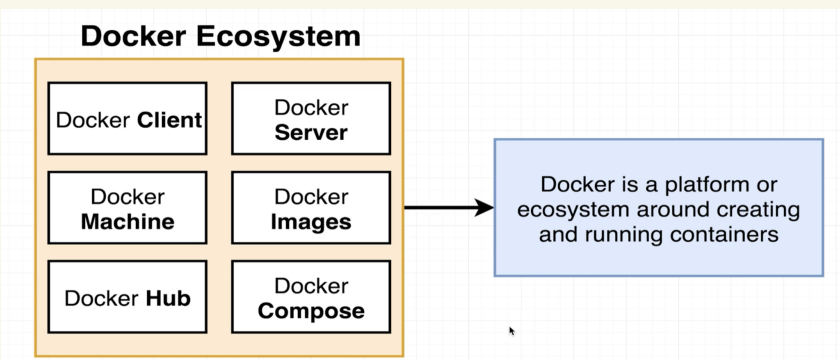
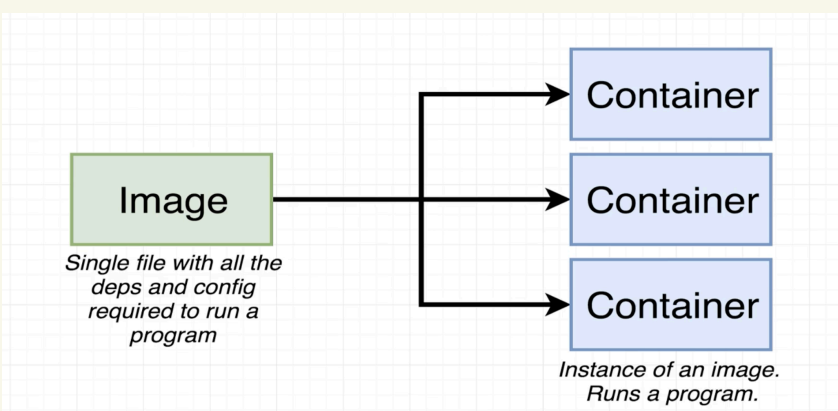

Docker

Why use Docker?

→ Docker makes it really easy to install and run software without worrying about setup and dependencies.

What is Docker?





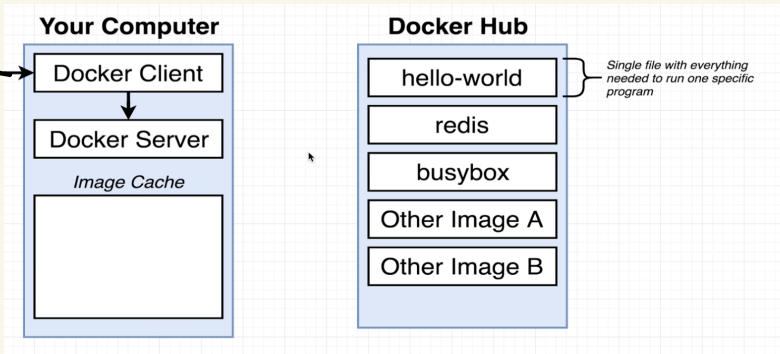
Docker Client / Docker CLI

→ Tool that we are going to issue commands to

Docker Server / Docker Daemon

→ Tool that is responsible for creating images, running containers, etc.

```
docker run hello-world
```



Step 1: Docker client takes our request.

Step 2: Docker server is responsible for running the request.

Step 3: Docker server looks for the hello-world image in Image cache (also known as local system)

Step 4: If not found, Docker server looks for the image in Docker Hub

Docker Hub ⇒ Free public repo of images

Step 5: Docker server creates a single container and runs the program.

