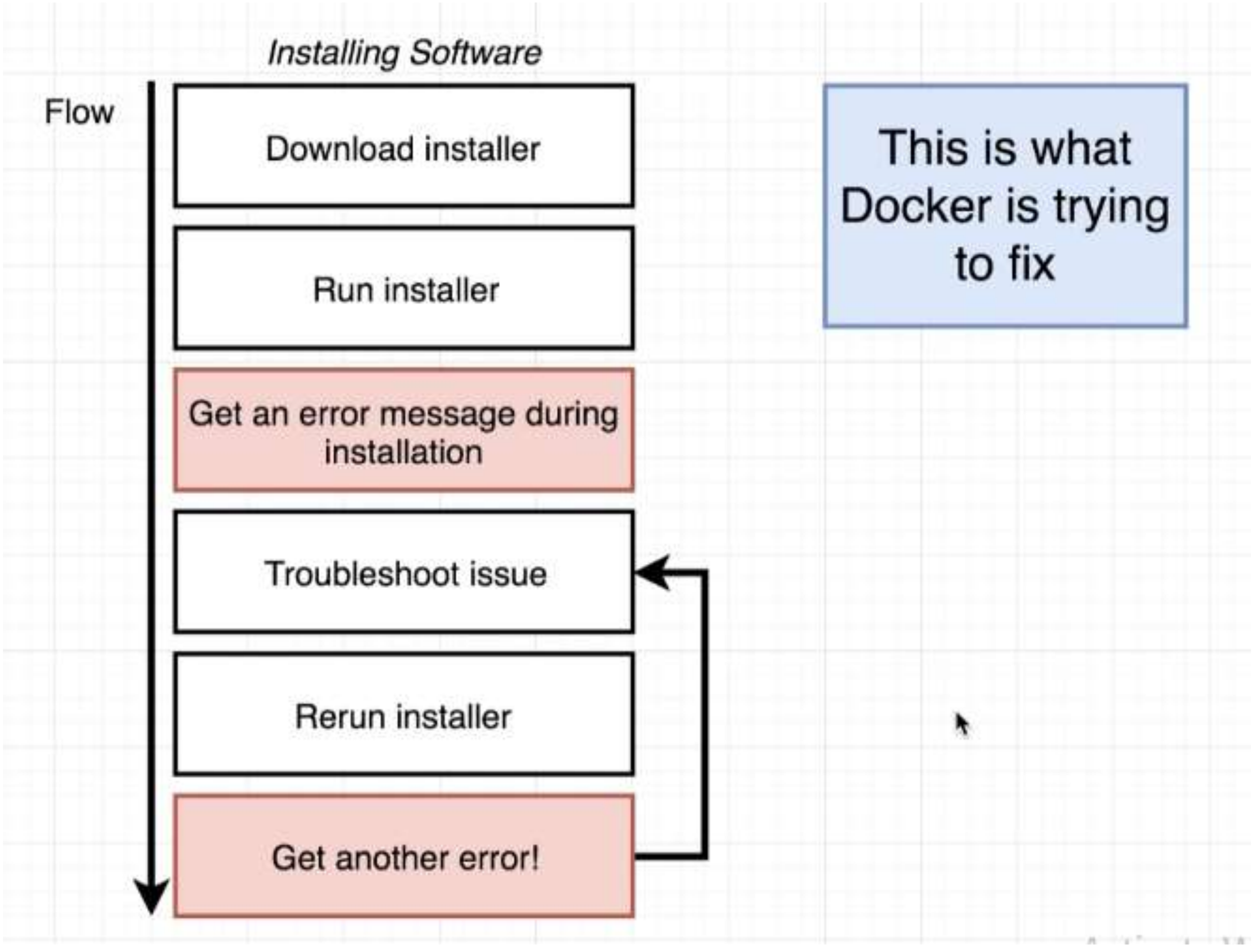
DEVOPS PART TWO: DOCKER AND K8S

CHAPTER ONE: Docker

What is docker

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.



The Docker platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

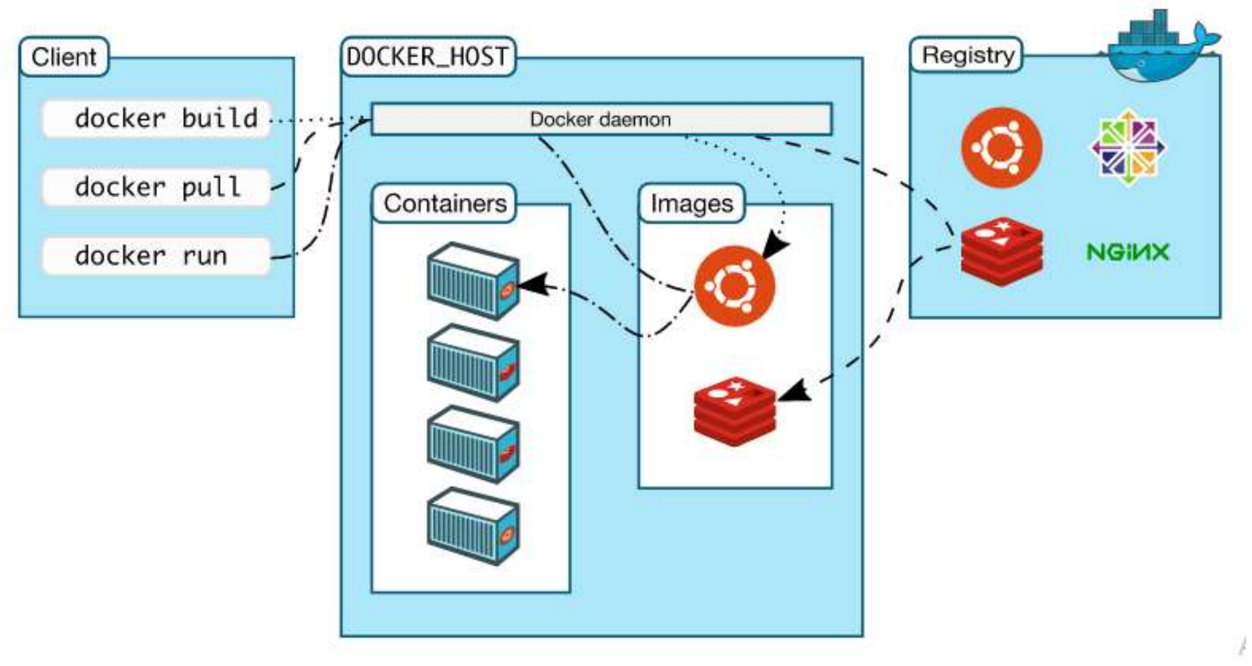
- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

- It used to be that when you wanted to run a web application, you bought a server, installed Linux, set up a LAMP stack, and ran the app. If your app got popular, you practiced good load balancing by setting up a second server to ensure the application wouldn't crash from too much traffic.
- Times have changed, though, and instead of focusing on single servers, the Internet is built upon arrays of inter-dependent and redundant servers in a system commonly called "the cloud". Thanks to innovations like Linux kernel namespaces and cgroups, the concept of a server could be removed from the constraints of hardware and instead became, essentially, a piece of software. These software-based servers are called containers, and they're a hybrid mix of the Linux OS they're running on plus a hyper-localized runtime environment (the contents of the container).

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.



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 Desktop

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. For more information, see Docker Desktop.

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 objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

Understanding containers

A container is a runnable instance of docker image.

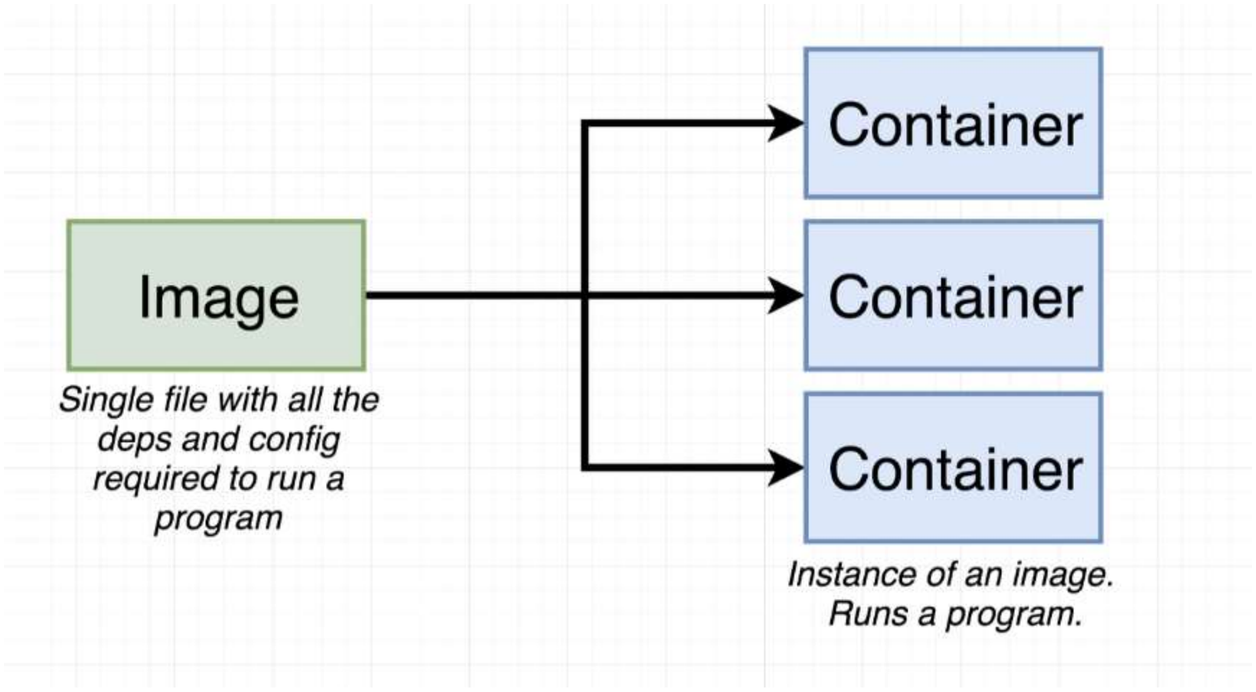


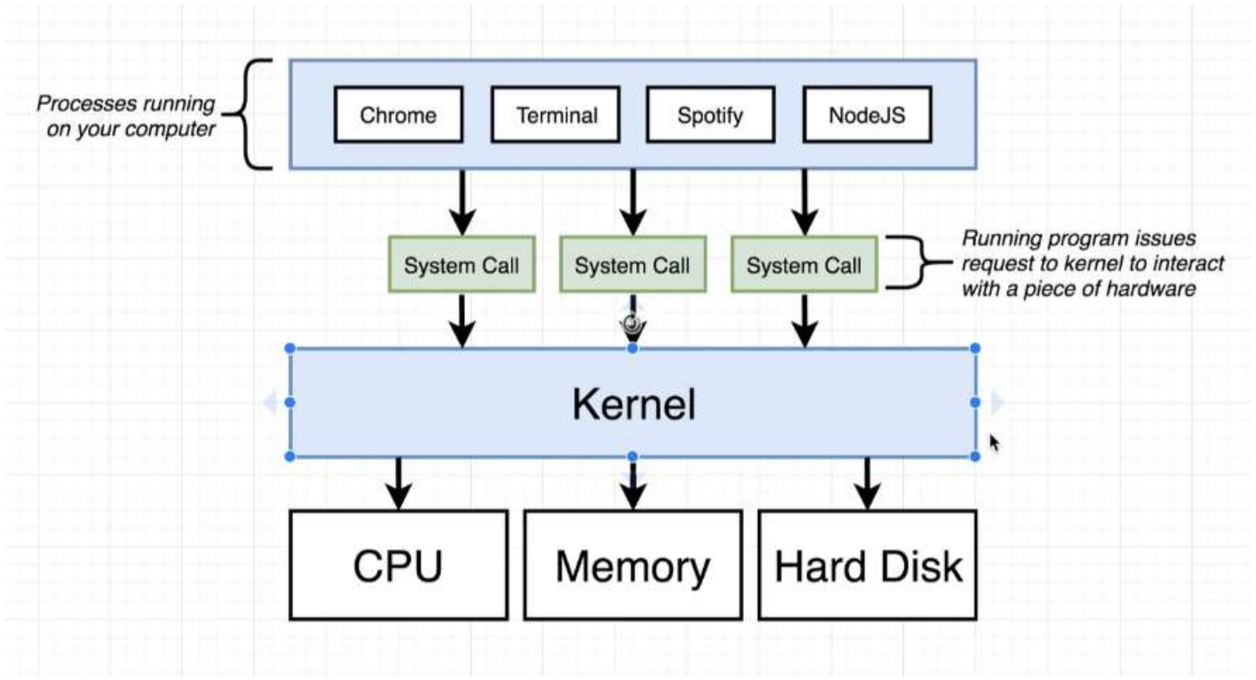
Image
Single file with all the deps and config required to run a program
Container
Container
Container
Instance of an image. Runs a program.

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.

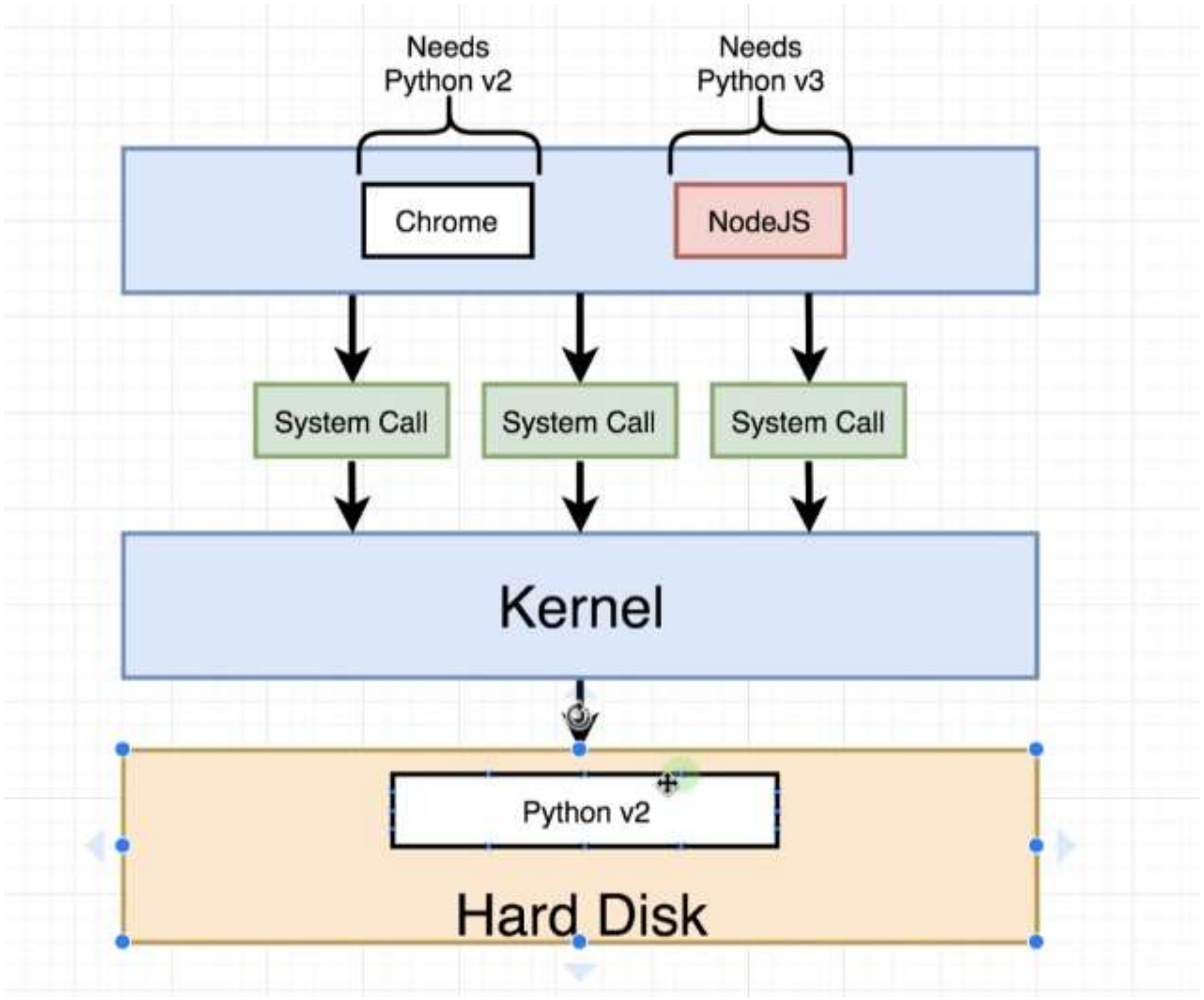
By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

Deep analysis of a container

In normal circumstance for applications we use in day to day are installed and work as diagram below shows. When an application say chrome is installed, it makes a call to kernel to access computer resources say hardware, memory, CPU etc


Processes running on your computer
Chrome
Terminal
Spotify
NodeJS
System Call
System Call
System Call
Running program issues request to kernel to interact with a piece of hardware
Kernel
CPU
Memory
Hard Disk

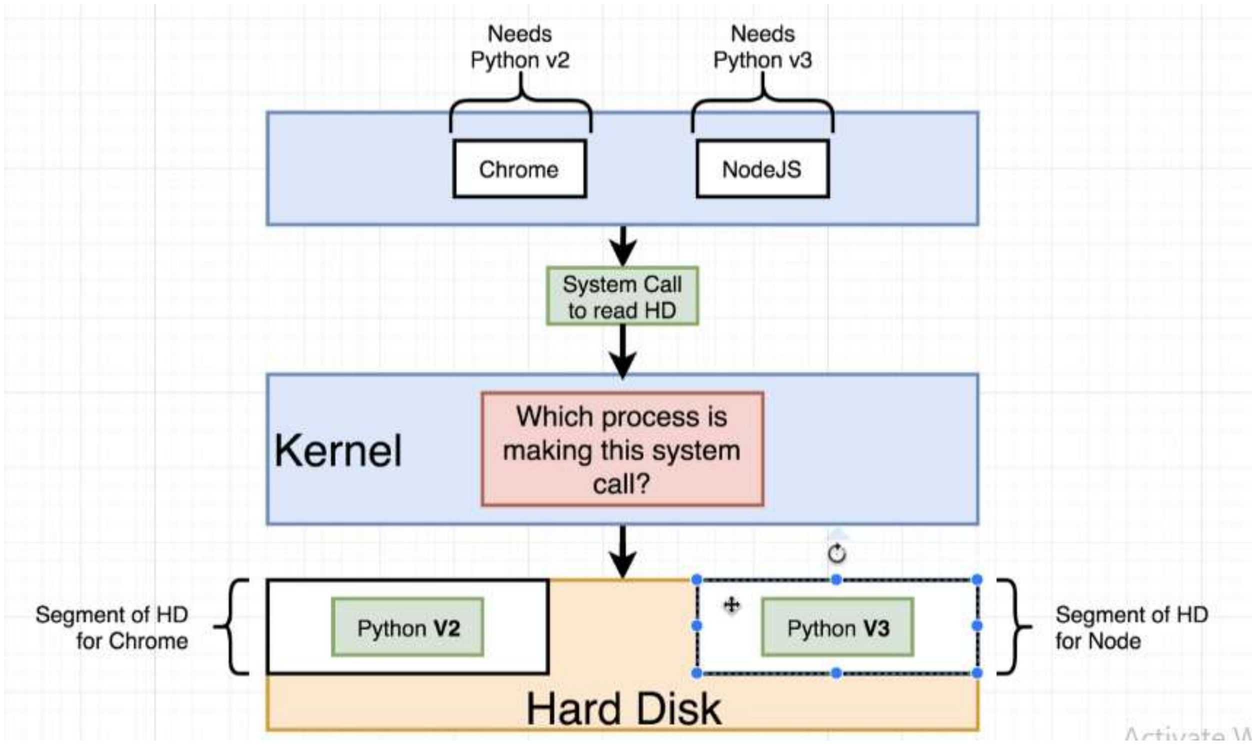
Suppose python v2 is installed on our computer and we have two applications: chrome which needs python v2 to run and nodejs which needs python v3 to run, and it is impossible to install two versions of python on the same computer.



