# Docker

Great! Let's start by breaking down the fundamental concepts and then proceed to a hands-on Docker installation and container setup on an EC2 instance.

**What is Docker?**

Docker is a platform that simplifies the process of building, running, managing, and distributing applications in containers. It ensures that your application behaves the same, regardless of where it's run: your laptop, a cloud server, or a CI/CD pipeline.

**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 use the host system’s kernel, which makes them much lighter and faster than traditional virtual machines (VMs).

**Containers vs. Virtual Machines**

**Virtual Machines**

* **Isolation**: Each VM runs a full OS.
* **Resources**: VMs are heavier since they require their own OS.
* **Boot Time**: Slower to start because of the full OS boot.
* **Portability**: VMs are portable but large and cumbersome to transfer.

**Containers**

* **Isolation**: Containers share the host OS kernel.
* **Resources**: Containers are lightweight and use fewer resources.
* **Boot Time**: Very fast startup as they do not need to boot an OS.
* **Portability**: Containers are highly portable and small.

**Docker Components**

1. **Docker Engine**: The runtime that allows you to build and run containers.
2. **Docker Images**: Read-only templates that define a container.
3. **Docker Containers**: Running instances of Docker images.
4. **Docker Hub**: A registry of Docker images where you can find and share containers.

**Setting up Docker on an EC2 Instance**

Let's go through the steps to set up Docker on an EC2 instance:

**Step 1: Launch an EC2 Instance**

1. **Log in to AWS Console**.
2. **Navigate to EC2 Dashboard**.
3. **Launch an Instance**.
   * Choose an Amazon Machine Image (AMI): Select "Ubuntu Server 20.04 LTS".
   * Choose an Instance Type: t2.micro (sufficient for this demo and eligible for free tier).
   * Configure Instance: Default settings.
   * Add Storage: Default settings.
   * Add Tags: Optional.
   * Configure Security Group: Create a new security group that allows SSH (port 22).
   * Review and Launch: Click "Launch", then choose or create a new key pair.

**Step 2: Connect to Your EC2 Instance**

Use SSH to connect to your instance. If you're using a local terminal:

sh

Copy code

ssh -i "path\_to\_your\_key\_pair.pem" ubuntu@your\_ec2\_public\_dns

**Step 3: Install Docker on Ubuntu**

Once connected to your EC2 instance, install Docker by following these steps:

1. **Update your package index**:

sh

Copy code

sudo apt-get update

1. **Install required packages**:

sh

Copy code

sudo apt-get install apt-transport-https ca-certificates curl software-properties-common

1. **Add Docker’s official GPG key**:

sh

Copy code

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

1. **Add Docker’s official APT repository**:

sh

Copy code

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"

1. **Update your package index again**:

sh

Copy code

sudo apt-get update

1. **Install Docker**:

sh

Copy code

sudo apt-get install docker-ce

1. **Start and enable Docker**:

sh

Copy code

sudo systemctl start docker

sudo systemctl enable docker

1. **Verify Docker installation**:

sh

Copy code

sudo docker --version

**Step 4: Run Your First Container**

1. **Run a simple hello-world container**:

sh

Copy code

sudo docker run hello-world

This command downloads the hello-world image from Docker Hub (if not already available locally) and runs it in a container.

1. **List running containers**:

sh

Copy code

sudo docker ps

1. **List all containers (running and stopped)**:

sh

Copy code

sudo docker ps -a

**Conclusion**

Docker is a powerful tool that simplifies the process of application deployment by using containers. Containers are lightweight and efficient compared to traditional VMs, making them ideal for modern cloud-native applications.

By setting up Docker on an EC2 instance, you can start experimenting with containerization and experience its benefits firsthand. You can further explore Docker features like networking, volumes, and Docker Compose for orchestrating multi-container applications.

# Up Docker

**Setting Up Docker on an EC2 Instance Running Ubuntu 18.04**

Let's go through the detailed steps to launch an EC2 instance, install Docker, and verify the setup. Follow these instructions to get Docker up and running on your EC2 instance.