Name Spacing

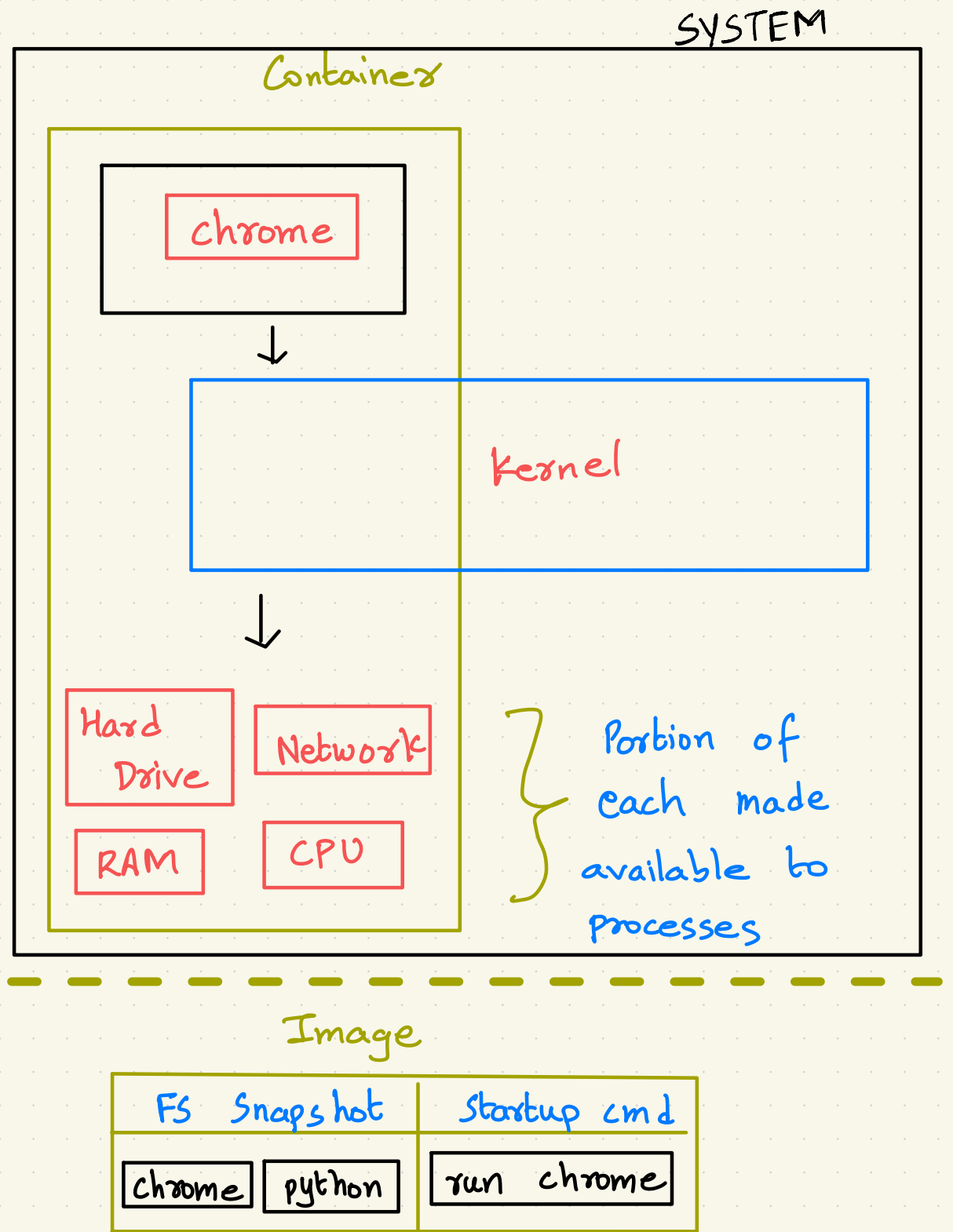
→ Isolates resources per process

- processes
- Hard drive
- Network
- Users
- Hostnames
- Inter process Communications

Control Groups / cgroups

→ Limit the amount of resources used per process

- Memory
- CPU usage
- HD I/O
- Network bandwidth


SYSTEM
Container
chrome
Kernel
Hard Drive
Network
RAM
CPU
Portion of each made available to processes
Image
FS Snapshot
Startup cmd
chrome
python
run chrome

> docker run <Image Name>

> docker run <ImageName> command

↓

To override the default start command

Note: Make sure that override command is supportable in that container

List running Containers

```
docker ps --all
```

ps ⇒ process status

--all ⇒ [optional] print all

Container life Cycle

`docker run` =

`docker create`

+

`docker start`

> docker create <Image Name>

> docker start <container ID>

> docker start -a <container ID>

-a ⇒ watches the output of the container.

