Docker Internals

Container and Resource Isolation

The Problem of Deployment

After development, software engineers need to deploy the software; however, there are possible issues.

- **Diverse libraries & versions** per project
- **Breaking API changes** that fail CI/CD builds
- **Different local setups** across machines
- **Operational complexity** maintaining consistency

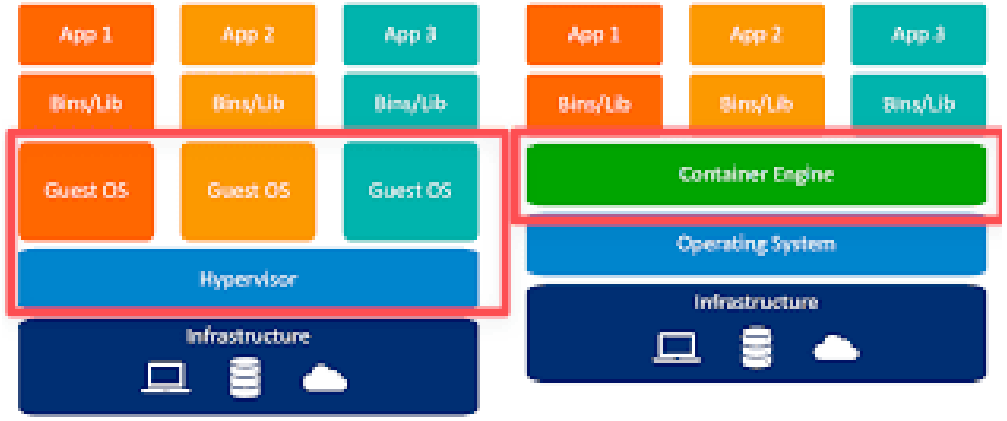
The Expensive Solution - Virtualization

- Use **virtual machines (VMs)** to ensure the same environment for all developers and operators.
- But VMs are often **too expensive**, since they require:
  - More computing power and memory
  - Extra setup and maintenance efforts

A Solution - Container

Instead of using a virtual machine, we can solve the problem easily by isolating environment in the Container.

- No hypervisor virtual machine
- No guest OS
- **Simple Container engine**



What is a Container?

Think of containers like **shipping containers**

- **Standardized**: Same container works on any ship, truck, or train
- **Isolated**: Contents don't mix with other containers
- **Portable**: Move anywhere without unpacking

In software:

- **Standardized**: Same container works on any Linux system
- **Isolated**: Apps don't interfere with each other
- **Portable**: Move between development, testing, and production

VM vs Container - Resource Comparison

Virtual Machine	Container
Full OS (several GB)	Shares host OS kernel (MB)
Minutes to start	Seconds to start
Heavy resource usage	Lightweight
Hardware-level isolation	Process-level isolation

Real Example: Running 10 web servers

- VMs: 10 × 2GB = 20GB RAM needed
- Containers: 10 × 200MB = 2GB RAM needed

Docker: The Container Engine

So we learned containers are better than VMs... but **how do we actually create and manage containers?**

Enter Docker!

Docker is like a **container management system** that:

- Creates containers
- Runs containers
- Manages containers
- Shares containers

Think of it as the **"operating system for containers"**

Container → Docker: The Analogy

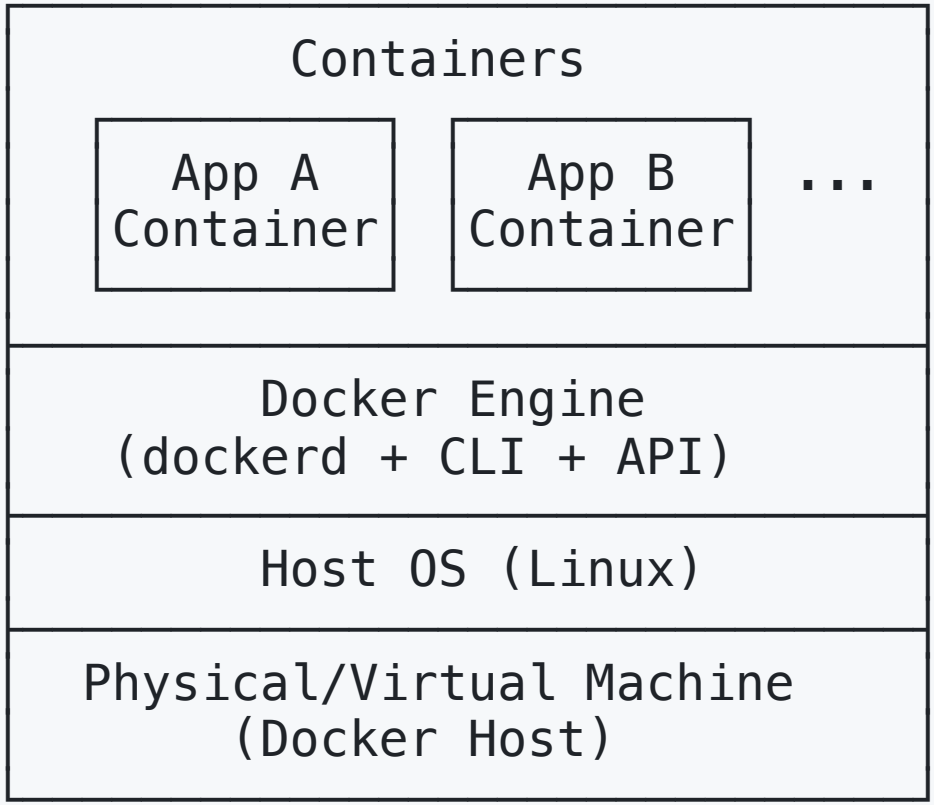
Concept	Real World	Software World
Container	Shipping container	Isolated app environment
Docker	Container ship + crane	Container platform
Docker Hub	Container port	Container registry
Dockerfile	Packing instructions	Build instructions

Without Docker: You have the container concept, but no way to use it!

With Docker: You have the complete system to build, ship, and run containers!

Docker Architecture

The Host runs the Engine, and the Engine runs Containers.

