Welcome aboard!

Ever faced issues while running or managing multiple versions of software, or with your code failing on another machine? What if there was a way to package up your code to ensure it runs smoothly in any environment, allowing you to ship it across any platform? Well, Docker is here to make your life easier 👊.

This article is associated with the `Docker Undocked` workshop, through which we will be trying to understand what Docker is and how it works before we dive right into a few examples and get our hands dirty to familiarise ourselves with its fundamentals.

Table of Contents

Prerequisites

Before we start do make sure that you are done with either one of the following steps:

Create an account on Docker Hub to follow along with the workshop on the Docker Playground or,

Install Docker on your local machine:

- Install for Windows
- Install for Linux
- Install for Mac

A note for Windows users: To run Linux Docker containers on a Windows 10 or 11 64-bit Home version, you will need to install and setup wsl version 2 with your preferred Linux distribution of choice. Before doing this make sure you have enable the `Windows Subsystem For Linux` and `Virtual Machine Platform` in the Windows Features Control Panel. Also make sure virtualization is enabled by checking your Task Manager. Once this is done, run `wsl --install` on your powershell and after successful installation of the default Ubuntu subsystem, proceed to install Docker Desktop on your host.

Also, all the code for this workshop can be found in this repo: https://github.com/achyuthcodes30/Docker-Undocked-Code

Before delving into Docker, let's explore some of the related technologies and foundational concepts that set the stage for containerization and Docker's inception and growth.

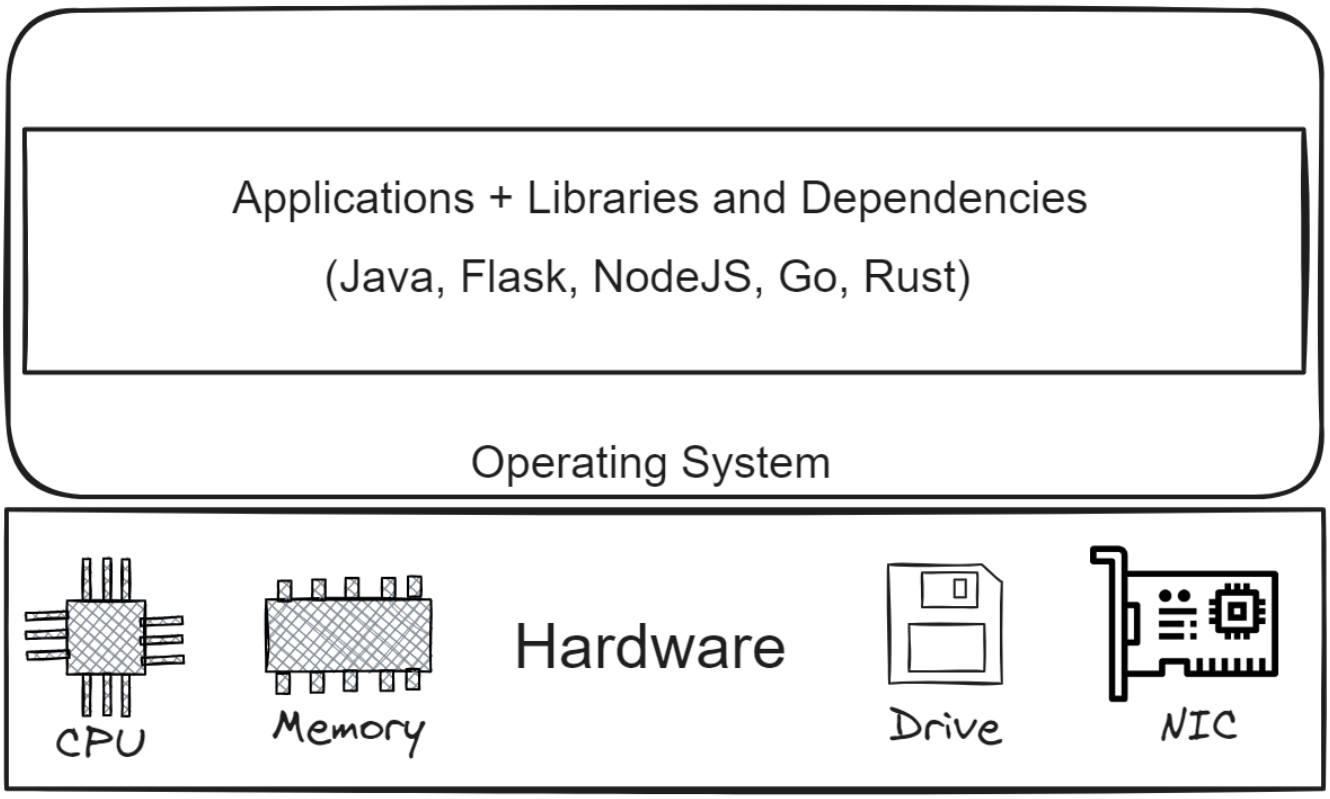
Virtualisation and the Cloud

Let us take a look at the state of things back in the 80s and 90s, back when organisations hosted and managed their applications and services on dedicated physical servers or hosts in on-premises data centers. IBM was a big player and mainframe computers were prominent for hosting centralized applications.

The term `server` here refers to a physical system or machine, but can also mean, in modern technology, a piece of software that is responsible for providing resources,services or some functionalities to other machines or clients requesting them over a network.

Say we are a small organisation that wants to set up and maintain our own traditional physical server. To host services and applications on our own dedicated phyiscal server that we want to maintain, we would be responsible for configuring and monitoring the necessary hardware components like CPU, memory (RAM) and external storage (HDDs and SSDs), power supply, network interfaces and ports, circuitry etc.

Next, we set up a host operating system of our choice with a set of device drivers (disk IO, audio, display etc.) and most importantly, the operating system kernel to manage physical resource allocation and facilitate interaction between software components and the server's hardware. Then, we install the required dependencies and libraries for each of our services/applications and start working with them.



Now, what happens when application dependencies clash, or disruption in one application causes hiccups in another resulting in multiple applications going down, or what if we have to run multiple flavours and versions of operating systems?

We could provision multiple such physical servers, each dedicated to hosting and running a single service or application along with its dependencies or its own host operating system and scale out in this fashion as well. However, this wouldn't be cost efficient and we do may not have the space to house these servers either. Also, while we are in complete control of the server, the complete responsibility for maintenance, security and scalability rests upon us.

So what if, instead, there was a way to isolate each of these applications and their dependencies in a separate environment or operating system of its own, making effecient utilization of one server's resources and compute. This is exactly what we try to accomplish with the help of virtualisation.

`Virtualization` is a technology that abstracts and decouples physical resources, such as CPU, memory, storage, and network, from the underlying physical hardware and present them in a way that allows multiple isolated virtual environments to coexist on a single physical system. Some common types include application, network, desktop and storage virtualisation and server virtualisation which allows us to run multiple isolated virtual servers running their own guest operating system and a set of applications and their dependencies on a single physical server.

Each virtual instance in this case, or `virtual machine`, operates as self-contained entity with its own virtualized CPU, memory, storage, and network interfaces, which is essentially abstracted away and allocated from the physical host machine. These VMs can also have access to devices for display, audio etc. This is all possible with the help of a `hypervisor`.

