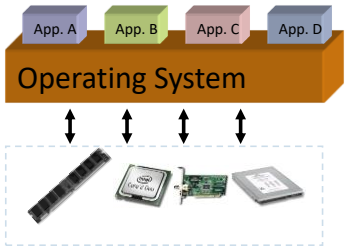
Virtualization

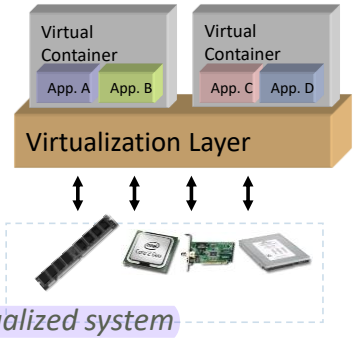
GIANLUCA REALI

What is virtualization?

- Compute Virtualization includes **CPU virtualization**, **memory virtualization** and **I/O virtualization**, producing virtual memory, storage, network, etc.
- Virtualization basically allows one computer to do the job of multiple computers, by sharing the resources of a single hardware across multiple environments



'Nonvirtualized' system
A single OS controls all hardware platform resources



Virtualized system
It makes it possible to run multiple Virtual Containers on a single physical platform

In the past

One operating systems in one machine, so the OS had completely control of the resources in that machine.

Virtualization Era

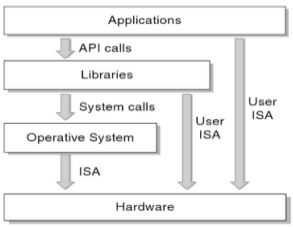
Virtualization consists of a software layer in between the machine and the operating system. Essentially what this software layer does is to divide the resources of the machine among all the guest operating systems. The Virtualization Layer is in charge of multiplexing the hardware resources to several operating systems. Each OS has the illusion that it controls the complete hardware but, in fact, the machine can now host a number of operating systems because the virtualization layer makes all the switching behind scenes.

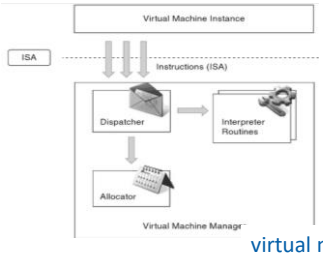
Supporting multiple instances of Operating Systems: Homogeneous or Heterogeneous

One physical machine can host several Linux/Window copies

Virtualization Requirements

- Virtual Machine Monitor
- A virtual machine monitor is a control program comprising:
  - A dispatcher
  - An allocator
  - A set of interpreters, one per privileged instruction.



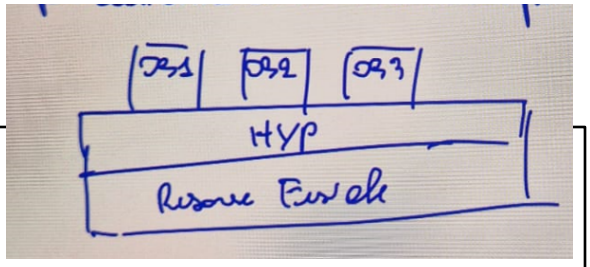


virtual machine monitor

ISA: Instruction Set Architecture. Model of the hardware that defines the instruction set of the processor. Essentially it consists of the machine language. Some instructions may be used by applications. Others are reserved for the OS.
ABI: application binary interface

Dispatcher: intercepts sensitive instructions executed by VMs and leaves control to the modules set up to manage the situation;
Allocator: provides virtual machines with the necessary resources avoiding conflicts (hardware resource management);
Interpreter: simulates instructions that refer to resources in ways that reflect their execution in the virtual machine environment (indispensable since VMs do not have direct access to physical resources).
Finally, the purpose of systems virtualization is to transparently share the hardware resources of a physical machine among the operating systems of multiple virtual machines, to make them execute as many instructions as possible directly on the processor without VMM interaction and to solve potential causes of malfunctions due to any lack of architecture virtualization requirements.

A VMM provides a *duplicate*, or *essentially identical* to the original machine, environment for programs. "Any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly, with the possible exception of differences caused by the availability of system resources and differences caused by timing dependencies."
It does so *efficiently*, requiring "a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor, with no software intervention by the VMM. This statement rules out traditional emulators and complete software interpreters (simulators) from the virtual machine umbrella." Thus programs that run in this environment show only minor decreases in speed.
It is in complete control of system resources (memory, peripherals, and the like). This requires two conditions: (i) it must not be possible for a program running in the created environment to access any resource not allocated to it (*isolation*), and (ii) it is possible under certain circumstances to regain control of resources already allocated.

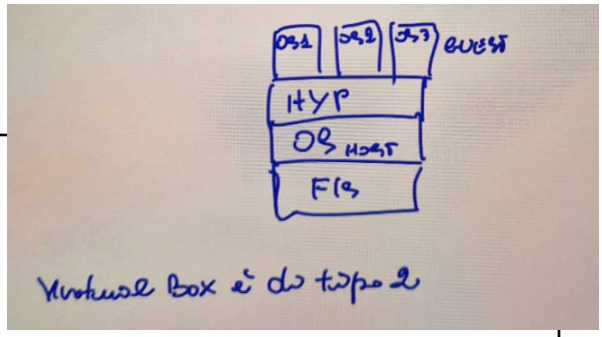


Type 1 hypervisors

- Type 1 –hypervisor: bare metal, it has direct access to hardware resources and does not need to access the host OS. Kind of customized OS implementing VMM and does not run other applications.
- It responds to privileged CPU instructions or protection instructions sent by VMs, schedules VM queues, returns VMs the results of physical hardware processing.
- VMM creates and manages virtual environments.
- Pros: possible implementation of different guest OS types. Cons: the kernel of the virtualization layer is hard to develop.

Type 1 Virtual Machine Monitor (VMM) features

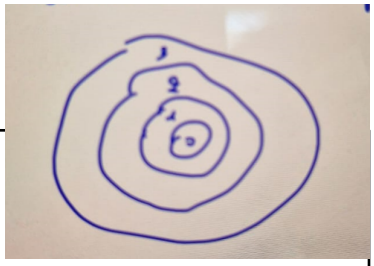
Two types of VMM can also be identified, based on the location of the physical machine in the environment: 1. Type 1 VMM is placed immediately above the hardware and has all the mechanisms of a normal kernel or operating system in terms of memory, peripheral and processor management; in addition, it implements the mechanisms for managing virtual machines. Virtual machines running above VMM have their own operating system and are called "guest" machines.



Type 2 hypervisors

- Called Hosted hypervisor. Physical resources are managed by host OS.
- VMM provides virtualization as application. VMM obtains resources by calling host OS services. The VM, after creation, is scheduled by the VMM as a process of the host OS.
- Pros: Easy to implement. Cons: Only applications supported by the host OS can ne installed and used. High perormance overhead.

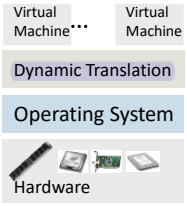
VMM type 2 is a normal process running under an operating system called "host". It directly manages the virtual machines, which are its sub-processes, while the management of the hardware is entrusted to the host system.



