**Docker overview:**

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. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

**The Docker platform:**

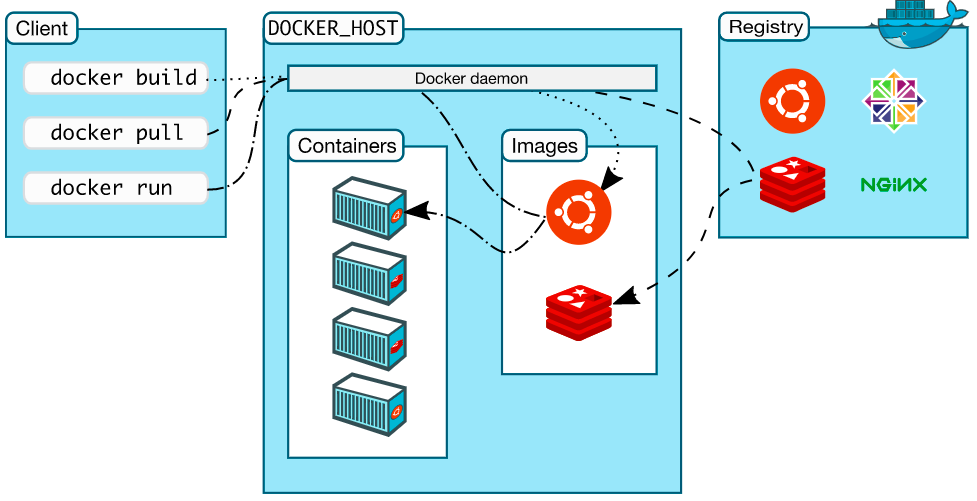
Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host.

**Why Docker:**

* Fast, consistent delivery of your applications
* Responsive deployment and scaling
* Running more workloads on the same hardware

**Docker architecture:**

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 registries:**

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. You can even run your own private registry.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

**Docker objects**

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those 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. 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.

You might create your own images or you might only use those created by others and published in a registry. 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. 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.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container’s network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Example docker run command

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

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

Pull:

docker pull Ubuntu

docker images

docker rmi 22 #first 2 letter

**Namespaces:**

Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources. The feature works by having the same namespace for a set of resources and processes, but those namespaces refer to distinct resources. Resources may exist in multiple spaces.

**Process ID (pid)**

The PID namespace provides processes with an independent set of process IDs (PIDs) from other namespaces. PID namespaces are nested,

meaning when a new process is created it will have a PID for each namespace from its current namespace up to the initial PID namespace.

Hence the initial PID namespace is able to see all processes, albeit with different PIDs than other namespaces will see processes with.

The first process created in a PID namespace is assigned the process id number 1 and receives most of the same special treatment as

the normal init process, most notably that orphaned processes within the namespace are attached to it. This also means that the

termination of this PID 1 process will immediately terminate all processes in its PID namespace and any descendants.

**Network (net):**

Network namespaces virtualize the [network stack](https://en.wikipedia.org/wiki/Network_stack). On creation a network namespace contains only a [loopback](https://en.wikipedia.org/wiki/Localhost) interface. Each [network interface](https://en.wikipedia.org/wiki/Network_interface) (physical or virtual) is present in exactly 1 namespace and can be moved between namespaces. Each namespace will have a private set of [IP addresses](https://en.wikipedia.org/wiki/IP_address), its own [routing table](https://en.wikipedia.org/wiki/Routing_table), [socket](https://en.wikipedia.org/wiki/Network_socket) listing, connection tracking table, [firewall](https://en.wikipedia.org/wiki/Firewall_(computing)), and other network-related resources. Destroying a network namespace destroys any virtual interfaces within it and moves any physical interfaces within it back to the initial network namespace.

**Mount (mnt):**

Mount namespaces control mount points. Upon creation the mounts from the current mount namespace are copied to the new namespace, but mount points created afterwards do not propagate between namespaces (using shared subtrees, it is possible to propagate mount points between namespaces.

**Interprocess Communication (ipc):**

IPC namespaces isolate processes from [SysV](https://en.wikipedia.org/wiki/UNIX_System_V" \o "UNIX System V) style inter-process communication. This prevents processes in different IPC namespaces from using, for example, the SHM family of functions to establish a range of shared memory between the two processes. Instead each process will be able to use the same identifiers for a shared memory region and produce two such distinct regions.

**UTS**

UTS (UNIX [Time-Sharing](https://en.wikipedia.org/wiki/Time-sharing)) namespaces allow a single system to appear to have different [host](https://en.wikipedia.org/wiki/Hostname) and [domain names](https://en.wikipedia.org/wiki/Domain_name) to different processes. "When a process creates a new UTS namespace ... the hostname and domain of the new UTS namespace are copied from the corresponding values in the caller's UTS namespace." [[6]](https://en.wikipedia.org/wiki/Linux_namespaces#cite_note-6)

**User ID (user):**

User namespaces are a feature to provide both privilege isolation and user identification segregation across multiple sets of processes available since kernel 3.8.[[7]](https://en.wikipedia.org/wiki/Linux_namespaces#cite_note-7) With administrative assistance it is possible to build a container with seeming administrative rights without actually giving elevated privileges to user processes. Like the PID namespace, user namespaces are nested and each new user namespace is considered to be a child of the user namespace that created it.

**Command:**

|  |  |
| --- | --- |
| docker run nginx  docker run ubuntu  ---------------------  mkdir /class  cd /class  vim Dockerfile  FROM ubuntu:latest  RUN apt update | docker build -t name/payment:0.0.1    docker images  ---------------  docker rmi imageid -f #force  docker images |

Ref:

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

<https://www.redhat.com/sysadmin/7-linux-namespaces>