Docker

Contents

1. YAML

What is YAML?

- YAML stands for **"YAML Ain't Markup Language"**.
  - YAML is not a markup language like HTML.
  - It is a **serialization format**, much like `.doc, .pdf`, or `.json`.
- YAML is commonly used for **configuration files** and **data exchange** between languages and structures.

Benefits of YAML

1. **Readability**:
   - YAML is more human-readable compared to JSON and HTML.
   - Its syntax is clean and straightforward.
2. **Flexibility**:
   - YAML supports **complex data structures** such as nested lists and dictionaries.
   - It accommodates various data types: strings, numbers, arrays, objects, Booleans, etc.
3. **Wide Adoption**:
   - YAML integrates with tools such as Kubernetes, Terraform, Ansible, and Grafana.
   - Extensively used for configuration in these environments.

YAML Syntax Overview

1. **Indentation**:
   - YAML is sensitive to indentation.
   - Errors often occur due to misplaced spaces or tabs.
2. **Key-Value Pairs**:
   - Syntax: `key: value`
   - Supports multiple data types:
     - Strings

        - Numbers
        - Booleans
        - Floats
        - Arrays
3. **Anchors and Aliases**:
    - **Anchor (`&`)**:
        - Assigns a label to a set of values or configuration settings.
    - **Alias (`*`)**:
        - References the anchor to reuse values.
    - Example:

```
default_settings: &default
  setting1: value1
  setting2: value2
custom_settings:
  <<: *default
  setting3: value3
```

4. **Comments**:
    - Use # to add comments.
5. **Null Values**:
    - Represent null values using `null` or ~.

Example YAML Configurations

1. Example 1: CI/CD Stages (GitLab)

```
stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script:
    - echo "Building the project..."
```

```
test_job:
  stage: test
  script:
    - echo "Running tests..."

deploy_job:
  stage: deploy
  script:
    - echo "Deploying the project..."
```

- YAML's indentation and key-value pair structure define workflows clearly.

2. Example 2: Ansible Configuration

```
- name: Install packages
  hosts: all
  tasks:
    - name: Install nginx
      apt:
        name: nginx
        state: present
```

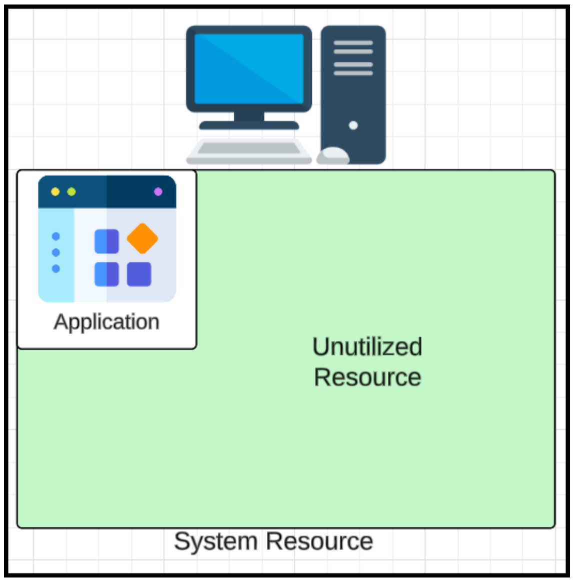
Key Takeaways

- YAML is a **file format** akin to `.pdf` or `.json`.
- It is widely adopted in tools such as Kubernetes, Ansible, and Terraform.
- YAML excels in **readability** and **flexibility**.
- It uses **indentation-sensitive syntax** for clarity.
- YAML's **anchors** and **aliases** enable reusability.
- Comments and null values are supported and easy to implement.

2. Physical Machines, Virtual Machines, and Containers

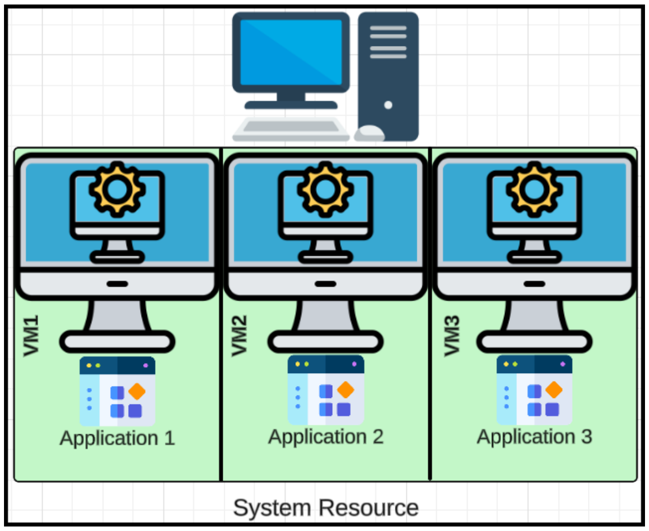
Physical Machines

- A physical machine refers to hardware like laptops, desktops, or servers.
- Traditionally, IT companies ran applications directly on physical machines.
    - Example: Running a simple web server on a physical machine.
    - **Limitation:** Resource underutilization since one application cannot fully utilize the machine's capacity.
    - Running multiple applications on one physical machine leads to interference between applications.
- This inefficiency led to the development of virtualization.