**Step 1: Launch an EC2 Instance**

1. **Log in to AWS Management Console**.
2. **Navigate to EC2 Dashboard**.
3. **Launch an Instance**:
   * **Choose an Amazon Machine Image (AMI)**: Select "Ubuntu Server 18.04 LTS".
   * **Choose an Instance Type**: t2.micro (eligible for free tier).
   * **Configure Instance Details**: Accept the default settings.
   * **Add Storage**: Accept the default storage settings.
   * **Add Tags**: Add a tag for easier identification, e.g., Name = docker-engine.
   * **Configure Security Group**:
     + Create a new security group named docker-security-group.
     + Add the following rules:
       - **SSH**: Port 22, Source: My IP
       - **All Traffic**: Source: My IP (for accessing Docker containers on different ports).
   * **Review and Launch**: Click "Launch", then select or create a new key pair for SSH access. Download the key pair file (.pem) and keep it safe.

**Step 2: Connect to Your EC2 Instance**

Use SSH to connect to your EC2 instance. Replace path\_to\_your\_key\_pair.pem with the path to your downloaded key pair file, and your\_ec2\_public\_dns with the public DNS of your EC2 instance.

sh

Copy code

ssh -i "path\_to\_your\_key\_pair.pem" ubuntu@your\_ec2\_public\_dns

**Step 3: Install Docker on Ubuntu 18.04**

Once connected to your EC2 instance, follow these steps to install Docker:

1. **Update your package index**:

sh

Copy code

sudo apt-get update

1. **Install required packages**:

sh

Copy code

sudo apt-get install apt-transport-https ca-certificates curl software-properties-common

1. **Add Docker’s official GPG key**:

sh

Copy code

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

1. **Add Docker’s official APT repository**:

sh

Copy code

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"

1. **Update your package index again**:

sh

Copy code

sudo apt-get update

1. **Install Docker**:

sh

Copy code

sudo apt-get install docker-ce

1. **Start and enable Docker**:

sh

Copy code

sudo systemctl start docker

sudo systemctl enable docker

1. **Verify Docker installation**:

sh

Copy code

sudo docker --version

**Step 4: Manage Docker as a Non-Root User**

By default, only the root user can run Docker commands. To enable your user (ubuntu) to run Docker commands without sudo, follow these steps:

1. **Add your user to the docker group**:

sh

Copy code

sudo usermod -aG docker ${USER}

1. **Apply the new group membership**:

sh

Copy code

su - ${USER}

1. **Verify that your user can run Docker commands**:

sh

Copy code

docker images

If you see a list of images (or an empty list if no images are downloaded yet), the setup is successful.

**Step 5: Run Your First Docker Container**

1. **Run the hello-world container to test your setup**:

sh

Copy code

docker run hello-world

This command will download the hello-world image from Docker Hub (if it's not already downloaded) and run it in a container. The container prints a "Hello from Docker!" message and then exits.

1. **Check running containers**:

sh

Copy code

docker ps

1. **Check all containers (running and stopped)**:

sh

Copy code

docker ps -a

You should see the hello-world container listed with a status of Exited.

**Conclusion**

You've successfully launched an EC2 instance, installed Docker, and verified the installation by running a test container. This setup allows you to start experimenting with Docker containers and further explore Docker's capabilities, such as networking, volumes, and orchestration with Docker Compose.

It looks like you have a comprehensive understanding of Docker and Docker Hub, including the concepts of images, containers, and registries. Docker Hub is indeed a crucial platform for Docker, as it provides a centralized repository for Docker images. Here are some key points and clarifications from your detailed explanation:

**Docker Hub and Docker Images**

* **Docker Hub**: A public registry to find, share, and manage Docker images. It contains both official images, provided by the creators or maintainers of the software, and community-contributed images.
* **Images**: Pre-configured environments with all dependencies needed to run a specific application. Docker images are composed of multiple layers, each layer representing a set of changes.
* **Official Images**: Verified and maintained by Docker or the software creators. Examples include images for Nginx, Redis, Postgres, Ubuntu, etc.