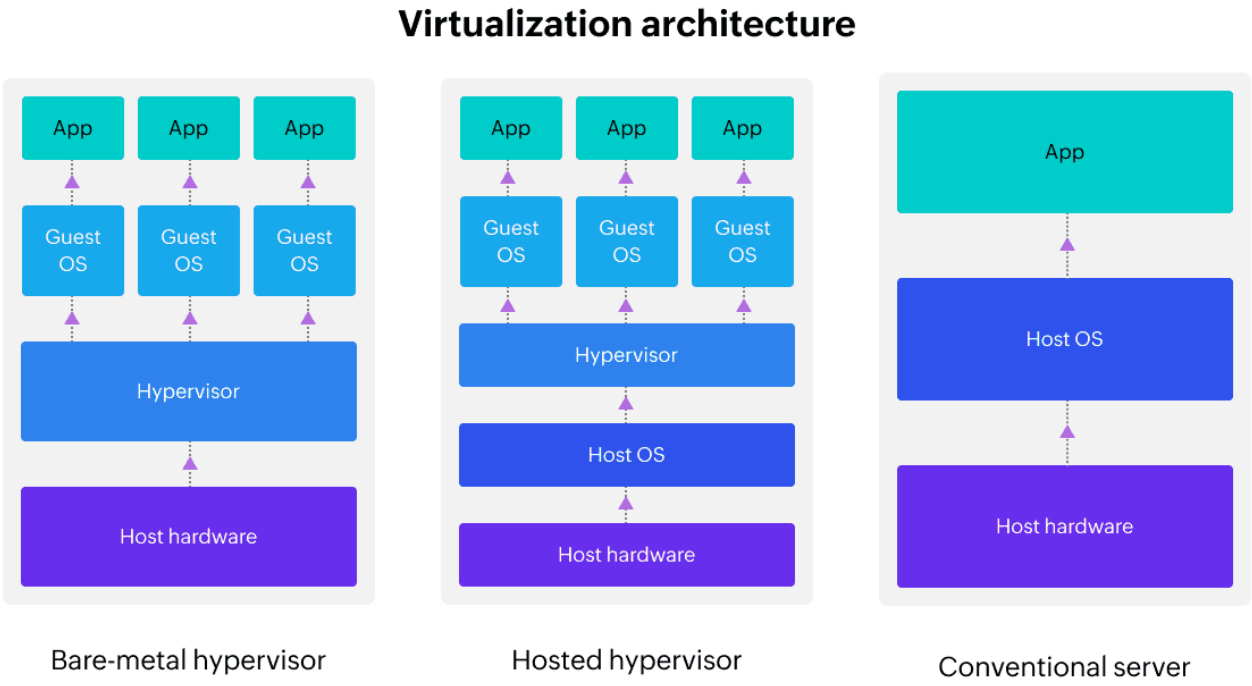
A `Hypervisor` is a piece of software or firmware that creates and manages virtual instances, manages the allocation of physical resources to each one and ensures their isolation. The hypervisor virtualizes the physical resources (CPU, memory, networking, storage) of the host machine and presents them to each VM as if they were dedicated resources.

There are two types of hypervisors that achieve the same in slightly different ways.

A `Type-1 hypervisor` or `Bare-Metal hypervisor` directly interacts with the underlying physical resources without the need for an operating system. It is more performant as it eliminates the overhead of an OS layer while making decisions like scheduling and storage access on it own and therefore provides a more streamlined and efficient virtualization environment. It is commonly used in data centers, cloud infrastructure, and server virtualization environments. Examples include VMware ESXi, Bare Metal Microsoft Hyper-V, XenServer and open source KVM.

A `Type-2 hypervisor` or `Hosted hypervisor` is generally less performant as the hypervisor relies on a host operating system layer. When an application or the guest OS in the VM issues a system call or requests access to a physical resource, the hypervisor intercepts these requests. The hypervisor communicates with the host OS to manage the sharing and allocation of physical resources among VMs. These are commonly used for development, testing, and desktop virtualization scenarios where optimal performance is not a primary concern. Examples include Oracle VirtualBox and VMWare Workstation.

Let us go ahead and try out a VM for ourselves. Visit the Docker Playground, login and click on the `Start` option. Once your session is ready add a new instance and you will have your very own Linux Virtual Machine to play around with (it also has a Docker set up).



Next, let us talk about the Cloud. What exactly is it?

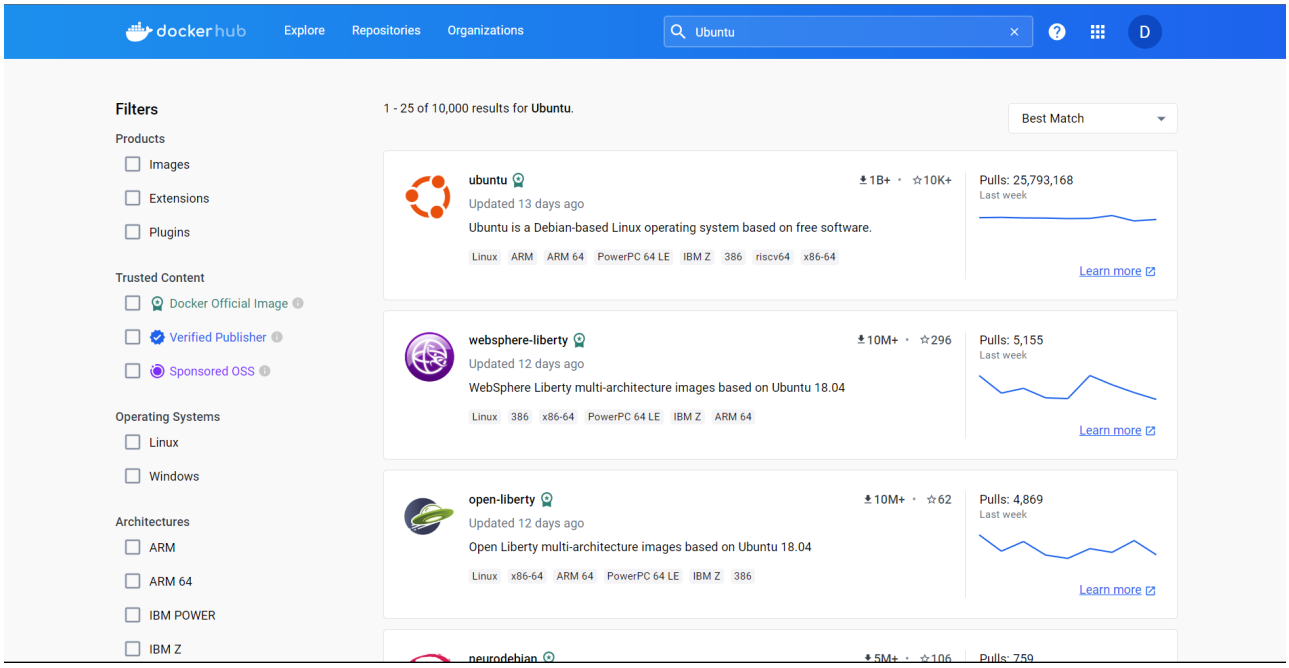
While it does share the name with what you see above, when people refer to "the cloud," they are talking about a vast, interconnected network of servers and services accessible over the internet, with the specific details of the infrastructure hidden behind a metaphorical cloud. Users can utilize computing resources over a network or the internet on-demand without worrying about managing the underlying hardware and configuration.