Evolution of Software solutions

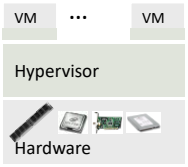
1st Generation: Full virtualization (Binary rewriting)

- Software Based
- QEMU, VMware and Microsoft



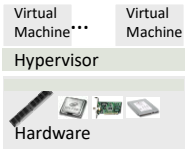
2nd Generation: Paravirtualization

- Cooperative virtualization
- Modified guest
- VMware, Xen



3rd Generation: Silicon-based (Hardware-assisted) virtualization

- Unmodified guest
- VMware and Xen on virtualization-aware hardware platforms



Server virtualization approaches

Virtual Machine Monitor

Each virtual machine interfaces with its host system via the virtual machine monitor (VMM). Being the primary link between a VM and the host OS and hardware, the VMM provides a crucial role. The VMM primarily:

Presents emulated hardware to the virtual machine

Isolates VMs from the host OS and from each other

Throttles individual VM access to system resources, preventing an unstable VM from impacting system performance

Passes hardware instructions to and from the VM and the host OS/hypervisor

When full virtualization is employed, the VMM will present a complete set of emulated hardware to the VM's guest operating system. This includes the CPU, motherboard, memory, disk, disk controller, and network cards. For example, Microsoft Virtual Server 2005 emulates an Intel 21140 NIC card and Intel 440BX chipset. Regardless of the actual physical hardware on the host system, the emulated hardware remains the same.

The next significant role of the VMM is to provide isolation. The VMM has full control of the physical host system's resources, leaving individual virtual machines with access only to their emulated hardware resources. The VMM contains no mechanisms for inter-VM communication, thus requiring that two virtual machines wishing to exchange data do so over the network.

Another major role of the VMM is to manage host system resource access. This is important, as it can prevent over-utilization of one VM from starving out the performance of other VMs on the same host. Through the system configuration console, system hardware resources such as the CPU, network, and disk access can be throttled, with maximum usage percentages assigned to each individual VM. This allows the VMM to properly schedule access to host system resources as well as to guarantee that critical VMs will have access to the amount of hardware resources they need to sustain their operations.

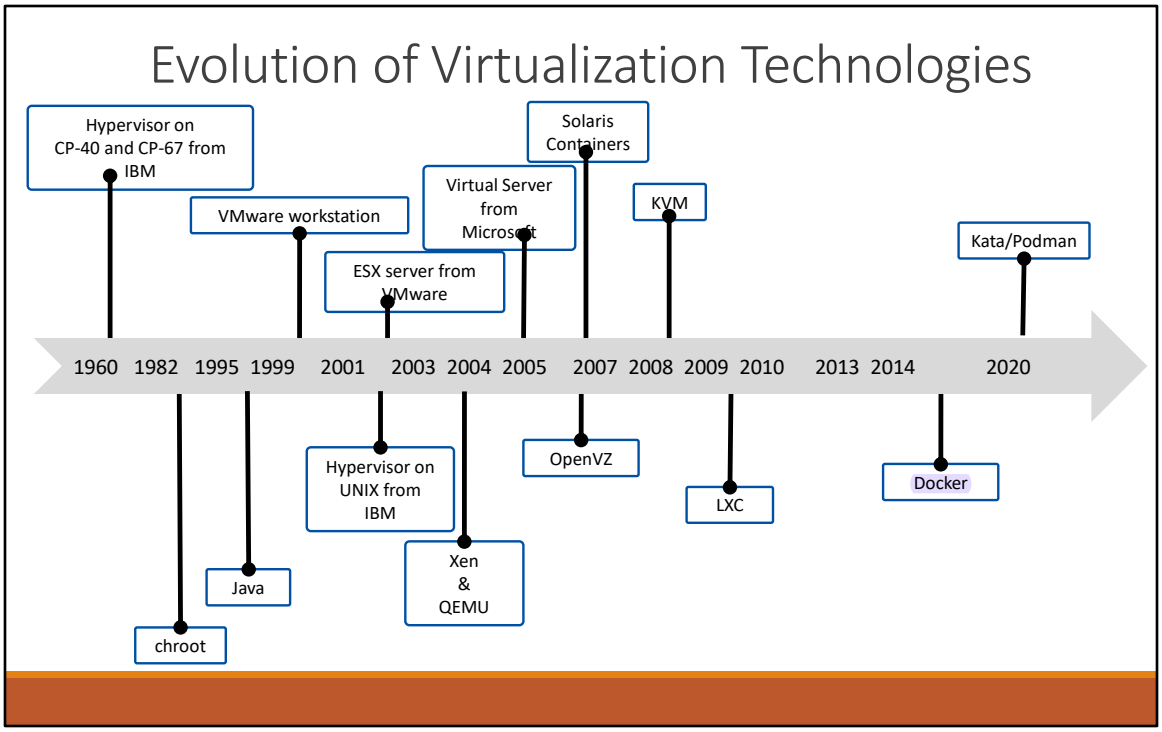
Host OS/Hypervisor

The primary role of the host operating system or hypervisor is to work with the VMM to coordinate access to the physical host system's hardware resources. This includes scheduling access to the CPU as well as the drivers for communication with the physical devices on the host, such as its network cards.

The term hypervisor is used to describe a lightweight operating shell that has the sole purpose of providing VM hosting services. The hypervisor differs from a traditional OS in that the OS may be designed for other roles on the network. As it is tailored to VM hosting, a hypervisor solution generally offers better performance and should have fewer security vulnerabilities because it runs few services and contains only essential code. Hypervisors written for hardware-assisted virtualization can embed themselves much deeper into the system architecture and offer superior performance improvements as a result.

Like any traditional OS, a hypervisor-based OS still contains its own operating system code; therefore, maintaining security updates is still important. Unlike a traditional OS, hypervisors are vendor specific, so any needed hypervisor patches or security updates will come directly from the virtualization software vendor. Because hypervisors are vendor-centric, individual device support often comes directly from the virtualization vendors. Hence, it is important for the organization to ensure that any planned virtualization products are compatible with its existing or planned system hardware. When hosting VMs on a traditional OS such as SUSE Linux Enterprise Server or Windows Server “Longhorn,” the organization will find that while the host OS has a larger footprint than a hypervisor, it does provide additional flexibility with hardware devices. With SAN integration, for example, if the host OS does not recognize a Fibre Channel host bus adapter (HBA), the administrator can download the appropriate driver from the vendor's website. With a hypervisor, the administrator will need to get the driver from the virtualization software vendor, or learn that the device is not supported.

Both hypervisors and operating systems have their strengths and weaknesses. Operating systems provide greater device support than hypervisors, but also require attention to ensure that they are current on all patches and security updates. Hypervisors run on minimal disk and storage resources, but patches and device drivers must come directly from the virtualization software vendor.



Server virtualization started back in the early 1960's and was pioneered by companies like General Electric (GE), Bell Labs, and International Business Machines (IBM), to run legacy software on newer mainframe hardware, and to support newer mainframe hardware capable of more than one simultaneous user. The CP-67 system was the first commercial mainframe to support virtualization. The CP approach to time sharing allowed each user to have their own complete operating system which effectively gave each user their own computer. The main advantages of using virtual machines vs a time sharing operating system was more efficient use of the system since virtual machines were able to share the overall resources of the mainframe, instead of having the resources split equally between all users. There was better security since each user was running in a completely separate operating system, and it was more reliable since no one user could crash the entire system; only their own operating system.

