



# Docker Fundamentals

---

.NET

*Docker provides the ability to run one or more applications, in an isolated environment called a Container, without a hypervisor, on a single computer.*

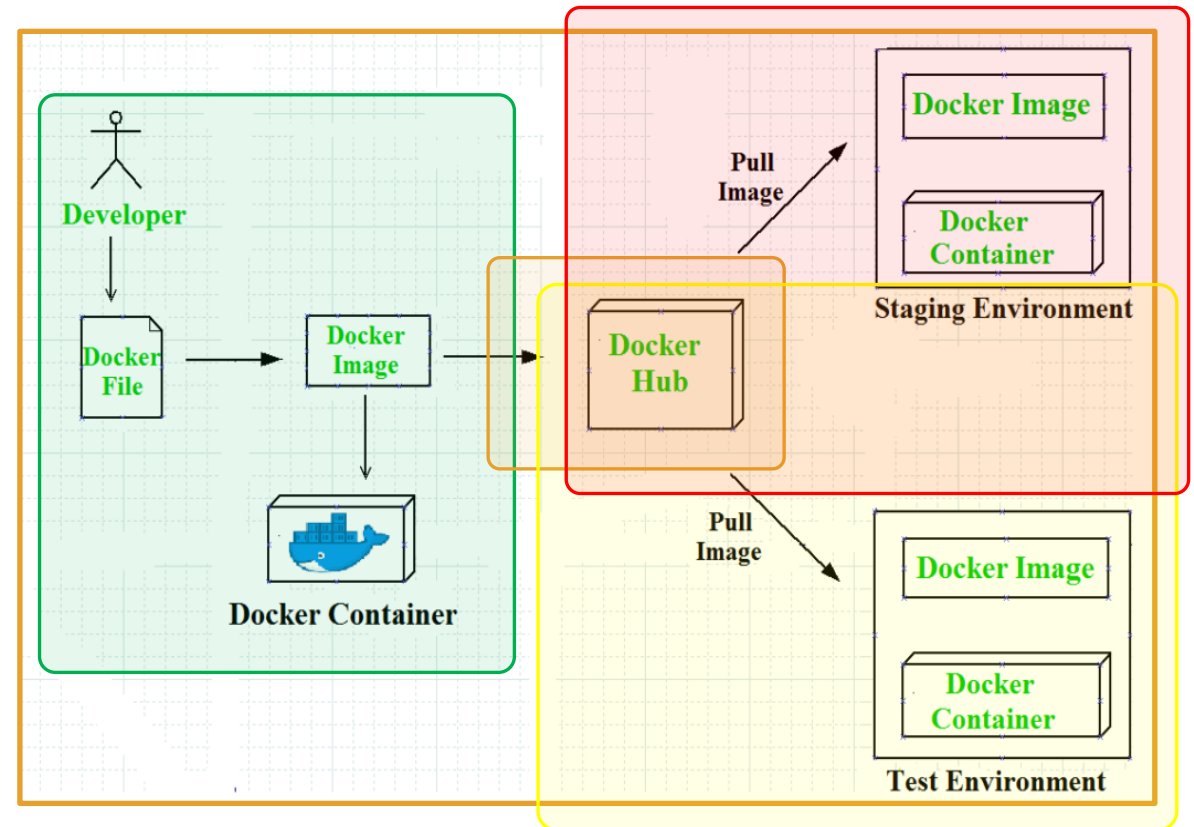
[HTTPS://DOCS.DOCKER.COM/ENGINE/DOCKER-OVERVIEW/](https://docs.docker.com/engine/docker-overview/)

# Docker – Purpose

<https://docs.docker.com/engine/docker-overview/#what-can-i-use-docker-for>

**Docker** allows developers to continue working, in standardized environments, while using **Containers**. **Containers** provide everything needed to run applications and services and are perfect for **CI/CD** workflows.

1. Developers write code locally in a **development environment** and share their work using **Docker images**.
2. They can use Docker to push their applications into a test environment to execute automated and manual tests.
3. Bugs can be fixed in the **development environment** and redeployed to the **test environment** for re-testing and validation.
4. When testing is complete, developers push the updated image to the **production environment**.



