3, etc.

Environment Setup: Set environment variables that contain private sensitive information.

Dockerfile & Commands

Working Directory: Specify the directory inside the container where the execution application files will be placed.

Copy Files: COPY or ADD into the container file system network. Ex.) requirements.txt

Commands: RUN will execute commands inside the container during the image building process. Ex.) RUN pip install -r requirements.txt

Expose Ports: Specify the port where project will be viewed. Ex.) EXPOSE 5000

Startup Command: CMD or ENTRYPOINT will identify the command when a container started from an image. Treat it as a main command to run the application

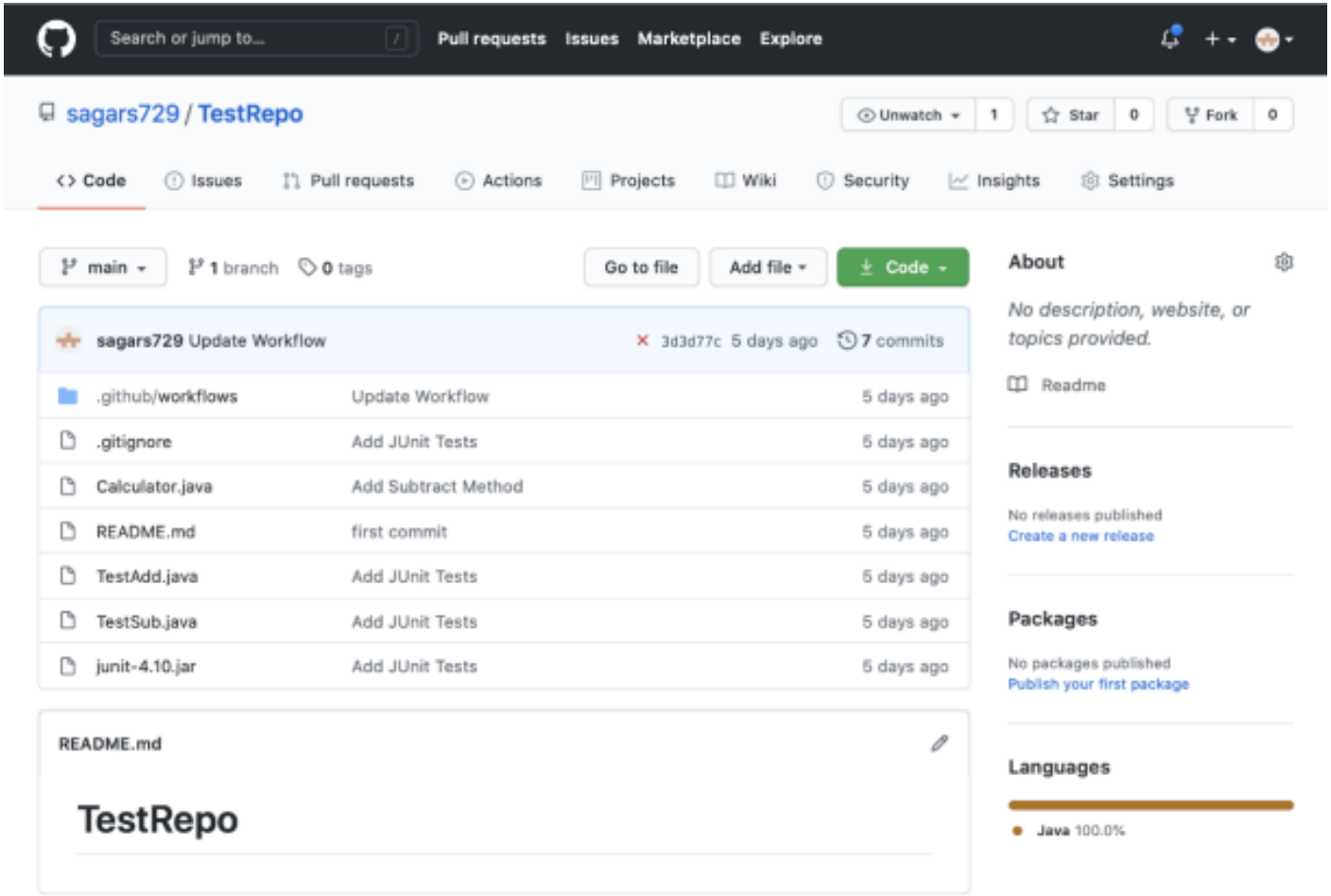
Example of Docker commands

- https://docker-curriculum.com/

Command	Purpose
`docker images`	List images
`docker ps`	List running containers
`docker ps -a`	List all containers
`docker build -t name .`	Build an image
`docker run name`	Run a container
`docker stop ID`	Stop a container
`docker rmi name`	Delete an image
`docker rm ID`	Delete a container
…	