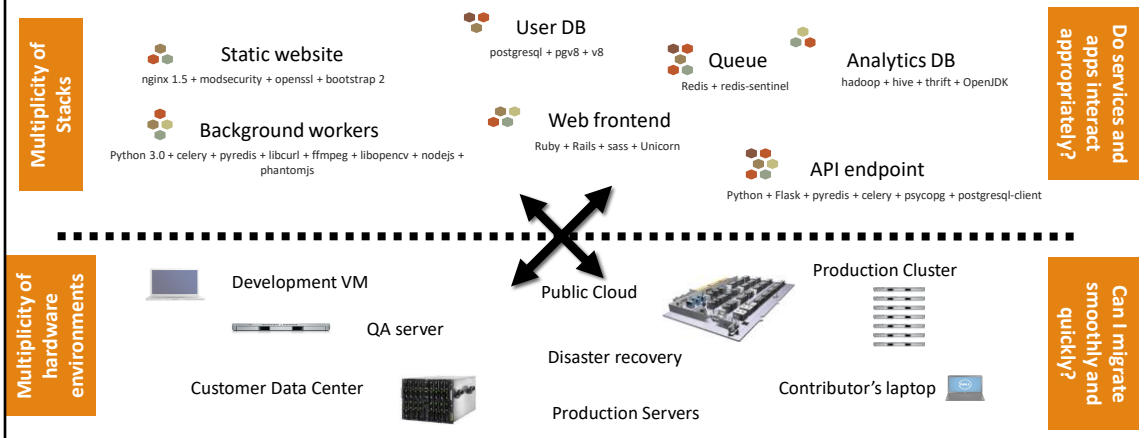
By the late 1980's *software emulators* were developed to allow users to run DOS and Windows applications on their Unix and Mac workstations. In 1999 VMware developed virtualization technology that *simulated enough hardware* to allow an unmodified guest OS to be run in isolation. In 2001 VMware released ESX Server that does not require a host operating system to run virtual machines. This is known as a *type 1 hypervisor*. That year they also released GSX Server that allowed users to run virtual machines on top of an existing operating system, such as Microsoft Windows. This is known as a *type 2 hypervisor*. Xen was the first open-source x86 hypervisor, released in 2003 by researchers at Cambridge University. The Xen hypervisor is a small, lightweight type 1 hypervisor derived from work done on the Linux kernel. The Xen hypervisor can run unmodified fully virtualized guests, or paravirtualized guests that use a special API to communicate with the hypervisor. KVM (Kernel-based Virtual Machine) was merged into the Linux kernel mainline in February 2007. It is a virtualization infrastructure for the Linux kernel that turns it into a type 2 hypervisor.

In 2005 Sun released Solaris Containers with Solaris 10 for x86 and SPARC systems. Containers

Containers – Introduction

- Containers virtualize the OS just like hypervisors virtualizes the hardware
- Containers enable any payload to be **encapsulated** as a lightweight, portable self-sufficient container, that can be manipulated using standard operations and run consistently on any hardware platform.
- **Wraps up** a piece of software in a **complete filesystem** that contains everything it needs to run such as : code, runtime, system tools, libraries etc., they share the OS kernel and bins/libs where needed, otherwise each of them operate in a self contained environment.

The Challenge



The Matrix From Hell

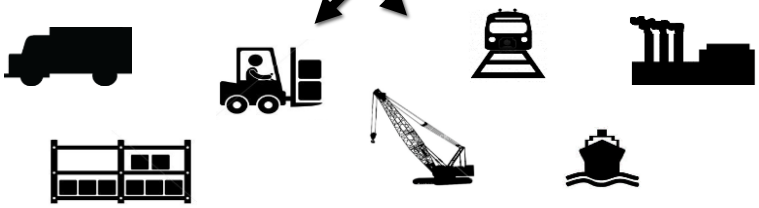
	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

Cargo Transport Pre-1960

Multiplicity of Goods

Do I worry about how goods interact (e.g. coffee beans next to spices)

Multipilicity of methods for transporting/storing

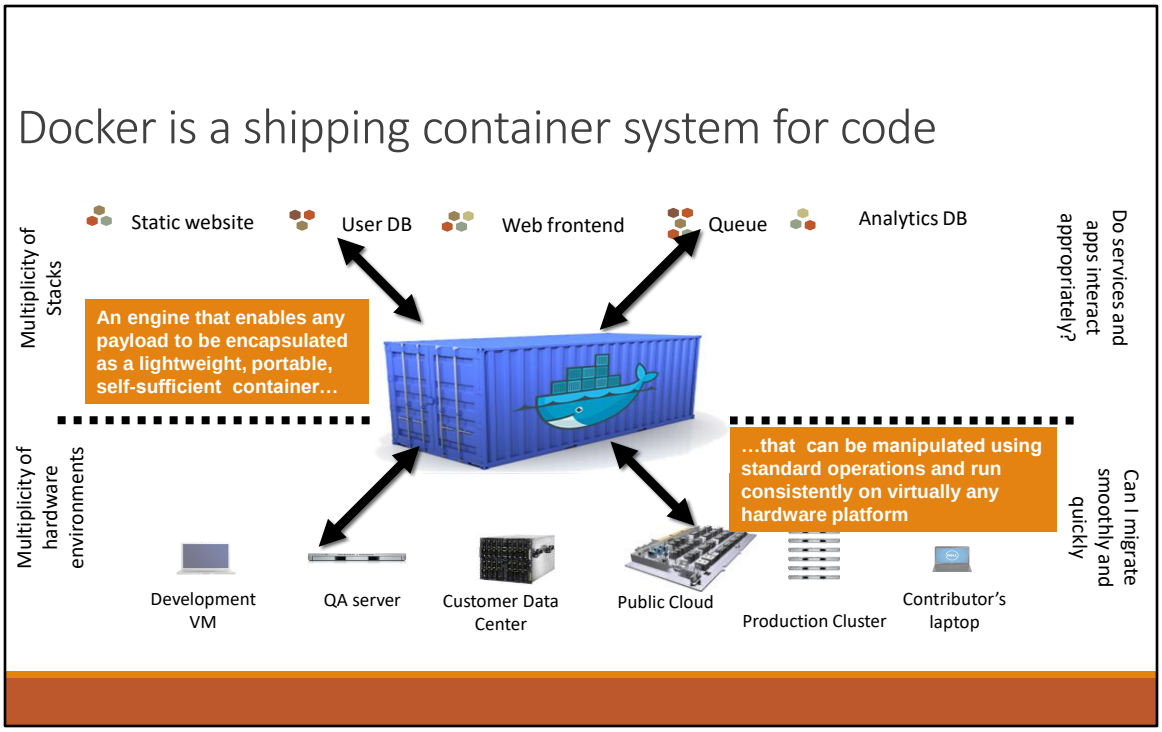


Can I transport quickly and smoothly (e.g. from boat to train to truck)

