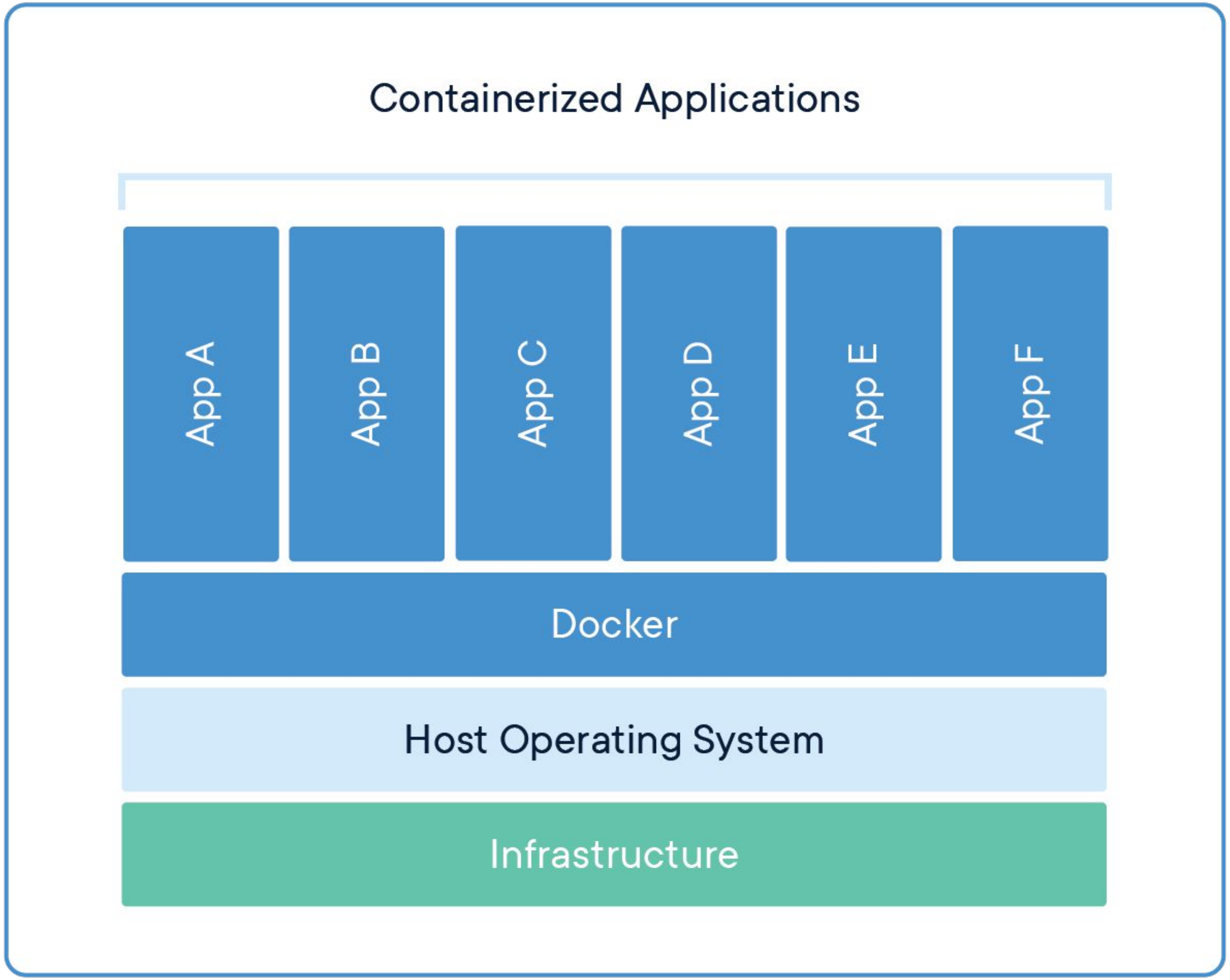
Containers (Docker)

What is a Container?

A **container** is a lightweight, standalone, and executable package that includes everything needed to run a piece of software—code, runtime, system tools, libraries, and settings. Containers provide a consistent environment from development to production, ensuring that software behaves the same regardless of where it is deployed.

Containers are isolated from each other and from the host operating system, yet they share the kernel of the host OS. This makes containers more efficient than traditional virtual machines (VMs) in terms of resource utilization, as they don't require a full OS installation for each instance.


