Apache Kafka:

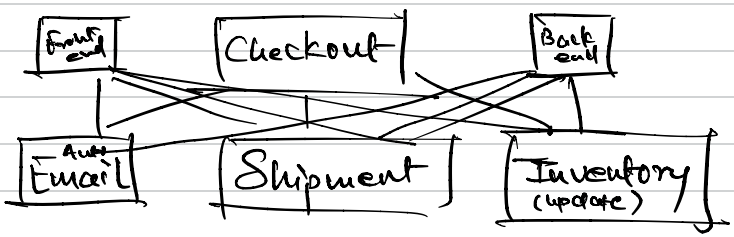
↳ Distributed streaming platform that allows for the development of real time event driven applications.

Allows developers to create applications that can:

- produce cousume (simultaneously)
- fast
- High accuracy with data records
- In Order (of occurrence)
- fall tauleraut (because its replicated)

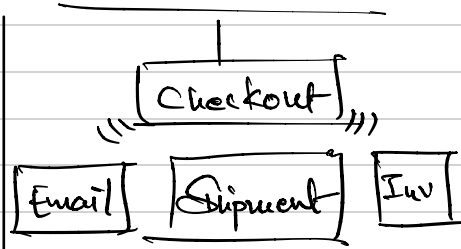
Before Kafka:

Ex: E-commerce



Every dept. is connected to every feature, if the system grows, it gets more complicated
↳ One change, needs multiple changes to be done

After Kafka:



Every time a check out happens, it will get streamed (not caring who's listening), its broadcasted basically, the other services: Email, Shipment, Inventory subscribes to the stream accordingly, they get the info they want, gets activated and work accord

One great usecase of kafka: Decouple system dependencies

How Kafka works?

↓

built on 4 core APIs:

① **Producer API:** allows your application to produce to make this streams of data. It creates the record and produces them to Topics.

↓

ordered list of events

↓

can persist disk

↓

can be saved for minutes / hrs / days / forever depending on how much storage you have.

② **Consumer API:** subscribes to one or more Topics. Listens and adjusts to that data. can subscribe to Topics in real time or can subscribe to the data which saved to the disk by the Producer.

Producer can directly produce to Consumer

↓

they can consume in real time and in original format as it was produced by the Producer.

(3) Streams API: very powerful leverages Producer and Consumer API so it will consume from Topic / Topics and it will analyze, aggregate or transform the data in real time and then produce the resulting stream into a Topic (either same or new).

(4) Connector API: enables developers to write connectors, which are reusable producers and consumers.

In a kafka cluster, many developers might need to integrate a same type of data source like MongoDB (ex.), not every single dev. have to write that integration, with a Connector API, it allows the integration to be written once and then all the devs need to do is to configure it in-order to get that data source in their cluster.

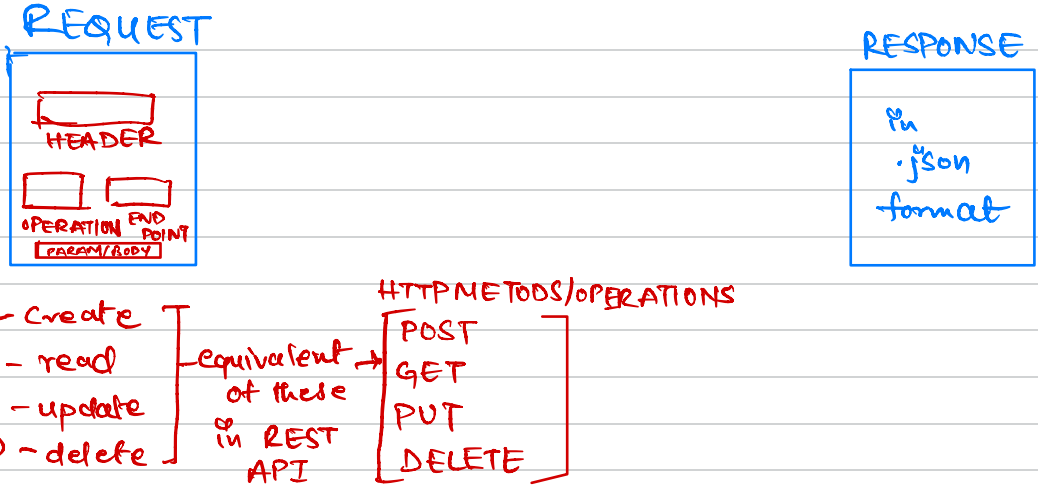
- REST API : fullform : Representational State Transfer API
  - API → Application Programming Interface

(SDK : Software Development Kit, calls APIs for you)

- REST API : ① is all about communication, client ↔ server.
  ② RESTful webservice, a service which uses REST API to communicate.