Also a matrix from hell

	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
							

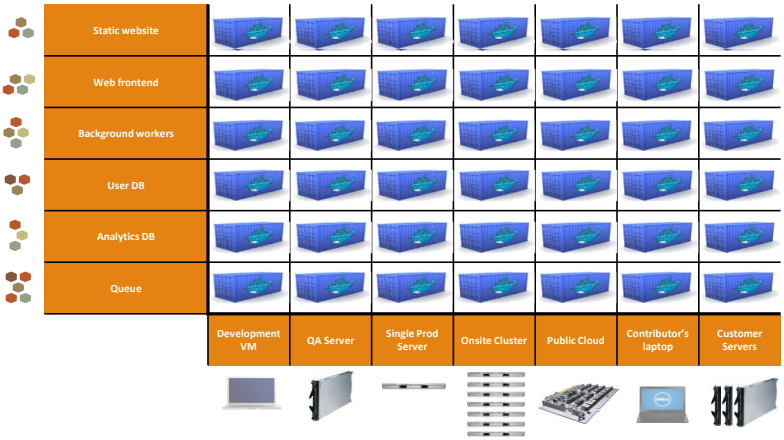
Solution: Intermodal Shipping Container



A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

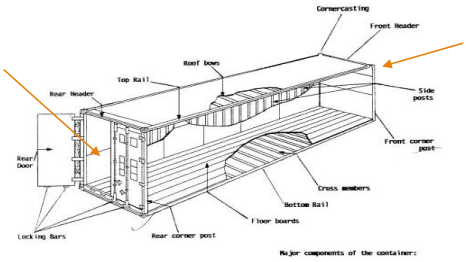
Docker eliminates the matrix from Hell



Why it works—separation of concerns

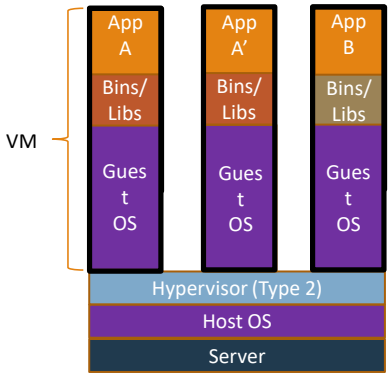
Dan the Developer

- Worries about what's "inside" the container
  - His code
  - His Libraries
  - His Package Manager
  - His Apps
  - His Data
- All Linux servers look the same



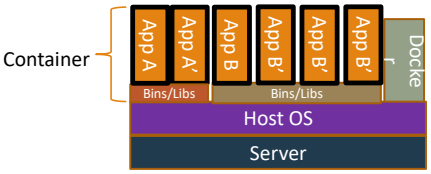
- Oscar the Ops Guy
  - Worries about what's "outside" the container
    - Logging
    - Remote access
    - Monitoring
    - Network config
  - All containers start, stop, copy, attach, migrate, etc. the same way

Containers vs. VMs



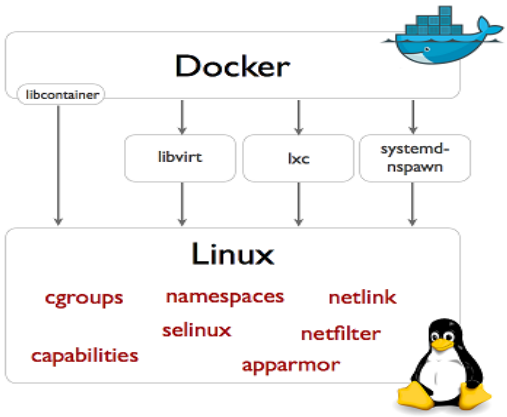
Containers are isolated, but share OS and, where appropriate, bins/libraries

…result is significantly faster deployment, much less overhead, easier migration, faster restart



Virtualization Technologies

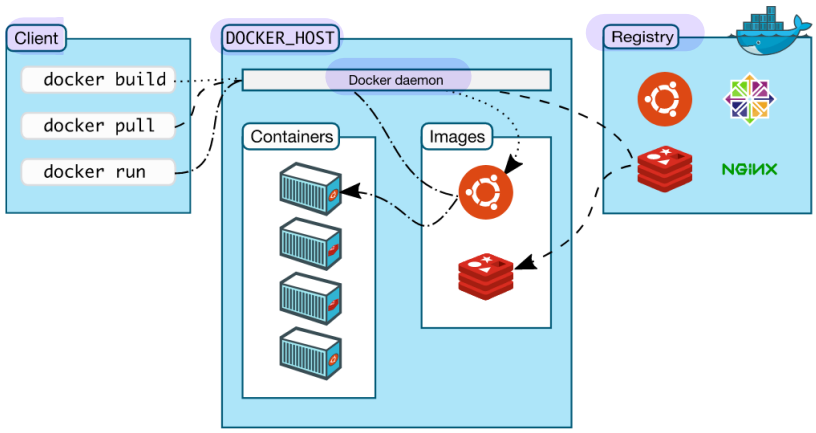
Docker – Underlying Technology



Docker makes use of several Linux kernel features to deliver the functionality.

Linux kernel namespaces essentially isolate what the application can see from the operating environment, including the process tree, network, user IDs and mounted file systems, while cgroups provide isolation of resources, including CPU, memory, block I / O devices and the network. Docker includes the libcontainer library to directly use the virtualization features of the Linux kernel, in addition to abstract virtualization interfaces such as libvirt, LXC and systemd-nspawn.

Docker architecture



Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

Docker architecture

The Docker daemon: The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client: The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker architecture

Docker registries A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

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

Docker Desktop is an easy-to-install application for your Mac or Windows environment that enables you to build and share containerized applications and microservices.

- Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

the Docker Dashboard, which gives you a quick view of the containers running on your machine. The Docker Dashboard is available for Mac and Windows. It gives you quick access to container logs, lets you get a shell inside the container, and lets you easily manage container lifecycle (stop, remove, etc.).

Docker objects

When you use Docker, you are creating and using **images**, **containers**, **networks**, **volumes**, **plugins**, and other objects.

Images An *image* is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

Containers A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

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

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine. A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

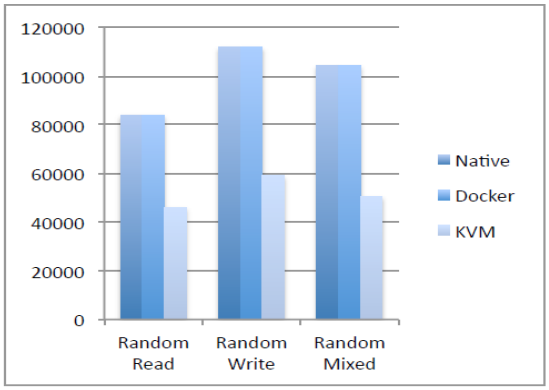
Example: docker run

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

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

When you run this command, the following happens (assuming you are using the default registry configuration):

I/O Performance



Random I/O throughput (IOPS)

IBM Research Report July, 2014

IOV method	throughput (Mb/s)	CPU utilization
bare-metal	950	20%
device assignment	950	25%
paravirtual	950	50%
emulation	250	100%

- `netperf` TCP_STREAM sender on 1Gb/s Ethernet (16K msgs)
- Device assignment best performing option
- Device assignment still 25% worse than bare metal.

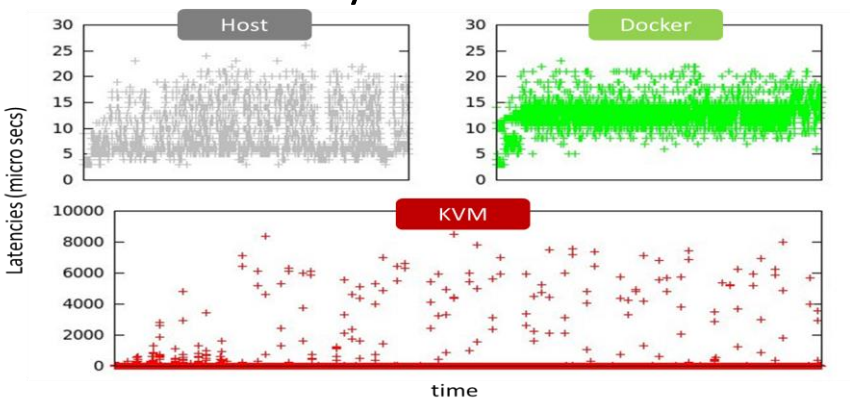
IBM®System x3650 M4 server
two 2.4-3.0 GHz Intel Sandy Bridge-EP Xeon E5-2665 processors
16 cores (plus HyperThreading)
256 GB of RAM

SAN-like block storage is commonly used in the cloud to provide high performance and strong consistency.
The Random I/O throughput benchmarks are slower on KVM because each I/O operation must go through QEMU (emulation).