Containerized Applications
App A
App B
App C
App D
App E
App F
Docker
Host Operating System
Infrastructure

Container vs Images

Container: A container is a running instance of an image. It is the active environment where an application runs, utilizing the file system, code, dependencies, and libraries bundled in the image.

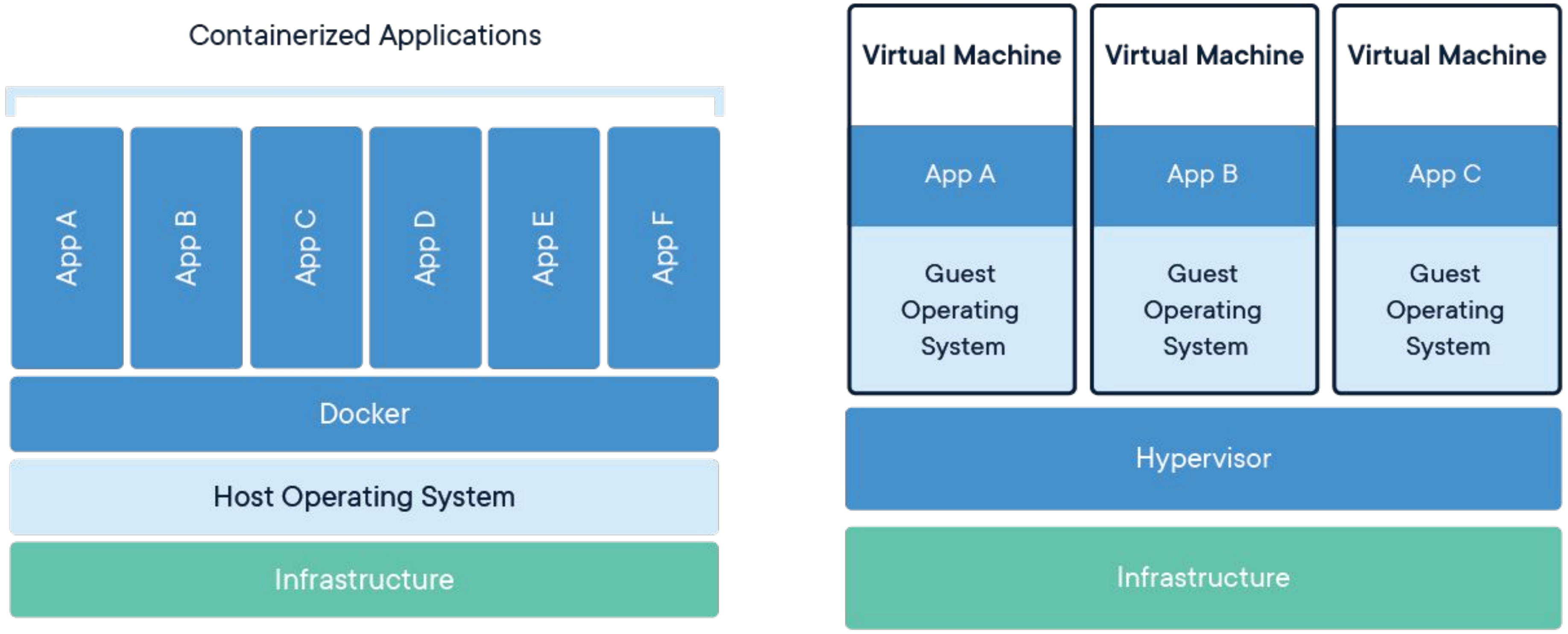
Image: An image is a static snapshot of the environment and software you want to run. It contains the instructions to create a container, including the application code, libraries, environment variables, and configurations. Think of an image as a blueprint, and a container as the actual running instance based on that blueprint.

Key Differences:

- **Image** is a read-only template, while a **container** is a running instance of that image.
- Containers can be started, stopped, and destroyed, but images remain static until updated or modified.

Container vs Virtual Machines

Aspect	Container	Virtual Machine (VM)
Isolation	Shares the host OS kernel, but each container is isolated.	Completely isolated with its own OS and kernel.
Resource Usage	Lightweight, as they share the host OS kernel.	Heavyweight, requires dedicated OS and resources.
Startup Time	Very fast (seconds).	Slower startup (minutes) due to OS boot process.
Performance	Near-native performance since no hypervisor is involved.	Slightly slower due to the overhead of running a full OS.
Portability	Highly portable, runs consistently across environments.	Less portable, requires compatible hypervisor and OS setup.
Size	Smaller size, usually in MBs.	Larger size, usually in GBs due to the full OS installation.


