

REFERENCE



August 15, 2025

The PB

Table of Contents

**Understanding Pre-Docker Era & Revolution**

The Problem: "It works on my machine"2

The Pre-Docker Era: Virtual Machines (VMs)2

The Docker Revolution: Containers …………………………………………………………………………………. 3

The Analogy ………………………………………………………………………………………………………………… 4

**Introduction**

The Docker platform5

**Architecture**

Docker Architecture ……………………………………………………………………………………………. 7

**Image and Container**

Docker Image ………………………….………………………………………………………………………………… 10

Docker Container ……………………………………………………………………………………………………… 11

Docker Image vs Docker Container ……………………………………………………………………………… 12

**Compose**

Docker Compose ………………………………………………………………………………………………………. 15

**Volumes and Persistence**

**Networking Between Containers**

**Docker in DevOps**

**Interview Tips**

**Understanding Pre-Docker Era & Revolution**

* **The Problem: "It works on my machine"**

An application might run perfectly on a developer's computer but fail spectacularly in production. This usually happens because of differences in the operating system, libraries, dependencies, or environment variables. **Virtual machines (VMs)** were the first attempt to solve this.

* **The Pre-Docker Era: Virtual Machines (VMs)**

A VM runs a full-fledged operating system (OS) on top of the host machine's OS. The VM virtualizes the entire hardware, including the CPU, memory, disk, and network interfaces. This creates a completely isolated environment. You could package your application and its dependencies into a VM and be sure it would run the same way everywhere.

Think of a VM as a complete computer running within your computer. It virtualizes the entire hardware stack.

* **Hypervisor**: A piece of software, like VMware or VirtualBox, that manages the VMs and allocates resources (CPU, RAM, disk) to each one.
* **Guest OS:** Each VM runs its own full-fledged operating system (e.g., Ubuntu, Windows Server). This includes the kernel and all system libraries.
* **Hardware Virtualization**: The hypervisor creates a virtual version of the hardware, tricking the guest OS into thinking it's running on its own dedicated machine.

This approach solves the "it works on my machine" problem because you're packaging the entire environment. However, this came at a cost:

* **Resource Intensive:** VMs are heavy. Each VM needs its own full OS, which consumes a significant amount of CPU, RAM, and disk space.
* **Slow to Boot:** Starting a VM can take minutes because it's booting a full operating system.
* **Lack of Portability:** Sharing a large VM image (often several gigabytes) is impractical.
* **The Docker Revolution: Containers**

Docker introduced a different approach called **containerization**. Instead of virtualizing the hardware, a container virtualizes the **operating system** at the process level. A container shares the host OS kernel with other containers. It isolates the application and its dependencies, but it doesn't need to run a separate, full OS.

This is the key difference.

* **Shared Host OS Kernel:** Containers do not have their own guest OS. They all share the kernel of the host machine.
* **Lightweight and Efficient**: Containers are much smaller and use fewer resources than VMs because they don't include an OS. They only contain the application code and its dependencies
* **Application-Layer Virtualization:** The container only packages the application and its dependencies (e.g., libraries, system tools, code). It virtualizes only what's necessary to run the application, not the entire machine.
* **Isolation:** While they share the kernel, containers are completely isolated from each other. Docker uses Linux kernel features called **Namespaces** (for process, network, and file system isolation) and **cgroups (Control Groups)** (for resource management like CPU and memory limits) to achieve this.
* **Fast to Start**: Containers can start in a matter of seconds, as they only need to load the application, not an entire OS.
* **Highly Portable**: A Docker image is a lightweight, self-contained package that can be easily shared and run on any machine with the Docker engine, regardless of the host OS (Linux, Windows, or macOS).

This design makes containers incredibly lightweight and fast. They can start in seconds because there's no OS to boot. A container image is typically in tens or hundreds of megabytes, a fraction of the size of a VM.

* **The Analogy: Apartment vs. House**

A helpful analogy is to think of a VM as a separate house with its own foundation, plumbing, and electrical system. Each house is completely self-sufficient.

The container is an apartment in a building. It shares the building's infrastructure (the kernel) but has its own locked doors, rooms, and a view of the outside world. This makes apartments (containers) much more efficient to build, maintain, and manage than separate houses (VMs). You can have many more apartments on the same plot of land, just as you can run many more containers on a single host.

**Introduction**

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code, you can significantly reduce the delay between writing code and running it in production.

