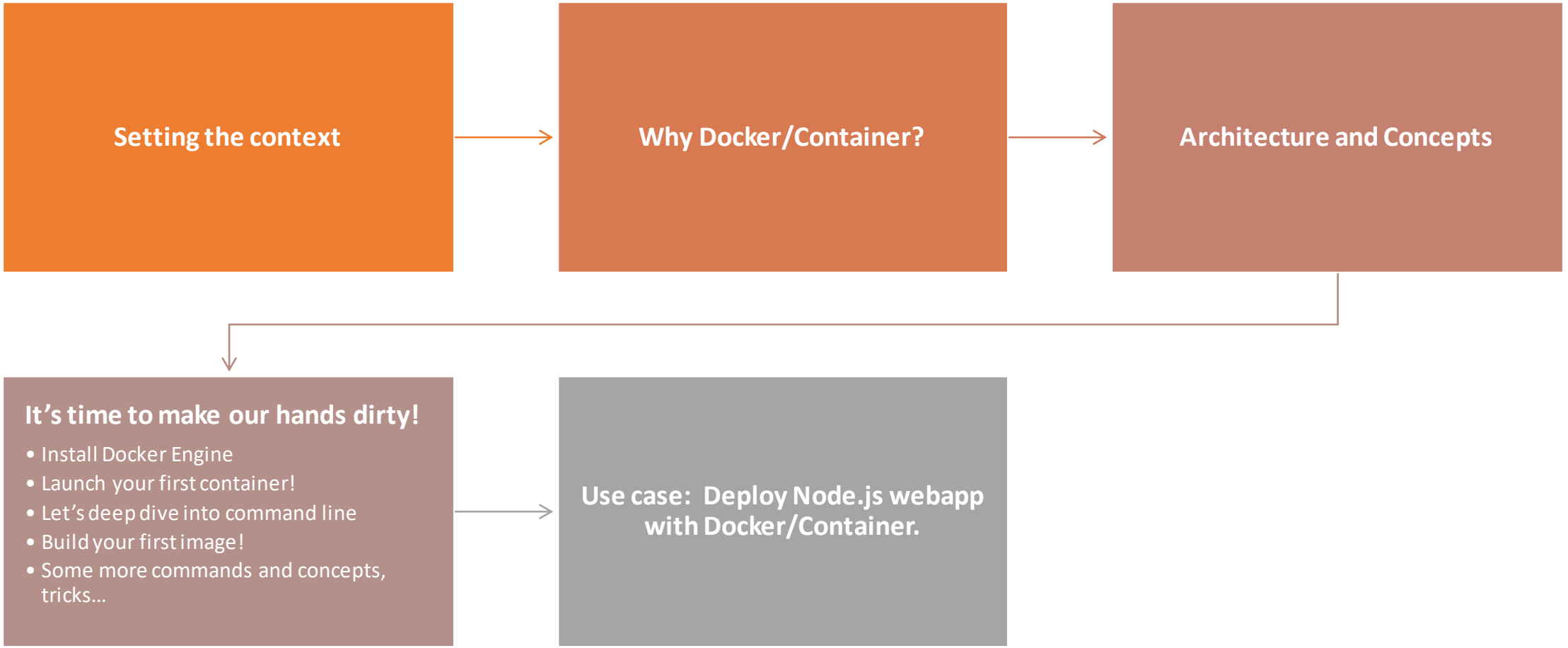
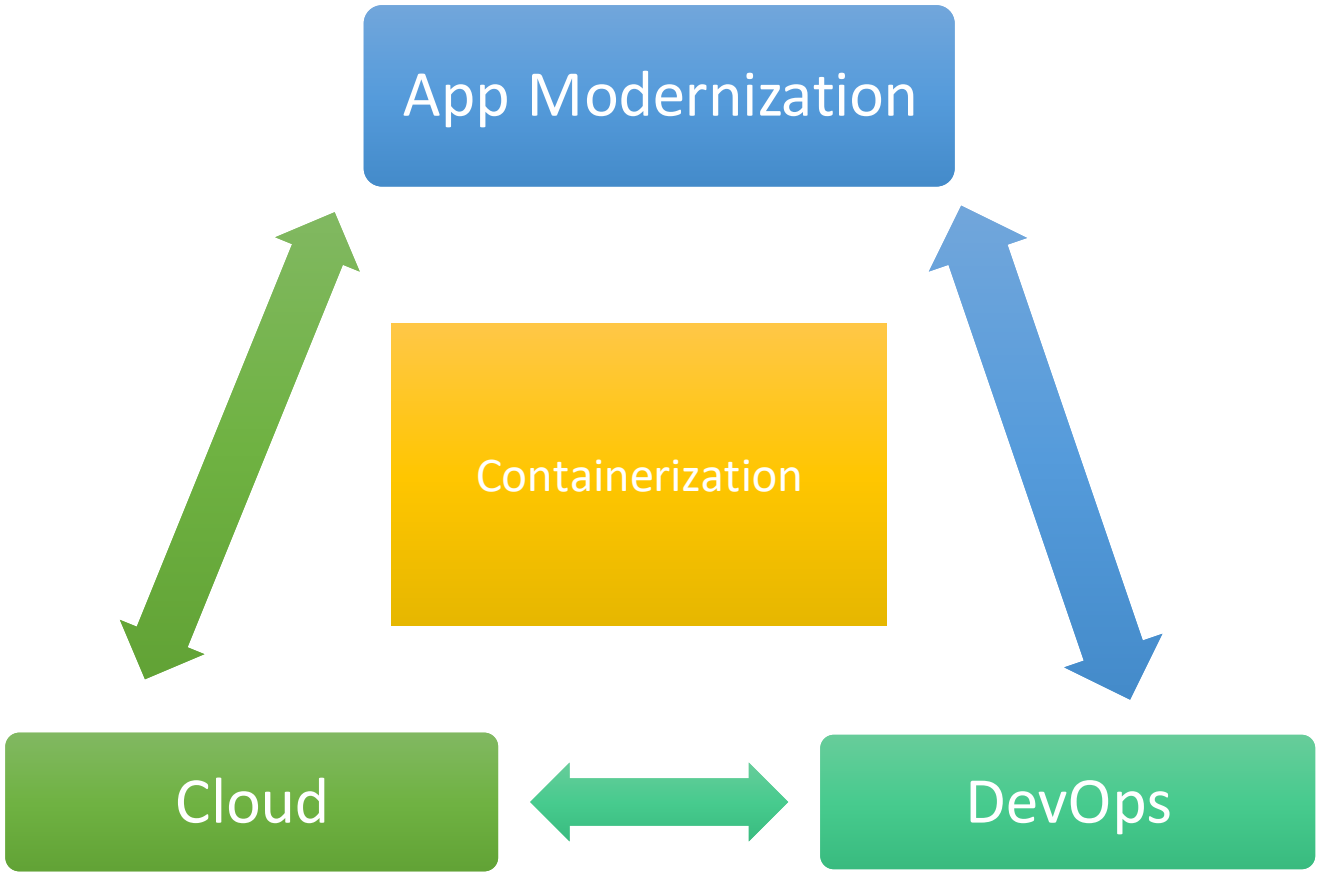
Introduction To Docker & Containers