# Docker – Benefits

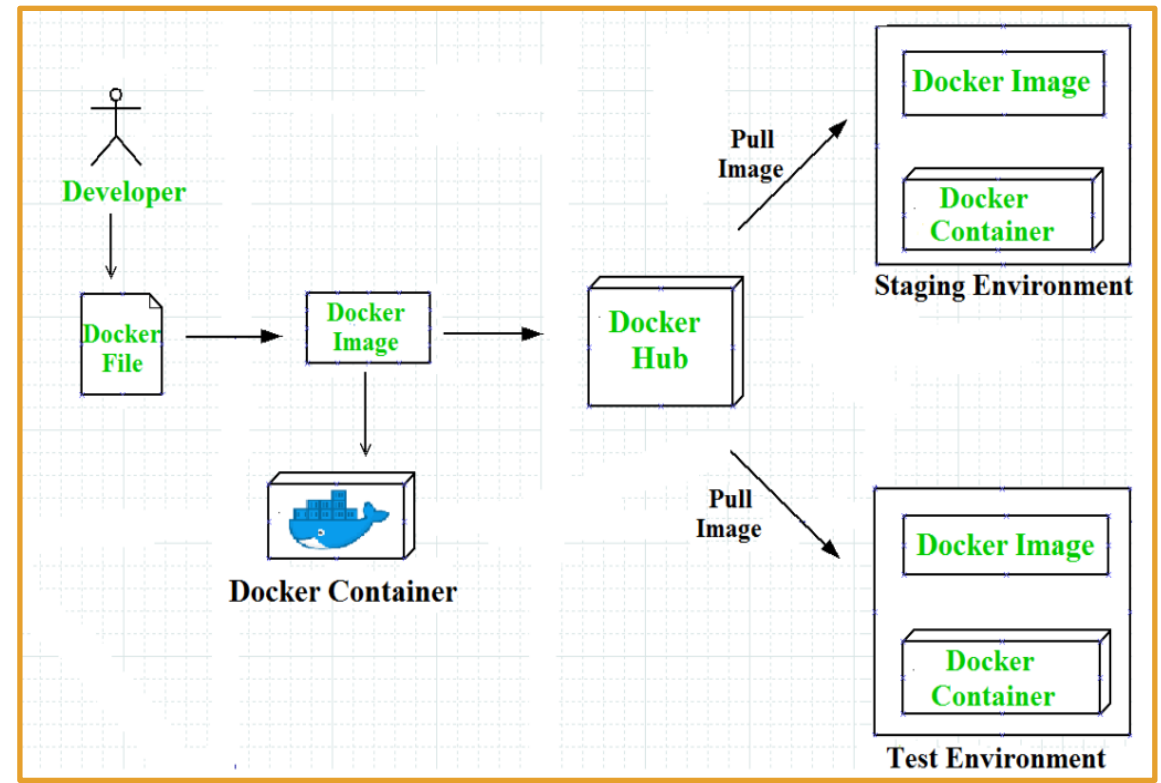
<https://docs.docker.com/engine/docker-overview/#what-can-i-use-docker-for>

## Responsive deployment and scaling:

- **Docker Images** are portable. They can run on a local laptop, on physical and virtual machines, in a data center, or on cloud providers.
- You can scale up or tear down applications and services as needed.

## Run more workloads on the same hardware:

- Docker is lightweight and fast.
- Docker is NOT a Virtual Machine.
  - a *virtual machine* (VM) runs a full-blown “guest” operating system with *virtual* access to host resources through a hypervisor. VMs incur a lot of overhead.



