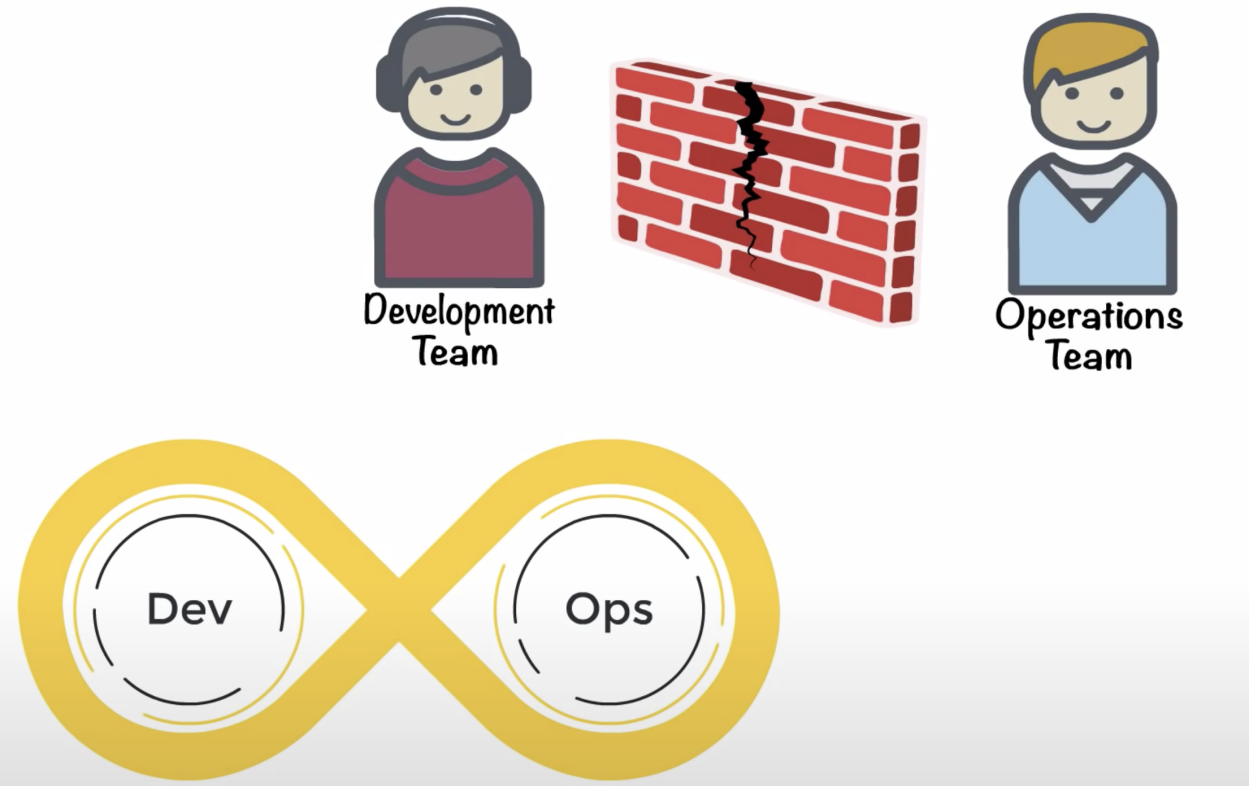
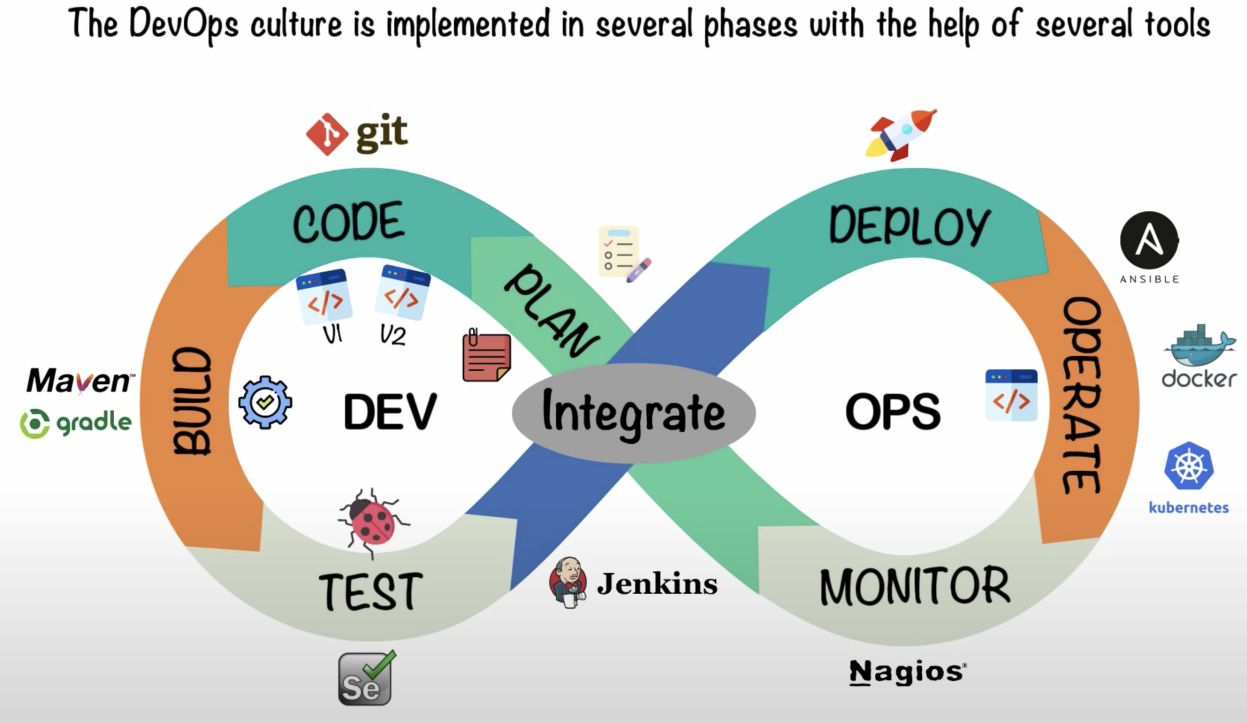
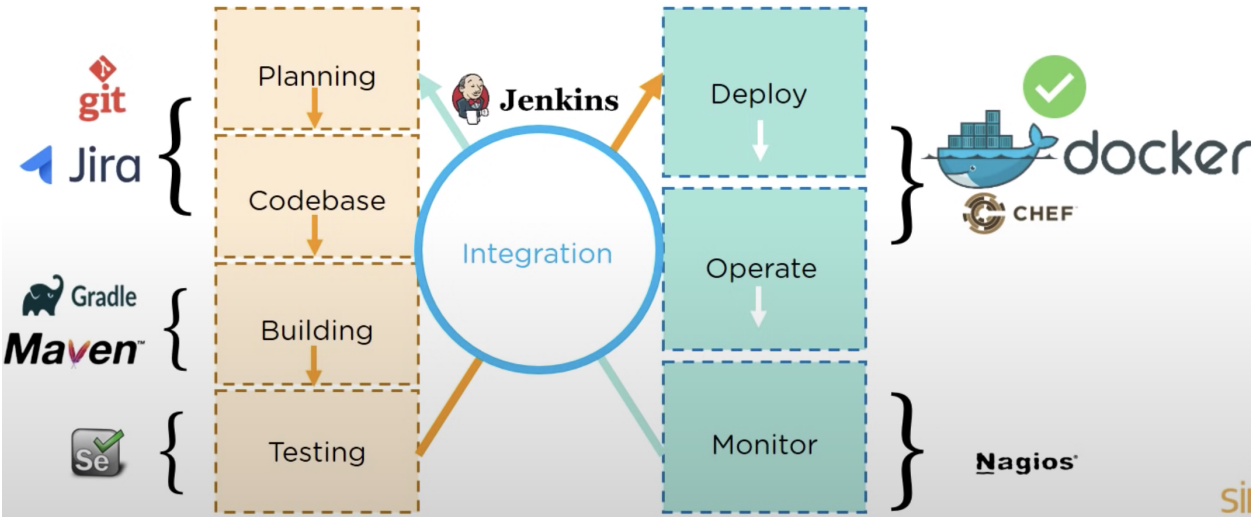
Inception

Devops : is a set of practices that combines software development and IT operations. It aims to shorten the systems development life cycle and provide continuous delivery with high software quality.



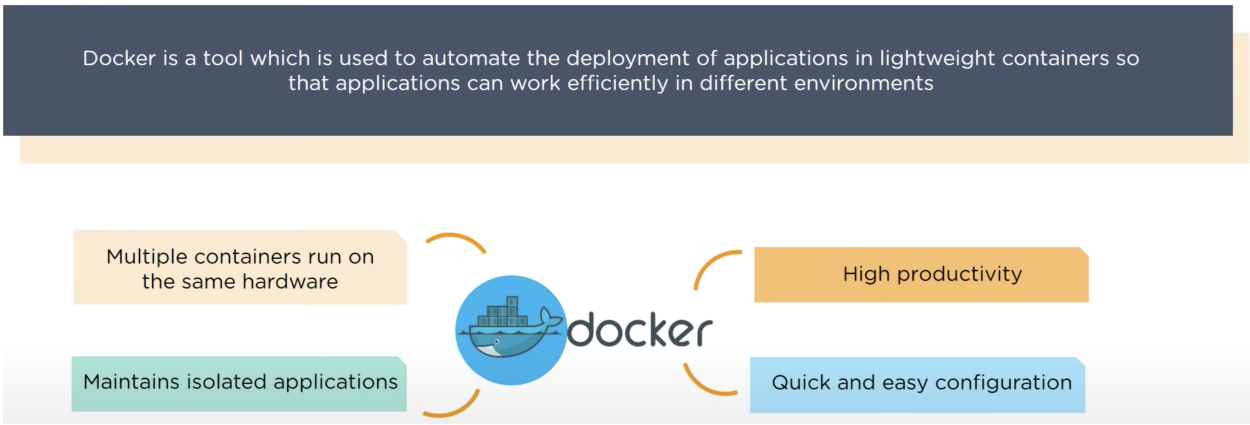


To Watch : https://www.youtube.com/watch?v=Xrgk023l4lI&ab_channel=Simplilearn



Docker : is an open source platform that enables developers to build, deploy, run, update and manage *containers* —standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment.

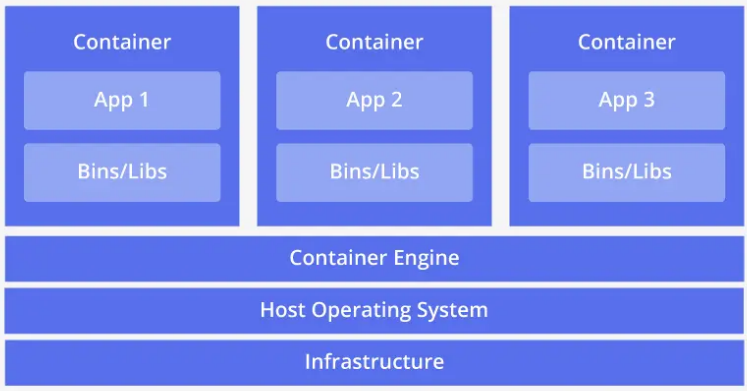
What is Docker?



Docker is a software platform (framework) 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.

An **environment variable** is a dynamic-named value that can affect the way running processes will behave on a computer. They are part of the environment in which a process runs

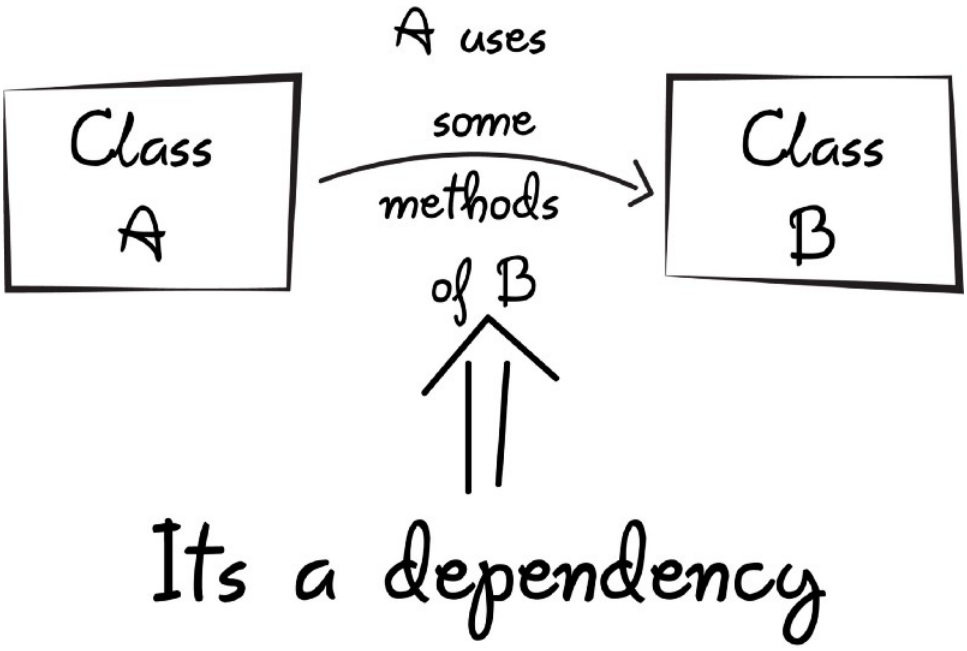
Docker Container



A container is nothing but a box that has the ability to run the docker image templates. The moment you create a container using those immutable images you essentially end up creating a read-write copy of that filesystem (docker image) inside the given container. This adds a container layer which helps you to modify the entire copy of the given Docker image.

A container can also be considered as a cohesive software unit that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.
Containers are executable units of software in which application code is packaged, along with its libraries and dependencies, in common ways so that it can be run anywhere, whether it be on desktop, traditional IT, or the cloud.

Suppose you built a simple program A to get number. And you are using a library B to get random number to program A. Let's say you are creating a program C which uses program A which in turn depends upon library B. So now program A is your library and library B becomes your dependency.



▼ **More about containers :**

Containers are packages of software that contain all of the necessary elements to run in any environment. In this way, containers virtualize the operating system and

run anywhere, from a private data center to the public cloud or even on a developer’s personal laptop.

From Gmail to YouTube to Search, everything at Google runs in containers.

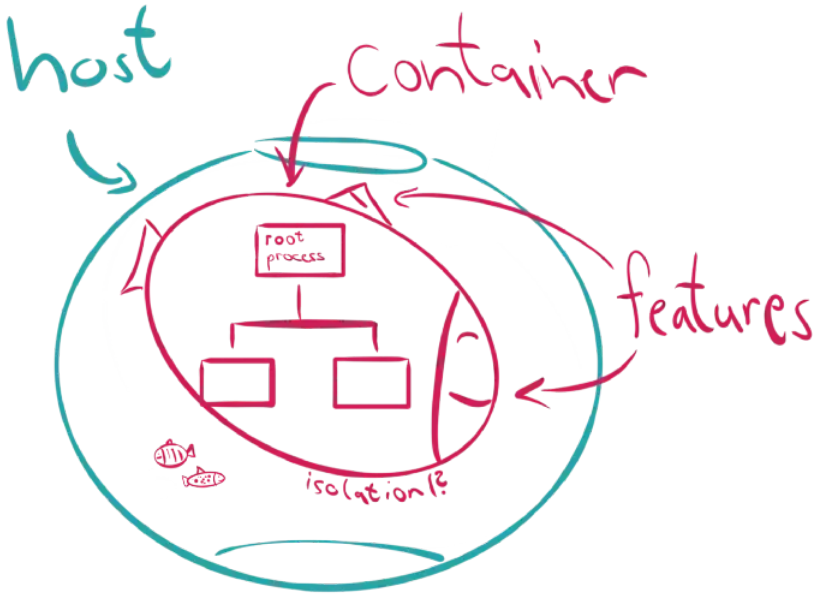
Containers make it easy to share CPU, memory, storage, and network resources at the operating systems level and offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run.

If we strip it down then containers are only isolated groups of processes running on a single host, which fulfill a set of “common” features. Some of these fancy features are built directly into the Linux kernel and mostly all of them have different historical origins.

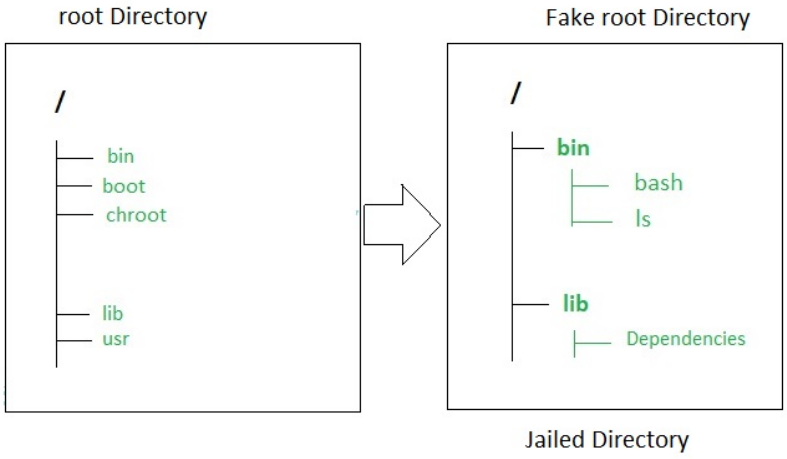
So containers have to fulfill four major requirements to be acceptable as such:

1. Not negotiable: They have to run on a single host. Okay, so two computers cannot run a single container.
2. Clearly: They are groups of processes. You might know that Linux processes live inside a tree structure, so we can say containers must have a root process.
3. Okay: They need to be isolated, whatever this means in detail.
4. Not so clear: They have to fulfill common features. Features in general seem to change over time, so we have to point out what the most common features are.

   Features in containers refer to the capabilities and characteristics that make containers a powerful tool for deploying and managing applications



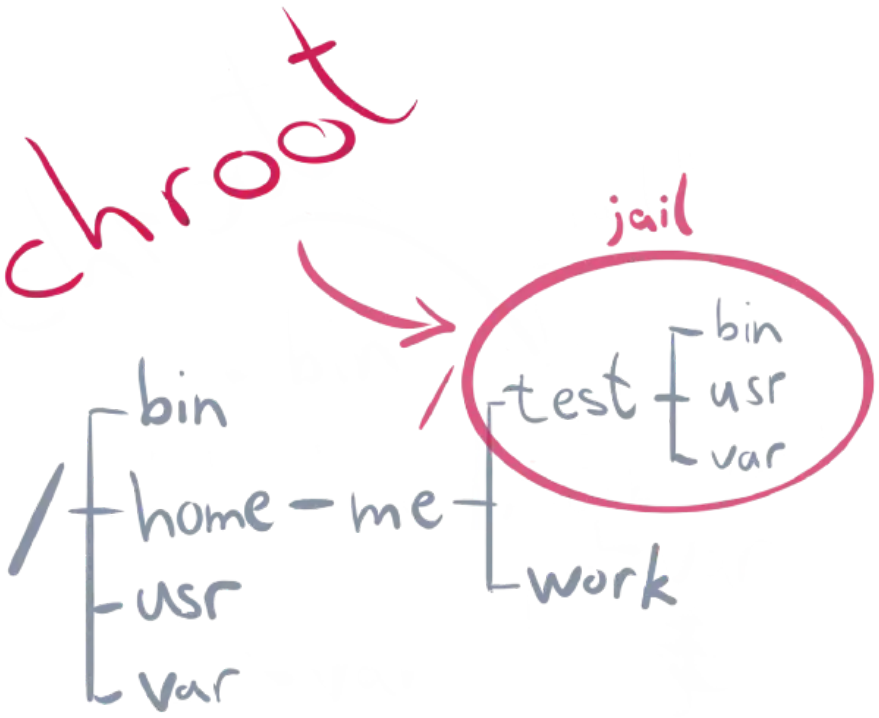
chroot command in Linux/Unix system is used to change the root directory. Every process/command in Linux/Unix like systems has a current working directory called **root directory** It changes the root directory for currently running processes as well as its child processes. A process/command that runs in such a modified environment cannot access files outside the root directory. This modified environment is known as "**chroot jail**" or **"jailed directory"**. Some root user and privileged process are allowed to use chroot command.



What is needed to run an own chroot environment? Not that much, since something like this already works:

```
> mkdir -p new-root/{bin,lib64}
> cp /bin/bash new-root/bin
> cp /lib64/{ld-linux-x86-64.so*,libc.so*,libdl.so.2,libreadline.so*,libtinfo.so*} ne
w-root/lib64
> sudo chroot new-root
```

We create a new root directory, copy a bash shell and its dependencies in and run `chroot`. This jail is pretty useless: All we have at hand is bash and its builtin functions like `cd` and `pwd`



The current working directory is left unchanged when calling chroot via a syscall, whereas relative paths can still refer to files outside of the new root. This call changes only the root path and nothing else. Beside this, further calls to chroot do not stack and they will override the current jail. Only privileged processes with the capability `CAP_SYS_CHROOT`
are able to call chroot.

Nowadays chroot is not used by container runtimes any more and was replaced by `pivot_root(2)`, which has the benefit of putting the old mounts into a separate directory on calling. These old mounts could be unmounted afterwards to make the filesystem completely invisible to broken out processes.

To continue with a more useful jail we need an appropriate root filesystem (rootfs). This contains all binaries, libraries and the necessary file structure. But where to get one? What about peeling it from an already existing Open Container Initiative (OCI) container, which can be easily done with the two tools `skopeo` and `umoci`:

```
> skopeo copy docker://opensuse/tumbleweed:latest oci:tumbleweed:latest
[output removed]
> sudo umoci unpack --image tumbleweed:latest bundle
[output removed]
```

▼ **What is a file system?**

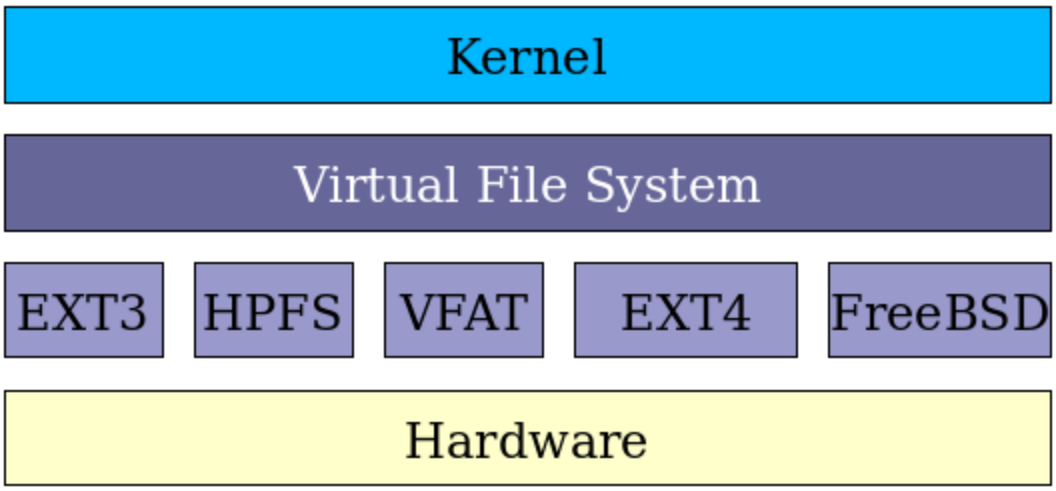
Let's start with a simple definition:

A **file system** defines how files are **named**, **stored**, and **retrieved** from a storage device.

Every time you open a file on your computer or smart device, your operating system uses its file system internally to load it from the storage device.

Or when you copy, edit, or delete a file, the file system handles it under the hood.

Whenever you download a file or access a web page over the Internet, a file system is involved too.



There is no process isolation available at all. We can even kill programs running outside of the jail, what a metaphor! Let's peek into the network devices:

```
> mkdir /sys
> mount -t sysfs sys /sys
> ls /sys/class/net
eth0 lo
```

There is no process isolation available at all. We can even kill programs running outside of the jail, what a metaphor! Let's peek into the network devices.