Nilesh R. Joshi

So, what we are covering today?



Get the context before we get into it…



- Founded in 2013 as a Linux developer tool.
- Fundamentally solves the “works on my machine” syndrome.
- Rapid Prototyping, demo and deployments
- Vital factor for “Digital Transformation” by enabling shift towards Microservices.
- Enabling most desperate mode that everyone wants and that’s “FAST” mode.

What is Docker and Why Container?

- The Docker Engine is an infrastructure plumbing software that runs and orchestrates containers. Docker Engine facilitate container runtime environment that runs containers on top.
- Docker provides light-weight, portable, isolated, fully customizable application environments in the form of containers.
- At core, Docker containers are just a new modernized way to deploy applications.

Two main editions:

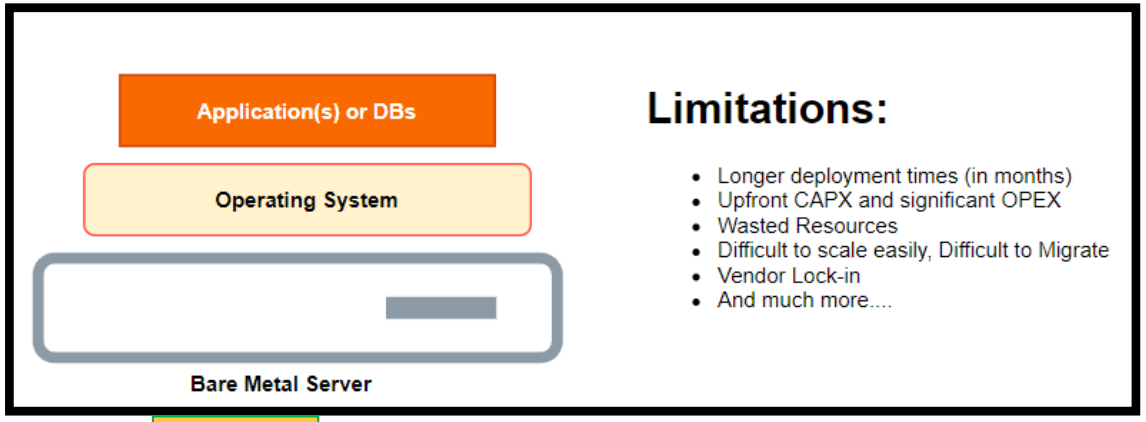
- Enterprise Edition (EE)
- Community Edition (CE)

NOTE: Each Community Edition will be supported for 4 months and each Enterprise Edition will be supported for 12 months.

The Docker Open Source Project – named "Moby". (https://mobyproject.org/)