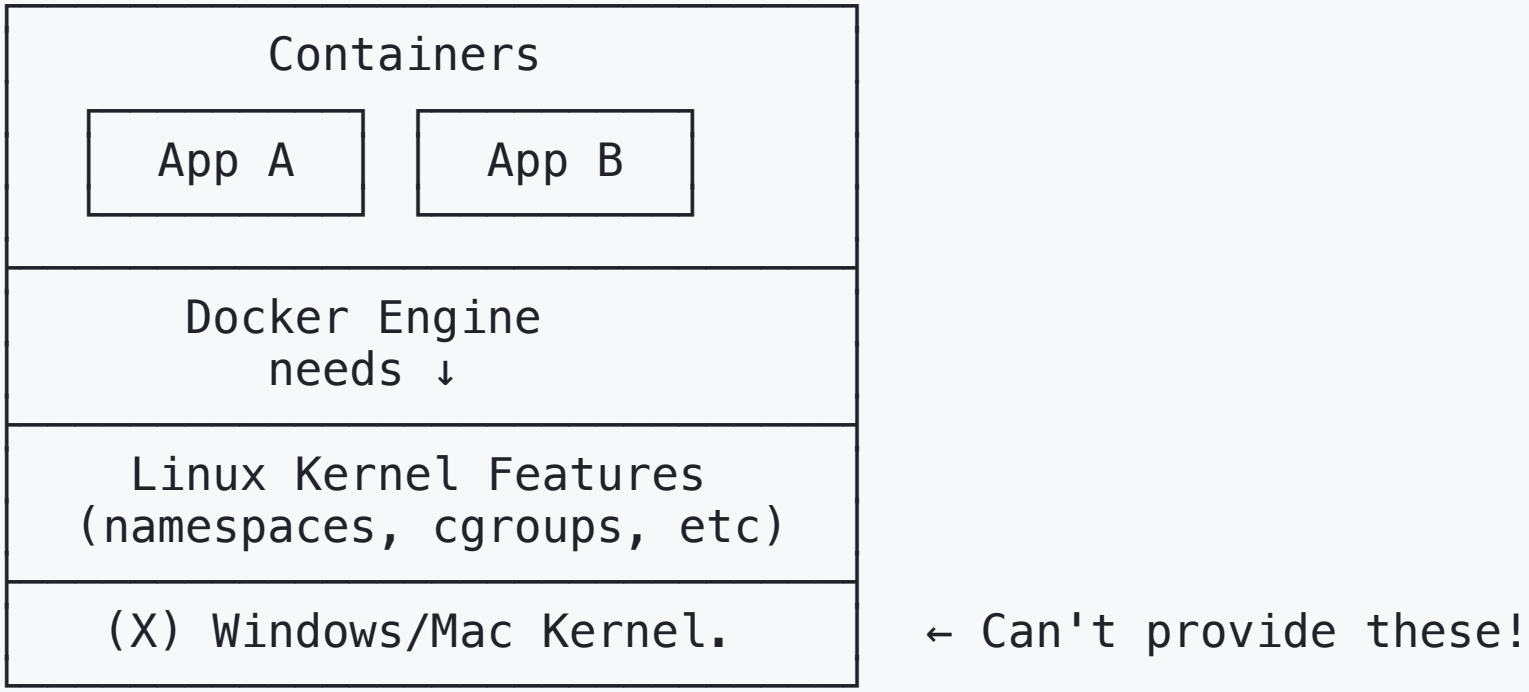


Key Components:

- **Docker Host**: The actual computer/server running everything
- **Host OS**: The Linux operating system installed on the machine
- **Docker Engine**: The software that manages containers
  - `dockerd` : The daemon (background service)
  - `CLI` : Command line interface (`docker` command)
  - `API` : For programmatic control
- **Containers**: Your isolated applications

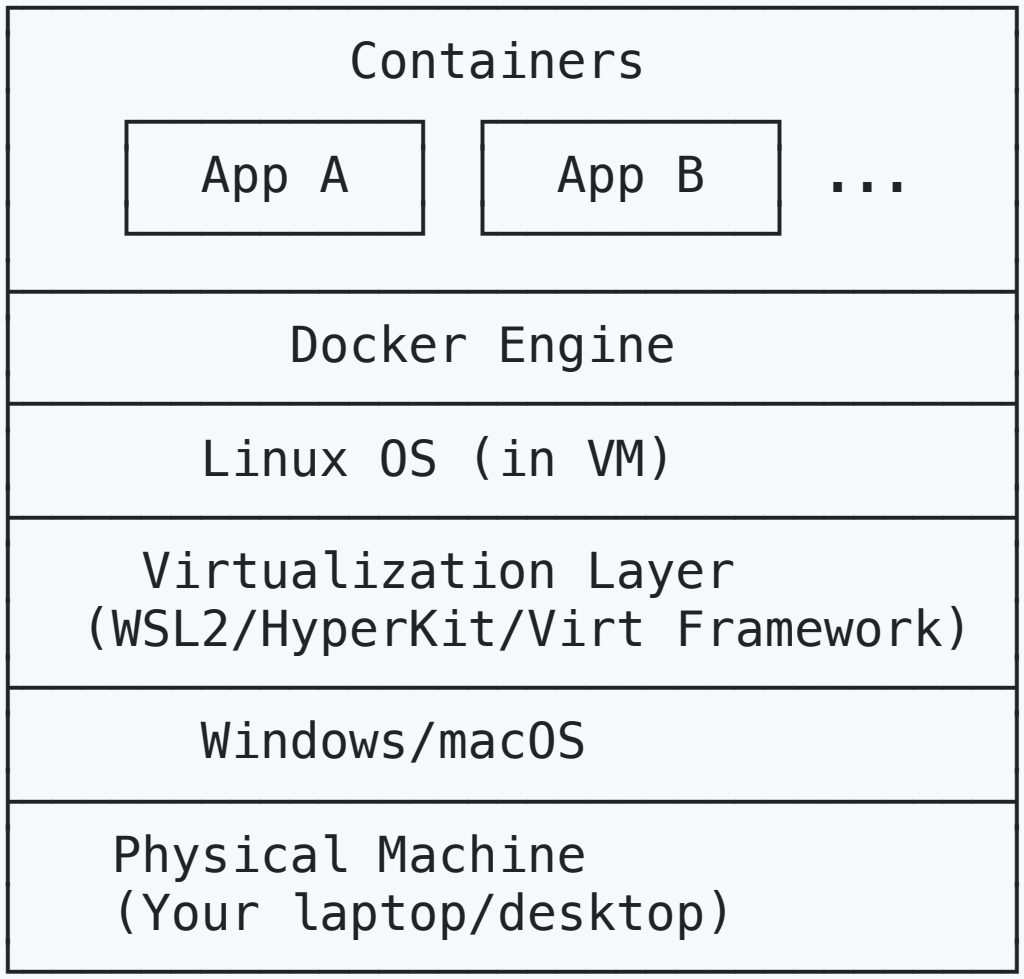
Docker on Windows/Mac - Why Different?