There is no network isolation, too. This missing isolation paired with the ability to leave the jail leads into lots of security related concerns, because jails are sometimes used for wrong (security related) purposes. How to solve this? This is where the Linux namespaces join the party.

Namespaces are a Linux kernel feature which were introduced back in 2002 with Linux 2.4.19. The idea behind a namespace is to wrap certain global system resources in an abstraction layer. This makes it appear like the processes within a namespace have their own isolated instance of the resource. The kernels namespace abstraction allows different groups of processes to have different views of the system.

Not all available namespaces were implemented from the beginning. A full support for what we now understand as "container ready" was finished in kernel version 3.8 back in 2013 with the introduction of the user namespace. We end up having

currently seven distinct namespaces implemented: mnt, pid, net, ipc, uts, user and cgroup. No worries, we will discuss them in detail. In September 2016 two additional namespaces were proposed (time and syslog) which are not fully implemented yet. Let's have a look into the namespace API before digging into certain namespaces.

Namespaces and cgroups are two Linux kernel features that are used to provide isolation and resource management for containers.

1. **Namespaces**: Namespaces provide a way to isolate different parts of the system, such as the file system, network, and process tree, from each other. When a container is created, it runs in its own namespace, which gives it its own isolated environment. This means that the processes running inside the container are isolated from the host system and from other containers.
2. **cgroups**: cgroups, or control groups, are a feature that allows you to manage and allocate system resources, such as CPU, memory, and I/O bandwidth, to containers. cgroups provide a way to limit, prioritize, and distribute resources to containers, ensuring that each container gets the resources it needs to run efficiently.

Together, namespaces and cgroups form the foundation for container isolation and resource management in Linux. By using namespaces to isolate the file system, network, and process tree, and cgroups to manage resources, containers can run isolated from each other and from the host system, while still being able to access the necessary resources to run efficiently.

In Linux, there are several types of namespaces that are used to provide isolation for containers:

1. **PID namespace**: Isolates the process tree, so that processes inside a container are not visible from outside the container.
2. **Network namespace**: Isolates the network stack, so that containers can have their own network interfaces, IP addresses, and firewall rules.
3. **Mount namespace**: Isolates the file system, so that containers can have their own file system view and can mount and unmount file systems without affecting the host system or other containers.
4. **IPC namespace**: Isolates inter-process communication (IPC) resources, such as System V IPC and POSIX message queues. By using the IPC namespace,

each container can have its own isolated IPC resources, so that the processes inside a container can communicate with each other, but cannot communicate with processes outside the container

5. **UTS namespace**: Isolates the host and domain name, so that each container can have its own host name.

▼ **More about Namespace**

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. Examples of such resources are process IDs, host-names, user IDs, file names, and some names associated with network access, and Inter-process communication.

Namespaces are a fundamental aspect of containers in Linux.

The term "namespace" is often used for a type of namespace (e.g. process ID) as well as for a particular space of names.

A Linux system starts out with a single namespace of each type, used by all processes. Processes can create additional namespaces and also join different namespaces.

Kernel resources :

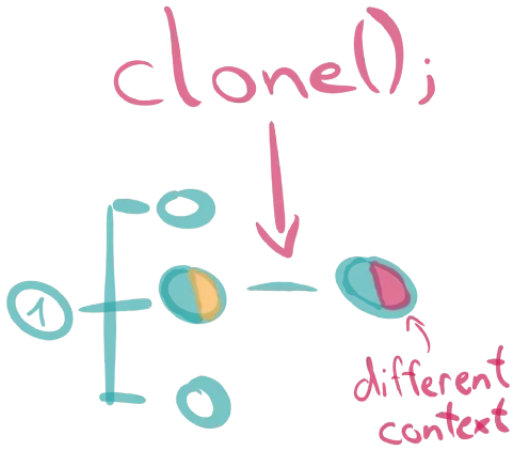
The kernel is the part of an OS that handles the system's resources and the various system calls (requests from software) and how the software and hardware respond to this.
The kernel, as the intermediary between the CPU and the process, is the one who allows or not the execution of certain processes... so, to finally answer, the resources allow a process, access to the services of the kernel. One of the main resources of the linux kernel is the c library, that provides the services that requests the process to the kernel, so, without the resources from the kernel to provide them services, the process would't be able to work properly.

API

The namespace API of the Linux kernel consists of three main system calls:

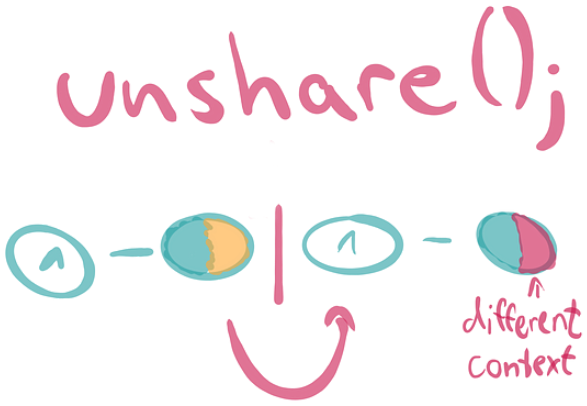
clone

The `clone(2)` API function creates a new child process, in a manner similar to `fork(2)`. Unlike `fork(2)`, the `clone(2)` API allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers. You can pass different namespace flags to `clone(2)` to create new namespaces for the child process.



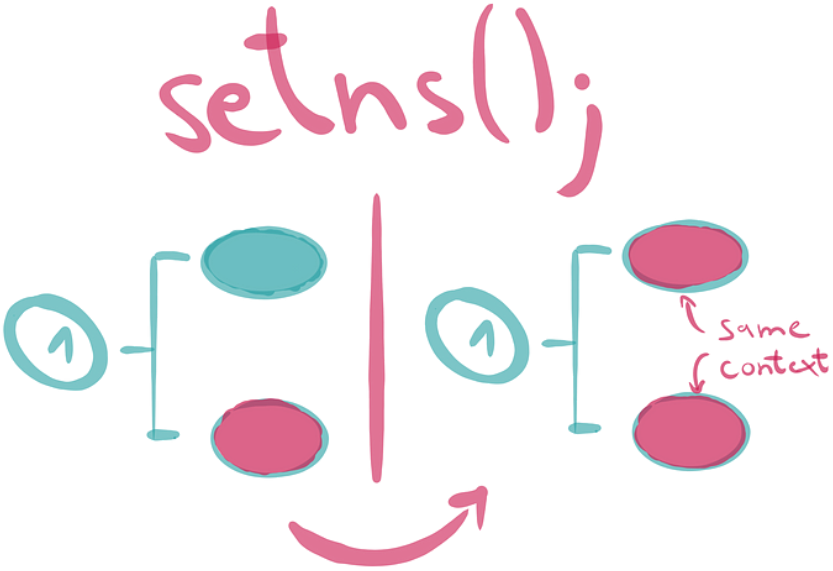
unshare

The function `unshare(2)` allows a process to disassociate parts of the execution context which are currently being shared with others.



setns

The function `setns(2)` reassociates the calling thread with the provided namespace file descriptor. This function can be used to join an existing namespace.



This allows us for example to track in which namespaces certain processes reside. Another way to play around with namespaces apart from the programmatic approach is using tools from the `util-linux` package. This

contains dedicated wrapper programs for the mentioned syscalls. One handy tool related to namespaces within this package is `lsns`. It lists useful information about all currently accessible namespaces or about a single given one.

Available Namespaces

Mount (mnt)

The first namespace we want to try out is the mnt namespace, which was the first implemented one back in 2002. During that time (mostly) no one thought that multiple namespaces would ever be needed, so they decided to call the namespace clone flag `CLONE_NEWNS`. This leads into a small inconsistency with other namespace clone flags (I see you suffering!). With the mnt namespace Linux is able to isolate a set of mount points by a group of processes.

A mount point is a directory where an external partition, storage device, or file system can become accessible to the user or application requiring it (e.g., when a Linux installation process requires access to the USB device containing the ISO image or installation files).

A great use case of the mnt namespace is to create environments similar to jails, but in a more secure fashion. How to create such a namespace? This can be easily done via an API function call or the unshare command line tool.

The actual memory being used for the mount point is laying in an abstraction layer called Virtual File System (VFS), which is part of the kernel and where every other filesystem is based on. If the namespace gets destroyed, the mount memory is unrecoverably lost. The mount namespace abstraction gives us the possibility to create entire virtual environments in which we are the root user even without root permissions.

UNIX Time-sharing System (uts)

The UTS namespace was introduced in Linux 2.6.19 (2006) and allows us to unshare the domain- and hostname from the current host system,Let’s give it a try:

```
> sudo unshare -u
# hostname
```

```
nb
# hostname new-hostname
# hostname
new-hostname
```

And if we look at the system level nothing has changed, hooray:

```
> hostname
nb
```

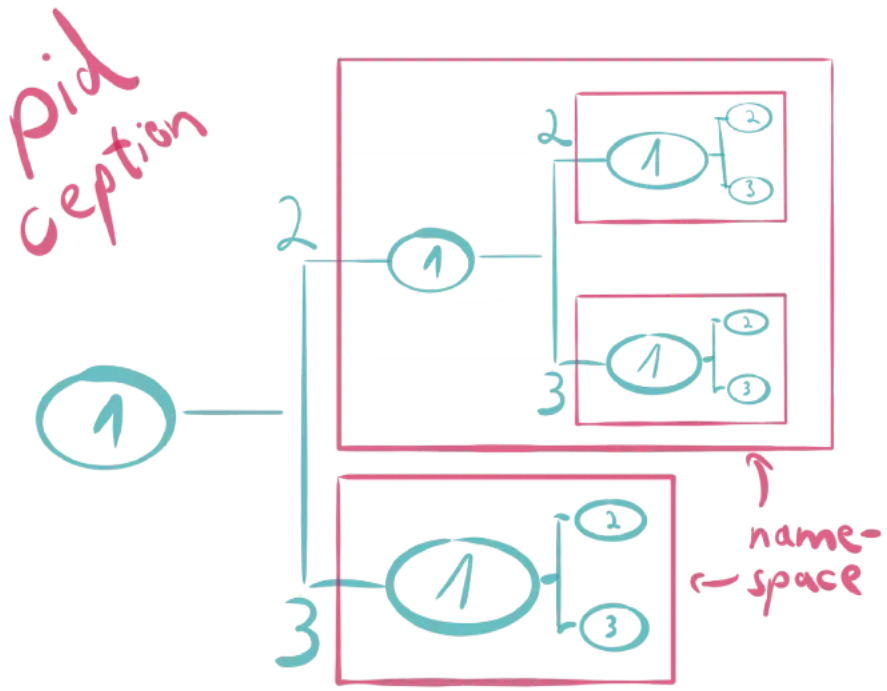
The UTS namespace is yet another nice addition in containerization, especially when it comes to container networking related topics.

Interprocess Communication (ipc)

IPC namespaces came with Linux 2.6.19 (2006) too and isolate interprocess communication (IPC) resources. In special these are System V IPC objects and POSIX message queues. One use case of this namespace would be to separate the shared memory (SHM) between two processes to avoid misusage. Instead, each process will be able to use the same identifiers for a shared memory segment and produce two distinct regions. When an IPC namespace is destroyed, then all IPC objects in the namespace are automatically destroyed, too.

Process ID (pid)

The PID namespace was introduced in Linux 2.6.24 (2008) and gives processes an independent set of process identifiers (PIDs). This means that processes which reside in different namespaces can own the same PID. In the end a process has two PIDs: the PID inside the namespace, and the PID outside the namespace on the host system. The PID namespaces can be nested, so if a new process is created it will have a PID for each namespace from its current namespace up to the initial PID namespace.



The first process created in a PID namespace gets the number 1 and gains all the same special treatment as the usual init process. For example, all processes within the namespace will be re-parented to the namespace's PID 1 rather than the host PID 1. In addition the termination of this process will immediately terminate all processes in its PID namespace and any descendants. Let's create a new PID namespace:

Network (net)

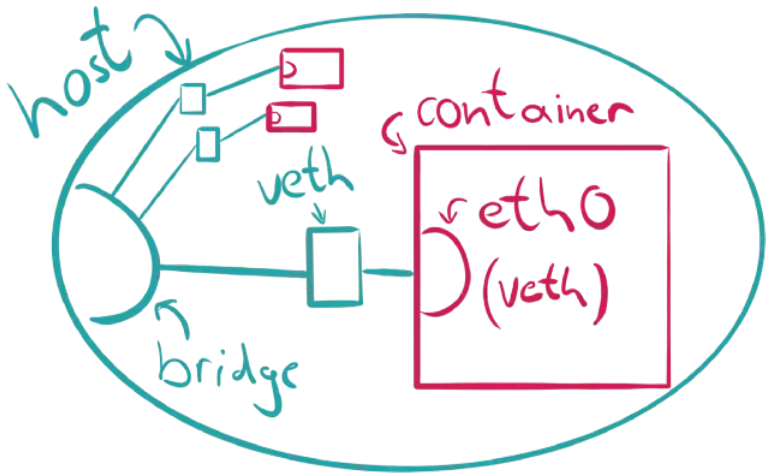
Network namespaces were completed in Linux 2.6.29 (2009) and can be used to virtualize the network stack. Each network namespace contains its own resource properties within `/proc/net`. Furthermore, a network namespace contains only a loopback interface on initial creation.

Every network interface (physical or virtual) is present exactly once per namespace. It is possible that an interface will be moved between namespaces. Each namespace contains a private set of IP addresses, its own routing table, socket listing, connection tracking table, firewall, and other network-related resources.

Destroying a network namespace destroys any virtual and moves any physical

interfaces within it back to the initial network namespace.

A possible use case for the network namespace is creating Software Defined Networks (SDN) via virtual Ethernet (veth) interface pairs. One end of the network pair will be plugged into a bridged interface whereas the other end will be assigned to the target container. This is how pod networks like flannel work in general.



Let's see how it works. First, we need to create a new network namespace, which can be done via `ip`, too:

```
> sudo ip netns add mynet
> sudo ip netns list
mynet
```

So we created a new network namespace called `mynet`. When `ip` creates a network namespace, it will create a bind mount for it under `/var/run/netns` too. This allows the namespace to persist even when no processes are running within it.

User ID (user)

With Linux 3.5 (2012) the isolation of user and group IDs was finally possible via namespaces. Linux 3.8 (2013) made it possible to create user namespaces even without being actually privileged. The user namespace enables that a user and group IDs of a process can be different inside and outside of the namespace. An interesting use-case is that a process can have a normal unprivileged user ID outside a user namespace while being fully privileged inside.

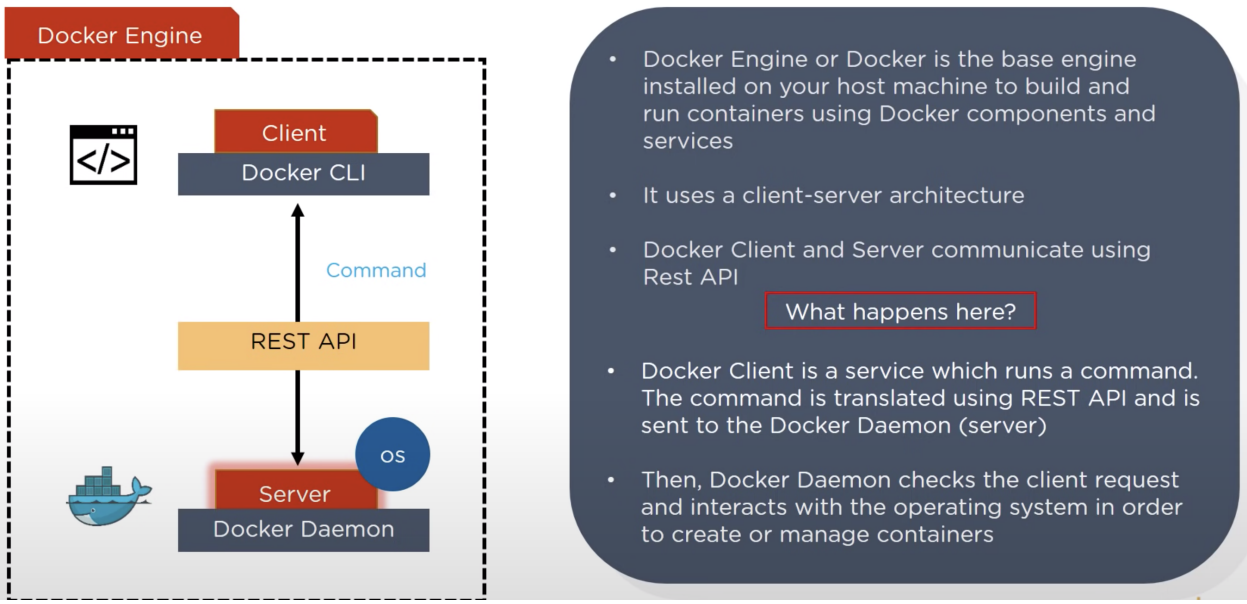
In a Unix system, a GID (group ID) is a name that associates a system user with other users sharing something in common (perhaps a work project or a department name). It's often used for accounting purposes. A user can be a member of more than one group and thus have more than one GID.

Microservices are both an architecture and an approach to writing software. With microservices, applications are broken down into their smallest components, independent from each other. Instead of a traditional, monolithic, approach to apps, where everything is built into a single piece, microservices are all separated and work together to accomplish the same tasks. Each of these components, or processes, is a microservice. This approach to software development values granularity, being lightweight, and the ability to share similar process across multiple apps. It is a major component of optimizing application development towards a cloud-native model.

▼ File Descriptors

In simple words, when you open a file, the operating system creates an entry to represent that file and store the information about that opened file. So if there are 100 files opened in your OS then there will be 100 entries in OS (somewhere in kernel). These entries are represented by integers like (...100, 101, 102....). This entry number is the file descriptor. So it is just an integer number that uniquely represents an opened file for the process. If your process opens 10 files then your Process table will have 10 entries for file descriptors.

Similarly, when you open a network socket, it is also represented by an integer and it is called Socket Descriptor. I hope you understand.



Docker Engine is an open source containerization technology for building and containerizing your applications. Docker Engine acts as a client-server application with:

- A server with a long-running daemon process `dockerd`.
- APIs which specify interfaces that programs can use to talk to and instruct the Docker daemon.
- A command line interface (CLI) client `docker`.

▼ **containerization**

Containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure. Traditionally, to run any application on your computer, you had to install the version that matched your machine's operating system. For example, you needed to install the Windows version of a software package on a Windows machine. However, with containerization, you can create a single software package, or container, that runs on all types of devices and operating systems.

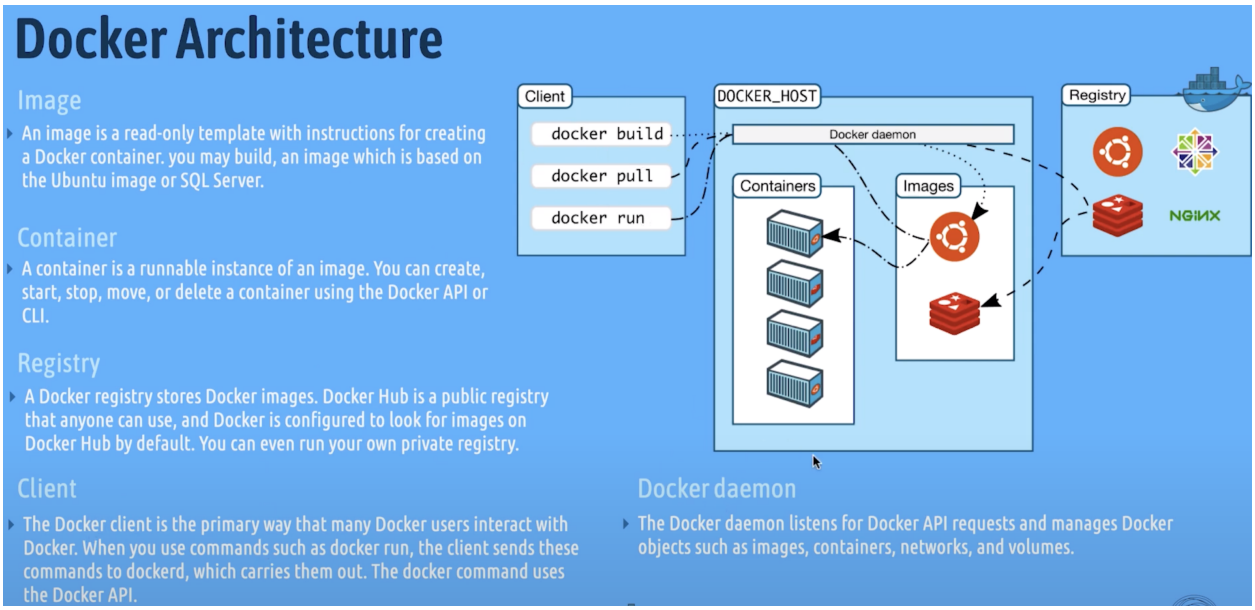
The CLI uses Docker APIs to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications use the underlying API and CLI. The daemon creates and manage Docker objects, such as images, containers, networks, and volumes.

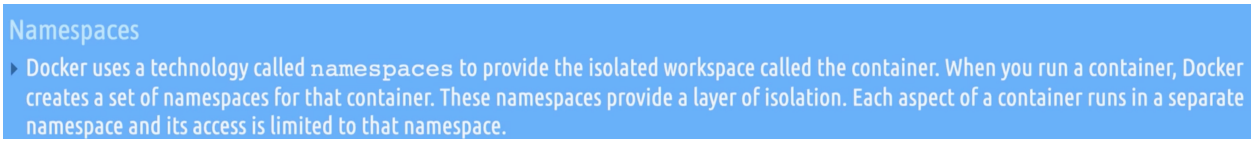
Dockerd (Docker Daemon): is the persistent process that manages containers. Docker uses different binaries for the daemon and client.

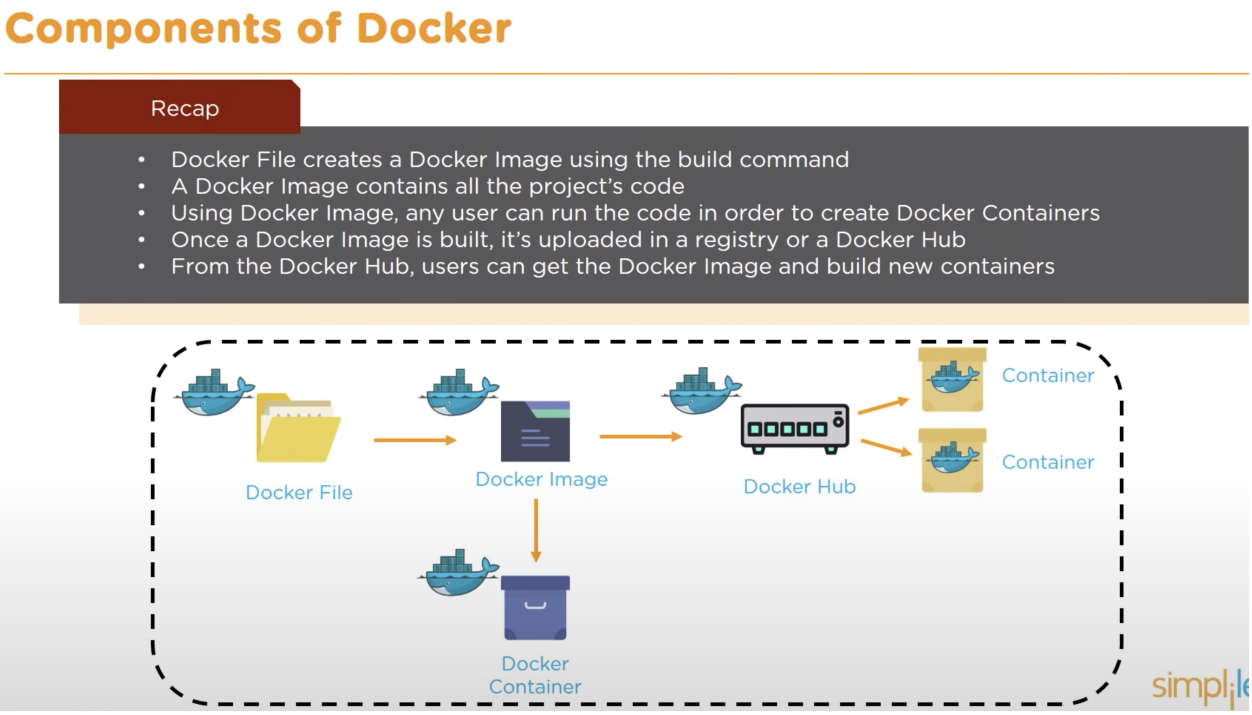
Docker daemon is the brain behind the whole operation, like aws itself. When you use `docker run` command to start up a container, your docker client will translate that command into http API call, sends it to docker daemon, Docker daemon then evaluates the request, talks to underlying os and provisions your container.

