Cloud Computing Fundamentals
IN401 – M1S1 – 5 Credits

Course Grading

- Partial Exam
- Final Exam
- Second Session

Course Contents

- Chapter 1: Introduction to Cloud Computing
- Chapter 2: Infrastructure as a Service (IaaS)
- Chapter 3: Platform as a Service (PaaS)
- Chapter 4: Software as a Service (SaaS)
- Chapter 5: Cloud Native Technologies & Architectures
- Chapter 6: Cloud Security
- Chapter 7: Practical Case Studies

Chapter 3:

Platform as a Service (PaaS)

Content

- Introduction
- Application deployment on cloud platforms (Google App Engine, Heroku, Azure App Service, AWS Elastic Beanstalk)
- Scaling and load balancing concepts
- Containerization basics (Docker introduction)

Introduction

- PaaS is a cloud computing model providing a platform and environment for developers to build, deploy, and manage applications without the complexities of managing the underlying infrastructure (servers, operating systems, databases, etc.).
- The cloud provider handles all infrastructure management, allowing developers to focus solely on application development and code.
- User can deploy and customize their applications (programming languages and tools supported by the cloud provider).

Application Deployment on PaaS Platforms

1. Google App Engine.

Application Deployment on PaaS Platforms

What is Google App Engine.

- **Overview**

Google App Engine (**GAE**) is a Platform as a Service (PaaS) cloud computing platform for developing and hosting web applications in Google-managed data centers.

Google App Engine lets you run web applications on Google's infrastructure.

Easy to build.
Easy to maintain.
Easy to scale as the traffic and storage needs grow.

Free ???

Yes, free for up to 1 GB of storage and enough CPU and bandwidth to support 5 million page views a month. 10 Applications per Google account.

Application Deployment on PaaS Platforms

What is Google App Engine.

- **Programming languages support**

Java:

- App Engine runs JAVA apps on a JAVA 7 virtual machine (currently supports JAVA 6 as well).
- Uses JAVA Servlet standard for web applications:
  - WAR (Web Applications ARchive) directory structure.
  - Servlet classes
  - Java Server Pages (JSP)
  - Static and data files
  - Deployment descriptor (web.xml)
  - Other configuration files
- Getting started :
  https://developers.google.com/appengine/docs/java/gettingstarted/

Application Deployment on PaaS Platforms

What is Google App Engine.

- Programming languages support

Python:

- Uses WSGI (Web Server Gateway Interface) standard.
- Python applications can be written using:
    - Webapp2 framework
    - Django framework
    - Any python code that uses the CGI (Common Gateway Interface) standard.
- Getting started :
  https://developers.google.com/appengine/docs/python/gettingstartedpython27/

Application Deployment on PaaS Platforms

What is Google App Engine.

- Programming languages support

PHP (Experimental support):

- Local development servers are available to anyone for developing and testing local applications.
- Only whitelisted applications can be deployed on Google App Engine. (https://gaeforphp.appspot.com/).
- Getting started: https://developers.google.com/appengine/docs/php/

Application Deployment on PaaS Platforms

What is Google App Engine.

- Programming languages support

Google's Go:

- Go is a Google's open-source programming environment.
- Tightly coupled with Google App Engine.
- Applications can be written using App Engine's Go SDK.
- Getting started:

https://developers.google.com/appengine/docs/go/overview

Application Deployment on PaaS Platforms

What is Google App Engine.

- **Data storage**

App Engine Datastore:

- NoSQL schema-less object-based data storage, with a query engine and atomic transactions.
- Data object is called a “Entity” that has a kind (~ table name) and a set of properties (~ column names).
- JAVA JDO/ JPA interfaces and Python datastore interfaces.

Google cloud SQL:

- Provides a relational SQL database service.
- Similar to MySQL RDBMS.

Application Deployment on PaaS Platforms

2. Microsoft Azure App Service:

Microsoft Azure App Services is an example of a PaaS, abstracting away the complexities of infrastructure management so that developers can efficiently develop and deploy enterprise-ready applications.

Microsoft Azure's PaaS offering for hosting web applications, APIs, and mobile backends.

It supports various languages and frameworks and offers features like continuous deployment, scaling, and integration with other Azure services.

Application Deployment on PaaS Platforms

Microsoft Azure App Service Features:

- Managed Infrastructure: Azure manages the servers, operating systems, runtime components, and network.

- Developer-Focused: Developers can focus on writing code and building applications, not managing servers or updates.

- Built-in Tools: App Services provides pre-built components, security features, and development tools to accelerate the application lifecycle.

Application Deployment on PaaS Platforms

- Hosting Environment: It supports various applications, including HTTP-based web apps, REST APIs, and mobile backends.

- Scalability: Applications can be easily scaled up or down to meet demand.

- Cross-Platform Support: App Services runs on both Windows and Linux environments and supports multiple programming languages like .NET, Java, Python, Node.js, and PHP.

Application Deployment on PaaS Platforms

3. Heroku:

A popular PaaS known for its developer-friendly workflow.

Deployment typically involves connecting a Git repository, pushing code, and Heroku automatically builds and deploys the application within "dynos" (isolated containers).

https://www.heroku.com/platform/

Application Deployment on PaaS Platforms

4. AWS Elastic Beanstalk:

Amazon Web Services' PaaS for deploying and scaling web applications and services.

Developers upload their application code, and Elastic Beanstalk automatically handles the provisioning of resources (EC2 instances, load balancers, etc.), deployment, and scaling.

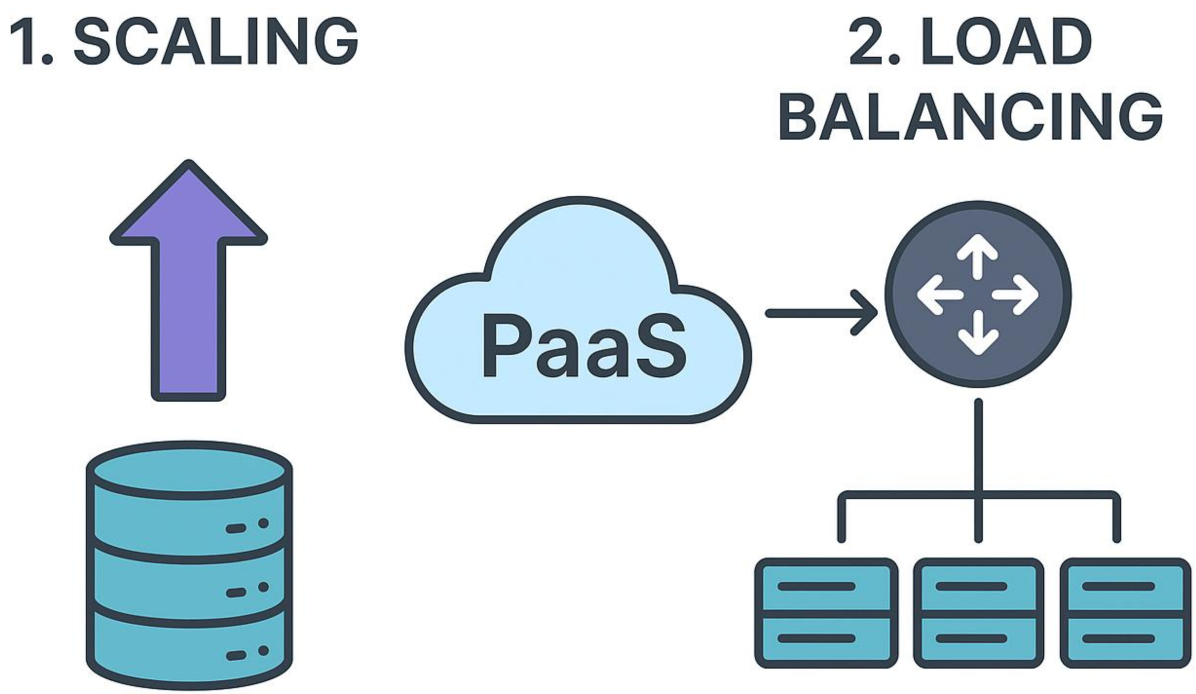
Scaling and Load Balancing

Scaling Vs Load Balancing in PaaS

- In a Platform as a Service (PaaS), **scaling** refers to the ability to automatically adjust the amount of resources (like server instances) available to an application based on its current demand, while **load balancing** is the process of distributing incoming network traffic across multiple instances to ensure no single server is overwhelmed, thereby improving performance, reliability, and fault tolerance.