The Problem: Docker Engine needs Linux kernel features!



The Solution: Add a Linux VM layer!

Docker on Windows/Mac



Isolations in Linux OS

Linux isolates environments in three levels: Docker uses these features.

1. Isolate root directory: chroot
2. Isolate process: Process Separation
3. Isolate file: Layered File System

1. Isolate root directory: chroot

- **Linux introduced** `chroot` to isolate environments.
- It changes the root directory (`/`) to create a **sandbox** inside the same OS.
- Any directory can be the Linux root.

2. Isolate process: Process Separation

- **Linux namespaces** isolate processes from each other.
- Each container sees only **its own processes**.
- Inside a container, a process **feels like** it’s the only one running.
- This gives the **illusion of a separate system**.

3. Isolate file: Layered File System

- **Layered (Union) filesystems** stack multiple directories into one view.
- Base layers are **read-only and shared**.
- Changes go to a **writable top layer**.
- This makes systems **space-efficient and fast**.

Layered File System - How It Works

Concept: Stack directories like transparent sheets

```
View from above (what you see):
/myapp/
  ├── config.txt      (from Layer 3 - your changes)
  ├── app.py          (from Layer 2 - application)
  └── libc.so         (from Layer 1 - base system)

Actual storage on disk:
Layer 3 (writable):   /storage/layer3/config.txt
Layer 2 (read-only):  /storage/layer2/app.py
Layer 1 (read-only):  /storage/layer1/libc.so
```