**The Docker platform**

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security let you run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host. You can share containers while you work and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

* Develop your application and its supporting components using containers.
* The container becomes the unit for distributing and testing your application.
* When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

**What can I use Docker for?**

* **Fast, consistent delivery of your applications**

Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

Consider the following example scenario:

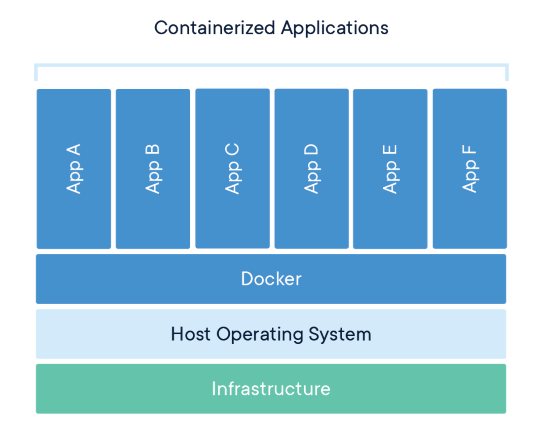
Developers write code locally and share their work with their colleagues using Docker containers. They use Docker to push their applications into a test environment and run automated and manual tests. When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation. When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

* **Responsive deployment and scaling**

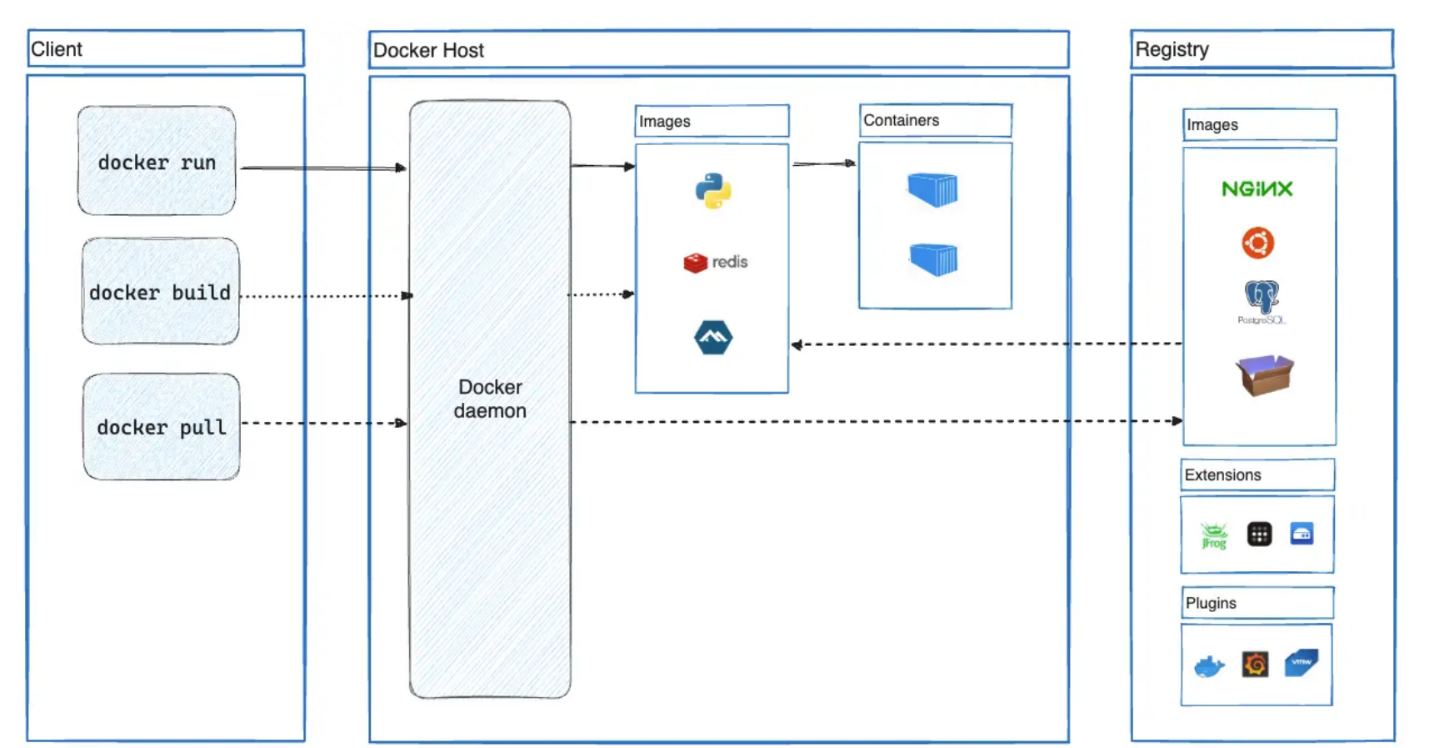
Docker's container-based platform allows for highly portable workloads. Docker containers can run on a developer's local laptop, on physical or virtual machines in a data center, on cloud providers, or in a mixture of environments. Docker's portability and lightweight nature also make it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs

* **Running more workloads on the same hardware**

Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines, so you can use more of your server capacity to achieve your business goals. Docker is perfect for high density environments and for small and medium deployments where you need to do more with fewer resources.