- PaaS environments simplify these processes by providing managed services that handle infrastructure, allowing developers to focus on their applications rather than managing the underlying hardware or complex configurations.

Scaling Vs Load Balancing in PaaS



Scaling

- **Scaling:**
  The ability to adjust computing resources to meet varying demand.

- **Vertical Scaling (Scale Up):**
  Increasing the resources (CPU, RAM) of a single server.
  Example: Upgrading a web server from 2 CPUs to 8 CPUs.

- **Horizontal Scaling (Scale Out):**
  Adding more instances of a server to distribute the workload. This is more common in cloud environments.
  Example: Adding 3 more instances of a web app in Azure App Service or AWS Elastic Beanstalk.

Scaling

•**Purpose**: To dynamically add or remove resources (like compute instances) to match the current user load.

•**How it works**: PaaS platforms typically offer auto-scaling features that monitor application performance and resource utilization.
When demand increases, the platform automatically provisions additional instances, and when demand decreases, it scales back down to save costs.

Scaling

- Most PaaS platforms (like Heroku, Google App Engine, Azure App Service) offer automatic scaling based on metrics such as:
    - CPU utilization
    - Request rate
    - Queue length

- You can set rules so that the system automatically adds or removes instances as demand changes.

Scaling

- **Benefits**:
    - **Cost Optimization**: You only pay for the resources you need.

    - **Performance**: Maintains consistent application performance even during traffic spikes.

    - **Resilience**: Increases availability by adding more resources to handle increased load or potential failures.

Load Balancing

- **Load Balancing:**

  Distributing incoming network traffic across multiple servers to ensure no single server become overwhelmed.

  Load balancers improve application availability and responsiveness.

  PaaS platforms <u>often include integrated load balancing</u>.

Load Balancing

•**Purpose**: To distribute incoming application or network traffic across multiple backend servers.

•**How it works**: PaaS platforms provide integrated load balancers that are configured to route traffic to different instances of your application. They use various algorithms to distribute the workload, preventing any single server from becoming a bottleneck.

Load Balancing

- **Techniques Used:**
  - **Round Robin:** Requests are distributed sequentially.
  - **Least Connections:** Traffic goes to the server with the fewest active connections.
  - **IP Hash:** Uses the client's IP to determine the server assignment.

Load Balancing

- **Benefits**:
  - **High Availability**: If one server fails, the load balancer can redirect traffic to the healthy ones, ensuring uninterrupted service.

  - **Fault Tolerance**: Spreads the load, making the entire system more robust against failures.

  - **Reduced Latency**: Distributes requests to available servers, which can reduce response times for users.

  - Optimize resource use, maximize throughput, and ensure application reliability and uptime.

How they work together in PaaS?

Scaling and load balancing are complementary and often work in tandem within a PaaS environment.

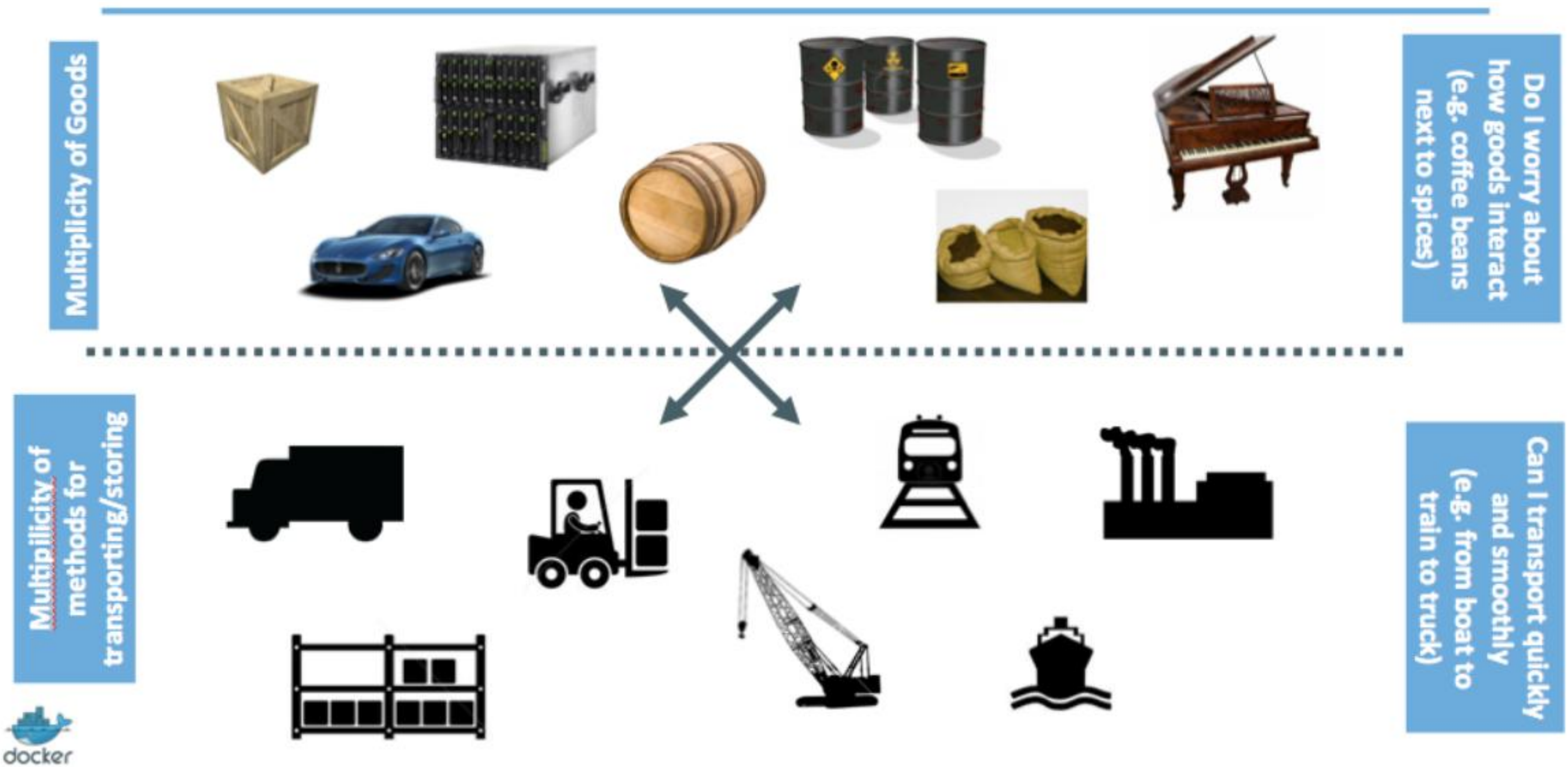
1. **Scaling** adds more servers (instances) to handle increased demand.

2. The **load balancer** then distributes the incoming traffic across this larger pool of servers.

3. When demand falls, **scaling** removes instances, and the **load balancer** adjusts to direct traffic to the remaining servers, ensuring continued efficiency and cost-effectiveness.