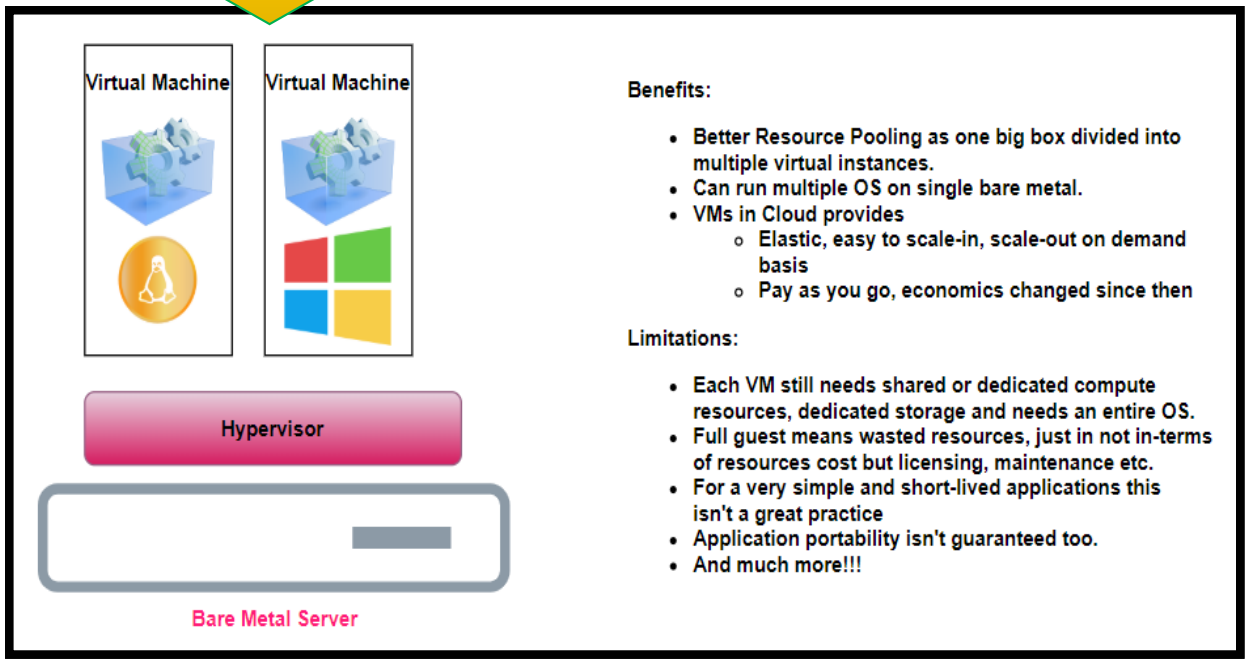
If you're a GoLang developer or aspiring developer then you've great opportunities here to contribute for future of IT.

Containerization is **not** Virtualization. Sure, they often go hand in hand !!



Limitations:

- Longer deployment times (in months)
- Upfront CAPX and significant OPEX
- Wasted Resources
- Difficult to scale easily, Difficult to Migrate
- Vendor Lock-in
- And much more....