KVM network paravirtualization does not have to go through an emulated NIC, but the paravirtualization results in an extreme amount of context switching, and interrupt processing still goes through the hypervisor.

Real-time Latency

Cyclictest



Intel Ivy bridge based 4 core with hyper-threading (8 logical cores) each running @ 2.2 GHz.
8 GB RAM

Latency is defined as the time interval between the occurrence of an event and the time when that event is "handled" (usually running a thread).

Examples:

- The time between when an interrupt occurs and the thread waiting for that interrupt is run
- The time between a timer expiration and the thread waiting for that timer to run
- The time between the receipt of a network packet and when the thread waiting for that packet runs

Cyclictest is one of the widely used tool for RT performance benchmarking. It measures the latency of response to a stimulus.

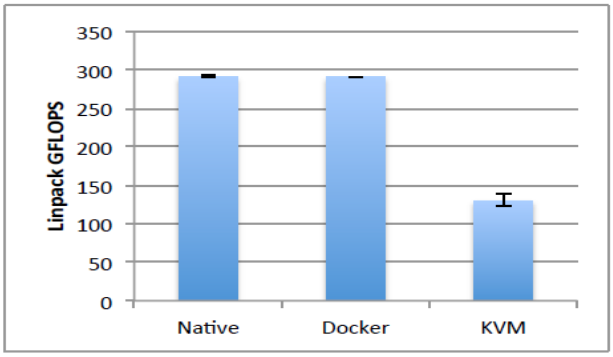
From the benchmarking figure:

Host – Latencies under 25 micro seconds

Docker – Same as the host

KVM – Latencies as high as 9000 micro seconds

Math Performance

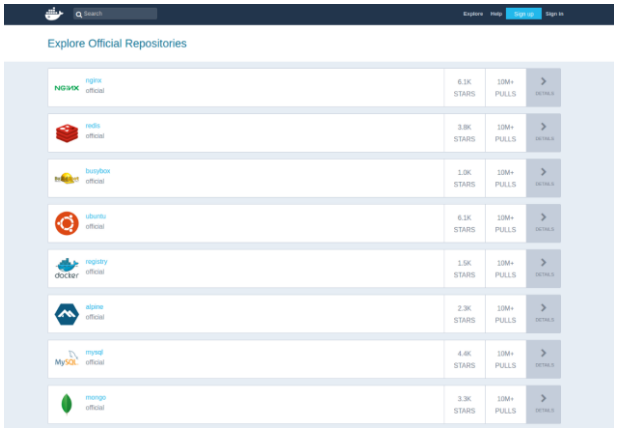


IBM Research Report July, 2014

IBM®System x3650 M4 server
two 2.4-3.0 GHz Intel Sandy Bridge-EP Xeon E5-2665 processors
16 cores (plus HyperThreading)
256 GB of RAM

Mathematics can be much slower with KVM because math libraries cannot tune themselves as well with the system-provided information that KVM offers.
By being unable to detect the exact nature of the system, the execution employs a more general algorithm with consequent performance penalties.

Docker Hub



Docker Hub is the world's largest repository of container images with an array of content sources including container community developers, open source projects and independent software vendors (ISV) building and distributing their code in containers. Users get access to free public repositories for storing and sharing images or can choose subscription plan for private repos.

Dockerfile

It is possible to build authomatically your own images reading instructions from a Dockerfile

```
FROM centos:7
RUN yum install -y python-devel python-virtualenv
RUN virtualenv /opt/indico/venv
RUN pip install indico
COPY entrypoint.sh /opt/indico/entrypoint.sh
EXPOSE 8000
ENTRYPOINT /opt/indico/entrypoint.sh
```

Containers are **designed for running specific tasks and processes**, not for hosting operating systems. You create a container to serve a single unit task. **Once it completes the given task, it stops**. Therefore, the container life-cycle depends on the ongoing process inside of it. Once the process stops, the container stops as well.
A Dockerfile defines this process. It is a script made up of instructions on how to build a Docker image.

The Dockerfile is a text file that contains the instructions needed to create a new container image. These instructions may include identifying an existing image to use as a base, commands to run during the image creation process, and a command that will be run when deploying new instances of the container image. Docker build is the Docker engine command that uses a Dockerfile and triggers the image creation process.

Dockerfile components

The **FROM** statement sets the container image that will be used during the process of creating a new image.

For example, when you use the statement

- FROM mcr.microsoft.com/windows/servercore
  - the resulting image derives and has a dependency on the Windows Server Core base operating system image.
  - If the specified image is not present on the system where the Docker build process is running, the Docker engine will attempt to download the image from a public or private image registry.

https://docs.microsoft.com/it-it/virtualization/windowscontainers/manage-docker/manage-windows-dockerfile
https://docs.docker.com/engine/reference/builder/

Dockerfile components

Environment variables are supported by the following list of instructions in the Dockerfile:

- ADD, COPY, ENV, EXPOSE, FROM, LABEL, STOPSIGNAL, USER, VOLUME, WORKDIR, ONBUILD

The **COPY** instruction copies files and directories to the file system container. The files and directories must be in a path relative to the Dockerfile.

- COPY ["<source>", "<destination>"]

  Linux format: COPY test1.txt c:\temp\

  Windows format: COPY test1.txt c:/temp/

Dockerfile components

The **RUN** instruction specifies the commands to execute and capture in the new container image.

These commands can include items, including installing software, creating files and directories, and creating the environment configuration. Two forms exist:

- exec form

  RUN ["<executable>", "<param 1>", "<param 2>"]

- shell form

  RUN <command>

  the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows

La differenza tra il formato exec e il formato della shell è la modalità di RUN esecuzione dell'istruzione. Quando si usa il formato exec, il programma specificato viene eseguito in modo esplicito.

Di seguito è riportato un esempio del modulo exec:

FROM mcr.microsoft.com/windows/servercore:ltsc2019
RUN ["powershell", “New-Item", "c:/test"]

L'immagine risultante esegue il comando: powershell New-Item c:/test

Al contrario, l'esempio seguente esegue la stessa operazione in formato shell:
FROM mcr.microsoft.com/windows/servercore:ltsc2019
RUN powershell New-Item c:\test

L'immagine risultante ha un'istruzione di esecuzione di cmd /S /C powershell New-Item c:\test .

Dockerfile components

The **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. By default, EXPOSE assumes TCP. You can also specify UDP:

- EXPOSE 80/tcp
- EXPOSE 80/udp

Dockerfile components

There are **two types of instructions** that can define the process running in the container:

- **CMD**
  - CMD defines default commands and/or parameters for a container. CMD is an instruction that is best to use if you need a default command which users can easily override.
- **ENTRYPOINT**
  - ENTRYPOINT is preferred when you want to define a container with a specific executable.

https://phoenixnap.com/kb/docker-cmd-vs-entrypoint

Docker CMD defines the default executable of a Docker image. You can run this image as the base of a container without adding command-line arguments. In that case, the container runs the process specified by the CMD command.
The CMD instruction is only utilized if there is no argument added to the **run** command when starting a container. Therefore, if you add an argument to the command, you override the CMD.

ENTRYPOINT is the other instruction used to configure how the container will run. Just like with CMD, you need to specify a command and parameters.

Assume to launch the command sudo docker run [image_name] [parameter]

When there is no command-line argument (the parameter), the container will run the default CMD instruction and executes the CMD instruction. However, if you add an argument when starting a container, it overrides the CMD instruction.