Removing Containers

Containers are saved until they are removed manually.

> docker system prune

→ Removes all stopped containers and all unused cache images.

> docker rm <container ID>

→ Remove single container

Retrieve the logs of Container

> docker logs <containerID>

Stopping a Container

> docker stop <container ID>

→ Does the cleanup and stops the container.

→ Only have 10 secs to clean up, after that it terminates.

> Docker kill <container ID>

→ Terminates the container immediately.

Executing Commands inside a Container

> docker exec -it <containerID> <command>

exec ⇒ run another command

-it ⇒ allows us to provide input to the container

⇒ [-i -t]

⇒ STDIN STDOUT

Getting a terminal to Container

```
> docker exec -it <container ID> sh
```

sh
shell

Note: docker run <Image> sh

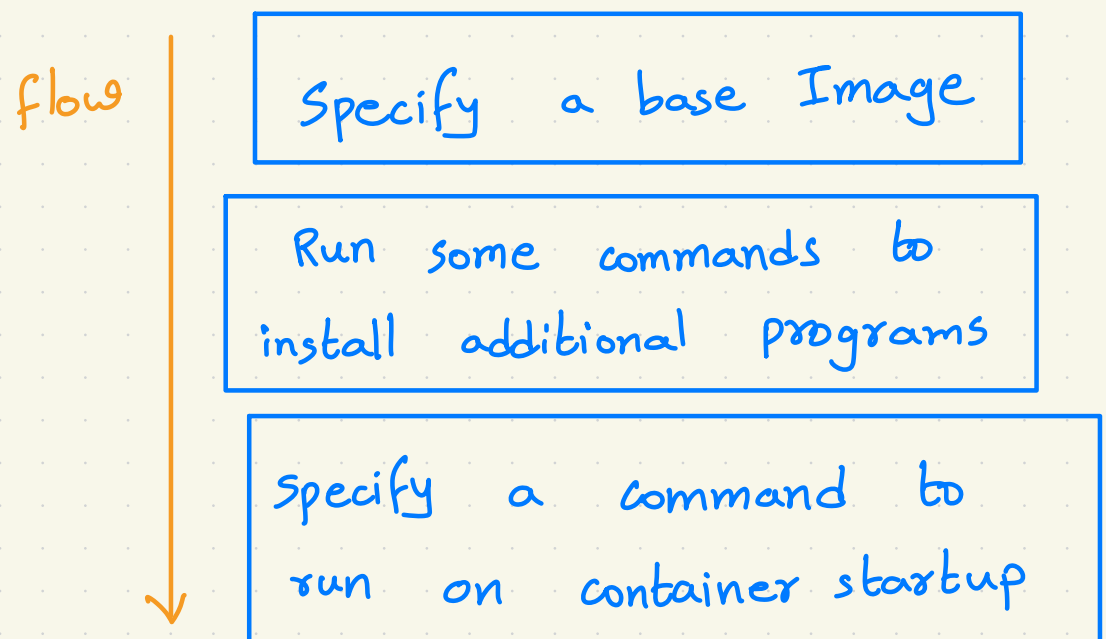
opens a shell directly

Building Custom Images

Dockerfile

→ Configuration to define how our container should behave.

Creating a Dockerfile



New Feature

Buildkit

> docker build --progress=plain .

progress ⇒ To print the output

>docker build --no-cache .

--no-cache ⇒ disables caching

> DOCKER_BUILDKIT=0 docker build .

→ Disable docker buildkit.