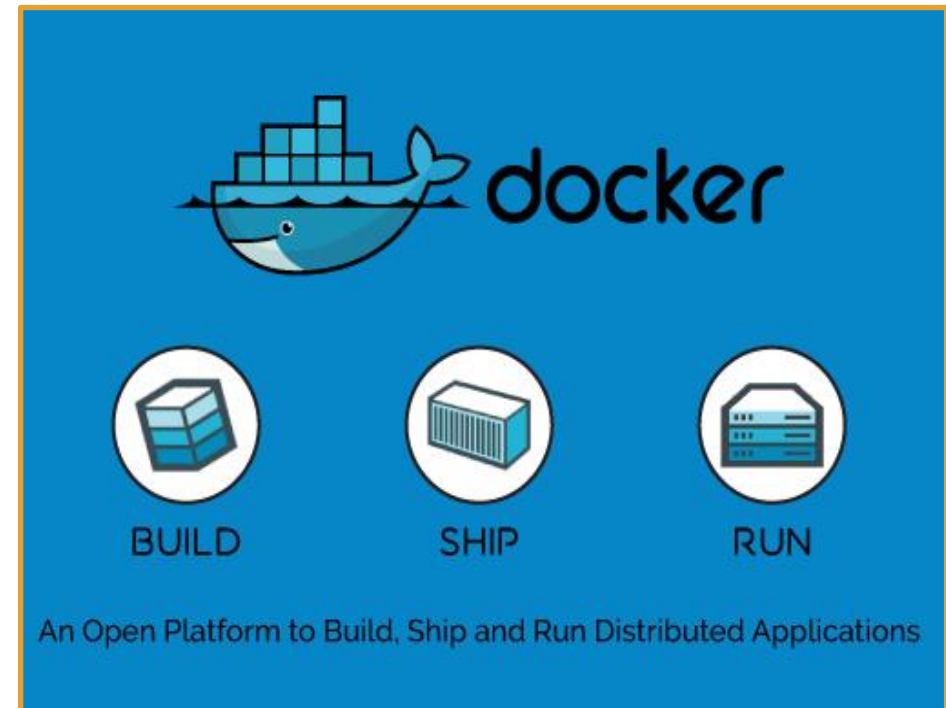
# The Docker Platform

<https://www.docker.com/resources/what-container>

---

**Docker** provides a platform to manage the entire lifecycle of **containers**:

1. You develop an application and its supporting components using a **Container**.
2. The **Container** becomes the unit for distributing and testing your application.
3. Deploy your application into your production environment as a **Container**.
4. This process is identical for all production environments:
  - in a local data center,
  - a cloud provider,
  - or a hybrid.



# Docker Install

---

## Windows:

Windows Home has additional requirement.

