DevOps

Containerization

Chandara Chea

DD/MM/YYYY

Week IV

Contents

What is “Bare Metal”?

What is Virtualization?

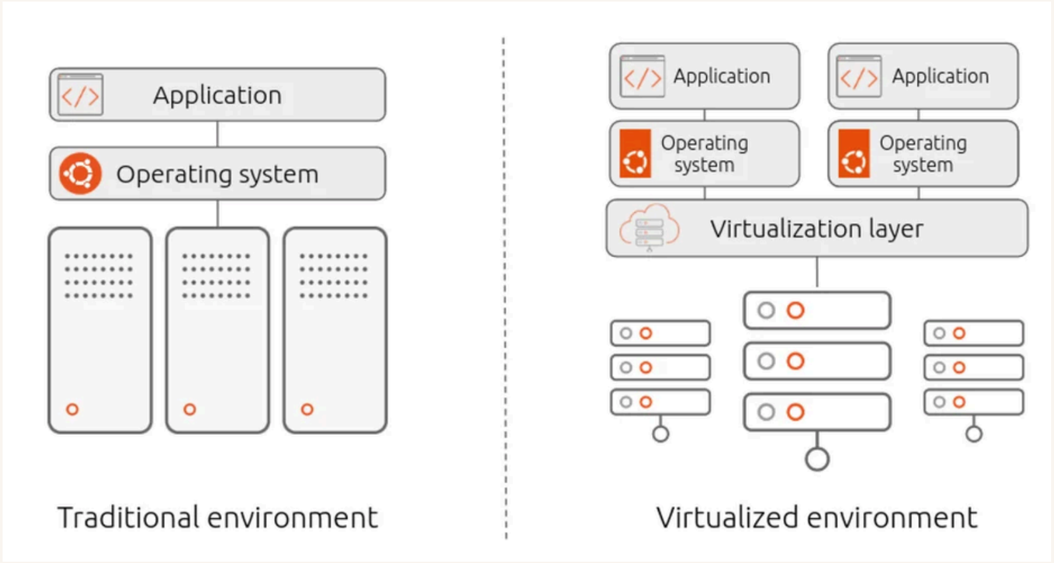
What is Containerization?

How does Containerization Work?

Container Orchestration

What is "Bare Metal"?

- Physical servers, much like a regular computer.
- The Operating System (OS) and applications are installed directly onto the physical machine.
- There is nothing in between the physical ("metal") hardware and the OS.
- These servers were typically dedicated to one specific purpose, such as running a single designated system

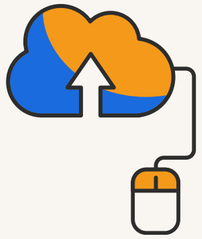


The Bare Metal Trade-Off



The Good: Simplicity

Management was simple, and issues were easier to treat because admins could focus their attention on one specific server. This also offered direct access to hardware, providing high performance.