Containerization

Containerization



Intermodal shipping containers

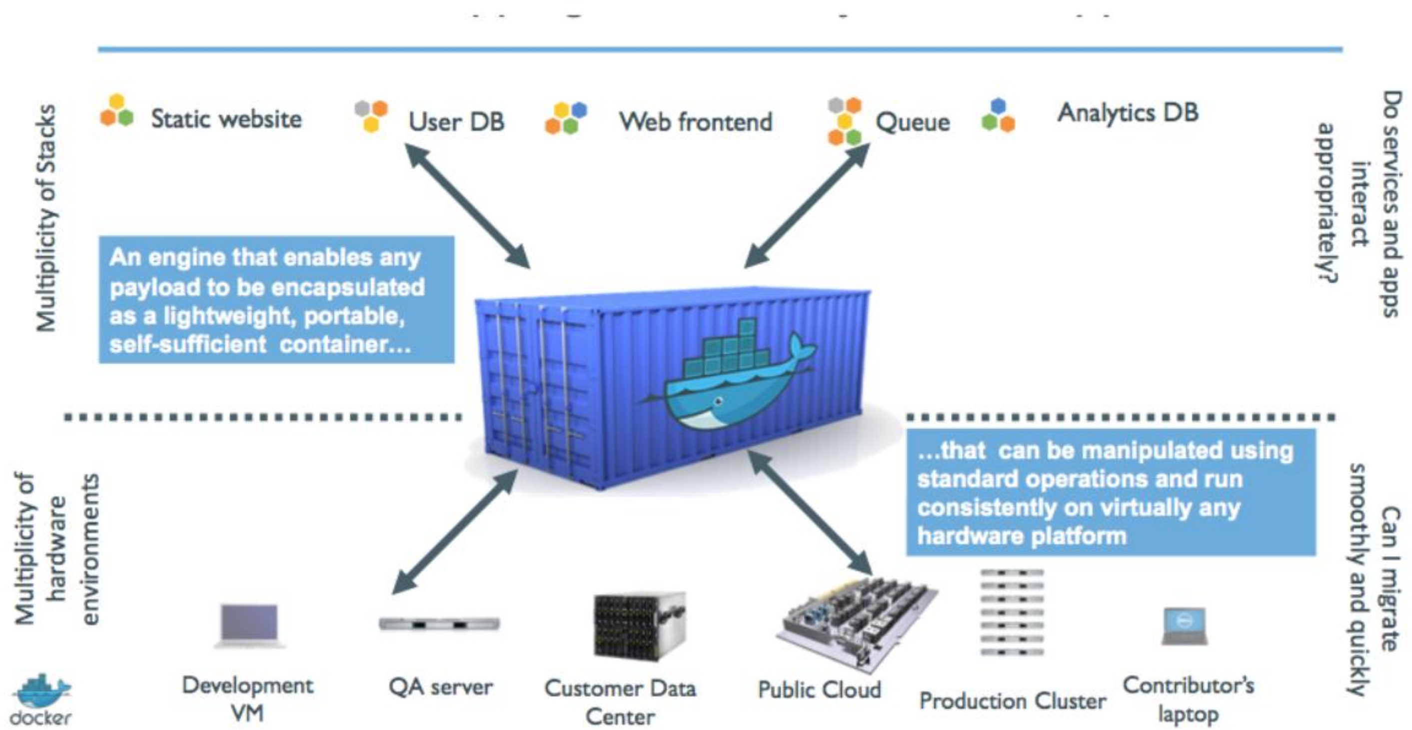
A new shipping ecosystem

- 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

A shipping container system for apps



Containerization basics

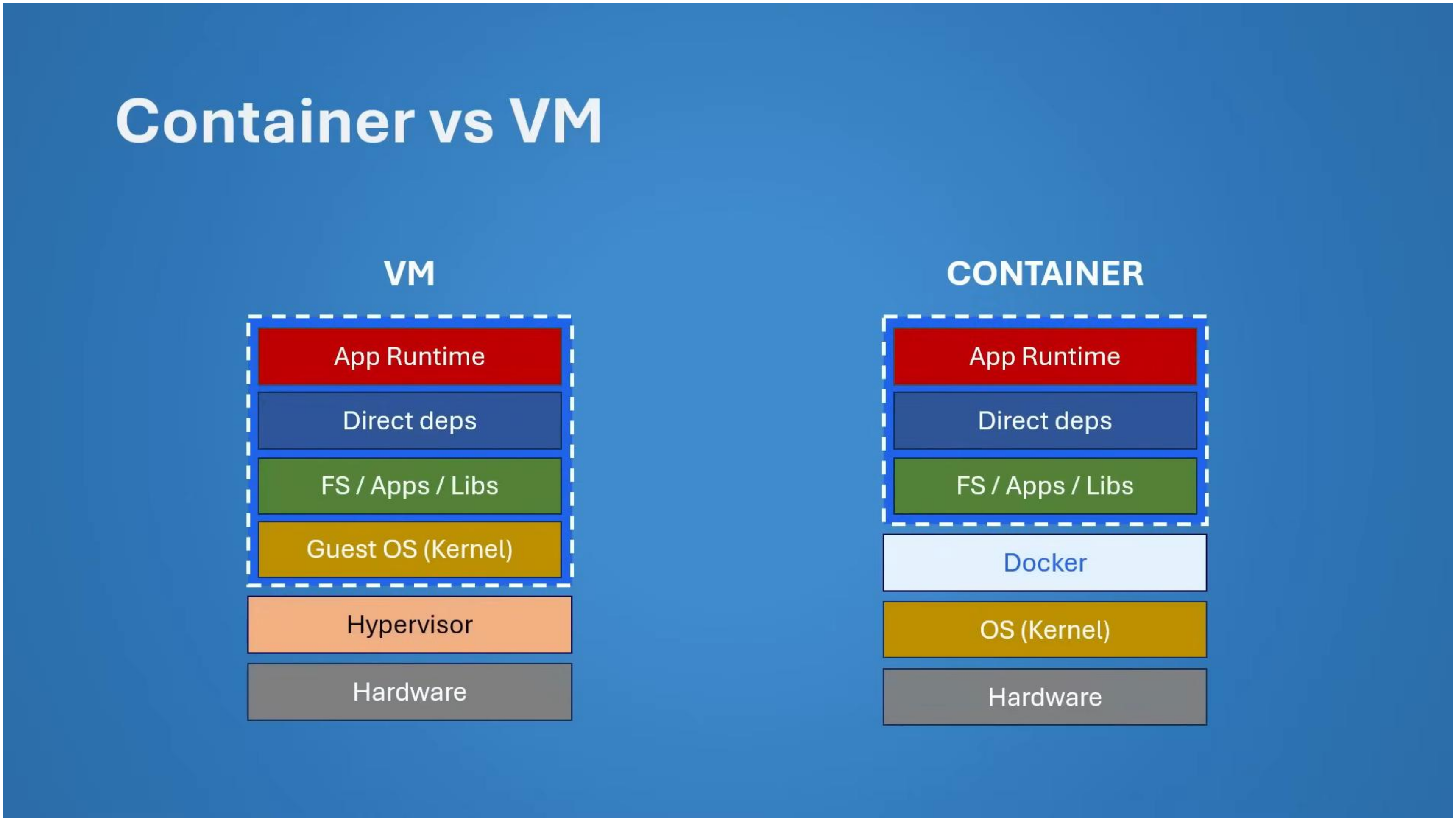
- **Containerization:**

  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.