In case of ENTRYPOINT, Docker does not override the initial instruction. It merely added the new parameter to the existing command.

Dockerfile components

The **CMD** and **ENTRYPOINT** instruction allows you to configure a container that will run as an executable. Two forms exist:

- exec form, which is the preferred form:

  CMD ["executable", "param1", "param2"]

  ENTRYPOINT ["executable", "param1", "param2"]

- shell form

  CMD command param1 param2

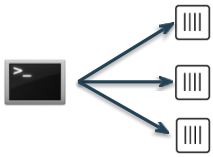
  ENTRYPOINT command param1 param2

  the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows

Docker Compose: Multi Container Applications

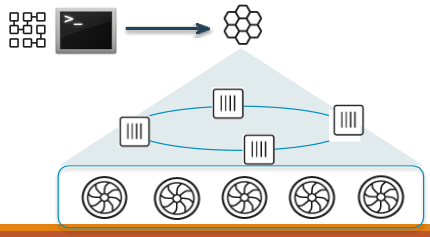
naive

- Build and run one container at a time
- Manually connect containers together
- Must be careful with dependencies and start up order



compose

- Define multi container app in compose.yml file
- Single command to deploy entire app
- Handles container dependencies
- Works with Docker Swarm, Networking, Volumes, Universal Control Plane



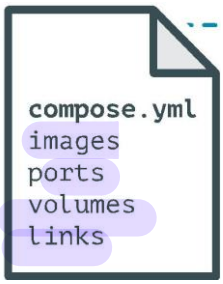
Docker Compose: Multi Container Applications

```
version: '2' # specify docker-compose version
# Define the services/containers to be run services:

angular: # name of the first service
build: client # specify the directory of the Dockerfile ports:
-   "4200:4200" # specify port forewarding

-   express: #name of the second service
build: api # specify the directory of the Dockerfile ports:
- "3977:3977" #specify ports forewarding

database: # name of the third service
image: mongo # specify image to build container from ports:
- "27017:27017" # specify port forewarding
```



Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.
Using Compose is basically a three-step process:
1.Define your app's environment with a Dockerfile so it can be reproduced anywhere.
2.Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3.Run docker compose up and the Docker compose command starts and runs your entire app. You can alternatively run docker-compose up using the docker-compose binary.

A db service is defined. It will be brought up and shut down whenever we run docker-compose up/down

You can to specify the alternate path to the Dockerfile in the build section of the Docker Compose file.

Docker Compose: Multi Container Applications

It allows running multi-container Docker applications reading instructions from a docker-compose.yml file

```
version: "2"
services:
  my-application:
    build: ./
    ports:
      - "8000:8000"
    environment:
      - CONFIG_FILE
  db:
    image: postgres
  redis:
    image: redis
    command: redis-server --save "" --appendonly no
    ports:
      - "6379"
```

Docker Compose provides a way to orchestrate multiple containers that work together. Examples include a service that processes requests and a front-end web site, or a service that uses a supporting function such as a Redis cache. If you are using the microservices model for your app development, you can use Docker Compose to factor the app code into several independently running services that communicate using web requests.

CONFIG_FILE includes environment variables.

--appendonly no start redis in no persistent storage mode

Redis provides a different range of persistence options:
•**RDB** (Redis Database): The RDB persistence performs point-in-time snapshots of your dataset at specified intervals.
•**AOF** (Append Only File): The AOF persistence logs every write operation received by the server, that will be played again at server startup, reconstructing the original dataset. Commands are logged using the same format as the Redis protocol itself, in an append-only fashion. Redis is able to rewrite the log in the background when it gets too big.
•**No persistence**: If you wish, you can disable persistence completely, if you want your data to just exist as long as the server is running.
•**RDB + AOF**: It is possible to combine both AOF and RDB in the same instance. Notice that, in this case, when Redis restarts the AOF file will be used to reconstruct the original dataset since it is guaranteed to be the most complete.

Basic Docker Commands

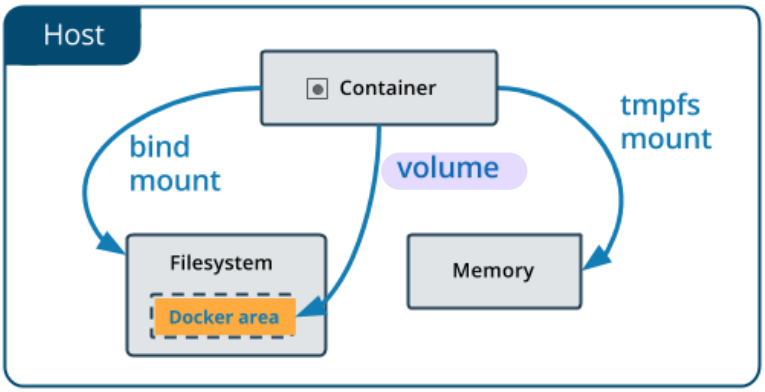
```
$ docker image pull node:latest
$ docker image ls
$ docker container run –d –p 5000:5000 ––name node node:latest
$ docker container ps
$ docker container stop node(or <container id>)
$ docker container rm node (or <container id>)
$ docker image rmi (or <image id>)
$ docker build –t node:2.0
$ docker image push node:2.0
$ docker --help
```

Docker Volumes

- Volumes mount a directory on the host into the container at a specific location
- Can be used to share (and persist) data between containers
  - Directory persists after the container is deleted
    - Unless you explicitly delete it
- Can be created in a Dockerfile or via CLI

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.

Why Use Volumes



Bind mounts have been around since the early days of Docker. Bind mounts have limited functionality compared to volumes. When you use a bind mount, a file or directory on the *host machine* is mounted into a container. The file or directory is referenced by its absolute path on the host machine. By contrast, when you use a volume, a new directory is created within Docker's storage directory on the host machine, and Docker manages that directory's contents.
The file or directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist. Bind mounts are very performant, but they rely on the host machine's filesystem having a specific directory structure available. If you are developing new Docker applications, consider using named volumes instead. You can't use Docker CLI commands to directly manage bind mounts.

While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:
•Volumes are easier to back up or migrate than bind mounts.
•You can manage volumes using Docker CLI commands or the Docker API.
•Volumes work on both Linux and Windows containers.
•Volumes can be more safely shared among multiple containers.
•Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
•New volumes can have their content pre-populated by a container.
•Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.
In addition, volumes are often a better choice than persisting data in a container's writable layer, because a volume does not increase the size of the containers using it, and the volume's contents exist outside the lifecycle of a given container.

If your container generates non-persistent state data, consider using a tmpfs mount to avoid storing the data anywhere permanently, and to increase the container's performance by avoiding writing into the container's writable layer.

Docker Networking overview

- One of the reasons Docker containers and services are so powerful is that you can connect them together, or connect them to non-Docker workloads.
- Docker containers and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not.
- Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.
- Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality

Docker Network Drivers

- **bridge:** The default network driver. Bridge networks are usually used when your applications run in standalone containers that need to communicate.
- **host:** For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly.
- **overlay:** Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons.
- **ipvlan:** IPvlan networks give users total control over both IPv4 and IPv6 addressing. The VLAN driver builds on top of that in giving operators complete control of layer 2 VLAN tagging and even IPvlan L3 routing for users interested in underlay network integration.
- .

•bridge: The default network driver. If you don't specify a driver, this is the type of network you are creating. **Bridge networks are usually used when your applications run in standalone containers that need to communicate.** See bridge networks.
•host: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. See use the host network.
•overlay: Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers. See overlay networks.
•ipvlan: IPvlan networks give users total control over both IPv4 and IPv6 addressing. The VLAN driver builds on top of that in giving operators complete control of layer 2 VLAN tagging and even IPvlan L3 routing for users interested in underlay network integration. See IPvlan networks.