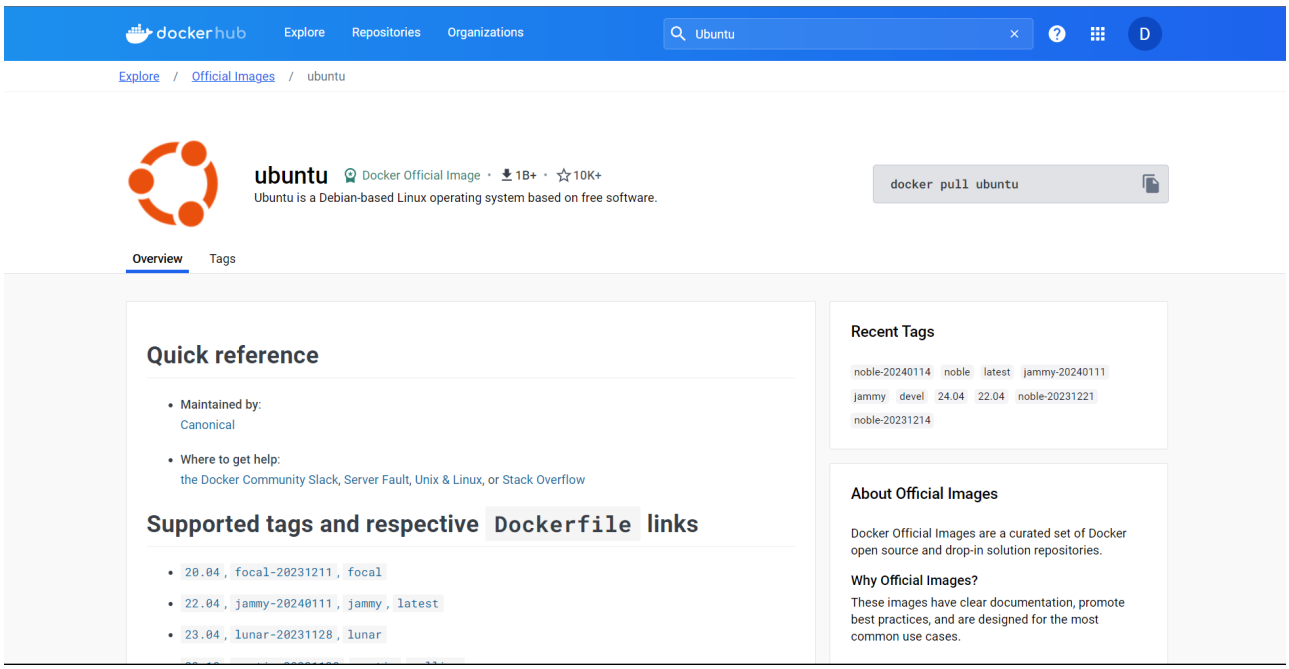
The cloud market continues to grow rapidly, with organizations increasingly adopting cloud services for flexibility, scalability, and cost-effectiveness. A few major cloud service providers include Amazon Web Services, Microsoft Azure and Google Cloud Platform.

Getting our hands dirty

Let us explore the Docker Hub Registry and pull our first Docker image. We will try to run an Ubuntu environment on your host machine (If you are on the Docker Playground, we will be doing this on your Alpine Linux Virtual Machine). Let's see how we can achieve this:

- Go to Docker Hub. This is a popular cloud based container registry where developers can find and share container images.
- Search for 'Ubuntu' in the search bar and click on the Ubuntu image.





- The `tags` field specifies human readable identifiers for specific version or variants of an image.
- Following the command provided on the page, let us run `docker pull ubuntu` on our host.
- We can verify that the image was build successfully by using the `docker images` command to list all the Docker images on our local respository.

- Next, we spin up a container from this Ubuntu image by running `docker run -it --name ubuntu_container ubuntu`. The -it flags combine the "interactive" and "pseudo-tty" options to provide us with an interactive terminal emulation within the container which will then allow us to interact with its processes.
- That was simple wasn't it? We were able to set up an Ubuntu environment on our host machine in just a couple of minutes.
- Now, exit the container using `Ctrl + c` or `Ctrl + p Ctrl + q` and run `docker stop <container_id_or_container_name>` to stop the container and `docker start` to start it again. Note that in the case of no-name containers, providing just the first few unique characters of the container ID will suffice.
- Run the `docker ps -a` command to list all containers on your machine regardless of their state. The `docker ps` command by itself returns information of only those containers that are alive/running.
- We can now easily spin up multiple containers from the Ubuntu image using the `docker run` command.
- We can also use the `docker exec <container_name_or_id>` command to run a new command inside the specified running container.
- We can also just run `docker run` without running `docker pull` and Docker will pull the image from the registry service as it will not be able to find the image on the local respository.

Containerization and Docker

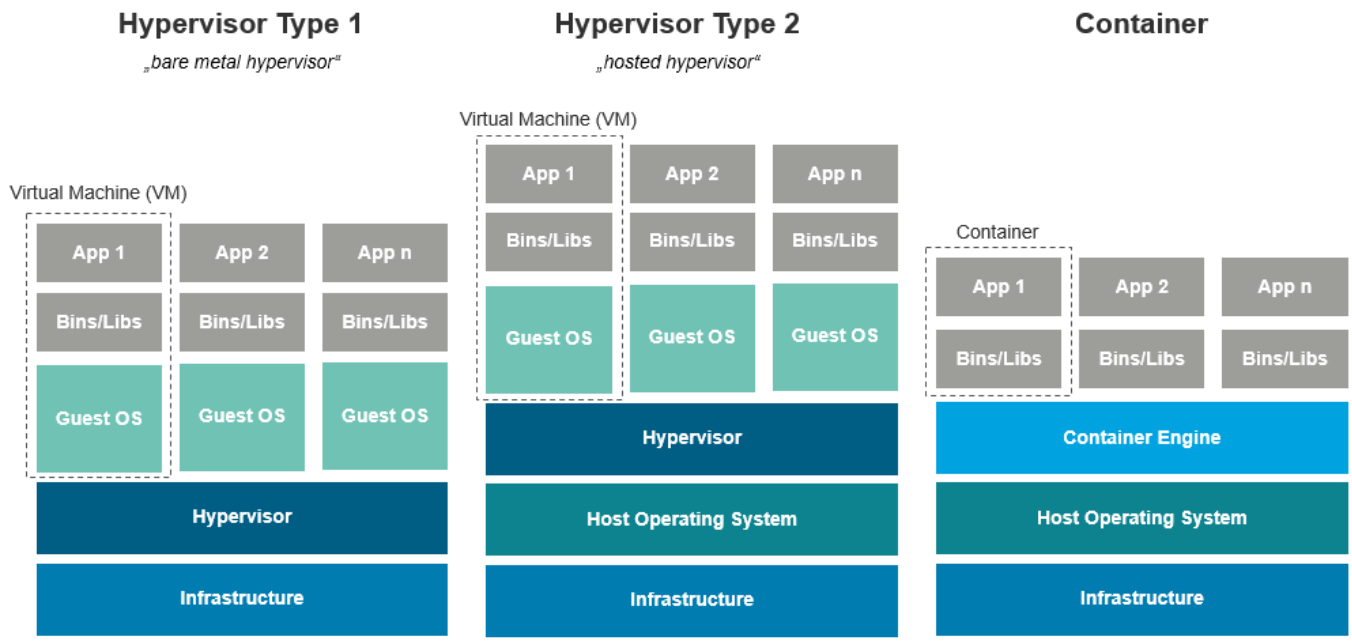
In a traditional Infrastructure as a Service (IaaS) model, applications are deployed on virtual machines on the cloud. Each VM consists of a complete operating system, along with the application and its dependencies. While this approach provides isolation and flexibility, it comes with overhead in terms of resource consumption, as each VM carries its own OS. Scaling applications requires scaling entire VMs, even if the changes are minimal.