<https://docs.docker.com/docker-for-windows/install/>

## Mac:

<https://docs.docker.com/docker-for-mac/install/>



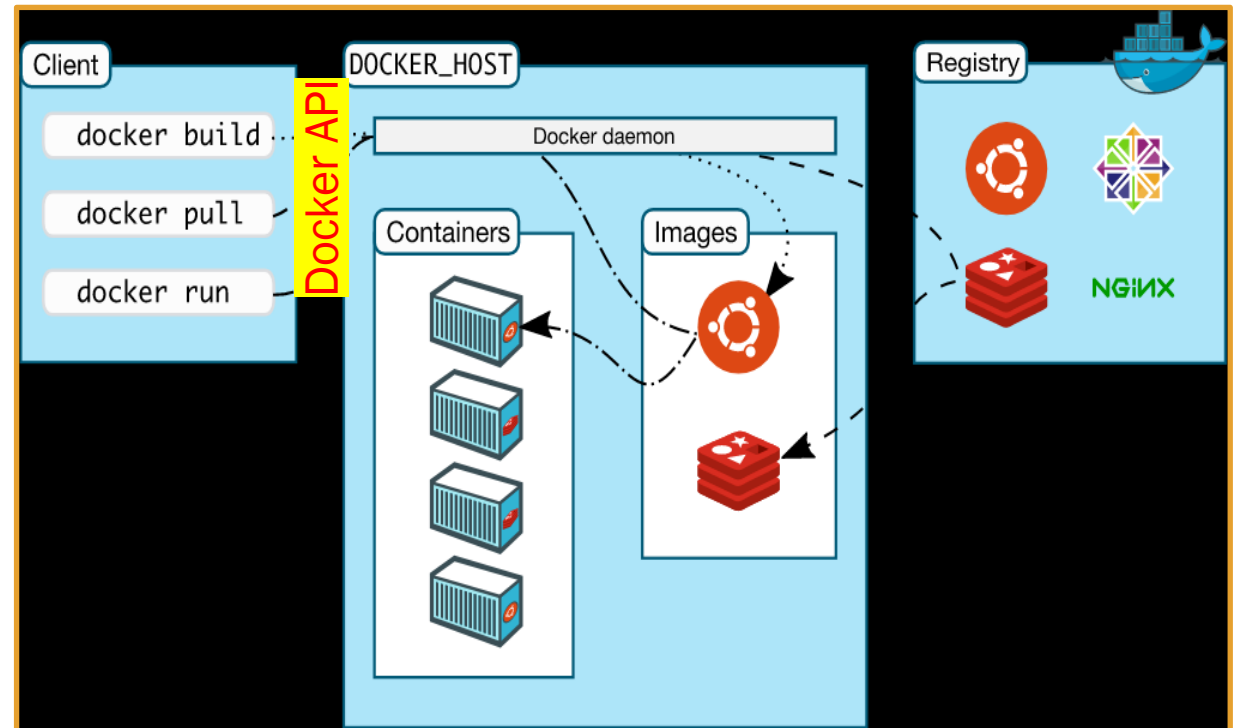
# Docker Architecture

<https://docs.docker.com/engine/docker-overview/#docker-architecture>

Docker uses a *client-server architecture*. The *Docker client* talks to the *Docker daemon* (server, *dockerd*), which builds, runs, and distributes *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*.