To run the daemon you type `dockerd`.

To run the daemon with debug output, use `dockerd --debug` or add `"debug": true` to the `daemon.json` file.





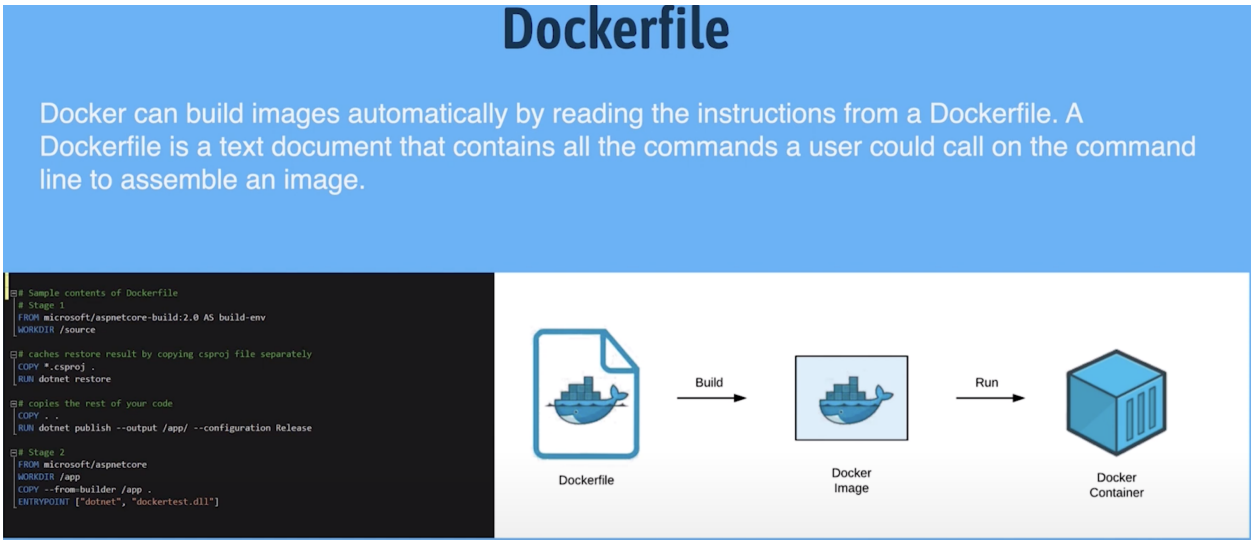


Docker Images

It is a kind of ready-to-use software read-only template crafted with source codes, libraries, external dependencies, tools, and other miscellaneous files that are needed for any software application to run successfully on any platform or OS.

The developer community also likes to call it Snapshots, representing the app and its virtual environment at a specific point in time.

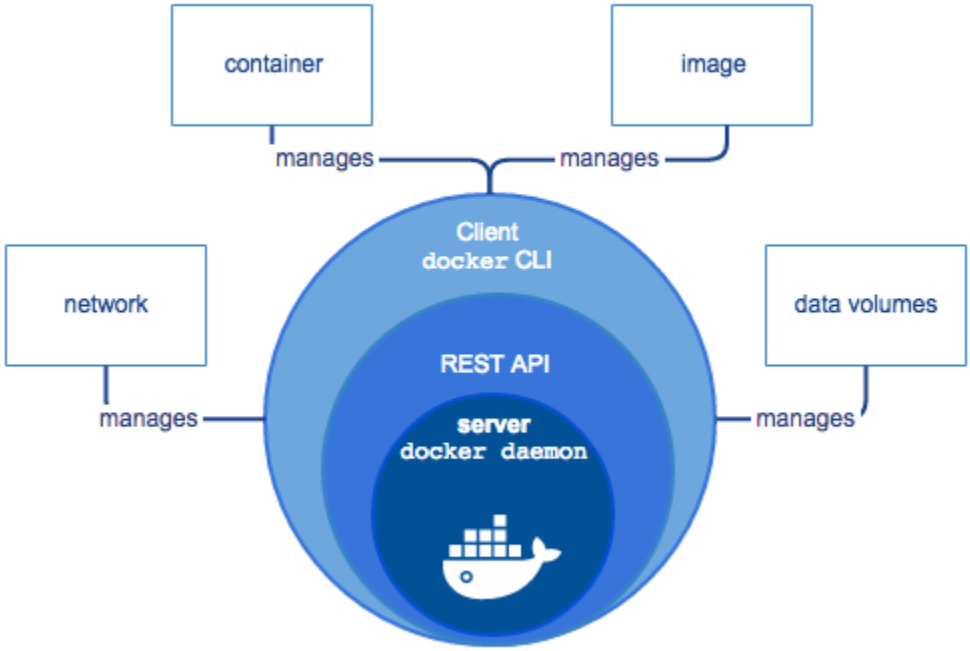
Docker builds images automatically by reading the instructions from a DockerFile. It is a text file that contains all commands needed to build a given image.



Why Docker Containers Are Useful?

All the docker images become docker containers when they run on Docker Engine and these containers are popular amongst developers and organizations of all shapes and sizes because it is :

- **Standardized:** Docker created the industry standard for containers, so they could be portable anywhere.
- **Lightweight:** Containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs.
- **Secure:** Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry.



Docker Commands :

docker run | docker pull | docker container run : One of the first and most important commands Docker users learn is the `docker run` command. This comes as no surprise since its primary function is to build and run containers.

There are many different ways to run a container. By adding attributes to the basic syntax, you can configure a container to run in detached mode, set a container name, mount a volume, and perform many more tasks.

docker ps -all | docker container ls -a : (ps = process) how many docker containers we have .

There is no difference between the `docker ps` (docker process status) and `docker container ls` (docker container list) commands in terms of functionality. They even allow the same set of flags. The only real difference between the two is the fact that the latter is newer and more verbose than the former.

The `docker container ls` command was introduced as a part of Docker's initiative to consolidate the Docker API by grouping commands by features and concepts such as networks, volumes, containers, etc. This makes the commands more readable and obvious, and the API more intuitive and much easier for users to get started quickly.

docker container run -d [image] : run with detaches (or —-detach)

- You can run container in attached mode (in the foreground) or in detached mode (in the background).
- By default, Docker runs the container in attached mode. In the attached mode, Docker can start the process in the container and attach the console to the process's standard input, standard output, and standard error.
- Detached mode, started by the option --detach or –d flag in docker run command, means that a Docker container runs in the background of your terminal. It does not receive input or display output. Using detached mode also allows you to close the opened terminal session without stopping the container.

While running a container in the foreground is that you cannot access the command prompt anymore. Which means you cannot run any other commands while the container is running. In the first screenshot we run in detached mode, in the second screenshot we run in attached mode.

docker run —publish 80:80 : Publish a container's port(s) to the host

docker —name —name : name of containers

`more information about option or shorthand :` https://docs.docker.com/engine/reference/commandline/run/

docker container run redis : tags : run with tags (docker tags are reference to docker images) like alpine

The Docker tag helps maintain the build version to push the image to the Docker Hub**. The Docker Hub allows us to group images together based on name and tag.**

The `latest` tag is applied by default to reference an image when you run Docker commands without specifying the tag value

Alpine Linux is an independent, non-commercial, general purpose Linux distribution designed for power users who appreciate security, simplicity and resource efficiency.

Alpine Linux is built around musl libc and busybox. This makes it small and very resource efficient. A container requires no more than 8 MB and a minimal installation to disk requires around 130 MB of storage. Not only do you get a fully-fledged Linux environment but a large selection of packages from the repository.
Binary packages are thinned out and split, giving you even more control over what you install, which in turn keeps your environment as small and efficient as possible.

docker images | docker image ls : how many images we have

docker rm [id_containers] : remove container

docker image rm [image] : remove images

In short, a container is a runnable instance of an image. which is why you cannot delete an image if there is a running container from that image. **You just need to delete the container first.**

docker inspect [image | container] : information about image

docker logs [id_containers | first-three-id | NAMES] : To display valuable logs, you can use a range of Docker log commands .Every container produces logs with valuable information. A log is basically data written by the container to STDOUT or STDERR. However, if you run a container in detached mode, you can't see the logs in your console because detached mode refers to running a container in the background of your terminal. Therefore, you won't see any logging or other output from your Docker container.

docker stats : command returns a live data stream for running containers, The `docker stats` command is a built-in feature of Docker that displays resource consumption statistics for the container in real-time. By default, it shows CPU and memory utilization for all containers. Stats here refers to "statistics". You can restrict the statistics by entering the container names or IDs you're interested in monitoring. The `docker stats` command output includes CPU stats, memory metrics, block I/O, and network IO metrics for all active containers.

docker info : display docker information

docker inspect --format='{{range.NetworkSettings.Networks}}{{.IPAddress}} {{end}}' [id-container] : get IP address

Docker format is used to manipulate the output format of commands and log drivers that have the '–format' option, which means if a command has the '–format' option, then we can use that option to change the output format of the command as per our requirement because default command does not shows all the fields related to that object. For example, if we run the 'docker image ls' command, it shows only repository, tag, image ID, created, and size; however, a few fields are associated with images like containers, digest, createdate, virtual size, etc.

docker container start [container] : Start one or more stopped containers

docker container stop [container] : Start one or more started containers

docker exec [OPTIONS] container command [ARG...] : Run a command in a running container

```
Run a process in a running container.
The command started using docker cexec will only run while the container's primary process
(PID1) is running, and will not be restarted if the container is restarted. if the contain
er is paused, then the docker exec command will wait until the container is unpaused, and
 then run
```

docker history [image] : show the tags for image

docker tag image name_compte/image : Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE

Docker tags are just an alias for an image ID.The tag's name must be an ASCII character string and may include lowercase and uppercase letters, digits, underscores, periods, and dashes. In addition, the tag names must not begin with a period or a dash, and they can only contain 128 characters.

docker login : login to compte in docker hub

docker push image : push docker to docker hub (first u have to login)

docker logout : logout in compte

docker build [OPTIONS] PATH : Build an image from a Dockerfile (path = '.' in our local)

docker volume ls : display volume

docker volume create name-volume : create volume

docker container run -it —name [name_of_container] -v name_volume:/var/lib/mysql [name_image] : create container and put in volume

The `-v` (or `--volume`) argument to `docker run` is for creating storage space inside a container that is separate from the rest of the container filesystem. There are two forms of the command.

The interactive mode in Docker allows us to execute commands while the container is in a *running* state.To run the Docker container in interactive mode, we

use the `-it` option. Further, we attach both the STDIN and STDOUT channels to our terminal with the `-it` flags.

Let's now run the *tomcat* container from earlier using *docker-compose* with an interactive shell:

```
version: "3"
services:
 server:
   image: tomcat:jre11-openjdk
   ports:
     - 8080:8080
   stdin_open: true
   tty: trueCopy
```

In this case, **we added the *stdin_open* and *tty* options in the *docker-compose.yml* file so that we can have an interactive shell with the *docker-compose* setup**.

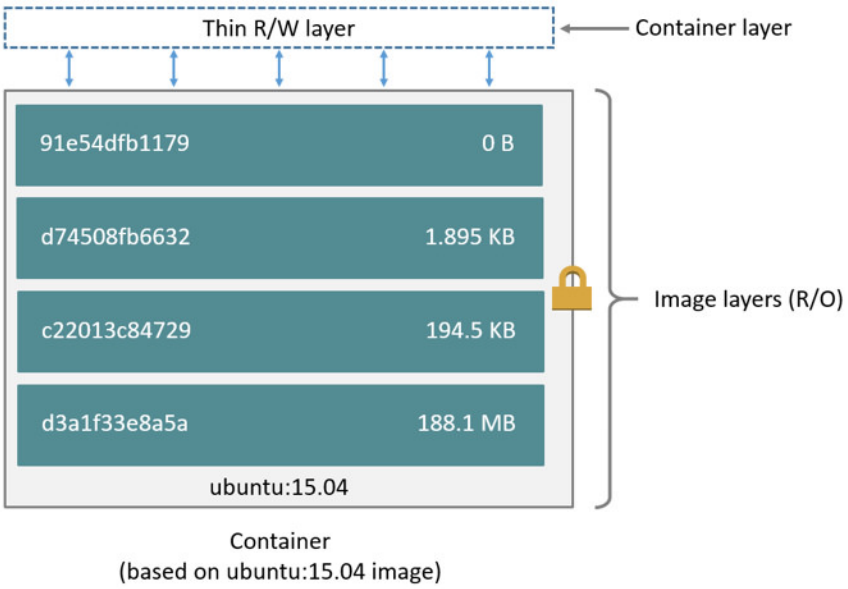
DockerFile Commands :

From : wish is baseImage to build dockerImage

CMD [”cmd”, “parameters”] : CMD instruction allows you to set a default command, which will be executed only when you run container without specifying a command. If Docker container runs with a command, the default command will be ignored. If Dockerfile has more than one CMD instruction, all but last CMD instructions are ignored.

RUN - RUN instruction allows you to install your application and packages required for it. It executes any commands on top of the current image and creates a new layer by committing the results. Often you will find multiple RUN instructions in a Dockerfile. **RUN** executes command(s) in a new layer and creates a new image. E.g., it is often used for installing software packages.

Layers are a result of the way Docker images are built. Each step in a Dockerfile creates a new “layer” that’s essentially a diff of the filesystem changes since the last step. Metadata instructions such as `LABEL` and `MAINTAINER` do not create layers because they don’t affect the filesystem.



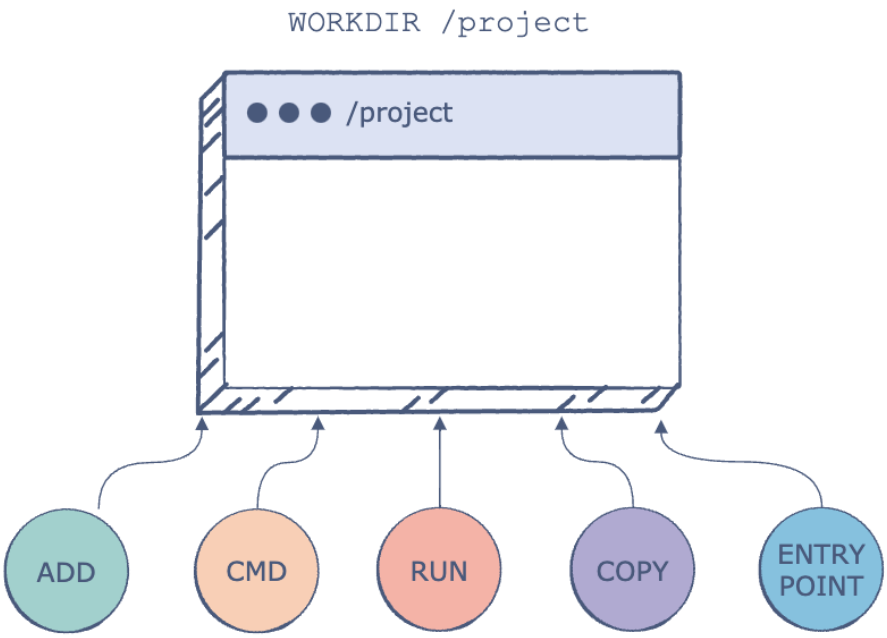
ENTRYPOINT allows specifying a command along with the parameters.

- **CMD.** Sets default parameters that can be overridden from the Docker Command Line Interface (CLI) when a container is running.
- **ENTRYPOINT.** Default parameters that cannot be overridden when Docker Containers run with CLI parameters.

WORKDIR : command is used to define the *working directory* of a Docker container at any given time. The command is specified in the Dockerfile.

Any `RUN`, `CMD`, `ADD`, `COPY`, or `ENTRYPOINT` command will be executed in the specified working directory.

If the `WORKDIR` command is not written in the Dockerfile, it will automatically be created by the Docker compiler. Hence, it can be said that the command performs `mkdir` and `cd` implicitly.



You can think of `WORKDIR` like a `cd` inside the container (it affects commands that come later in the Dockerfile, like the `RUN` command). If you removed `WORKDIR` in your example above, `RUN npm install` wouldn't work because you would not be in the `/usr/src/app` directory inside your container.

I don't see how this would be related to where you put your Dockerfile (since your Dockerfile location on the host machine has nothing to do with the pwd inside the container). You can put the Dockerfile wherever you'd like in your project. However, the first argument to `COPY` is a relative path, so if you move your Dockerfile you may need to update those `COPY` commands.

If the *WORKDIR* instruction is not specified in a Dockerfile, the default *WORKDIR* is the root directory (*/*). In other words, if there is no *WORKDIR* instruction in a Dockerfile, then all the subsequent instructions will execute relative to the root directory.

COPY : Dockerfiles can contain several different instructions, one of which is *COPY*.

The *COPY* instruction lets us copy a file (or files) from the host system into the image. This means the files become a part of every container that is created from that image.

The syntax for the *COPY* instruction is similar to other copy commands we saw above:

```
COPY <SRC> <DEST>Copy
```

Just like the other copy commands, *SRC* can be either a single file or a directory on the host machine. It can also include wildcard characters to match multiple files.

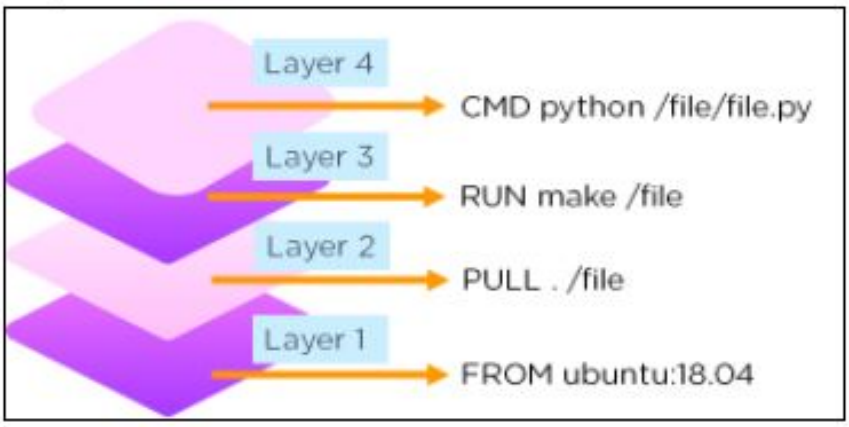
COPY .. : It means the same thing yes, the dot is “where i am now”So it will copy everything from the same place as the dockerfile, to “where i am now” in the container.

The “where i am now” in the image/container is defined by https://docs.docker.com/engine/reference/builder/#workdir 500

So if you set:WORKDIR /tmpand do COPY . .

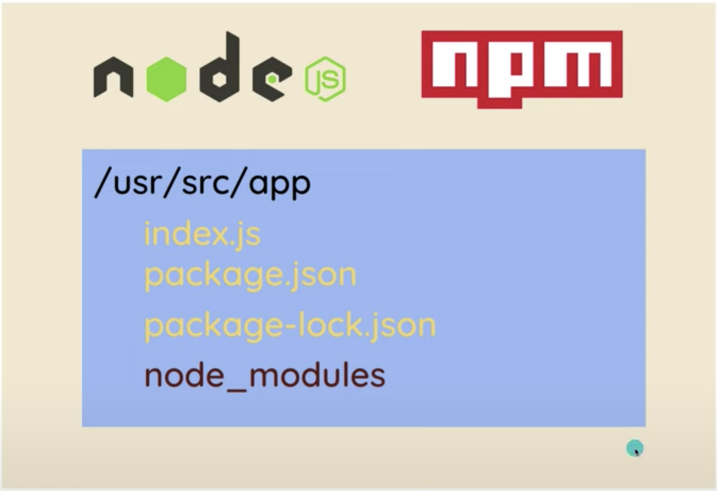
It will copy everything in the current folder, to /tmp

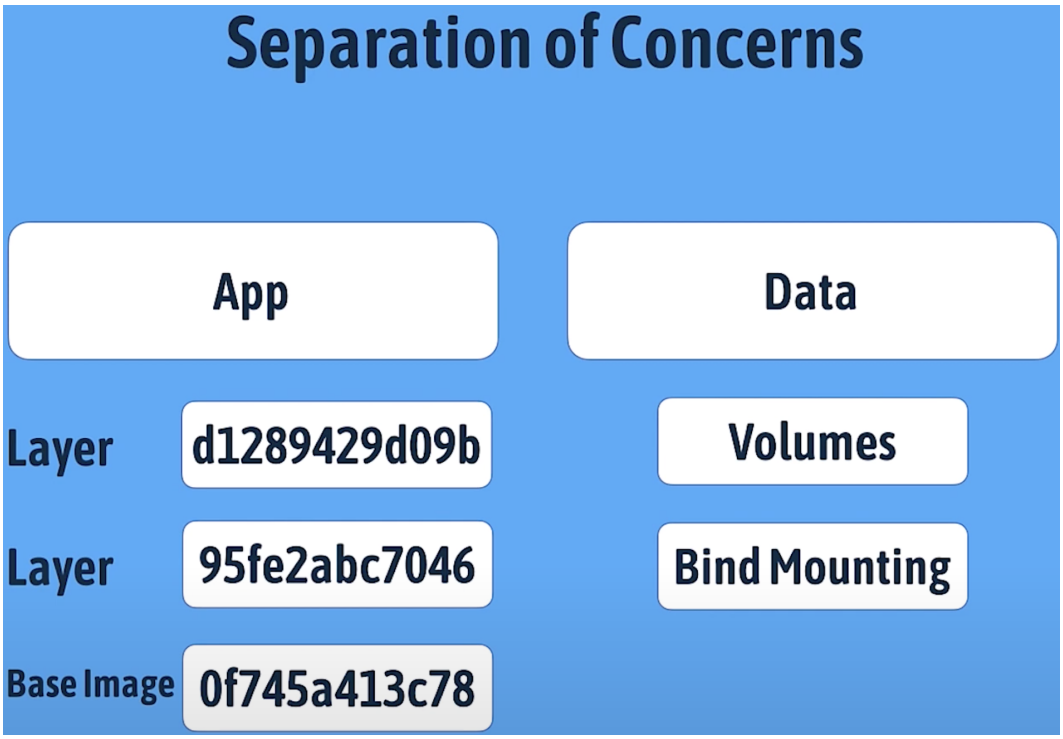
Have a look at the diagrammatic representation of how a dockerfile looks in a docker image:

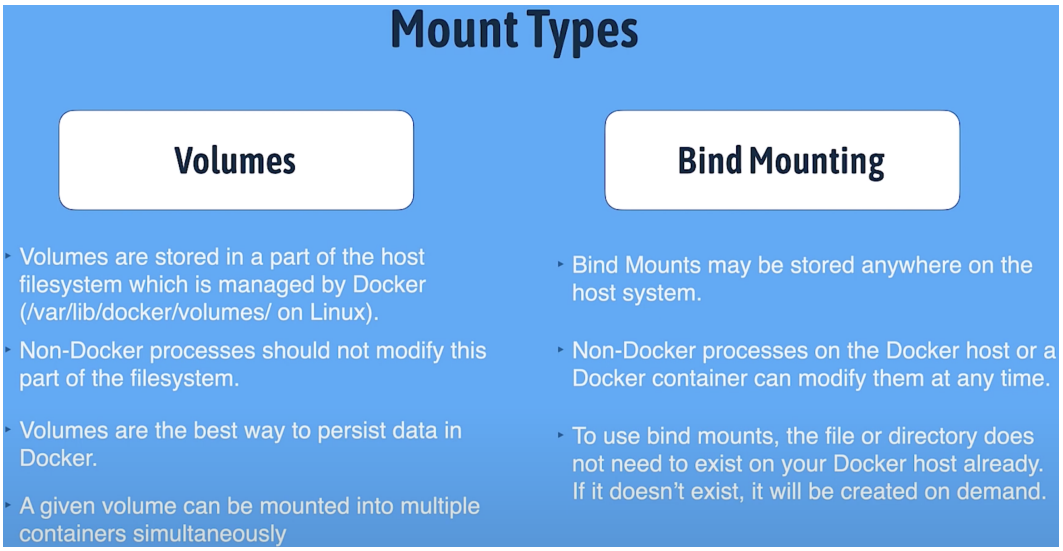


EXPOSE : instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.
Thus, Writing EXPOSE in your Dockerfile, is merely a hint that a certain port is useful

```
FROM node:alpine

WORKDIR /usr/src/app

COPY . .

RUN npm install
```







Volume is a persistent data storage mechanism. It allows data to persist even after a container is deleted, and can also be used to share data between multiple containers

Volumes in Docker are a way to persist data generated by and used by Docker containers. A volume is a separate area within the host file system which is directly accessible to the container. The main advantage of using volumes is that the data within the volume exists outside of the container's filesystem, so even if the container is deleted, the data remains.