Containerized Applications
App A
App B
App C
App D
App E
App F
Docker
Host Operating System
Infrastructure
Virtual Machine
App A
Guest Operating System
Virtual Machine
App B
Guest Operating System
Virtual Machine
App C
Guest Operating System
Hypervisor
Infrastructure

Docker Architecture and its Components

Docker is a platform that enables developers to build, run, and share applications using containers. Docker's architecture is composed of several key components:

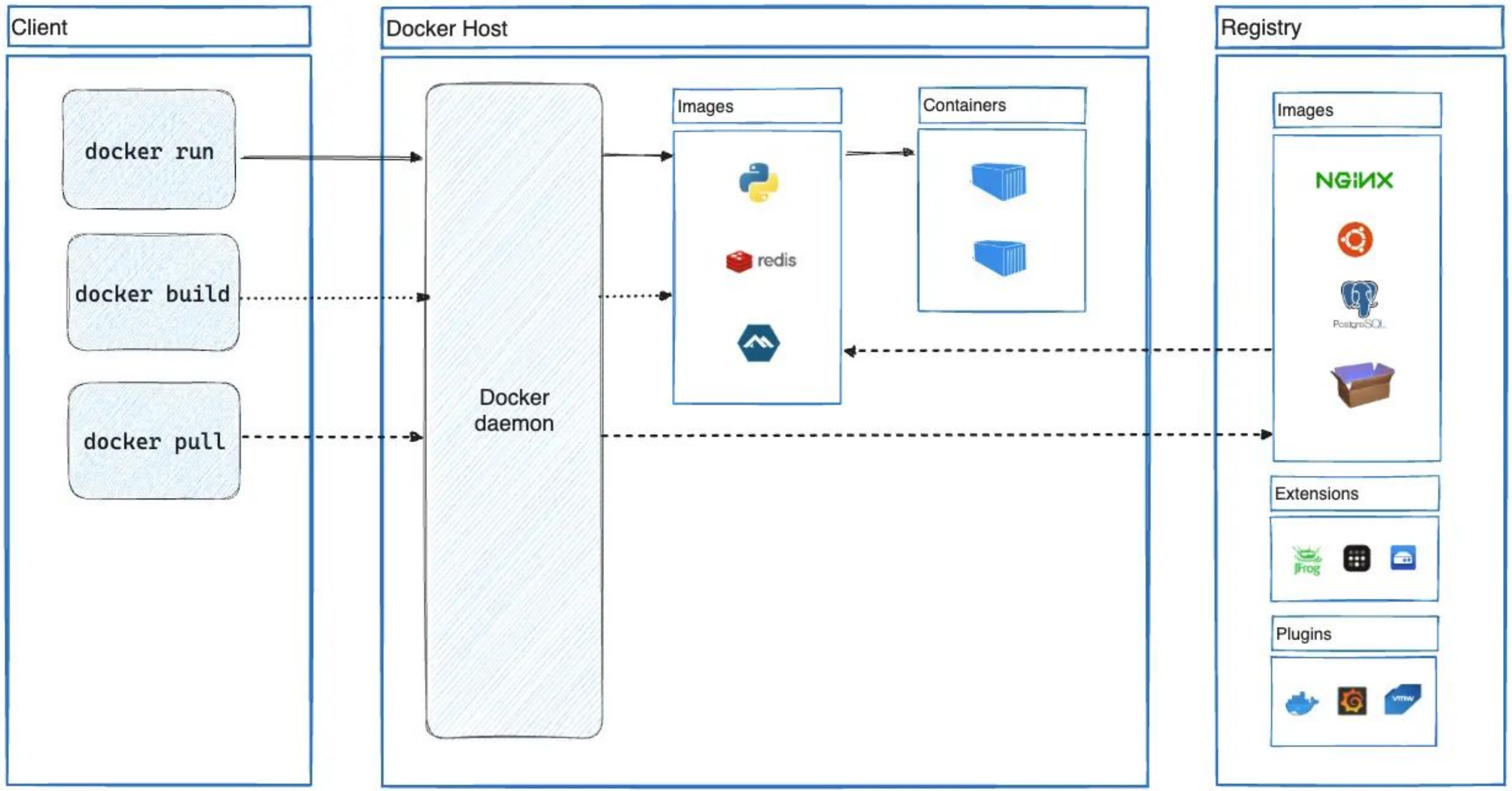
1. **Docker Client**: The user interface where you interact with Docker. When you run commands like `docker run`, the client communicates with the Docker Daemon to manage containers.
2. **Docker Daemon (dockerd)**: The background service that manages Docker objects (containers, images, networks, volumes). It listens to Docker API requests and handles the tasks of creating, running, and stopping containers.
3. **Docker Images**: These are read-only templates that contain a set of instructions to create a container. Docker images are built from Dockerfiles, which define the application's environment.

