Docker Instalation

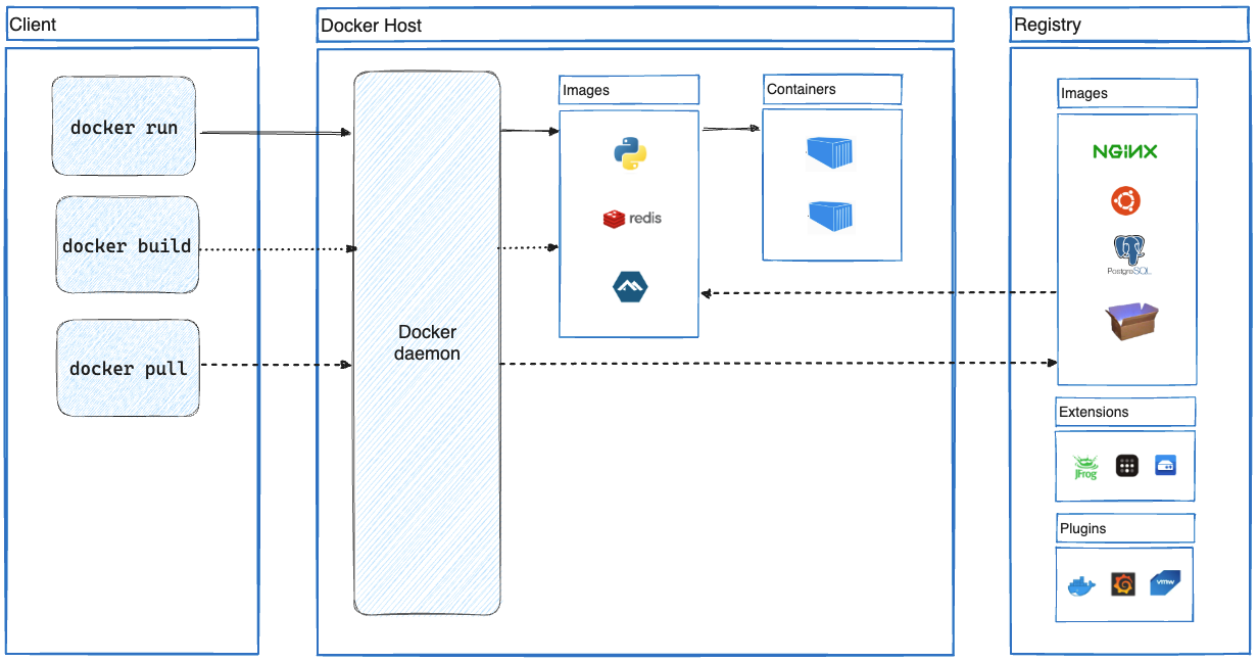
A. What Is Docker?

Docker is a software platform that allows you to build, test, and deploy applications quickly. Docker packages software into standardized units called containers that have everything the software needs to run including libraries, system tools, code, and runtime. Using Docker, you can quickly deploy and scale applications into any environment and know your code will run.

Running Docker on AWS provides developers and admins a highly reliable, low-cost way to build, ship, and run distributed applications at any scale.

B. How Docker works

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, 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



- **The Docker daemon**

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**

  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. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

- **Docker Desktop**

  Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

- **Docker registries**

  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.

  When you use the `docker pull` or `docker run` commands, Docker pulls the required images from your configured registry. When you use the `docker push` command, Docker pushes your image to your configured registry.

- **Docker objects**

  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. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

  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.

C. Docker Install

Prerequisites

- A system running Ubuntu 22.04.
- A user account with sudo privileges.

Step 1: Update your system

```
$ sudo apt update
$ sudo apt upgrade
```

Step 2: Install required dependencies

To install Docker on Ubuntu, we need to install some necessary dependencies first. Run the following command to install them:

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

Step 3: Add Docker repository

Next, we will add the official Docker repository to our system. First, add the GPG key for the Docker repository:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg -
  -dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

Then, add the Docker repository to your system with the following command:

```
$ echo "deb [arch=$(dpkg --print-architecture) signed-
  by=/usr/share/keyrings/docker-archive-keyring.gpg]
  https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
  | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Step 4: Install Docker

Now that we have added the Docker repository, update your package index:

```
$ sudo apt update
```

Then, install Docker with the following command:

```
$ sudo apt install docker-ce docker-ce-cli containerd.io
```

Step 5: Verify Docker installation

To verify that Docker has been installed successfully, run the following command:

```
$ sudo docker --version
```

Step 6: Manage Docker service

Docker should now be installed and running as a background service. To check the status of the Docker service, run:

```
$ sudo systemctl status docker
```

If the Docker service is not running, you can start it with:

```
$ sudo systemctl start docker
```

To enable the Docker service to start automatically at boot, run:

```
$ sudo systemctl enable docker
```

Step 7: Running Docker without sudo (optional)

By default, Docker requires sudo privileges to run. If you want to use Docker without typing 'sudo' every time, add your user to the 'docker' group with the following command:

```
$ sudo usermod -aG docker $USER sudo systemctl enable docker
```

After running this command, log out and log back in for the changes to take effect.

D. Docker Command

Using docker consists of passing it a chain of options and commands followed by arguments. The syntax takes this form:

```
$ docker [option] [command] [arguments]
```

To view all available subcommands, type:

```
$ docker
```

To view the options available to a specific command, type:

```
$ docker docker-subcommand --help
```

To view system-wide information about Docker, use:

```
$ docker info
```

Step 1 : Working with Docker

Docker containers are built from Docker images. By default, Docker pulls these images from Docker Hub, a Docker registry managed by Docker, the company behind the Docker project. Anyone can host their Docker images on Docker Hub, so most applications and Linux distributions you'll need will have images hosted there.

To check whether you can access and download images from Docker Hub, type:

```
$ docker run hello-world
```

The output will indicate that Docker in working correctly:

```
Output:
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest:
  sha256:bfea6278a0a267fad2634554f4f0c6f31981eea41c553fdf5a83e95a41d40
  c38
Status: Downloaded newer image for hello-world:latest
……
```

Docker was initially unable to find the hello-world image locally, so it downloaded the image from Docker Hub, which is the default repository. Once the image downloaded, Docker created a container from the image and the application within the container executed, displaying the message.

You can search for images available on Docker Hub by using the docker command with the search subcommand. For example, to search for the Ubuntu image, type:

```
$ docker search ubuntu
```

Check what is output from the command above?

In the OFFICIAL column, OK indicates an image built and supported by the company behind the project. Once you've identified the image that you would like to use, you can download it to your computer using the pull subcommand.

Execute the following command to download the official ubuntu image to your computer:

```
$ docker pull ubuntu
```

You'll see the following output:

```
Output
Using default tag: latest
latest: Pulling from library/ubuntu
e0b25ef51634: Pull complete
Digest:
  sha256:9101220a875cee98b016668342c489ff0674f247f6ca20dfc91b91c0f2858
  1ae
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

After an image has been downloaded, you can then run a container using the downloaded image with the run subcommand. As you saw with the hello-world example, if an image has not been downloaded when docker is executed with the run subcommand, the Docker client will first download the image, then run a container using it.

To see the images that have been downloaded to your computer, type:

```
$ docker images
```

As you'll see later in this tutorial, images that you use to run containers can be modified and used to generate new images, which may then be uploaded (pushed is the technical term) to Docker Hub or other Docker registries.

Let's look at how to run containers in more detail.

Step 2 : Running a Docker Container

The hello-world container you ran in the previous step is an example of a container that runs and exits after emitting a test message. Containers can be much more useful than that, and they can be interactive. After all, they are similar to virtual machines, only more resource-friendly.

As an example, let's run a container using the latest image of Ubuntu. The combination of the -i and -t switches gives you interactive shell access into the container:

```
$ docker run -it ubuntu
```

Your command prompt should change to reflect the fact that you're now working inside the container and should take this form:

```
Output
root@d9b100f2f636:/#
```

Note the container id in the command prompt. In this example, it is **d9b100f2f636**. You'll need that container ID later to identify the container when you want to remove it.

Now you can run any command inside the container. For example, let's update the package database inside the container. You don't need to prefix any command with **sudo**, because you're operating inside the container as the root user:

```
root@d9b100f2f636:/# apt update
```

Then install any application in it. Let's install Node.js:

```
root@d9b100f2f636:/# apt install nodejs
```

This installs Node.js in the container from the official Ubuntu repository. When the installation finishes, verify that Node.js is installed:

```
root@d9b100f2f636:/# node -v
```

You'll see the version number displayed in your terminal:

```
Output
v12.22.9
```

Any changes you make inside the container only apply to that container.
To exit the container, type exit at the prompt.
Let's look at managing the containers on our system next.