Building a DockerFile

Step 1: Create a "Dockerfile"

Note: No extension and D is in Uppercase

Base Template for Dockerfile

```
# Use an existing docker image as a base

# Download and install dependencies

# Tell the image what to do when it starts as a container
```

Example Dockerfile :-

→ Creating Redis server

→ Dockerfile

```
1. FROM alpine
2. RUN apk add --update redis
3. CMD ["redis-server"]
```

Instructions telling Docker server what to do

Argument to the instructions

Base Image

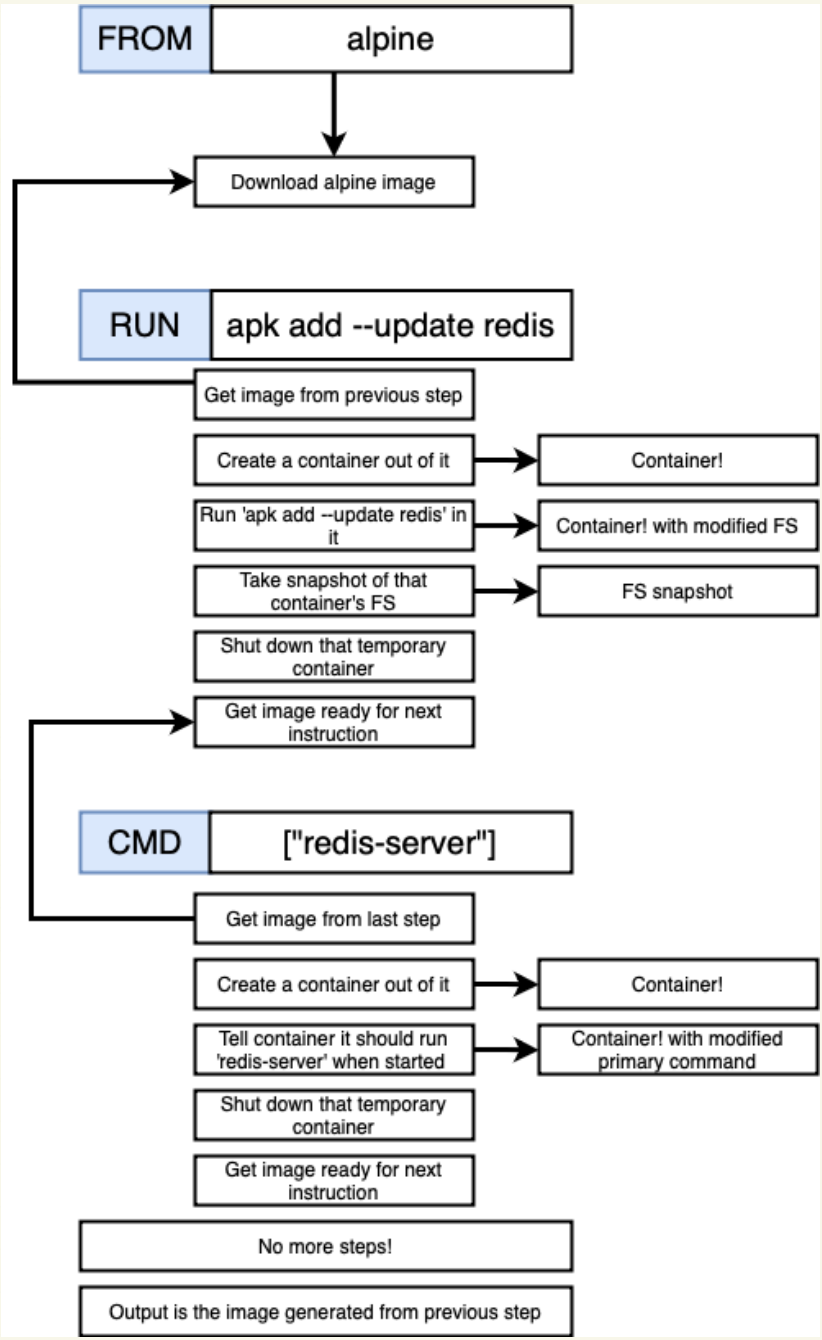
→ Base image is an empty first layer, which allows to build your docker images from scratch.

Ex:- alpine

Note: alpine is one of the base image. You can use other base images aswell.

→ Base images can also contain some pre-installed packages and programs.

Build Process in detail



Naming a docker Image

>docker build -t <name> .