**Key Commands and Concepts**

* **docker pull [image\_name]**: Fetches an image from Docker Hub.
* **docker run [image\_name]**: Creates and starts a new container from the specified image.
* **docker ps**: Lists all running containers.
* **docker ps -a**: Lists all containers, including those that are stopped.
* **docker exec -it [container\_name] /bin/bash**: Attaches to a running container and opens an interactive shell session.

**Container Management**

* **Creating Containers**: Containers are created using the docker run command. They run processes in isolation from the host system.
* **Port Mapping**: Containers run in a private network. To access container services, you map container ports to host ports using the -p flag.
* **Volumes**: For persistent storage, you can attach volumes to containers, allowing data to persist across container restarts.

**Storage and Layers**

* **Layered File System**: Docker images use a layered file system. Each command in the Dockerfile creates a new layer. These layers are read-only, and containers add a thin read-write layer on top.
* **Image and Container Relationship**: Containers are instances of images. Images can be thought of as blueprints, and containers as the running instances.

**Docker Registry Alternatives**

* **GCR (Google Container Registry)**, **Amazon ECR (Elastic Container Registry)**: Cloud-based registries provided by Google and Amazon respectively.
* **Private Registries**: Tools like Nexus, JFrog Artifactory, and Docker Trusted Registry (DTR) allow hosting private Docker registries.

**Practical Examples**

* **Running an Nginx Container**:

bash

Copy code

docker run -d -p 8080:80 --name mynginx nginx

This runs Nginx in detached mode, maps port 80 of the container to port 8080 of the host, and names the container "mynginx".

* **Exploring Container Filesystem**:

bash

Copy code

docker exec -it mynginx /bin/bash

This command opens a bash shell inside the running Nginx container.

**Clean-up Commands**

* **Stop and Remove Containers**:

bash

Copy code

docker stop [container\_id]

docker rm [container\_id]

* **Remove Images**:

bash

Copy code

docker rmi [image\_id]

**Conclusion**

Docker provides a powerful and flexible way to package, distribute, and run applications in isolated environments. Docker Hub serves as a central hub for finding and sharing container images, making it easier to develop and deploy applications consistently across different environments.

Welcome to the lecture on Docker logs. In this session, we'll delve into how to view and utilize logs from your Docker containers, an essential part of troubleshooting and maintaining your containerized applications.

**Docker Logs Overview**

Docker logs allow you to view the output of commands and processes running within your containers. This is crucial for debugging and monitoring the behavior of your applications.

**Basic Docker Logs Command**

The primary command to view logs in Docker is:

sh

Copy code

docker logs [container\_name\_or\_id]

This command shows the stdout (standard output) and stderr (standard error) streams from the container.

**Practical Example with Nginx**

1. **Pull the Nginx Image**:

sh

Copy code

docker pull nginx

This command fetches the latest Nginx image from Docker Hub.

1. **Run the Nginx Container**:

sh

Copy code

docker run -d -P nginx

* + -d: Runs the container in detached mode (in the background).
  + -P: Automatically maps container ports to the host ports.

1. **Check Running Containers**:

sh

Copy code

docker ps

This lists all currently running containers.

1. **View Logs of the Running Nginx Container**:

sh

Copy code

docker logs [container\_name\_or\_id]

This command displays the logs, showing the output from the entry point script and Nginx daemon command.

**Understanding Logs and Metadata**

1. **Inspect the Image**:

sh

Copy code

docker inspect nginx

This command provides detailed metadata about the image in JSON format, including the Cmd and Entrypoint fields, which indicate the initial commands run by the container.

1. **Foreground Execution without Detached Mode**:

sh

Copy code

docker run -P nginx

Running the container without -d will show the process output directly in your terminal. Terminating it with Ctrl+C will stop the container.

**Troubleshooting with Logs**

1. **Run a MySQL Container**:

sh

Copy code

docker run -d mysql:5.7

If the container exits immediately, use:

sh

Copy code

docker ps -a

to list all containers, including those that have stopped.

1. **Check Logs for Errors**:

sh

Copy code

docker logs [container\_name\_or\_id]

The logs may reveal errors such as missing environment variables. For example, MySQL requires a root password to be set:

sh

Copy code

ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)

1. **Run the MySQL Container with Environment Variables**:

sh

Copy code

docker run -d -e MYSQL\_ROOT\_PASSWORD=my-secret-pw mysql:5.7

This sets the required environment variable and starts the container successfully.

