Agenda

1. Introduction to Containers
2. How Docker Works (Internet Environment)
3. How Docker Works (Airgapped Environment)
4. Base Image selection criteria
5. Demo

Introduction to Containers

Docker is a containerization platform that simplifies application deployment and management

It works by packaging your application and all its dependencies (libraries, configuration files) into a standardized unit called a container

This container image includes everything needed to run the application regardless of the underlying environment

Advantages of Containers

Portability: Containers are self-contained, they can run consistently on any machine with Docker installed. This simplifies deployment across different environments

Isolation: Docker containers use namespaces and control groups to isolate applications from each other and the host system

Efficiency: Containers share the host kernel, making them lightweight and faster to start compared to virtual machines

Containers

Virtual Machines

contain binaries, libraries, and configuration files along with the application itself	has its own copy of an operating system along with the application, necessary binaries, libraries, and configuration files.
do not contain a guest OS for each container and rely on the underlying OS kernel, which makes the containers lightweight.	runs its own guest operating system, which includes its own kernel, this makes it significantly larger and it requires more resources and slower to boot.
namespaces and cgroups offer isolation within the shared kernel, however, a security vulnerability in the host kernel could potentially impact all containers running on that machine.	strong isolation barrier due to virtualizing the entire hardware layer, including the CPU, memory, storage, and network devices. If attacked on the host kernal, it will be harder to access the VM's hardware or OS.

How Docker Works

Daemon: A persistent background process that manages Docker images, containers, networks and storage volumes.
It constantly listens for Docker API requests and executes commands by translating them into actionable operations within the Docker environment.

Dockerfile

The Dockerfile is a text file that uses a DSL (Domain Specific Language) and contains instructions for generating a Docker image. Each instruction defines the steps to create an image layer.

When building an image, the Docker daemon runs all of the instructions in the Dockerfile from top to bottom.

When re-building an image, Docker will attempt to reuse layers from earlier builds, skipping over instructions that it does not need to repeat. If a layer has changed, that layer and all following layers must be rebuilt.

Docker Image & Container

Docker image is an executable package of software that includes everything needed to run an application and is immutable

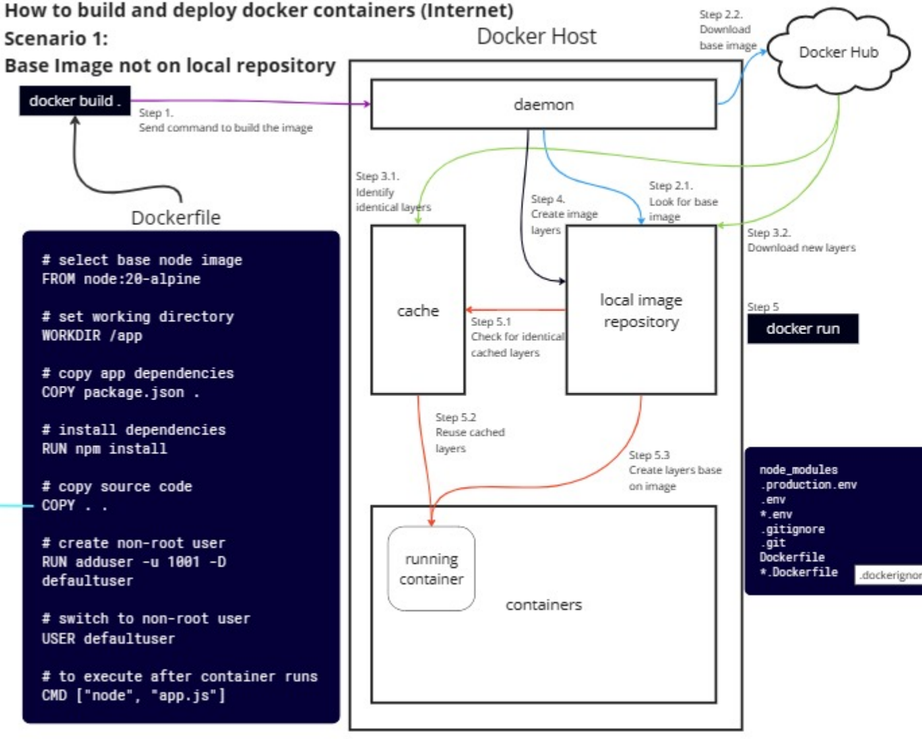
This image informs how a container should instantiate, determining which software components will run and how