`Containerization` addresses these challenges by encapsulating an application and its dependencies into a lightweight, standalone unit known as a `container`. Containers share the host OS kernel, making them more efficient and faster to deploy than VMs.

Containers are essentially processes running on the Linux Kernel that are isolated with the help of Linux namespaces provide isolated namespaces for processes, networking, filesystem, and other resources. and Cgroups for resource management, allowing limits to be set on CPU, memory, and other resource usage for each container. You can actually see container processes running on your machine with the `ps` command or the `pstree` command as opposed to processess running on a guest OS inside a Virtual Machine which are completely isolated.

In the words of Jérôme Petazzoni,

"Containers are processes, born from tarballs, anchored to namespaces, controlled by cgroups."



Now, type in the following commands in the docker playground VM - `apk add neofetch` followed by `neofetch`.

We see that we are running an Alpine Linux instance (as you might have inferred from the `apk` package manager). However, when you check `pstree` you can see that it was inited by `dockerd`, which is the Docker daemon and `containerd`, a high level container runtime. This means we are inside an Alpine Linux Docker container in a VM! A lot of virtualization indeed and this is slowly becoming the norm.

But what is Docker and how is it related to any of this?

Docker is a popular open-source container platform consisting of a set of tools designed to simplify the development, deployment, and operation of applications using containers.

But why is it so popular? How is it relevant in today's industry?

Docker is instrumental in microservices architecture, where applications are composed of small, independent services. Each microservice can run in its own Docker container, enabling isolation, scalability, and easy management of microservices. This allows developers to package and deploy microservices independently, making it easier to update and scale specific components without affecting the entire application.

Docker's containerization technology provides consistency across different environments on the Cloud. Docker containers are also integral to DevOps teams as they are consistent, portable and resource efficient, making them a viable solution for more reliable and stable releases through Continuous Integration and Continuous Deployment (CI/CD) pipelines.

Now that we have a basic understanding of what Docker is and why we use it, let us learn more about it's fundamentals.

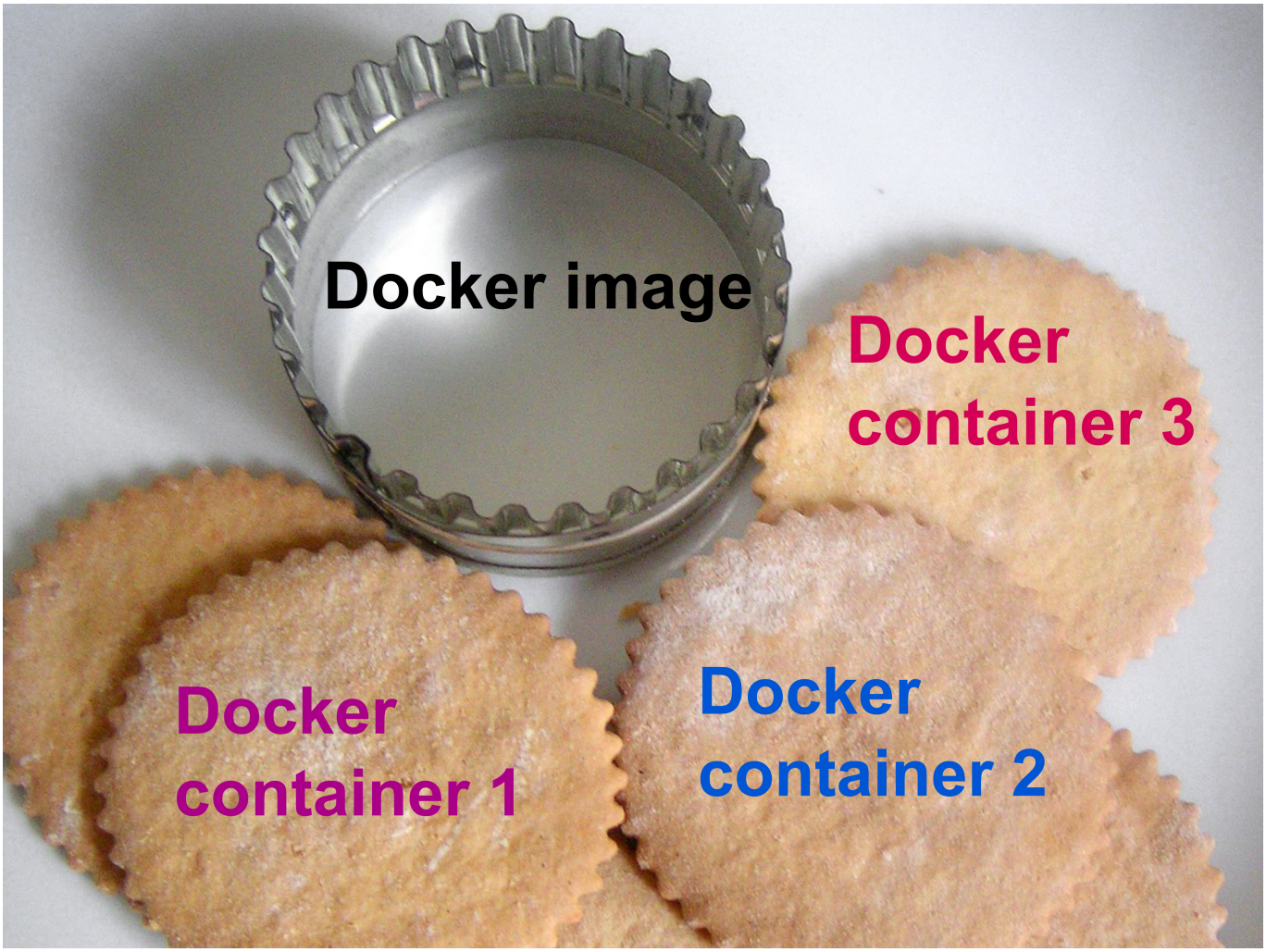
Docker images vs containers

The process of containerising an application with Docker involves pulling and building images and spinning up containers. But what exactly is an image and what do we mean by the term 'container'?

A Docker image is a read-only, executable template containing a set of instructions, libraries, dependencies, configurations and files packaged together to create a runtime instance or a container environment. It is a snapshot that references a set of read-only or immutable layers with a base layer that defines the environment or the minimal operating system. These layers are stacked on top of each other to form the container's root file system. Docker images can be shared and stored in multiple locations and can be pulled from/ pushed to a public or private repository (The Docker Hub Registry for example).

A Docker container is a lightweight, isolated, running instance of an image. Docker images become containers when they run on the Docker engine. The instructions and code, dependencies and libraries that were packaged up as read-only, shared layers during the build process of the image form the base of the containers file system as mentioned earlier. Docker containers create a thin, top writeable layer on top of its base to perform modifications to the container file system and data. Now we can run commands in this container, assign and bind it to a port and perform other actions as opposed to a Docker image which is just a snapshot.