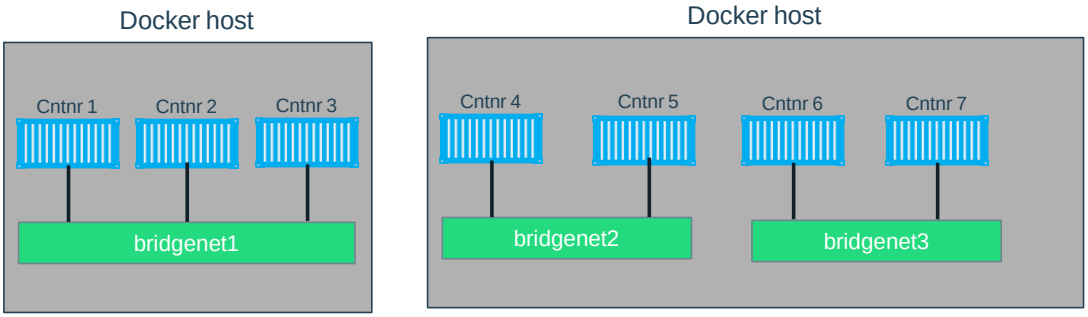
•**User-defined bridge networks** are best when you need multiple containers to communicate on the same Docker host.
•**Host networks** are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
•**Overlay networks** are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
•**Macvlan networks** are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
•**Third-party network plugins** allow you to integrate Docker with specialized network stacks.

Docker Network Drivers

- **macvlan:** Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack.
- **none:** For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services.
- **Network plugins:** You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors. See the vendor's documentation for installing and using a given network plugin.

•macvlan: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack. See Macvlan networks.
•none: For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services. See disable container networking.
•Network plugins: You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors. See the vendor's documentation for installing and using a given network plugin.

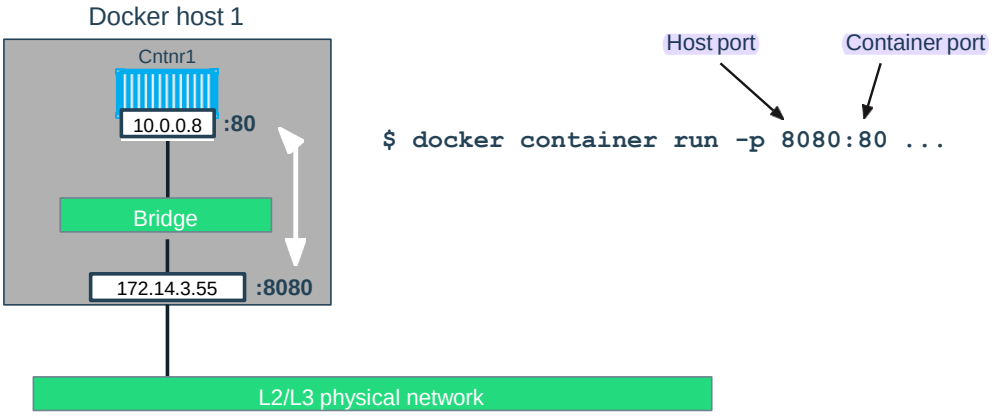
What is Docker Bridge Networking




```
docker network create -d bridge --name bridgenet1
```

You can specify the subnet, the IP address range, the gateway, and other options.

Docker Bridge Networking and Port Mapping



When you create or remove a user-defined bridge or connect or disconnect a container from a user-defined bridge, Docker uses tools specific to the operating system to manage the underlying network infrastructure (such as adding or removing bridge devices or configuring iptables rules on Linux). These details should be considered implementation details. Let Docker manage your user-defined networks for you.

Container Orchestration

- In order to scale containers, you need a **container orchestration** tool—a framework for managing multiple containers.
- Today, the most prominent container orchestration platforms are Docker Swarm and Kubernetes. They both come with advantages and disadvantages, and they both serve a particular purpose.
- Docker Swarm is an open-source container orchestration platform built and maintained by Docker.
- Kubernetes is an open source container orchestration platform that was initially designed by Google to manage their containers.

Docker Swam is straightforward to install, especially for those just jumping into the container orchestration world. It is lightweight and easy to use. Also, Docker Swarm takes less time to understand than more complex orchestration tools. It provides automated load balancing within the Docker containers, whereas other container orchestration tools require manual efforts.

Kubernetes has a more complex cluster structure than Docker Swarm.

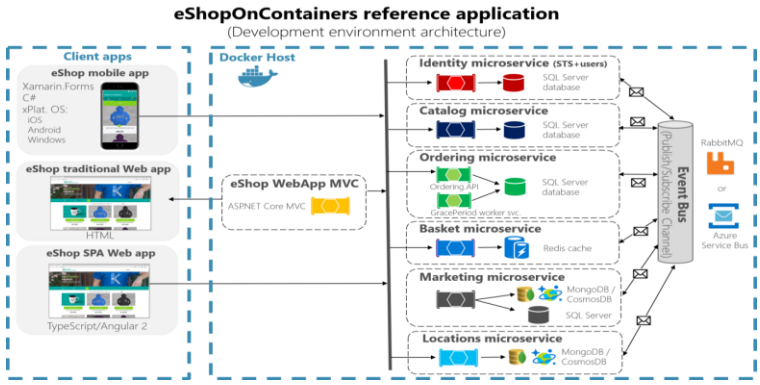
It usually has a builder and worker nodes architecture divided further into pods, namespaces, config maps, and so on.

Container Orchestration

Kubernetes advantages.

- It has a large open-source community, and Google backs it.
- It supports every operating system.
- It can sustain and manage large architectures and complex workloads.
- It is automated and has a self-healing capacity that supports automatic scaling.
- It has built-in monitoring and a wide range of available integrations.
- It is offered by all three key cloud providers: Google, Azure, and AWS.

An example of microservice-based system



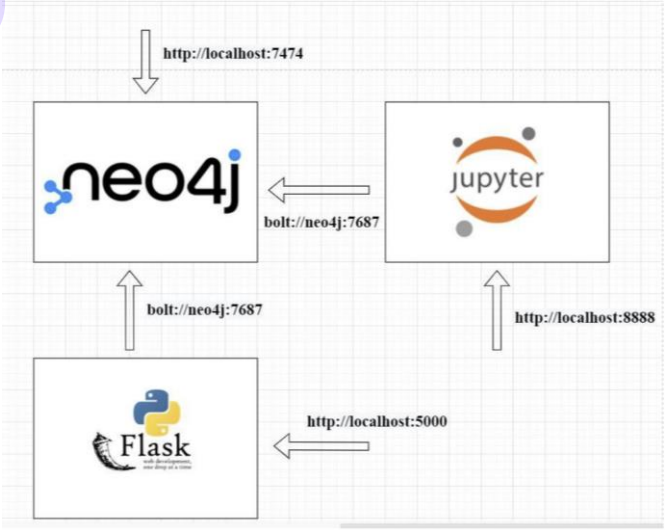
https://blogs.msdn.microsoft.com/dotnet/2017/08/02/microservices-and-docker-containers-architecture-patterns-and-development-guidance/