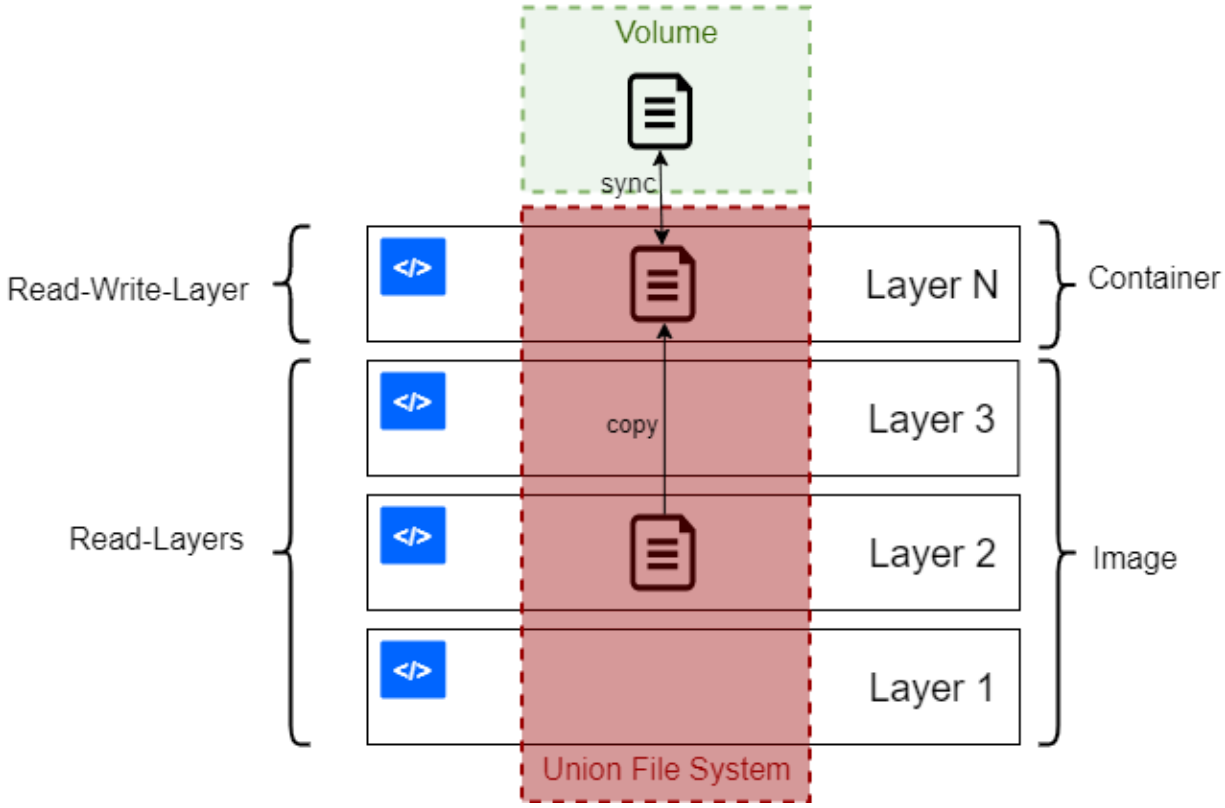
Bind mounting is a method of mounting a host file or directory into a container. It allows you to access host files from within a container, and also make changes to the host files from within the container. The main difference between volumes and bind mounts is that bind mounts can be created from any directory on the host machine and can be mounted to any location within the container's filesystem. With volumes, you are limited to directories within Docker's own storage driver, but the advantage of this is that Docker manages the lifecycle of volumes for you, so if a container is deleted, the volume still persists.

A docker container runs the software stack defined in an image. Images are made of a set of read-only layers that work on a file system called the Union File System. When we start a new container, Docker adds a read-write layer on the top of the image layers allowing the container to run as though on a standard Linux file system.

In a computer, a file system -- sometimes written filesystem -- is the way in which files are named and where they are placed logically for storage and retrieval. Without a file system, stored information wouldn't be isolated into individual files and would be difficult

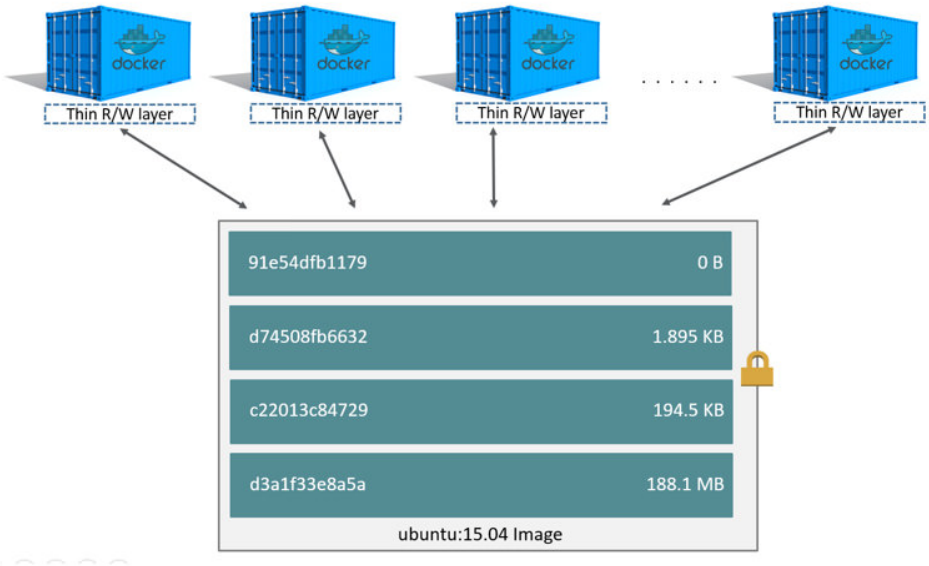
to identify and retrieve. As data capacities increase, the organization and accessibility of individual files are becoming even more important in data storage.

So, any file change inside the container creates a working copy in the read-write layer. However, **when the container is stopped or deleted, that read-write layer is lost**.



The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.

Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.




```
volumes:
  my_mariadb:
    driver: local
    driver_opts:
      type: none
      device: /home/odakhch/Desktop/data/my_mariadb
      o: bind
```

This is a YAML definition of a Docker volume in a Docker Compose file. The volume is named "my_mariadb" and is using the "local" driver.

the `driver: local` option in a volume definition refers to the storage driver that should be used to create the volume. The `local` driver indicates that the volume should be stored on the host machine, rather than on a remote networked storage system.

The "driver_opts" section specifies options for the volume driver.

In this case, the options are:

- "type": "none" specifies the type of the device as "none", which means that the volume is not associated with a named volume or a data container.
- "device": "/home/odakhch/Desktop/data/my_mariadb" specifies the path on the host where the data for this volume is stored.
- "o": "bind" specifies the mount type as "bind", which means that the host file or directory is mounted directly into the container.

This volume definition creates a Docker volume that uses the host file system to persist data for a MariaDB database, mapping the host directory "/home/odakhch/Desktop/data/my_mariadb" to the container's filesystem.

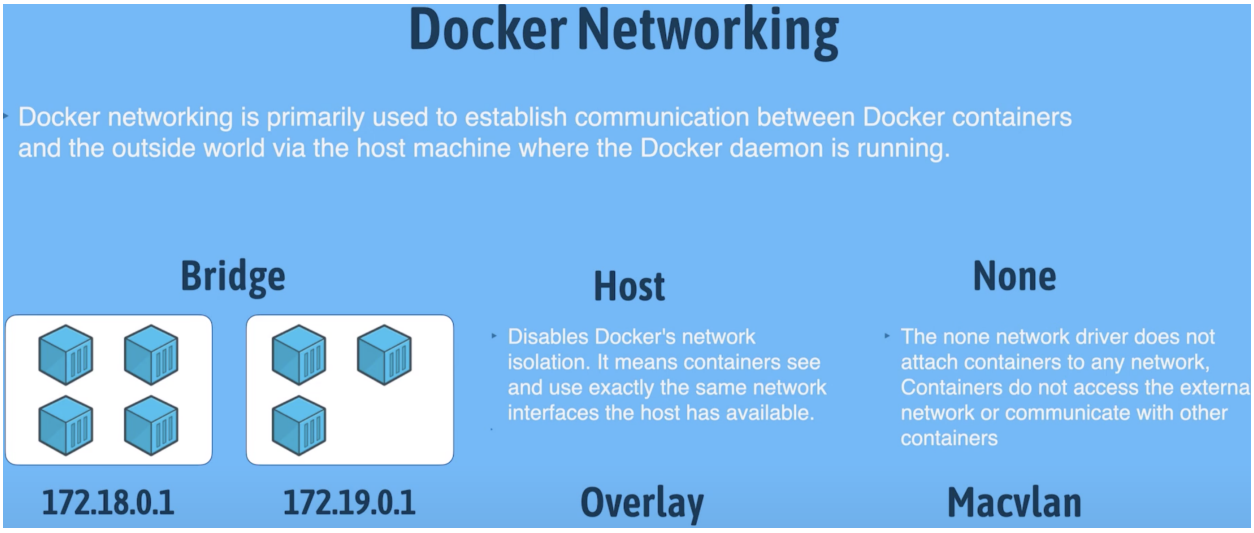
Why a Docker container terminates :

- **The main process inside the container has ended successfully:** This is the most common reason for a Docker container to stop! When the process running inside your container ends, the container will exit.

  Here are a couple of examples:

  - You run a container, which runs a shell script to perform some tasks. When the shell script completes, the container will exit, because there's nothing left for the container to run.
  - You run a utility which is packaged as a Docker container, like the Busybox or Maven images. When the utility finishes, the container exits.

- **You're running a shell in a container, but you haven't assigned a terminal:** If you're running a container with a shell (like `bash`) as the default command, then the container will exit immediately if you haven't attached an interactive terminal. If there's no terminal attached, then your shell process will exit, and so the container will exit. You can stop this by adding `-interactive --tty` (or just `it`) to your `docker run ...` command, which will let you type commands into the shell.



Docker networking is the process of connecting Docker containers to each other and to external networks. It is an essential aspect of Docker as it enables communication between containers and external resources, allowing you to build and run multi-container applications.

Docker provides a default network named "bridge" that is created automatically when you install Docker. This network acts as an isolated network for containers and provides basic connectivity. You can also create custom networks, such as overlay networks, to connect containers across multiple Docker hosts, or use network plugins to connect to external networks such as a physical network.

In the context of Docker, a `bridge` network is a type of network that connects containers running on the same host. The `bridge` driver is the default network driver in Docker and is used to create a virtual network that provides isolation between containers running on the same host.

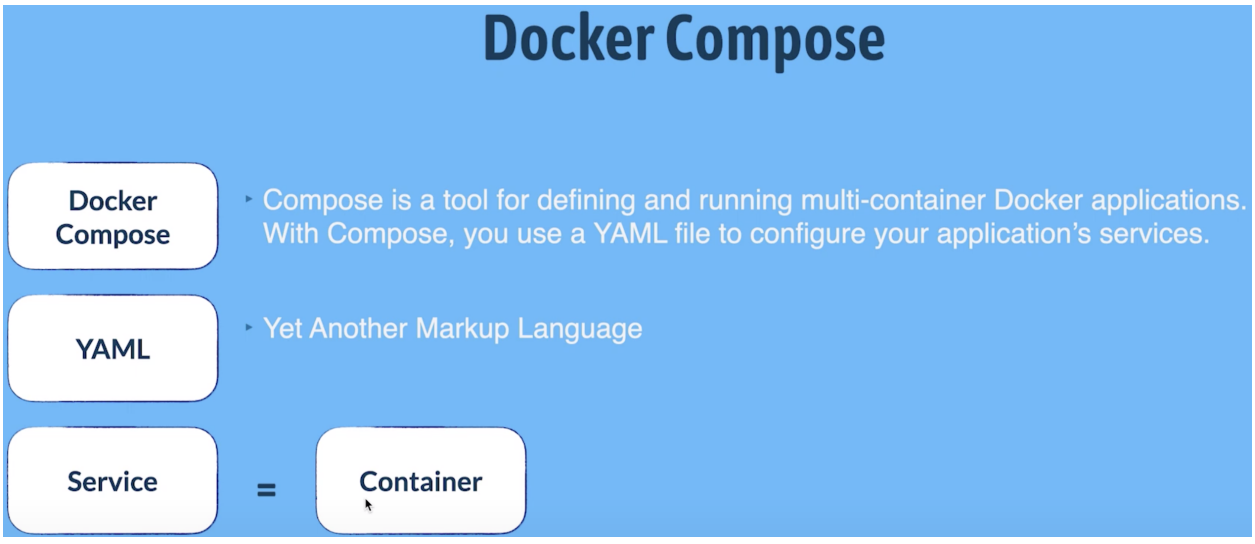
Each container can be connected to one or more networks, and each network can have one or more containers connected to it. Docker also provides network-level tools such as DNS resolution and load balancing, making it easy to manage network communication between containers.

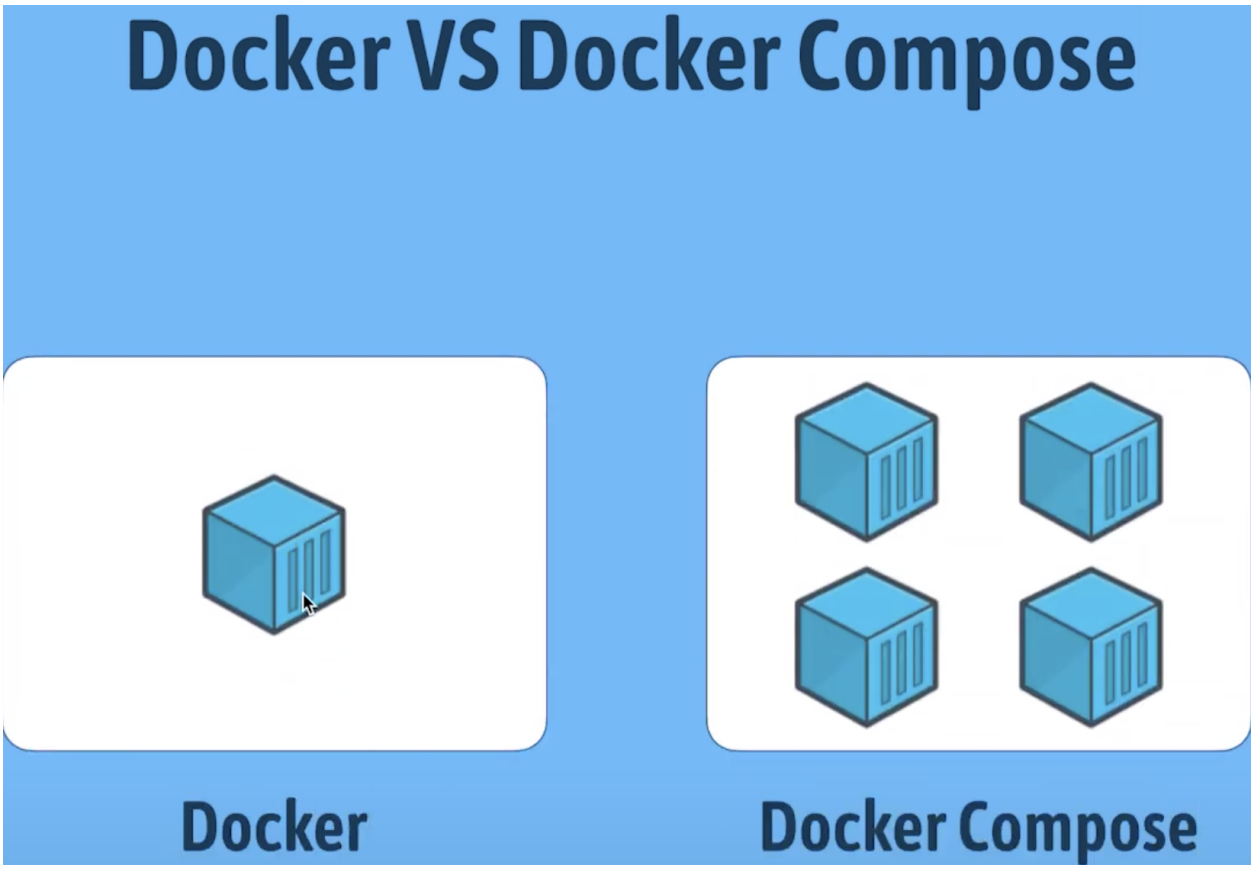
docker network ls: display network

docker network inspect [network] : information about network

docker network create —driver [type_network] [name_network] : create network

docker network [cmd-enter] : more information





Docker Compose is a tool for defining and running multi-container Docker applications. With Docker Compose, you can define an entire application stack, including multiple services and their dependencies, in a single file called a `docker-compose.yml` file. You can then use the `docker-compose` command to start and stop the entire application stack, as well as perform other tasks such as building images and managing containers.

Docker Compose *depends_on*

***depends_on* is a Docker Compose keyword to set the order in which services must start and stop.**

For example, suppose we want our web application, which we'll build as a *web-app* image, to start after our Postgres container.

Docker will pull the images and run the containers based on the given dependencies. So, in this case, the Postgres container is the first in the queue to run.

However, there are limitations because *depends_on* doesn't explicitly wait for dependencies to be ready.

Let's imagine our web application needs to run some migrations scripts at startup. If the database isn't accepting connections, although the Postgres service has started correctly, we can't execute any script.

However, we can avoid this if we control the startup or shutdown order using specific tools or our own managed scripting.

Docker Compose *links*

***links* instructs Docker to link containers over a network. When we link containers, Docker creates environment variables and adds containers to the known hosts list so they can discover each other.**

We'll check out a simple Docker example running a Postgres container and link it to our web application.

First, let's run our Postgres container:

```
docker run -d --name db -p 5342:5342 postgres:latestCopy
```

Then, we link it to our web application:

```
docker run -d -p 8080:8080 --name web-app --link dbCopy
```

Let's convert our example to Docker Compose:

```
services:
  db:
    image: postgres:latest
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    ports:
      - 5432:5432
  web-app:
    images: web-app:latest
    ports:
      - 8080:8080
    links:
      - db
```

Docker Compose *network*

We can find Docker *links* still in use. However, Docker Compose deprecates it since version 2 because of the introduction of the *network*.

This way, we can link applications with complex networking, for example, overlay networks.

However, in a standalone application, we can typically use a bridge as the default when we don't specify a network.

Difference Between Docker *links* and *depends_on*

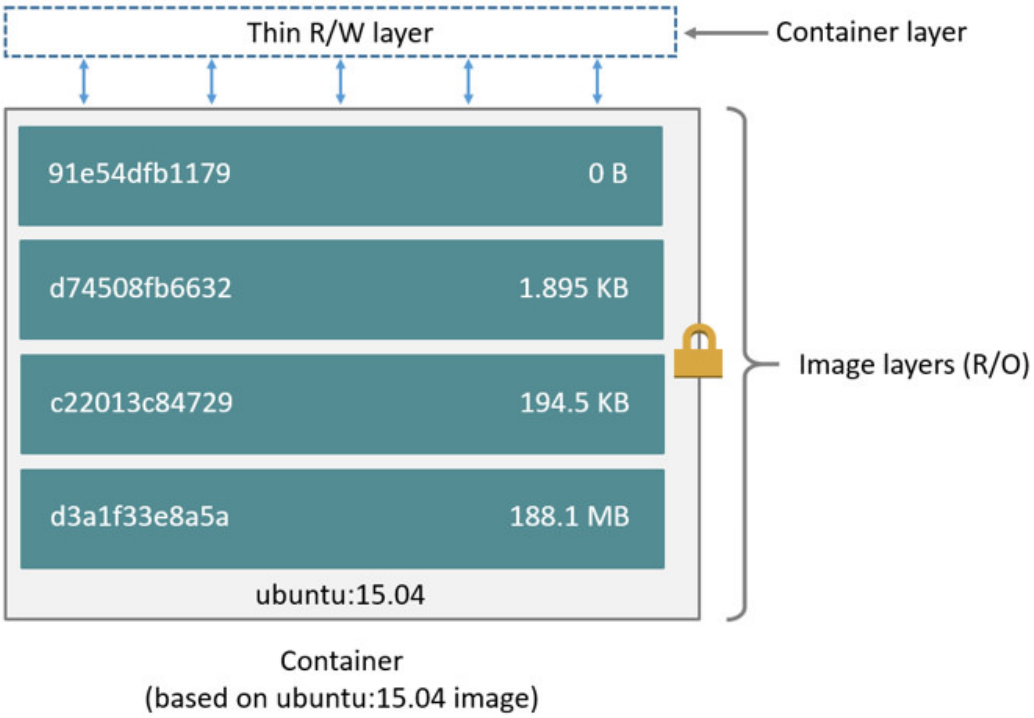
Although they involve expressing dependencies, Docker *links* and *depends_on* have different meanings.

While *depends_on* indicates the order in which the services must start and stop, the *links* keyword handles the communication of containers over a network.

Docker images and layers

When Docker runs a container, it runs an *image* inside it. This image is usually built by executing Docker instructions, which add *layers* on top of existing image or *OS distribution*. *OS distribution* is the initial image and every added layer creates a new image.

Final Docker *image* reminds an onion with OS distribution inside and a number of layers on top of it. For example, your image can be built by installing a number of deb packages and your application on top of Ubuntu 14.04 distribution.



more information : https://codewithyury.com/docker-run-vs-cmd-vs-entrypoint/

Kernel :

A Kernel is a computer program that is the heart and core of an Operating System. Since the Operating System has control over the system so, the Kernel also has control over everything in the system. It is the most important part of an Operating System. Whenever a system starts, the Kernel is the first program that is loaded after the bootloader because the Kernel has to handle the rest of the thing of the system for the Operating System. The Kernel remains in the memory until the Operating System is shut-down.

The Kernel is responsible for low-level tasks such as disk management, memory management, task management, etc. It provides an interface between the user and the hardware components of the system. When a process makes a request to the Kernel, then it is called System Call.

A Kernel is provided with a protected Kernel Space which is a separate area of memory and this area is not accessible by other application programs. So, the code of the Kernel is loaded into this protected Kernel Space. Apart from this, the memory used by other

applications is called the User Space. As these are two different spaces in the memory, so communication between them is a bit slower.