4. **Docker Containers**: The runnable instance of a Docker image. Containers are lightweight and isolated, but they share the kernel of the host operating system.

5. **Docker Registry**: A place where Docker images are stored. The most common public registry is Docker Hub, but you can also have private registries. Registries allow you to store and distribute your images.

6. **Dockerfile**: A text file with a set of instructions used to build a Docker image. It contains commands like `COPY`, `RUN`, and `CMD` to specify how the image should be built and what it will contain.

7. **Volumes**: A feature in Docker that allows containers to persist data even after the container is stopped or destroyed.


Client
docker run
docker build
docker pull
Docker Host
Docker daemon
Images
redis
Containers
Registry
Images
NGINX
PostgreSQL
Extensions
JFrog
Plugins
vmw

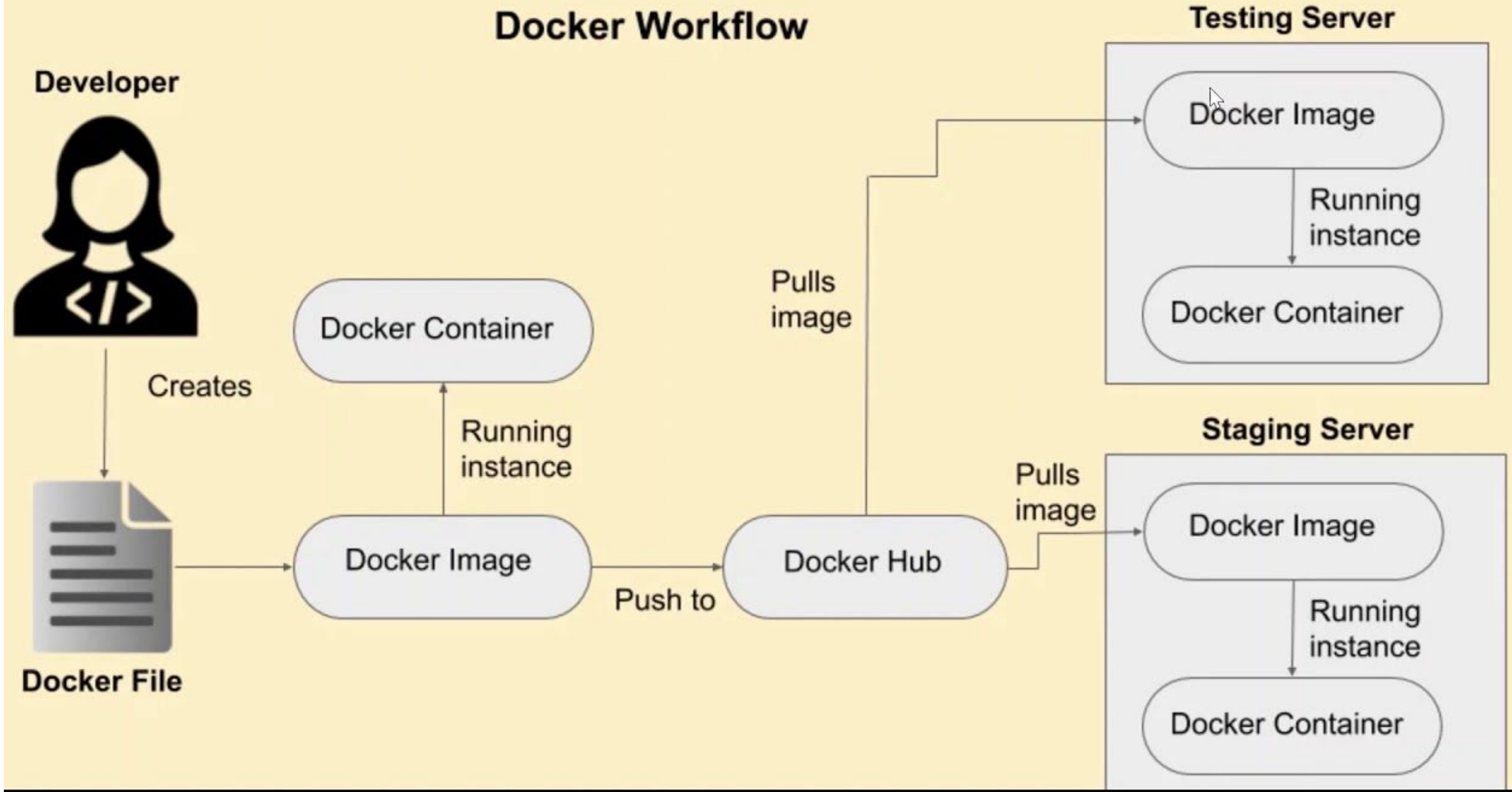
Docker Workflow

Build: You create an image using a Dockerfile.

Ship: You push the image to a Docker registry.

Run: You pull the image from the registry and create a container to run it.

Docker Workflow



Basic Docker Commands

Here are some of the key Docker commands for managing containers, images, and other Docker resources:

1. **docker --version**: Shows the installed Docker version.
2. **docker build**: Builds an image from a Dockerfile.

   ```
   docker build -t <image-name> .
   ```

3. **docker pull**: Downloads an image from a Docker registry.

   ```
   docker pull <image-name>
   ```

- **docker run**: Creates and starts a container from an image.
  ```
  docker run -d --name <container-name> <image-name>
  ```

  Options:

  `-d`: Run container in detached mode (background).

  `--name`: Assign a name to the container.

  `-p`: Map ports (e.g., `-p 8080:80` maps port 80 in the container to port 8080 on the host).

- **docker ps**: Lists all running containers.

  ```
  docker ps
  ```

- **docker stop**: Stops a running container.
  ```
  docker stop <container-id>
  ```

- **docker start**: Starts a stopped container.
  ```
  docker start <container-id>
  ```

- **docker rm**: Removes a container (it must be stopped first).
  ```
  docker rm <container-id>
  ```

- **docker rmi**: Removes an image.
  ```
  docker rmi <image-id>
  ```

- **docker exec**: Runs a command inside a running container.
  ```