Format for naming.

<dockerID> / redis : latest

dockerID → your docker ID

redis → Repo / Project name

latest → version

Note: -t

⇒ The use of -t is only for specifying version

> docker tag <image ID> <name>

→ To name an existing image

Manual Image Generation

→ Useful when you want to take a snapshot of currently running container and save it as an image.

> docker commit -c

continued

'CMD ["redis-server"]' <container ID>

Assigning default run command to the new docker Image

Fact : You don't have to give complete container ID, Docker will automatically detect the container ID if the provided ID is uniquely distinguishable

Port forwarding

```
>docker run -p 8080:8081 <image ID>
```

-P ⇒ port mapping

8080 → 8081

→ Any incoming traffic on localhost 8080 is forwarded to docker's localhost 8081.

Dockerfile Supports

Instruction	Description
`ADD`	Add local or remote files and directories.
`ARG`	Use build-time variables.
`CMD`	Specify default commands.
`COPY`	Copy files and directories.
`ENTRYPOINT`	Specify default executable.
`ENV`	Set environment variables.
`EXPOSE`	Describe which ports your application is listening on.
`FROM`	Create a new build stage from a base image.
`HEALTHCHECK`	Check a container's health on startup.
`LABEL`	Add metadata to an image.
`MAINTAINER`	Specify the author of an image.
`ONBUILD`	Specify instructions for when the image is used in a build.
`RUN`	Execute build commands.
`SHELL`	Set the default shell of an image.
`STOPSIGNAL`	Specify the system call signal for exiting a container.
`USER`	Set user and group ID.
`VOLUME`	Create volume mounts.
`WORKDIR`	Change working directory.

COPY ./ ./

local system

Container folder

WORKDIR ./usr/app

→ makes ./usr/app as working directory in the container.

Docker Compose

→ Docker compose is a tool for defining and running multi-container applications.

→ Used to start up multiple docker containers at the same time.

→ Automates some of the long-winded arguments passing to "docker run"

Similar to Dockerfile,

docker-compose.yml

Note: Contains all the options we'd normally pass to docker-cli

Example

docker-compose.yml

```
1. version : '3'
2. services :
3.       redis-server:
4.             image : 'redis'
5.       node-app :
6.             build : .
7.             ports :
                 - "4001 : 8081"
```

Version ⇒ version of docker compose

services ⇒ Defining the containers

⇒ We don't need to explicitly mention port connections between containers

"-" ⇒ Refers to an array in yml

```
docker run myimage
```

⇓ is equals to

```
docker-compose up
```

```
docker build .
docker run myimage
```

⇓ is equals to

```
docker-compose up --build
```

docker run -d redis

-d ⇒ [--detach]

⇒ Won't print the output in terminal

⇒ Useful when you want to start multiple containers at -a- time.

> docker-compose up -d

⇒ starts the containers

> docker_compose down

⇒ stops the containers

Automatic Restart of Containers

```
services:
  my-app:
    image: my-app-image
    restart: always
```

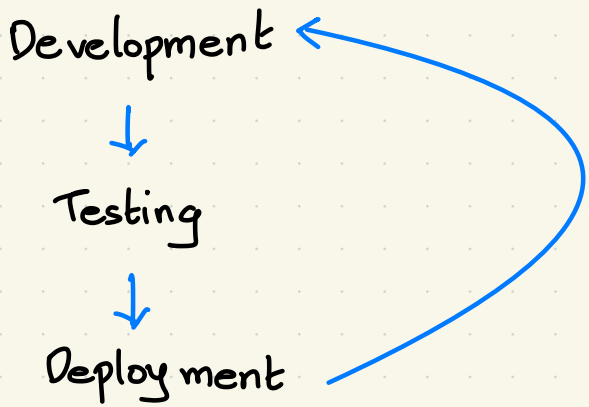
Restart policies

"no"	Does not restart
always	always restart
on-failure	only restart if container stops with error code
unless-stopped	restart unless dev's forcibly stopped it.

```
> docker-compose ps
```