****

**Docker Architecture**

****Docker uses client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. Docker client and Daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. Docker client and daemon communicate using REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

**The Docker Daemon**

Docker Daemon (dockerd) listens to Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

**The Docker client**

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

**Docker Desktop**

Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker Daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper. For more information, see [Docker Desktop](https://docs.docker.com/desktop/).

**Docker registries**

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker looks for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, Docker pulls the required images from your configured registry. When you use the docker push command, Docker pushes your image to your configured registry.

**Docker objects**

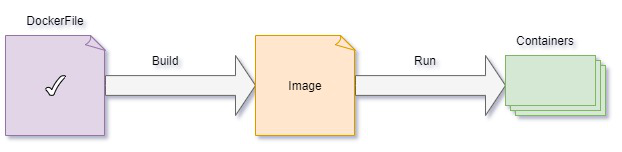
When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

**Docker Image**

Docker Image is an executable package of software that includes everything needed to run an application. This image informs how a container should be instantiated, determining which software components will run and how.

**Docker Container**

Docker Container is a virtual environment that bundles application code with all the dependencies required to run the application. The application runs quickly and reliably from one computing environment to another.



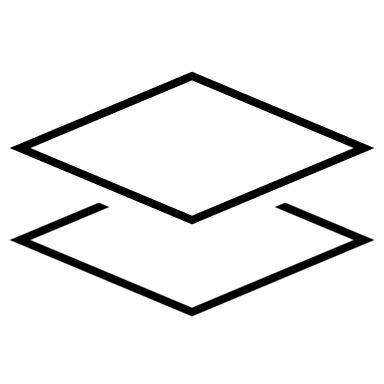
What's happening under the hood?

Docker leverages two core features of the Linux kernel:

* Namespaces: Provide a layer of isolation. They ensure a container has its own isolated view of the system, including process IDs, networking, users, and filesystems. A process inside one container can't see or interact with a process in another container.
* Control Groups (cgroups): Limit the resources that a container can use, such as CPU, memory, and I/O. This prevents one container from consuming all the host's resources and starving other containers or the host system itself.

This fundamental difference between a VM and a container - virtualizing hardware vs. virtualizing the OS - is the most critical concept to grasp when learning Docker. It explains why Docker is so much faster, more efficient, and more suitable for modern, agile development.

**Docker Image and Container**



**What Is a Docker Image?**

A Docker Image is a read-only blueprint for creating docker containers. Think of it like a class in Java - it defines everything needed to run an application, but it’s not running yet.

Often, an image is based on another image, with some additional customization.

Key Characteristics:



* Contains application code, runtime, libraries, environment variables, and configuration files.
* Built using a Dockerfile, which is a script of instructions.
* Immutable: once built, it doesn’t change.
* It can be stored in Docker Hub or private registries.

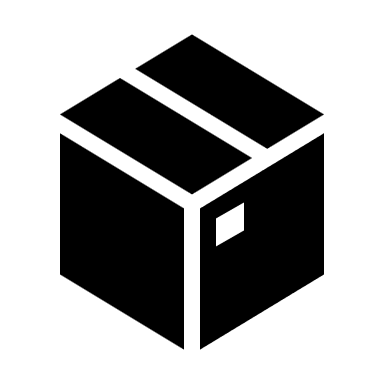
📦 Analogy:

* Dockerfile = Recipe
* Docker Image = Prepared dish based on the recipe
* Docker Container = The dish served on a plate, ready to eat

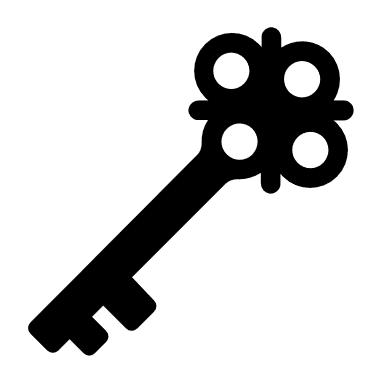
Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but install the Apache web server and your application, as well as the configuration details needed to make your application run. You might create your own images, or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile file with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Reference Links:

[**Dockerfile ＞Docker Image ＞ Docker Container | Beginners Hands-On | Step by Step - YouTube**](https://www.youtube.com/watch?v=C-bX86AgyiA)

**What Is a Docker Container?**

A Docker Container is a running instance of a Docker Image. It’s like creating an object from a class in Java - it’s live, isolated, and executable.