Virtual Machines

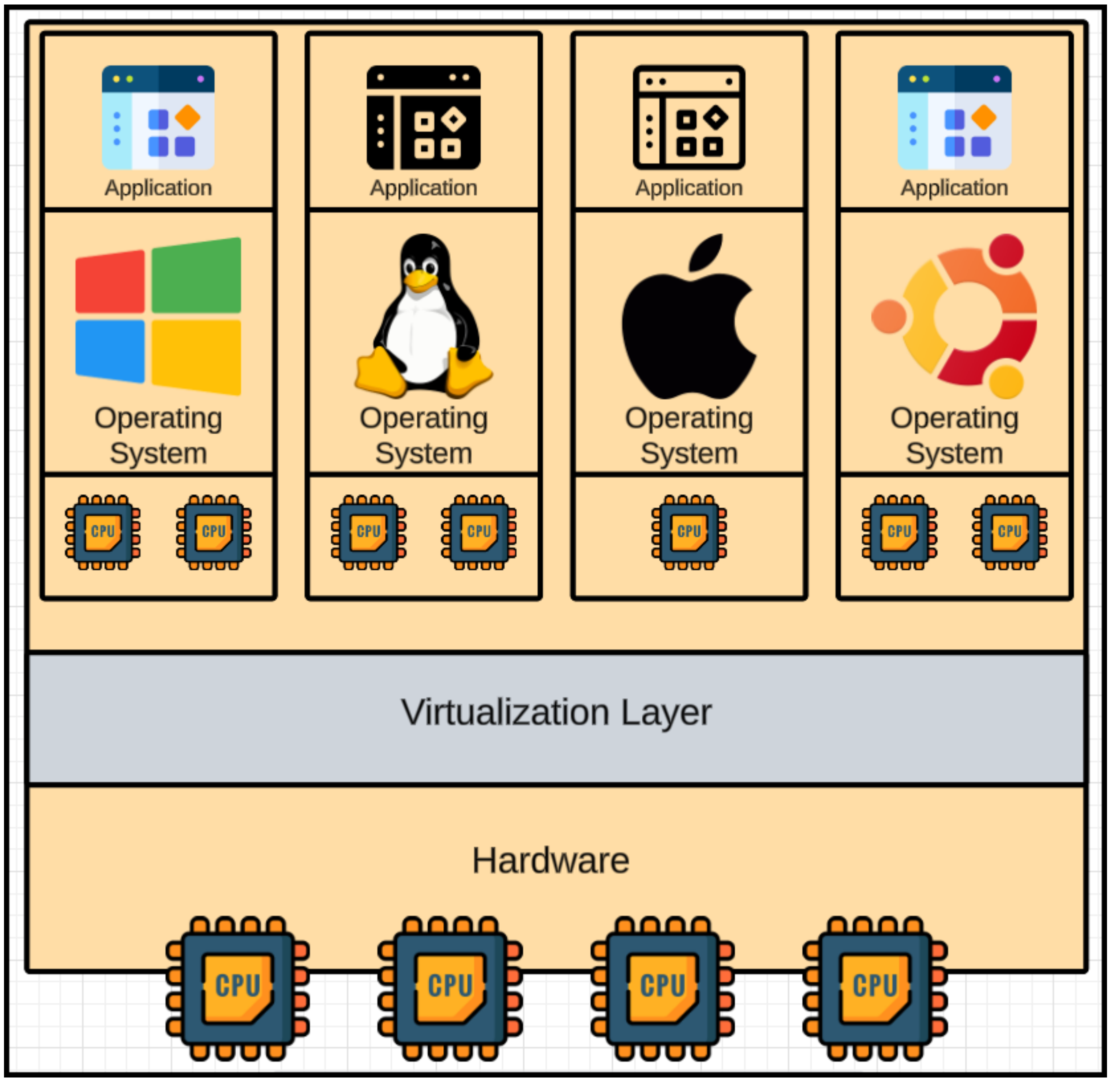
- **Concept:** Virtualization technology was introduced to improve resource utilization.
- **How it works:**
    - A software called a **hypervisor** is installed on the physical machine's infrastructure.
    - The hypervisor divides the physical machine's resources into multiple **virtual machines (VMs)**.
- **Benefits:**
    - Each virtual machine acts as a separate entity, meaning it operates as an independent system with its own dedicated operating system,



virtualized hardware, and allocated resources. In practical terms, this isolation ensures that each VM can run distinct applications, configurations, and processes without affecting others, similar to how separate physical machines function.