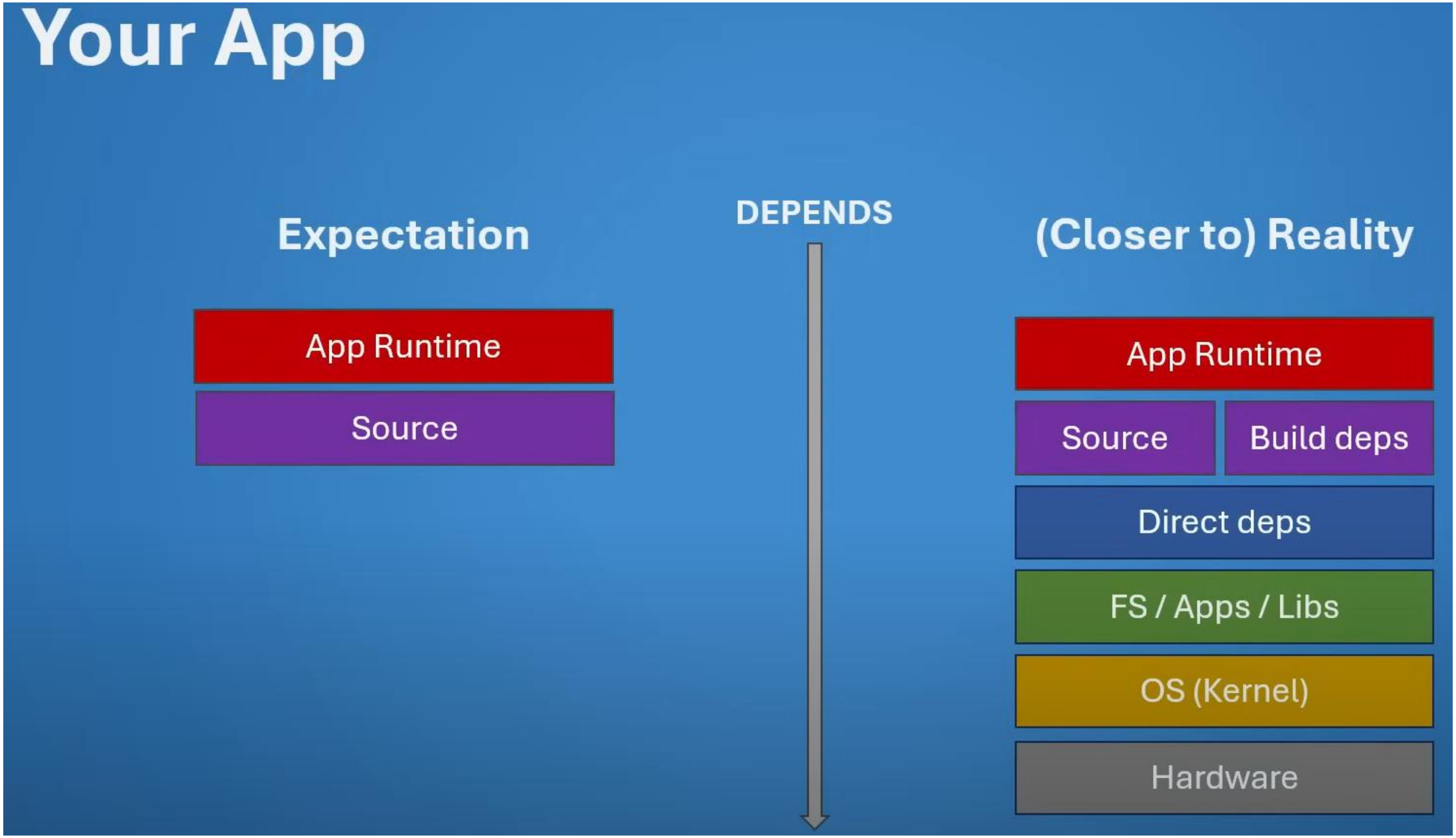
Docker Benefits

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.

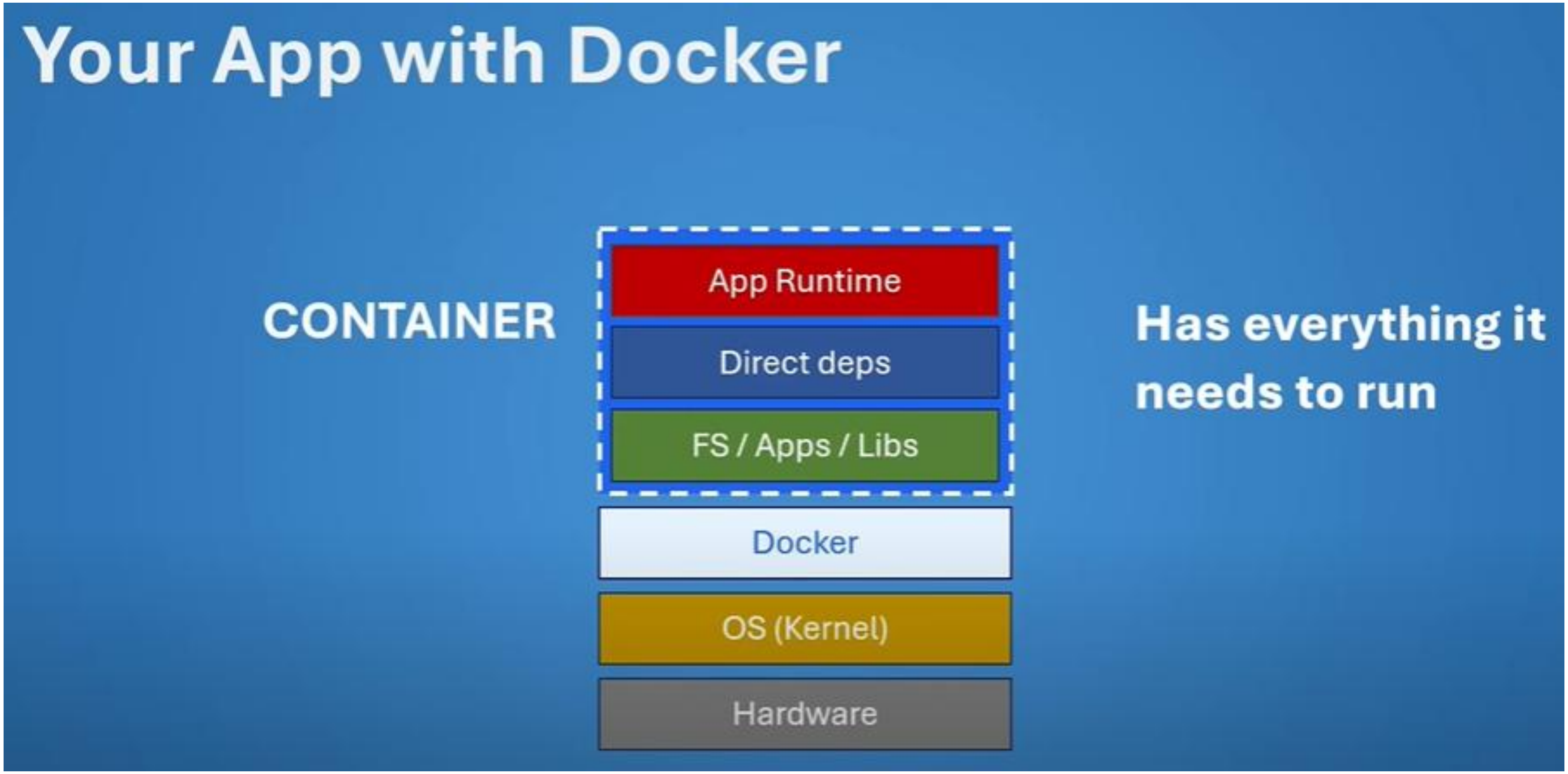
Docker Benefits

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.

Why we need docker?



Why we need docker?



Docker Images and Containers

file saved in disk

- **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.

adden fi kza container 3am yshtghlo 3le

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.

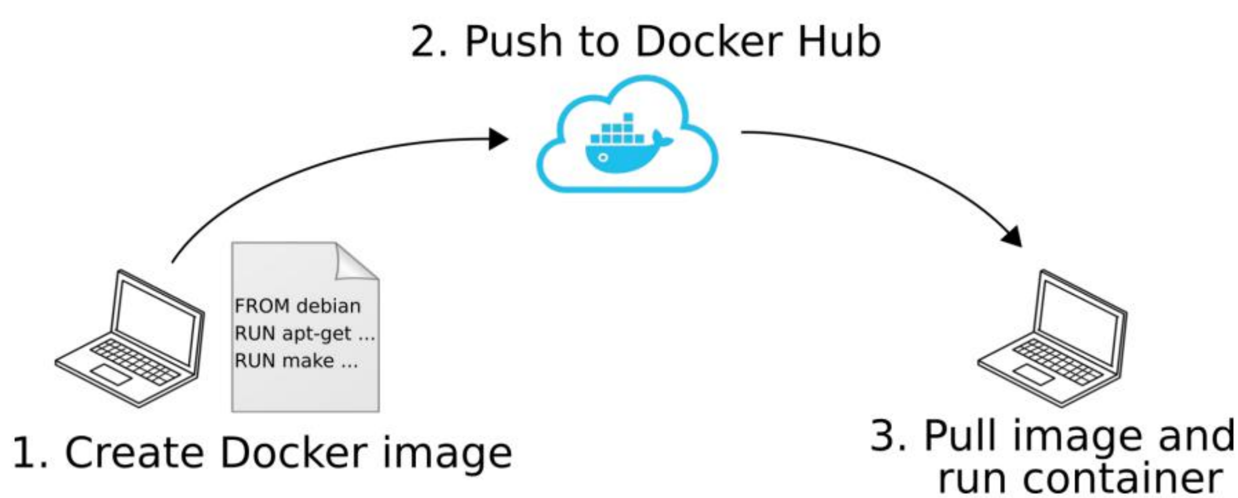
→ create an image (file libraries dependencies, ...)

- 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.