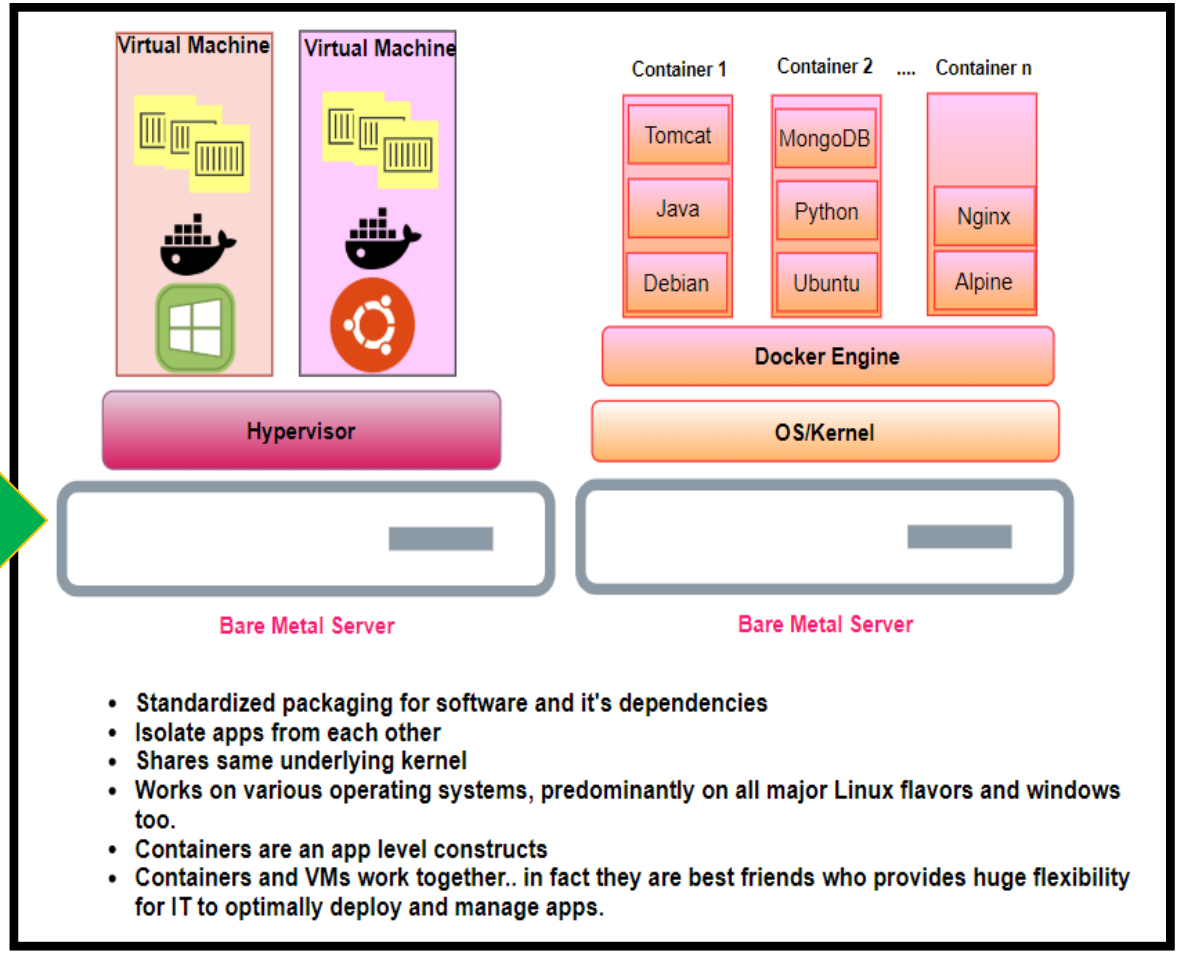


Benefits:

- Better Resource Pooling as one big box divided into multiple virtual instances.
- Can run multiple OS on single bare metal.
- VMs in Cloud provides
  - Elastic, easy to scale-in, scale-out on demand basis
  - Pay as you go, economics changed since then

Limitations:

- Each VM still needs shared or dedicated compute resources, dedicated storage and needs an entire OS.
- Full guest means wasted resources, just in not in-terms of resources cost but licensing, maintenance etc.
- For a very simple and short-lived applications this isn't a great practice
- Application portability isn't guaranteed too.
- And much more!!!



- Standardized packaging for software and it's dependencies
- Isolate apps from each other
- Shares same underlying kernel
- Works on various operating systems, predominantly on all major Linux flavors and windows too.
- Containers are an app level constructs
- Containers and VMs work together.. in fact they are best friends who provides huge flexibility for IT to optimally deploy and manage apps.

Difference between VM and Container:

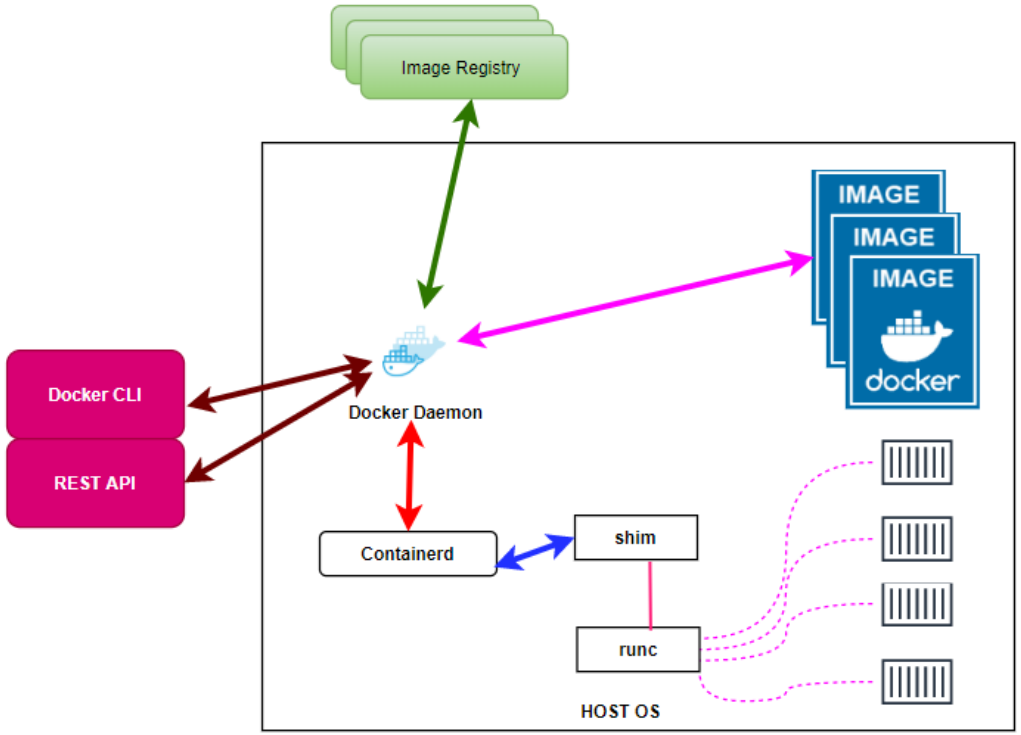
The VM is a hardware abstraction: it takes physical CPUs and RAM from a host, and divides and shares it across several smaller virtual machines. There is an OS and application running inside the VM, but the virtualization software usually has no real knowledge of that hence you can run any type of OS on top of hypervisor.

A container is an application abstraction: the focus is really on the OS and the application, and not so much the hardware abstraction. Many customers actually use both VMs and containers today in their environments and, in fact, may run containers inside of VMs.

VM is typically in GB's and Containers are lightweight, typically in MB's or in fact the smallest one I've seen is 2KB.

- Containers are with us since quite long, remember LXC, FreeBSD chroot jail, zones, WPAR etc.
- Containers are nothing but "chroot" on Steroids !! ☺
- Containers run inside the Docker Engine, which abstracts away the host OS/infrastructure, allowing our apps to “run anywhere.”
- Containers are created from a read-only template called "docker image".
- Container is a living thing! This is where our apps actually runs!
- The whole and sole purpose of Containers is to run app or a service.