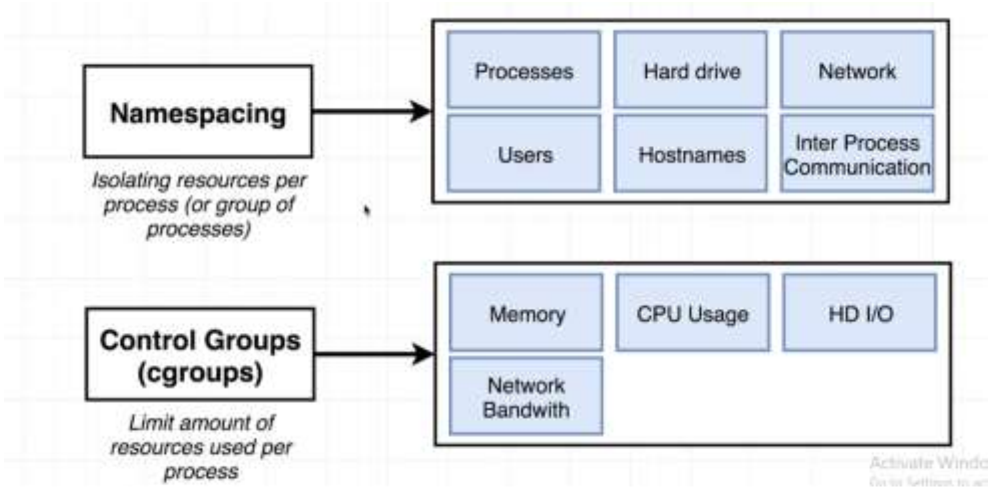
In this case only chrome will work because its dependency is available but nodejs will not work. How can we solve that issue and have nodejs also work? Think about this situation for a second because it leads to us what exactly a container is.

Use of Namespacing

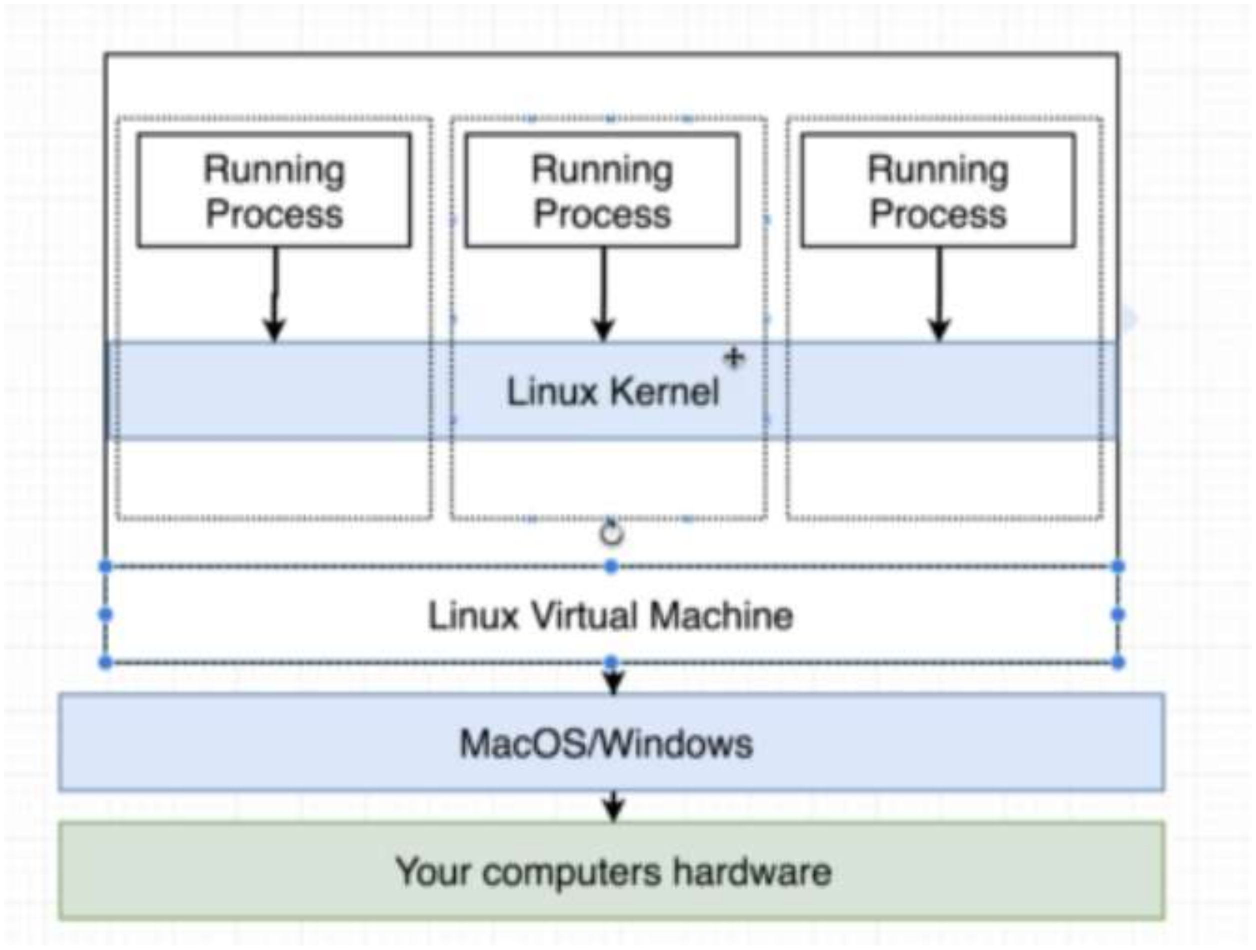
We use what we call namespace. Where basically looks at all hardware resources connected to our computer and we can essentially segment out portions of those resources.



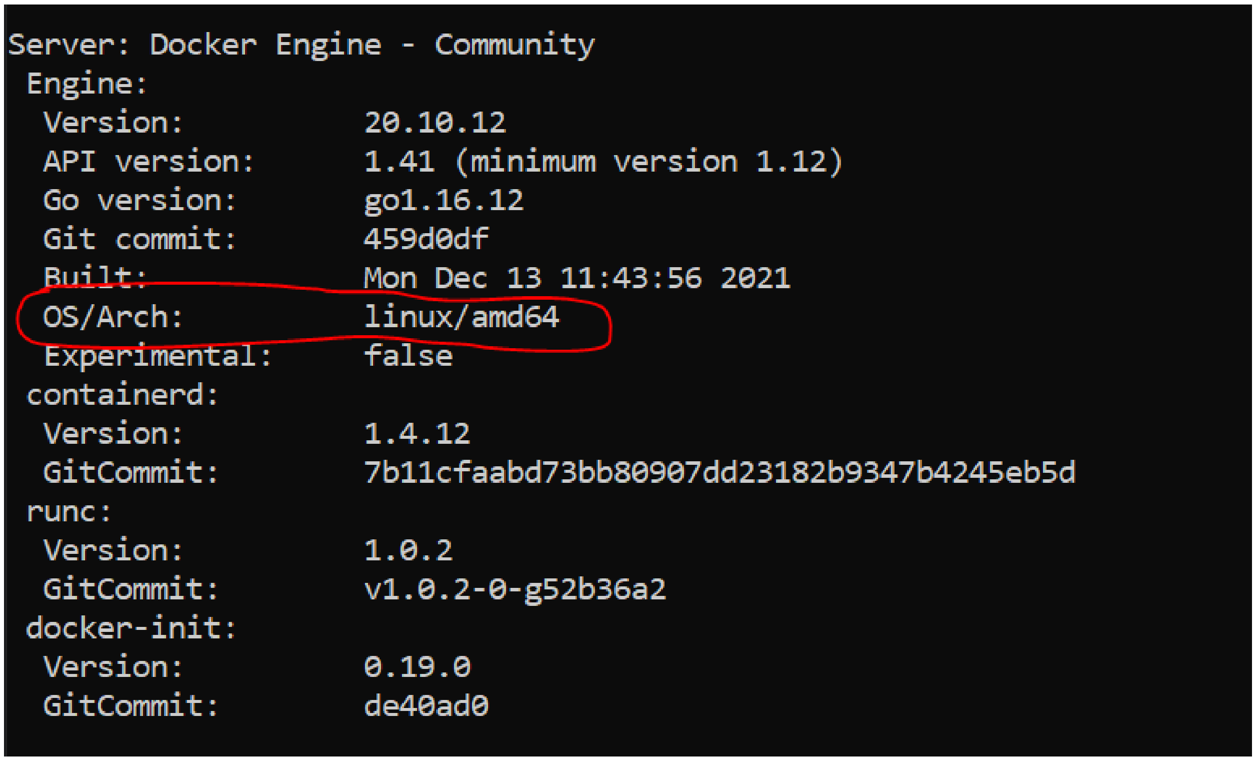
So we can specifically segment out resources of our hard disk to house python v2 and another segment for version 3. So when the call comes from chrome kernel will direct the call to segment housing python v2 and when the call comes from nodejs it will be directed to segment housing python v3. Eventually both versions will co-exist on the same hardware but housed in different segments. That mechanism of isolating resources depending on a process that needs it is called namespacing. It works pretty much the same for even other resources like network, I/O interfaces etc. Namespacing is also similar to another process called control groups. Both concepts are linux specific features.


Namespacing
Isolating resources per process (or group of processes)
Processes
Hard drive
Network
Users
Hostnames
Inter Process Communication
Control Groups (cgroups)
Limit amount of resources used per process
Memory
CPU Usage
HD I/O
Network Bandwith

You can now wonder how does docker work on windows/mac systems. There must be a linux virtual machine on top of your OS for docker to work.