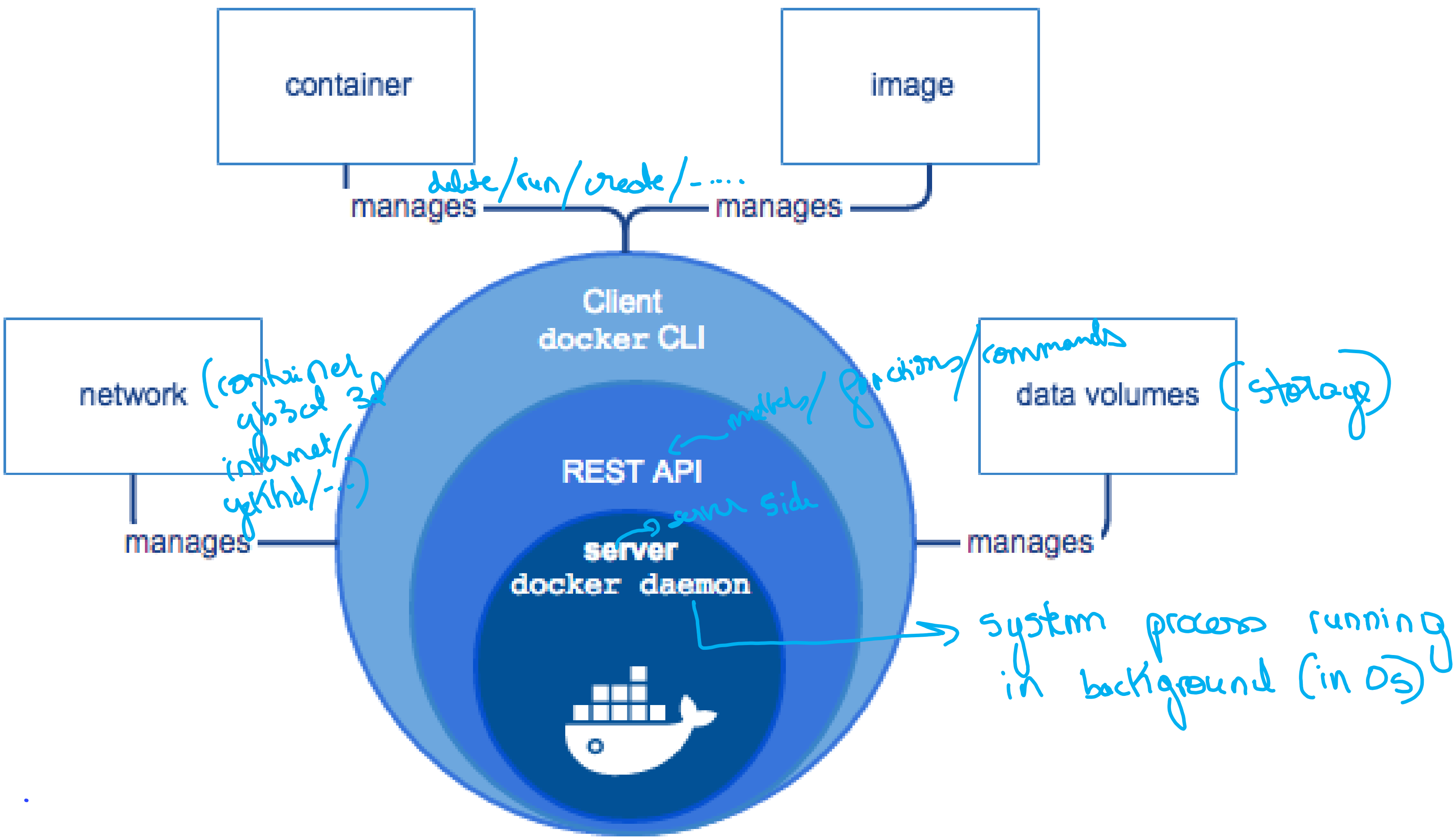
Docker Workflow

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



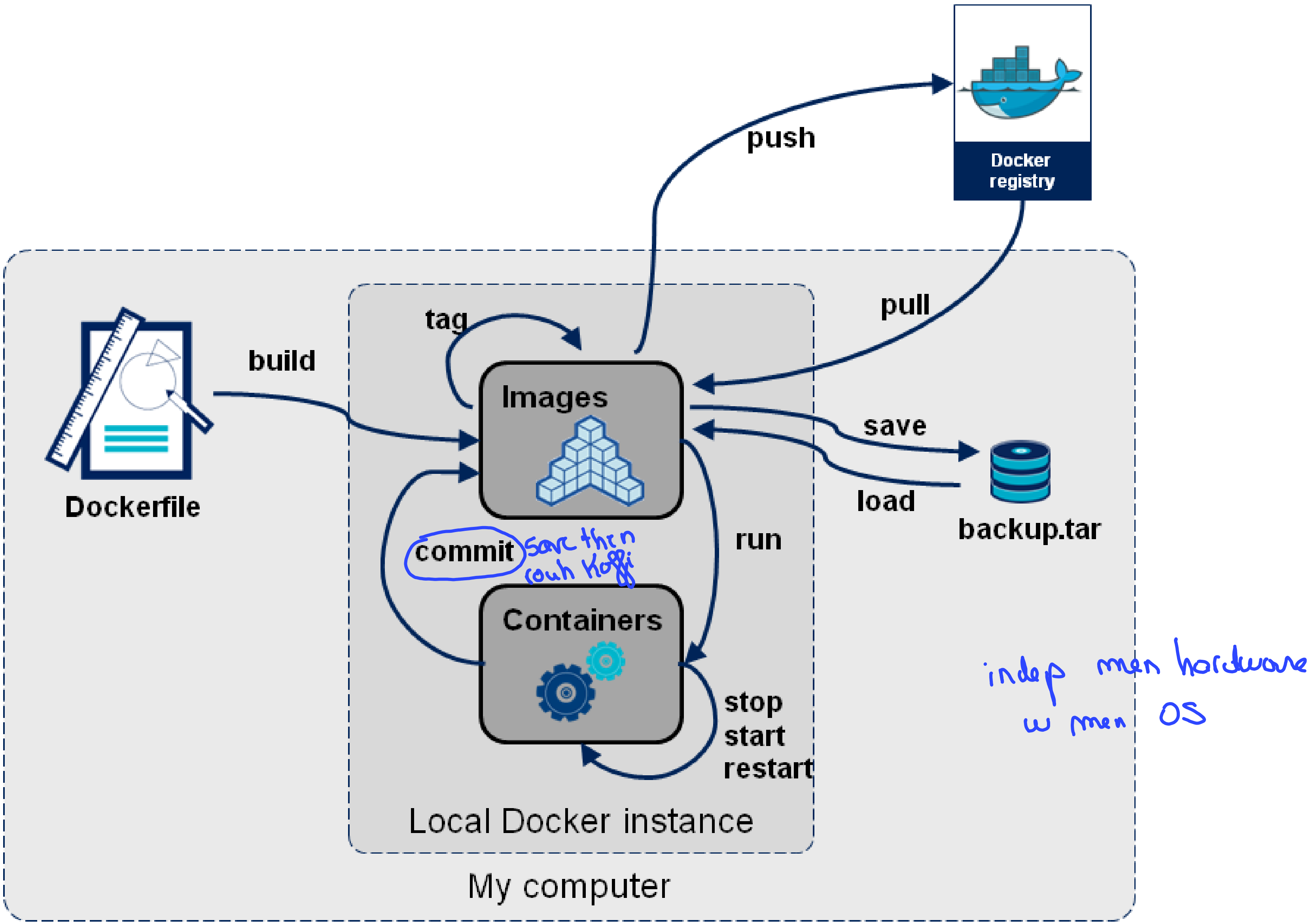
Doker Architecture



Docker Daemon

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

Docker CLi

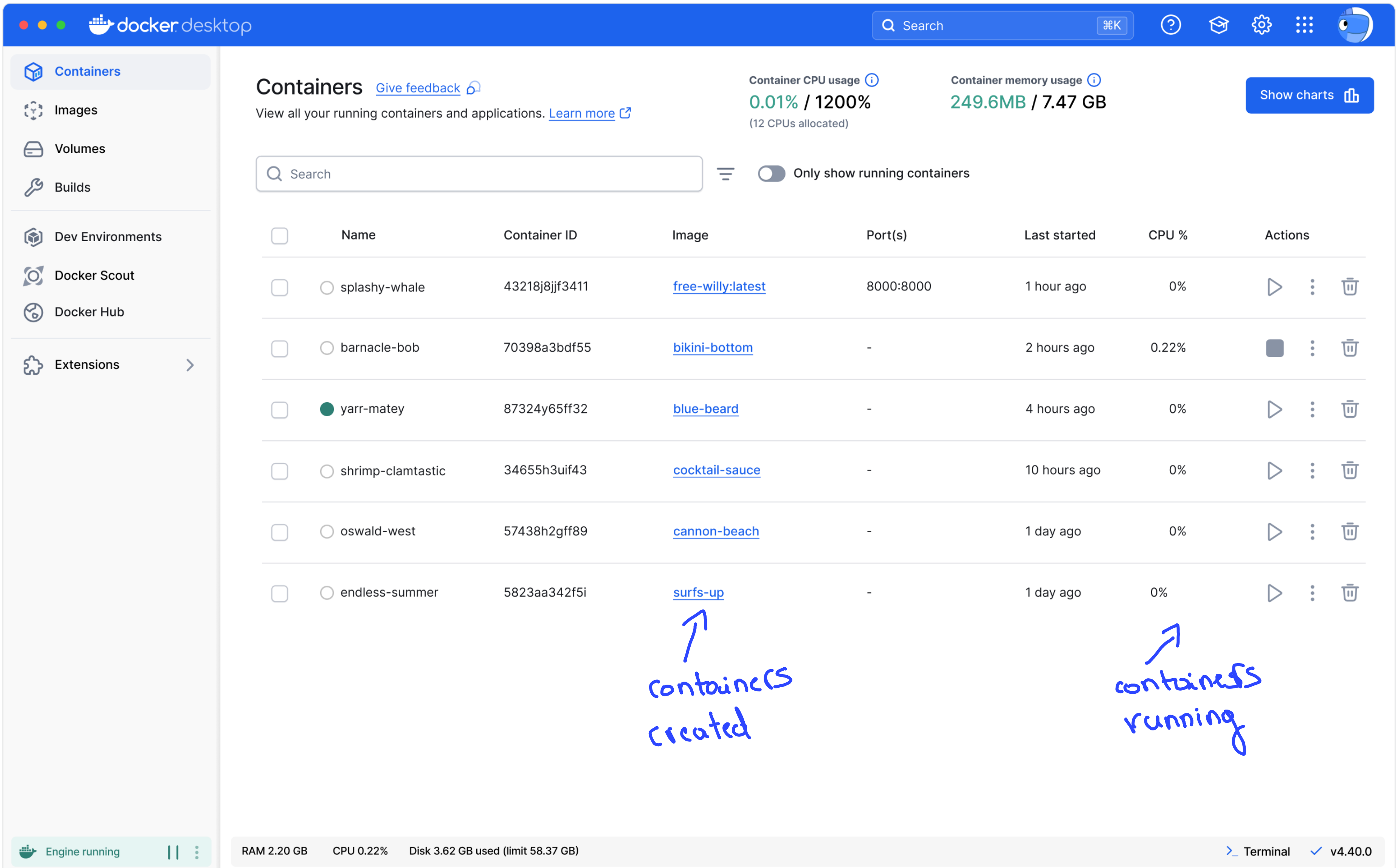


push
Docker registry
tag
build
pull
Images
save
Dockerfile
load
backup.tar
commit
save then couh koffi
run
Containers
stop
start
restart
Local Docker instance
My computer
indep mem hardware
w mem OS

Docker CLi

- docker image: manages images
- docker images: list images
- Docker create: creates a new container
- Docker Restart: Restarts a container
- Rmi**:** Force remove one or multiple images
- Docker Commit: commits changes.
- Docker build: Builds an image from docker file
- Docker Pull: Pull an image from registry
- Docker Push.
- Save**:** allows a user to save images to an archive.