Docker Architecture



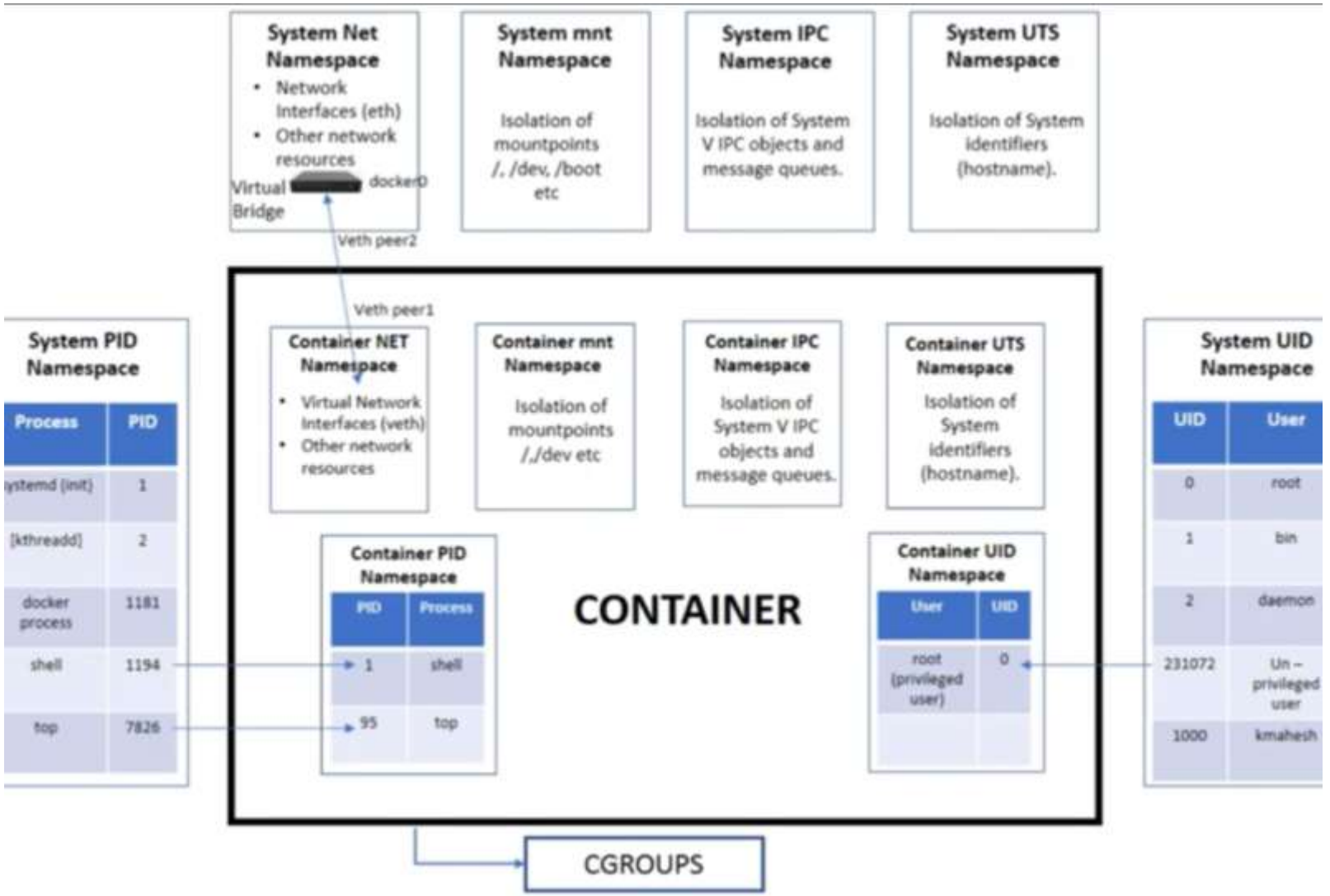
Docker architecture is modularized into:

```
dockerd (the Docker daemon)
docker-containerd (containerd)
docker-containerd-shim (shim)
docker-runc (runc)
```

dockerd is responsible for image management, image builds, the REST API, authentication, security, core networking, and orchestration.

containerd converts the required Docker image into an OCI bundle and tells runc to use this to create a new container.

runc interfaces with the OS kernel to pull together all of the constructs necessary to create a container (namespaces, cgroups etc.). The container process is started as a child-process of runc, and as soon as it is started runc will exit and pass control to shim.



Container Architecture

Let’s get our hands dirty! @GitHub

- Install Docker (Not covering Windows at this time…)
  - But on Windows you must have : Must be 64-bit, Hyper-V and Container features to be enabled, H/W virtualization support enabled.

```
#!/bin/bash
sudo dnf config-manager \
--add-repo=https://download.docker.com/linux/centos/docker-ce
sudo dnf install --nobest docker-ce
sudo systemctl enable --now docker
sudo systemctl is-active docker
# add current user to docker group so there is no need to use
sudo usermod -aG docker $(whoami)
```