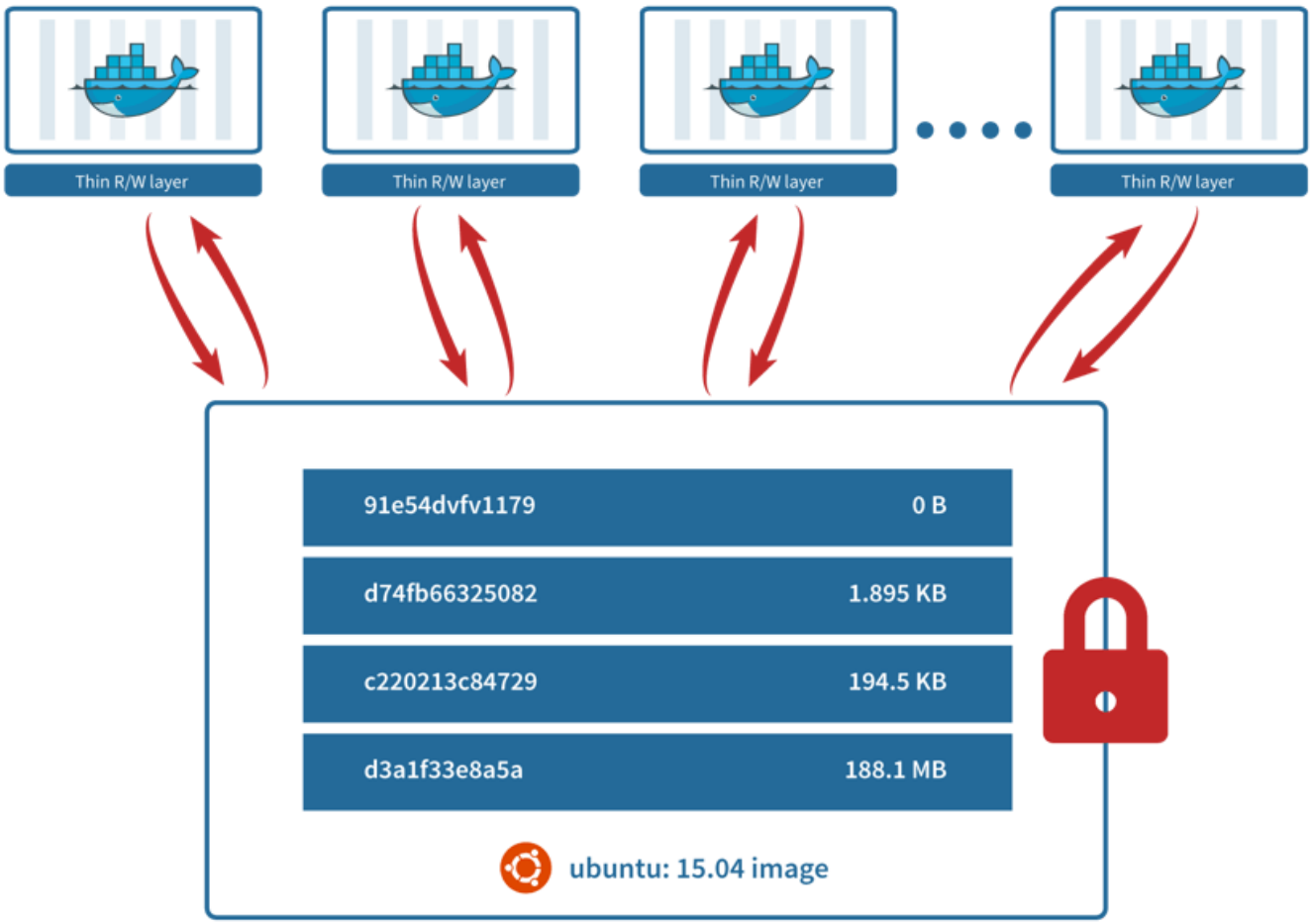
If you are familiar with Object Oriented Programming, you can think of a Docker image as a class and a container as an instance of a class or a Docker image, or you can think of Docker images as cookie cutters, using which one can prepare multiple cookies or containers.



Docker containers can be made lightweight thanks to Union mounts/UnionFS and the Copy-On-Write strategy. Union mount is a type of a filesystem that can create an illusion of merging contents of several directories into one without modifying its original (physical) sources. This can be useful as we might have related sets of files stored in different locations or media, and yet we want to show them in single, merged

view. In our case, we want to present the Container's writeable layer and the read-only layers of the image as a merged view giving the writeable layer more priority.

Many images that we use to spin up our containers are quite bulky whether it's ubuntu with size of 72MB or nginx with size of 133MB. It would be quite expensive to allocate that much space every time we'd like to create a container from these images. Thanks to union filesystem, Docker only needs to create thin layer on top of the image and rest of it can be shared between all the containers. This also provides the added benefit of reduced start time, as there's no need to copy the image files and data.



But what happens when we want to make changes to files in the read-only layer? That's where the Copy-On-Write strategy comes to play. The contents of the read-only layer are not copied into the container writeable layer until the first attempt to write to it. The files can then be modified safely and the changes are visible in the writeable layer.

Can we create our own images?

YES! That is exactly what we are going to explore now by dockerising a simple Node + Express server.

Before we start though, let us make sure we have installed version of NodeJS that is >=16. Run the `node -v` command on your shell to check the version of NodeJS running on your system and make a note of it. If your system does not have NodeJS installed, you can visit the NodeJS Website to install NodeJS and npm or simply run `apk add nodejs npm` if you are on the Docker playground. You can also run `apt install nodejs npm` if you are running a Debian based Linux distribution like Ubuntu.

Once you have the necessary installations, create a project directory with the `mkdir` command and switch directories using the `cd` command.

Once you are inside the project directory, run npm init -y to initialize a new project and create the package.json file. Then run the following command `npm install express`

Now open the project on your preferred editor of choice and create a file called `index.js` (You can also do this with the `touch` command if you are on a Linux machine).

Type in the following code inside `index.js`:

```
const app = require("express")();

app.get("/", (req, res) => {
  res.json({ Page: "This is the homepage" });
});

app.get("/first", (req, res) => {
  res.json({ Page: "This is the first page." });
});

app.get("/second", (req, res) => {
  res.json({ Page: "This is the second page." });
});

app.all("*", (req, res) => {
  res.status(404).send("Oops! Cannot find that page");
});

app.listen(3000, () => {
  console.log("Listening on port 3000");
});
```

Now, to make sure this works locally, type in `node index.js` in the terminal inside the project directory. Then either visit `localhost:3000` on a browser or run `curl localhost:3000` on the command line. To check if this works from the Docker Playground, expose the port 3000 of the VM with the `OPEN PORT` option provided above and follow the link that is generated to verify that our code works.

Once we have confirmed that the app works fine locally, we will proceed towards containerizing this app by creating our own custom image. To do this, go ahead and create a file called `Dockerfile` in the project directory.

A Dockerfile is a text file inside which we list instructions for the docker daemon to follow when building Docker images. Each layer of a Docker image can be thought of as the byproduct of the execution of an instruction inside this file. It can be thought of as a recipe required to prepare a Paneer sandwich one layer at a time.