Docker desktop tool to manage



containers created

containers running

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.

la n3adil Fiya
mtl mo bdna

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.

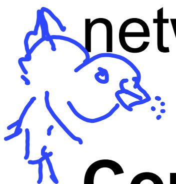
Environment Setup: Set environment variables that contain private sensitive information.

Dockerfile & Commands

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

wen ha yetsayavo

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



hiye w aam baamil built shu el osas li btaamil run

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

shu el osas li bdna naamelo Run.

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

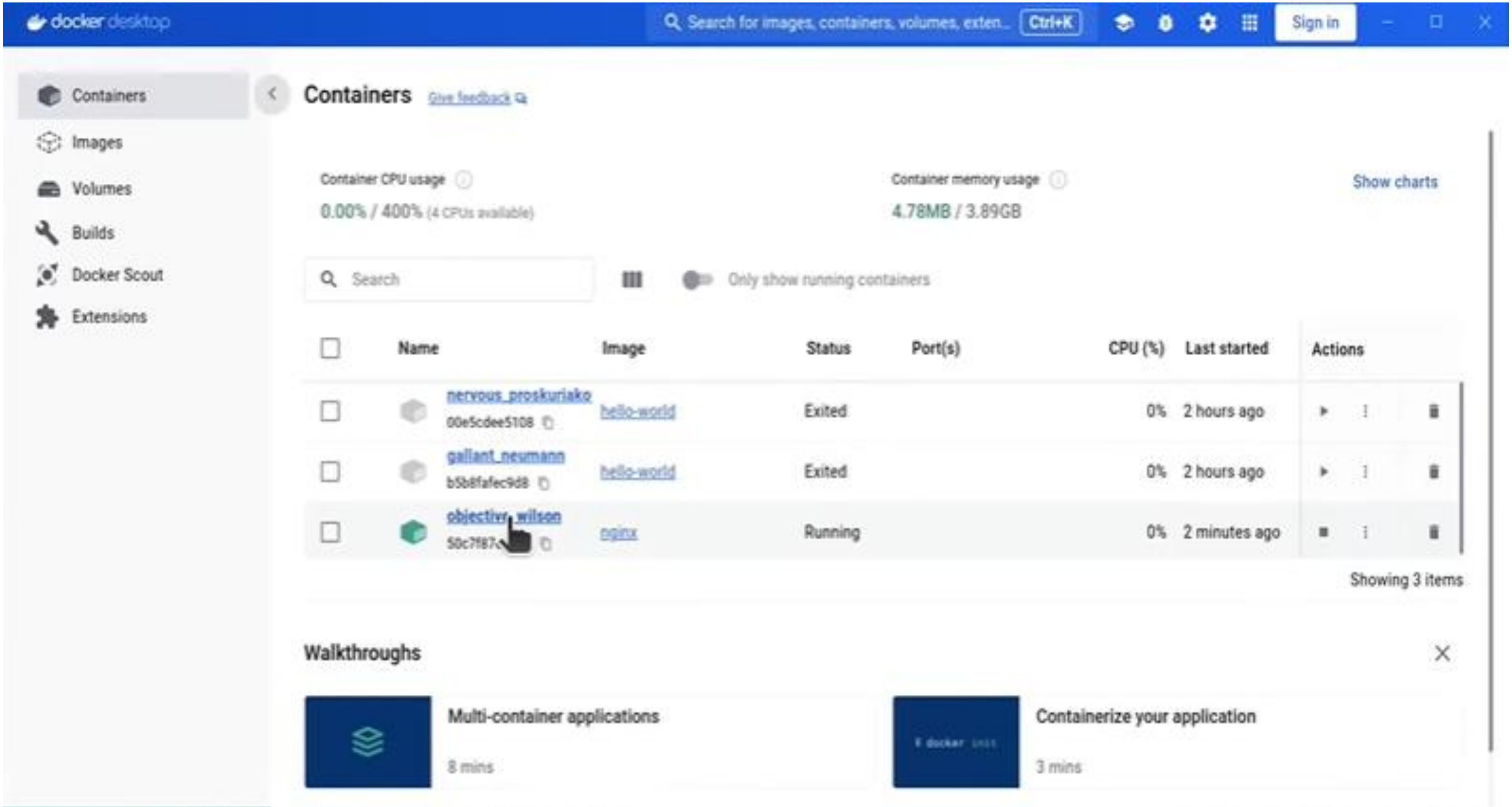
Port Mapping (nginx example)

- Let's run the command: docker run nginx
- It will install the image nginx and run it:

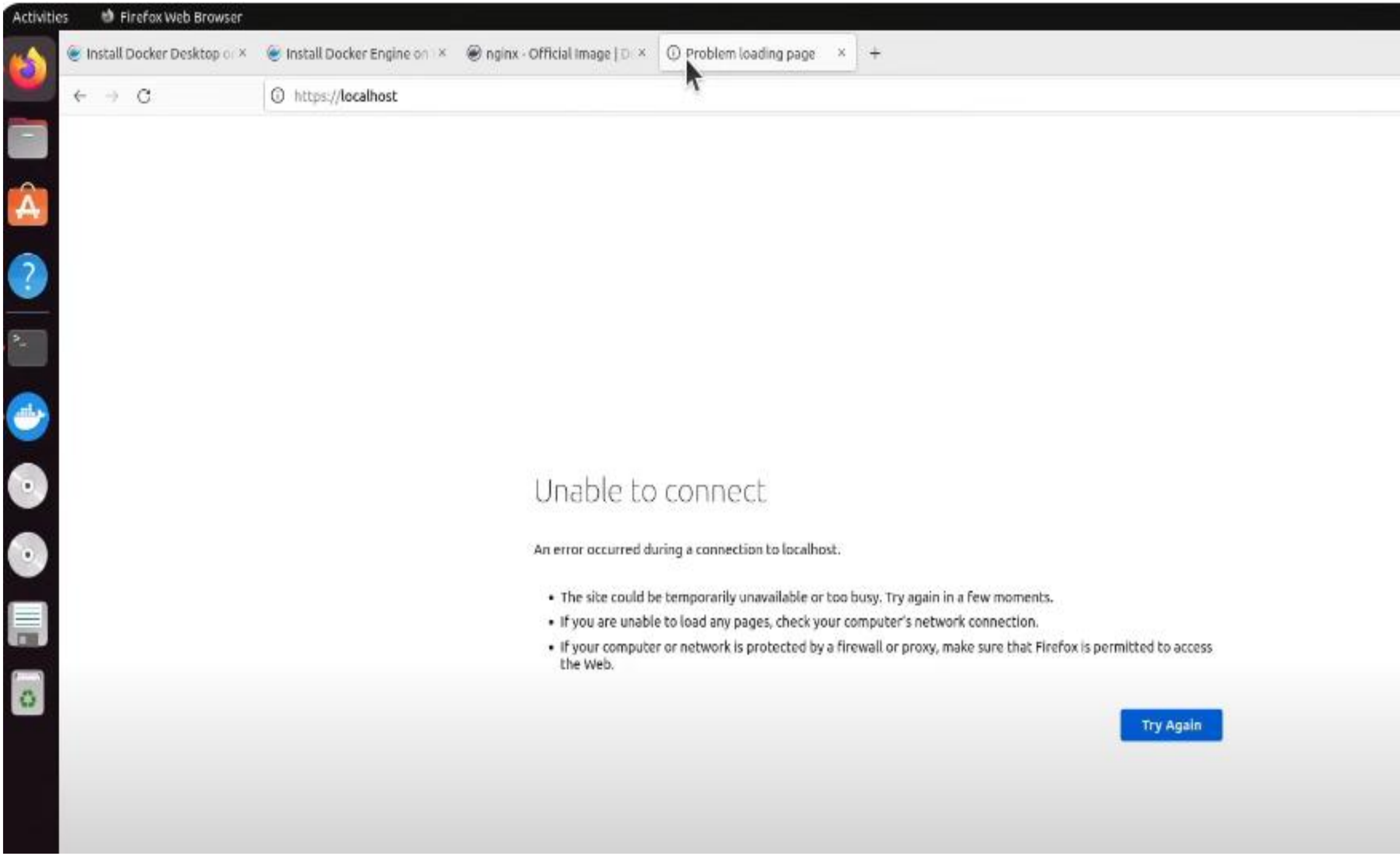
```
james@dockerdemo:~$ docker run nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
efc2b5ad9eec: Pull complete
8fe9a55eb80f: Pull complete
045037a63be8: Pull complete
7111b42b4bfa: Pull complete
3dfc528a4df9: Pull complete
9e891cdb453b: Pull complete
0f11e17345c5: Pull complete
Digest: sha256:6af79ae5de407283dcea8b00d5c37ace95441fd58a8b1d2aa1ed93f5511bb18c
Status: Downloaded newer image for nginx:latest
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2024/07/26 00:43:11 [notice] 1#1: using the "epoll" event method
2024/07/26 00:43:11 [notice] 1#1: nginx/1.27.0
2024/07/26 00:43:11 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2024/07/26 00:43:11 [notice] 1#1: OS: Linux 6.6.32-linuxkit
2024/07/26 00:43:11 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2024/07/26 00:43:11 [notice] 1#1: start worker processes
2024/07/26 00:43:11 [notice] 1#1: start worker process 29
2024/07/26 00:43:11 [notice] 1#1: start worker process 30
2024/07/26 00:43:11 [notice] 1#1: start worker process 31
2024/07/26 00:43:11 [notice] 1#1: start worker process 32
```

In Docker Desktop

- We see the running container (nginx) in the GUI:

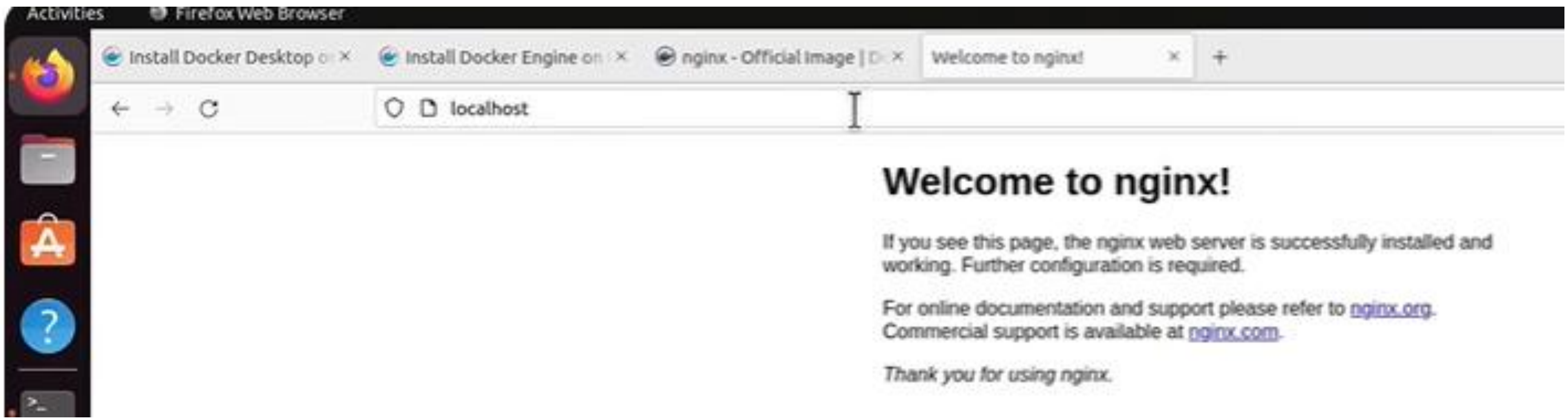


http://localhost



Solution : port mapping to port 80

- Use the command: Docker run –p 80:80 nginx
- Then :



Example

To build an image:
docker build -t <project-name>-image.
👆🏽 Replace <project-name> with what you want your project image to be.

To run:
docker run -p <port> --rm --name <image-name>- container <image-name>
👆🏽 Specify <port> corresponding what is in docker file
& <image-name> relating to last step

built → image
run → container

Example of Docker commands

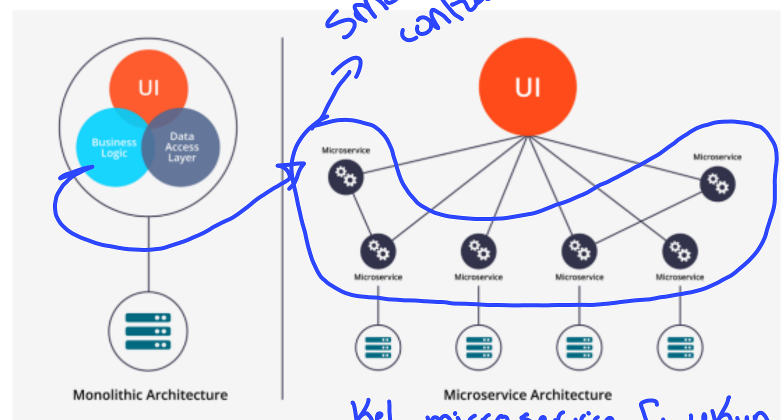
- https://docker-curriculum.com/ (until Docker on AWS)

Microservices

Microservices

- Breaks large applications down into smaller executable components
- Easy to maintain and test
- Loosely coupled and can be deployed independently
- Can be combined with serverless architecture (AWS Fargate)

smaller executable containers



Kel microservice fi ykun 3nda container.