   - Multiple applications can run in isolated VMs without interfering with each other.

- **Structure of Virtual Machines:**
   - Infrastructure (physical machine)
   - Hypervisor (virtualization layer)
   - Multiple virtual machines
      - Each VM has its own operating system (OS) and application.



- **Advantages:**
    - Better resource utilization compared to physical machines.
    - Applications run in isolation.
- **Limitations:**
    - Even with virtualization, some resources remain underutilized.

Hypervisor

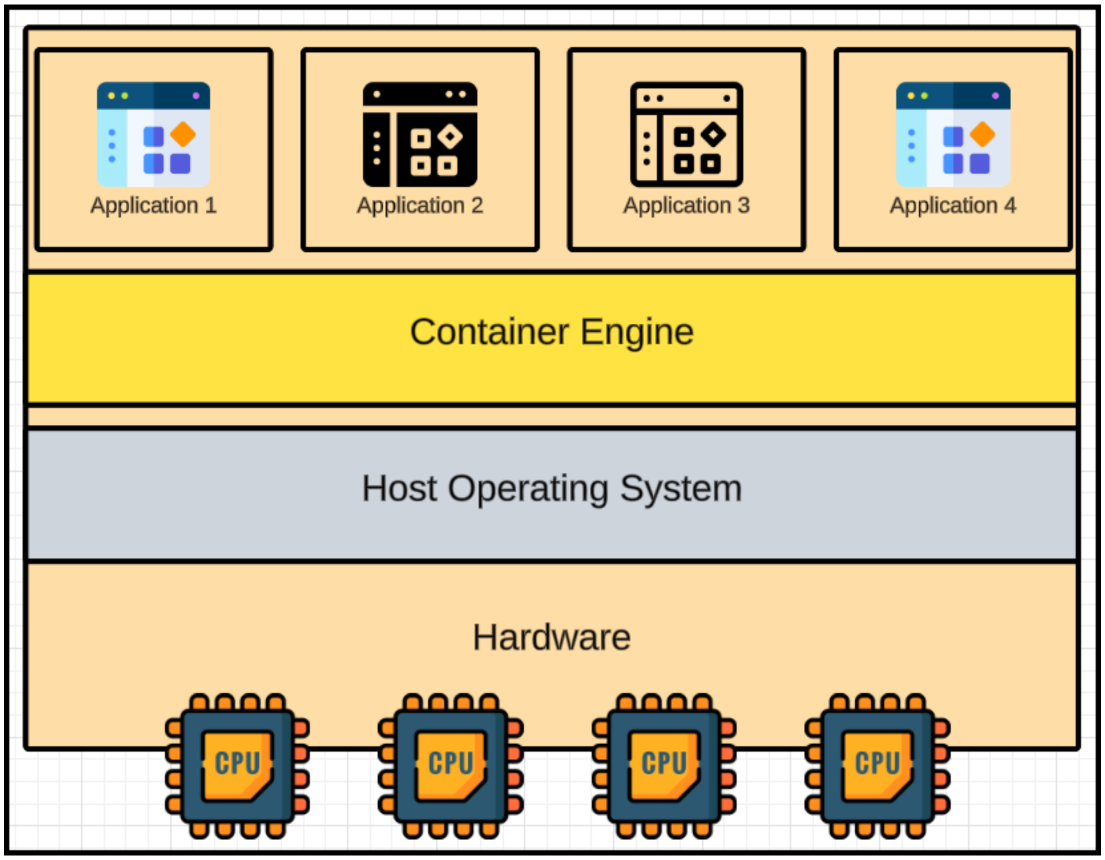
- A **hypervisor**, also known as a **virtual machine monitor (VMM)**, is a software layer or hardware component that allows multiple operating systems (OS) or virtual machines (VMs) to run on a single physical machine by sharing the underlying hardware resources. Hypervisors are key to virtualization technology, which is widely used in data centers, cloud computing, and desktop environments.

Types of Hypervisors

1. **Type 1 Hypervisors (Bare-metal Hypervisors):**
    - Run directly on the physical hardware without the need for an underlying operating system.
    - Examples:
        - VMware ESXi, Microsoft Hyper-V, Xen
    - Advantages:
        - High performance and efficiency.
        - Direct access to hardware.
2. **Type 2 Hypervisors (Hosted Hypervisors):**
    - Run on top of an existing operating system (host OS) and rely on it for resource management.
    - Examples:
        - VMware Workstation, Oracle VirtualBox, Parallels Desktop
    - Advantages:
        - Easy to install and use on personal computers.
        - Good for testing and development.