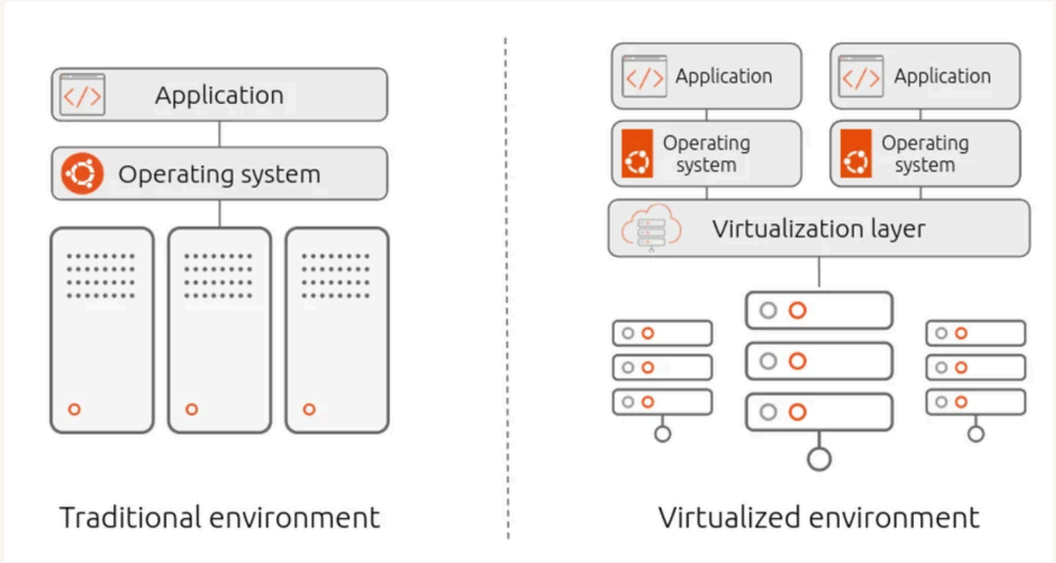


The Bad: High Cost

As a business grew, it needed more and more servers. This model was expensive, not only in hardware costs but also in the physical space and power required to host them. Resource utilization was often very low.

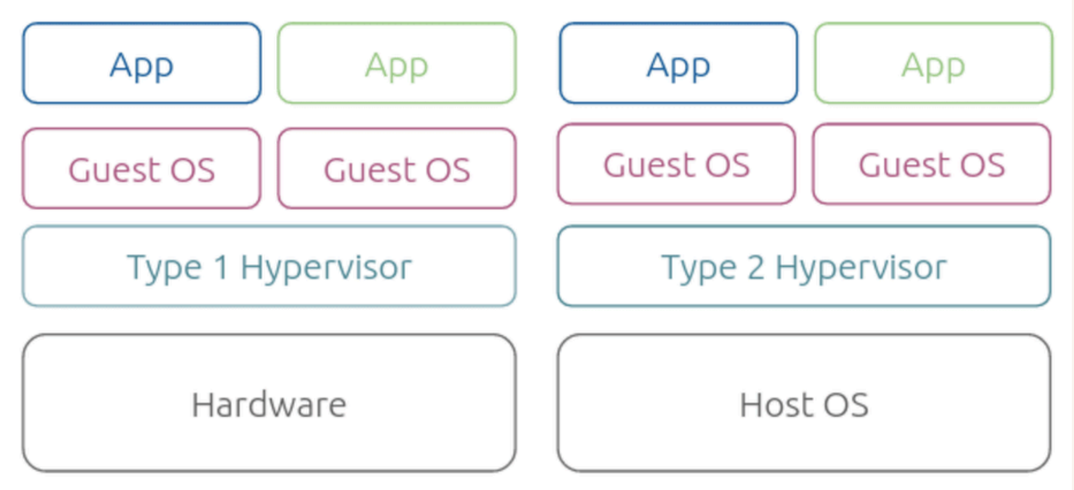
What is Virtualization?

- While the tech dates to the 1960s, it took off in the early 2000s.
- An additional virtualization layer (a "**Hypervisor**") is added between the physical hardware and the operating systems.
- This enables multiple, isolated virtual servers (VMs) to run on one physical machine.
- Each VM has its own independent operating system.
- Virtualization is foundational to cloud computing.



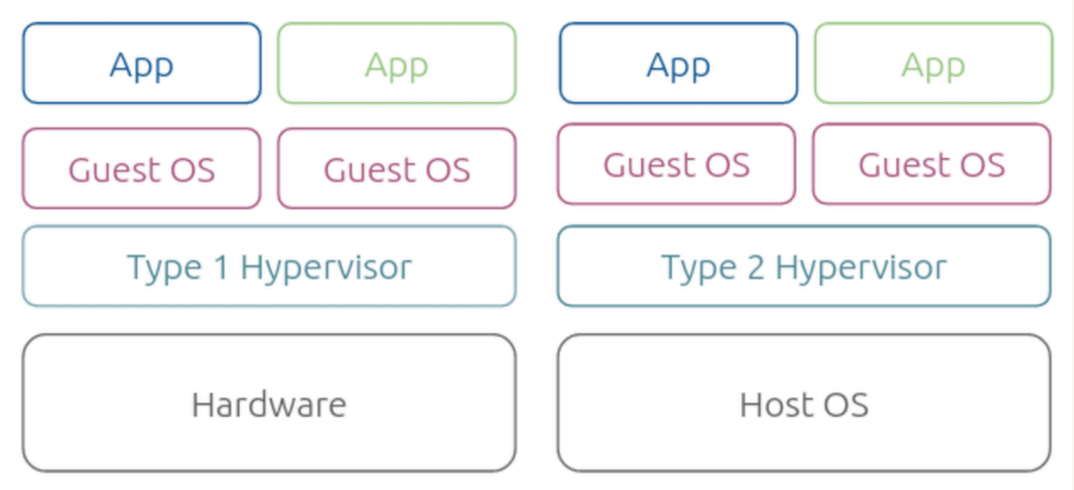
The Role of a Hypervisor

- Virtualization wouldn’t be possible without a **hypervisor** (also known as a virtual machine monitor).
- Hypervisor is a software layer enabling multiple operating systems to co-exist while sharing the resources of a single hardware host.
- The hypervisor acts as an intermediary between virtual machines and the underlying hardware, allocating host resources such as memory, CPU, and storage.