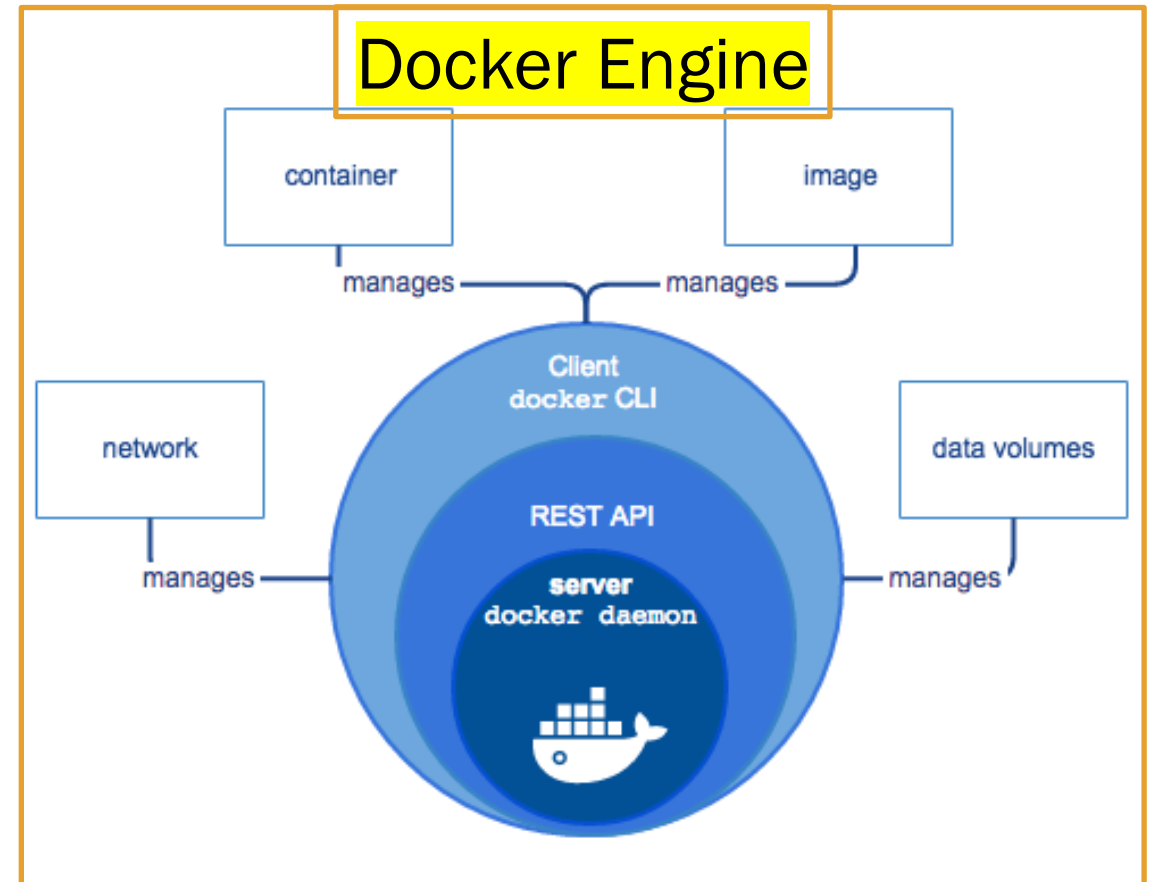


# Docker Engine

<https://docs.docker.com/engine/docker-overview/#docker-engine>

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

1. A server, which is a long-running program called a **daemon**. The daemon is also known as 'dockerd';
2. A **REST API** which specifies interfaces used to talk to the daemon and instruct it what to do. You interact with the **daemon** with the **docker** command.
3. A command line interface (**CLI**) client (**docker <command>**).



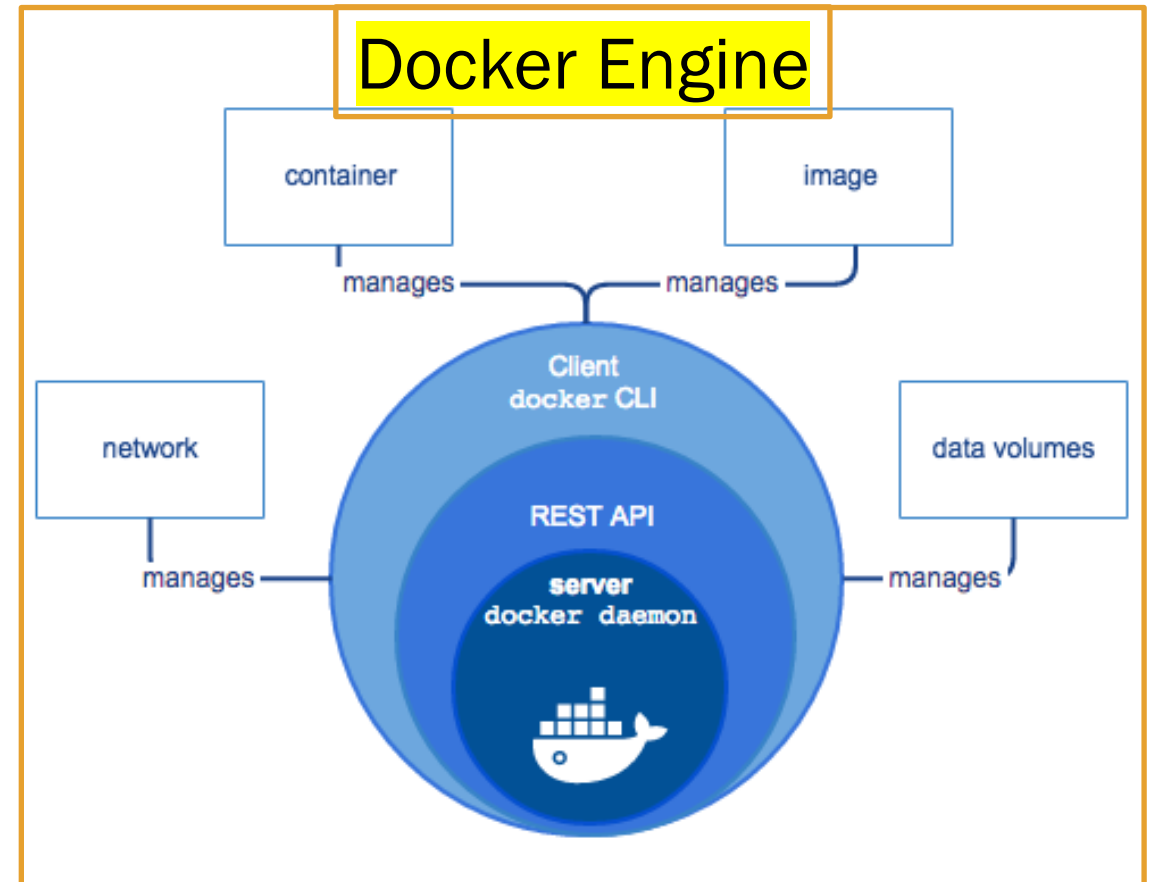


# Docker Engine

<https://docs.docker.com/engine/docker-overview/#docker-engine>

The Docker **CLI** uses the **Docker REST API** to interact with the **Docker daemon** through scripting and/or **CLI** commands. Many **Docker** applications use the underlying **API** and **CLI**.