Containers

- Containers were introduced to further optimize resource utilization by addressing the inefficiencies of virtual machines. Unlike virtual machines, containers share the host operating system's kernel, eliminating the need for each instance to include a full OS. This makes containers more lightweight, faster to start, and efficient in resource usage compared to virtual machines, which are bulkier due to their full OS overhead.
- **Concept Inspiration:**
    - Containers are similar to shipping containers in transport.
    - Each container is self-contained, holding specific applications and their dependencies, without interfering with others.
- **How it works:**
    - A single physical machine can host multiple containers.
    - Containers share the same OS kernel, making them lightweight compared to VMs.
- **Structure of Containers:**



    - Infrastructure (physical machine)
    - Operating System (installed on the machine)
    - Container engine (e.g., Docker) installed on the OS
    - Multiple containers created by the container engine
        - Each container runs an application in isolation. For instance, a container might host a web server like Nginx or Apache, along with all its dependencies, allowing it to run independently of other applications on the same machine.
- **Benefits:**
    - Optimal resource utilization compared to VMs.
    - Applications run in isolated containers without interference.
    - Lightweight and faster to deploy than virtual machines.

Container Engines

- **Definition:** Tools used to create and manage containers.
- Examples:
    - Docker (most commonly used container engine).
    - ContainerD (used by Kubernetes and AWS EKS).
    - GitLab’s Calico (among others).
- Core functionality and commands are similar across container engines; however, each engine may have unique features or optimizations tailored for specific use cases. For example, Docker provides a comprehensive ecosystem with tools like Docker Compose for multi-container setups, while ContainerD focuses on simplicity and integration with Kubernetes.

Key Differences: Virtual Machines vs Containers

1. Architecture

Virtual Machines (VMs):

- Each VM includes a full operating system, a hypervisor layer, and emulates hardware.
- Runs on a **hypervisor**, which interacts with the host machine's hardware.
- Heavyweight due to the inclusion of a full OS.

Containers:

- Containers share the host operating system kernel and isolate applications at the process level.
- Uses a **container engine** (e.g., Docker, Podman) rather than a hypervisor.
- Lightweight since they don't require a full OS for each instance.

---

2. Isolation

Virtual Machines:

- Provides strong isolation as each VM operates as an independent system.
- VMs are better suited for scenarios requiring complete OS separation.

Containers:

- Provide process-level isolation but share the host OS kernel.
- Isolation is less complete than VMs, which may raise security concerns in certain cases.

---

3. Resource Utilization

Virtual Machines:

- Heavier due to full OS replication, resulting in higher memory and storage overhead.
- Slower startup and shutdown times.

Containers:

- Lightweight, as they share the host OS and reuse many components.
- Faster startup and shutdown times with lower resource overhead.

---

4. Portability

Virtual Machines:

- VMs are less portable due to their larger size and dependencies on the hypervisor.
- Migration can be slower and more resource-intensive.

Containers:

- Highly portable, as containers package the application and its dependencies together.
- Easy to run on any system with a container engine (e.g., Docker, Kubernetes).

---

5. Use Cases

Virtual Machines:

- Running different operating systems on the same hardware.
- Legacy applications that require specific OS configurations.
- Strong isolation for security-critical environments.

Containers:

- Microservices architectures.
- Rapid development, testing, and deployment pipelines.
- Cloud-native applications and high-density application hosting.

---

6. Performance

Virtual Machines:

- Performance overhead is higher due to OS emulation and hypervisor management.
- Good for applications requiring strong isolation and resource guarantees.

Containers:

- Closer to bare-metal performance due to shared OS kernel.
- Ideal for workloads that demand high scalability and efficiency.

---

7. Security

Virtual Machines:

- Stronger security due to complete OS isolation.
- Each VM operates as an independent system.

Containers:

- More prone to kernel-level vulnerabilities since the host OS kernel is shared.
- Requires additional security measures (e.g., SELinux, AppArmor, and namespaces).