Example : Dockerize an Application

- Dockerize your own application in a custom docker image.
- Let's dockerize the code in the following repository : a calculator class and two Junit tests for Add and Subtract:



Dockerfile

Adding A Dockerfile

To create a custom docker image. We need to create a Dockerfile. The dockerfile specifies how our image should be built.

```
FROM openjdk

RUN useradd -ms /bin/bash ojdk

RUN mkdir -p /home/ojdk/app/ && chown -R ojdk:ojdk
/home/ojdk/app

WORKDIR /home/ojdk/app

COPY *.java ./
COPY junit-* ./

USER ojdk

RUN javac -cp "junit-4.10.jar:." *.java

COPY --chown=ojdk:ojdk . .

CMD [ "java", "-cp", "junit-4.10.jar:.",
"org.junit.runner.JUnitCore", "TestAdd", "TestSub"]
```

Dockerfile

Dockerfile

Dockerfile Syntax

- **From** - The base image to use
- **Run** - Runs commands when building the docker image
- **Workdir** - Specifies the directory that commands are run from
- **User** - Switches users
- **Copy** - Copies Files
- **CMD** - Runs commands when running the container

```
FROM openjdk

RUN useradd -ms /bin/bash ojdk

RUN mkdir -p /home/ojdk/app/ && chown -R ojdk:ojdk
/home/ojdk/app

WORKDIR /home/ojdk/app

COPY *.java ./
COPY junit-* ./

USER ojdk

RUN javac -cp "junit-4.10.jar:." *.java

COPY --chown=ojdk:ojdk . .

CMD [ "java", "-cp", "junit-4.10.jar:.",
"org.junit.runner.JUnitCore", "TestAdd", "TestSub"]
```

Dockerfile

Dockerfile

Dockerfile Explained

1. Use the OpenJDK image to have a pre-configured java environment
2. Add a new user “ojdk” that we will be using for executing scripts
3. Create a directory that will contain our files and give permission to our user
4. Change the working directory to the directory we created
5. Copy the java and junit files
6. Switch to user ojdk
7. Compile all of our code
8. Copy files to the working directory and give permissions to ojdk
9. Run all tests

```
FROM openjdk

RUN useradd -ms /bin/bash ojdk

RUN mkdir -p /home/ojdk/app/ && chown -R ojdk:ojdk
/home/ojdk/app

WORKDIR /home/ojdk/app

COPY *.java ./
COPY junit-* ./

USER ojdk

RUN javac -cp "junit-4.10.jar:." *.java

COPY --chown=ojdk:ojdk . .

CMD [ "java", "-cp", "junit-4.10.jar:.",
"org.junit.runner.JUnitCore", "TestAdd", "TestSub"]
```

Dockerfile

Dockerfile

- Build the Docker image
- **docker build -t my-calculator-app .**
  - **docker build**: builds a Docker image.
  - **-t my-calculator-app**: assigns a name (tag) to the image.
  - **.**: specifies the current directory as the build context.

- Check the created image
- **docker images**

- Run a container from the image
- **docker run my-calculator-app**

Registry

- Docker is the container technology, while a registry is a service that stores and distributes container images. You can build an image using Docker and push it to different registries.

- Besides Docker Hub, there are several other container image registries where you can store, manage, share, and download Docker images.

Registry

1. GitHub Container Registry (GHCR)



- Website: GitHub Packages
- Address: `ghcr.io`
- Stores public and private images.
- Good integration with GitHub repositories and GitHub Actions.

2. GitLab Container Registry

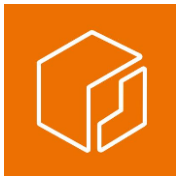


- Website: GitLab
- Address: `registry.gitlab.com`
- Integrated with GitLab projects and CI/CD.
- Supports private and public images.

Registry

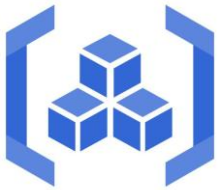
3. Amazon Elastic Container Registry (ECR)

- Website: Amazon ECR
- Cloud-based registry.
- Integrated with AWS services.
- Public and private repositories.



4. Google Artifact Registry

- Website: Google Artifact Registry
- Integrated with Google Cloud.
- Supports Docker and other package formats.



Registry

5. Azure Container Registry (ACR)



- Website: Azure ACR
- Integrated with Microsoft Azure.
- Supports private image repositories.

6. Harbor



- Website: Harbor
- Open-source registry.
- Can be installed on your own server.
- Suitable for organizations and universities.

Registry

7. Red Hat Quay



- Website: [Red Hat Quay]
- Enterprise container registry.
- Image management and security features.

How to use another registry?

The Docker commands remain almost the same.