Inside the `Dockerfile`, type in the following:

```
FROM node:latest
WORKDIR /app
COPY . /app
RUN npm install
EXPOSE 3000
CMD ["node","index.js"]
```

Let's talk about what each line in this code does:

- FROM `<base_image:tag>`: This line specifies a base image to build our custom image from. It does the work of building the node environment for us.
- WORKDIR `<directory>`: This directive specifies the working directory for the subsequent instructions in the Dockerfile and inside the containers that we spin off of the image built.
- COPY `<path_to_copy_from> <path_to_copy_to>`: This line copies the contents of directory to the specified directory while building the image and inside the container. In this case we copy everything from our current directory to the working directory `/app` inside our container.
- RUN `<command>`: Here we specify the command to run while building the image and the results of this command will be reflected inside of any container we spin up from this image.
- CMD `<command>`: The default command to run when we start ant container that we spin up from our custom image.
- EXPOSE `<port>`: This line informs docker that all derived containers listen on the specified network port at runtime.

Next, create a `.dockerignore` file in the same directory that excludes certain files/directories from the built image and its containers. Before the build context for an image is sent to the Docker daemon, it searches for this file in the root directory of the build context so it knows which paths to ignore during the build process. This helps in reducing the size of containers. It is pretty similar to the `.gitignore` file for Git.

Inside the `.dockerignore` file, type in the following:

```
node_modules
Dockerfile
.dockerignore
```

We do this to ensure `node_modules` is not unnecessarily copied each time we build the image and so that we can hide the Dockerfile and .dockerignore file from others building our image and also prevent them from taking up unnecessary space inside our containers.

Once that is done, run the following command in the project directory: `docker build -t <image_name>:<image_tag> .`

Where we specify the image name and tag after the `-t` flag and the directory inside which our Dockerfile resides to build our custom image.

We can then check if the image was created by running `docker images` to list all images present on our machine or the specified repository.

To spin up a container from this image, we need to run the command `docker run --name <container_name> -p <host_port>:<container_port> <image_name>:<image_tag>` which tells docker what the container name should be and which network port of the host machine should point to the specified network port of the container and what image should be used to spin up the container.

Then either visit `localhost:<host_port>` on a browser or run `curl localhost:<host_port>` on the command line and verify that you have successfully containerized and run your own application. If you are on the Docker playground follow the same steps as we did before dockerising our web server and expose the host port that we mapped to the container port 3000 using the `OPEN PORT` option and follow the link to verify that our container works fine.

Voila! We have successfully containerised a Node + Express server and it can now be run in any environment without any hiccups.

But let us try something else here. In the index.js file add another route with the following code:

```
app.get("/third", (req, res) => {
  res.json({ Page: "This is the third page." });
});
```

Now, check the result on `localhost:<host_port>`. You will notice that there is no change and we get the 404 message when we try to view `localhost:3000/third`. This is because Docker images are read-only as you read earlier and once built they are locked in and their contents cannot be changed.

So to view these changes, build the image again using the same command as before with a new name. We can see that the image is being built with the exact same steps as before and `npm install` runs again to install packages even though we did not install any new packages. This is unnecessary and inefficient.

All or most instructions in a Dockerfile create a layer in the build process of an image and these layers are shared and cacheable. This means that if nothing has been changed in the context of a certain instruction, that layer does not have to rebuilt and the cached result of previous build can be reused. The instructions in a Dockerfile are executed sequentially from top to bottom and this is the order in which layers are built. Therefore, only the instruction for which the context has been modified, and subsequent instructions, need to be executed again and hence only those layers need to be rebuilt. The layers before this point can reuse their respective caches.

To exploit this feature and optimize our build, we can rewrite our Dockerfile like this:

```
FROM node:21-alpine3.19
WORKDIR /app
COPY package*.json /app
RUN npm install
COPY . /app
EXPOSE 3000
```

```
CMD ["node","index.js"]
```

This ensures that we do not unnecessarily reinstall packages each time we build the image making the build process quicker and more efficient. We also specifiy a tag/version for our node environment instead of the "latest" tag to ensure that our app remains stable regardless of any updates to the base image. We also run `npm ci` instead of `npm install` for a clean installation of the exact versions of dependencies in the `package-lock.json` file.

Some more Docker commands

Here is a cheatsheet containing some useful Docker commands:

- `docker version`: Version of the Docker Engine running on your host. -`docker --help <command>`: Provides more information for a particular command.
- `docker images`: Lists all images that are available locally.
- `docker rmi <image_id_or_name>`: Deletes the specified image.
- `docker image prune`: Removes all unused images i.e. images not being used by any containers.
- `docker build`: To build an image from a Dockerfile. Use the `-t` flag to specify an image name and tag and provide the path to the Dockerfile.
- `docker pull`: Pull an image from Docker hub or a specified registry.
- `docker push`: Push an image to Docker hub or a specified regsitry.
- `docker search`: Search Docker hub for an image.
- `docker start|stop|restart <container_id_or_name>`: Start/stop/restart a container. -`docker run <image_id_or_name>`: Spin up and start a Docker container based off of the specified image. Use the `--name` flag to assign a name to the container, `-it` for an interactive shell inside it or `-d` to run it in detached mode.
- `docker exec -it <container_id_or_name>`: Run a new command inside the container.
- `docker inspect <container_or_image_id_or_name>`: Inspect a running container.
- `docker ps`: List running containers. Use `-a` flag to list all containers.
- `docker rm <container_id_or_name>`: Delete a container.

Docker Compose and Multicontainer Applications

While we did succeed in dockerising a simple web server, that was just a single container and when we divide a large application into microservices we may need to create and manage multiple containers. This will allow us to scale them differently in a cluster environment like Kubernetes (an open source container orchestration services) and manage versions for each service in isolation.

It can become time consuming and tedious to spin up and manage multiple containers individually and deal with the networking to get these containers to talk to each other from scratch.

Fortunately Docker provides a very simple way to spin up and manage multiple containers with Docker Compose.

Docker Compose is a tool that helps you define and share multi-container applications. With Compose, you can create a YAML file to define the services and with a single command, you can spin everything up or tear it all down.

YAML Ain't Markup Language (YAML) is a data serialization language that can be easily read by us. Data serialization is the process of converting data or an object into a stream of bytes for easier storage and transmission. Let us understand this with the help of another human readable, data serialization format, JSON or Javascript Object Notation.

JSON represents structured data in Javascript Object syntax that can easily be read by us. Even though it closely resembles JavaScript object literal syntax, it can be used independently from JavaScript, and many programming environments feature the ability to read (parse) and generate JSON. While we most certainly can work with a simple format such as JSON, it needs to be serialized into a JSON string so it can be transmitted over a network or stored in files. To do this, we use `JSON.stringify()` and then we deserialize the JSON string with `JSON.parse()` to reconstruct the Javascript Object defined by the string.

Coming back to YAML, it is often used to write configuration files such as our Docker Compose file in the DevOps world, but its object serialization abilities and support for multiple programming environments make it a viable replacement for JSON as it can be considered a superset and is easier to read. While we won't be diving any deeper into YAML in this article, here is a good read to familiarise yourself more with its specs and syntax - https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started

Let's see how we can build multicontainer applications with Docker Compose. To start with, go ahead and clone this repo where with the `git clone` command: https://github.com/achyuthcodes30/Docker-Undocked-Code.