8. Comparison Table

Feature	Virtual Machines	Containers
Architecture	Full OS per VM	Shared host OS kernel
Overhead	High	Low
Startup Time	Minutes	Seconds
Isolation	Strong	Moderate
Portability	Moderate	High
Use Cases	Legacy apps, mixed OSs	Microservices, cloud-native
Security	Stronger	Requires careful management

Summary

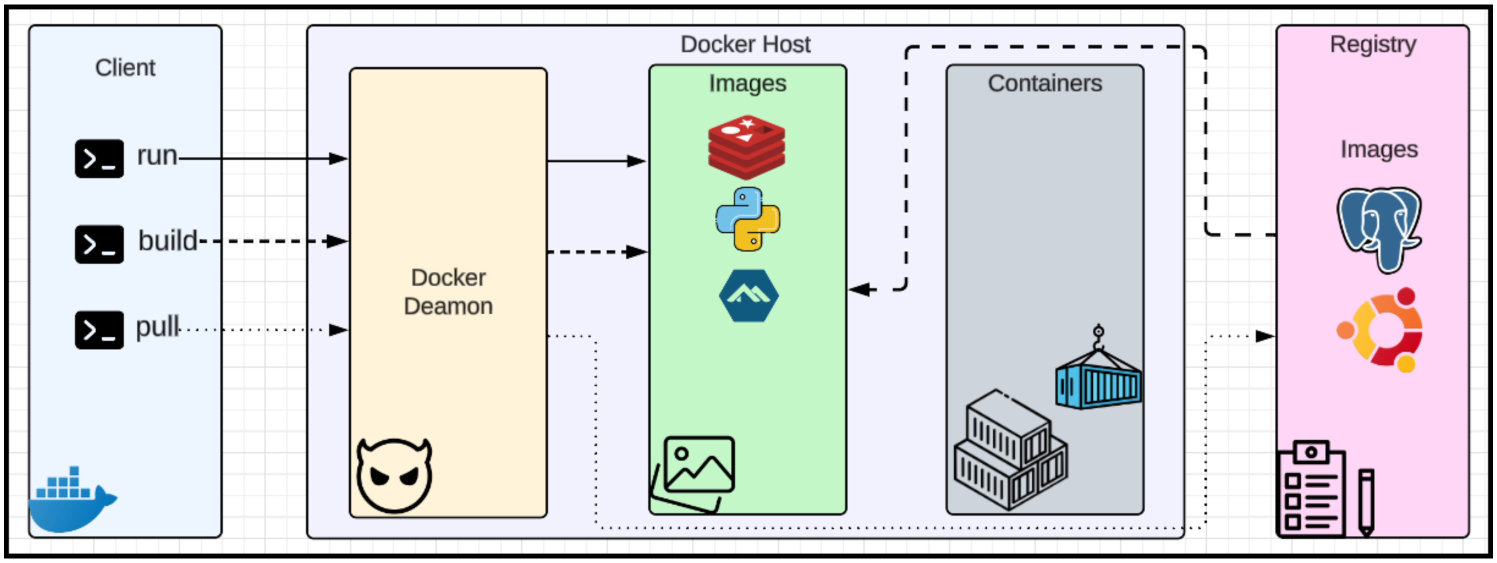
- Use **Virtual Machines** when you need strong isolation, run multiple OSs, or host legacy applications.
- Use **Containers** when you need lightweight, portable environments for modern, scalable applications, especially in DevOps and microservices architectures.

3. Docker

Docker Is Not Equal to Containers

- Docker is just one type of container engine, much like Xerox is a company, not a technology for photocopying. Much like Maggi which is a company that makes noodles.
- There are other container engines like Podman, rkt, etc., but Docker is widely used.

Core Components of Docker



Docker Client:

- Acts as the interface for users to interact with Docker. It sends commands (e.g., `docker run, docker build`) to the Docker daemon.
- When user is interacting with docker, user is just interacting with docker client and docker client interacts with other components.

Docker Host:

- The environment where Docker operates. It includes:
    - Docker Daemon: Handles building, running, and managing containers.
    - Images: Blueprints or templates for containers.
    - Containers: The actual running instances derived from images.

Docker Daemon

- **Core Functions**:
  - **Build Images**:
    - Creates new image using a Dockerfile or fetches pre-existing images from a registry.
  - **Run Containers**:
    - Runs containers based on the images.
    - Manages multiple containers.

Docker Images

- **Definition**:
  - Blueprints or templates for applications.
  - Read-only; cannot directly run applications on images.
  - To run an application, a container must be built from the image.
- **Layers in Images**:
  - Images consist of multiple layers.
  - Layers make images lightweight and reusable.
  - Updates to images involve adding new layers instead of modifying the existing image.
  - Example:
    - Initial image: 10 MB.
    - Added a layer: +2 MB.
    - Updated image: 12 MB.
    - Docker daemon fetches only the delta (new layer) instead of the full updated image.
- **Use Case for Layers**:
  - Example:
    - Base image: Ubuntu (100 MB).
    - Developer A builds an application on top, resulting in a 200 MB image.
    - Developer B builds another application, resulting in a 300 MB image.

        - Base image (Ubuntu) remains shared across both applications.
    - Changes to the base image affect all images built on top of it.