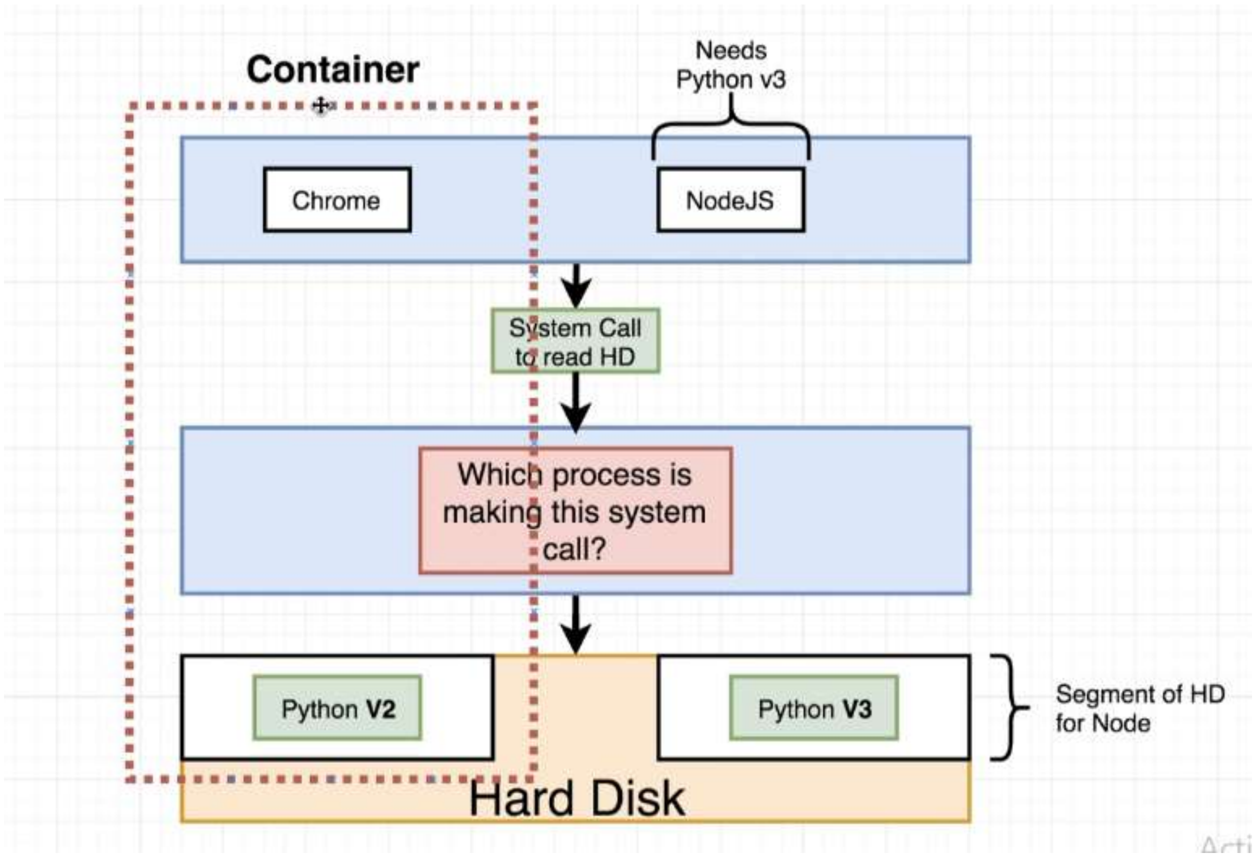


You can actually see it by running docker version command

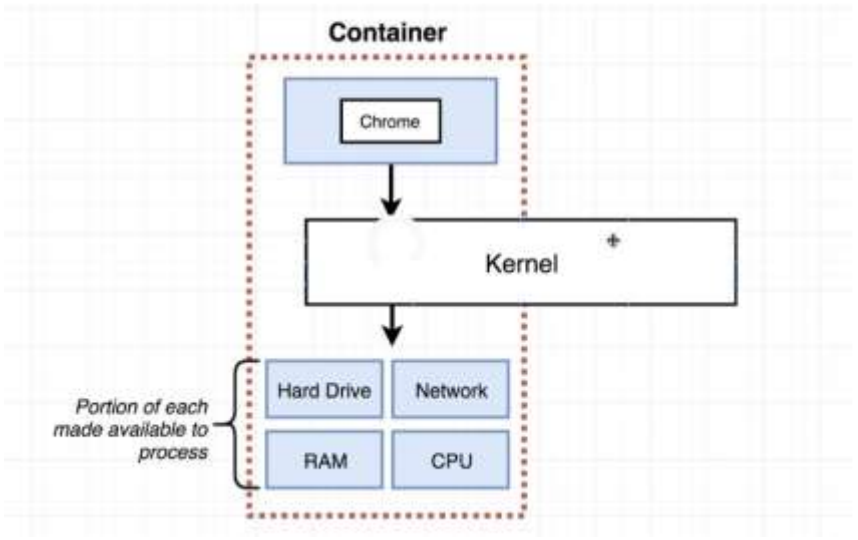


```
Server: Docker Engine - Community
 Engine:
  Version:          20.10.12
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.16.12
  Git commit:       459d0df
  Built:            Mon Dec 13 11:43:56 2021
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.4.12
  GitCommit:        7b11cfaabd73bb80907dd23182b9347b4245eb5d
 runc:
  Version:          1.0.2
  GitCommit:        v1.0.2-0-g52b36a2
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

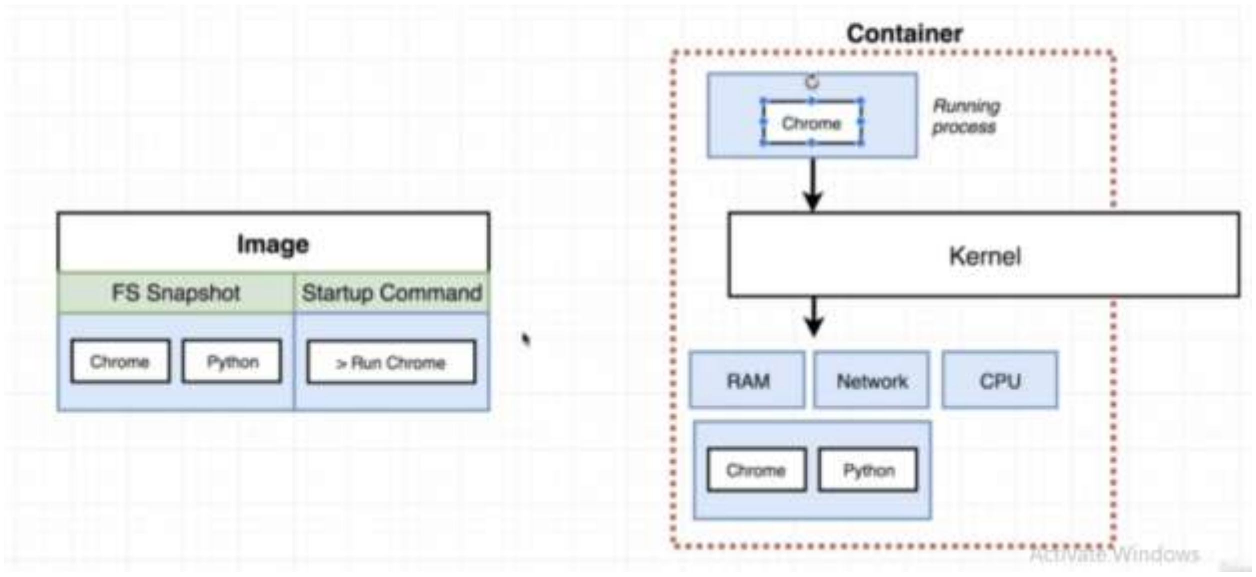
This entire process of isolating resources for chrome and nodejs to co-exist is also known as containerization.



So a container is a process or processes that have a group of resources assigned to it as figure below illustrates



Now that we have understood how containers work and what exact they are, one can ask him/her self how does the one image file eventually creating a container.



An image is abasically a file system snapshot (a copy paste of a very specific set of files or directories), we might have an image containing just python and chrome, an image will also contain a specific start up command. When an instance of an image is run/created, a container is created in its own segment as shown above with isolated resources on our computer memory, only chrome and python are placed/installed in that drive/segment availed then depending on the start up command say chrome will start to run.

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 command
The following command runs an ubuntu container, attaches interactively to your local command-line session, and runs /bin/bash.

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

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

1. 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.
2. Docker creates a new container, as though you had run a docker container create command manually.

3. 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.
4. 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.
5. 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.
6. 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.

Container technology can be thought of as three different categories:

- **Builder**: a tool or series of tools used to build a container, such as distrobuilder for LXC, or a Dockerfile for Docker.

- **Engine**: an application used to run a container. For Docker, this refers to the **docker** command and the **dockerd** daemon. For others, this can refer to the **containerd** daemon and relevant commands (such as **podman**.)

- **Orchestration**: technology used to manage many containers, including Kubernetes and OKD.

Containers often deliver both an application and configuration, meaning that a sysadmin doesn't have to spend as much time getting an application in a container to run compared to when an application is installed from a traditional source. Dockerhub and Quay.io are repositories offering images for use by container engines.

The greatest appeal of containers, though, is their ability to "die" gracefully and respawn when load balancing demands it. Whether a container's demise is caused by a crash or because it's simply no longer needed because server traffic is low, containers are "cheap" to start, and they're designed to seamlessly appear and disappear. Because containers are meant to be ephemeral/short-lived and to spawn new instances as often as required, it's expected that monitoring and managing them is not done by a human in real time, but is instead automated.

Container orchestration

Like any other element of your IT infrastructure, containers need to be monitored and controlled. Otherwise, you literally have no idea what's running on your servers.

You can use DevOps programs to deploy and monitor Docker containers but they're not optimized for containers. As DataDog, a cloud-monitoring company, points out in its report on real-world Docker adoption, "Containers' short lifetimes and increased density have significant implications for infrastructure monitoring. They represent an order-of-magnitude increase in the number of things that need to be individually monitored."

The answer is cloud orchestration tools. These monitor and manage container clustering and scheduling. In May 2017, there were three major cloud container orchestration programs: Docker Swarm, Kubernetes, and Mesosphere. Today, these are all still around, but Kubernetes is by far the most dominant cloud-orchestration program.

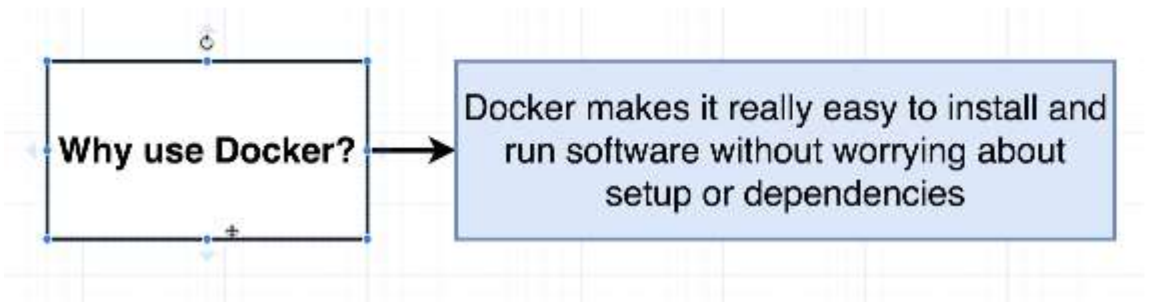
Indeed, in early October, Mesosphere jumped on the Kubernetes bandwagon. Docker announced it will integrate Kubernetes into the Docker platform. Users can choose to use Kubernetes and/or Docker Swarm for orchestration.

The underlying technology

Docker is written in the Go programming language and takes advantage of several features of the Linux kernel to deliver its functionality. Docker uses a technology called namespaces to provide the isolated workspace called the *container*. When you run a container, Docker creates a set of *namespaces* for that container.

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Why docker


Why use Docker?
Docker makes it really easy to install and run software without worrying about setup or dependencies

These are main reason why docker is important:

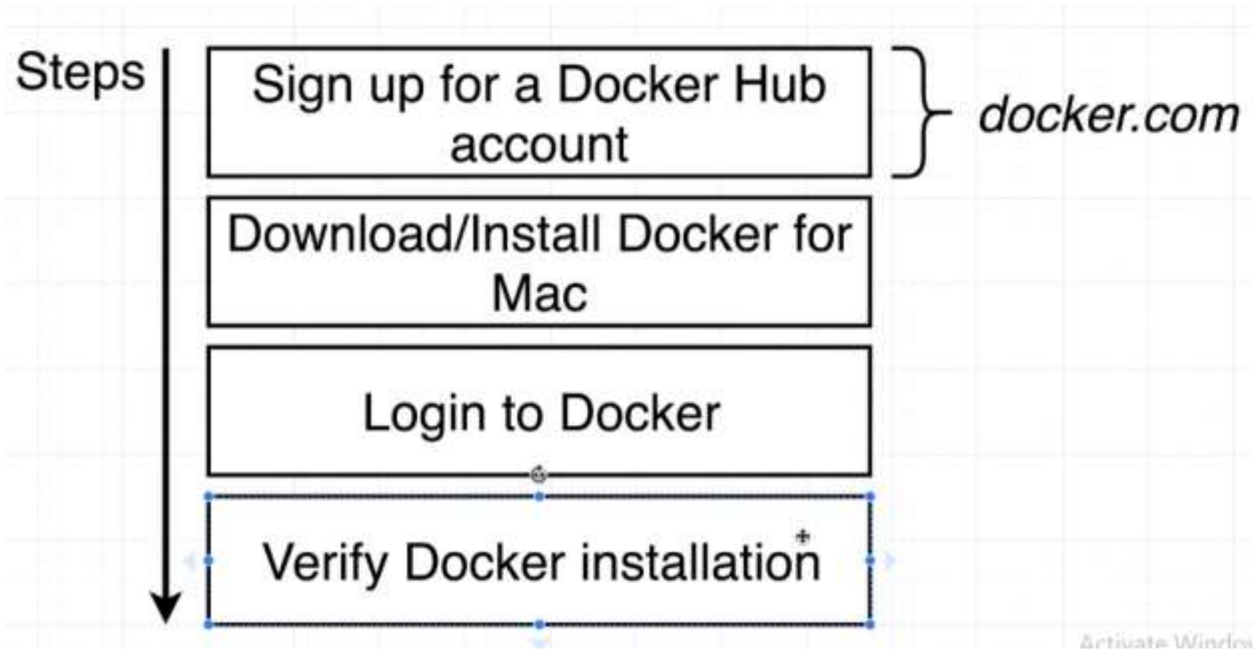
- **Keep it Simple**: Docker's friendly, CLI-based workflow makes building, sharing, and running containerized applications accessible to developers of all skill levels.

- **Move Fast**: Install from a single package to get up and running in minutes. Code and test locally while ensuring consistency between development and production.

- **Collaborate**: Use Certified and community-provided images in your project. Push to a cloud-based application registry and collaborate with team members.

Docker setup

Installation of docker goes through below steps for users of linux based OS


Steps
Sign up for a Docker Hub account
docker.com
Download/Install Docker for Mac
Login to Docker
Verify Docker installation

For more details and guidance for docker installation go to the docker portal https://docs.docker.com/get-docker/

Installing Docker with WSL2 on Windows 10 Home and Pro

WSL2

Windows 10 Pro and some Windows 10 Home users will be able to install Docker Desktop if their computer supports the Windows Subsystem for Linux (WSL2).

Please see this guide first which includes all of the steps to install and enable WSL2:

https://docs.microsoft.com/en-us/windows/wsl/install-win10

Once you have successfully installed and enabled WSL2 and then installed the Linux OS of your choice, continue to the Docker Desktop installation docs here:

https://docs.docker.com/docker-for-windows/wsl/

With Docker Desktop installed using WSL2 as a backend, you will now be able to follow along with the course. You will access the applications at localhost just like we do in the video lectures.

Important - A significant difference when using WSL2 is that you will need to create and run your project files from within the Linux filesystem, not the Windows filesystem. This will be very important in later lectures when we cover volumes.
You can access your Linux system by running `wsl` in the Cortana bar. Going forward, all Docker commands should be run within WSL2 and not on the Windows file system.

Docker Toolbox

If you are using a Windows 10 machine that does not support WSL2, or, you are using even older versions such as Windows 7, you will need to install Docker Toolbox. It is important to note, however, that this software has been officially deprecated by the Docker engineers and may not continue to work into the future.

Release downloads are available here:
https://github.com/docker/toolbox/releases
Download the .exe for the latest release and run the installer. Docker Toolbox will setup and install everything you need including VirtualBox.

You may also need to enable virtualization in your computer's BIOS settings. This will be different for each manufacturer, please refer to their documentation on which keys to use to access these settings on reboot.

After Toolbox is finished installing, open the Docker Quickstart Terminal. This will complete the setup and provision your VirtualBox machine.

Launch the Docker QuickStart terminal and type the command:

```
docker run hello-world
```

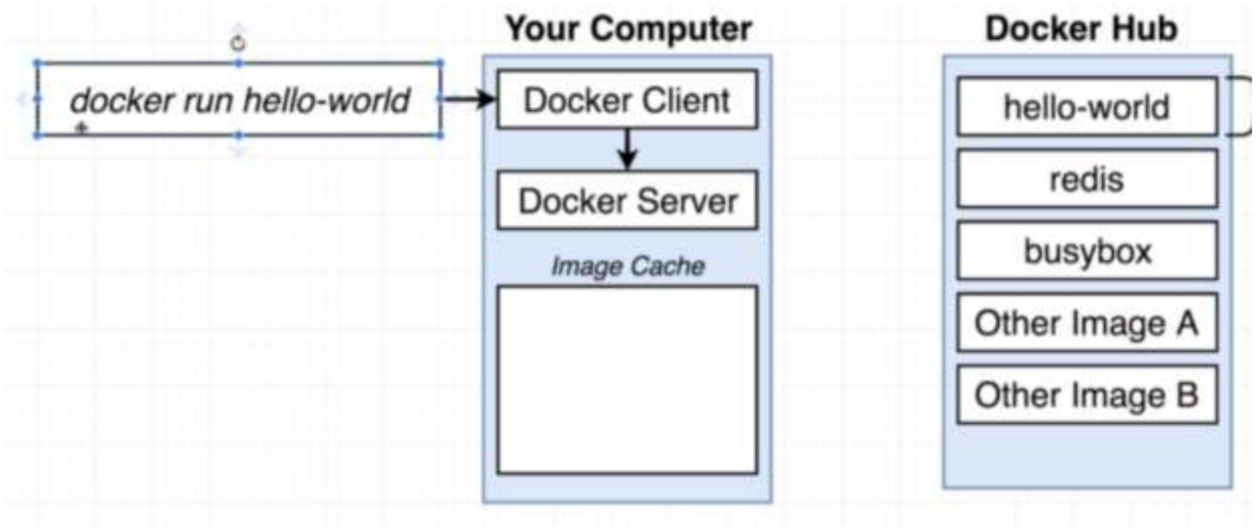
This should pull down the test container and print hello-world to your screen.

Important - A major difference between the course lectures using Docker Desktop vs. using Docker Toolbox is that you will not be able to use localhost anymore. Instead, you will need to use the IP address returned by running `docker-machine ip`

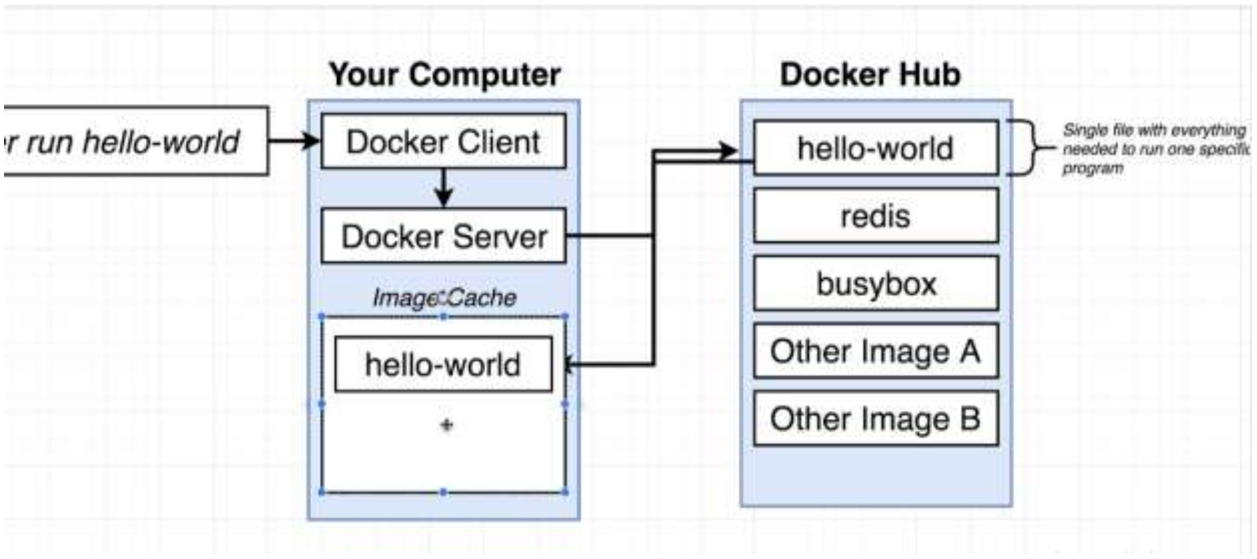
Very commonly this is **192.168.99.100** but for you, it could be slightly different.

Using docker client

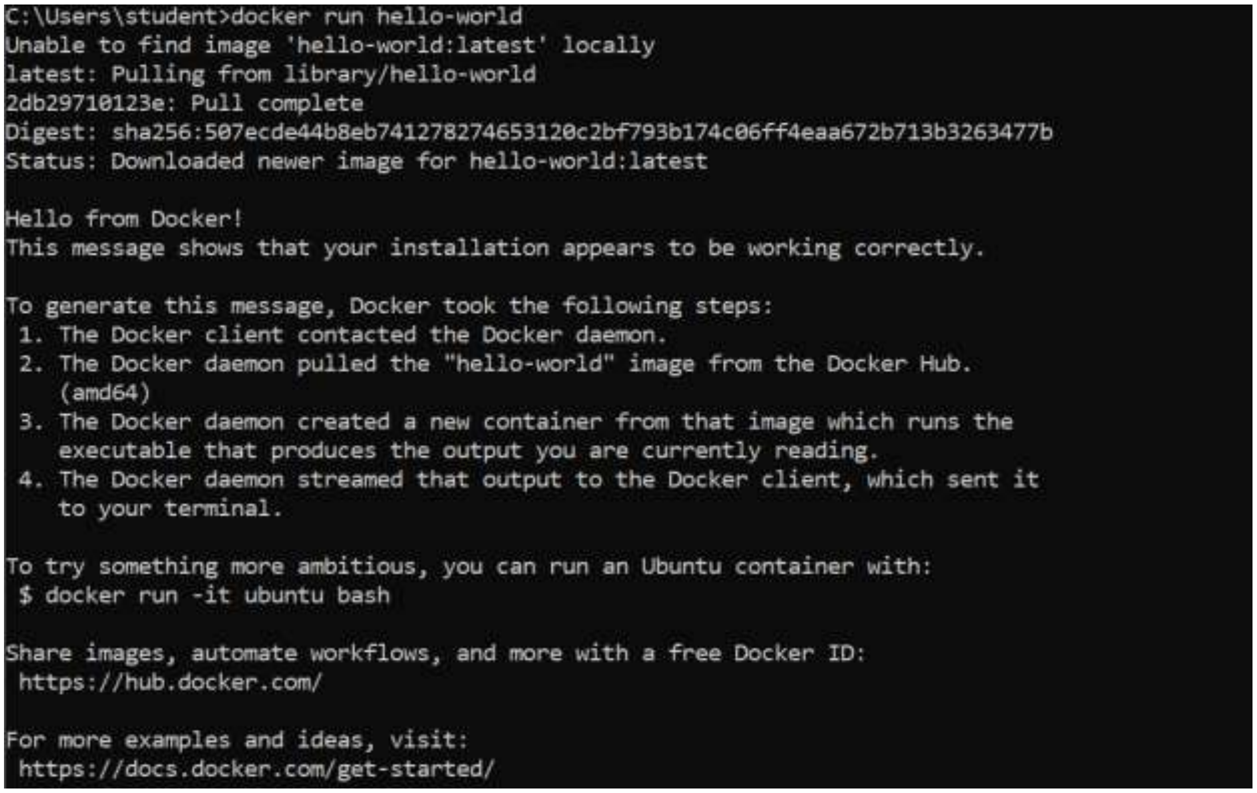
If you run command docker run hello-world the docker runs many processes summarized in below diagram behind the scenes



- Docker clients asks the docker server to check for an image from the image cache called hello-world
- Initially the image cache is empty, so the docker deamon makes a call to docker registry(hub) online for the same image
- Docker hub finds the requested image and it is saved in the image cache

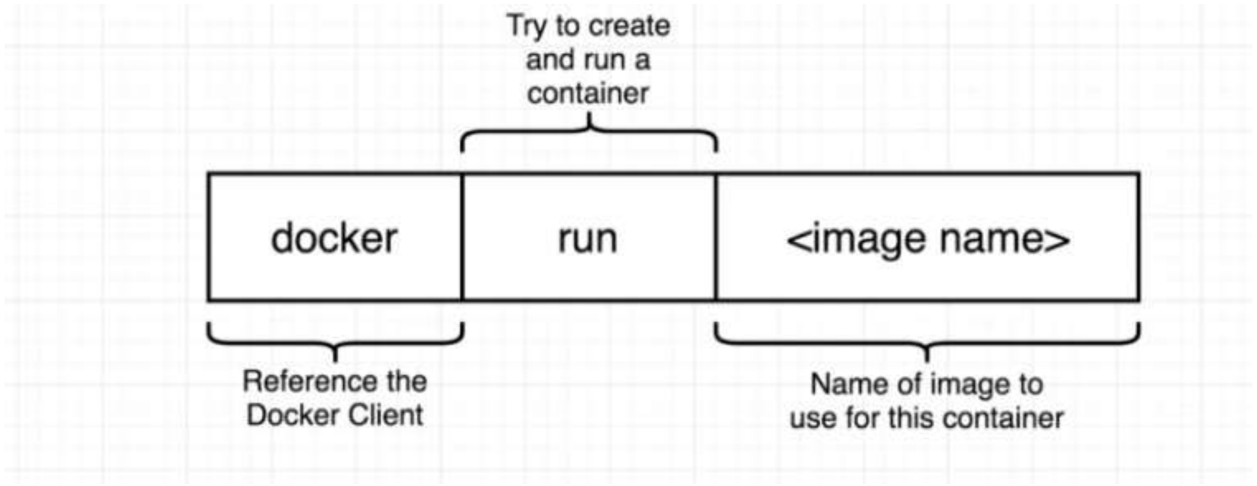


- Then docker deamon creates and runs a container of hello-world image
- As a result the container prints out below output