Note: Make sure that you are running the command where the yml file exists

Work flow



In Development

Docker Container

npm run start

In Production

Docker Container

npm run build

Dockerfile.dev ⇒ for development

1. FROM node:alpine
2. WORKDIR '/app'
3. COPY package.json .
4. RUN npm install
5. COPY . .
6. CMD ["npm", "run", "start"]

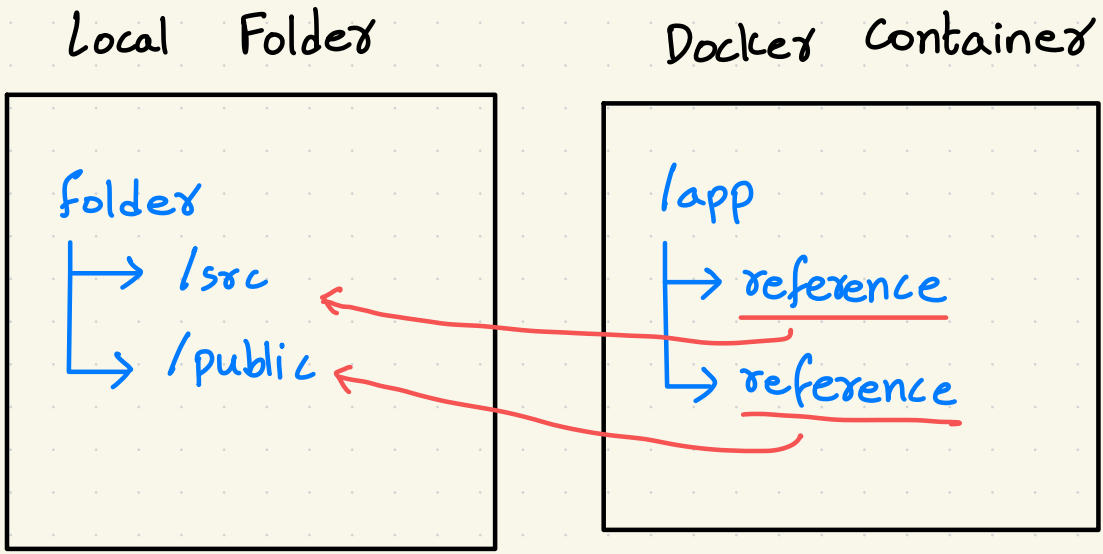
To run custom dockerfiles with extensions

> docker build -f Dockerfile.dev .

-f ⇒ refers to filename

Docker Volumes

⇒ Instead of copying files, we will reference to the file location



Command for referencing

```
> docker run -p 3000:3000
         -v /app/node_modules
         -v $(pwd):/app
         <image ID>
```

`-v` ⇛ [--volume]

`$(pwd):/app`

→ map everything inside (pwd) to /app folder in container.

`-v /app/node_modules`

⇛ This is to create a volume in container, which cannot be overwritten by referencing (by $(pwd):/app)

Since the above command is too long, its better to use Docker Compose

```
1.  version: '3'
2.  services:
3.    web:
4.      build:
5.        context: .
6.        dockerfile: Dockerfile.dev
7.      ports:
8.        - "3000:3000"
9.      volumes:
10.       - /app/node_modules
11.       - .:/app
```

Nginx

→ Popular web server

→ All it does is take incoming traffic and respond to that traffic.

→ It is used as

1) Reverse proxy
2) Load balancer
3) Caching
4) Web serving
5) and lot more

Process

Build Phase

Use node:alpine

↓

Copy the package.json file

↓

Install dependencies

↓

Run 'npm run build'

Run phase

Use nginx

↓

Copy over the result of 'npm build'

↓

Start nginx

Example:

Dockerfile

```
1. FROM node:16-alpine AS builder
2. WORKDIR '/app'
3. COPY package.json .
4. RUN npm install
5. COPY . .
6. RUN npm run build
7. # build stuff ⟶ /app/build
8. FROM nginx
9. COPY --from=builder /app/build
   /usr/share/nginx/html
```

--from ⇒ Copy data from builder container

/app/build ⇒ Files to copy

/usr/..../html ⇒ Paste at (provided by nginx)