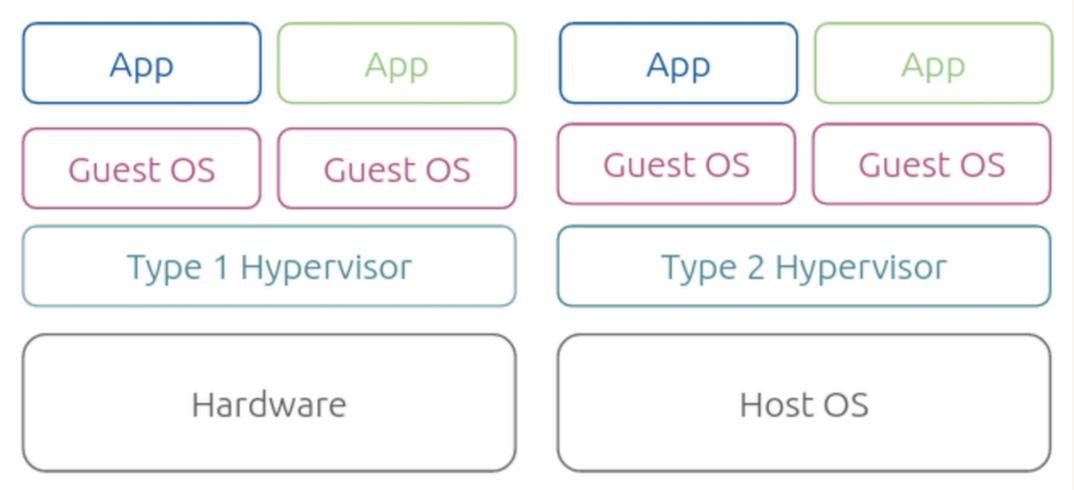
2 Type of Hypervisor

- **Type 1 hypervisors,** also known as **bare-metal hypervisors**, run directly on the host's hardware and are responsible for managing the hardware resources and running the virtual machines.
- They run directly on the hardware so they are often more efficient and have a smaller overhead than Type 2 hypervisors.
- Examples of Type 1 hypervisors include VMware ESXi, Microsoft Hyper-V, and Citrix XenServer.



2 Type of Hypervisor

- **Type 2 hypervisors**, also known as **hosted hypervisors**, run on top of a host operating system and rely on it to provide the necessary hardware resources and support.
- They run on top of an operating system, so they are often easier to install and use than Type 1 hypervisors, but they might be less efficient.
- Examples of Type 2 hypervisors include VMware Workstation and Oracle VirtualBox.



Benefits of Virtualization

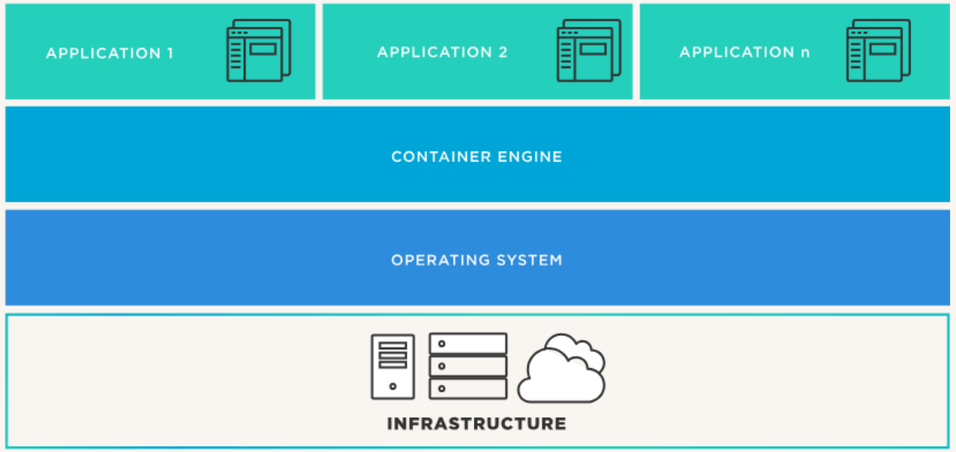
- **Cost Savings:** This is the biggest benefit. By running multiple virtual machines on a single physical server, you save money, space, and are able to do "more with less."
- **Better Resource Utilization:** Prevents your physical servers from standing idle. You can run multiple VMs on a single server, sharing the hardware resources efficiently.
- **Dynamic Management:** You can easily create, destroy, and migrate Virtual Machines between different physical hosts with minimal downtime.
- **Easier Scalability:** Makes it much easier to scale and manage your computing resources as your needs change.
- **Disaster Recovery:** Greatly simplifies the process of implementing and managing disaster recovery and high-availability plans.