Functions of a Kernel

Following are the functions of a Kernel:

- **Access Computer resource:** A Kernel can access various computer resources like the CPU, I/O devices and other resources. It acts as a bridge between the user and the resources of the system.
- **Resource Management:** It is the duty of a Kernel to share the resources between various process in such a way that there is uniform access to the resources by every process.
- **Memory Management:** Every process needs some memory space. So, memory must be allocated and deallocated for its execution. All these memory management is done by a Kernel.
- **Device Management:** The peripheral devices connected in the system are used by the processes. So, the allocation of these devices is managed by the Kernel.

Kernel Mode and User Mode

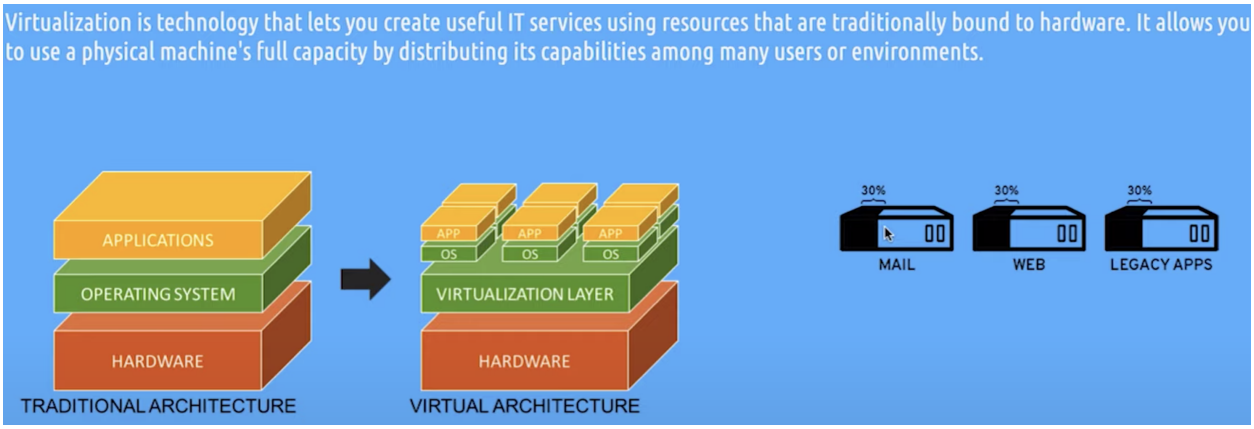
There are certain instructions that need to be executed by Kernel only. So, the CPU executes these instructions in the Kernel Mode only. For example, memory management should be done in Kernel-Mode only. While in the User Mode, the CPU executes the processes that are given by the user in the User Space.

Virtualization :

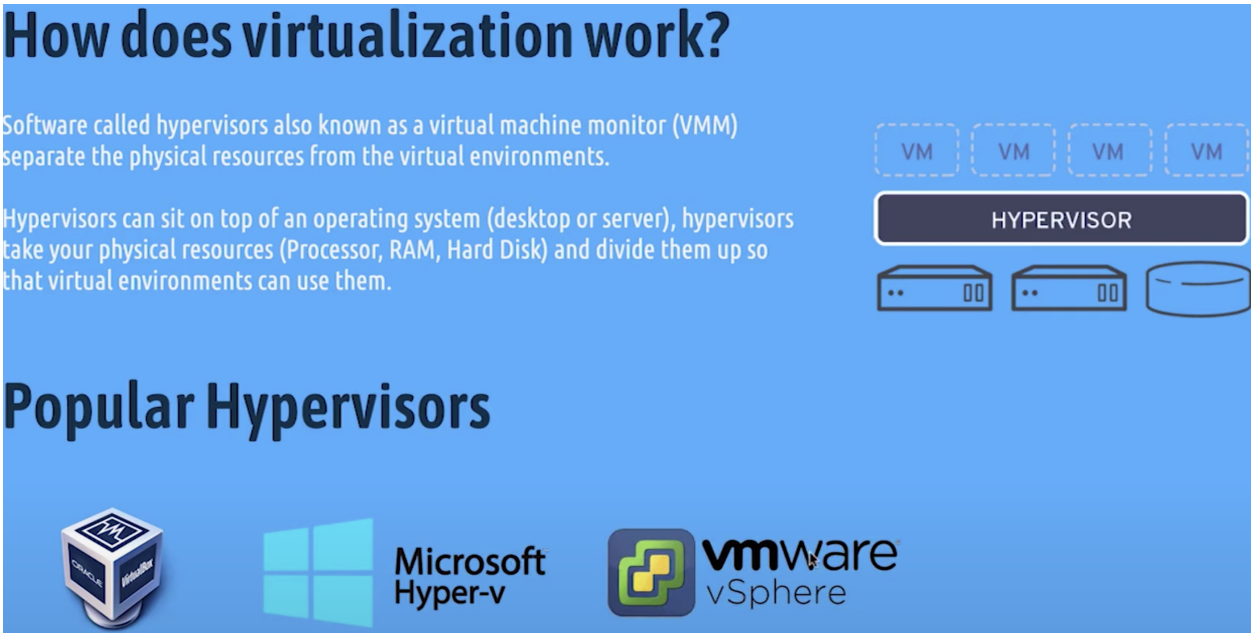
Virtualization in computing is the process of simulating hardware and software, such as computers, operating systems, storage, and networking and it does it in a virtual or software environment.

Virtualization is a technique of how to separate a service from the underlying physical delivery of that service. It is the process of creating a virtual version of something like computer hardware. It was initially developed during the mainframe era. It involves using specialized software to create a virtual or software-created version of a computing resource rather than the actual version of the same resource. With the help of Virtualization, multiple operating systems and applications can run on same machine

and its same hardware at the same time, increasing the utilization and flexibility of hardware.

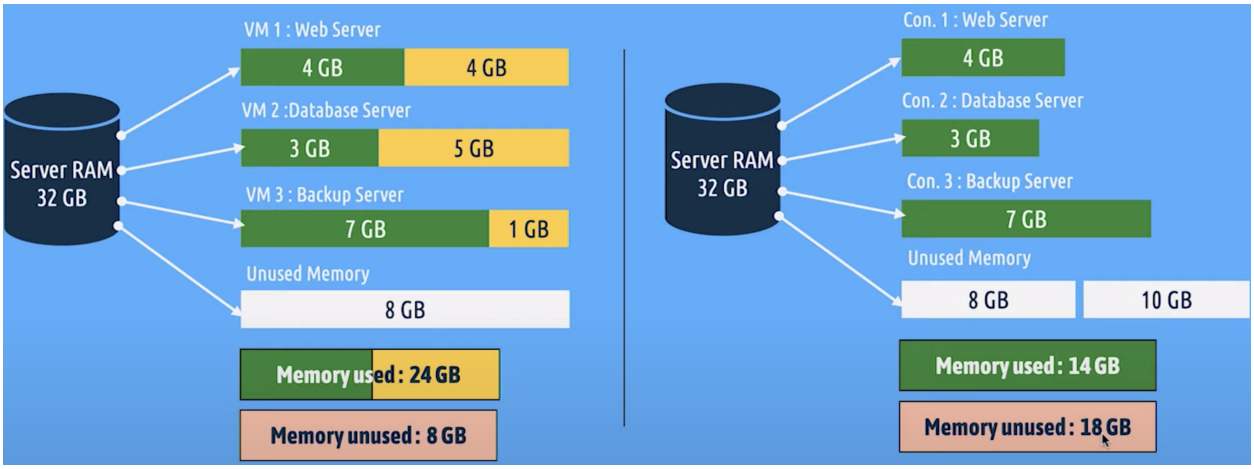


hypervisor : also known as a virtual machine monitor or VMM, is software that creates and runs virtual machines (VMs). A hypervisor allows one host computer to support multiple guest VMs by virtually sharing its resources, such as memory and processing.



Virtual Machine (VM) (**Hardware virtualization**) is a compute resource that uses software instead of a physical computer to run programs and deploy apps. One or more virtual “guest” machines run on a physical “host” machine. Each virtual machine runs its own operating system and functions separately from the other

VMs, even when they are all running on the same host. This means that, for example, a virtual MacOS virtual machine can run on a physical PC.



▼ Hard disk file type

VDI is the native format of VirtualBox. Other virtualization software generally don't support VDI, but it's pretty easy to convert from VDI to another format, especially with `qemu-img convert`.

VMDK is developed by and for VMWare, but VirtualBox and QEMU (another common virtualization software) also support it. *This format might be the the best choice for you because you want wide compatibility with other virtualization software.*

VHD is the native format of Microsoft Virtual PC. Windows Server 2012 introduced VHDX as the successor to VHD, but VirtualBox does not support VHDX.

▼ **HTTP**

is a protocol for fetching resources such as HTML documents. It is the foundation of any data exchange on the Web and it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser. A complete document is reconstructed from the different sub-documents fetched, for instance, text, layout description, images, videos, scripts, and more, and is the protocol used to transfer data over the Web .

HTTP (Hypertext Transfer Protocol) is an application layer protocol that operates on top of the TCP/IP (Transmission Control Protocol/Internet Protocol) suite. HTTP

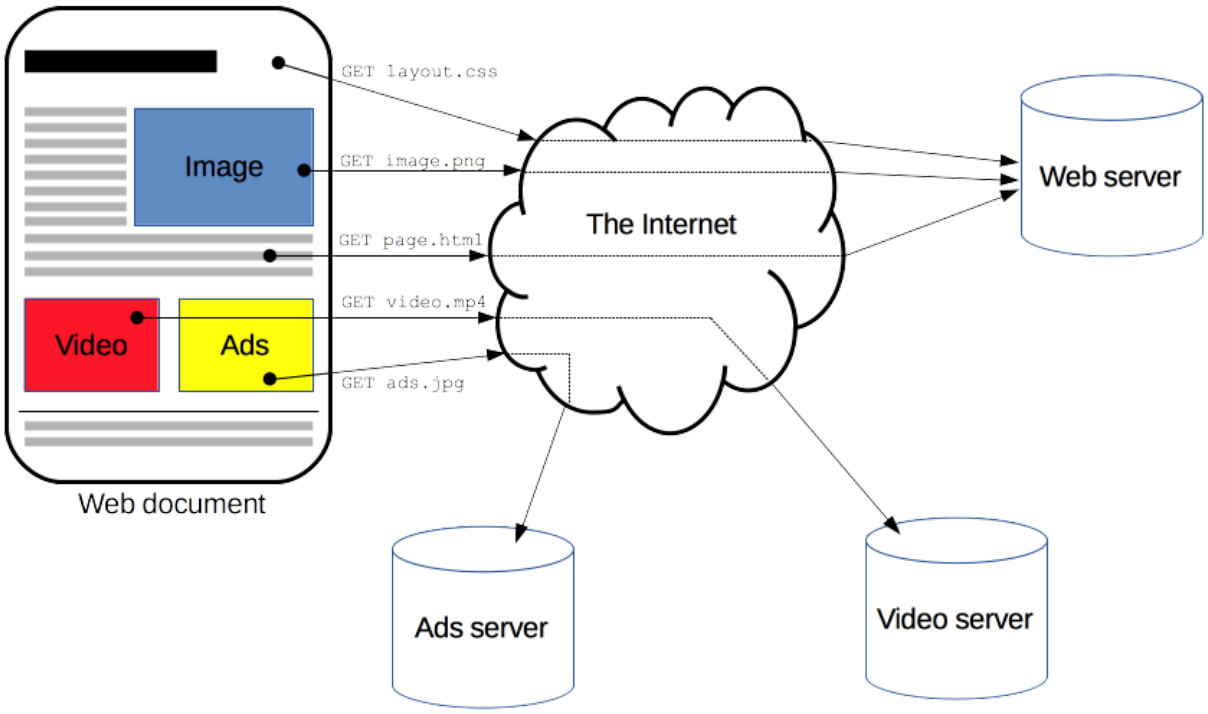
uses TCP/IP as the underlying transport protocol for communication between clients (such as web browsers) and servers.

Here's how HTTP and TCP/IP work together:

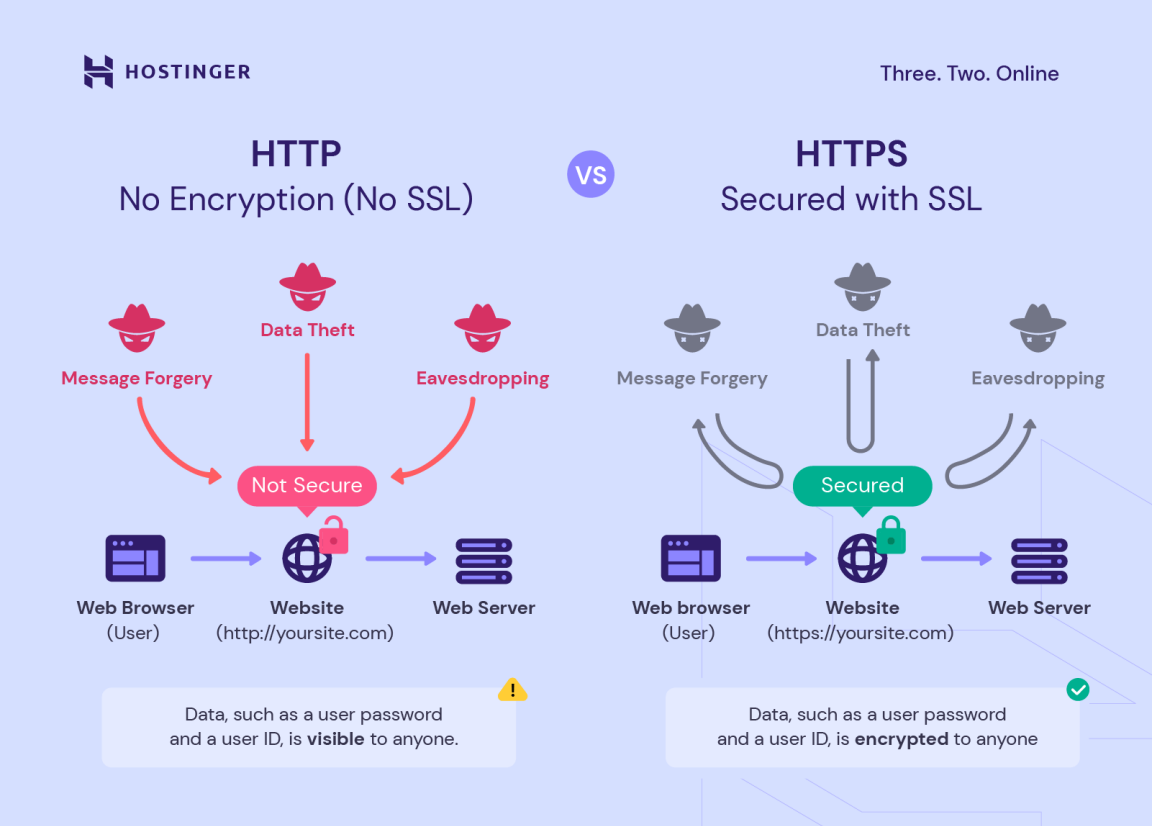
1. TCP Connection: When a client initiates an HTTP request, a TCP connection is established between the client and the server. The client sends a TCP SYN packet to the server, and the server responds with a SYN-ACK packet. Finally, the client sends an ACK packet to complete the three-way handshake and establish the TCP connection.
2. HTTP Request: Once the TCP connection is established, the client sends an HTTP request to the server over the TCP connection. The HTTP request contains information such as the method (GET, POST, etc.), the requested resource (URL), headers, and sometimes a request body.
3. TCP Reliability: The TCP protocol ensures the reliable delivery of data between the client and server. It breaks down the HTTP request into smaller packets and adds sequence numbers to each packet. The server acknowledges the receipt of each packet, and the client retransmits any packets that are not acknowledged within a certain timeframe.
4. HTTP Response: The server processes the received HTTP request, performs the necessary operations, and generates an HTTP response. The response includes a status code (indicating the success or failure of the request), headers, and the requested data or resource.
5. TCP Data Transfer: The server sends the HTTP response back to the client over the established TCP connection. The response is divided into TCP packets, and the server waits for acknowledgments from the client for each packet.
6. TCP Connection Termination: Once the HTTP response is fully transferred, the client and server may choose to terminate the TCP connection. This is done by exchanging FIN packets, initiating a four-way handshake to gracefully close the connection.

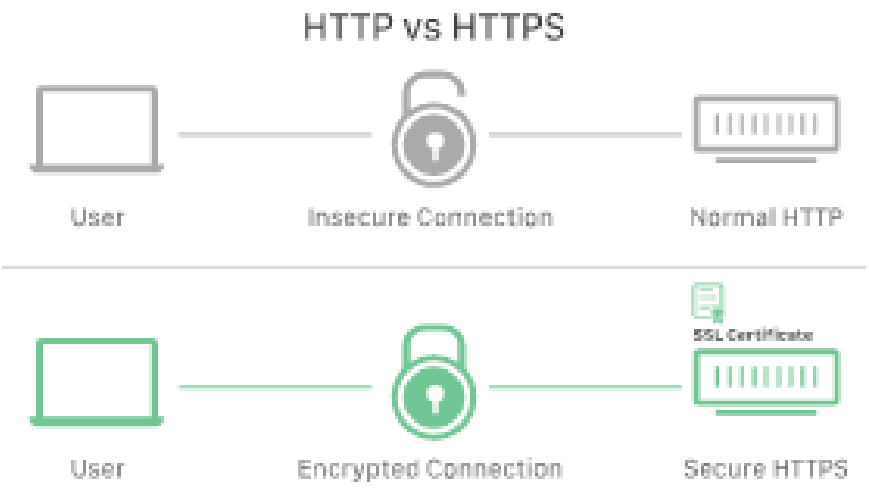
By utilizing TCP/IP as the transport layer, HTTP benefits from TCP's reliable data delivery, congestion control mechanisms, and connection management capabilities.

TCP ensures that the HTTP data is transmitted accurately and in the correct order, even in the presence of network congestion or packet loss.



Clients and servers communicate by exchanging individual messages (as opposed to a stream of data). The messages sent by the client, usually a Web browser, are called *requests* and the messages sent by the server as an answer are called *responses*.





SSL (Secure Sockets Layer) : In order to provide a high degree of privacy, SSL encrypts data that is transmitted across the web. This means that anyone who tries to intercept this data will only see a garbled mix of characters that is nearly impossible to decrypt.

SSL initiates an **authentication** process called a handshake between two communicating devices to ensure that both devices are really who they claim to be.

SSL also digitally signs data in order to provide **data integrity**, verifying that the data is not tampered with before reaching its intended recipient.

There have been several iterations of SSL, each more secure than the last. In 1999 SSL was updated to become TLS.

Why is SSL/TLS important?

Originally, data on the Web was transmitted in plaintext that anyone could read if they intercepted the message. For example, if a consumer visited a shopping website, placed an order, and entered their credit card number on the website, that credit card number would travel across the Internet unconcealed.

SSL was created to correct this problem and protect user privacy. By encrypting any data that goes between a user and a web server, SSL ensures that anyone who intercepts the data can only see a scrambled mess of characters. The consumer's credit card number is now safe, only visible to the shopping website where they entered it.

SSL also stops certain kinds of cyber attacks: It authenticates web servers, which is important because attackers will often try to set up fake websites to trick users and steal data. It also prevents attackers from tampering with data in transit, like a tamper-proof seal on a medicine container.

Are SSL and TLS the same thing?

SSL is the direct predecessor of another protocol called TLS (Transport Layer Security). In 1999 the Internet Engineering Task Force (IETF) proposed an update to SSL. Since this update was being developed by the IETF and Netscape was no longer involved, the name was changed to TLS. The differences between the final version of SSL (3.0) and the first version of TLS are not drastic; the name change was applied to signify the change in ownership.

Since they are so closely related, the two terms are often used interchangeably and confused. Some people still use SSL to refer to TLS, others use the term "SSL/TLS encryption" because SSL still has so much name recognition.

Is SSL still up to date?

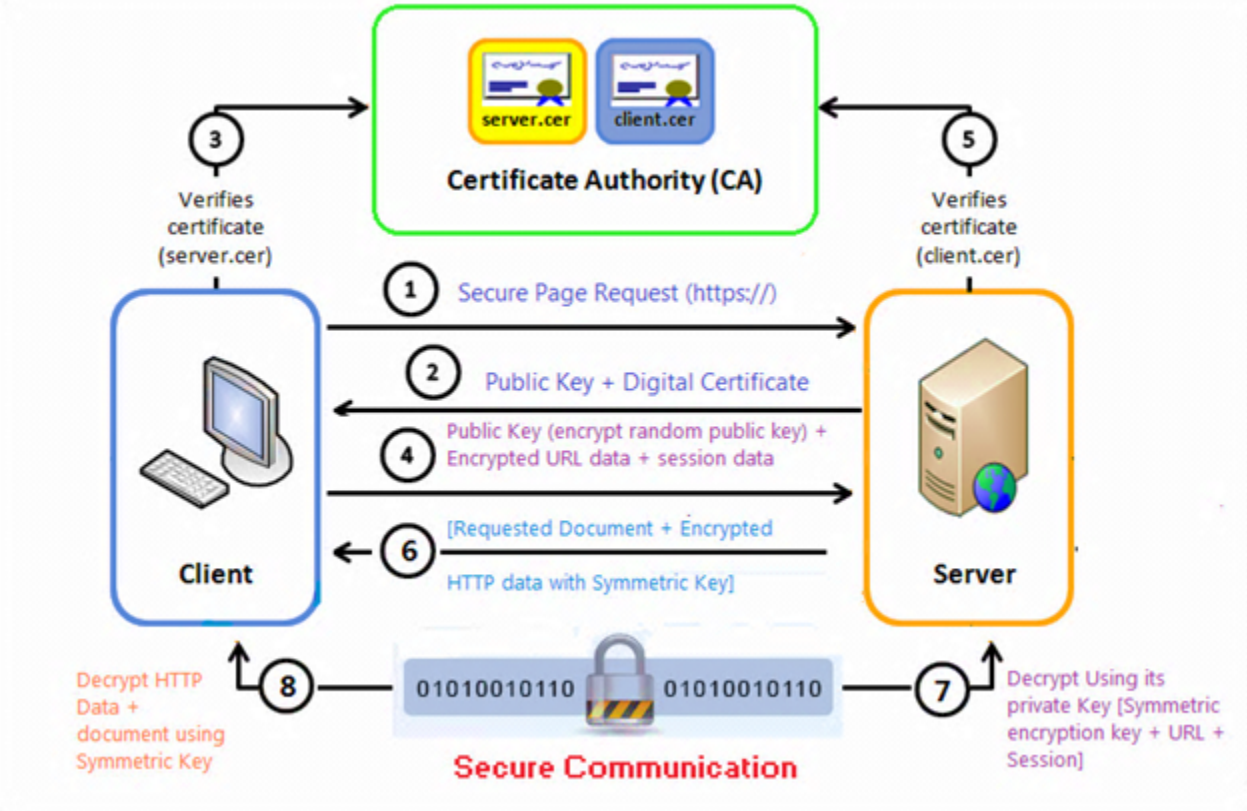
SSL has not been updated since SSL 3.0 in 1996 and is now considered to be deprecated. There are several known vulnerabilities in the SSL protocol, and security experts recommend discontinuing its use. In fact, most modern web browsers no longer support SSL at all.

TLS is the up-to-date encryption protocol that is still being implemented online, even though many people still refer to it as "SSL encryption." This can be a source of confusion for someone shopping for security solutions. The truth is that any vendor offering "SSL" these days is almost certainly providing TLS protection, which has been an industry standard for over 20 years. But since many folks are still searching for "SSL protection," the term is still featured prominently on many product pages.

What is an SSL certificate?

SSL can only be implemented by websites that have an SSL certificate (technically a "TLS certificate"). An SSL certificate is like an ID card or a badge that proves someone is who they say they are. SSL certificates are stored and displayed on the Web by a website's or application's server.