```
C:\Users\student>docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:507ecde44b8eb741278274653120c2bf793b174c06ff4eaa672b713b3263477b
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

➢ **Creating and running a container from an image**


Try to create and run a container
docker
run
<image name>
Reference the Docker Client
Name of image to use for this container

Eg. Docker run hello-world

- **Overriding default commands**

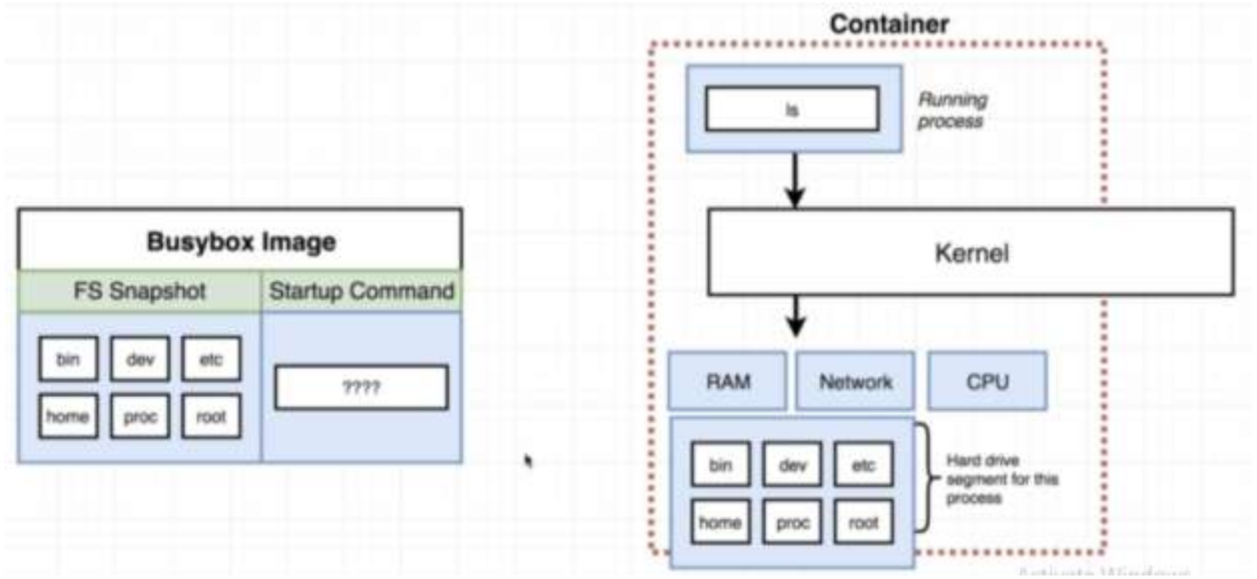
Let us run another image say busybox, here is the command

$ docker run busybox

We can now override it as follows

$ docker run busybox echo I am testing

$ docker run busybox ls



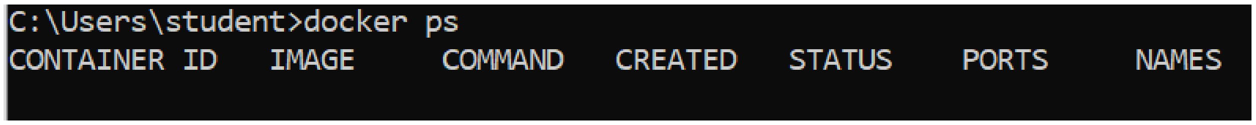
So what happens when we run above commands say on our initial image hello-world?

The answer is it will display an error because those commands are not executable as per file system of hello-world image as opposed to busybox image which has them supported already. The point is, you do not just run any command on an image, the image its own file system and supported commands to go with it during container creation.

- **Listing running containers**

$ docker ps

Out put would be



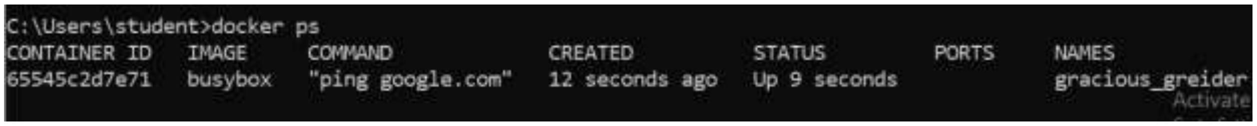
```
C:\Users\student>docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

All the images we're running sofar do run and exit immediately, that's why you can't see any in above. To be able to see list or running containers let us run busybox but now with a command that will stay longer say

$ docker run busybox ping google.com

Then run ps again

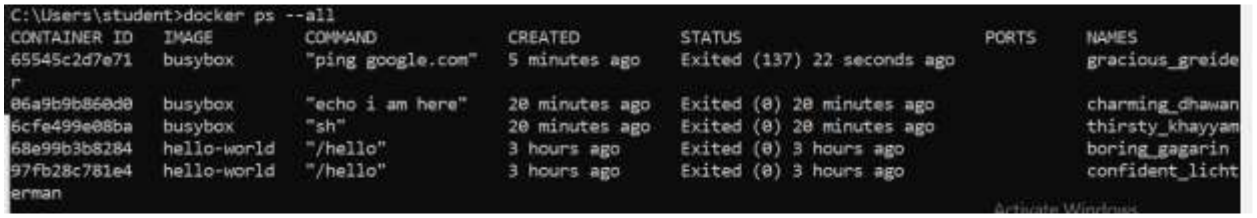
$ docker ps



```
C:\Users\student>docker ps
CONTAINER ID   IMAGE     COMMAND             CREATED          STATUS         PORTS     NAMES
65545c2d7e71   busybox   "ping google.com"   12 seconds ago   Up 9 seconds             gracious_greider
```

$ docker ps --all

List all containers that we have ever run on this computer



```
C:\Users\student>docker ps --all
CONTAINER ID   IMAGE         COMMAND             CREATED          STATUS                        PORTS     NAMES
65545c2d7e71   busybox       "ping google.com"   5 minutes ago    Exited (137) 22 seconds ago             gracious_greide
r
06a9b9b860d0   busybox       "echo i am here"    20 minutes ago   Exited (0) 20 minutes ago               charming_dhawan
6cfe499e08ba   busybox       "sh"                20 minutes ago   Exited (0) 20 minutes ago               thirsty_khayyam
68e99b3b8284   hello-world   "/hello"            3 hours ago      Exited (0) 3 hours ago                  boring_gagarin
97fb28c781e4   hello-world   "/hello"            3 hours ago      Exited (0) 3 hours ago                  confident_licht
erman
```

Important Docker Commands List

Here are some commands to try out on the command line. Just make sure you have Docker running!

docker run

Description: docker run is probably the most important command on this Docker list. It creates a new container, then starts it using a specified command. It is the equivalent of using docker create and then immediately using docker start (we‘ll go through those below).

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Example: docker run hello-world

This example will download, create and run the ‘hello-world’ container from docker hub. Try it out!

docker create

Description: This command creates a new container and prepares it to run the specified command, but doesn’t run it. The ID is printed to STDOUT. This command is similar to docker run, but the container is never started. It can be started at any time with docker start <container_id>.

This can be useful if you want to create and prepare a container for running at a later time.

Usage: docker create [OPTIONS] IMAGE [COMMAND] [ARG...]

Example: docker create hello-world

This will download and create the hello-world container same as above, but it won’t run it.

docker start

Description: Starts container, either created with docker create or one that was stopped.

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Example: docker start hello-world

docker stop

Description: Stops a running container. More info

Usage: docker stop [OPTIONS] CONTAINER [...]

Example: docker start hello-world

This example will stop the container we started in the example above. It sends message SIGTERM, it notifies the dependents to do clean up, like saving files etc. It therefore shuts down a container gracefully but if the process does not stop for 10 seconds then docker kill process will take control and kills the process brutally. You can use docker start to start it back up again.

docker ps

Description: Use this command to list the docker containers available. By default, only running ones are listed, but you can add options such as **-a** to get a docker container list. More info

Usage: docker ps [OPTIONS]

Example: docker ps --all

The above command will display all containers on your system. We saw the output above

docker rm

Description: Removes a container. Once you use docker ps to list available containers, you can use this command to remove one or more of them.

Usage: docker rm [OPTIONS] CONTAINER [CONTAINER...]

Example: docker rm -f redis

This example will remove the container called ‘redis’. The -f tag forces the action, meaning the container will receive a SIGKILL command first.

docker system prune

Description: Removes all images, their containers. Once you use docker ps –all to list available containers, the list will be empty and you will need to download them again when you run docker run <image>.

Usage: docker system [OPTIONS] prune

Example: docker rm -f redis

This example will remove the container called ‘redis’. The -f tag forces the action, meaning the container will receive a SIGKILL command first.

docker rmi

Description: rmi stands for *remove image*. Similar to docker rm but removes the docker image rather than a running instance of an image (called a container).

Usage: docker rmi [OPTIONS] IMAGE [IMAGE...]

Example: docker rmi test

This example will remove the image name ‘test’. Use the docker images command to get a list of images and ids.

docker login

Description: Let’s you login to a docker registry. You can use this command to log into any public or private registry. You’ll probably use this to log into Docker Hub.

Usage: docker login [OPTIONS] [SERVER]

Example: docker login localhost:8080

Here we are logging into a self-hosted registry by specifying the server name. Otherwise, just use docker login to login to your Docker Hub registry.

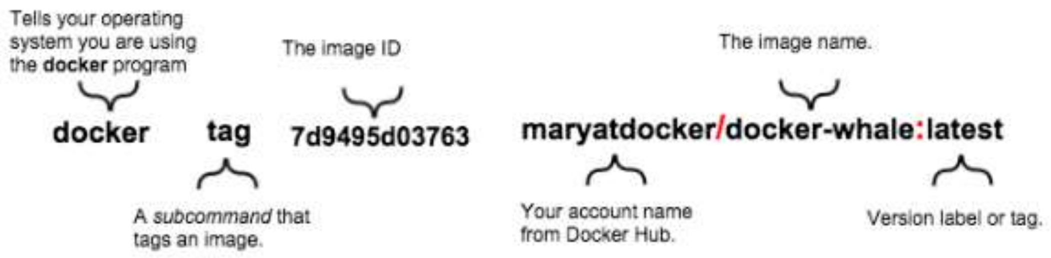
docker tag

Description: Use this command to create a tag TARGET_IMAGE that refers to SOURCE_IMAGE.

Usage: docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]

Example: docker tag 7d9495d03763 maryatdocker/docker-whale:latest

Here we are tagging a local image with ID “7d9495d03763″ into the “maryatdocker/docker-whale” repository with the tag “latest”


Tells your operating system you are using the docker program
The image ID
The image name.
docker
tag
7d9495d03763
maryatdocker/docker-whale:latest
A subcommand that tags an image.
Your account name from Docker Hub.
Version label or tag.

docker push

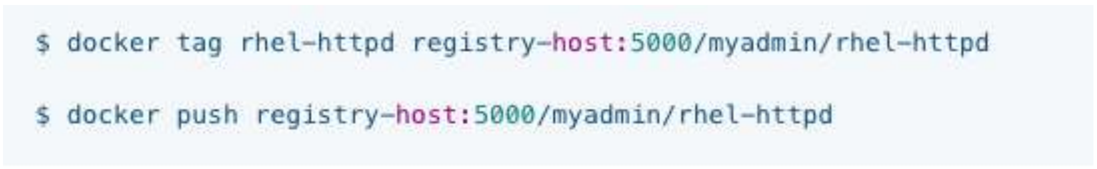
Description: docker push is the command you'll use to share your images to Docker Hub or another registry. It's a key part of the deployment process, and you'll need to use it with docker tag and docker login to work properly.

Usage: docker push [OPTIONS] NAME[:TAG]

Example: docker push maryatdocker/docker-whale

This example will push the image we tagged above to the repository. It's actually the example from the docs, check out the full page here.

Combine it with the tag command like this:

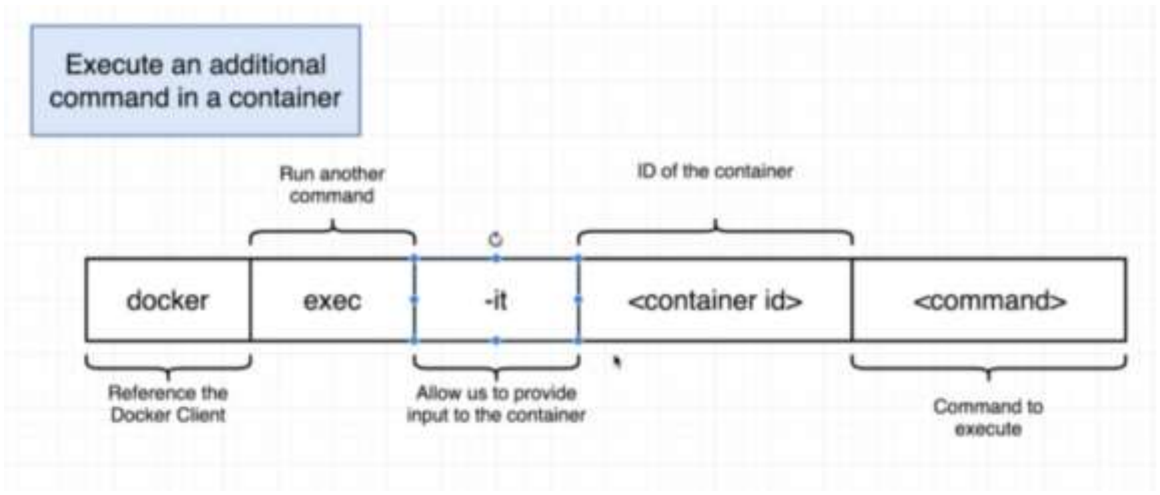


```
$ docker tag rhel-httpd registry-host:5000/myadmin/rhel-httpd

$ docker push registry-host:5000/myadmin/rhel-httpd
```

docker exec

Description: The exec command can be used to run a command in a container that is already running. docker run lets us run a command when starting a container, docker exec can be used to feed subsequent commands to a container.

Usage: docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

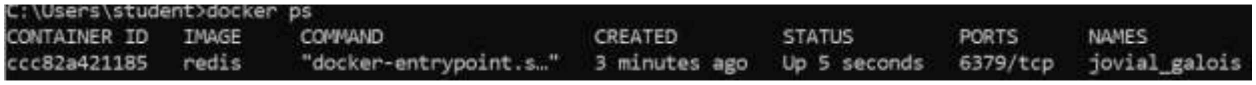

Execute an additional command in a container
Run another command
ID of the container
docker
exec
-it
<container id>
<command>
Reference the Docker Client
Allow us to provide input to the container
Command to execute

Example:

First start a container with docker run redis and run docker ps to check if redis container is running

$ docker run redis

$ docker ps



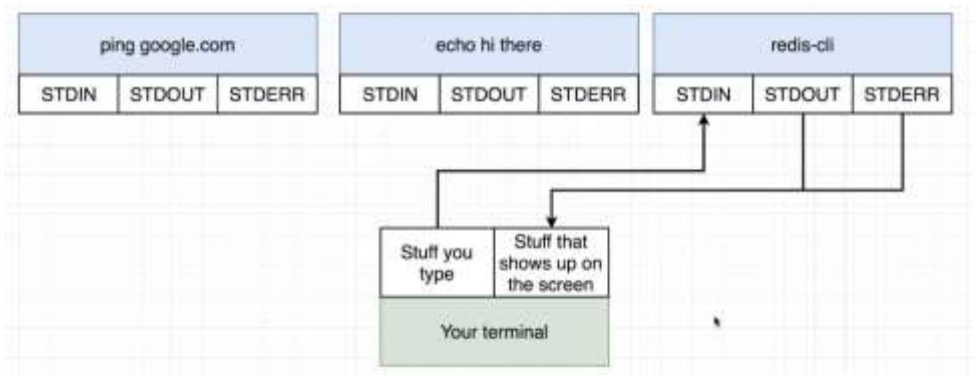
```
C:\Users\student>docker ps
CONTAINER ID   IMAGE    COMMAND                  CREATED         STATUS         PORTS      NAMES
ccc82a421185   redis    "docker-entrypoint.s…"   3 minutes ago   Up 5 seconds   6379/tcp   jovial_galois
```

Now we can execute a command on that container to interact with redis-cli with docker exec

$ docker exec –it ccc82a421185 redis-cli

Note: the use of –it, if you ignore it you wont be able to enter text in redis-cli. This is because we are working with linux virtual environment and as we know linux process involves three operation STDIN, STDOUT, STDERR as illustrated in diagram below. Basically –it is a combination of two commands i.e –i which attaches our input to this container and -t which formats any output in good and user friendly display format. We would also write our command as

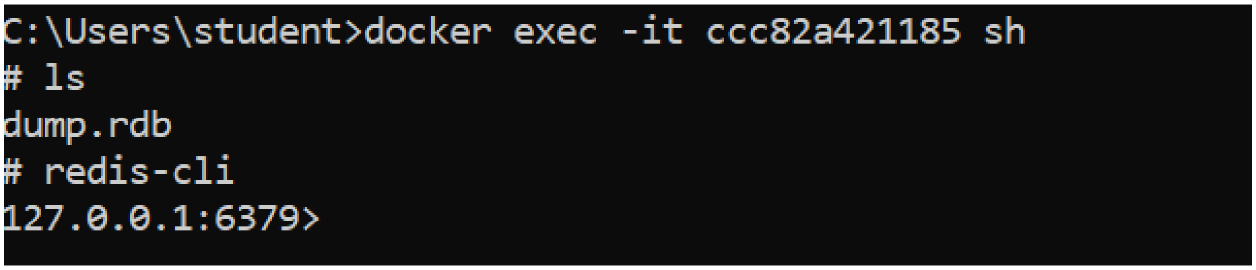
$ docker exec –i –t ccc82a421185 redis-cli



Command prompt in a container

docker exec is used to enter into powershell/bash/zsh terminal by using sh as shown below and you can run linux commands inside a container.

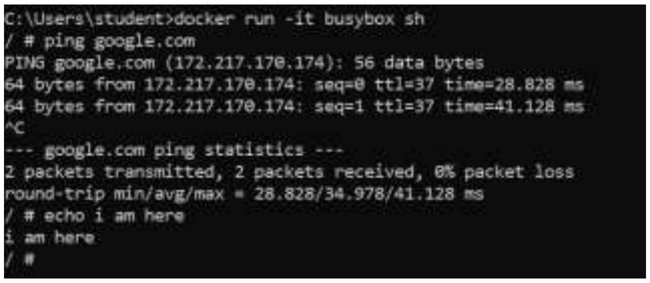
$ docker exec –it <containerId> sh