  Benefits:
  ① Simple and Standardized
  ② Scalable and Stateless
  ③ High performance / they support caching.

Ex: Icecream website allows employees to mix and match and work with its flavors through their website (through RESTAPI).



DOCKER:

- ↳ virtualization slice.
- ↳ makes developing and deploying applications much easier.
- ↳ Packages application with all the necessary dependencies, configuration, system tools and runtime.
- ↳ Portable artifact, easily shared and distributed.

Container: A standardized unit, that has everything that application needs to run.

- ↳ Standardizes process of running any service on any local dev environment.
- ↳ Easy to run different versions of same app. without any conflicts.

↳ Docker artifact includes everything the app needs, instead of textual instruction on how to install

Like old times

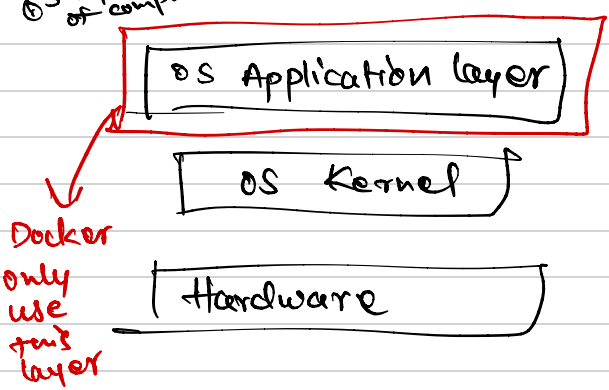
Configure the slug, everything is packaged inside the Docker artifact.

↳ Leads to No configuration needed on the server, less room for errors, the operation team just needs to run docker command to fetch and run the Docker artifacts. (installing Docker runtime on the server beforehand).

Docker vs VM:

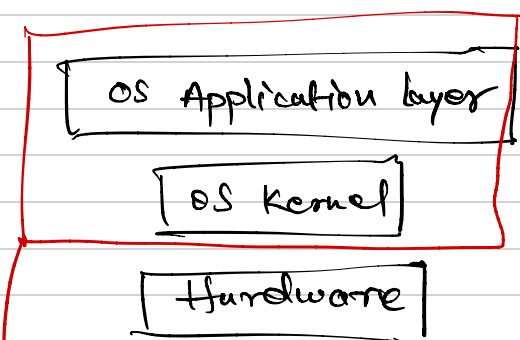
Docker

3 layers of computer Arch.



↳ i.e. the artifact only contains the OS application layer & services and apps installed on top of that layer.

VM



→ use both of these layers.

Docker	VM
② Docker Images are smaller, in MBs	VM images are larger ↳ in GBs.
③ Container takes seconds to start	takes minutes
④ Compatible with only Linux distros. (aka or Linux distribution)	Compatible with all OS

④ Ex: RedHat, Debian, Suse

Linux distros: term used to describe a specific version of Linux that is built from common Linux OS and includes additional applications.

- Linux based Docker Images cannot use Windows Kernel.

Most containers are Linux based

Docker Desktop: allows you to run Linux containers on Windows or MacOS.

↳ uses a hypervisor layer with a lightweight Linux distribution on top of it.

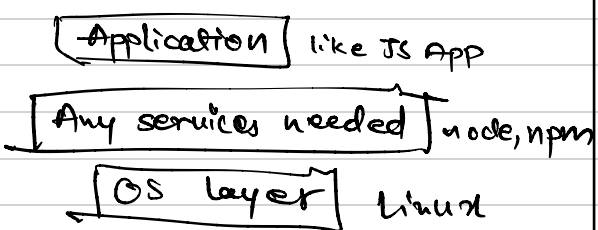
Docker Engine: A server with a long running daemon process "dockerd".

↳ programs running in the background without human interaction.

↳ manages Images and Containers.

Docker Images vs Docker Containers

- Executable application artifact
- Includes app source code, but also complete environment configuration
- Add environment variables, create directories, files etc.



→ Immutable template that defines how a container will be realized.

- Actually starts the application.
- Running instance of an image
  ↳ that's when the container environment is created.

Image downloaded to local/server
↓
run the image → container starts the application

You can run multiple containers from 1 image.

How to get these Images?

** Docker Registeries : storage and distribution system for Docker images.

- Official images available from application like Redis, Mongo, Postgres etc.
- Official images are maintained by the oss authors or in collaboration with the Docker community.