A container is a running instance of an image

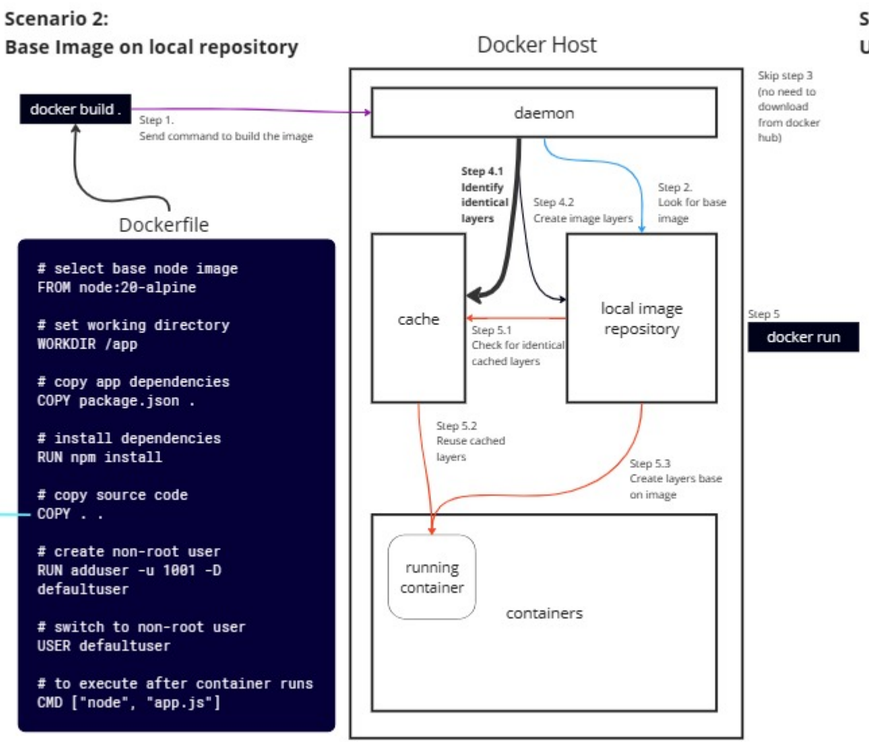
Docker images can be shared and can run as a docker container in any machine where docker is installed without depending on the environment

How to build and deploy docker containers (Internet)

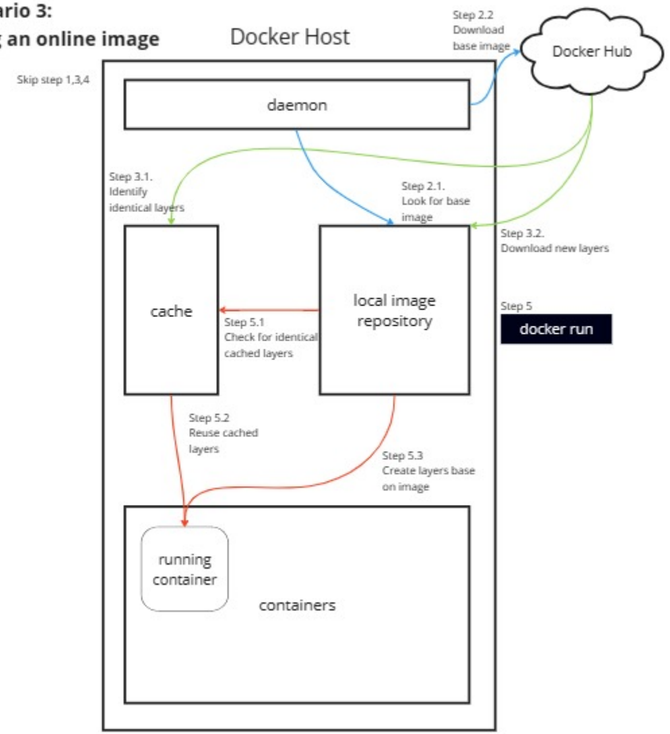
Scenario 1:
Base Image not on local repository



Scenario 2:
Base Image on local repository



Scenario 3:
Using an online image



How to build and deploy docker containers (Airgap)

STEP 1

A. Prepare Dev Environment (with Internet Access)

Download Node 20 Alpine image and export

```
# pull node:20-alpine image
docker pull node:20-alpine
```

```
# save image as a tar archive
docker save --output node20a.tar node:20-alpine
```