 Key Characteristics:

* It has a writable layer on top of the image.
* Runs in an isolated environment (like lightweight VM).
* Can be started, stopped, moved, or deleted.
* It communicates with other containers via networks

📦 Analogy: If the image is the blueprint, the container is the actual house built from it.

We can create, start, stop, move, or delete a container using Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state. By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine. A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that aren't stored in persistent storage disappear.

🔄 How They Work Together?

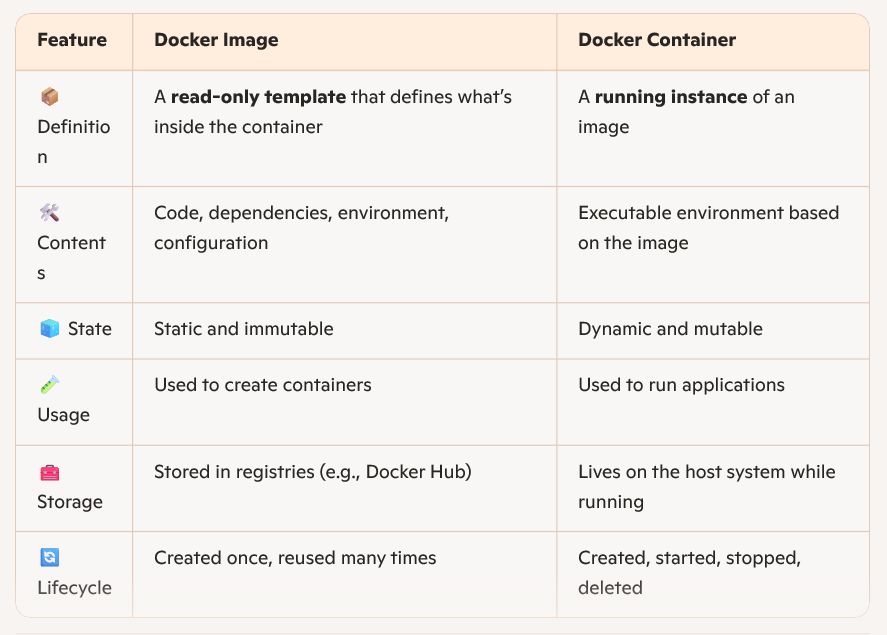
1. Write a Dockerfile: Define the environment and app setup.
2. Build an Image: docker build -t myapp
3. Run a Container: docker run -d -p 8080:8080 myapp

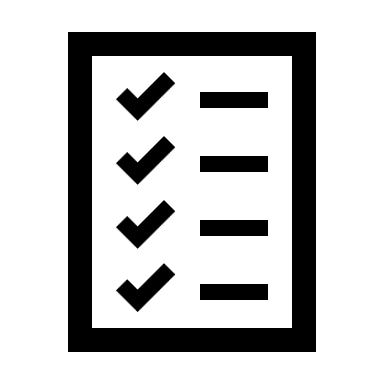
Each container is isolated but can be networked together - ideal for microservices.

**Reference Links:**

[**What is a Container? | What is an Image? | Docker Containers and Images | Geekific - YouTube**](https://www.youtube.com/watch?v=X2hpxp3Kq6A)

**~ Docker Image vs Docker Container**

****

**Step-by-Step: Spring Boot REST API + Docker**

**1. Sample Spring Boot REST API**

**Create a simple controller:**

|  |
| --- |
| **Java Project** |
| **@RestController**  **@RequestMapping("/api")**  **public class HelloController {**  **@GetMapping("/hello")**  **public String sayHello() {**  **return "Hello from Dockerized Spring Boot!";**  **}**  **}** |