**Viewing Logs of Running Containers**

You can always check the logs of a running container to monitor its behavior:

sh

Copy code

docker logs [running\_container\_name\_or\_id]

This will show the output from the process that started the container.

**Clean-Up**

Before moving on to the next lecture, ensure you clean up your environment:

1. **Remove All Containers**:

sh

Copy code

docker rm $(docker ps -a -q)

This command removes all stopped containers.

1. **Remove All Images**:

sh

Copy code

docker rmi $(docker images -q)

This command removes all Docker images.

By mastering Docker logs, you can effectively troubleshoot and manage your Docker containers, ensuring that your applications run smoothly. In the next lecture, we'll delve deeper into Docker volumes and how to manage persistent storage for your containers.

The concept of containers is rooted in their disposable and ephemeral nature. This design philosophy ensures that containers are lightweight and can be quickly replaced without worrying about the underlying state. Here's a detailed breakdown of how this works and how to manage persistent data in containers, especially for stateful applications like MySQL:

**Volatile Nature of Containers**

* **Disposability**: Containers are designed to be disposable. This means that when you need to make changes, you don't modify a running container directly. Instead, you update the container image and recreate the container from this new image.
* **Ephemeral Data**: Any data stored inside the container is lost when the container is removed. This is particularly important to understand when dealing with applications that require data persistence, like databases.

**Managing Persistent Data**

To handle persistent data, Docker provides two primary mechanisms: **Volumes** and **Bind Mounts**.

**1. Volumes**

Volumes are the preferred mechanism for persisting data generated and used by Docker containers.

* **Docker Managed**: Volumes are created and managed by Docker. They are stored in a specific location on the host (/var/lib/docker/volumes on Linux).
* **Isolation from Host**: Volumes provide better isolation from the host filesystem.
* **Example**:

bash

Copy code

docker volume create mydbdata

docker run -d --name mydbcontainer -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -p 3306:3306 -v mydbdata:/var/lib/mysql mysql:5.7

**2. Bind Mounts**

Bind mounts allow you to mount a directory from the host machine into a container.

* **Direct Mapping**: You specify the exact path on the host machine to mount into the container.
* **Dynamic Development**: Useful for scenarios where you need to frequently update files on the host and see those changes in the container immediately.
* **Example**:

bash

Copy code

mkdir -p ~/mydbdata

docker run -d --name mydbcontainer -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -p 3306:3306 -v ~/mydbdata:/var/lib/mysql mysql:5.7

**Using MySQL with Docker**

To run a MySQL container with persistent storage, you can choose between a volume or a bind mount. Here's a step-by-step guide:

1. **Pull the MySQL Image**:

bash

Copy code

docker pull mysql:5.7

1. **Inspect the Image** (Optional):

bash

Copy code

docker inspect mysql:5.7

1. **Run MySQL with a Volume**:

bash

Copy code

docker volume create mydbdata

docker run -d --name mydbcontainer -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -p 3306:3306 -v mydbdata:/var/lib/mysql mysql:5.7

1. **Run MySQL with a Bind Mount**:

bash

Copy code

mkdir -p ~/mydbdata

docker run -d --name mydbcontainer -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -p 3306:3306 -v ~/mydbdata:/var/lib/mysql mysql:5.7

**Inspecting and Managing Containers**

* **Inspect Container**:

bash

Copy code

docker inspect mydbcontainer

This command provides detailed information about the container, including the volume mounts, environment variables, and more.

* **Access Container Logs**:

bash

Copy code

docker logs mydbcontainer

* **Cleanup**: It's important to clean up unused containers and volumes to free up resources.

bash

Copy code

docker stop mydbcontainer

docker rm mydbcontainer

docker volume rm mydbdata

docker rmi mysql:5.7

**Summary**

By using Docker volumes and bind mounts, you can ensure that your data persists even if the container is replaced. This is crucial for stateful applications like databases, where losing data is not an option. Volumes are managed by Docker and provide a more isolated and controlled environment, while bind mounts offer more flexibility for development purposes.