```
C:\Users\student>docker exec -it ccc82a421185 sh
# ls
dump.rdb
# redis-cli
127.0.0.1:6379>
```

However, –it and sh can be used with docker run as well though not recommended since you wont be able to run other process.

$ docker run –it busybox sh



```
C:\Users\student>docker run -it busybox sh
/ # ping google.com
PING google.com (172.217.170.174): 56 data bytes
64 bytes from 172.217.170.174: seq=0 ttl=37 time=28.828 ms
64 bytes from 172.217.170.174: seq=1 ttl=37 time=41.128 ms
^C
--- google.com ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 28.828/34.978/41.128 ms
/ # echo i am here
i am here
/ #
```

docker logs

Description: Getting deeper into using docker, the docker logs command lets you see the log file for a running container. However, it can only be used with containers that were started with the json-file or journa ld logging driver

Usage: docker logs [OPTIONS] CONTAINER

Example: docker logs -f MyCntr

docker kill

Description: Sometimes you just need to kill one or all running containers. Similar to SIGKILL (this is actually the message it will receive), docker kill will stop a container running. It stops the container brutally and no signal sent to dependents for clean up, like saving files, etc

Usage: docker kill [OPTIONS] CONTAINER [CONTAINER...]

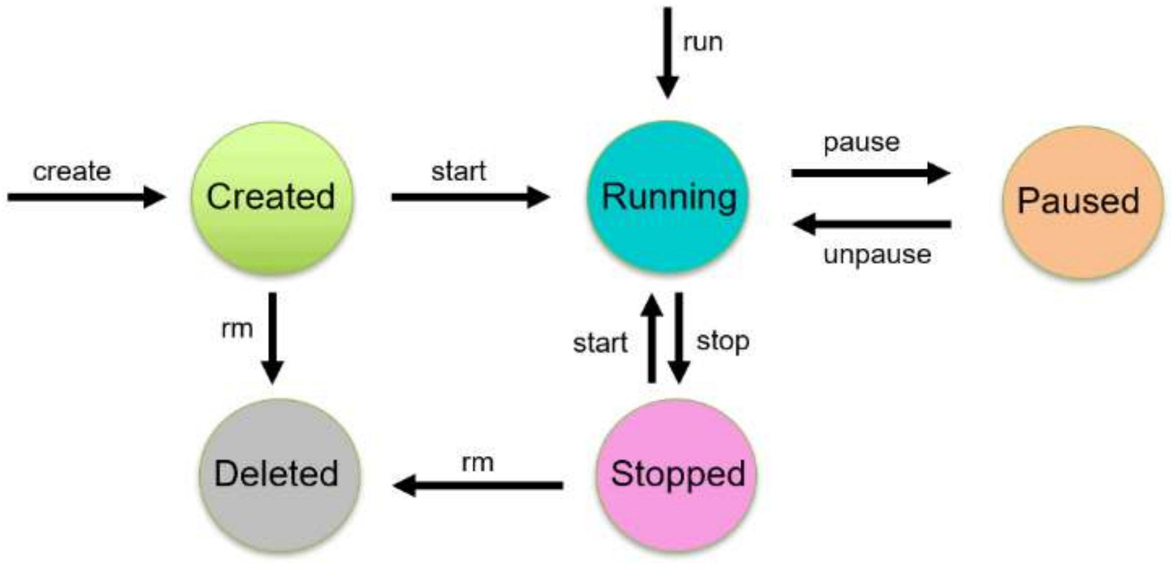
Example: docker kill MyCntr

docker swarm

Description: Use docker swarm to manage a swarm. You can initialize a swarm, add manager and worker nodes, and update a swarm. This command is actually a whole bunch of commands in one – take a look at all of the child commands here.

Example: docker swarm init --advertise-addr <MANAGER-IP>

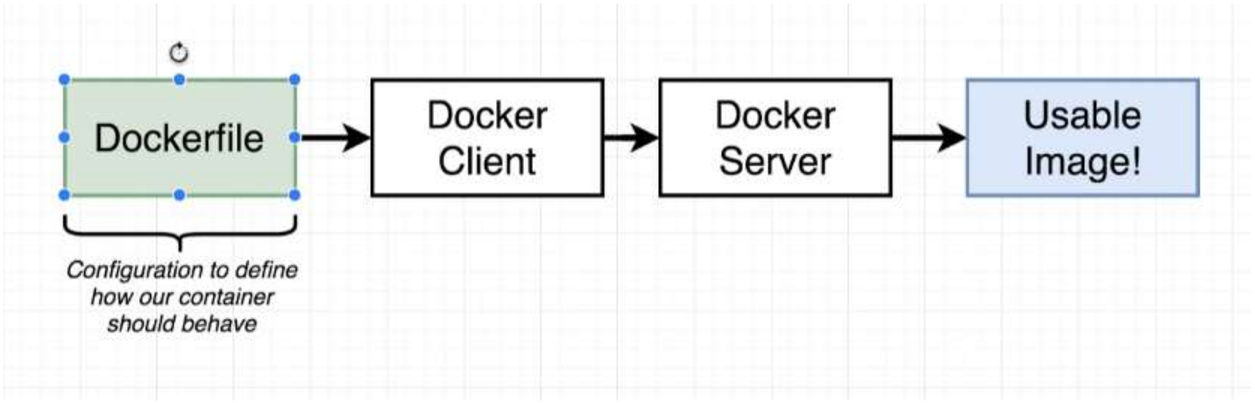
This will initialize a swarm and give us back the address to add workers to the swarm later. More about docker swarm is explained in next sections of this.



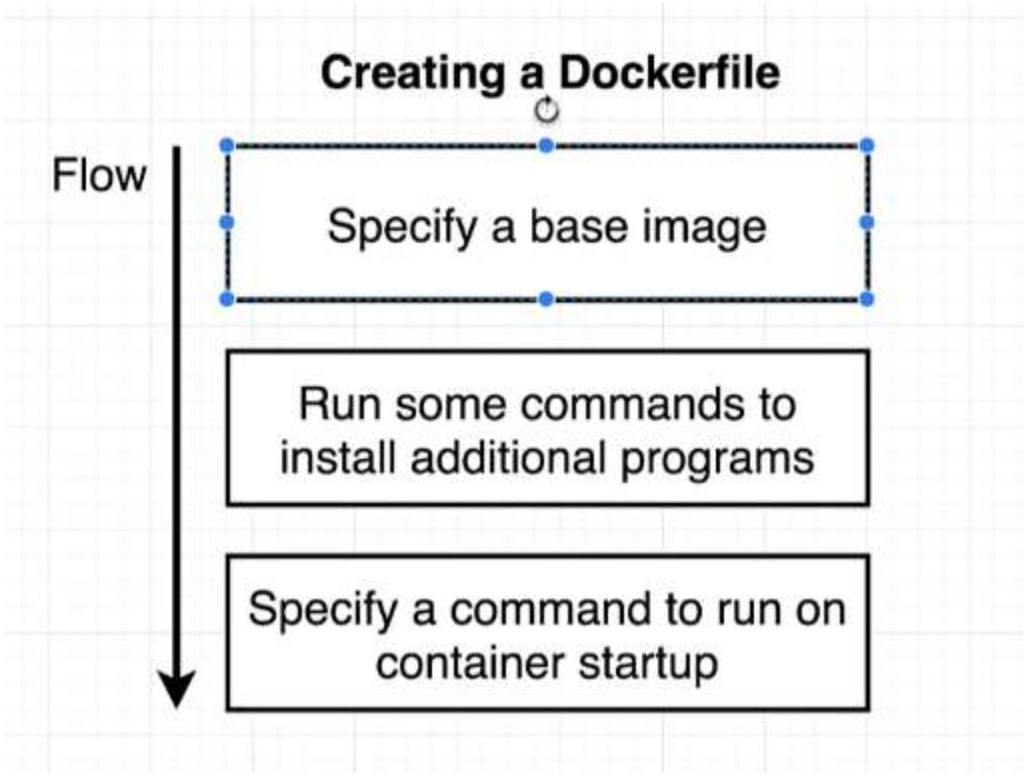
Summary of important docker commands

Building custom images through docker server

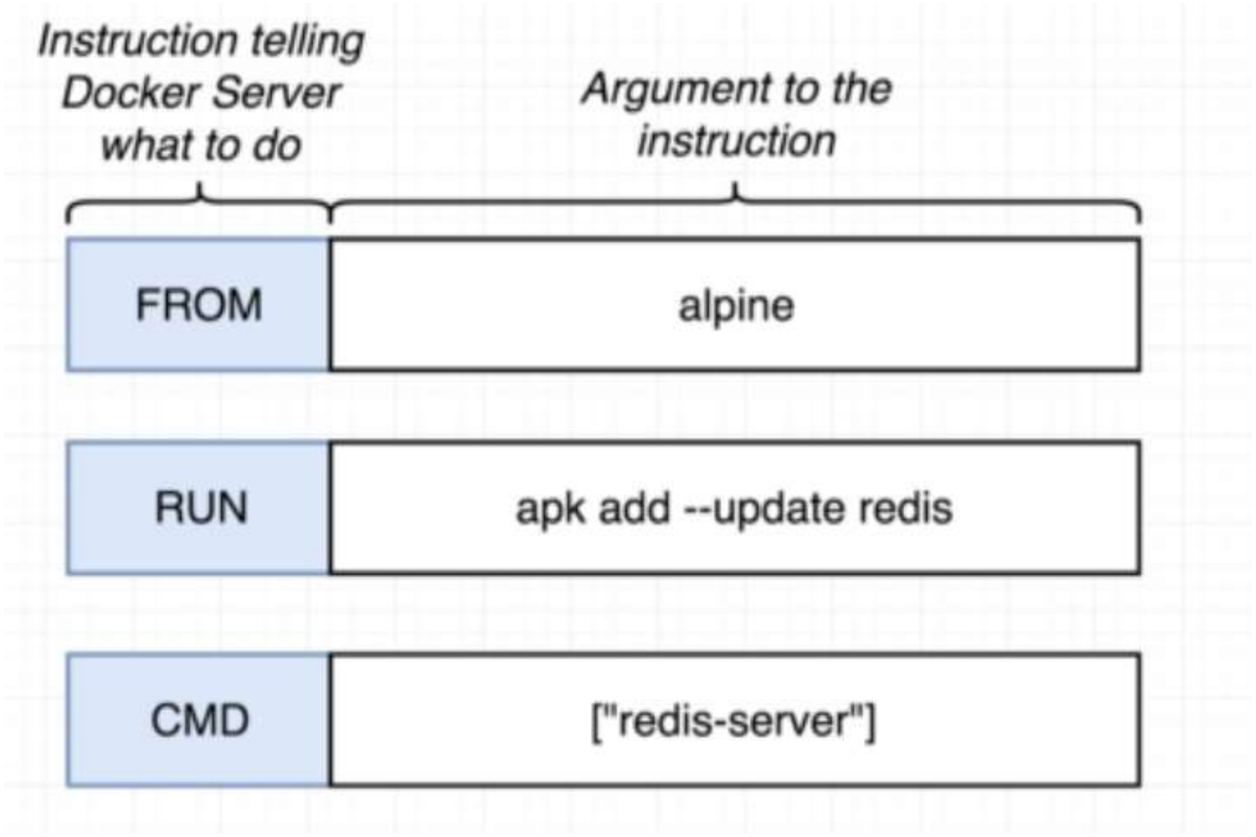
Building custom images is very simple you just have to follow below steps. You create a docker file which basically has everything the image needs from base image to start up commands. Once the file is ready the rest are docker commands as we are going to see them next.



Dockerfile is the base of building a docker image and below steps are key pretty much similar to all images created.


Creating a Dockerfile
Flow
Specify a base image
Run some commands to install additional programs
Specify a command to run on container startup

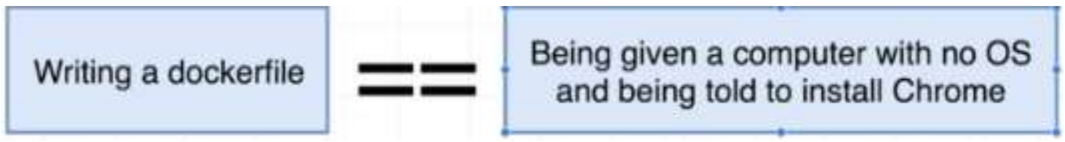
The docker file has a syntax similar to be below diagram


Instruction telling Docker Server what to do
Argument to the instruction
FROM
alpine
RUN
apk add --update redis
CMD
["redis-server"]

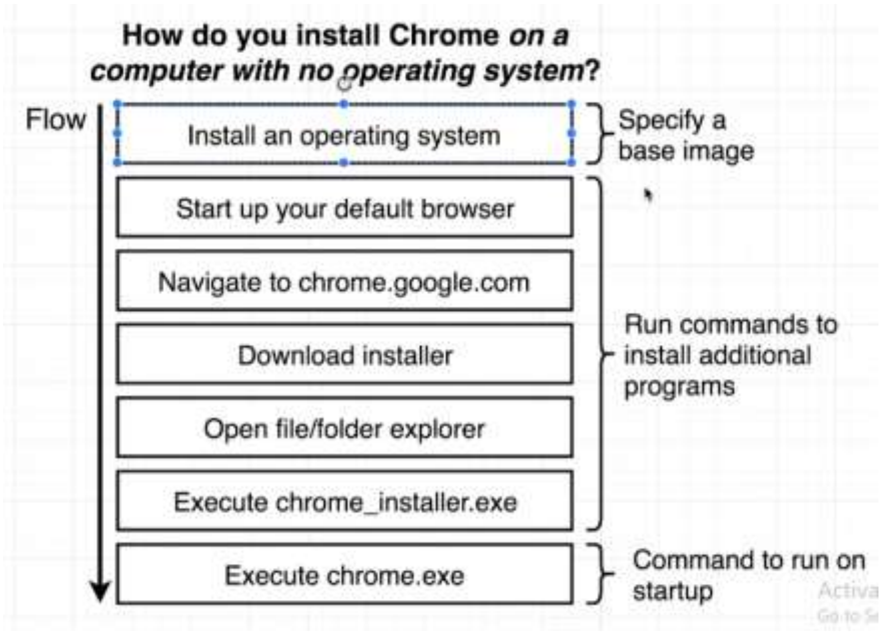
- The instruction tell docker server what to do during image creation, we have used FROM, RUN and CMD which are most common but there a handful of many other instructions that we shall get familiar with as we go a long in this course.
- The arguments, customizes how instructions are going to be executed

What is a base image

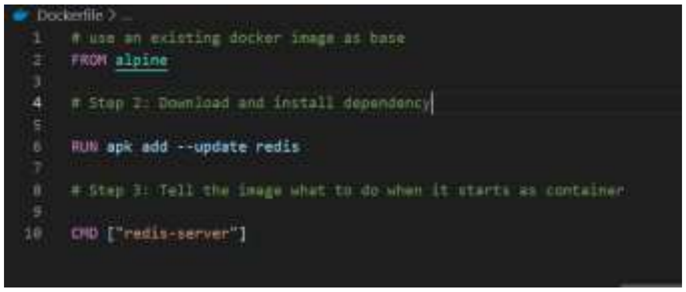
Let us begin with analogy, basically wring a docker file or building an image is like being given a computer without an OS and being told to install applications say chrome.


Writing a dockerfile
==
Being given a computer with no OS and being told to install Chrome

In that case what would you do? You'll pretty much do the following:


How do you install Chrome on a computer with no operating system?
Flow
Install an operating system
Start up your default browser
Navigate to chrome.google.com
Download installer
Open file/folder explorer
Execute chrome_installer.exe
Execute chrome.exe
Specify a base image
Run commands to install additional programs
Command to run on startup

Example docker file would be:



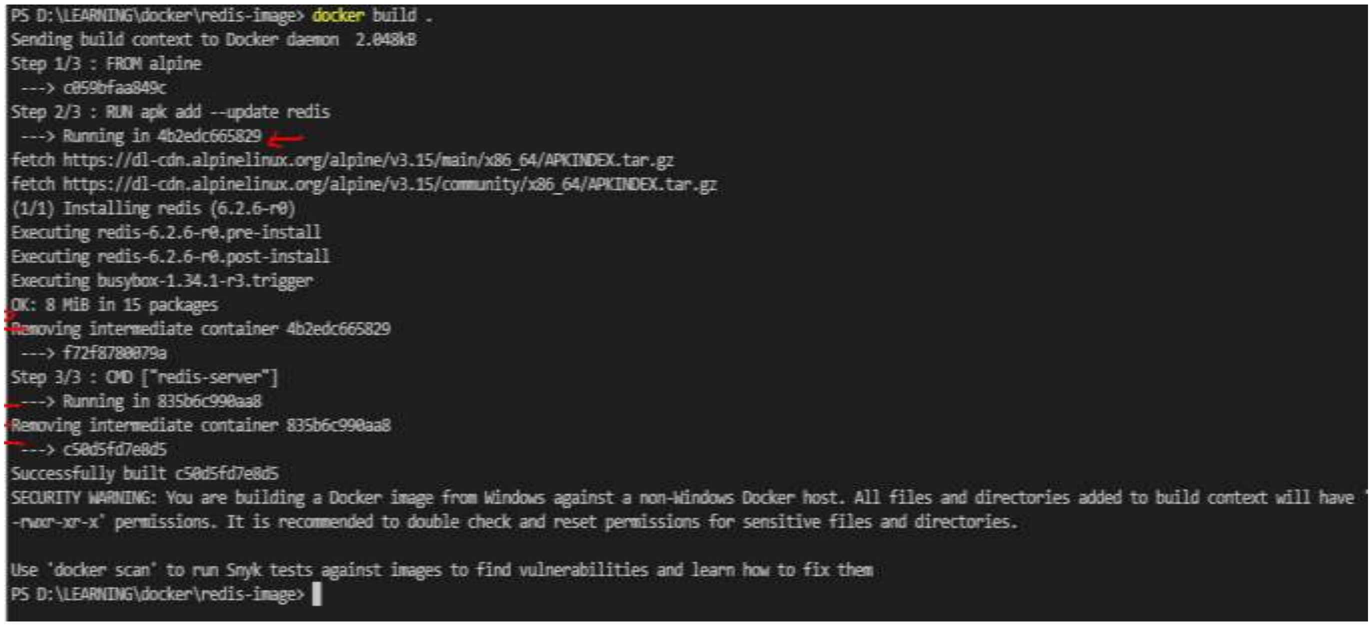
```
Dockerfile > ...
# use an existing docker image as base
FROM alpine