Docker Registry:

- Stores Docker images, similar to a GitHub repository for code. Popular registries include Docker Hub and private registries.
- Docker Deamon fetches the image from the registry, uses this image and builds a container.

Images vs Containers

Images:

- Immutable blueprints of containers.
- Comparable to negatives in photography—can create multiple identical containers.
- Read-only and stored in registries.

Containers:

- **Running instances of images,** i.e., the actual environment where applications run.

Image Layers

- Images are made up of layers to enable reusability and efficiency.
- Changes or updates to an image only create a new layer (delta), reducing the need to fetch the entire image again.

Practical Analogy

- Think of an image as a negative, and the container as the photo developed from it.
- Modifications to an image (negative) result in a different container (photo).

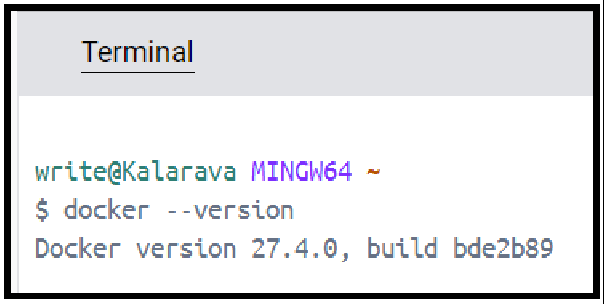
DevOps Integration

- Code: Stored in repositories like GitHub.
- Jenkins: A continuous integration tool that pulls code, builds Docker images, and deploys containers.
- Docker integrates with various tools in the DevOps pipeline to streamline application development and deployment.

4. Docker – Practice

- Install docker from the docker official website.
- After installation, open the docker app.
- Check the docker version using following command…

```
$ docker --version
Docker version 27.4.0, build bde2b89h
```



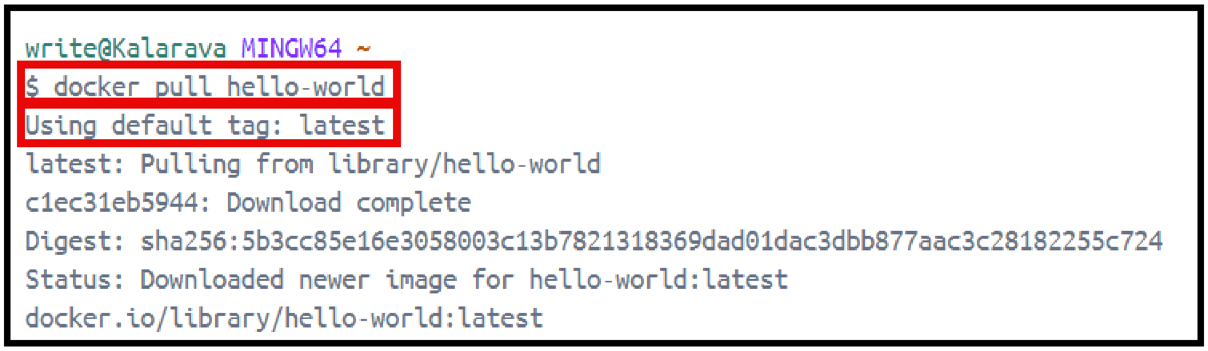
Terminal

```
write@Kalarava MINGW64 ~
$ docker --version
Docker version 27.4.0, build bde2b89
```

Retrieving Image

- The `docker pull` command is used to download Docker images from a registry (usually Docker Hub) to your local machine. This command is essential for retrieving images that you want to use to create containers.

```
$ docker pull hello-world
```

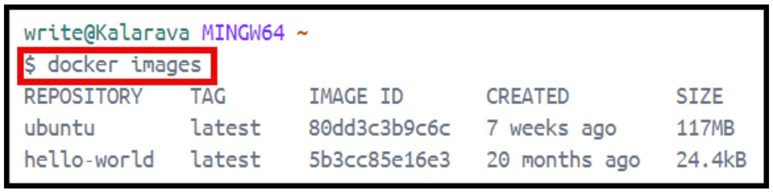


```
write@Kalarava MINGW64 ~
$ docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
c1ec31eb5944: Download complete
Digest: sha256:5b3cc85e16e3058003c13b7821318369dad01dac3dbb877aac3c28182255c724
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

- To get the latest image, use tag.
- TAG: The specific version of the image (e.g., `latest`, `1.0`, `stable`). If not specified, Docker will pull the image with the `latest` tag by default.

List all the available Images

- The `docker images` command is used to list all the Docker images that are available on your local machine. It provides details about the images that have been pulled or built locally.