**2. Build the project:**

mvn clean package // for maven build

This generates a JAR file in target/, e.g., demo-0.0.1-SNAPSHOT.jar

**3. Create Dockerfile**

Here’s a sample Dockerfile that includes Java options for memory, garbage collection, time zone, and Spring profile

|  |
| --- |
| Dockerfile |
| # Use a lightweight OpenJDK base image  FROM openjdk:17-jdk-alpine  # Set environment variables for Java options  ENV JAVA\_OPTS="-Xms256m -Xmx512m -XX:+UseG1GC -Duser.timezone=Asia/Kolkata -Dspring.profiles.active=prod"  # Set working directory  WORKDIR /app  # Copy the JAR file into the container  COPY target/demo-0.0.1-SNAPSHOT.jar app.jar  # Run the application with custom Java arguments  ENTRYPOINT ["sh", "-c", "java $JAVA\_OPTS -jar app.jar"] |

~ FROM: Specifies the base image. openjdk:17-jdk-alpine is a small, fast image with Java 17 and Alpine Linux

~ ENV: Defines environment variables. JAVA\_OPTS sets JVM memory and runtime options. JAVA\_OPTS includes:

* -Xms256m: Initial heap size
* -Xmx512m: Max heap size
* -XX:+UseG1GC: Use G1 Garbage Collector
* -Duser.timezone=Asia/Kolkata: Set JVM timezone
* -Dspring.profiles.active=prod: Activate Spring profile

~ WORKDIR: Sets /app as the working directory for subsequent commands inside container

~ COPY: Copies the built JAR file from your machine into the container

~ ENTRYPOINT: Defines the command to run when the container starts. Runs the app with custom Java arguments

~ Uses sh -c to evaluate $JAVA\_OPTS dynamically.

**Build the Docker Image**

|  |
| --- |
| Linux Bash |
| docker build -t springboot-docker-app |

**Run the Docker Container**

|  |
| --- |
| Linux Bash |
| docker run -p 8080:8080 springboot-docker-app |

Visit [http://localhost:8080/api/hello](https://localhost:8080/api/hello) and we can see below output:

Hello from Dockerized Spring Boot!

**✅ Optional: We can Override Java Arguments or change configurations at Docker container runtime. Please check the below examples.**

|  |
| --- |
| **Linux Bash** |
| # Add java arguments  docker run -p 8080:8080 -e JAVA\_OPTS="-Xmx1024m -Dserver.port=9090" springboot-docker-app  # Spring profile  docker run -e SPRING\_PROFILES\_ACTIVE=dev -e JAVA\_OPTS="-Xmx512m" springboot-app  # Useful for performance tuning and preventing resource hogging. Control CPU and memory usage  docker run --memory="512m" --cpus="1.0" springboot-app  # Run at different ports  docker run springboot-app java -jar app.jar --server.port=9090 |

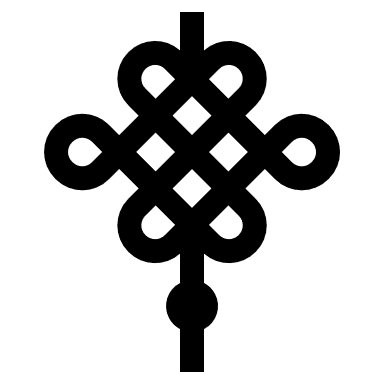
❌ **What You Cannot Change Easily**

• Image contents (e.g., baked-in config files)

• Dockerfile instructions like COPY, RUN, or EXPOSE

• Internal container networking setup

**If we need to change these, we must rebuild the image.**

**Pro Tips**

* Use Spring Boot’s externalized configuration (application.properties, application.yml) with volume mounts or environment variables.
* Use Docker Compose for complex setups with multiple containers and shared configs.
* Monitor resource usage with docker stats

**Reference:**

[**How to Modify the Configuration of Running Docker Containers**](https://www.howtogeek.com/devops/how-to-modify-the-configuration-of-running-docker-containers/)

**Docker Compos**

**Volumes and Persistence**

**Networking Between Containers**

**Docker in DevOps**

**Interview Tips**

**🧠 Interview Tips**

* Explain the analogy: Class vs Object → Image vs Container.
* Mention Dockerfile: It’s the source of truth for building images.
* Highlight isolation: Containers don’t interfere with each other or the host.
* Discuss layering: Images are built in layers, which improves caching and efficiency.

While I haven’t had direct hands-on experience with Docker in a production environment yet, I’ve taken the initiative to understand its core concepts—like containerization, image creation, and orchestration with Docker Compose. I’ve studied how Docker integrates with Spring Boot applications and CI/CD pipelines, and I’ve even started experimenting with Dockerfiles and containerizing simple REST APIs locally. Would you like me to walk you through how I’d Dockerize a Spring Boot app and run it with a database using Docker Compose?