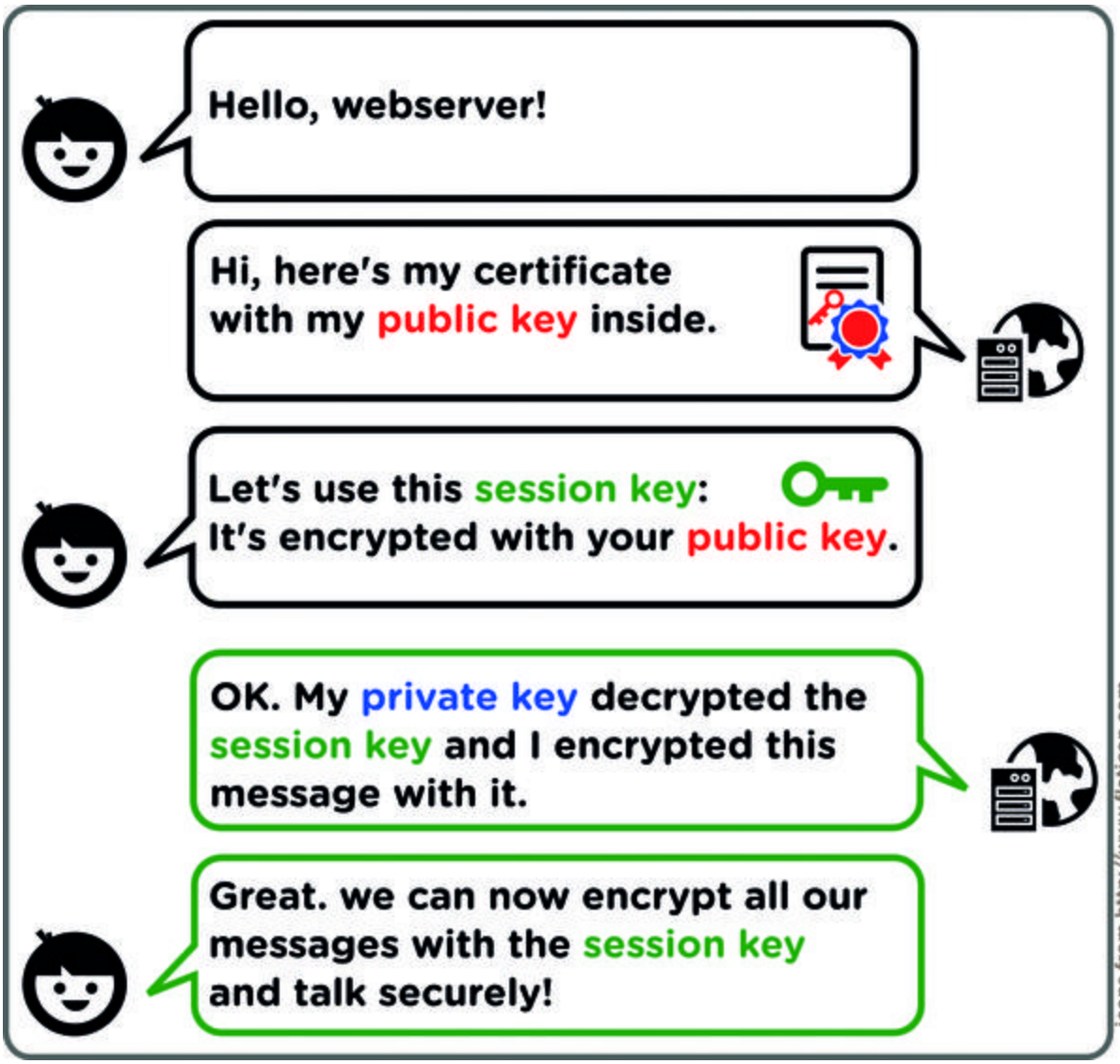
One of the most important pieces of information in an SSL certificate is the website's public key. The public key makes encryption and authentication possible. A user's device views the public key and uses it to establish secure encryption keys with the web server. Meanwhile the web server also has a private key that is kept secret; the private key decrypts data encrypted with the public key.



To enable HTTPS on your website, you need to get a certificate (a type of file) from a Certificate Authority (CA). Let's Encrypt is a CA. In order to get a certificate for your website's domain from Let's Encrypt, you have to demonstrate control over the domain. With Let's Encrypt, you do this using software that uses the ACME protocol which typically runs on your web host.

What's Certbot?

Certbot is a free, open source software tool for automatically using Let's Encrypt certificates on manually-administrated websites to enable HTTPS.



Certificate authorities (CA) are responsible for issuing SSL certificates.

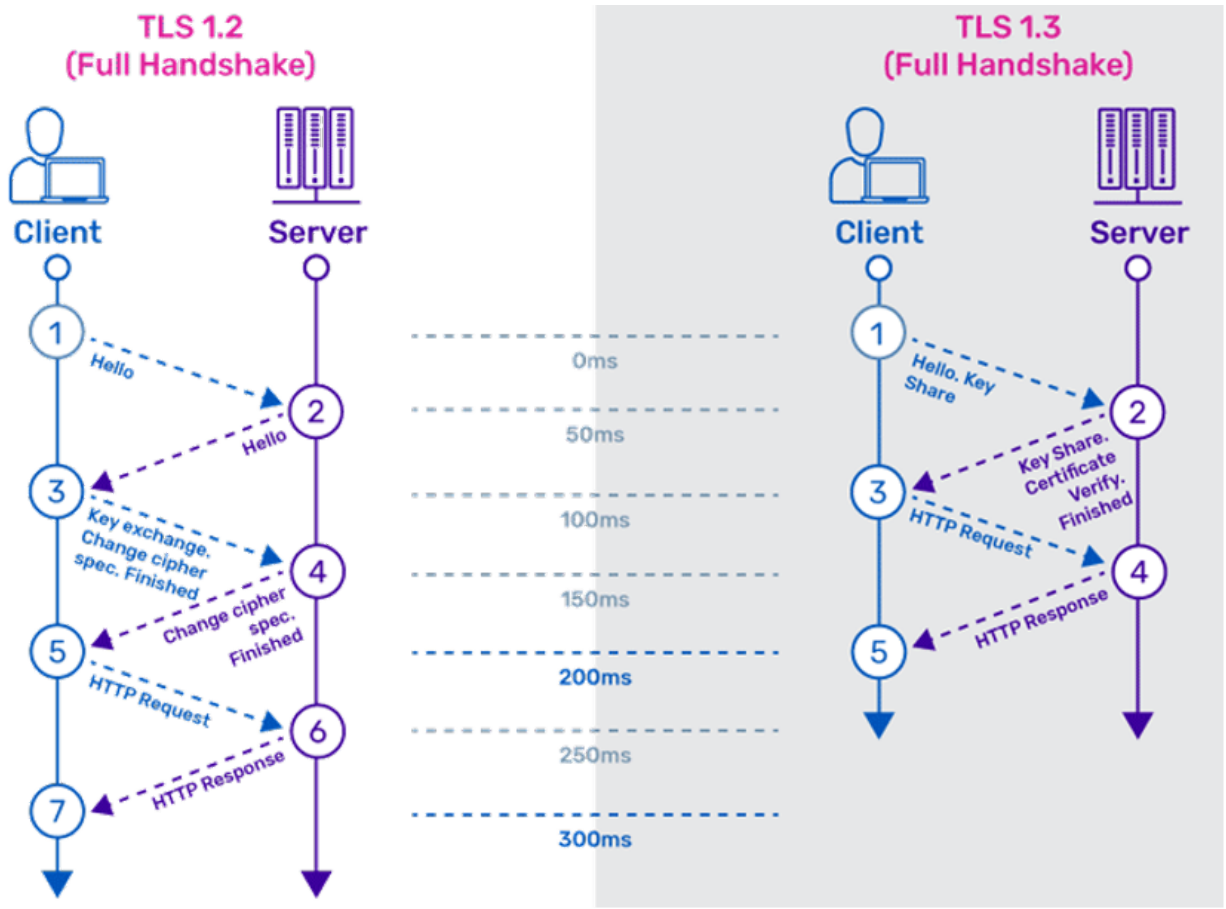
The difference between TLS 1.3 and TLS 1.2 is significant

The most important difference is that a TLS version 1.3 handshake takes less time than a TLS version 1.2 handshake. TLS 1.3 benefits include:

- Reduction of round-trip processing, resulting in a faster handshake
- Improvement of latency times by reducing the number of round trips
- Improvement of website performance and user experience due to reduced
- Use of perfect forward secrecy
- Removal of vulnerable algorithms and ciphers

Secure client-server connections are established by what is commonly referred to as the SSL/TLS handshake. The handshake involves a series of steps that require verification and authentication prior to establishing the secure connection between the client and the server. Essentially, the handshake creates a secure tunnel for communication over the Internet.

The TLS 1.2 handshake involves multiple communications or round trips between the server and client before finalizing a secure connection, imposing unnecessary performance and network overhead. A roundtrip results in a slower connection between the client and the server. TLS 1.3 reduces the number of roundtrips during the handshake. The shorter handshake results in faster secure connections. It also improves HTTPS performance by reducing page load times on mobile devices, which reduces latency and improves user experience.



1. **The 'client hello' message:** The client initiates the handshake by sending a "hello" message to the server. The message will include which TLS version the client supports, the cipher suites supported, and a string of random bytes known as the "client random."
2. **The 'server hello' message:** In reply to the client hello message, the server sends a message containing the server's SSL certificate, the server's chosen cipher suite, and the "server random," another random string of bytes that's generated by the server.
3. **Authentication:** The client verifies the server's SSL certificate with the certificate authority that issued it. This confirms that the server is who it says it is, and that the client is interacting with the actual owner of the domain.

   A domain is the name of a website, a URL is how to find a website, and a website is what people see and interact with when they get there. In other words, when you buy a domain, you have purchased the name for your site, but you still need to build the website itself.

4. **The premaster secret:** The client sends one more random string of bytes, the "premaster secret." The premaster secret is encrypted with the public key and can only be decrypted with the private key by the server. (The client gets the public key from the server's SSL certificate.)
5. **Private key used:** The server decrypts the premaster secret.
6. **Session keys created:** Both client and server generate session keys from the client random, the server random, and the premaster secret. They should arrive at the same results.
7. **Client is ready:** The client sends a "finished" message that is encrypted with a session key.
8. **Server is ready:** The server sends a "finished" message encrypted with a session key.
9. **Secure symmetric encryption achieved:** The handshake is completed, and communication continues using the session keys.

What is public key cryptography?

Public key cryptography is a method of encrypting or signing data with two different keys and making one of the keys, the public key, available for anyone to use. The other key is known as the private key. Data encrypted with the public key can only be decrypted with the private key. Because of this use of two keys instead of one, public key cryptography is also known as asymmetric cryptography. It is widely used, especially for TLS/SSL, which makes HTTPS possible.

What is a cryptographic key?

In cryptography, a key is a piece of information used for scrambling data so that it appears random; often it's a large number, or string of numbers and letters. When unencrypted data, also called plaintext, is put into a cryptographic algorithm using the key, the plaintext comes out the other side as random-looking data. However, anyone with the right key for decrypting the data can put it back into plaintext form.

For example, suppose we take a plaintext message, "hello," and encrypt it with a key; let's say the key is "2jd8932kd8." Encrypted with this key, our simple "hello" now reads

"X5xJCSycg14=", which seems like random garbage data. However, by decrypting it with that same key, we get "hello" back.

The original data is known as the *plaintext*, and the data after the key encrypts it is known as the *ciphertext*.

Plaintext + key = ciphertext:

```
hello + 2jd8932kd8 = X5xJCSycg14=
```

Ciphertext + key = plaintext:

```
X5xJCSycg14= + 2jd8932kd8 = hello
```

This is an example of symmetric cryptography, in which only one key is used. In public key cryptography, there would instead be two keys. The public key would encrypt the data, and the private key would decrypt it.

What is TCP?

TCP stands for Transmission Control Protocol a communications standard that enables application programs and computing devices to exchange messages over a network. It is designed to send packets across the internet and ensure the successful delivery of data and messages over networks.

TCP works by dividing data into segments, which are then transmitted over the network, reassembled, and checked for accuracy at the destination. If a segment is lost or damaged during transmission, TCP will automatically retransmit the segment to ensure reliable delivery. This makes TCP a reliable and secure protocol for transmitting data over the internet.

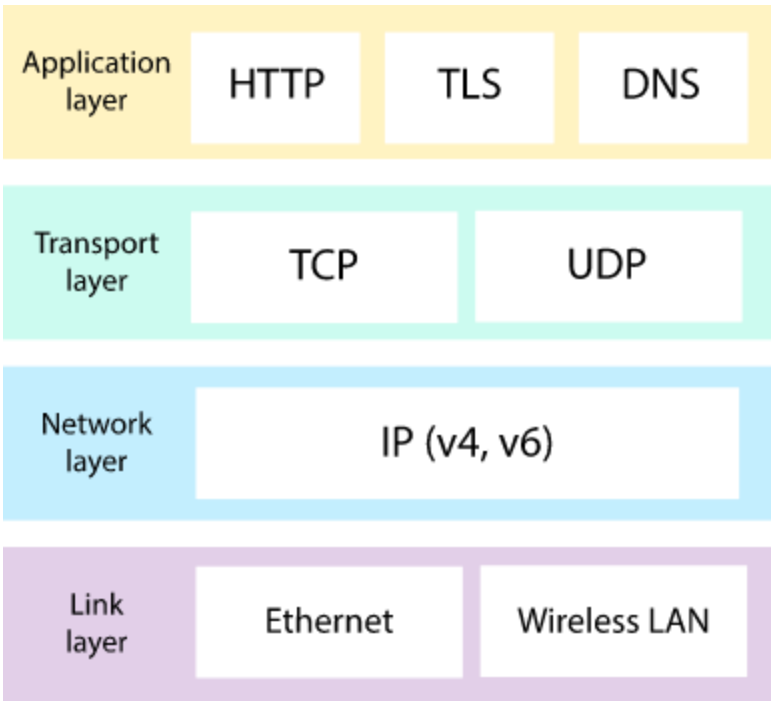
TCP/IP is a versatile protocol suite that supports various application layer protocols, and HTTP is just one of them.

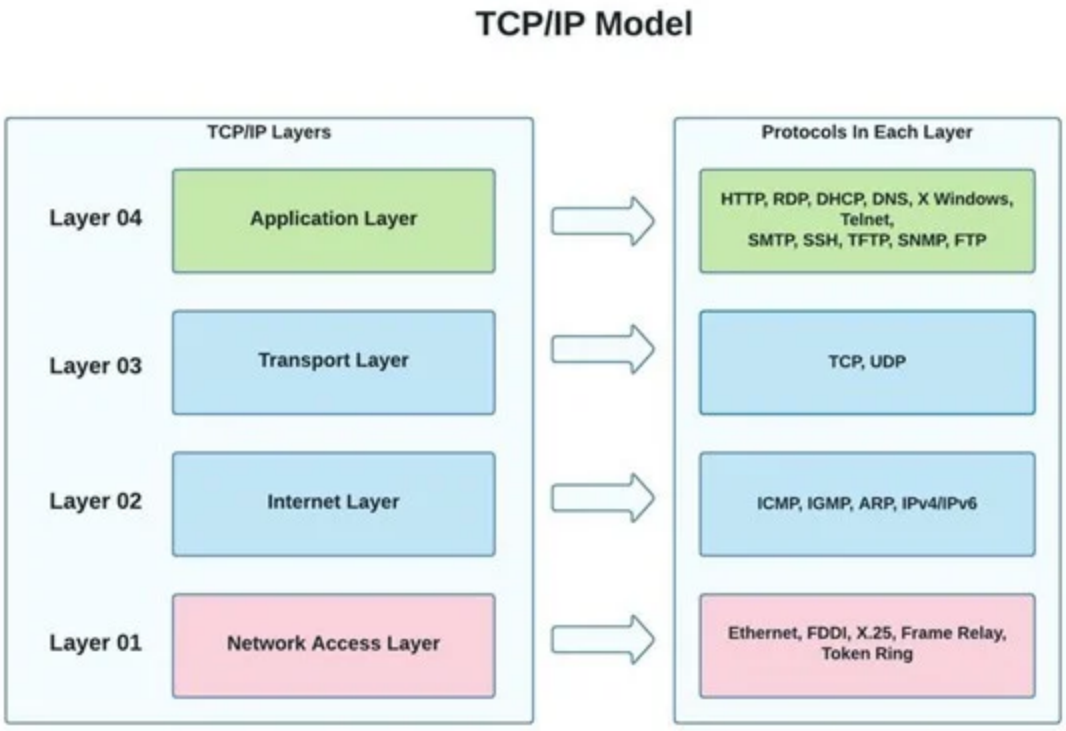
Here are a few examples where TCP/IP is used :

1. FTP (File Transfer Protocol): FTP is an application layer protocol that utilizes TCP/IP for transferring files between a client and a server. It operates on top of TCP to establish connections, transfer data, and manage file operations.
2. SMTP (Simple Mail Transfer Protocol): SMTP is an application layer protocol used for sending and receiving email. It relies on TCP/IP for establishing connections, transmitting email messages, and communicating between mail servers.
3. SSH (Secure Shell): SSH is a network protocol that provides secure remote login and command execution. It utilizes TCP/IP for establishing encrypted connections between clients and servers, enabling secure communication and remote administration.
4. DNS (Domain Name System): DNS is a distributed database system that translates domain names into IP addresses. It operates on top of UDP (User Datagram Protocol), which is another protocol within the TCP/IP suite. DNS uses UDP for lightweight and faster communication, as opposed to the connection-oriented nature of TCP.

These are just a few examples of protocols that rely on TCP/IP but are not based on HTTP. TCP/IP provides the underlying infrastructure for reliable data transmission, packet routing, and network communication. Different application layer protocols can be built on top of TCP/IP to cater to specific needs and services.

So, while HTTP is the dominant protocol for web communication, TCP/IP can be used independently with other protocols to fulfill various networking requirements.


Application layer
HTTP
TLS
DNS
Transport layer
TCP
UDP
Network layer
IP (v4, v6)
Link layer
Ethernet
Wireless LAN


TCP/IP Model
TCP/IP Layers
Protocols In Each Layer
Layer 04
Application Layer
HTTP, RDP, DHCP, DNS, X Windows, Telnet, SMTP, SSH, TFTP, SNMP, FTP
Layer 03
Transport Layer
TCP, UDP
Layer 02
Internet Layer
ICMP, IGMP, ARP, IPv4/IPv6
Layer 01
Network Access Layer
Ethernet, FDDI, X.25, Frame Relay, Token Ring

FTP:

FTP (File Transfer Protocol) is a standard network protocol used for the transfer of files from one host to another over a TCP-based network, such as the Internet. FTP is commonly used to transfer website files from a client computer to a web server, or to download files from a server to a client computer. FTP uses a client-server architecture, where a client establishes a connection to an FTP server and sends commands to initiate the transfer of files. The server responds to the client's requests, and the files are transferred between the two systems. FTP provides basic authentication and data encryption, but is generally considered to be less secure than other file transfer methods, such as SFTP or HTTPS.

The Main Differences Between HTTP and TCP

- HTTP typically uses port 80 – this is the port that the server “listens to” or expects to receive from a Web client. TCP doesn’t require a port to do its job.
- HTTP is faster in comparison to TCP as it operates at a higher speed and performs the process immediately. TCP is relatively slower.
- TCP tells the destination computer which application should receive data and ensures the proper delivery of said data, whereas HTTP is used to search and find the desired documents on the Internet.
- TCP contains information about what data has or has not been received yet, while HTTP contains specific instructions on how to read and process the data once it’s received.
- TCP manages the data stream, whereas HTTP describes what the data in the stream contains.
- TCP operates as a three-way communication protocol, while HTTP is a single-way protocol.

Difference between Port 80 and Port 8080 :

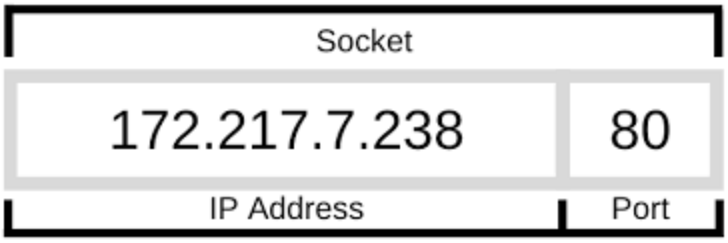
Before understanding the differences between port 80 and port 8080. let's know about Ports and Sockets

Everything in this universe which is connected via the internet has an address that we called it Socket Address. This `Socket Address` is unique whole over the world. Socket

Address is a combination of `IP address` and `port number`. Socket uniquely identifies a TCP connection.

Ip address helps to find out devices on the internet or local network. **Port number** helps to find out a process running within the devices. Port number is considered to be a communication endpoint. With its help, we can find where the message is been requested to a server.

To find out a particular process, the port number must be different. So, we can say Port 80 and Port 8080 are different.



Port 80 is the most commonly used port in TCP.Web Browser by default uses port 80 to send and receive pages from the Web Server. This site: ourtechroom.com seems to not have a port, but the browser automatically requests to web server as ourtechroom.com:80, and the web server understands it is requesting for port 80 and responds back the pages or data back to the web browser. So, when you go to the browser and type: ourtechroom.com:80 then it will give the same response as you type ourtechroom.com in the browser.

Port 8080 is the port that is generally used by web servers to make TCP connections if default port 80 is busy. Generally, Port 8080 is chosen by the web server as the best alternative to 80 because it has two 80s and is above the restricted well-known port. So here Port 8080 overrides the default service port which is implied by HTTP protocol, we place a colon(:) after the domain name portion followed by the port 8080 we wish to use.

Note that two-port with the same IP doesn't exist at a time. But two-port in different IP address/hostname exists. Suppose you host your web application on an 8080 port then the user would need to navigate to for example http://xyz.com:8080 for domain xyz.com to access the web page. You can host multiple websites in one server that are using the same port because a whole socket address is distinct.

Port 80 vs Port 8080

Port 80	Port 8080
The default port for HTTP Request	Alternative port for HTTP requests.
When you type just the domain name in the URL it will request port 80 to a web server.	You have to type the domain name along with port 8080 to request the webserver for the process /program to call.
Port 80 is a well-known port.	Port 8080 is a user or registered port
Port 80 is assigned and controlled by IANA	Port 8080 is registered and controlled by IANA but registered by IANA.
Example: ourtechroom.com and ourtechroom.com:80 are the same. Similarly, localhost and locahost:80 are the same.	Example:ourtechroom.com and ourtechroom.com:8080 are different. Similarly, localhost and localhost:8080 are different.
Most of the networks and firewalls do not block traffic going through 80 port	Networks and firewalls block traffic going through 8080 so we must enable rules for requests in the firewall to send/receive a request

What is NGINX?

NGINX is open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more. It started out as a web server designed for maximum performance and stability. In addition to its HTTP server capabilities, NGINX can also function as a proxy server for email (IMAP, POP3, and SMTP) and a reverse proxy and load balancer for HTTP, TCP, and UDP servers.

How Does Nginx Work?

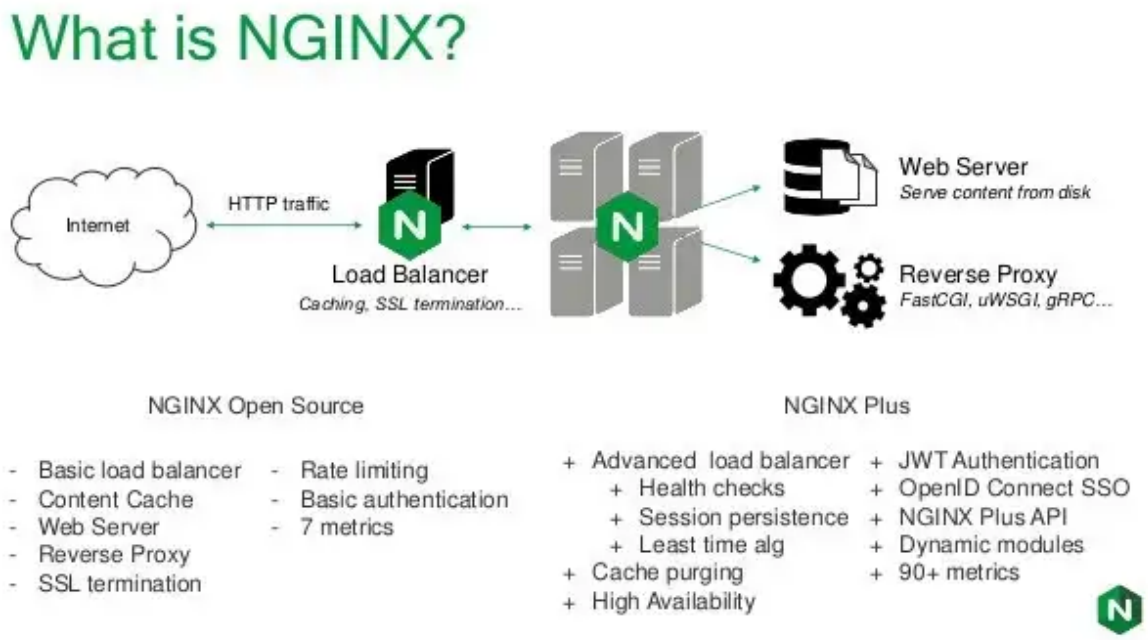
Nginx is built to offer **low memory usage** and high concurrency. Rather than creating new processes for each web request, Nginx uses an asynchronous, event-driven approach where requests are handled in a single thread.