  docker exec -it <container-id> <command>
  ```

- **docker logs**: Shows logs from a container.

  ```
  docker logs <container-id>
  ```

- **docker inspect**: Provides detailed information about a container or image.

  ```
  docker inspect <container-id>
  ```

- **docker network**: Manages Docker networks.

  ```
  docker network ls
  ```

- **docker volume**: Manages Docker volumes.

  ```
  docker volume ls
  ```

Summary

Docker containers provide a lightweight, efficient, and consistent way to run applications across environments.

They differ from virtual machines in that they share the host OS kernel, making them faster and more resource-efficient.

Docker's architecture includes components like the Docker client, daemon, images, containers, and registries.

By mastering Docker commands, you can effectively manage containerized applications, improving the development, testing, and deployment processes.

Lab

https://yourtechie.hashnode.dev/dive-into-docker-a-hands-on-lab-for-getting-started

Install Docker Desktop - https://docs.docker.com/desktop/install/windows-install/

Docker Compose

What is Docker Compose?

Docker Compose is a tool for defining and running multi-container Docker applications.

Key Features:

- Uses a single YAML file to configure application services.
- Allows for simplified service orchestration.
- Supports consistent development and deployment environments.

Why use Docker Compose?

Multi-container orchestration: Manage complex microservices setups.

Development Consistency: Avoids the "it works on my machine" problem.

CI/CD Automation: Integrates seamlessly with testing and deployment pipelines.

Key Concepts of Docker Compose

Services: Defines the individual containers.

Networks: Allows services to communicate.

Volumes: Persists data for containers.

Environment Variables: Pass configuration data to containers.

Labs

- Lab 1: Create Multi-Containers with Docker Compose
  - https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-docker-compose/
- Lab 2:
  - Create a Dockerfile
  - Setup Azure Container Registry
  - Push to Azure Container Registry
  - Deploy to Azure Container Instances
  - https://yourtechie.hashnode.dev/creating-your-custom-docker-image-on-azure-container-registry-and-deploying-with-container-instances

Assignment

https://docs.docker.com/get-started/workshop/