**Docker**

Docker is an open-source platform that automates the deployment, scaling, and management of applications by using containerization. Here's a breakdown of what Docker is and how it works:

**1. Containerization**

* **Containers**: Docker containers are lightweight, standalone, and executable software packages that include everything needed to run a piece of software: the code, runtime, system tools, libraries, and settings.
* **Isolation**: Containers provide isolated environments, which means they can run consistently across different environments (e.g., your local machine, a cloud server) without being affected by the underlying system. This makes Docker great for avoiding the "it works on my machine" problem.

**2. Images**

* Docker containers are built from **images**, which are snapshots of the application and its dependencies at a specific point in time.
* **Dockerfile**: A Dockerfile is a script that contains instructions on how to build a Docker image. It specifies what software and dependencies to install and configure.

**3. Portability and Efficiency**

* **Portability**: Docker containers can run on any system that supports Docker (Linux, Windows, macOS), making them portable across different environments.
* **Efficiency**: Unlike virtual machines (VMs), Docker containers share the host system’s kernel, making them faster to start and more lightweight (they don’t include a full OS like VMs do).

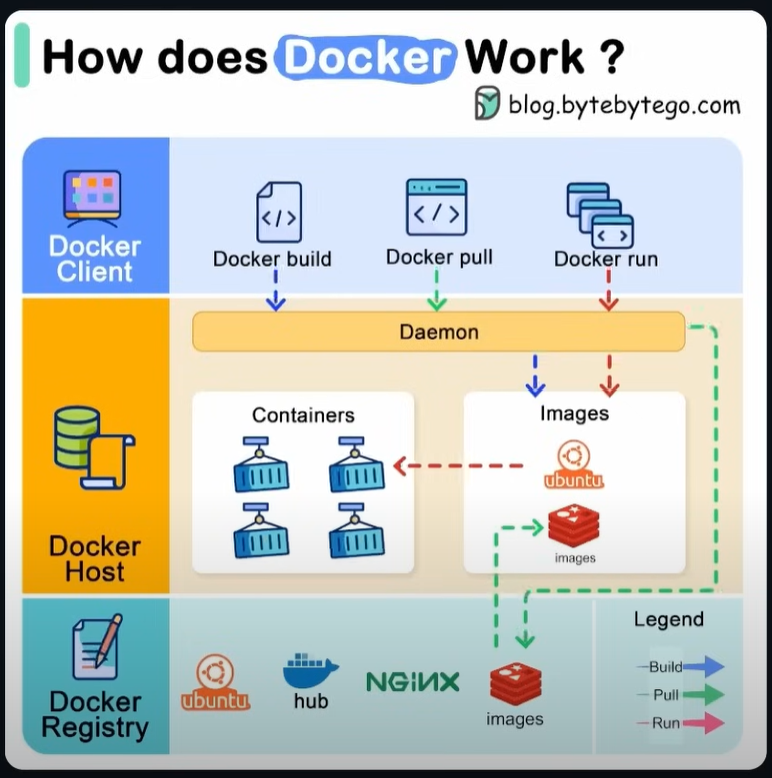
**4. Key Components of Docker**

* **Docker Engine**: The core part of Docker that runs and manages containers.
* **Docker Hub**: A public registry where Docker users can find and share container images. You can pull existing images or upload your own.
* **Docker Compose**: A tool to define and run multi-container Docker applications. It allows you to manage multiple containers as part of a single application.

**5. Use Cases**

* **Development and Testing**: Developers use Docker to create consistent environments across different stages of development, from testing to production.
* **Microservices Architecture**: Docker is widely used for building microservices, where each service runs in its own container.
* **Continuous Integration/Continuous Deployment (CI/CD)**: Docker plays a significant role in automating the testing and deployment of applications.

In short, Docker simplifies the process of deploying and managing applications by using container technology, which is portable, fast, and resource efficient.

****

**Docker Image vs Container vs Volume**

In Docker, the terms **Image**, **Container**, and **Volume** refer to different components of the containerization system. Each plays a specific role in how Docker packages, runs, and stores application data. Here's a detailed comparison:

**1. Docker Image**

* **Definition**: A Docker Image is a read-only template that contains the instructions for creating a Docker container. It includes the application code, runtime, libraries, environment variables, and any other dependencies required for the app to run.
* **Immutable**: Once built, an image cannot be modified. Instead, you can create a new image by layering changes over an existing image.
* **Layers**: Docker images are composed of multiple layers, each layer representing a change or instruction (e.g., adding files, installing dependencies). These layers are cached, which helps speed up the building process.
* **Building**: You typically build an image using a **Dockerfile**, which contains instructions to create the environment needed for the application.
* **Usage**: You use an image to instantiate containers. An image can be reused to create multiple containers.
* **Example**:
  + You might pull an image from Docker Hub, such as nginx, and use it to create a running web server container.