# Step 2: Download and install dependency

RUN apk add --update redis

# Step 3: Tell the image what to do when it starts as container

CMD ["redis-server"]
```

Therefore alpine is more of an OS for docker images, one would ask why alpine? So would be asked why people use mac or windows or Ubuntu, is it because they come with a preinstalled set of programs that are useful.

To build a new docker file from above example we run command

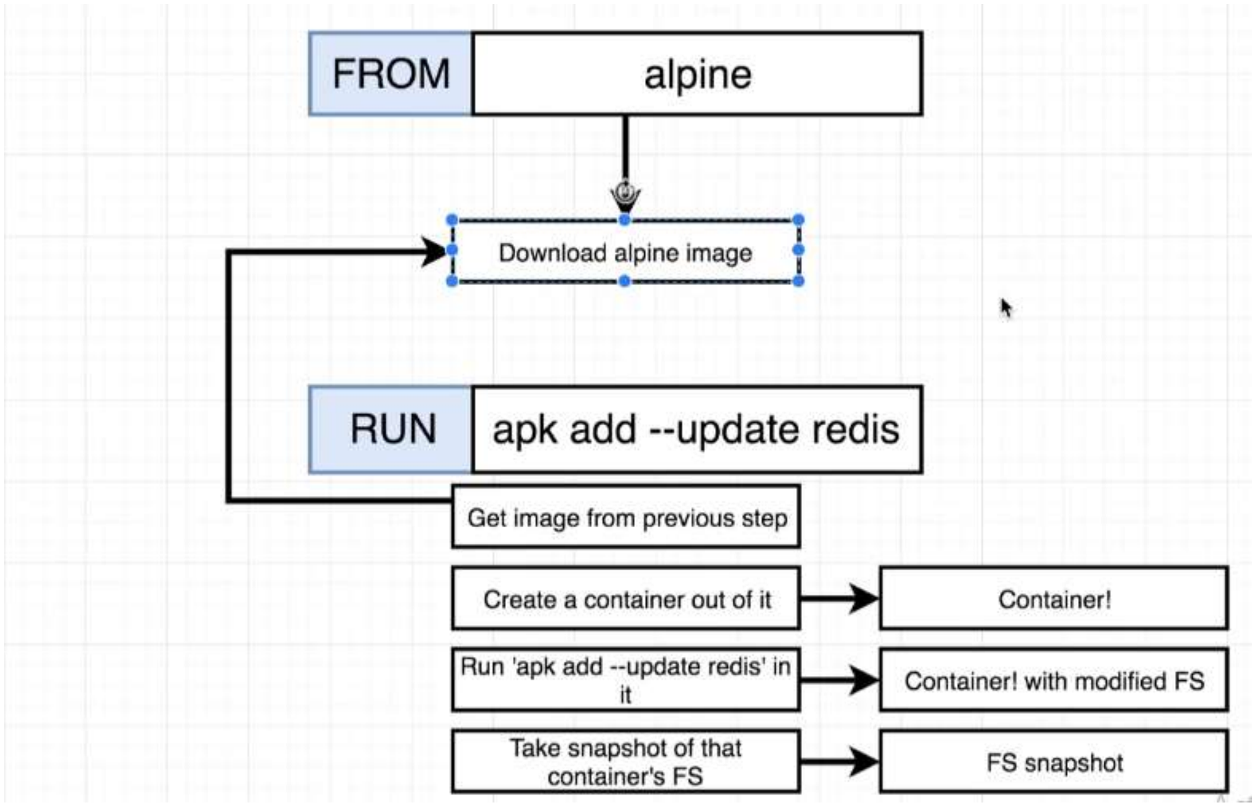
$ Docker build .

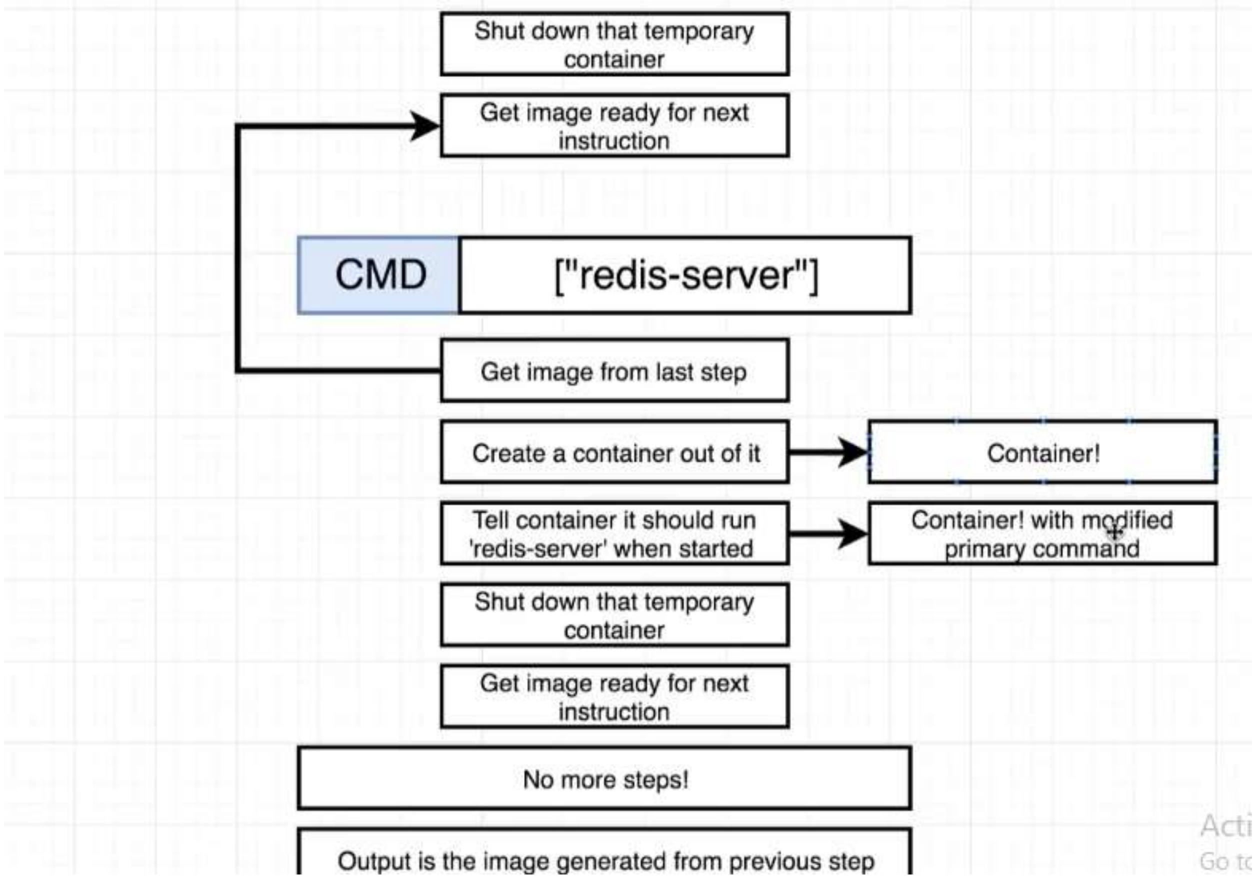
Where . is the context of the folder in which the command is run. Below is how docker goes through different processes to build a final image



```
PS D:\LEARNING\docker\redis-image> docker build .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM alpine
 ---> c059bfaa849c
Step 2/3 : RUN apk add --update redis
 ---> Running in 4b2edc665829
fetch https://dl-cdn.alpinelinux.org/alpine/v3.15/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.15/community/x86_64/APKINDEX.tar.gz
(1/1) Installing redis (6.2.6-r0)
Executing redis-6.2.6-r0.pre-install
Executing redis-6.2.6-r0.post-install
Executing busybox-1.34.1-r3.trigger
OK: 8 MiB in 15 packages
Removing intermediate container 4b2edc665829
 ---> f72f8780079a
Step 3/3 : CMD ["redis-server"]
 ---> Running in 835b6c990aa8
Removing intermediate container 835b6c990aa8
 ---> c50d5fd7e8d5
Successfully built c50d5fd7e8d5
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and directories added to build context will have '
-rwxr-xr-x' permissions. It is recommended to double check and reset permissions for sensitive files and directories.

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
PS D:\LEARNING\docker\redis-image>
```

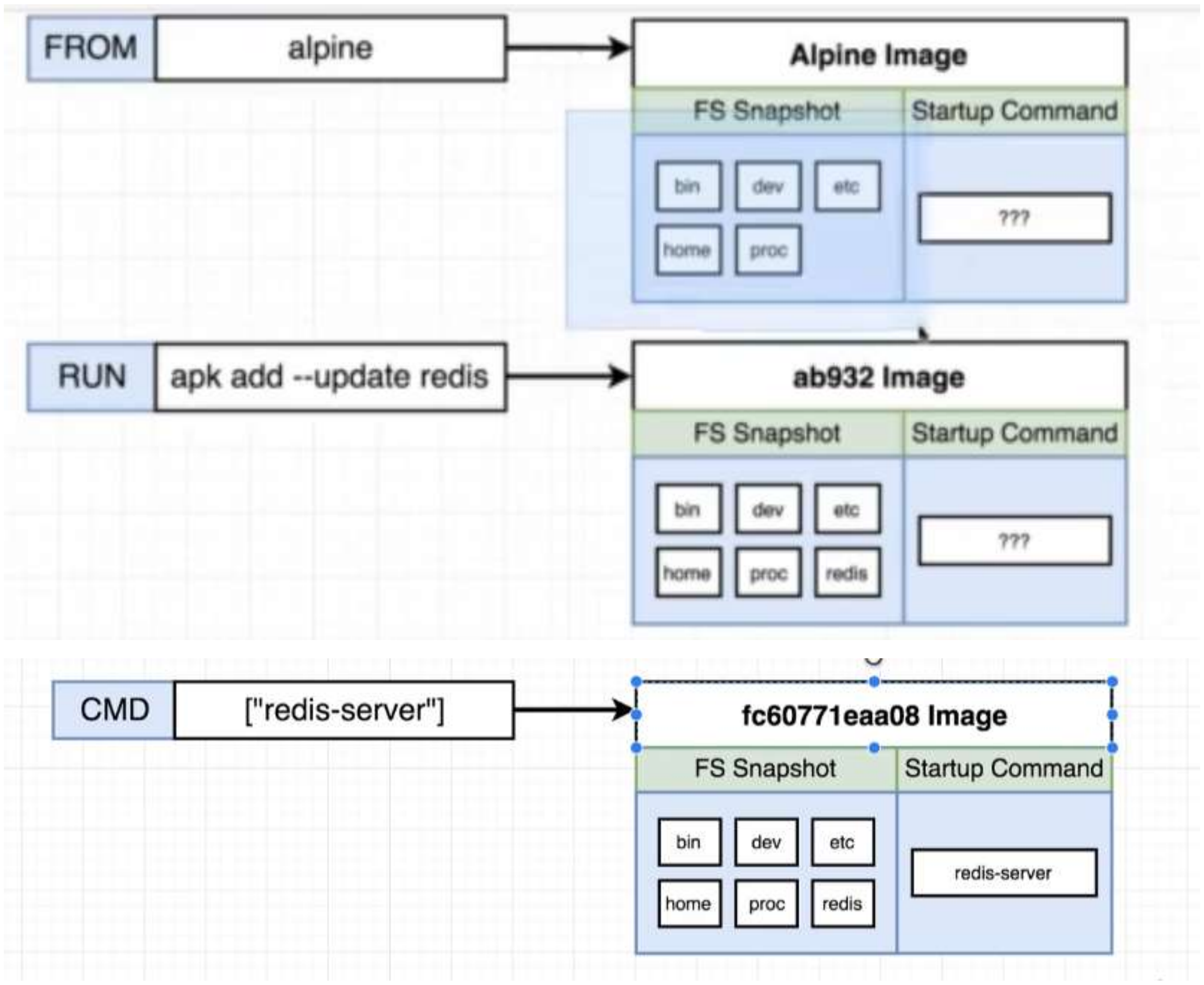
From above diagram one can note that for each step an image is created with ID, that image is used to create a temporary in the next step, that temporary container is removed immediately before the step completes, and final image is eventually created when there are no more steps to execute. The figure below shows step by step how image is created from a temporary container up to last step where final image is created.


FROM
alpine
Download alpine image
RUN
apk add --update redis
Get image from previous step
Create a container out of it
Container!
Run 'apk add --update redis' in it
Container! with modified FS
Take snapshot of that container's FS
FS snapshot


Shut down that temporary container
Get image ready for next instruction
CMD
["redis-server"]
Get image from last step
Create a container out of it
Container!
Tell container it should run 'redis-server' when started
Container! with modified primary command
Shut down that temporary container
Get image ready for next instruction
No more steps!
Output is the image generated from previous step

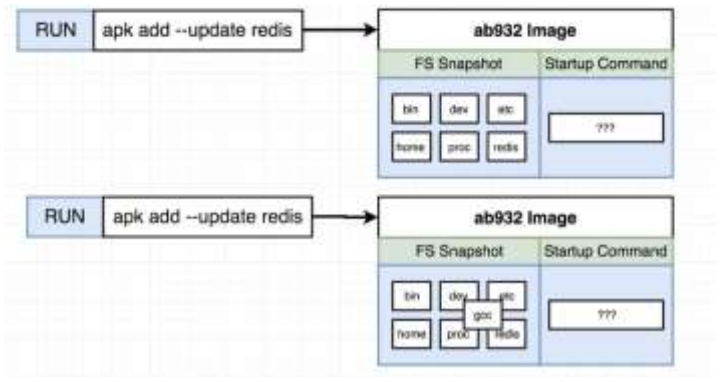
Rebilds and Caching

As we have seen from each step we have an image to start with and startup command to run while creating a temporary container as figure below summarizes it.

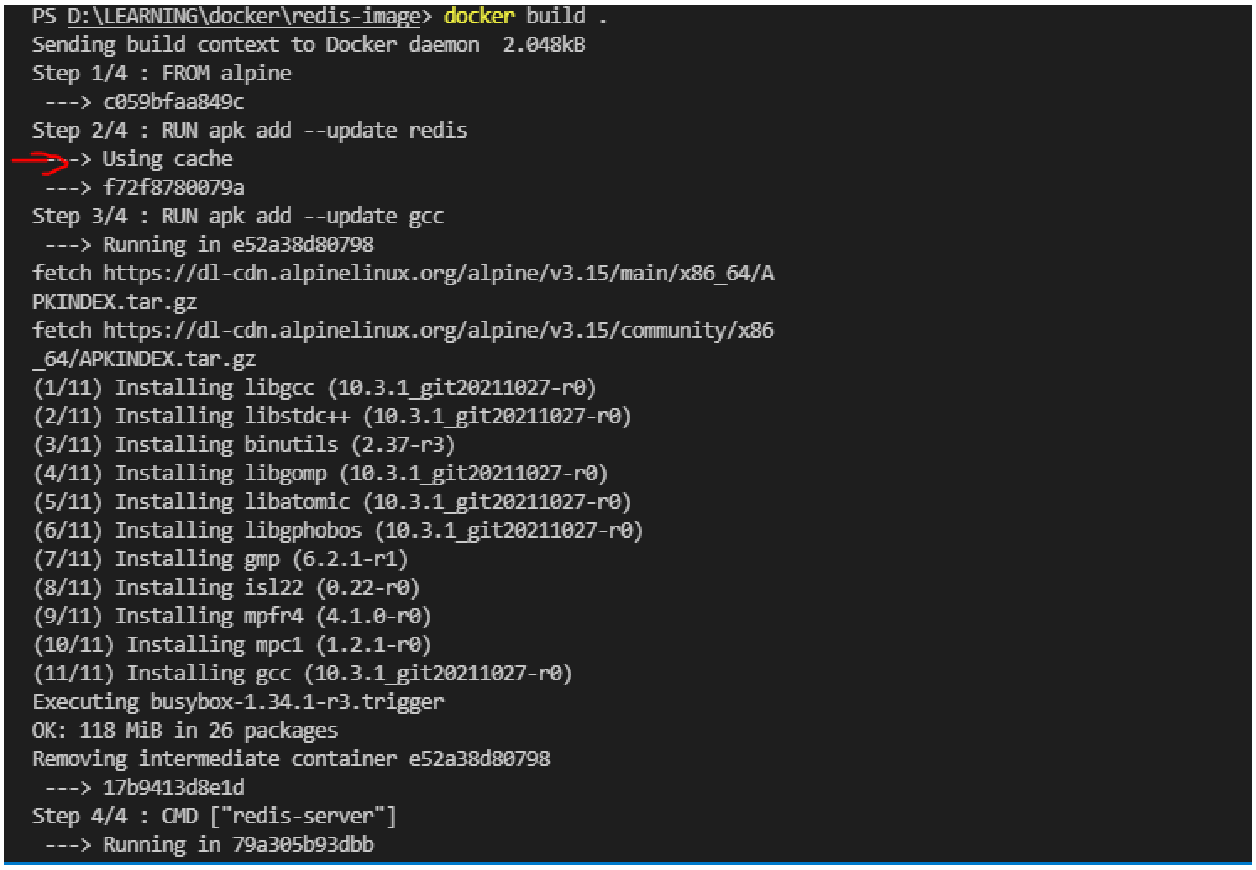


What do you think will happen if we add another run command say RUN apk add --- gcc

We are very sure that it will add another step with gcc added in the FS of the container as follows

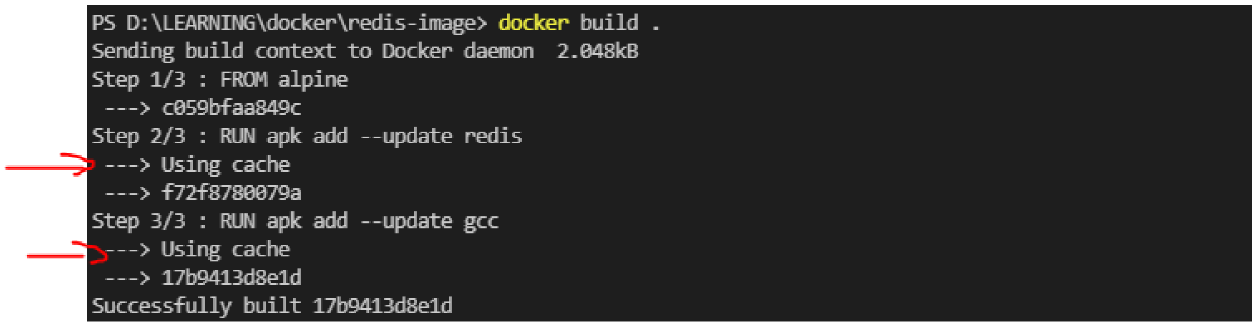


Now if we run docker build . command again something we'll see the trick docker uses to process very fast called caching