Layered File System - Example

Scenario: You want to run 3 Python applications

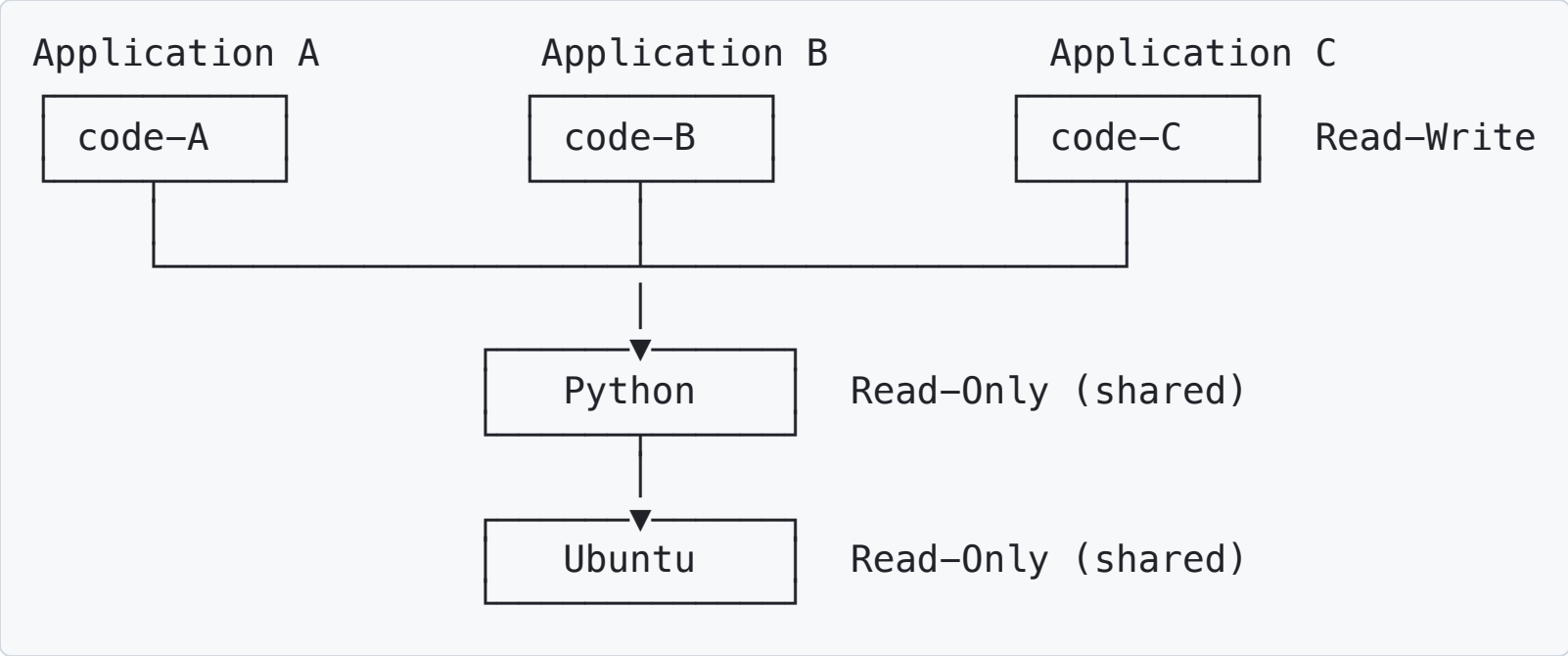
Traditional way (copy everything):

```
App A: Ubuntu (200MB) + Python (100MB) + code (1MB) = 301MB
App B: Ubuntu (200MB) + Python (100MB) + code (1MB) = 301MB
App C: Ubuntu (200MB) + Python (100MB) + code (1MB) = 301MB
Total: 903MB
```

Layered way (share base, add only changes):

```
Base layer (shared):  Ubuntu (200MB) + Python (100MB) = 300MB
Top layer A:          code-A (1MB)
Top layer B:          code-B (1MB)
Top layer C:          code-C (1MB)
Total: 303MB **Saves 600MB!**
```

Layered File System - Visual



Like a sandwich: Same base (bread + lettuce), different toppings!

Git and Docker Layered File System

Git and Docker Layered File System are practically the same.

Git Structure:

```
Commit 3 (your work)
  └─> Tree 3
        ├─> blob: config.txt (new)
        └─> references Tree 2
Commit 2 (add app)
  └─> Tree 2
        ├─> blob: app.py (new)
        └─> references Tree 1
Commit 1 (base)
  └─> Tree 1
        └─> blob: libc.so
```