Microservices used in docker

- Develop applications that work on any OS
- Easy to share applications among teams
- Easy to scale across multiple servers
- Large applications can be broken into multiple containers - one for each microservice
- Great solution for Cloud Computing
- Big community and library of Docker Images

→ python, C++, mobile/web app/- - - .

Serverless

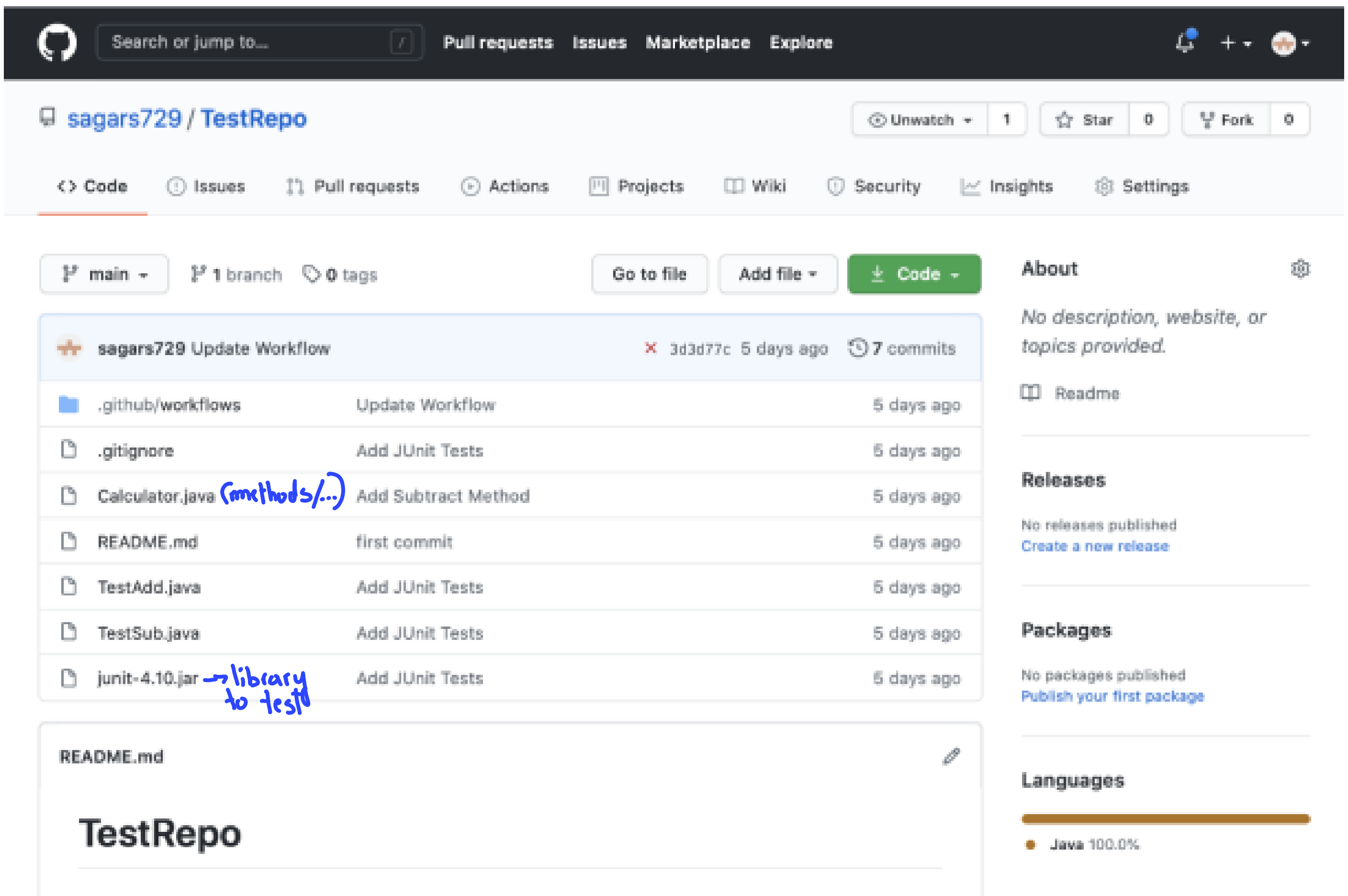
more dependent on infrastructure

–

- Removes Dependency on Infrastructure
- Allows developers to focus on application development
- Microservices can be decoupled with different cloud services
- Usually more cost effective

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: PaaS



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
```

base image

```
RUN useradd -ms /bin/bash ojdk
```

zedet user ojdk

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

kel a3om bi3mal build la image

read

permission

```
WORKDIR /home/ojdk/app
```

working directory

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

hatayneh 3al image/working directory (add)

```
USER ojdk
```

hal user li hwe bdo ysh8l hal

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

compile bas on this library kil el java

```
COPY --chown=ojdk:ojdk . .
```

kil shi in ojdk 3melo copy

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

when running container 3mela

3mellon run in container

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

Build a docker image

Build A Docker Image

To build a docker image using a Dockerfile we can use the **docker image build** command and provide it the directory where the Dockerfile exists. The **--tag** option allows us to name and tag the docker image.

```
[TestRepo $ docker image build . --tag "calculator:latest"
 Sending build context to Docker daemon  590.3kB
 Step 1/9 : FROM openjdk
  ---> e105e26a0a75
Step 9/10 : COPY --chown=ojdk:ojdk . .
 ---> 47cff2b55e3c
Step 10/10 : CMD [ "java", "-cp", "junit-4.10.jar:.", "org.junit.runner.JUnitCore", "TestAdd", "TestSub"]
 ---> Running in c8395bc770b4
Removing intermediate container c8395bc770b4
 ---> 6b345c94e511
Successfully built 6b345c94e511
Successfully tagged calculator:latest
```

mnowliya tag ek image banata aur docker daemon.

Run our docker image

We can use the **docker run** command to run our image and we can see that our tests are being run:

```
TestRepo $ docker run calculator
JUnit version 4.10
..
Time: 0.006

OK (2 tests)

TestRepo $
```

Run interactively

We can use **docker run -it** to run our image interactively and open a bash shell in our working directory:

```
[TestRepo $ docker run -it calculator bash
[[ojdk@419a727a1ca1 app]$ ls
Calculator.class   README.md        TestSub.class
Calculator.java    TestAdd.class    TestSub.java
Dockerfile         TestAdd.java     junit-4.10.jar
[ojdk@419a727a1ca1 app]$
```

stuff in working directory

Docker compose file

Docker Compose File

aktir men container w naamelon run kelon at the same time.

Docker Compose files can be used to run multiple services at once and is great once you have many microservices as part of your application. For our project we create a single service calculator that

- is built using a custom Dockerfile
- tagged with the name calculator
- restarts unless it is stopped

```
version: "3"

services:
  calculator:
    build:
      context: .
      dockerfile: Dockerfile
    image: calculator
    container_name: calculator
    restart: unless-stopped
```

docker-compose.yml

Run Docker compose

To run our docker compose file, we use the **docker-compose up** command. This builds all images and and runs and containers.

```
TestRepo $ docker-compose up
Creating network "testrepo_default" with the default driver
Creating calculator ... done
Attaching to calculator
calculator     | JUnit version 4.10
calculator     | ..
calculator     | Time: 0.005
calculator     |
calculator     | OK (2 tests)
calculator     |
TestRepo $ 
```

Stop Containers

The calculator container will keep restarting unless its stopped. To stop all services, we can use the **`docker-compose down`** command:

```
TestRepo $ docker-compose down
Stopping calculator ... done
Removing calculator ... done
Removing network testrepo_default
TestRepo $
```

Clean up

To remove all unused docker resources, we can use the **docker system prune** command with the **--all** flag:

```
TestRepo $ docker system prune --all
WARNING! This will remove:
  - all stopped containers
  - all networks not used by at least one container
  - all images without at least one container associated to them
  - all build cache

Are you sure you want to continue? [y/N]
```

bi nadde kishi resources hoje to u mesh aam estaamla.

(mo aam azmlan run hla)

Other services

- Xaas: Everything as a service : from food to medical consultations….
- Aaas: Analytics as a service: from data to insights : ex. Outlier.
- Daas: Desktop as a service ex. Citrix
- Faas: Functions as a service
- Staas: Storage as a service
- ….