The **daemon** creates and manages Docker objects, such as **images**, **Containers**, networks, and **volumes**.



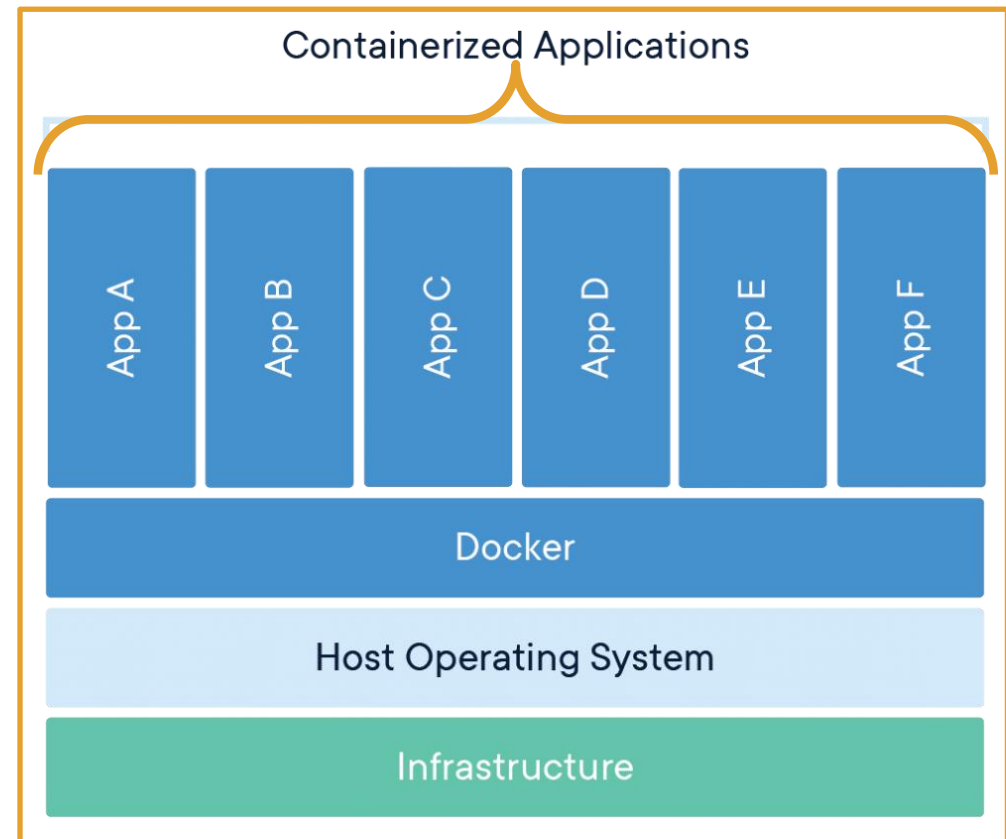
# Docker Image and Container

<https://www.docker.com/resources/what-container>  
<https://docs.docker.com/get-started/>

A **Docker Image** is a standalone executable package that includes everything needed to run an application: code, runtime, system tools and libraries, and settings.

A **Docker Container** is created from a **Docker Image** at the moment you build the container (runtime). **Containers** run identically on Linux or PC machines.

A **Container** is a running process with encapsulation features applied to it to keep it isolated from the host. A **Container** interacts with its own private filesystem.