Drawbacks of Virtualization

- **Performance Overhead:** The extra virtualization layer can decrease performance. This is noticeable unless significant RAM and CPU resources are allocated.
- **Upfront Investment:** While it saves money long-term, the initial investment in powerful hardware, virtualization software, and training can be high.
- **Increased Complexity:** Your team now needs to manage and maintain both the physical servers and all the virtual instances running on top of them.

What is Containerization?

- **Containerization** 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 **containe**r—that runs consistently on any infrastructure.
- More portable and resource-efficient than virtual machines (VMs), containers have become the de facto compute units of modern cloud-native applications.



What is Containerization?

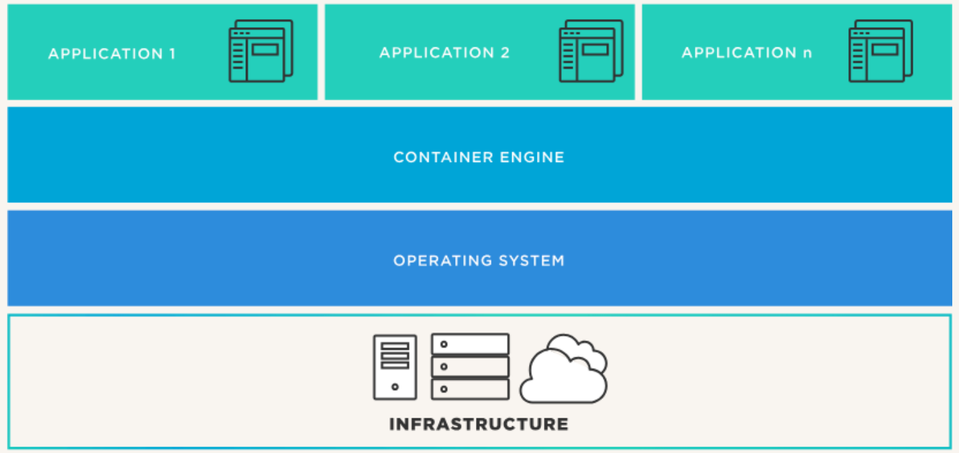
- The concept of containerization and process isolation is decades old. However, the emergence in 2013 of the open-source Docker—an industry standard for containers with simple developer tools and a universal packaging approach—accelerated the adoption of this technology.
- Today, organizations increasingly use containerization to create new applications and modernize existing applications for the cloud.
- According to a report from Forrester[1], 74 percent of US infrastructure decision-makers say that their firms are adopting containers within a platform as a service (PaaS) in an on-premises or public cloud environment.

Benefits of Containerization

- Portability and consistency - build once, run anywhere"
  - Eliminates “it works on my machine” issues
  - Consistent behavior across dev, test, and production
- Increased speed and agility
  - Lightweight → no guest OS needed
- Process and Dependency Isolation\
  - Each container has its own user space, process tree, and network stack
- Simplified Operational Management
  - Standardized deployment units → manage containers, not entire servers
- Rapid Scalability
  - Experiences a spike in demand → new container instances can be provisioned almost instantly

Containerization Architecture

- **The underlying IT infrastructure**: is a base layer that includes the physical compute resources (for example, desktop computer, bare-metal server).
- **Host operating system**: runs on the physical or virtual machine. The OS manages system resources and provides a runtime environment for container engines.
- **Container engine:** provides the execution environment for container images (read-only templates containing instructions for creating a container). Container engines run on top of the host OS and virtualize the resources for containerized applications.
- **Containerized applications**: consists of the software applications run in containers.