**2. Docker Container**

* **Definition**: A container is a running instance of a Docker image. It includes the application and all its dependencies, but also the runtime state, like memory, processes, and file system changes.
* **Ephemeral**: Containers are designed to be short-lived. Once a container stops, it can be restarted or removed, but its changes (outside of volumes) are lost unless committed to a new image.
* **Isolation**: Containers provide process and filesystem isolation. This means that different containers running on the same host do not interfere with each other or the host system.
* **Mutable**: Although the image is immutable, a container can change its internal state while it runs (e.g., writing files or logs). However, these changes are not reflected in the original image.
* **Lifecycle**: You can create, run, stop, pause, and remove containers. Each container is tied to an image but represents a distinct instance with its own running environment.
* **Example**:
  + If you run an image like nginx, you will create a container that starts an Nginx web server. If you stop or remove the container, the underlying image remains unchanged.

**3. Docker Volume**

* **Definition**: A volume is a persistent storage solution in Docker. It allows you to store data outside the container’s filesystem so that data remains even after the container is stopped or removed.
* **Persistence**: While containers themselves are ephemeral, volumes provide a way to store data persistently. When a container writes to a volume, the data persists on the host machine or a remote storage system, even if the container is deleted.
* **Decoupling Data from Containers**: Volumes are used to decouple application data from the container itself. This is especially useful when you need to persist databases, log files, or user-generated content.
* **Types**:
  + **Anonymous Volumes**: Created by Docker, tied to a specific container, but not named.
  + **Named Volumes**: Explicitly named by the user, making it easier to manage and reuse across containers.
  + **Bind Mounts**: Allow containers to access specific directories on the host’s filesystem.
* **Sharing**: Volumes can be shared across multiple containers. For example, if you’re running a web app and a database in separate containers, both containers can use the same volume to read and write data.
* **Example**:
  + When you run a MySQL container, you would use a volume to store the database data. This way, if you stop or remove the MySQL container, the data in the volume remains intact.

**Tutorial**

**Try existing Docker Image from Docker Hub**

1. Look for ubuntu in <https://hub.docker.com/>

A screenshot of a computer

Description automatically generated

1. Run docker pull ubuntu in the terminal (VS Code)

A screenshot of a computer

Description automatically generated

1. Run docker run -it ubuntu

**What Happens When You Run docker run -it ubuntu?**

1. **Image Retrieval**: Docker checks if the ubuntu image is available locally. If not, it pulls the latest version from Docker Hub.
2. **Container Creation**: Docker creates a new container based on the Ubuntu image.
3. **Interactive Terminal**: The -it flag provides you with an interactive terminal (a shell) inside the newly created Ubuntu container. You are now "inside" the container and can execute shell commands, just as if you were using a regular Ubuntu machine.
4. **Run Commands**: Once inside the container, you can run typical Linux commands, install software, or make modifications.

A screenshot of a computer

Description automatically generated

1. Play around with the ubuntu machine (from the image)

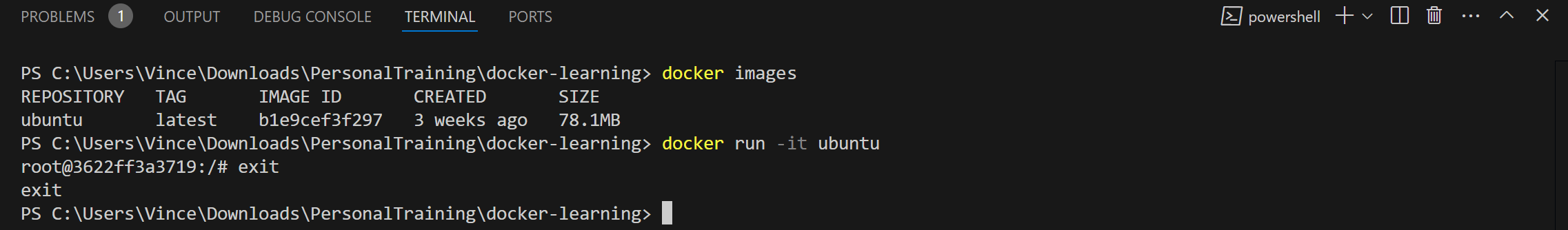
A black screen with blue text

Description automatically generated

A black screen with white text

Description automatically generated

1. Use exit command or trash icon to exit the image



A screenshot of a computer

Description automatically generated

**Create First Image (Hello Docker Demo)**

1. ada