## Docker Client

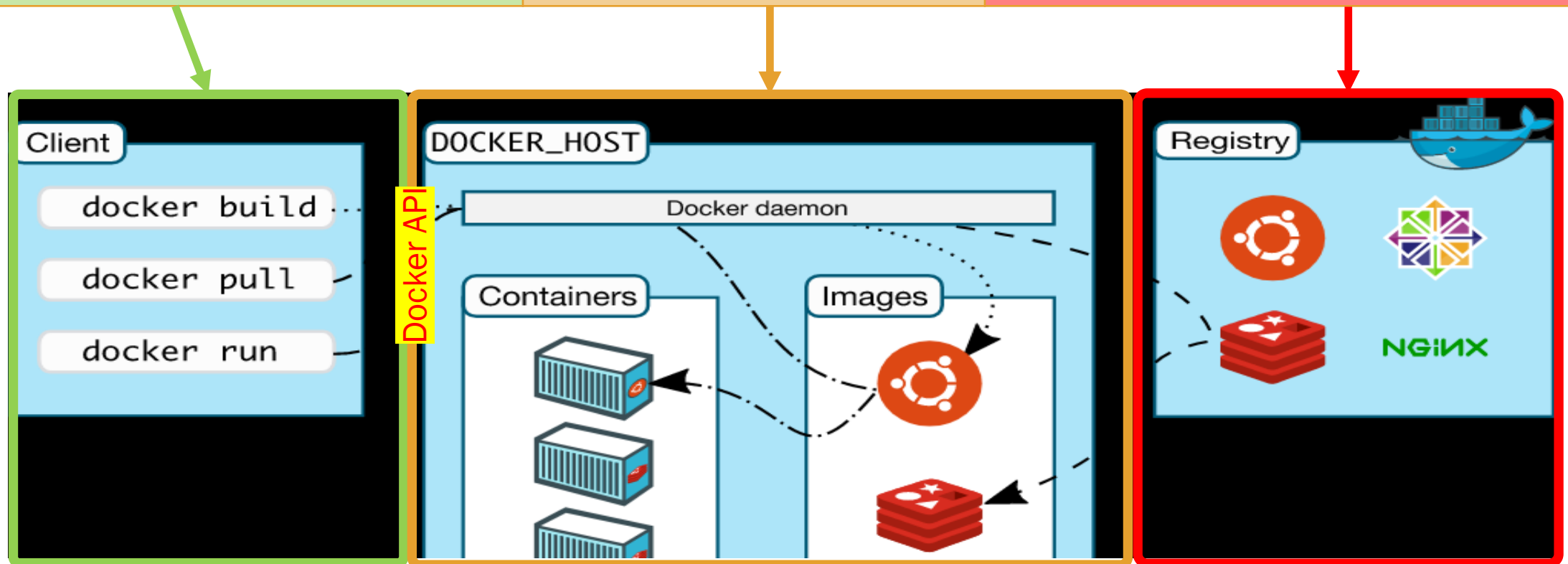
The *Docker client* is the primary way that most Docker users interact with Docker. With **docker run**, the client sends commands to **dockerd**, which carries them out. **docker** specifies the *Docker API*.

## Docker daemon

The *Docker daemon* (dockerd) listens for *Docker API* requests and manages Docker objects such as *images*, *networks*, *containers*, and *volumes*.

## Docker registries

A Docker registry stores Docker images. [hub.docker.com](https://hub.docker.com) is a public registry. With the **docker pull** or **docker run** commands, images can be pulled from a DockerHub registry. When you use **docker push**, an **image** is pushed to the configured registry.



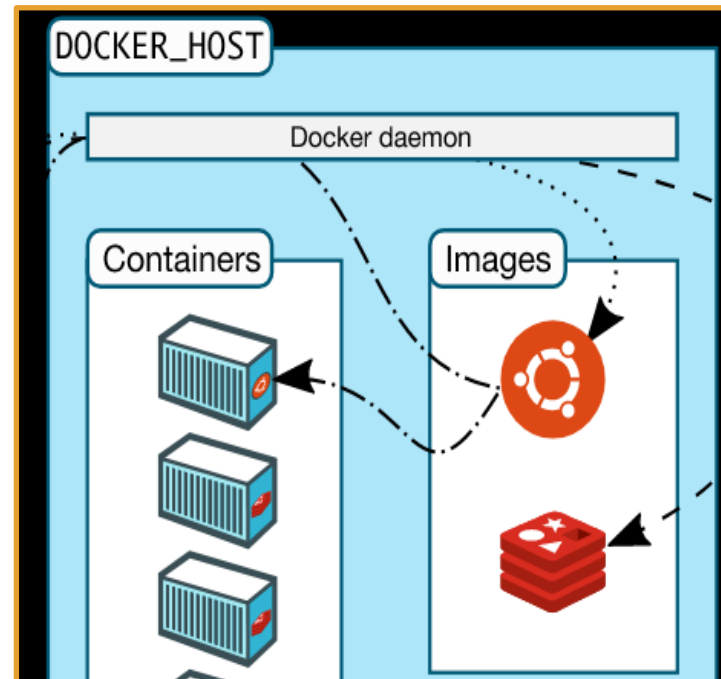
# Docker Image and Container

<https://docs.docker.com/engine/docker-overview/#docker-architecture>

A **container** is a runnable instance of an **image**. You can create, start, stop, move, or delete a **Container** using the **Docker API**.