```
PS D:\LEARNING\docker\redis-image> docker build .
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM alpine
 ---> c059bfaa849c
Step 2/4 : RUN apk add --update redis
 ---> Using cache
 ---> f72f8780079a
Step 3/4 : RUN apk add --update gcc
 ---> Running in e52a38d80798
fetch https://dl-cdn.alpinelinux.org/alpine/v3.15/main/x86_64/A
PKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.15/community/x86
_64/APKINDEX.tar.gz
(1/11) Installing libgcc (10.3.1_git20211027-r0)
(2/11) Installing libstdc++ (10.3.1_git20211027-r0)
(3/11) Installing binutils (2.37-r3)
(4/11) Installing libgomp (10.3.1_git20211027-r0)
(5/11) Installing libatomic (10.3.1_git20211027-r0)
(6/11) Installing libgphobos (10.3.1_git20211027-r0)
(7/11) Installing gmp (6.2.1-r1)
(8/11) Installing isl22 (0.22-r0)
(9/11) Installing mpfr4 (4.1.0-r0)
(10/11) Installing mpc1 (1.2.1-r0)
(11/11) Installing gcc (10.3.1_git20211027-r0)
Executing busybox-1.34.1-r3.trigger
OK: 118 MiB in 26 packages
Removing intermediate container e52a38d80798
 ---> 17b9413d8e1d
Step 4/4 : CMD ["redis-server"]
 ---> Running in 79a305b93dbb
```

Both alpine and redis are not downloaded neither temporary container created again instead a cache is used but when it reaches step 3, gcc is not found so it is downloaded and process of creating temporary container comes into play again but if we run docker build . for the second time even gcc will be found in cache and thus nothing will be downloaded nor temporary containers be created.

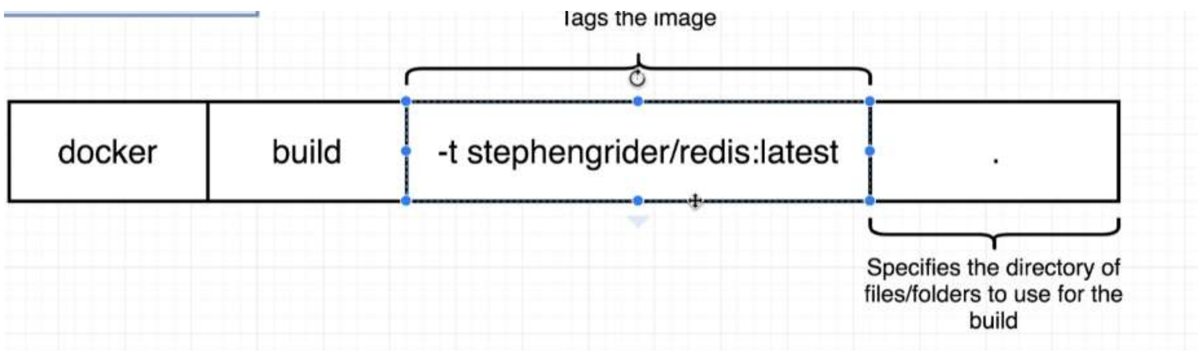


```
PS D:\LEARNING\docker\redis-image> docker build .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM alpine
 ---> c059bfaa849c
Step 2/3 : RUN apk add --update redis
 ---> Using cache
 ---> f72f8780079a
Step 3/3 : RUN apk add --update gcc
 ---> Using cache
 ---> 17b9413d8e1d
Successfully built 17b9413d8e1d
```

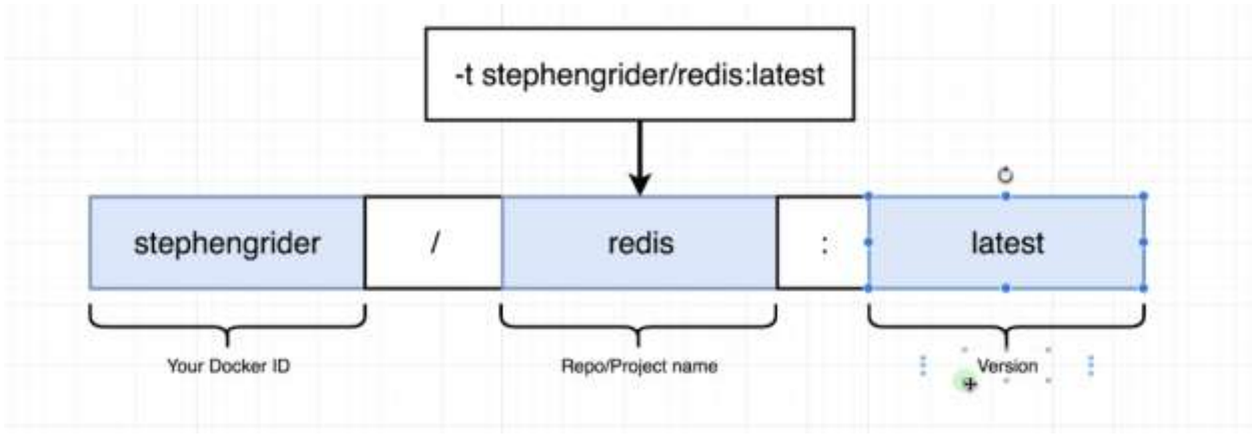
However, if you change the order for example put gcc first, downloading and temporary containers creation will take place since order of execution has changed so cache cant serve you.

Tagging docker image

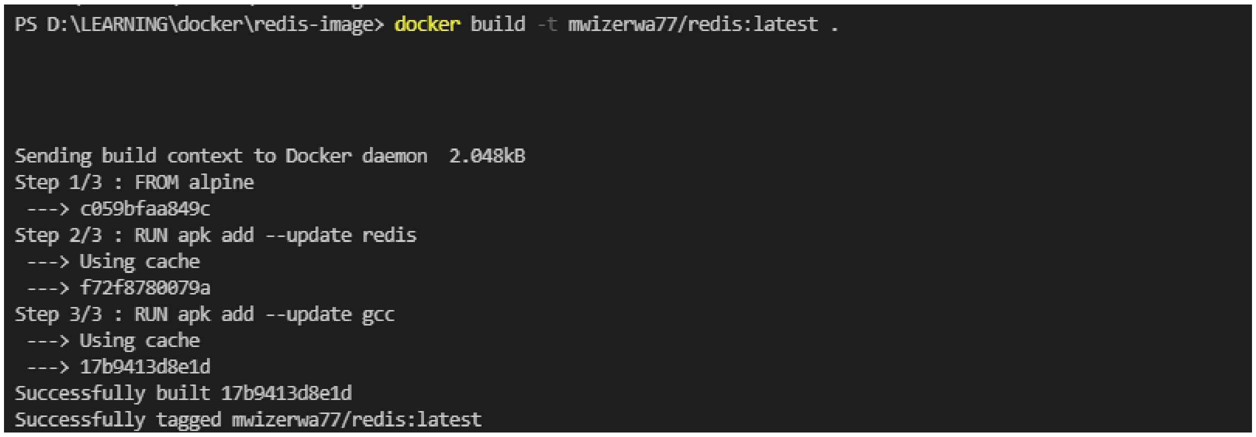
If we want to run our image we shall have to use its image ID say docker run 17b9413d8e1 but this is kind of boring since we are used to run images using friendly names like docker run helloword to fix that issue we've to tweak the command to build docker image and use tag. The syntax is as shown below


Tags the image
docker
build
-t stephengrider/redis:latest
.
Specifies the directory of files/folders to use for the build

The convention for the tag is as follows



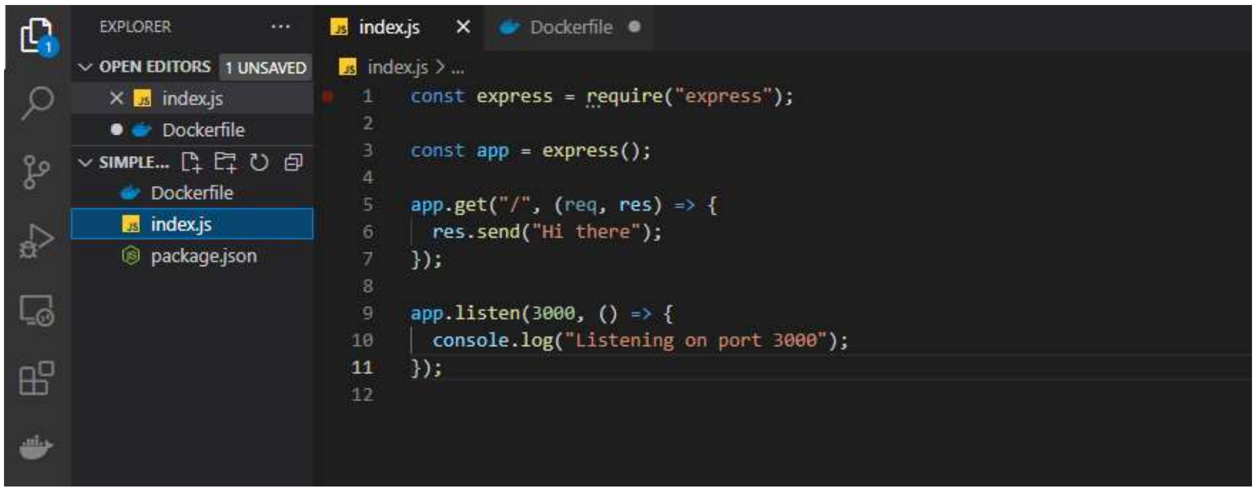
You might wonder why other images were simple like busybox,redis, etc, and now yours is somehow lengthy and complicated, well all those are community images, they were written by community users and were authorized to be open source but if you build your own images they have to start with your docker id/username. Examine the example below



```
PS D:\LEARNING\docker\redis-image> docker build -t mwizerwa77/redis:latest .



Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM alpine
 ---> c059bfaa849c
Step 2/3 : RUN apk add --update redis
 ---> Using cache
 ---> f72f8780079a
Step 3/3 : RUN apk add --update gcc
 ---> Using cache
 ---> 17b9413d8e1d
Successfully built 17b9413d8e1d
Successfully tagged mwizerwa77/redis:latest
```

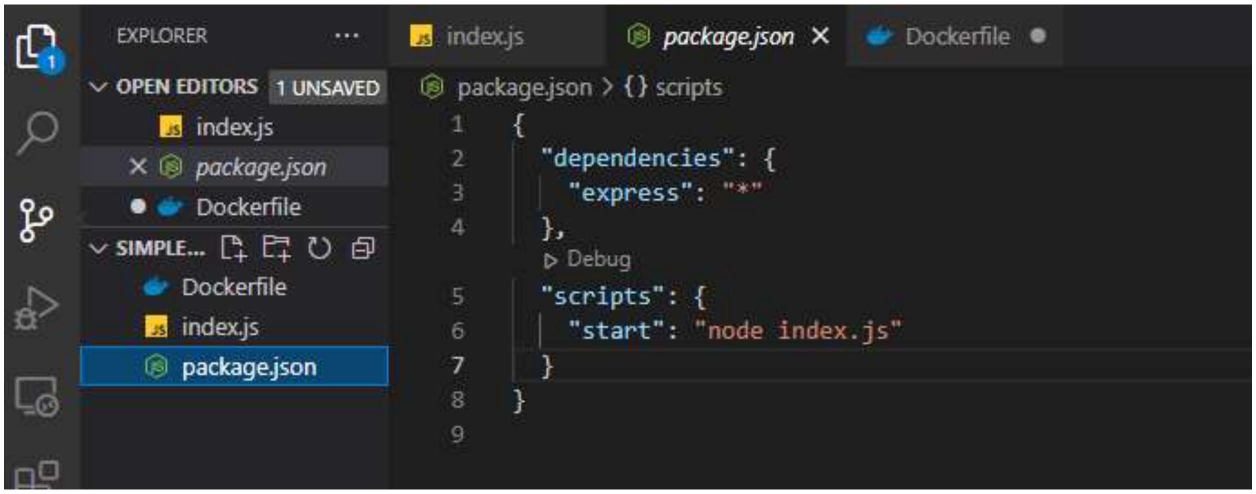
Working project

Let us create a nodejs project with only package.json and index.js using express js as vscode screen show below shows


EXPLORER
OPEN EDITORS 1 UNSAVED
index.js
Dockerfile
SIMPLE...
Dockerfile
index.js
package.json

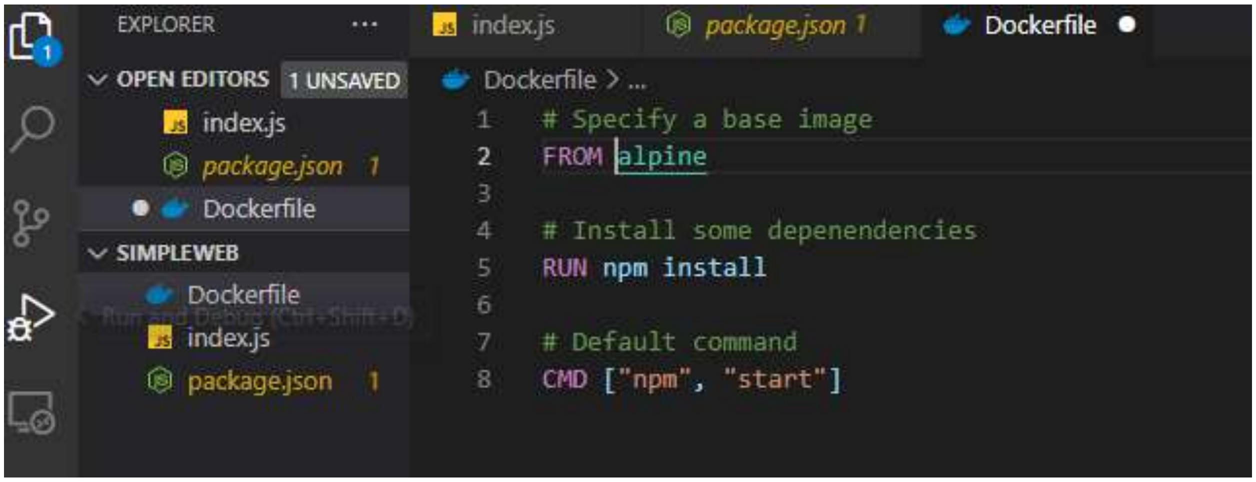
```
const express = require("express");

const app = express();

app.get("/", (req, res) => {
  res.send("Hi there");
});

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

Index.js file


EXPLORER
OPEN EDITORS 1 UNSAVED
index.js
package.json
Dockerfile
SIMPLE...
Dockerfile
index.js
package.json

```
{
  "dependencies": {
    "express": "*"
  },
  "scripts": {
    "start": "node index.js"
  }
}
```

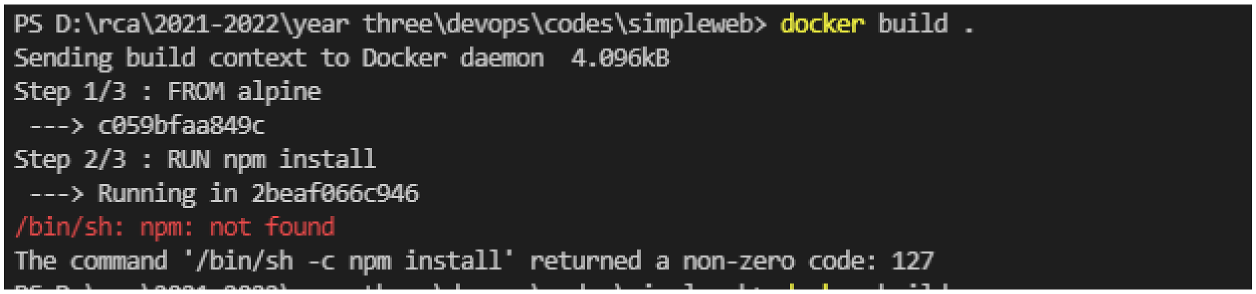
Package.json file


EXPLORER
OPEN EDITORS 1 UNSAVED
index.js
package.json 1
Dockerfile
SIMPLEWEB
Dockerfile
index.js
package.json 1

```
# Specify a base image
FROM alpine

# Install some depenendencies
RUN npm install

# Default command
CMD ["npm", "start"]
```

Dockerfile

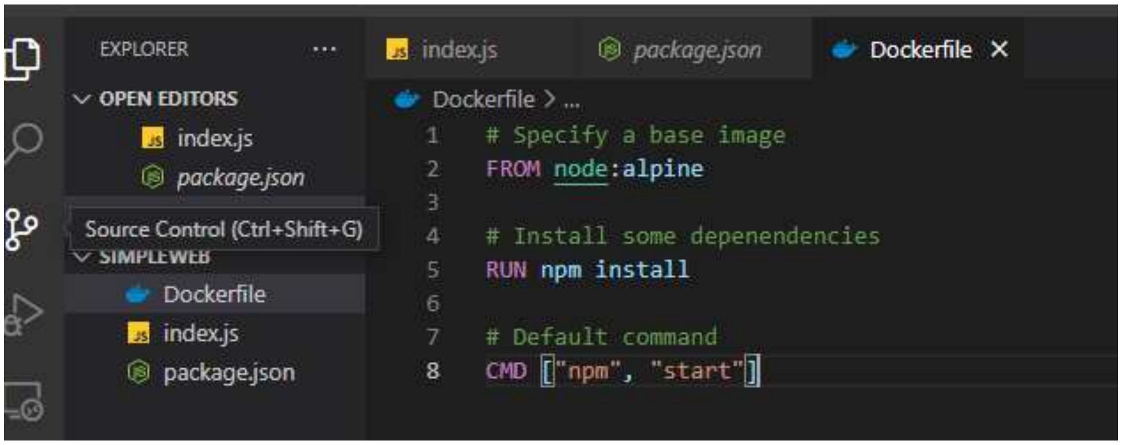
Now run docker build . to build an image



```
PS D:\rca\2021-2022\year three\devops\codes\simpleweb> docker build .
Sending build context to Docker daemon  4.096kB
Step 1/3 : FROM alpine
 ---> c059bfaa849c
Step 2/3 : RUN npm install
 ---> Running in 2beaf066c946
/bin/sh: npm: not found
The command '/bin/sh -c npm install' returned a non-zero code: 127
```