1. copy package.json to a new empty folder
2. ensure "bundledDependencies": true in package.json
3. npm install and npm pack

```
npm install
npm pack
# compressed archive (.tgz) of packages created
```

STEP 2

Copy to Dev Env (with security measures)

Security Steps

1. Generate SHA256 hash of files in powershell:
   - Check file integrity (ensure file has not been tampered with)
   - deterministic: same input will always produce same hash
   - high resistance to hash collision

```
Get-FileHash -filepath-

# Get-FileHash .\node20a.tar >> output.txt
```

Example output:

```
Algorithm       Hash        Path
---------       ----        ----
SHA256          3F7E..      C:\..
```

2. Compress both files with any compression utility (e.g. 7-Zip, WinRAR, WinZip), secured with password

3. SHA256 hashes and passwords to be sent to airgapped environment separately

STEP 3

Dev Environment (no internet access)

B. Dev, Build, and Test

Pre-req: Check SHA256 hash


```
docker load --input node20a.tar
```

```
# select base node image
FROM node:20-alpine

# set working directory
WORKDIR /app

# copy app dependencies
COPY /<module>.tgz .

# install dependencies
RUN npm install <module>.tgz

# copy source code
COPY . .

# check for differing packages
RUN diff package.json node_modules/<module_name>/package.json || exit 1

# move modules to their proper folder structure
RUN mv node_modules/<module_name>/node_modules/* node_modules

# remove leftover files/folders
RUN rm -r node_modules/<module_name>
RUN rm <module>.tgz

# create non-root user
RUN adduser -u 1001 -D defaultuser

# switch to non-root user
USER defaultuser

#to execute after container runs
CMD ["node", "app.js"]
```

```
node_modules
.production.env
.env
*.env
.gitignore
.git
Dockerfile
*.Dockerfile
```

.dockerignore

Dockerfile

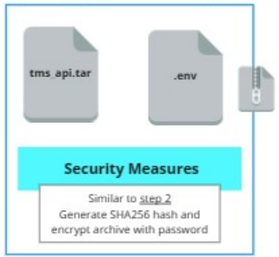
```
# docker build . -t <imagename:tagname>
docker build . -t tms_api:1.0.0
```

```
docker save --output tms_api.tar tms_api:1.0.0
```

tms_api.tar

STEP 4

Copy to prod/pre-prod (with security measures)



STEP 5

Test/Prod Environment

C. Deployment

```
# Load image to docker image repository
docker load --input tms_api.tar

# Create and run container
docker run --name tms_api --rm -p 8080:8080 --env-file .\.env
tms_api:1.0.0
```

Release Engineer will perform the following steps

1. Load image
2. Update .env file (server info, username, password for different environment)
3. Run container using imported image

Base Image Selection Criteria

Factors affecting selection:

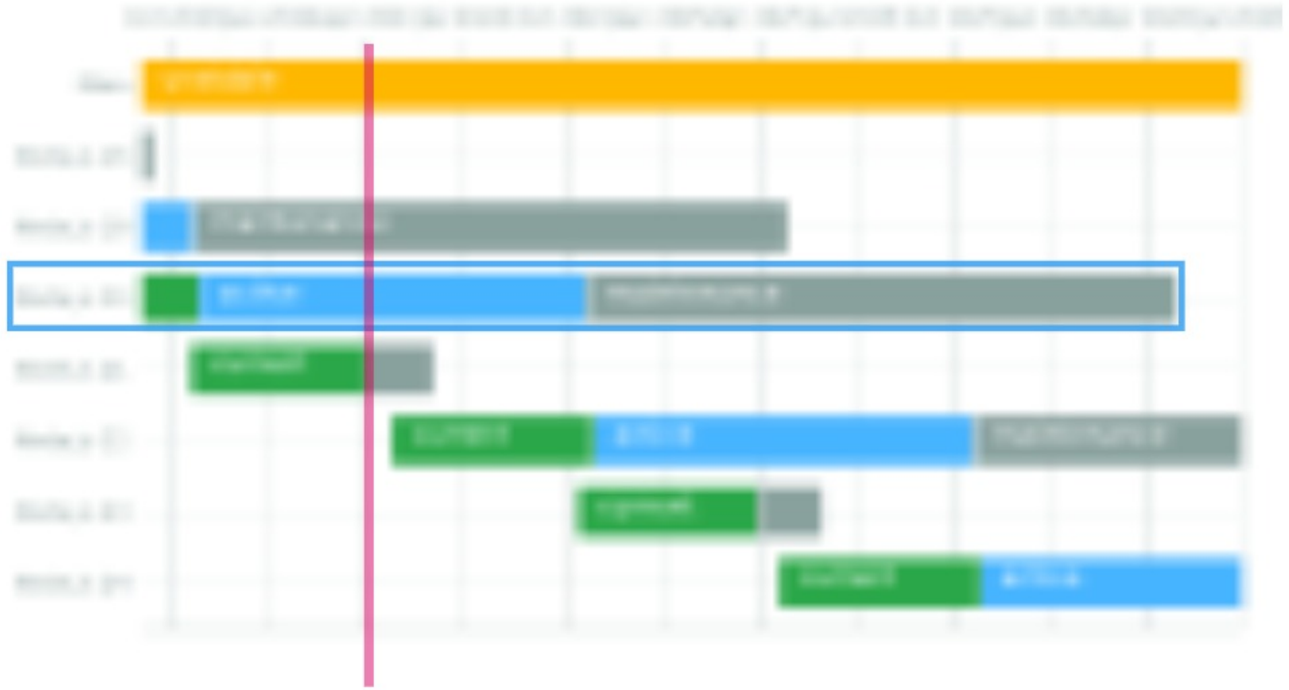
- Application image size (Must be under 200MB)
- Image must have minimal vulnerabilities
- Availability of long term support (Critical bugs will be fixed for 30 months)

Node Version	Base Image Size	Vulnerabilities as of 20 Mar 2024				
		Critical	High	Medium	Low	Unspecified
node:18	1.09 GB	0	1	6	86	0
node:18-slim	196.33 MB	0	0	2	18	0
node:18-alpine	131.67 MB	0	0	1	0	0
node:20	1.09 GB	0	1	6	86	0
node:20-slim	200.74 MB	0	0	2	18	0
node:20-alpine	136.63 MB	0	0	1	0	0
node:21	1.1 GB	0	1	5	86	0
node:21-slim	205 MB	0	0	1	18	0
node:21-alpine	140.82 MB	0	0	0	0	0

What is the Common Vulnerability Scoring System (CVSS)

Severity	Base Score
None	0
Low	0.1-3.9
Medium	4.0-6.9
High	7.0-8.9
Critical	9.0-10.0

After six months, **odd-numbered releases** (9, 11, etc.) **become unsupported**, and **even-numbered releases** (10, 12, etc.) **move to *Active LTS* status** and are ready for general use.



Present Date

```
# test run your built image
# docker run --name <container_name> --rm -p <host port>:<container port> --env-file
<.env filepath> <image name>
docker run --name tms_api_1.0.0 --rm -p 8080:8080 --env-file .\.env tms_api:1.0.0
```

Daemon: A persistent background process that manges Docker images, containers, networks and storage volumes. The daemon constantly listens for Docker API requests and executes commands by translating them into actionable operations within the Docker environment.

Container: Docker container is a self-contained, runnable software application or service

Image: Template/blueprint for containers (contains code + required tools/runtime). Read-only template with instructions for creating a Docker container

daemon: A persistent background process that manages Docker images, containers, networks, and storage volumes

Airgap Environment

Prerequisite:

npm pack

will output a file e.g. digital-academy-api-0.0.1.tgz

```bash
## extract tgz inside (in the container)
## don't need npm on the host
docker load --input node-20_alpine.tar
docker build -t tms-api-airgap .
docker run --name tms-api-airgap -p 3001:3000 tms-api-airgap

## extract tgz outside (on the host)
docker load --input node-20_alpine.tar
tar -xvzf digital-academy-api-0.0.1.tgz
cd package
docker build -t tms-api-airgap . -f airgap.Dockerfile
docker run --name tms-api-airgap -p 3001:3000 tms-api-airgap

## debugging
docker run -it tms-api-airgap sh
docker build prune
docker build --no-cache -t tms-api-airgap .
```

#create a non-root user
#RUN adduser -u 1001 -D defaultuser
#switch to non-root user
#USER defaultuser

cached build

Dockerfile is essentially the build instructions to build the image.

Docker uses a technology called namespaces to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container.

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.