Then, move to the `compose_example` directory and open the `compose.yaml` Docker Compose file in your preferred editor of choice.

```
version: "3.9"

services:
  mongodb:
    container_name: mongo_container
    image: mongo:6
    restart: always
    healthcheck:
      test: ["CMD", "mongosh", "--eval", "db.runCommand({ ping: 1 })"]
      interval: 10s
      timeout: 10s
      retries: 5
    volumes:
      - mongodb_data:/data/db

  backend:
    container_name: node_container
    build:
      context: ./backend
      dockerfile: Dockerfile
    ports:
    environment:
      PORT: 3000
      WHICH_DB: mongodb://mongodb:27017/composexample
    depends_on:
      mongodb:
```

```
        condition: service_healthy

  frontend:
    container_name: react_container
    build:
      context: ./frontend
      dockerfile: Dockerfile
    environment:
      VITE_PORT: 8080
    depends_on:
      - backend

  nginx:
    container_name: nginx_container
    build:
      context: ./nginx
      dockerfile: Dockerfile
    ports:
      - 8000:80
    restart: always
    depends_on:
      - frontend
      - backend

volumes:
  mongodb_data: {}
```

This is what we find. Let us try unwrap what each line does here:

- `version` : The top-level version property is defined by the Compose Specification for backward compatibility. It is only informative and the Compose specification has been merged with versions 2.x and 3.x of the Compose file format. It is however common practice to define this although Docker Compose V2 that uses the Go plugin (`docker compose` and not `docker-compose`) searches for a compose.yaml file and ignores this line and selects the latest version.

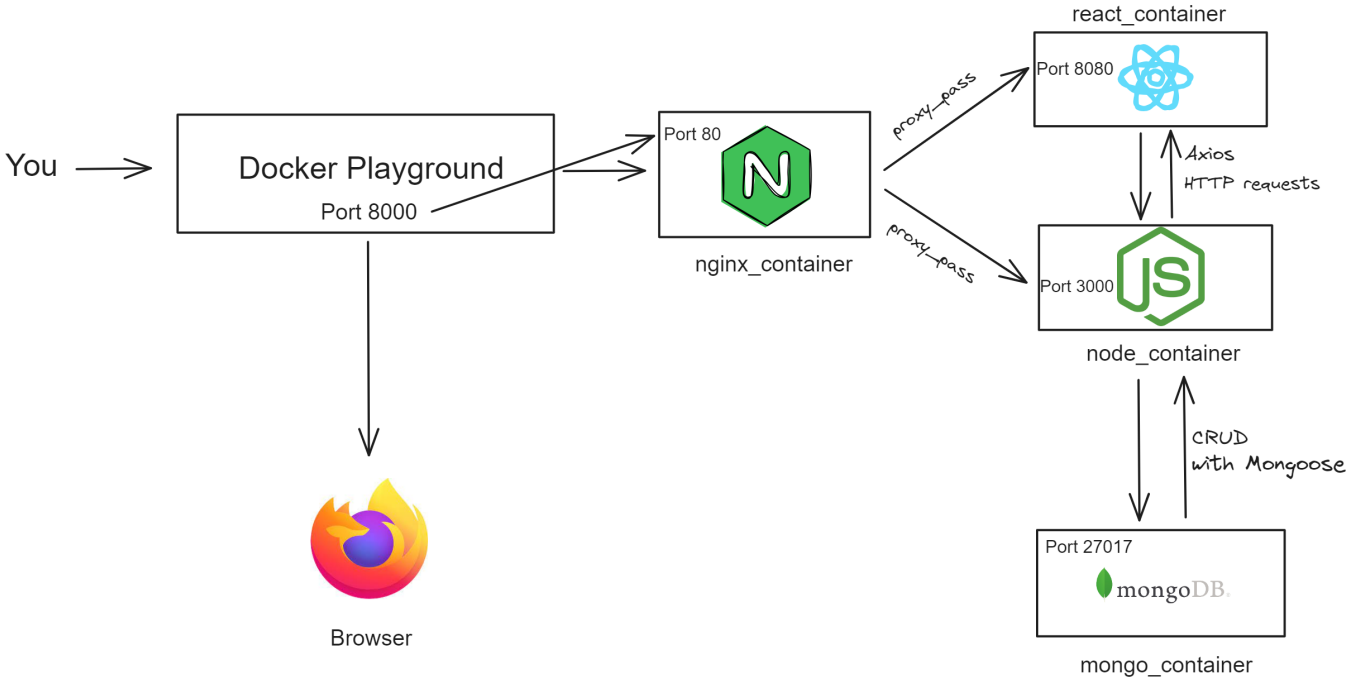
- `services`: A Compose file must declare a services top-level element as a map whose keys are string representations of service names, and whose values are service definitions. A service definition contains the configuration that is applied to each service container. In our case, we define service names `mongodb`,`backend` and `frontend` and provide configurations and build contexts for some of them too.

- `mongodb`: We define the database service name as "mongodb" and give it a container name, define the base image it should use and ask Docker Compose to always restart this service when we run `docker compose restart`. We also ask Docker to create a volume for us and map a location with the db data directory inside of the container. Volumes are used for persistent storage of data so that even if a container has to be stopped or deleted, data is persisted across such events and we can spin up new containers with this data. We also define a simple healthcheck by pinging the DB service inside the container as we want it to be up and running before our Node container can attempt to connect to it.

-`backend`: For the backend service we first pass in a name for the container and the path to the build context or the Dockerfile, which is written in a similar fashion as the basic Node + Express server we dockerised earlier.

We also need to pass in environment variables for which port the container should run on and the database URI, where we specify the service/container name and the port of the container that listens to DB connections (27107). The `depends_on` option specifies the dependencies between services and defines the order in which these containers should start and stop. We want the DB container and the service inside of it to be up before starting the backend service and attempting to establish a connection.

- `frontend`: Similar to the backend service. We pass in environment variable `VITE_PORT` to define which port we wnat our React app to run on inside the container.
- `nginx` : We have set up a container called `nginx_container` to use `NGINX` as a reverse proxy. The nginx service runs on port 80 inside the container and is mapped to the host port 8000. This service proxies requests to our React App on port 8080 of the `react_container` that runs on the browser and can send HTTP requests to our backend service from there. The `nginx` service proxies backend HTTP requests to our node_container's port 3000 and so the browser does not have to worry about resolving the IP address of that container. Docker compose assigns all these containers a common network and handles that.



Now let's run this application. Run `docker compose build` inside the `compose_example` directory. This will look for services defined in the `compose.yaml` file and run a build for each context and Dockerfile referenced there.