```
write@Kalarava MINGW64 ~
$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
ubuntu        latest    80dd3c3b9c6c   7 weeks ago     117MB
hello-world   latest    5b3cc85e16e3   20 months ago   24.4kB
```

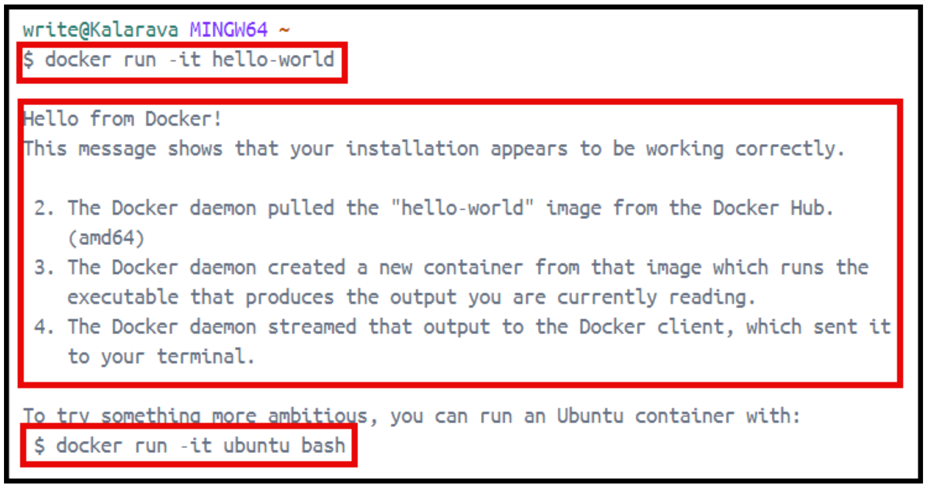
- Common Options
  - `-a, --all`: Show all images (default hides intermediate images).
  - `-q, --quiet`: Only show numeric IDs.

Start A New Container

- The `docker run` command is used to create and start a new container from a Docker image.
- Syntax:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

- Common Options
  - `-d, --detach`: Run the container in the background.
  - `-i, --interactive`: Keep STDIN open even if not attached.
  - `-t, --tty`: Allocate a pseudo-TTY
  - `-e, --env`: Set environment variables.

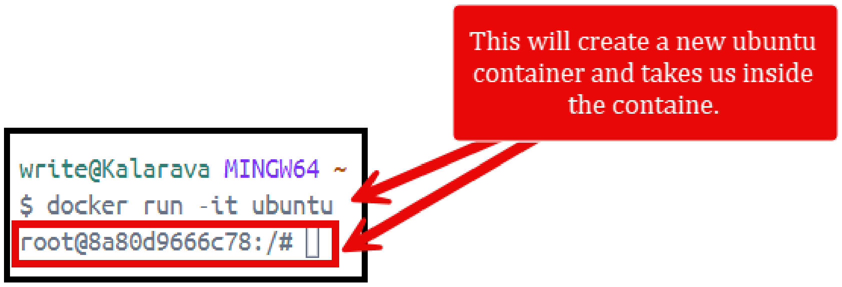


```
write@Kalarava MINGW64 ~
$ docker run -it hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash
```


This will create a new ubuntu container and takes us inside the containe.

```
write@Kalarava MINGW64 ~
$ docker run -it ubuntu
root@8a80d9666c78:/#
```

- To get out of the container, use `exit` command at the ubuntu terminal.

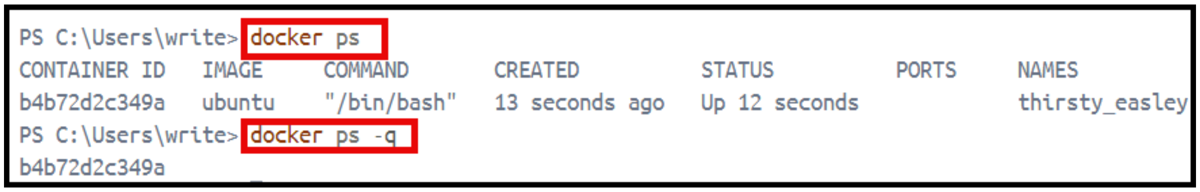
- Use `-d` option to run the container in detached mode.

List All the Running Containers

- The `docker ps` command is used to list all the running containers in Docker. It provides details about the containers currently running on your system.
- Syntax:

```
docker ps [OPTIONS]
```

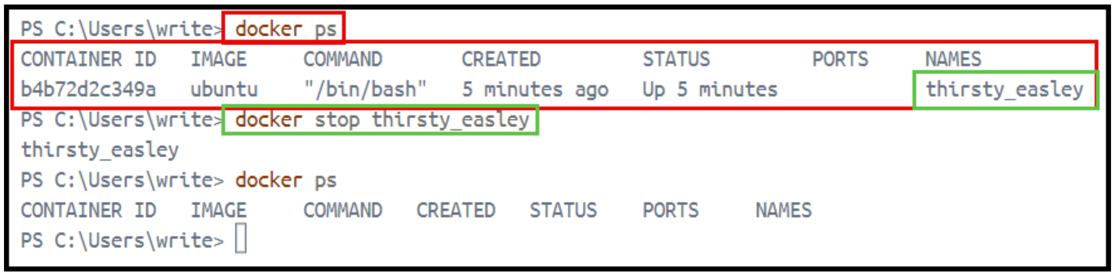
- Common Options
  - `-a, --all`: Show all containers (default shows just running containers).
  - `-q, --quiet`: Only display numeric IDs.
  - `-l, --latest`: Show the latest created container, including those that are not running.



```
PS C:\Users\write> docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS          PORTS     NAMES
b4b72d2c349a   ubuntu    "/bin/bash"   13 seconds ago   Up 12 seconds             thirsty_easley
PS C:\Users\write> docker ps -q
b4b72d2c349a
```

Stop Running Container

- The `docker stop` command is used to stop one or more running containers. This command sends a signal to the container, requesting it to gracefully stop its execution.



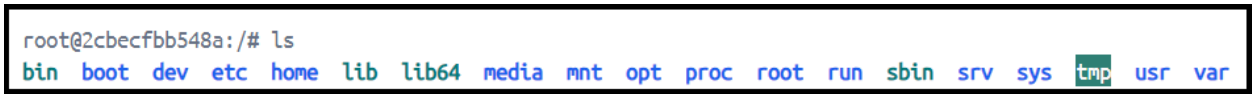
```
PS C:\Users\write> docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED         STATUS         PORTS     NAMES
b4b72d2c349a   ubuntu    "/bin/bash"   5 minutes ago   Up 5 minutes             thirsty_easley
PS C:\Users\write> docker stop thirsty_easley
thirsty_easley
PS C:\Users\write> docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
PS C:\Users\write>
```

Run a Command

- The `docker exec` command is used to run a command in a running container. This command is useful for executing tasks within a container that is already running, such as checking logs, managing files, or running applications.
- Syntax:

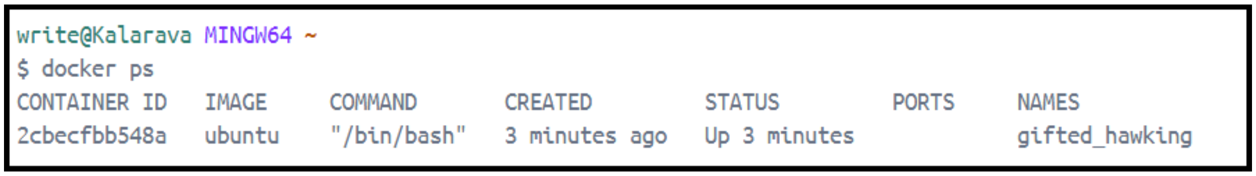
```
docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

- Common Options
  - `-i, --interactive`: Keep STDIN open even if not attached.
  - `-t, --tty`: Allocate a pseudo-TTY (useful for interactive processes).
  - `-d, --detach`: Run the command in the background.
- Example:
  - Before:



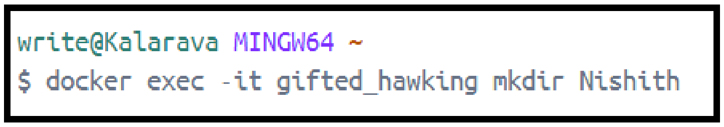
```
root@2cbecfbb548a:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

  - Container is running



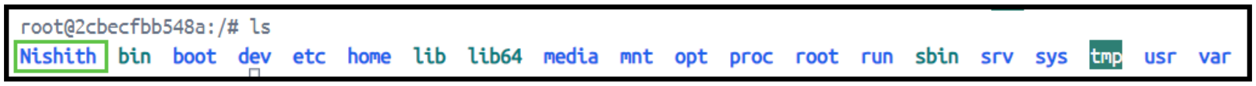
```
write@Kalarava MINGW64 ~
$ docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED         STATUS         PORTS     NAMES
2cbecfbb548a   ubuntu    "/bin/bash"   3 minutes ago   Up 3 minutes             gifted_hawking
```

  - Executing a command on running container…



```
write@Kalarava MINGW64 ~
$ docker exec -it gifted_hawking mkdir Nishith
```

  - After:



```
root@2cbecfbb548a:/# ls
Nishith  bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

Exit Without Stopping

- If you want to exit an interactive session within a container without stopping the container, you can do so by using the following key combination:

```
Ctrl + P followed by Ctrl + Q
```