A **Container** is defined by an **image** as well as any configuration options you provide to it when you create or start it.

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.



An **image** is a read-only template with instructions for creating a **Docker container**.

An **image** often is based on another **image**. An **image** could be based on another **image**, then install a different web server, an application, and the configuration details needed to make the application run.

To build an **image**, create a **Dockerfile** which defines the steps to create an **image** and run it.

When you change a **Dockerfile** and rebuild the **image**, only those layers which have changed are rebuilt.

# List of Basic Docker commands

Command	Purpose
<a href="#">docker start &lt;containername&gt;</a>	Start a container.
<a href="#">docker stop &lt;containername&gt;</a>	Stop a running container
<a href="#">docker container &lt;command&gt;</a>	Manage containers
<a href="#">docker image ls</a>	list the images on your machine.
<code>docker ps -a</code>	Lists all containers, running or stopped
<a href="#">docker ps</a>	Lists the running containers
<a href="#">docker run</a> <containername>	Re-run a container
<a href="#">docker build -t myimage .</a>	Build an image to be called myimage) from a Dockerfile at '.' (in the same directory).
<a href="#">docker rm &lt;containername&gt;</a>	Delete a stopped container
<a href="#">docker push username/reponame:&lt;tagname&gt;</a>	Push an image to a repo in the Docker Registry
<a href="#">docker create myimage</a>	Create a Container from an image, but don't start it.
<a href="#">docker attach &lt;containername&gt;</a>	Connect to a running container

# Docker – Setup and Test a Container

<https://docs.docker.com/get-started/>

---

1. Download Docker Desktop.
2. Go to Docker.com and create an account
3. Run `docker --version` in the Command Line to see what Docker version you have.
4. Run `docker run hello-world` to test that docker is running correctly. You don't have this image so it will get downloaded automatically and run.
5. Run `docker image ls` to list the downloaded hello-world image on your machine.
6. Run `docker ps -a` to see the container created from the `hello-world` *image*.
7. Do the Docker tutorial [here](#).
8. Then complete the [Getting Started Walk-through for Developers](#) tutorial.



# Docker in action

The following command runs an ubuntu container, attaches interactively to your local command-line session, and runs the `/bin/bash` script.

```
$ docker run -i -t ubuntu /bin/bash
```

The following happens (assuming default registry configuration):

1. If you do not have the **ubuntu** image locally, Docker pulls it from your configured registry, as though you had run **docker pull ubuntu** manually.
2. Docker creates a new container, as though you had run a **docker container create** command manually.
3. Docker allocates a read-write filesystem to the container, as its final layer.
  - This allows a running container to create or modify files and directories in its local filesystem.
4. Docker creates a network interface to connect the container to the default network,
  - because you did not specify any networking options.
  - This includes assigning an IP address to the container.
  - By default, containers can connect to external networks using the host machine's network connection.
5. Docker starts the container and executes **/bin/bash**.
  - Because the container is running interactively and attached to your terminal (due to the **-i** and **-t** flags), you can provide input using your keyboard while the output is logged to your terminal.
6. When you type **exit** to terminate the **/bin/bash** command, the container stops but is not removed.
  - You can start it again or remove it.

# Assignment

---

After finishing the tutorials, redo this tutorial with modifications.

The modifications are:

- you will run it on your local machine and
- make the necessary changes to get it to work.

You don't have the .html and .png files that are **COPY**d into the container in the Dockerfile. There are certainly other things missing and it is your task to find out what must be changed and change them.

The first person who finishes and presents to the class tomorrow morning will get 5 points added to their quiz on Monday.

You must be the first to contact me with a working site and explanation of exactly what had to be changed.