Dockerhub
↳ one of the biggest Docker Registry.

docker pull {name}:{tag} = Pull image from registry

docker run {name}:{tag} = Creates a container from given image and starts it.

Docker generates a random name automatically for the container, if you don't specify one.

-d or --detach = Runs container in background and prints the container ID.

docker logs {container} = view logs from service running inside the container.
(which are present at the time of execution).

Docker pulls image automatically, if it doesn't find it locally.

Port Binding

How to access containers?

Container Port vs Host Port

- Application inside container runs in an isolated Docker network.
- This allows us to run the same app running on the same port multiple times.

- We need to expose the container port to the host (the machine the container runs on)

Port Binding: Bind the container's port to the host's port to make the service available to the outside world.

① – Container runs on a specific port.
Ex:
NGINX runs on Port 80
Redis runs on Port 6379

② – Publish container's port to the host, using an extra Port (any localhost Port, 8080, ...)

③ — docker stop {container}: to stop one or more containers

-p or --publish= Publish a container's port to host

④ docker run -d -p {HOST-PORT}:{CONTAINER-PORT} {CONTAINER}
(multiple)

only 1 service can run on a specific port on the host

→ Choosing host port

↳ standard to use the same port on your host as container is using.

-a or --all = Lists all containers (stopped and running)

docker start {container} = Start one or more stopped containers.

--name = Assign a name to the container

→ Private Docker Registry:
↳ like AWS, Cloud Cloud, Microsoft Azure
↳ You need to authenticate before accessing the registry.
→ Nexus (popular artifact repository manager)

Docker Registry vs Docker Repository:

Docker Registry	Docker Repository
↳ A service providing storage. ↳ Can be hosted by a third party, like AWS, or by yourself. ↳ Collection of repositories	↳ Collection of related images with same name but different versions.

Docker Hub: is a registry.
↳ On Docker Hub, you can host private or public repositories for your applications

- **Dockerfile - Build Instruction:**

We need to create a "definition" of how to build an image from our application.
↳ Dockerfile.

→ Dockerfile is a text document that contains commands to assemble an image.

→ Docker can then build an image by reading those instructions.

① Build Dockerfile

** Structure of Dockerfile.

- Dockerfile start from a parent image or "base image".
- It's a Docker image that your image is based on.

You choose the base image, depending on which tools you need to have available.

① FROM

- Dockerfile must begin with FROM instruction.
- Build this image from the specified image.

Multi Layer Approach:

- Every image consists of multiple image layers.
- This makes Docker efficient because image layers can be cached etc.

Ex: FROM node:19-alpine

Image Environment Blueprint	DOCKERFILE
Install node	FROM node

① or ② or ③ ...

① RUN

▸ Will execute any command in a shell inside the container environment.

Ex: RUN npm install

② COPY

▸ Copies files or directories from <src> and adds them to the filesystem of the container at the path <dest>.

▸ While RUN is executed in the container, COPY is executed on the host.

FROM node:19-alpine

Ex: COPY package.json /app/

RUN npm install

↳ COPY <src> on our machine <dest> in the container.

↳ can copy whole directory too.

Ex: COPY src /app/

③ WORKDIR:

▸ sets the working directory for all the following commands.

▸ Like changing into a directory: "cd..."

Ex: WORKDIR /app

probably last command:

④ CMD ["executable", "parameter", ...]:

▸ The instruction that is to be executed when a

Docker container starts.

- There can be only one "CMD" instruction in a Dockerfile.

Ex: CMD ["node", "server.js"]

② Build Image:

Ex:

① docker build -t node-app:1.0 .

- -t or --tag: sets a name and optionally tag in the "name:tag" format
- node-app:1.0 → image name
- . → location of Dockerfile. '.' → current file/dir.

Remember: Docker image consists of layers.

- Each instruction in the Dockerfile creates one layer.
- These layers are stacked & each one is a delta of the changes from the previous layer.

③ Run as Docker container:

docker run -d -p host_port:container_port image name

Ex: docker run -d -p 3000:3000 node-app:1.0

Extra points to note:

- The commands in a dockerfile are not RUN/MAPPED in the container, only while building the image.
- If it were, spinning up new containers would take longer (as dependencies would have to be installed for each new container.
- Building a Docker Image creates an object (the final layer) which specifies folder structures and contains all the necessary files i.e. base image files, installed package files, your program code and all other artifacts created as a result of the instructions in your Dockerfile.
- Running the image creates a process (the container) to manage an ISOLATED instance with a folder structure and content continuing from where the image ended i.e. "run your code".

This is why two containers based on the exact same image may have different contents at run time depending on what your code is doing to the file system: docker exec [containerID]