Step 6 : Managing Docker Containers

After using Docker for a while, you'll have many active (running) and inactive containers on your computer. To view the active ones, use:

```
$ docker ps
```

You will see output similar to the following:

```
Output
CONTAINER ID        IMAGE               COMMAND             CREATED
```

In this tutorial, you started two containers; one from the **hello-world** image

and another from the **ubuntu** image. Both containers are no longer running, but they still exist on your system. To view all containers — active and inactive, run **docker ps** with the **-a** switch:

```
$ docker ps -a
```

You'll see output similar to this:

```
Output
CONTAINER ID   IMAGE         COMMAND   CREATED    STATUS PORTS     NAMES
1c08a7a0d0e4   ubuntu        "bash"     About a minute ago
587000e49d53   hello-world   "/hello"   5 minutes ago
```

To view the latest container you created, pass it the -l switch:

```
$ docker ps -l
```

To start a stopped container, use docker start, followed by the container ID or the container's name. Let's start the Ubuntu-based container with the ID of 1c08a7a0d0e4:

```
$ docker start 1c08a7a0d0e4ocker run hello-world
```

The container will start, and you can use **docker ps** to see its status, To stop a running container, use docker stop, followed by the container ID or name. This time, we'll use the name that Docker assigned the container, which is dazzling_taussig:

```
$ docker stop dazzling_taussig
```

Once you've decided you no longer need a container anymore, remove it with the **docker rm** command, again using either the container ID or the name. Use the **docker ps -a** command to find the container ID or name for the container associated with the **hello-world** image and remove it.

```
$ docker rm adoring_kowalevski
```

You can start a new container and give it a name using the **--name** switch. You can also use the **--rm** switch to create a container that removes itself when it's stopped. See the **docker run help** command for more information on these options and others.

Containers can be turned into images which you can use to build new

containers. Let's look at how that works.

Step 7 : Committing Changes in a Container to a Docker Image

When you start up a Docker image, you can create, modify, and delete files just like you can with a virtual machine. The changes that you make will only apply to that container. You can start and stop it, but once you destroy it with the docker rm command, the changes will be lost for good.

This section shows you how to save the state of a container as a new Docker image.

After installing Node.js inside the Ubuntu container, you now have a container running off an image, but the container is different from the image you used to create it. But you might want to reuse this Node.js container as the basis for new images later.

Then commit the changes to a new Docker image instance using the following command.

```
$ docker commit -m "What you did to the image" -a "Author Name"
  container_id repository/new_image_name
```

The -m switch is for the commit message that helps you and others know what changes you made, while -a is used to specify the author. The container_id is the one you noted earlier in the tutorial when you started the interactive Docker session. Unless you created additional repositories on Docker Hub, the repository is usually your Docker Hub username.

For example, for the user sammy, with the container ID of d9b100f2f636, the command would be:

```
$ docker commit -m "added Node.js" -a "sammy" d9b100f2f636
  sammy/ubuntu-nodejs
```

When you commit an image, the new image is saved locally on your computer. Later in this tutorial, you'll learn how to push an image to a Docker registry like Docker Hub so others can access it.

Listing the Docker images again will show the new image, as well as the old one that it was derived from:

```
$ docker images
```

In this example, `ubuntu-nodejs` is the new image, which was derived from the

existing **ubuntu** image from Docker Hub. The size difference reflects the changes that were made. And in this example, the change was that NodeJS was installed. So next time you need to run a container using Ubuntu with NodeJS pre-installed, you can just use the new image.

You can also build Images from a **Dockerfile**, which lets you automate the installation of software in a new image. However, that's outside the scope of this tutorial.

Now let's share the new image with others so they can create containers from it.

Step 8 : Pushing Docker Images to a Docker Repository

The next logical step after creating a new image from an existing image is to share it with a select few of your friends, the whole world on Docker Hub, or other Docker registry that you have access to. To push an image to Docker Hub or any other Docker registry, you must have an account there.

To push your image, first log into Docker Hub.

```
$ docker login -u docker-registry-username
```

You'll be prompted to authenticate using your Docker Hub password. If you specified the correct password, authentication should succeed

Note: If your Docker registry username is different from the local username you used to create the image, you will have to tag your image with your registry username. For the example given in the last step, you would type:

```
$docker tag sammy/ubuntu-nodejs docker-registry-username/ubuntu-nodejs
```

Then you may push your own image using:

```
$ docker push docker-registry-username/docker-image-name
```

To push the ubuntu-nodejs image to the sammy repository, the command would be

```
$ docker push sammy/ubuntu-nodejs
```

Now, check your docker registry, open hub.docker.io.

```
$ docker run hello-world
```

Reference

- https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-22-04#step-5-running-a-docker-container.
- https://tecadmin.net/how-to-install-docker-on-ubuntu-22-04/