How Does Containerization Work?

The containerization development follows a logical, step-by-step process that moves an application from source code to a running, isolated instance.

Step 1: Define the environment

- The process begins with the developer creating a file (a popular choice is a Dockerfile). This file acts as a recipe or set of instructions for building the application's environment.
- It specifies everything needed, including:
  - A base image to start from (Example, a specific version of Debian or a language-specific runtime like Python 3.9)
  - Commands to copy the application's source code into the image
  - Instructions to install necessary packages, libraries, and other dependencies
  - The command to execute when the container starts

How Does Containerization Work?

Step 2: Build the container image

- Using the instructions in the file, a developer uses a command to create a container image. This image is a static, immutable, and portable file that acts as a self-contained blueprint for the application.
- It encapsulates the application code and all its dependencies into a single, layered package. Think of the image as a class in object-oriented programming; it's the template from which running instances will be created.

Step 3: Store and distribute the image in a registry

- Once built, the container image is pushed to a container registry. A registry is a centralized repository for storing and managing images. For enterprise use, a private, secure registry like *Docker Hub, AWS ECR, GCP Artifact Registry, or Azure Container Registry* is essential.
- Storing the image in a registry makes it easy to share across teams, version control, and access from any server in your production environment.

How Does Containerization Work?

Step 4: Run the container

- The final step is to create a **running instance of the image**, which is the **container** itself.
- A command is sent to the container engine to run a specific image from the registry. The engine then uses the host operating system's kernel to:
  - Create an isolated environment for the container using kernel features like *namespaces*. This gives the container its own dedicated network stack, process tree, and file system view, making it believe it's running on its own OS.
  - Allocate and limit resources like CPU and memory, ensuring that one container cannot monopolize the host's resources.
  - Execute the startup command defined in the original file, bringing the application to life inside its isolated sandbox.

What are containerization types of tools?

The container ecosystem is composed of several types of tools that work together.

- **Container engine:** The core software that runs and manages containers.
  - E.g. Docker Engine
- **Container image builders:** Tools used to create container images from a definition file.
  - E.g. Docker CLI (docker build)
- **Container registries:** Storage systems for container images.
  - E.g. Docker Hub (public registry), AWS Elastic Container Registry (ECR), Google Artifact Registry / GCP Container Registry, Azure Container Registry (ACR)
- **Container orchestration platforms:** For managing containers at scale, an orchestration platform like Kubernetes is essential. It automates deployment, scaling, and networking.
  - E.g. Kubernetes – most widely adopted orchestration platform,
  - Docker Swarm – simpler orchestration for small clusters
  - Amazon ECS/EKS, Google Kubernetes Engine (GKE), Azure AKS – managed orchestration

Container orchestration

- Today an organization might have hundreds or thousands of containers—an amount that would be nearly impossible for teams to manage manually. This is where container orchestration comes in.
- **A container orchestration platform** schedules and automates management like container deployment, networking, load balancing, scalability and availability.
- **Kubernetes**, the most popular container orchestration tool available, is an open-source technology (originally open-sourced by Google, based on their internal project called Borg) that automates Linux container functions. Kubernetes works with many container engines, like Docker Engine. It also works with any container system that conforms to the Open Container Initiative (OCI) standards for container image formats and runtimes.
- While Kubernetes is the industry standard, other popular container orchestration platforms include Apache Mesos, Nomad and Docker Swarm.

References

https://cloud.google.com/discover/what-is-containerization
https://www.ibm.com/think/topics/containerization
https://ubuntu.com/blog/containerization-vs-virtualization