With Nginx, one master process can control multiple worker processes. The master maintains the worker processes, while the workers do the actual processing. Because Nginx is asynchronous, each request can be executed by the worker concurrently without blocking other requests.

NGINX is a web server but commonly used as a reverse proxy. It can be scaled efficiently as a web server as well as a reverse proxy. It does not allow you to allocate a process to a particular connection, but it creates a process pool that can be easily shared among multiple connections within the network. Whenever a request is made, a resource will be allocated to the process resulting in better resource utilization that can easily handle extensive connections.

Advantages of using NGINX

- The written code base is more consistent than other alternatives.
- It provides a friendly configuration format and has a modern design than any other web server alternatives.
- It is event-based and allows you to handle multiple connections without having overhead due to the context switching.
- It uses less memory and resources.
- NGINX makes the website faster and helps them to get a better Google ranking.
- It shows compatibility with commonly-used web applications like ruby, python, Joomla, etc.
- It helps in transforming the dynamic content to static content.
- It helps in handling thousands of concurrent connections at the same time.



The location directive within NGINX server block allows to route request to correct location within the file system. The directive is used to tell NGINX where to look for a resource by including files and folders while matching a location block against an URL.

NGINX Configuration: Understanding Directives

Every NGINX configuration file will be found in the /etc/nginx/ directory, with the main configuration file located in /etc/nginx/nginx.conf **.**

NGINX configuration options are known as “directives”: these are arranged into groups, known interchangeably as **blocks** or **contexts** .

When a # appears before a line, these are comments and NGINX won’t interpret them. Lines that contain directives should end with a semicolon (;). If not, NGINX will be unable to load the configuration properly and report an error.

Below, we’ve included a shortened copy of the /etc/nginx/nginx.conf file included with installations from NGINX repositories. This file begins with four directives:

- **user**
- **worker_processes**
- **error_log**

- **pid**

These exist outside any particular context or block, and are said to be within the **main** context.

Additional directives are found within the **events** and **http** blocks, and these also exist within the **main** context.

File: /etc/nginx/nginx.conf

What is the Http Block?

The **http** block includes directives for web traffic handling, which are generally known as ***universal*** . That's because they get passed on to each website configuration served by NGINX.

What are Server Blocks?

The **http** block shown above features an **include** directive. This informs NGINX where website configuration files can be found.

- When installing from NGINX's official repository, the line will read include /etc/nginx/conf.d/*.conf; just as you can see in the **http** block placed above. Every website hosted with NGINX should feature a unique configuration file in /etc/nginx/conf.d/, and the name will be formatted as example.com.conf.Those sites that have been disabled — not served by NGINX — should be titled example.com.conf.disabled.

No matter the installation source, though, server configuration files feature one or more server blocks for a site. As an example, let's look at the below:

File: /etc/nginx/conf.d/example.com.conf

```
server {
listen [::]:80 default_server;
root /var/www/example.com;
try_files $uri /index.html;
```

try_files : We usually use the Nginx try_files directive to recursively check if files exist in a specific order and serve the file located first.

The try_file directive is in the server and location blocks and specifies the files and directories in which Nginx should check for files if the request to the specified location is received. A typical try_files directive syntax is as:

```
location /
{    try_files $uri $uri/ /default/index.html;
}
```

The **location /** block specifies that this is a match for all locations unless explicitly specified location /<name>

Inside the second block, the try_files means if Nginx receives a request to the URI that matches the block in the location, try the $uri first, and if the file is present, serve the file.

`$uri/` is the same as `$uri` with a trailing `/` appended to it. This is used to check if the request is for a directory rather than a file.

For example, if a request such as **https://linuxhint.com/blocks/io.sh** is received, Nginx will first look for the file inside the / blocks directory and serve the file if available.

The next part (/default/index.html) specifies a fallback option if the file is not in the first param. For example, if the file is not in the / block directory, Nginx will search for the / default directory and serve the file if it exists.

By default, Nginx forbids directory listing, and you will get 403 Forbidden unless you have auto index set to on.

If Nginx fails to find the file in the specified locations, it displays a 404 not found error to the user.

NOTE: Nginx try_files directive recursively searches for files and directories specified from left to right until it finds ones. Specifying this directive in the location / can cause performance issues, especially on sites with massive traffic. Therefore, you should explicitly specify the location block for try_files.

What are Listening Ports?

The listen directive informs NGINX of the hostname/IP and TCP port, so it recognizes where it must listen for HTTP connections.

The argument default_server means that this virtual host will be answering requests on port 80 which don't match the listen statement of a separate virtual host. When it comes to the second statement, this will listen over IPv6 and demonstrate similar behavior.

What is Name-based Virtual Hosting?

The server_name directive enables a number of domains to be served from just one IP address, and the server will determine which domain it will serve according to the request header received.

Generally, you should create one file for each site or domain you wish to host on your server. Let's delve into some examples:

1. Process requests for example.com and www.example.com:
2. The server_name directive can utilize wildcards. *.example.com and .example.com tell the server to process requests for all example.com subdomains:

File: /etc/nginx/conf.d/example.com.conf

```
server_name *.example.com;
server_name .example.com;
```

1. Process requests for all domain names starting with example.:

File: /etc/nginx/conf.d/example.com.conf

```
server_name example.*;
```

With NGINX, you can define server names that are invalid domain names: it utilizes the name from the HTTP header to answer requests regardless of whether the domain name is valid or invalid.

You may find non-domain hostnames helpful if your server is on a LAN or you know all the clients likely to make requests on the server. This encompasses front-end proxy servers with /etc/hosts entries set up for the IP address NGINX is listening on.

What are Location Blocks?

NGINX's location setting helps you set up the way in which NGINX responds to requests for resources inside the server. As the server_name directive informs NGINX

how it should process requests for the domain, location directives apply to requests for certain folders and files (e.g. http://example.com/blog/.) .

Let's consider a few examples:

File: /etc/nginx/sites-available/example.com

```
location / { }
location /blog/ { }
location /planet/blog/ { }
```

These locations are literal string matches and match any part of an HTTP request following the host segment:

Request: http://example.com/

Returns: Let's assume there's a server_name entry for example.com. In this case, the location / directive determines what occurs with this request.

With NGINX, requests are always fulfilled with the most specific match possible:

Request: http://example.com/planet/blog or http://example.com/planet/blog/about/

Returns: This will be fulfilled by the location /planet/blog directive as it's more specific, despite location /planet being a match too.

File: /etc/nginx/sites-available/example.com

```
location ~ IndexPage\.php$ { }
location ~ ^/BlogPlanet(/|/index\.php)$ { }
```

When location directives are followed by a ~ (tilde), NGINX will perform a regular expression match, which is always case-sensitive.

For example, IndexPage.php would be a match with the first of the above examples, while indexpage.php wouldn't.

In the second example, the regular expression ^/BlogPlanet(/|/index\.php)$ { } would match requests for /BlogPlanet/ and /BlogPlanet/index.php but not /BlogPlanet, /blogplanet/, or /blogplanet/index.php. NGINX utilizes Perl Compatible Regular Expressions (PCRE).

What if you prefer matches to be case-insensitive? Well, you should use a tilde followed closely by an asterisk: ~*. You can see the above examples define that NGINX should process requests ending in a certain file extension: the first example determines that files ending in .pl, PL, .cgi, .perl, .Perl, .prl, and .PrL (as well as others) will all be a match for the request.

```
location ^~ /images/IndexPage/ { }
location ^~ /blog/BlogPlanet/ { }
```

When you add a caret and a tilde (^~) to location directives, you're informing NGINX that, should it match a particular string, it should stop searching for more specific matches and utilize these directives here instead.

Beyond this, these directives function as the literal string matches do in the first group. Even if a more specific match comes along at a later point, the settings will be utilized if a request is a match for one of these directives.

```
location ~ \.php$
{
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass wordpress:9000;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param SCRIPT_NAME $fastcgi_script_name;
 }
```

This is an example of a configuration block for Nginx, a popular web server. This specific block is used to configure the handling of PHP files by Nginx.

The `location` directive specifies a URL pattern that this block of configuration applies to. The pattern `~ \.php$` matches any URL that ends with ".php". This means that any request for a URL that ends with ".php" will be handled by the configuration block.

The `fastcgi_split_path_info` directive is used to split the requested URL into the script name and the path information. The regular expression `^(.+\.php)(/.+)$` is used to match the script name (the part of the URL before the path information) and the path information (the part of the URL after the script name). The script name is captured in the first set of parentheses, and the path information is captured in the second set of parentheses.

The `fastcgi_pass` directive is used to specify the address and port of the FastCGI server that will handle the PHP script. In this example, the server is running on the host "wordpress" and is listening on port 9000.

The `fastcgi_index` directive is used to specify the default script file name to use when a directory is requested. In this example, the default script file is "index.php".

The `include fastcgi_params` directive is used to include a file with additional FastCGI parameters. This file typically contains a set of common parameters that are used by most PHP scripts.

The `fastcgi_param` directive is used to set additional parameters for the FastCGI server. In this example, two parameters are set: `SCRIPT_FILENAME` and `SCRIPT_NAME`.

The `SCRIPT_FILENAME` is set to the value of the `$document_root` variable, which contains the root of the document tree, concatenated with the value of the `$fastcgi_script_name` variable, which contains the script name. This tells the FastCGI server the path of the script file to execute.

The `SCRIPT_NAME` is set to the value of the `$fastcgi_script_name` variable. This tells the PHP script the requested script name.

In summary, this configuration block is used to handle PHP files requested by clients in Nginx, it specifies a location pattern for PHP files, splits the requested URL into script name and path information, specify the address and port of the FastCGI server, specifies the default script file name, includes the common parameters, and sets additional parameters for the FastCGI server.

Suppose we have a PHP script located at `/var/www/html/my_script.php`, and a client makes a request to `https://example.com/my_script.php/path/to/data?query=value`.

In this example, the script name is `/my_script.php` and the path information is `/path/to/data`.

The script name is the part of the URL that identifies the PHP script to be executed. In this case, it is the `/my_script.php` part of the URL.

The path information is additional data that is passed to the script as part of the URL. It is the part of the URL that comes after the script name. In this case, it is the `/path/to/data` part of the URL.

With the `fastcgi_split_path_info` directive, the script name and path information can be captured separately and passed as arguments to the PHP script, so the script can use

them in its execution.

It's worth noting that the path information is optional and it depends on the script design and the web application's requirements.

FastCGI :

FastCGI is a protocol that allows web servers to communicate with server-side web applications in a more efficient way than traditional CGI. In traditional CGI, a new process is created for each incoming web request, which can be slow and resource-intensive for the server. With FastCGI, a single process is reused for multiple requests, which reduces the overhead associated with CGI and improves the performance of the web server.
When a web server receives a request for a web page, it checks to see if a FastCGI application is configured to handle that request. If it is, the web server sends the request to the FastCGI application, which processes the request and sends back the response to the web server. The web server then sends the response to the user's browser.

One of the main advantages of FastCGI is that it allows server-side web applications, such as PHP or Python scripts, to be executed more efficiently. It also allows multiple server-side languages to be used on the same web server. This means that web developers can choose the language that best suits the needs of their application, rather than being limited by the capabilities of the web server.

FastCGI is widely supported by popular web servers such as Apache, Nginx and IIS, and it is often used to run server-side web applications, such as WordPress, Drupal, and Joomla.

```
server
{
    listen 443 http2 ssl;
    listen [::]:443 http2 ssl;

    server_name odakhch.42.fr www.odakhch.42.fr;

    ssl_certificate /etc/ssl/certs/odakhch.crt;
    ssl_certificate_key /etc/ssl/private/odakhch.key;

    ssl_protocols TLSv1.3;

    root /var/www/html;
    index index.php index.html index.htm;
```

```
    location / {
    try_files $uri $uri/ /index.php?$args;
    }

    location ~ \.php$ {
        try_files $uri =404;
        include /etc/nginx/fastcgi_params;
        fastcgi_pass wordpress:9000;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_index index.php;
    }
}
```

This configuration block is for an Nginx server. It sets up the server to listen on port 443, which is the standard port for secure HTTPS connections, and to use the HTTP/2 protocol and SSL encryption.

The `listen` directive in Nginx is used to specify the IP addresses and ports on which the Nginx server should listen for incoming requests.

In this example, `listen 443 http2 ssl;` and `listen [::]:443 http2 ssl;` tell Nginx to listen on port 443 for incoming requests using the HTTPS protocol (HTTP over SSL/TLS) and HTTP/2. The first line listens on the IPv4 address (`listen 443 http2 ssl;`), while the second line listens on the IPv6 address (`listen [::]:443 http2 ssl;`).

IPv4 (Internet Protocol version 4) and IPv6 (Internet Protocol version 6) are two different versions of the Internet Protocol (IP), the communication protocol used for transmitting data over the internet.

The main difference between IPv4 and IPv6 is the length of their addresses. IPv4 addresses are 32-bit addresses and can support approximately 4.3 billion unique addresses. On the other hand, IPv6 addresses are 128-bit addresses and can support approximately 340 trillion unique addresses.

Additionally, IPv6 addresses are represented in a different format compared to IPv4 addresses. IPv4 addresses are written in dot-decimal notation (e.g. 192.168.0.1), while IPv6 addresses are written in hexadecimal notation (e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334).

Another difference is that IPv6 supports advanced features such as autoconfiguration, improved security, and larger address space, which makes it more suited for future internet needs compared to IPv4. Many organizations and service providers have started to adopt IPv6 as the number of available IPv4 addresses is running out.

This configuration is intended to provide secure, encrypted communication for incoming requests to the server. The server will use the certificate and private key specified in the `ssl_certificate` and `ssl_certificate_key` directives for SSL/TLS encryption.

The server is configured with two hostnames: `odakhch.42.fr` and `www.odakhch.42.fr`. The SSL certificate and key used by the server are located at `/etc/ssl/certs/odakhch.crt` and `/etc/ssl/private/odakhch.key` respectively. The SSL protocol used is TLSv1.3.

The root directory for the server's web content is set to `/var/www/html`, and the default index files (i.e., the first file that will be served if no specific file is requested) are `index.php`, `index.html`, and `index.htm` in that order.

The `location` block inside the server block defines the behavior of the server when requests are made to specific URL paths. The first `location` block specifies that requests should be handled by trying to serve the requested file directly, or by passing the request to `/index.php` with the URL arguments appended as query parameters.

The second `location` block is a regular expression that matches URLs ending in `.php`. The block specifies that when a request is made for a PHP file, Nginx should try to serve the file and return a 404 error if it cannot be found. The `fastcgi_pass` directive specifies that requests for PHP files should be passed to a FastCGI server running on `wordpress:9000`. The `SCRIPT_FILENAME` and `fastcgi_index` parameters are set to configure the FastCGI server.

In Nginx, `$uri` and `$uri/` are variables that represent the URL path requested by the client.

`$uri` is the exact URL path requested by the client, including the query string if there is one.

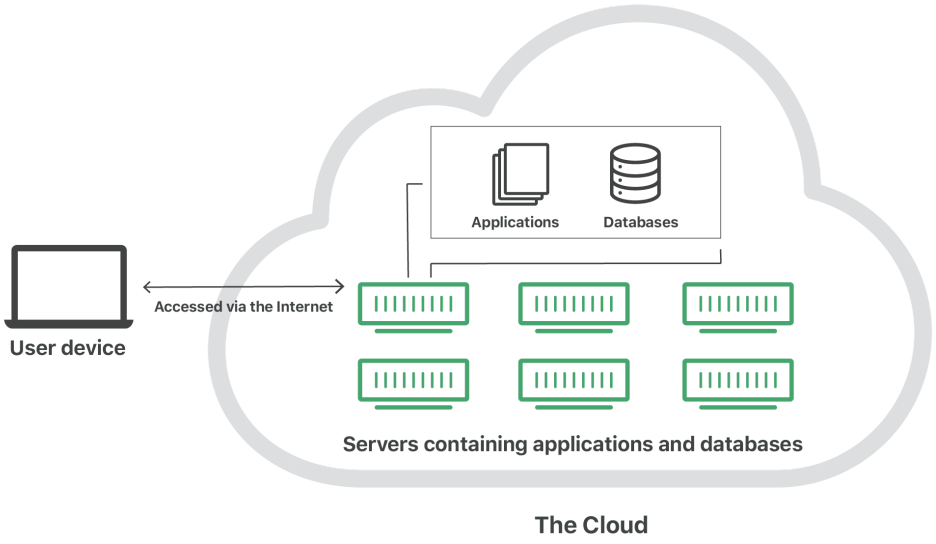
`$uri/` is the same as `$uri` with a trailing `/` appended to it. This is used to check if the request is for a directory rather than a file.

`/index.php$args` is a URL path that specifies that requests should be passed to `index.php` for further processing. The `$args` variable is used to preserve any query string parameters in the original URL so that they can be passed to `index.php` for processing.

For example, if a client requests `/example?foo=bar`, Nginx would set `$uri` to `/example?foo=bar` and `$args` to `?foo=bar`. If the `try_files` directive is set to `try_files $uri $uri/ /index.php$args;`, Nginx would pass the request to `/index.php?foo=bar` for processing.

What is the cloud?

"The cloud" refers to servers that are accessed over the Internet, and the software and databases that run on those servers. Cloud servers are located in data centers , all over the world. By using cloud computing, users and companies do not have to manage physical servers themselves or run software applications on their own machines.



create nginx with TLS : https://mindsers.blog/post/https-using-nginx-certbot-docker/

MariaDb :

MariaDB is an open-source relational database management system (RDBMS). It is a fork of the popular MySQL database system, and is widely used as an alternative to MySQL in many web-based applications and websites.

MariaDB is a database management system used to store and retrieve data. It is open-source and designed to be a drop-in replacement for MySQL, offering similar features and capabilities. MariaDB is often used in web applications and websites and is known for its reliability, efficiency, and scalability. With MariaDB, you can organize and manage

large amounts of data, run complex queries, and ensure the security of your data through advanced security features.

mysql -uroot -p : acces to database

show databases; : show databa base

create database name-database: create database

```
FROM debian:buster

RUN apt-get -y update && apt-get -y install mariadb-server dumb-init

EXPOSE 3306

COPY ./conf/create_db.sql ./

COPY ./conf/50-server.cnf ./etc/mysql/mariadb.conf.d/50-server.cnf

RUN service mysql start && mysql -uroot < create_db.sql

ENTRYPOINT ["/usr/bin/dumb-init"]

CMD ["mysqld"]
```

1. `FROM debian:buster`
   This line specifies the base image to use for building the Docker image. In this case, it's using the latest version of the Debian buster image. This is the starting point for the image being built. **“Buster"** is the codename for the Debian 10 stable release, which was released in July 2019. In the Dockerfile, `FROM debian:buster` specifies that the base image to use for building the Docker image should be the latest version of the Debian 10 (buster) distribution. This is the starting point for the image being built, and it includes the operating system and all its dependencies, libraries, and utilities.

2. `RUN apt-get -y update && apt-get -y install mariadb-server dumb-init`
   This line updates the package lists on the base image and installs the MariaDB database server and `dumb-init` utility. The `y` flag means "yes" and it automatically confirms the installation without asking for confirmation. The `dumb-init` utility is used to ensure that signals are properly propagated to all processes in the container, which helps with clean shutdowns and error handling.

3. `EXPOSE 3306`
   This line exposes port 3306 on the container, which is the default port for MariaDB. This makes the database accessible from outside the container.

4. `COPY ./conf/create_db.sql ./` and `COPY ./conf/50-server.cnf ./etc/mysql/mariadb.conf.d/50-server.cnf`
   These lines copy the `create_db.sql` and `50-server.cnf` files from the host machine to the container. The `create_db.sql` file contains SQL statements to create a database, while the `50-server.cnf` file contains configuration settings for the MariaDB server.

5. `RUN service mysql start && mysql -uroot < create_db.sql`
   This line starts the MariaDB service and runs the `create_db.sql` script to create a database. The `uroot` option specifies that the script should be run as the root user.

6. `ENTRYPOINT ["/usr/bin/dumb-init"]`
   This line specifies the default command to run when a container is created from the image. The `dumb-init` utility is used to ensure that signals are properly propagated to all processes in the container, which helps with clean shutdowns and error handling.

7. `CMD ["mysqld"]`
   This line specifies the command to run by default when a container is started from the image. This will start the MariaDB server.

```
CREATE DATABASE IF NOT EXISTS inception;
CREATE USER 'odakhch' IDENTIFIED BY 'root';
GRANT ALL PRIVILEGES ON inception.* TO 'odakhch'@'%';
ALTER USER 'root'@'localhost' IDENTIFIED BY 'root';
FLUSH PRIVILEGES;
```

1. `CREATE DATABASE IF NOT EXISTS inception;`
   This statement creates a database named "inception". The `IF NOT EXISTS` clause checks if the database already exists before trying to create it. If the database already exists, the statement will not generate an error. Instead, it will simply skip the creation of the database and move on to the next statement.

2. `CREATE USER 'odakhch' IDENTIFIED BY 'root';`
   This statement creates a database user named "odakhch" with the password "root". The `IDENTIFIED BY` clause specifies the password for the user.

3. `GRANT ALL PRIVILEGES ON inception.* TO 'odakhch'@'%';`
   This statement grants all privileges to the "odakhch" user for all tables in the "inception" database. The `.*` symbol refers to all tables in the database, and the `%` symbol refers to all hosts that the user is allowed to connect from. This statement allows the "odakhch" user to perform any action on the "inception" database, including SELECT, INSERT, UPDATE, DELETE, and more.

4. `ALTER USER 'root'@'localhost' IDENTIFIED BY 'root';`
   This statement changes the password for the "root" user when connecting from the localhost. The `@'localhost'` clause specifies that the password change only applies when connecting from the localhost.

5. `FLUSH PRIVILEGES;`
   This statement reloads the grant tables and updates the user privileges. This statement must be run after any changes to user privileges in order for the changes to take effect. The grant tables store information about user privileges and control access to the database. After running the `FLUSH PRIVILEGES` statement, the changes to the user privileges will be in effect.

What is OpenSSL?

OpenSSL is an all-around cryptography library that offers an open-source application of the TLS protocol. It allows users to perform various SSL-related tasks, including CSR (Certificate Signing Request) and private keys generation, and SSL certificate installation.

openssl-req : This command primarily creates and processes certificate requests (CSRs) in PKCS#10 format. It can additionally create self-signed certificates for use as root CAs for example.

out *filename :* This specifies the output filename to write to or standard output by default.

-x509 : This option outputs a certificate instead of a certificate request. This is typically used to generate test certificates. It is implied by the **-CA** option.

If an existing request is specified with the **-in** option, it is converted to the a certificate; otherwise a request is created from scratch.

Unless specified using the **-set_serial** option, a large random number will be used for the serial number.

Unless the **-copy_extensions** option is used, X.509 extensions are not copied from any provided request input file.