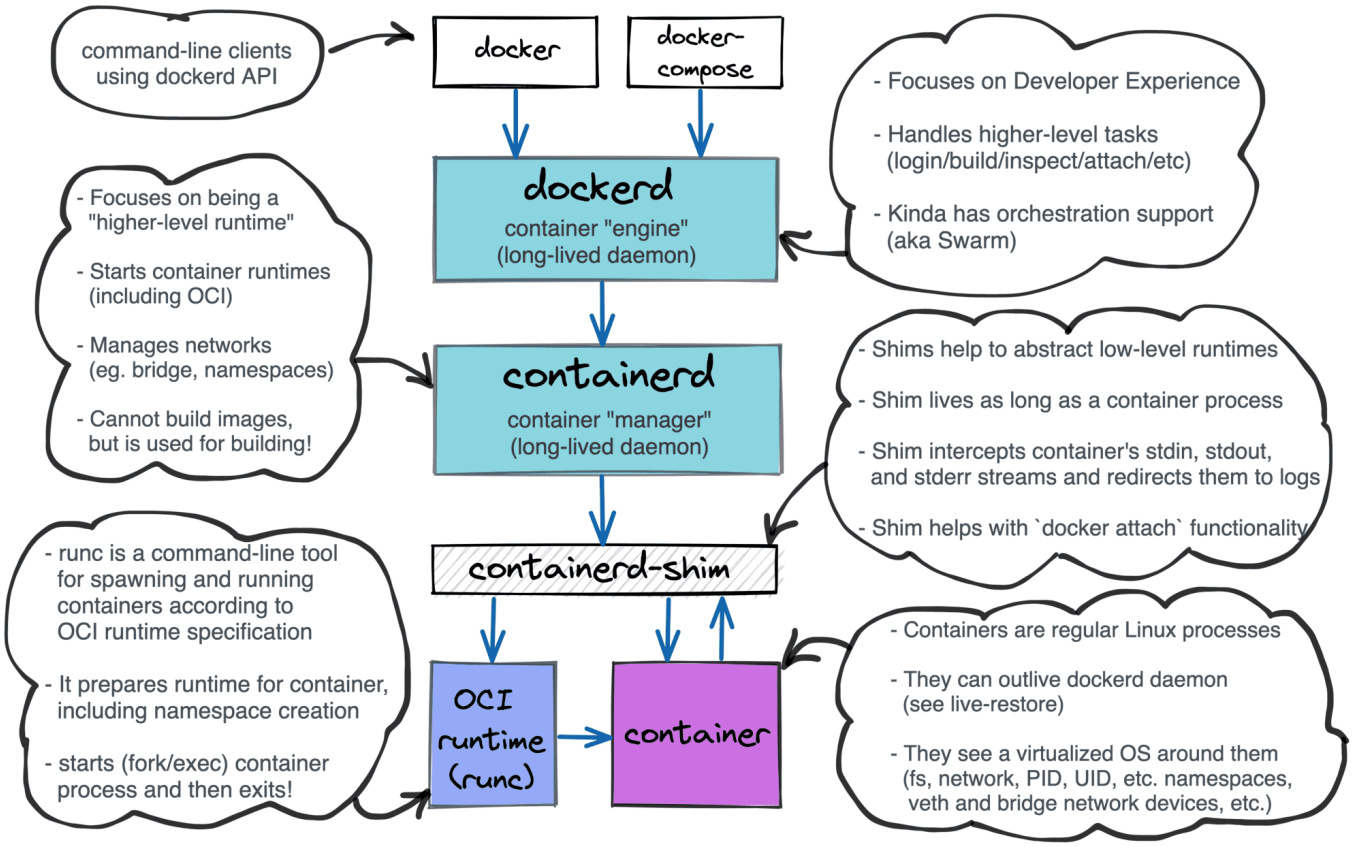
Verify that the image was built successfully with the `docker images` command. Then run `docker compose up` in the `compose_example directory`. This will create the required containers and start them in attached mode.

You can now view the site by visiting `localhost:8000` on your browser. If you are on the Docker Playground, expose port 8000 with the `OPEN PORT` option and follow the link generated to view the site.

Docker Internals

Note: This section is being covered in detail with better illustrations in the workshop.

The Docker platform provides us with a Docker engine or environment consisting of several tools like the Docker CLI, Docker API and Docker Compose and it also consists of a background process called the Docker daemon.



The Docker daemon is responsible for pulling and building images as well as creating and managing containers, volumes and networks. A `daemon` is a process running in the background that manages specific utilities and services. It listens to Docker API requests and performs the necessary actions. For example, when we run the `docker pull`command, the docker daemon is responsible for pulling and building the image if it is not available locally, either from a public repository like Docker Hub or a private repository.

`Containerd` is described as a "high-level container runtime". However, this does not tell us the full story. The `Docker daemon` actually relies on `containerd` and interacts with it through a gRPC-based API to pull and manage images, manage networks and storage, container lifecycle management, security and a container runtime interface allowing it to support different low level OCI compliant container runtimes like runC.

The `container-shim` assists in facilitating communication between the `Docker daemon` or rather `Containerd` and the container runtime (e.g., runc), ensuring smooth operation and security.

Open Container (OCI) specifications are a set of specs (runtime,build and distribution) that basically define a structure for images and container runtimes and how to interact with them. At a high-level, images are unpacked into OCI Runtime Filesystem bundles that are then executed by an OCI compliant runtime like runC.

So the actual spinning up and running of Docker containers is done by runC, as self-sufficient CLI tool that is basicaly a low level implementation of the OCI runtime specifications. As an OCI compliant low level runtime, it is responsible for executing OCI runtime bundles to create containers and assigns `namespaces` and `cgroups`.

Namespaces? Cgroups? What are these and how are they related to containers?

Well, as you read earlier in brief, containers are essentially just isolated processes. Containers are isolated with the help of Linux `namespaces` and `cgroups`.

Linux `namespaces` control what a container or rather a process can "see". In other words, what directories and and resources the process can see, basically a `chroot jail` as people like to call it. The types of namespaces in Linux include:

- PID: The `PID` namespace allows the child processes under the isolated process to have their own set of Process IDs.
- MNT: The `mnt` or "mount" namespace is used to isolate mount points such that processes in different namespaces cannot view each others' files. If you are familiar with the chroot command, it functions similarly.
- UTS: Sounds fancy, but the `UTS` namespace basically allows you to have a hostname inside the process that is different from that on the host machine.
- NET: The `NET` namespace allows you to isolate the network environment(ports, interfaces)
- USER: Allows isolation of UIDs and GIDs that allows processes to run as root.
- TIME
- IPC

What about `cgroups`? `Cgroups` or control groups allow you to isolate and limit resources allocated to processes like CPU, storage etc. The way this works is we assign groups of processes to a `controller` or a `subsystem` which manage specific type of resources or components like CPU, memory and block or disk I/O. These groups are structured in a hierarchy where each child `cgroup` inherits the properties of a parent, similar to the Linux Process and once a `cgroup` is assigned to a controller we can set resource limits for it. For a more detailed

Next Steps and Resources

- If you haven't already, go ahead and set up Docker on your host machine to enhance your developer experience. You can follow the instructions provided in the beginning of this article.
- Explore the internals of Docker containers further with this excellent resource: Jérôme Petazzoni at DockerCon EU 2015
- To put your understanding on Linux container internals to the test check out the `Container from scratch` code here - https://github.com/achyuthcodes30/Docker-Undocked-Code. Start by figuring out the crux of what's going on here and figure out the best way to `chroot` and to assign it to `cgroups` and contribute!!
- Explore container services on cloud platforms like AWS (ECS and EKS), Microsoft Azure and Google Cloud Platform. Try to push your images to the Docker Hub Public Registry and try to deploy your containerised applications using the cloud services mentioned above.
- Now that you have a good understanding of Docker and its internals, explore a container orchestration platform like `Kubernetes` and other containerization tools like `Containerd` and `Podman`. As we did with Docker, try to understand the internals of each of these technologies, their use cases, the trade offs between them in different scenarios and get hands-on with each of them. These technologies are crucial in the mordern world of DevOps and the Cloud.

- Keep learning, exploring and growing! 😀