Docker Command Hands-On

@GitHub

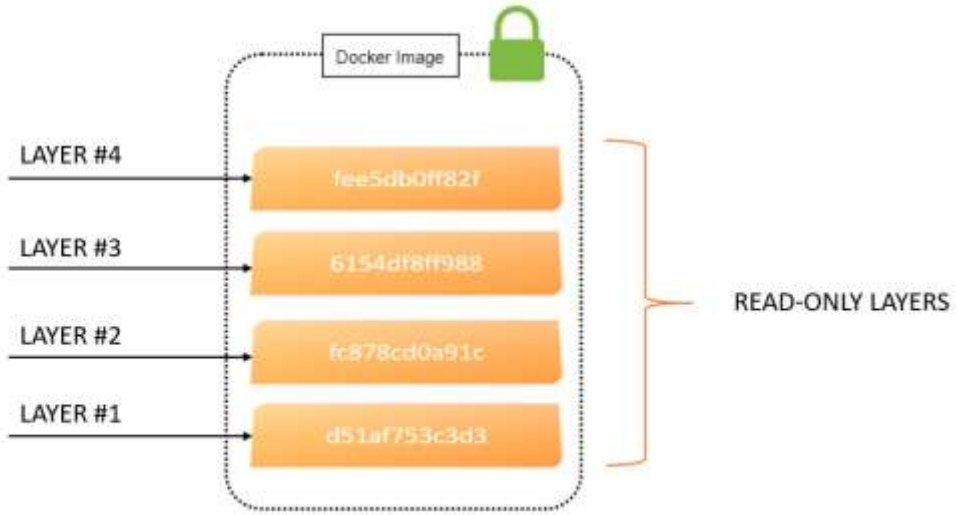
Docker Image

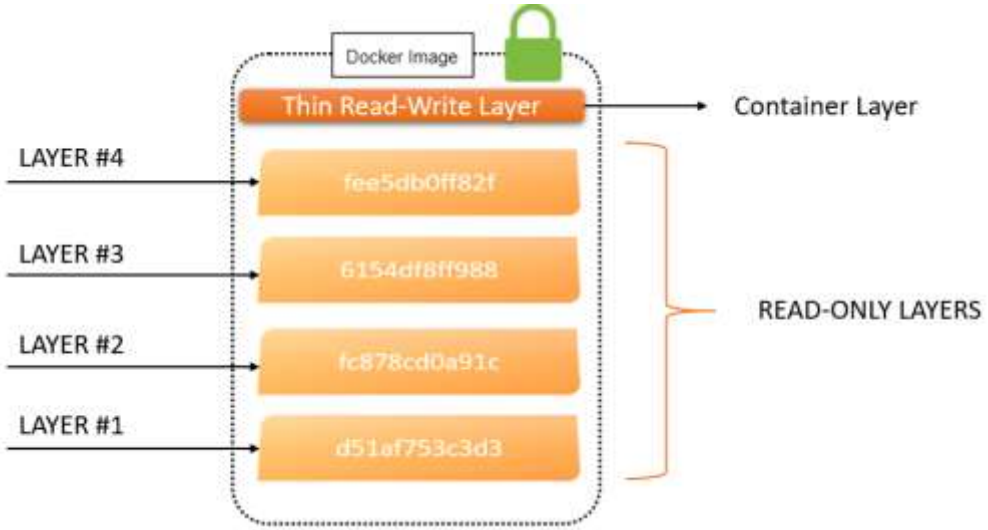
- A Docker image is made up of a collection of files that bundle together all the essentials, such as installations, application code and dependencies, required to configure a fully operational container environment.
- Image is a combination of layer and each layer is made up of each instruction written in Dockerfile.

You can create a Docker image in one of two ways:

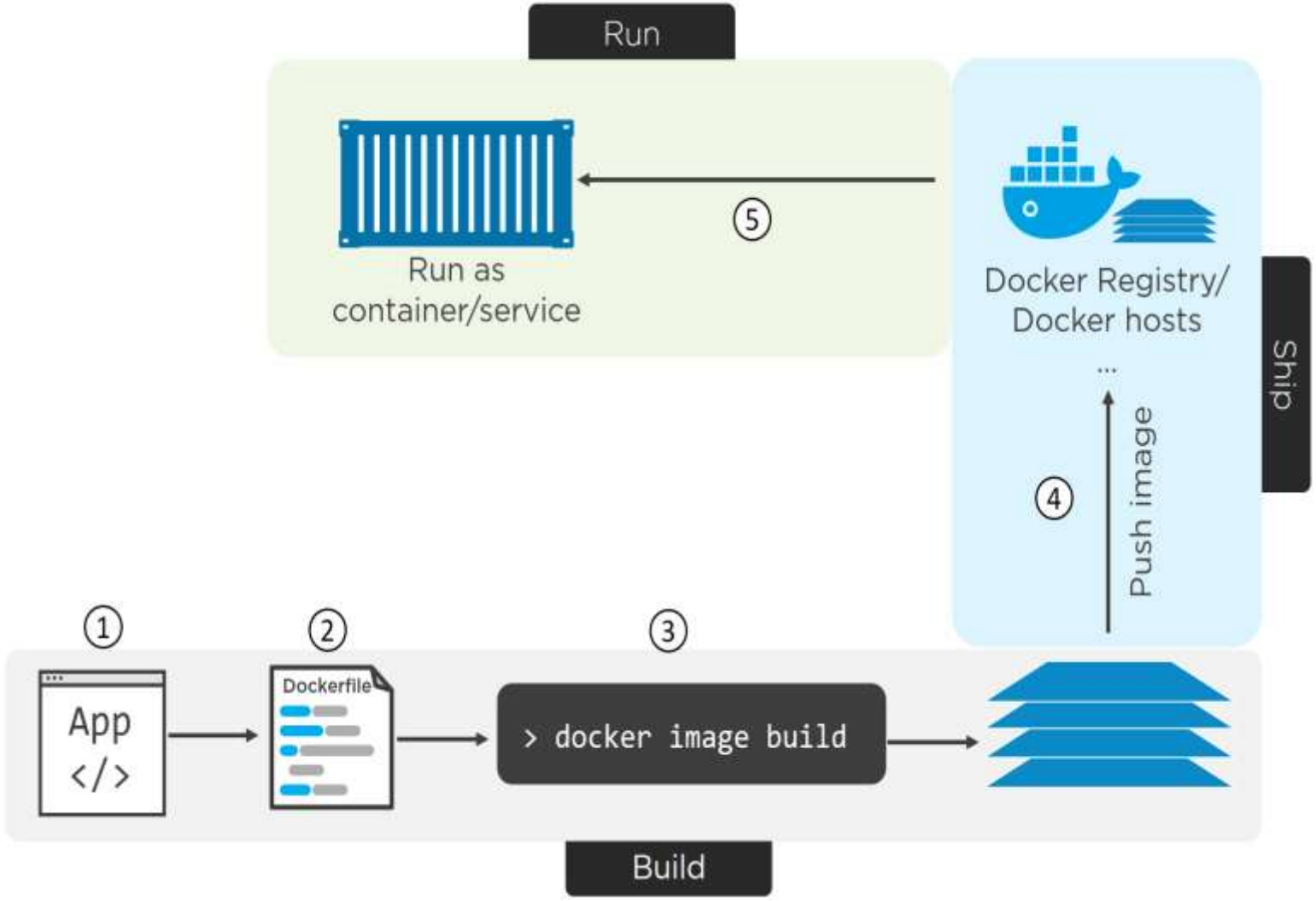
  - Interactive Method: By running a container from an existing Docker image, manually changing that container environment through a series of live steps and saving the resulting state as a new image.
  - Dockerfile Method: By constructing a plain-text file, known as a Dockerfile, which provides the specifications for creating a Docker image.