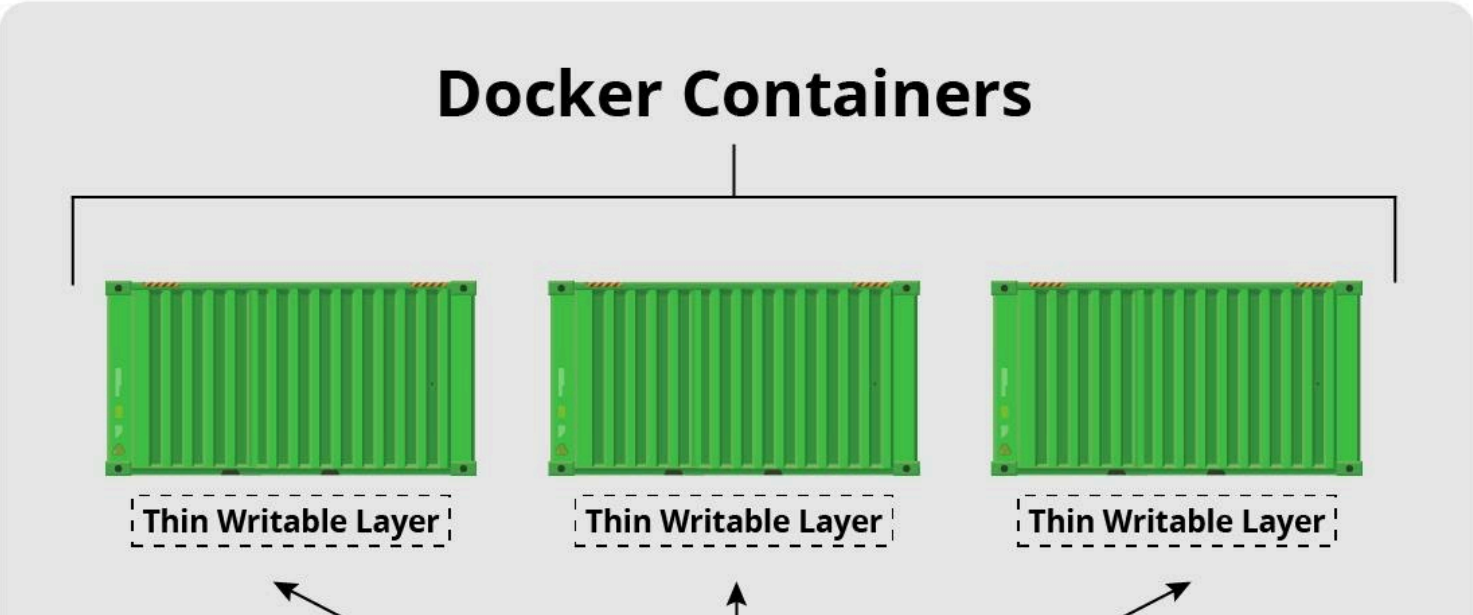
Layered File System:

```
Layer 3 (writable)
  └─> config.txt (new)
  └─> references Layer 2

Layer 2 (read-only)
  └─> app.py (new)
  └─> references Layer 1

Layer 1 (read-only)
  └─> libc.so
```

Remember Linux and Git are made by the same person - Linus Torvalds.

Docker Image

- Docker image – Read-only template with instructions to create containers.
- Images build in layers – Each change adds a new layer.
- Container – A running instance of an image.



Docker Image vs Container

Simple Analogy:

- **Docker Image** = Java Class (blueprint, template)
- **Docker Container** = Java Object (running instance)

Image vs Container - Java Analogy

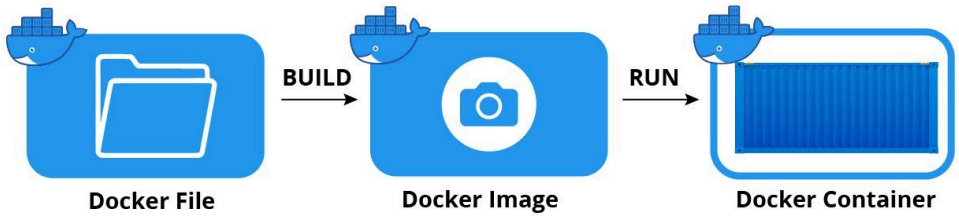
```java
// Docker Image (Class definition)
class WebServer {
    String port = "8080";
    String name;

    void start() {
        System.out.println("Server running on port " + port);
    }
}

// Docker Container (Object instances)
WebServer container1 = new WebServer();  // 1st container
WebServer container2 = new WebServer();  // 2nd container

container1.start();  // Each runs independently
container2.start();
```

Key Point: One class → many objects

Same as: One image → many containers

Building and Running Docker Images

- A Docker image is built from the Dockerfile.
- A Docker image contains all the files to make an isolated environment.
- The built Docker image is run to make an isolated (Linux) environment.



Docker Container & Image Example

```
# Step 1: Write Dockerfile (like writing .java file)
# Dockerfile
FROM nginx:latest
COPY index.html /usr/share/nginx/html/

# Step 2: Build Docker Image (like javac compile)
docker build -t my-web-app:v1.0 .

# Step 3: Create multiple containers (like new Object())
docker run my-web-app:v1.0  # container1 (web server 1)
docker run my-web-app:v1.0  # container2 (web server 2)
docker run my-web-app:v1.0  # container3 (web server 3)
```

Result: 3 separate web servers running from 1 image template!

Just like: `.java` → javac → `.class` → new ClassName()