X.509 extensions to be added can be specified in the configuration file or using the **-addext** option.

-**newkey** *arg :* This option is used to generate a new private key unless **-key** is given. It is subsequently used as if it was given using the **-key** option. This option implies the **-new** flag to create a new certificate request or a new certificate in case **-x509** is given.

The argument takes one of several forms.

[**rsa:**]*nbits* generates an RSA key *nbits* in size. If *nbits* is omitted, i.e., **-newkey rsa** is specified, the default key size specified in the configuration file with the **default_bits** option is used if present, else 2048.

noenc If this option is specified then if a private key is created it will not be encrypted.

nodes This option is deprecated since OpenSSL 3.0; use **noenc** instead.

keyout *filename :* This gives the filename to write any private key to that has been newly created or read from **-key**. If neither the **-keyout** option nor the **-key** option are given then the filename specified in the configuration file with the **default_keyfile** option is used, if present. Thus, if you want to write the private key and the **-key** option is provided, you should provide the **-keyout** option explicitly. If a new key is generated and no filename is specified the key is written to standard output.

days *n :* When **-x509** is in use this specifies the number of days to certify the certificate for, otherwise it is ignored. *n* should be a positive integer. The default is 30 days.

- **subj** *arg :* Sets subject name for new request or supersedes the subject name when processing a certificate request.
  The arg must be formatted as `/type0=value0/type1=value1/type2=...`. Special characters may be escaped by `\` (backslash), whitespace is retained. Empty values are permitted, but the corresponding type will not be included in the request. Giving a single `/` will lead to an empty sequence of RDNs (a NULL-DN). Multi-valued RDNs can be formed by placing a `+` character instead of a `/` between the AttributeValueAssertions (AVAs) that specify the members of the set. Example:
  `/DC=org/DC=OpenSSL/DC=users/UID=123456+CN=John Doe`

PHP-FPM :

php-fpm: (FastCGI Process Manager) is an alternative to FastCGI implementation of PHP with some additional features useful for sites with high traffic. It is the preferred method of processing PHP pages with NGINX and is faster than traditional CGI based methods such as **SUPHP**or `mod_php` for running a PHP script. The main advantage of using PHP-FPM is that it uses a considerable amount of less memory and CPU as compared with any other methods of running PHP. The primary reason is that it demonizes PHP, thereby transforming it to a background process while providing a CLI script for managing PHP request.

Install PHP-FPM

Nginx doesn't know how to run a PHP script of its own. It needs a PHP module like PHP-FPM to efficiently manage PHP scripts. PHP-FPM, on the other hand, runs outside the NGINX environment by creating its own process. Therefore when a user requests a PHP page the nginx server will pass the request to PHP-FPM service using FastCGI. The installation of **php-fpm** in Ubuntu 18.04 depends on PHP and its version. Check the documentation of installed PHP before proceeding with installing FPM in your server. Assuming you have already installed the latest PHP 7.3, then you can install FPM using the following apt-get command.

```
FROM debian:latest

RUN apt-get update && apt-get install -y \
    apache2 \
    libapache2-mod-php \
    php \
    php-cli \
    php-gd \
    php-mysql \
    wget

RUN wget https://wordpress.org/latest.tar.gz
RUN tar -xzvf latest.tar.gz
RUN mv wordpress /var/www/html/
RUN chown -R www-data:www-data /var/www/html/wordpress

COPY wp-config.php /var/www/html/wordpress/

EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

- `FROM debian:latest`: This line specifies that the base image for this container should be the latest version of Debian. This means that the container will include all of the files and system libraries that are present in a standard Debian installation.
- `RUN apt-get update && apt-get install -y`: This command updates the package list and installs the necessary packages for running WordPress on Apache and PHP.
- `libapache2-mod-php`: This package is a module for Apache web server which allow to process PHP scripts
- `php-cli` : this package is a command line interface for PHP
- `php-gd`: This package is a library for the GD graphics library for PHP
- `php-mysql`: This package is a library for PHP that allows it to interact with a MySQL database
- `wget` : this package is a utility for downloading files from the web
- `RUN wget https://wordpress.org/latest.tar.gz` : This command downloads the latest version of WordPress in tar format
- `RUN tar -xzvf latest.tar.gz`: This command extract the tar file
- `RUN mv wordpress /var/www/html/`: This command move the wordpress files to the apache web root directory
- `RUN chown -R www-data:www-data /var/www/html/wordpress`: This command changes the ownership of the files to the user and group that runs the Apache web server.
- `COPY wp-config.php /var/www/html/wordpress/`: This command copies a file named `wp-config.php` from the host machine into the container's `/var/www/html/wordpress` directory. This file is used to configure the connection to the MySQL database and other settings for WordPress.
- `EXPOSE 80`: This command tells Docker that the container will listen on port 80.
- `CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]`: This command starts the Apache web server in the foreground.

Foreground vs Background :

Foreground and background refer to the process state in which a program runs.

A foreground process is a program that is running in the active terminal and has control over the terminal. This means that the terminal is blocked and cannot be used for other tasks until the foreground process finishes or is stopped. Foreground processes are typically used for programs that require user interaction, such as a text editor or a shell.

A background process, on the other hand, runs in the background, without control over the terminal. This means that the terminal can be used for other tasks while the background process is running. Background processes are typically used for long-running or background tasks, such as a web server or a database, that need to run continuously even when the user is not actively using them.

In summary, foreground processes have control over the terminal and block the terminal while they are running, while background processes run in the background and do not block the terminal.

PHP is a server-side scripting language that is used to create dynamic web pages. WordPress is written in PHP and uses it to interact with the MySQL database, which stores all of the content for the site.

In this Dockerfile, we install PHP and the necessary libraries (php-cli, php-gd, php-mysql) so that the Nginx web server that is running inside the container can process PHP scripts and interact with the MySQL database. Without PHP, the web server would not be able to understand or execute the code that makes up the WordPress application, and the site would not work properly.

Additionally, we install the `libapache2-mod-php` package which allows the Apache web server to process PHP script, it's a module for Apache web server and it'll enable PHP processing.

In short, PHP is an essential component for running a WordPress site, and installing it in the container is necessary for the container to function properly as a WordPress server.

```
openssl req -x509 -sha256 -nodes -days 365 \
-newkey rsa:4096 -keyout /etc/nginx/ssl/odakhch.key \
-out /etc/nginx/ssl/odakhch.crt \
-subj "/C=MA/ST=Béni Mellal-Khénifra/L=Khouribga/O=1337/OU=odakhch/CN=odakhch.42.fr"
```

The `req` command is used for creating a certificate request or a self-signed certificate.

The `-x509` option tells OpenSSL to create a self-signed certificate instead of a certificate request.

A certificate signing request (CSR) is one of the first steps towards getting your own SSL/TLS certificate. Generated on the same server you plan to install the certificate on, the CSR contains information (e.g. common name, organization, country) the Certificate Authority (CA) will use to create your certificate. It also contains the public key that will be included in your certificate and is signed with the corresponding private key. We'll go into more details on the roles of these keys below.

The `-sha256` option specifies the SHA-256 hashing algorithm to be used to sign the certificate.

The `-nodes` option tells OpenSSL to not encrypt the private key.

The `-days 365` option sets the number of days that the certificate will be valid for.

The `-newkey rsa:4096` option generates a new RSA key with a key length of 4096 bits.

RSA is a public-key encryption algorithm and the most widely-used encryption standard for secure data transmission. It is named after its inventors, Ron Rivest, Adi Shamir, and Leonard Adleman, who first published the algorithm in 1977.

RSA uses two keys for encryption and decryption: a public key and a private key. The public key is used to encrypt the data, while the private key is used to decrypt it. This allows users to send encrypted messages to each other without having to exchange secret keys beforehand.

The `-keyout /etc/nginx/ssl/odakhch.key` option specifies the location and filename where the private key will be saved.

The `-out /etc/nginx/ssl/odakhch.crt` option specifies the location and filename where the certificate will be saved.

The `-subj "/C=MA/ST=BéniMellalKhénifra/L=Khouribga/O=1337/OU=odakhch/CN=odakhch.42.fr"` option sets the subject of the certificate. The subject is a set of attributes that identify the entity the certificate is issued to.

C= represents the country code, ST= represents the state, L= represents the location, O= represents the organization, OU= represents the organizational unit and CN= represents the common name which is the hostname.

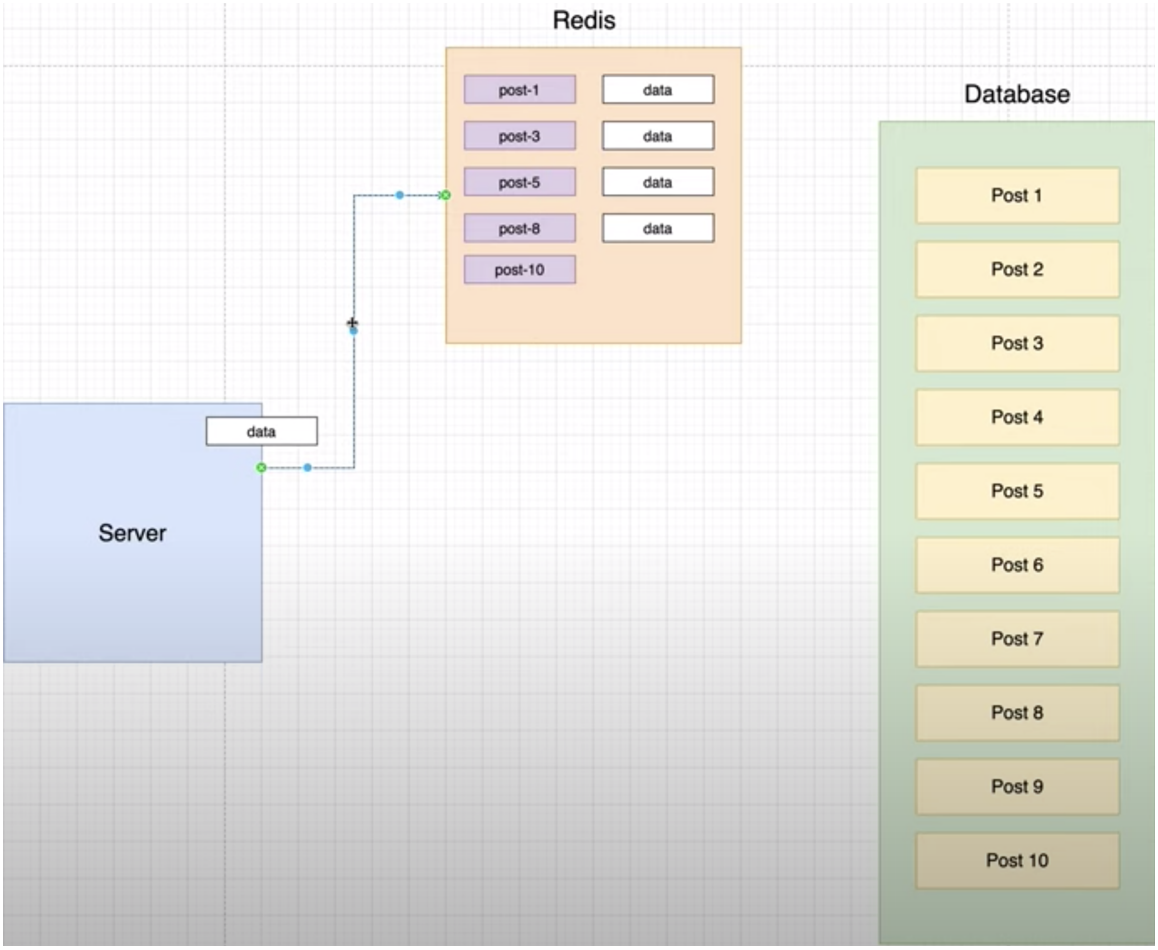
So in this case, the certificate is issued to entity odakhch.42.fr that located in Morocco, Béni Mellal-Khénifra, Khouribga, with organization name 1337 and organizational unit

odakhch.

curl is a command-line tool that allows you to transfer data to or from a server, using various protocols such as HTTP, HTTPS, FTP, and others. In this case, the line `curl -O https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar` downloads the wp-cli.phar file from the specified URL, and saves it in the current working directory.

Redis :

Redis is an in-memory data store that can be used as a cache and a database (caching). It is often used for high-performance, scalable web applications, providing quick access to data through its in-memory storage and support for complex data structures. Redis supports multiple data structures such as strings, hashes, lists, sets, and sorted sets.



```
bind 0.0.0.0
port 6379

logfile "/var/log/redis/redis.log"
loglevel notice

maxmemory 128mb
maxmemory-policy volatile-lru
```

These are configuration options for a Redis server, a popular in-memory data store.

- `bind 0.0.0.0`: This option tells Redis to bind to all available network interfaces on the system. When a Redis server is bound to a specific IP address, it will only listen for incoming connections on that address. By binding to 0.0.0.0, the server will listen on all IP addresses and will accept incoming connections from any network interface.
- `port 6379`: This option specifies the port that Redis will listen on. By default, Redis listens on port 6379, so unless you need to run multiple Redis instances on the same machine or need to change the port for some other reason, you don't need to modify this option.
- `logfile "/var/log/redis/redis.log"`: This option sets the location of the Redis log file. The log file will contain information about Redis events, such as server start and stop events, database changes, and error messages.
- `loglevel notice`: This option sets the log level for Redis. The log level determines the amount of detail that is logged in the log file. Redis supports multiple log levels, including "debug", "verbose", "notice", "warning", and "error". The "notice" level provides a moderate amount of logging information. If you need more detailed information, you can set the log level to "debug" or "verbose", but keep in mind that this will increase the size of the log file and could affect performance.

notice: A moderate level of logging that provides information about important events and messages. This level is a good balance between detail and performance.

"loglevel notice" is a configuration setting in Redis, which determines the level of logging that the Redis server will produce. The available log levels in Redis are "debug", "verbose", "notice", "warning" and "silent".

"Notice" is a moderate logging level that logs important events and warnings. The logs generated with the "notice" log level can be used to troubleshoot problems and monitor the behavior of the Redis server.

For example, if you set the loglevel to "notice", the Redis server will log events such as client connections and disconnections, configuration changes, and other important messages.

- `maxmemory 128mb`: This option sets the maximum amount of memory that Redis is allowed to use. In this case, the maximum is set to 128 MB. When Redis reaches its maximum memory limit, it will start to evict data to free up memory.
- `maxmemory-policy volatile-lru`: This option sets the eviction policy to use when Redis reaches its maximum memory limit. The "volatile-lru" policy means that Redis will remove the least recently used keys among all keys with an expire set. This policy is suitable for use cases where Redis is used to store ephemeral data that can be safely expired.

FTP :

FTP (File Transfer Protocol) server is a computer system that provides file transfer service using FTP. It allows users to upload and download files to/from the server over a network. FTP servers are commonly used to store and share files, such as images, audio, video, and software programs, with multiple users over the internet.

FTP (File Transfer Protocol) is a standard network protocol used for transferring files from one host to another over a TCP-based network, such as the Internet. Installing an FTP server software allows you to host and manage your own FTP server, allowing you to easily transfer files to and from the server, and providing secure access to the files for authorized users. The FTP server software can be installed on a dedicated server, virtual machine, or even on a personal computer, providing a convenient and efficient way to transfer and manage large files.

https://www.youtube.com/watch?v=L9aZpg0ip70

In an FTP server configuration, the ports used play a crucial role in establishing an FTP connection.

- Port 20: FTP uses port 20 for data transfers. When a client connects to an FTP server, port 21 is used for the control connection and port 20 is used for data transfers.
- Port 21: This is the default port for FTP control connections. It is used to establish the initial connection with the FTP server, and also for authentication and sending commands.
- Port 40000-40005: These are ports that are typically used for passive mode FTP transfers. Passive mode FTP is an alternative to active mode FTP and is used when the client is behind a firewall or is unable to establish a direct connection to the server.

By specifying these port numbers, the FTP server is telling the clients what ports it is listening on and that it is ready to receive incoming connections and transfers.

The vsftpd (Very Secure FTP Daemon) is an FTP server for Unix-like systems, including Linux. It is designed to be secure and efficient, and is widely used for hosting FTP services.

When creating a new FTP user, you want to ensure that their data and activities are separated from other users on the system, and that they are only able to access a specific set of directories. This is why the script creates a new user and sets up a specific directory structure for them, with appropriate permissions.

```
adduser $FTP_CLIENT --disabled-password

echo "$FTP_CLIENT:$FTP_PASS" | /usr/sbin/chpasswd

echo "$FTP_CLIENT" | tee -a /etc/vsftpd.userlist

mkdir -p /home/$FTP_CLIENT/ftp

chown nobody:nogroup /home/$FTP_CLIENT/ftp
chmod a-w /home/$FTP_CLIENT/ftp

mkdir /home/$FTP_CLIENT/ftp/files
chown $FTP_CLIENT:$FTP_CLIENT /home/$FTP_CLIENT/ftp/files

service vsftpd start
service vsftpd stop

/usr/sbin/vsftpd
```

- `adduser`: The `adduser` command is used to create a new user account on the system. The `-disabled-password` option means that the user will not have a password set, and must authenticate using other methods (such as SSL/TLS).
- `chpasswd`: The `chpasswd` command is used to set the password for a user account. The format of the input is "username:password", and the `chpasswd` utility updates the appropriate entry in the `/etc/shadow` file (which stores the encrypted passwords for all users on the system).

  `echo "$FTP_CLIENT:$FTP_PASS" | /usr/sbin/chpasswd`: This command sets the password for the newly created user using the `chpasswd` utility. The password is specified in the `$FTP_PASS` variable, and is passed to `chpasswd` in the format "username:password".

  In the script, the user is created with `--disabled-password`, but a password is set for the user via the `chpasswd` command. This means that the user can log into the FTP server using either an alternative authentication method or a username and password combination.

  The reason for this might be that the script is intended to be flexible and allow for different authentication methods to be used, depending on the user's requirements. If the user wants to use an alternative authentication method, then they can simply omit the `chpasswd` command and the user will still be created with `--disabled-password`. But if the user wants to use a username and password combination, then they can set the password using the `chpasswd` command.

  In other words, the script is designed to allow the user to choose the authentication method that they prefer, while still being able to create the user with a password if necessary.
- `echo "$FTP_CLIENT" | tee -a /etc/vsftpd.userlist`: This command adds the new user to the vsftpd user list by appending their username to the `/etc/vsftpd.userlist` file. The `tee -a` command is used to append the output of the `echo` command to the file.
- `mkdir -p /home/$FTP_CLIENT/ftp`: This command creates a directory for the FTP user in their home directory. The `-p` option is used to create any necessary intermediate directories.
- `chown nobody:nogroup /home/$FTP_CLIENT/ftp`: This command sets the ownership of the FTP directory to the `nobody:nogroup` user and group.

- `chmod a-w /home/$FTP_CLIENT/ftp`: This command revokes write permission for all users on the FTP directory.

The `chmod +x wp-cli.pha` is a Unix command used to set the execute permissions for a file. The "+x" option allows the file to be executed as a program. In this case, the file `wp-cli.phar` is a PHP archive, which contains a command line interface for the popular WordPress content management system.

- `mkdir /home/$FTP_CLIENT/ftp/files`: This command creates a subdirectory in the FTP directory for storing files.
- `chown $FTP_CLIENT:$FTP_CLIENT /home/$FTP_CLIENT/ftp/files`: This command sets the ownership of the `files` directory to the newly created FTP user.
- `service vsftpd start`: This command starts the vsftpd FTP server.
- `service vsftpd stop`: This command stops the vsftpd FTP server.
- `/usr/sbin/vsftpd`: This command starts the vsftpd FTP server directly, bypassing the service manager.

/usr/sbin/vsftpd and service vsftpd start are both commands used to start the vsftpd FTP server. However, there is a difference between the two.

/usr/sbin/vsftpd is a standalone executable that starts the vsftpd server directly. It is typically used when starting vsftpd as a foreground process for testing or debugging purposes.

service vsftpd start is a system command that starts the vsftpd service through the init system. This is typically used when starting vsftpd as a background service that runs continuously. The init system is responsible for starting and stopping services on the system, and it provides a unified interface for managing services, regardless of the underlying operating system.

So, to put it simply, /usr/sbin/vsftpd starts the vsftpd server directly, while service vsftpd start starts the vsftpd server as a service managed by the init system. The latter is the more commonly used method for starting vsftpd as a background service.

Overall, this script creates a new FTP user with a specific directory structure and permissions, and starts the vsftpd FTP server. The `vsftpd` server is then stopped and started directly, which may be required to apply the changes made by the script.

An FTP script can be useful in a variety of scenarios, such as:

- Automating the process of setting up a new FTP user
- Simplifying the management of multiple FTP users
- Ensuring that the same configuration is applied consistently to all FTP users

CAdvisor :

CAdvisor (Container Advisor) is an open-source tool for analyzing and monitoring the resource usage and performance of containers. It provides information about resource usage and performance for containers running on a host machine, including information about CPU, memory, disk, and network usage.

CAdvisor is designed to run as a container itself, making it easy to deploy and use on any system that runs containers. It provides detailed information about the performance of individual containers, as well as aggregated information about the performance of all containers running on a host.

This information can be useful for a variety of purposes, such as:

- Troubleshooting performance issues with containers.
- Monitoring resource usage to ensure that containers are using resources efficiently.
- Determining the resource requirements for new containers.
- Optimizing the resource allocation for containers to improve performance and reduce costs.

CAdvisor is widely used in production environments to monitor the performance and resource usage of containers, and is a valuable tool for anyone working with containers.

Adminer :

Adminer is a free and open-source database management tool written in PHP. It is used to manage various databases including MySQL, PostgreSQL, SQLite, and more. It provides a simple and user-friendly interface for managing databases, tables, columns, and records.
Adminer allows you to perform common database tasks such as creating tables, inserting data, updating records, and running SQL queries. It is widely used as a substitute for more complex database management tools like phpMyAdmin.

Adminer is lightweight, easy to install, and requires only a web server with PHP installed to run. It also supports multiple authentication methods, including HTTP authentication, cookie authentication, and encrypted login. This makes it ideal for managing databases on small projects or personal servers.

In summary, Adminer is a tool that helps you manage databases, while MariaDB is a specific type of database that you can manage using Adminer or other database management tools.

Adminer is a tool for managing databases. It provides a web-based interface for managing databases, including performing common database administration tasks such as creating and modifying tables, running SQL queries, and managing users.

MariaDB is a database management system (DBMS) that is an open-source, community-driven fork of the popular MySQL database management system. MariaDB is designed to be a drop-in replacement for MySQL, meaning it has the same syntax, functionality, and APIs, but with some additional features and improvements.

So the difference between Adminer and MariaDB is that Adminer is a tool for managing databases, while MariaDB is a database management system. Adminer can be used to manage multiple database management systems, including MariaDB.

In other words, Adminer is a tool that provides a web-based interface for managing databases, while MariaDB is the actual database management system that stores and manages the data. You can use Adminer to manage a MariaDB database, or you can use the command-line or other tools to manage the database. The choice of tool depends on your specific needs and preferences.

The `restart: always` and `restart: on-failure` options in a Docker Compose file or `--restart` option in the `docker run` command are used to specify the behavior of Docker when a container exits.

- `restart: always` means that Docker will automatically restart the container whenever it exits, regardless of the exit status.
- `restart: on-failure` means that Docker will restart the container only if it exits with a non-zero exit status, which typically indicates an error or failure.

So, if you want to ensure that the container is always running, use `restart: always`. On the other hand, if you only want to restart the container in case of a failure, use `restart: on-failure`.

In computing, an exit code (or exit status) is a numeric value that is returned by a program to indicate whether it completed successfully or not. In most cases, a zero exit code indicates success, while a non-zero exit code indicates an error or failure of some kind.

So, when a container in Docker exits with a non-zero exit code, it means that it encountered an error or some kind of failure during its execution. The `restart: on-failure` policy in a Docker Compose file tells Docker to automatically restart the container if it exits with a non-zero exit code. This can be useful in ensuring that the container is always running, even if it encounters an error.