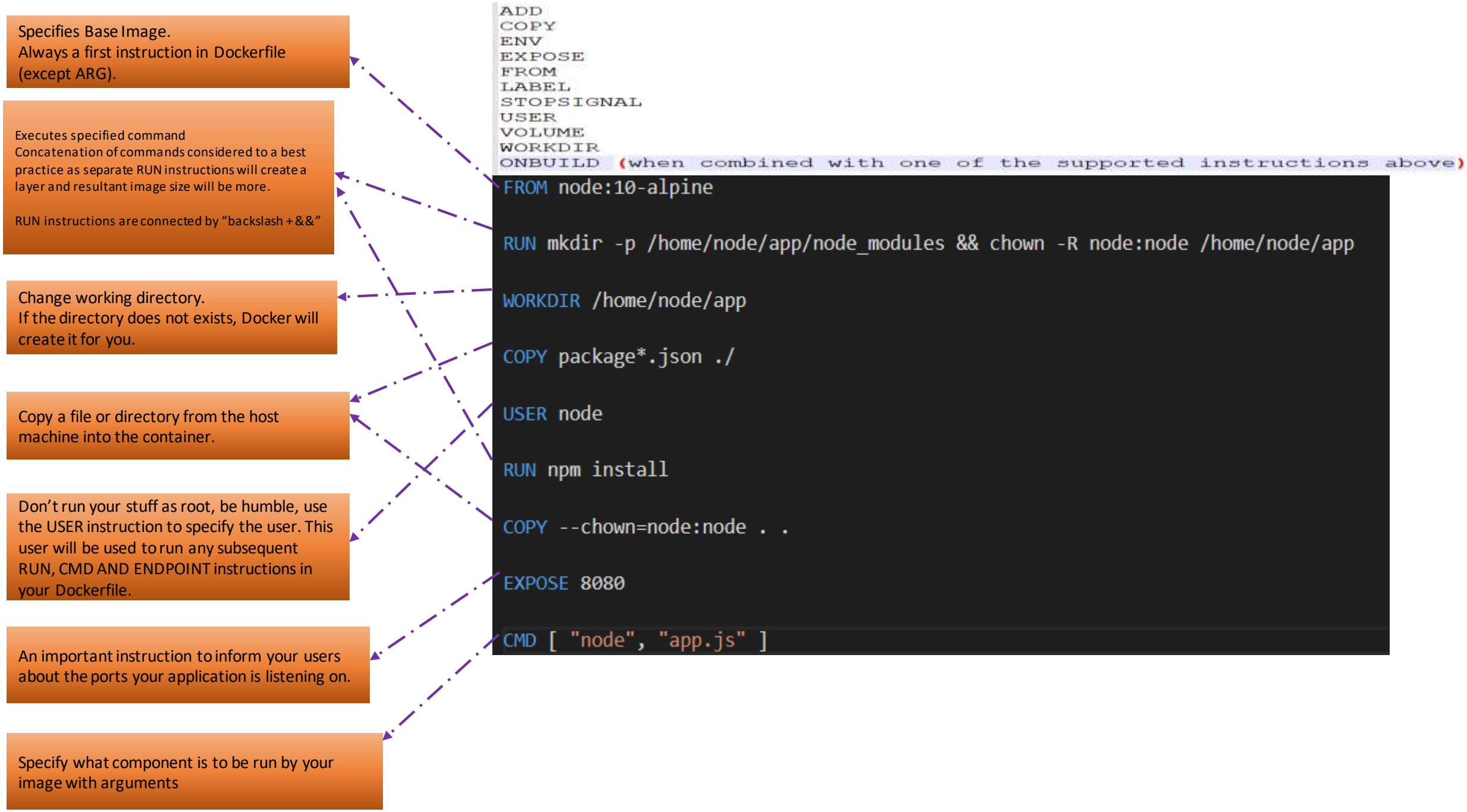
```
[root@            ~]# docker image pull ubuntu:latest
latest: Pulling from library/ubuntu
d51af753c3d3: Pull complete
fc878cd0a91c: Pull complete
6154df8ff988: Pull complete
fee5db0ff82f: Pull complete
Digest: sha256:747d2dbbaaee995098c9792d99bd333c6783ce56150d1b11e333bbceed5c54d7
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

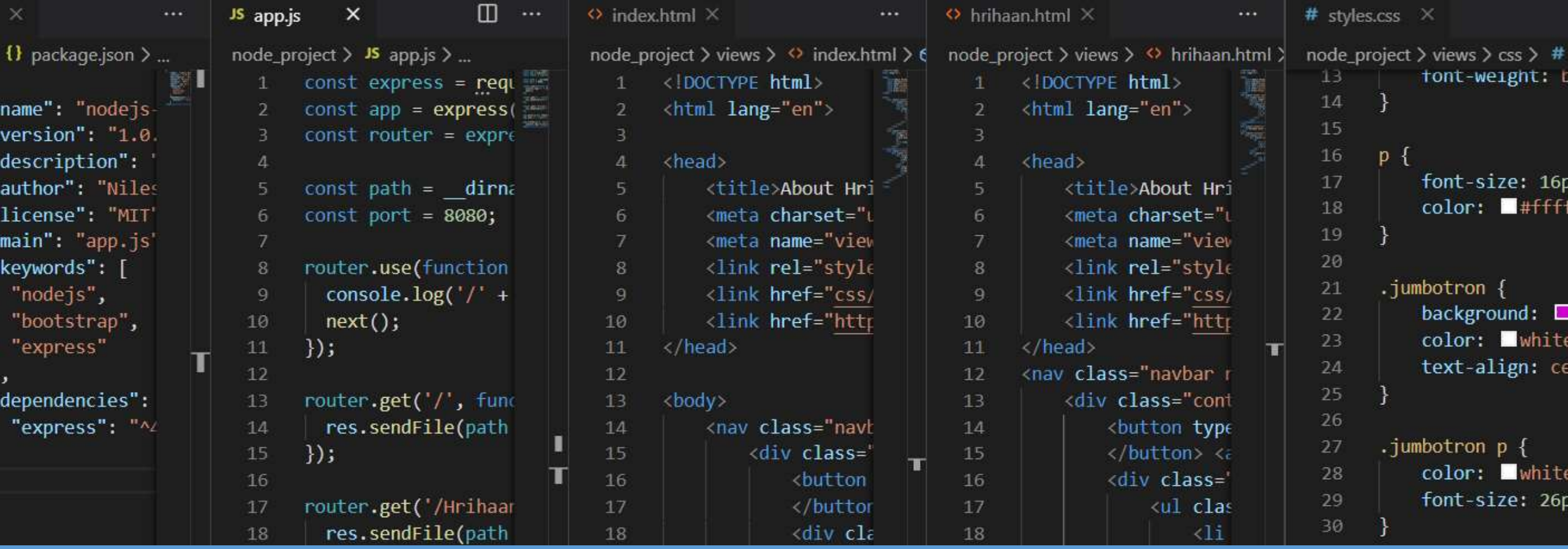




Containerized App Build Process




Specifies Base Image.
Always a first instruction in Dockerfile (except ARG).
Executes specified command
Concatenation of commands considered to a best practice as separate RUN instructions will create a layer and resultant image size will be more.
RUN instructions are connected by “backslash + &&”
Change working directory.
If the directory does not exists, Docker will create it for you.
Copy a file or directory from the host machine into the container.
Don’t run your stuff as root, be humble, use the USER instruction to specify the user. This user will be used to run any subsequent RUN, CMD AND ENDPOINT instructions in your Dockerfile.
An important instruction to inform your users about the ports your application is listening on.
Specify what component is to be run by your image with arguments
ADD
COPY
ENV
EXPOSE
FROM
LABEL
STOPSIGNAL
USER
VOLUME
WORKDIR
ONBUILD (when combined with one of the supported instructions above)
FROM node:10-alpine
RUN mkdir -p /home/node/app/node_modules && chown -R node:node /home/node/app
WORKDIR /home/node/app
COPY package*.json ./
USER node
RUN npm install
COPY --chown=node:node . .
EXPOSE 8080
CMD ["node", "app.js"]



Containerized App Demo

Code Available @GitHub

Docker Image - `docker pull nileshjoshi/nodejs-webapp:v2`

Run Container - `docker run -itd --name demo -p 8080:8080 nileshjoshi/nodejs-webapp:v2`