For example, using GitHub Container Registry:

Step 1. Login

```
docker login ghcr.io -u YOUR_USERNAME
```

Step 2. Build your image

```
docker build -t myapp .
```

Step 3. Tag the image

```
docker tag myapp
  ghcr.io/YOUR_USERNAME/myapp:1.0
```

Step 4. Push the image

```
docker push ghcr.io/YOUR_USERNAME/myapp:1.0
```

Step 5. Pull the image from another computer

```
docker pull ghcr.io/YOUR_USERNAME/myapp:1.0
```

Podman

- **Podman** (Pod Manager) is an open-source container management tool used to build, run, manage, and distribute containers and container images.
- It is an alternative to Docker and provides many similar commands and features.
- It is developed by Red Hat and is widely used in Linux environments.



Podman

- The most important difference is that Docker traditionally uses a **background daemon**, while Podman uses a **daemonless** architecture.
- One interesting feature is that many Docker commands have equivalent Podman commands.

Docker	Podman
`docker pull nginx`	`podman pull nginx`
`docker images`	`podman images`
`docker build -t myapp .`	`podman build -t myapp .`
`docker run nginx`	`podman run nginx`
`docker ps`	`podman ps`
`docker stop ID`	`podman stop ID`
`docker tag`	`podman tag`
`docker push`	`podman push`

Docker Compose

- Docker Compose is a tool that allows you to define, configure, and run multiple Docker containers as one application.
- Instead of starting each container separately using several `docker run` commands, you describe all the services in a single YAML file called `compose.yaml` (or `docker-compose.yml`).

- Docker Compose is particularly useful when an application requires multiple services, such as:
- A web server (Nginx).
- A backend application (Python, Java, Node.js).
- A database (MySQL or PostgreSQL).
- A cache (Redis).

Docker Compose

- Imagine you want to build a web application with two components:
  - A web application.
  - A MySQL database.
- Without Docker Compose, you must create and configure each container separately.
- With Docker Compose, you define both services in one file and start them with one command.

Docker Compose

- **YAML** (a recursive acronym for **YAML Ain't Markup Language**) is a human-readable data serialization language commonly used for configuration files, cloud infrastructure, and CI/CD pipelines.
- **Key Features & Syntax**
  - **Indentation:** Uses space characters (never tabs) to define data hierarchy and nesting.
  - **Key-Value Pairs:** Expressed using a colon followed by a space (e.g., app: user-authentication).
  - **Lists / Sequences:** Indicated with a dash and a space (e.g., - item).
  - **Comments:** Started using the hash symbol (#).
  - **JSON Compatibility:** YAML is a strict superset of JSON, meaning any valid JSON file is also valid YAML.
  - **Documents:** Can separate multiple documents inside a single file using three hyphens (---).

Docker Compose

Create a Docker Compose file

Create a directory:

```
mkdir myproject
cd myproject
```

Create a file named `compose.yaml`:

```
services:                    #defines the containers.
  web:                       #the first service.
    image: nginx:latest      #specifies the image to use.
    ports:                   #maps port 8080 on your computer to port 80 in the web container.
      - "8080:80"


  database:                  #the second service.
    image: mysql:8.0
    environment:             #defines environment variables for MySQL.
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: mydb
```

Docker Compose

Run the application

From the directory containing `compose.yaml`, execute:

```
docker compose up -d
```

- `docker compose`: invokes Docker Compose.
- `up`: creates and starts the services.
- `-d`: runs the containers in the background (detached mode).

Docker will automatically:

- Download the required images if they are not available locally.
- Create the containers.
- Create a network for communication between services.
- Start the containers.

Docker Compose

Check the running containers

```
docker compose ps
```

- You should see two running services:

Service	Image	Port
web	nginx:latest	8080:80
database	mysql:8.0	3306

- Open your browser: http://localhost:8080

You should see the Nginx welcome page.

Docker Compose

Command	Description
`docker compose up`	Start all services
`docker compose up -d`	Start in background
`docker compose down`	Stop and remove containers and network. Does not normally delete named volumes, so persistent database data can remain.
`docker compose ps`	List services
`docker compose images`	List images used
`docker compose logs`	Display logs
`docker compose stop`	Stop services
`docker compose restart`	Restart services
`docker compose pull`	Download updated images
`docker compose build`	Build services from Dockerfiles

Docker Compose

- Suppose you want MySQL data to remain available even after removing and recreating the container.
- Modify your Compose file:

```
services:
  database:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: mydb
    volumes:
      - mysql_data:/var/lib/mysql
volumes:
  mysql_data:
```

```
#mysql_data is a named volume.
   /var/lib/mysql is the
   directory where MySQL stores
   its data. The volume
   preserves database data
   independently of the
   container.
```