DevOps

Docker

Chandara Chea

DD/MM/YYYY

Week IV

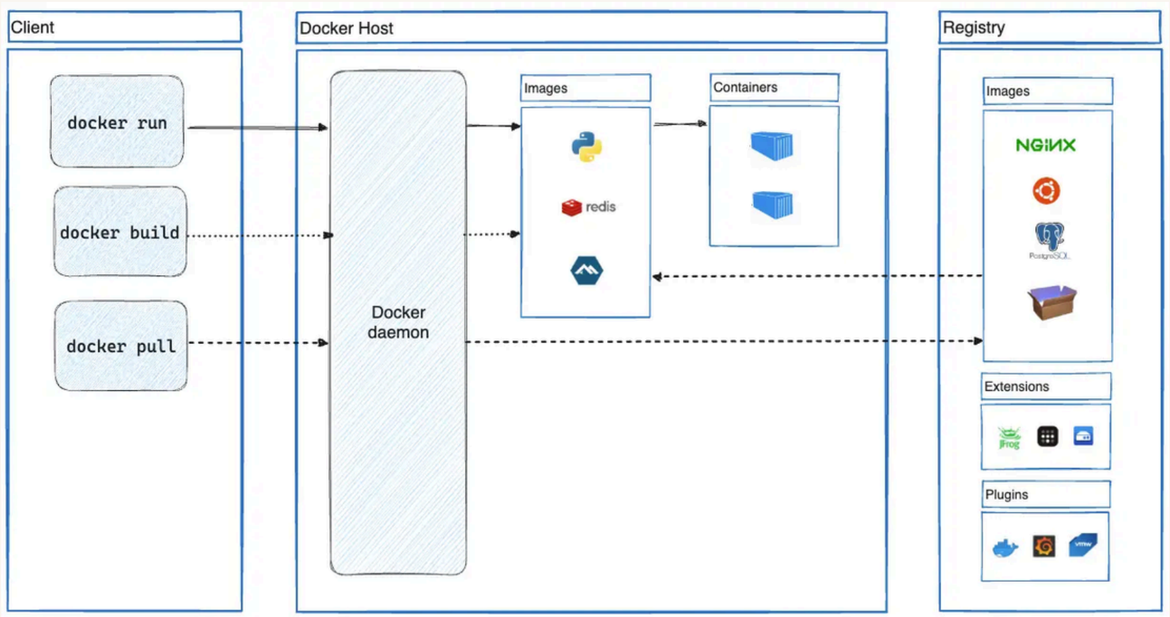
What is Docker?

- **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.
- Docker provides the ability to package and run an application in a **loosely isolated environment** called a **container**. The isolation and security lets you run many containers simultaneously on a given host.



Docker Architecture

- Docker uses a **client-server architecture.**
- The Docker client talks to the **Docker daemon *(dockerd)***, which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.
- The Docker client and daemon communicate using a 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.

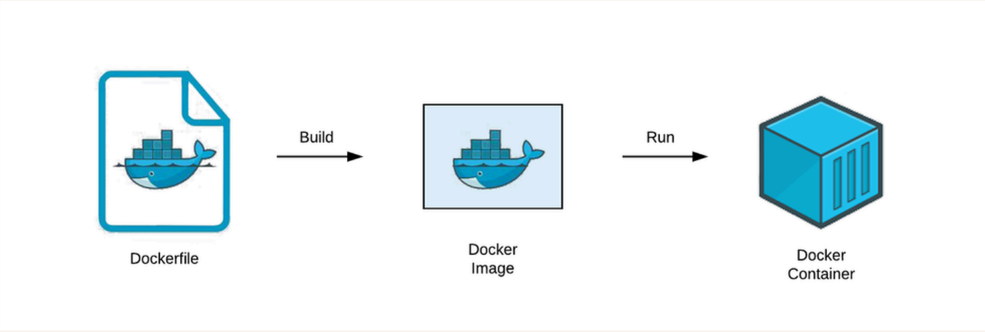


Docker Components

- **The Docker daemon (dockerd)** listens for 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 (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.
- **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.
- **An image** is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization.
- **A container** is a runnable instance of an image. You can create, start, stop, move, or delete a container using the 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.

Docker Image vs Dockerfile

- **A Docker image** is like a blueprint or snapshot of an application and its environment.
  - It contains everything your application needs to run:
  - The base operating system (e.g., Ubuntu, Alpine)
  - The application code
  - Dependencies (libraries, runtimes, tools)
  - Configuration (environment variables, startup commands)
  - Images are read-only templates — you can create multiple containers (running instances) from the same image.
- **A Dockerfile** is a text file that contains a list of instructions to build a Docker image.
  - Each instruction tells Docker what to do — such as which base image to use, what files to copy, and what command to run.



Dockerfile

```
FROM node:22-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "./src/index.js"]
```

Docker Hands-on Lab

https://kodekloud.com/free-labs/docker/docker-run