Example: multi-container application

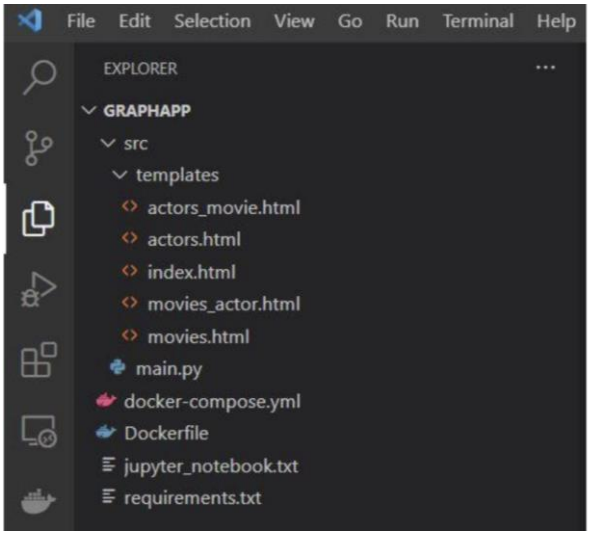
Docker

- **Flask:** micro-framework written in Python for the development of web applications, according to thr **Web Server Gateway Interface** (**WSGI**, pronounced *whiskey* or *WIZ-ghee*) protocol for web servers to forward requests to web applications, written in the Python.
- **Neo4j**: graph database software in which information is stored in the form of nodes, relationships and properties. The Cypher query language is used to interact with this database
- **Jupyter Notebook**: interactive computing environment supporting different programming languages. In this case, it is used for illustrative purposes to interact with Neo4j and enter data into the database, through instructions written in Python.

Architecture



Front-end



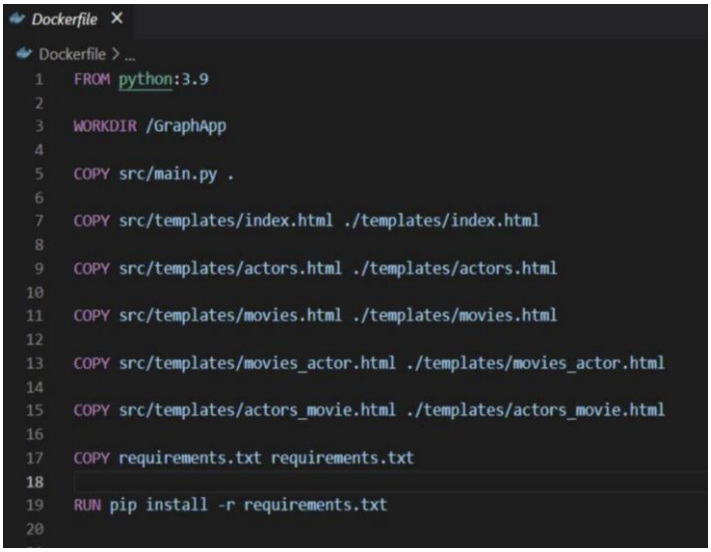
- Within the 'templates' directory we find the 'index.html' file, which represents the front-end of the application that manages the interaction with the user,
- the other files with the same extension are used to return results, from the server to the user, resulting from the possible actions that the user can perform through the interface accessible at the endpoint: http: // localhost: 5000.

Server-side

The 'main.py' file is the Python script that represents the server side of the application, which interacts with Neo4j through a special driver and manages requests coming from the client.

It is implemented by using Flask.

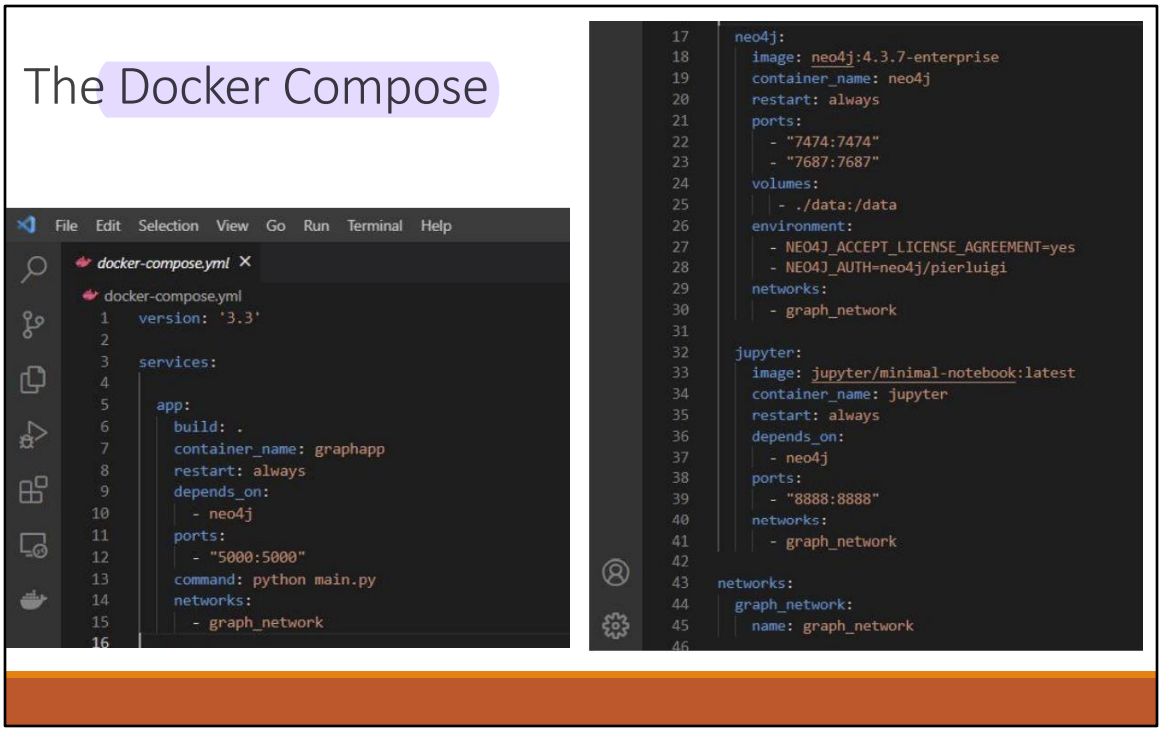
The Dockerfile



The 'Dockerfile' contains the commands for the creation of the image that includes the front-end and the back-end of the application, without the database, which is instantiated in another container.

- The Python image is downloaded from the Docker Hub, on which the new image.
- The 'GraphApp' directory is created in the container and set as working directory for the instructions below.
- The 'main.py' file, the 'templates' directory and the related files mentioned above, it is also copied the 'requirements.txt' file which contains the libraries necessary for the operation of 'main.py'
- The command “pip install "which installs the libraries specified in the 'requirements.txt' file.

The Docker Compose



To allow the containers to communicate with each other, a network is created virtual internal and it is possible to note, precisely, the presence of 'networks' in the definition of each service mentioned above.

- The first service refers to the 'Flask' application. With 'build' creates the image using the Dockerfile in the current directory. The dependence from the 'neo4j' service means that this container is started after the container containing the database.
- 'ports' shows the mapping between the external port of the host and the internal one of the container.
- The second service refers to Neo4j, whose image is downloaded from the Docker Hub. The first port is for the 'http' protocol and the second for the 'bolt' protocol, used from the Python driver for communicating with the database. Since this service is created, with 'volumes', a volume in such a way as to store data persistently. Furthermore, some environment variables are included.
- The third service refers to the Jupyter Notebook. The image is downloaded from the Docker Hub.

Jupyter notebook

The 'jupyter_notebook.txt' file contains the Python instructions that are executed inside the Jupyter Notebook in order to upload the data inside Neo4j.

docker compose up/down