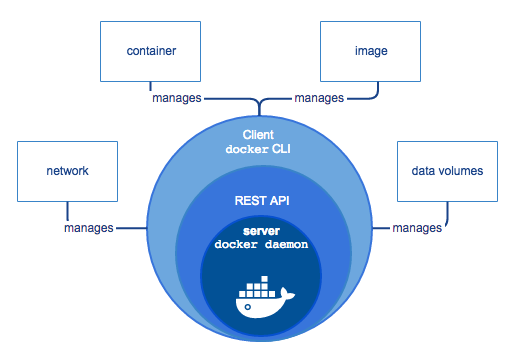
**DOCKER**

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.

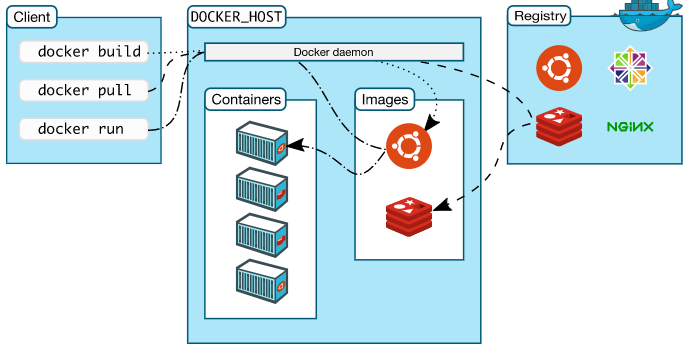
Docker provides the ability to package and run an application in a loosely isolated environment called a container.



Docker Engine is a client-server application with these major components:

* A server which is a type of long-running program called a daemon process (the dockerd command).
* A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
* A command line interface (CLI) client (the docker command).

Docker containers can run on a developer’s local laptop, on physical or virtual machines in a data center, on cloud providers, or in a mixture of environments.



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 (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.

A Docker registry stores Docker images. **Docker Hub** is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry.

**Docker objects:**

**IMAGES**

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.

To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

**CONTAINERS**

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. By default, a container is relatively well isolated from other containers and its host machine. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

**SERVICES**

Services allow you to scale containers across multiple Docker daemons, which all work together as a swarm with multiple managers and workers. Each member of a swarm is a Docker daemon, and the daemons all communicate using the Docker API.

**Best Practices:**

* Use small size base img. Like basing your image on the official openjdk image, rather than starting with a generic ubuntu image and installing openjdk as part of the Dockerfile.
* Use multistage builds. For instance, you can use the maven image to build your Java application, then reset to the tomcat image and copy the Java artifacts into the correct location to deploy your app, all in the same Dockerfile. This means that your final image doesn’t include all of the libraries and dependencies pulled in by the build, but only the artifacts and the environment needed to run them.
* Try to reduce the number of layers in your image by minimizing the number of separate RUN commands in your Dockerfile. You can do this by consolidating multiple commands into a single RUN line.
* RUN apt-get -y update
* RUN apt-get install -y python

Instead use

* RUN apt-get -y update && apt-get install -y python
* If you have multiple images with a lot in common, consider creating your own base image with the shared components, and basing your unique images on that. Docker only needs to load the common layers once, and they are cached.