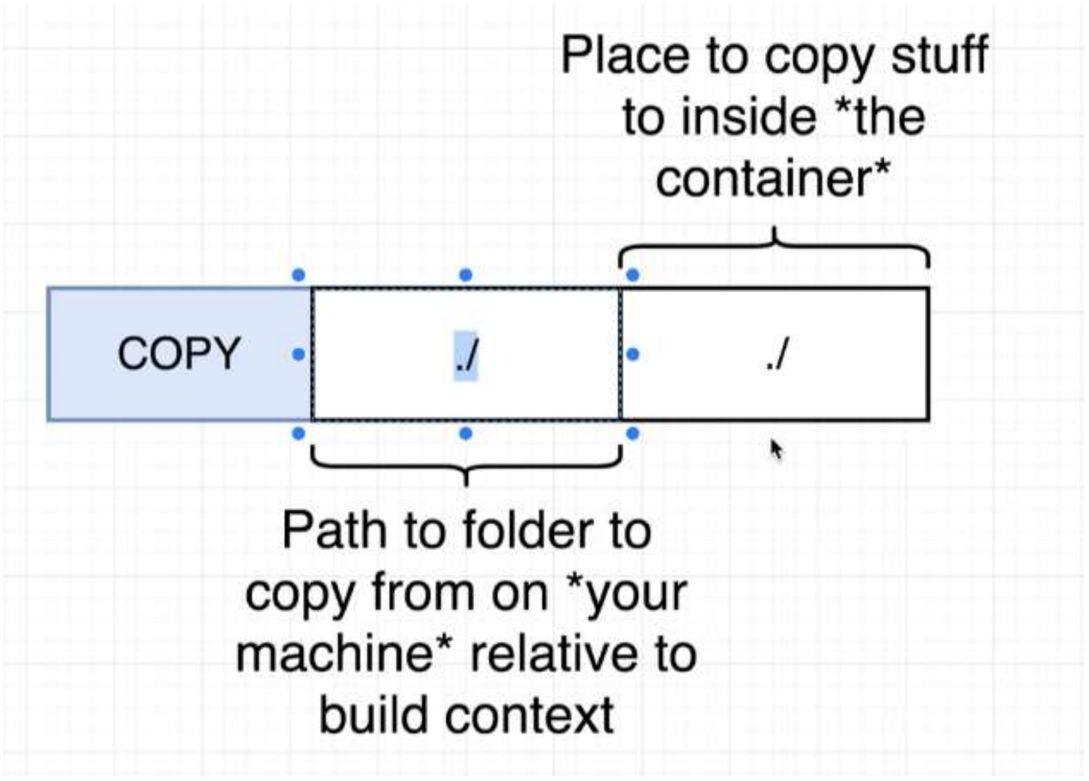
The project wont run with alpine only because node js is installed out of the box, alpine has no additional applications installed so you have two options

- Include step to install nodejs image as well in docker file
- Or use abase image that has node already, and we recommending this option therefore we can go to docker hub and search for node in the repositories, then click on tags to search for alpine tag

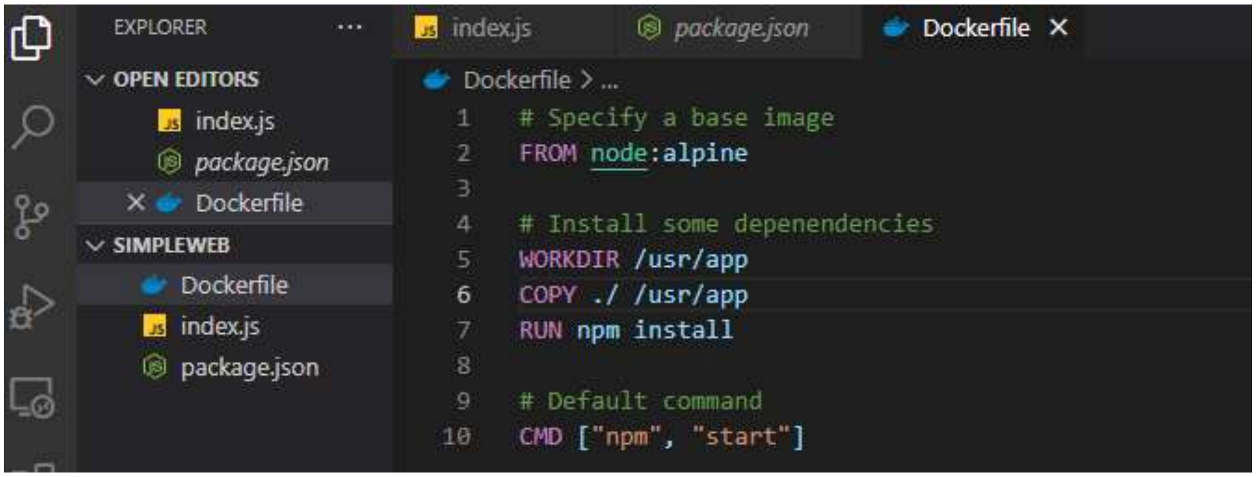


```
# Specify a base image
FROM node:alpine

# Install some depenendencies
RUN npm install

# Default command
CMD ["npm", "start"]
```

But still the project wont run as package.json file is not found in our temporary containers, we need to include a step to copy it to our temporary container so that it can be included in built image. We use below syntax to copy file from our local folders/FS to container's FS

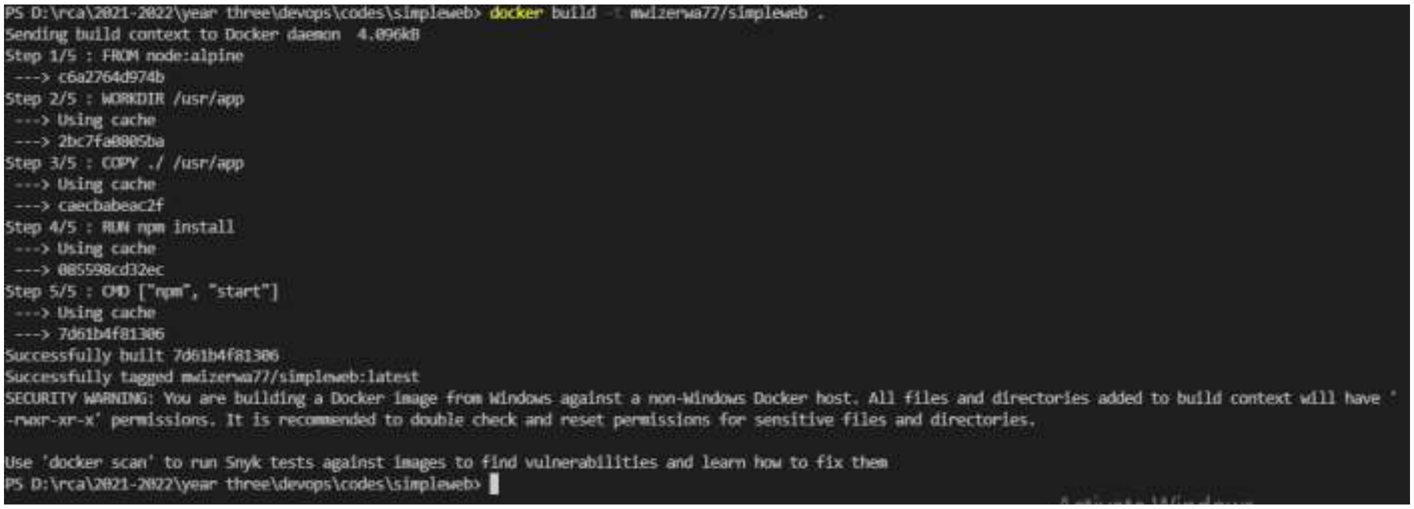

Place to copy stuff to inside *the container*
COPY
./
./
Path to folder to copy from on *your machine* relative to build context

Therefore the final Dockerfile looks like this:



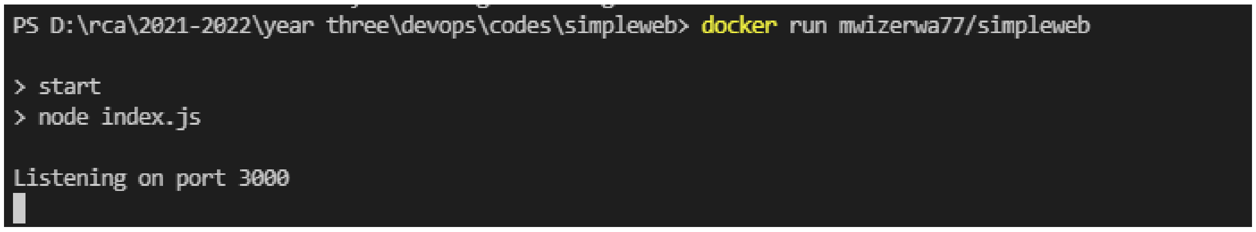
```
# Specify a base image
FROM node:alpine

# Install some depenendencies
WORKDIR /usr/app
COPY ./ /usr/app
RUN npm install

# Default command
CMD ["npm", "start"]
```

Build image again



```
PS D:\rca\2021-2022\year three\devops\codes\simpleweb> docker build -t mwizerwa77/simpleweb .
Sending build context to Docker daemon  4.096kB
Step 1/5 : FROM node:alpine
 ---> c6a2764d974b
Step 2/5 : WORKDIR /usr/app
 ---> Using cache
 ---> 2bc7fa0805ba
Step 3/5 : COPY ./ /usr/app
 ---> Using cache
 ---> caecbabeac2f
Step 4/5 : RUN npm install
 ---> Using cache
 ---> 085598cd32ec
Step 5/5 : CMD ["npm", "start"]
 ---> Using cache
 ---> 7d61b4f81306
Successfully built 7d61b4f81306
Successfully tagged mwizerwa77/simpleweb:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and directories added to build context will have '
-rwxr-xr-x' permissions. It is recommended to double check and reset permissions for sensitive files and directories.

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
PS D:\rca\2021-2022\year three\devops\codes\simpleweb>
```

Now run the docker image

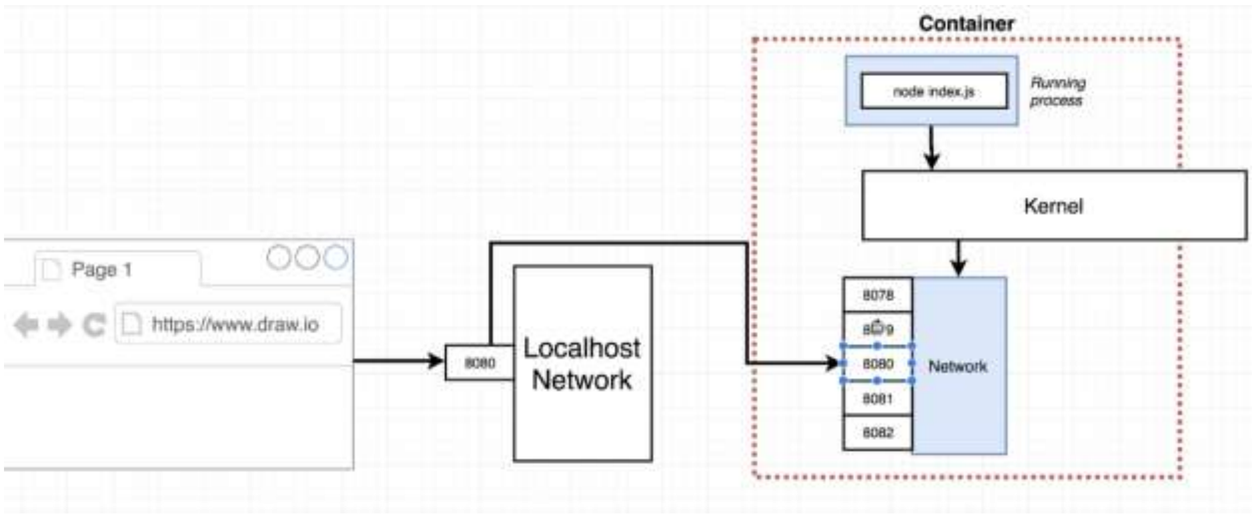


```
PS D:\rca\2021-2022\year three\devops\codes\simpleweb> docker run mwizerwa77/simpleweb

> start
> node index.js

Listening on port 3000
```

Our application running on port 3000 but you cant browse it since this is a container port. How can we access our app then?

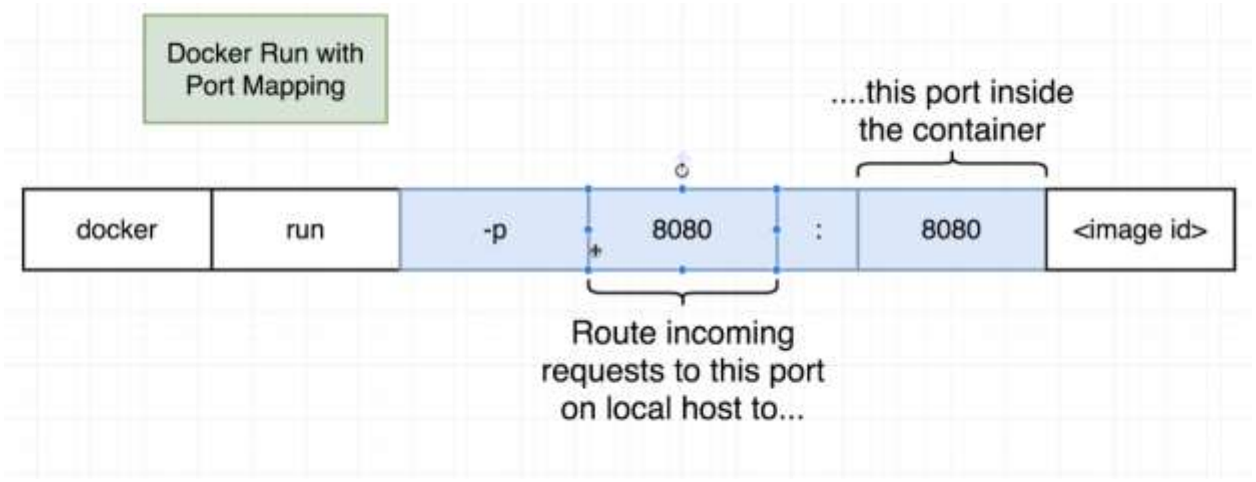
Container port mapping

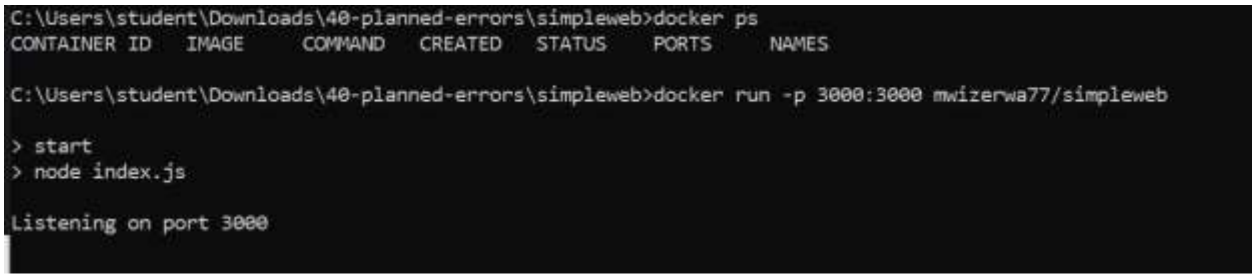
Port mapping is where we map any port on the local machine to container port(s)



Note: we do this for incoming request since container has no limitation to outgoing requests as we have seen container is capable of reaching internet while running npm install and other commands that downloaded libraries from internet. It is only incoming that has limitations

The syntax to use while mapping ports is as follows


Docker Run with Port Mapping
....this port inside the container
docker
run
-p
8080
:
8080
<image id>
Route incoming requests to this port on local host to...



```
C:\Users\student\Downloads\40-planned-errors\simpleweb>docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES

C:\Users\student\Downloads\40-planned-errors\simpleweb>docker run -p 3000:3000 mwizerwa77/simpleweb

> start
> node index.js

Listening on port 3000
```

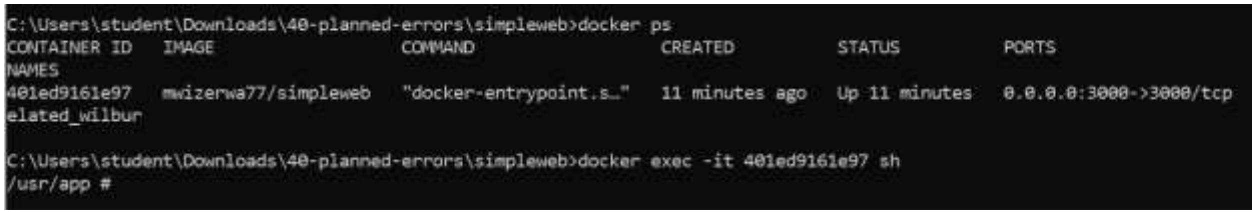
If you want to ssh into the container you can easily run

Docker run –it mwizerwa77/simpleweb sh

Or

$ Docker ps

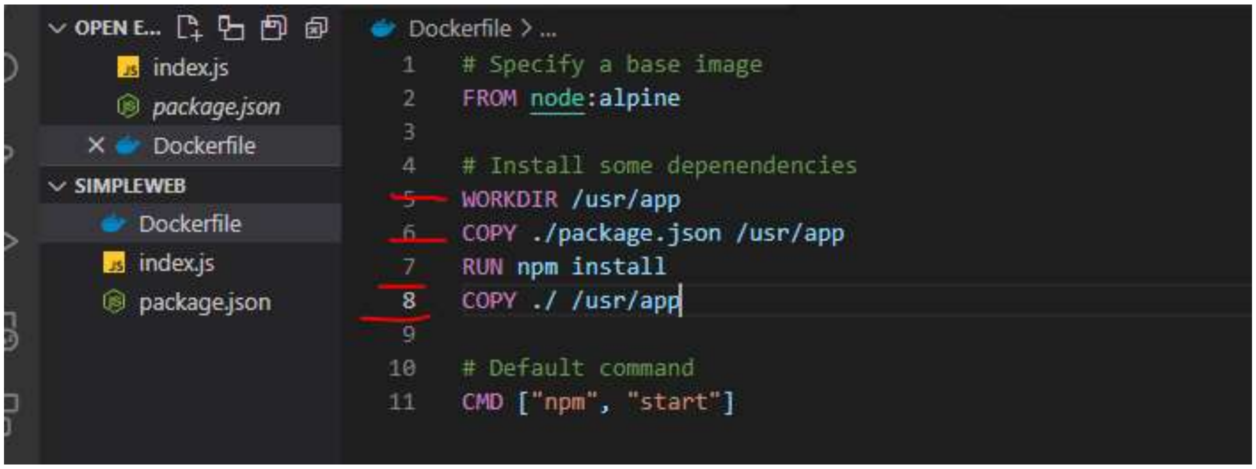
$ Docker exec –it <image> sh



```
C:\Users\student\Downloads\40-planned-errors\simpleweb>docker ps
CONTAINER ID   IMAGE                 COMMAND                  CREATED          STATUS          PORTS
NAMES
401ed9161e97   mwizerwa77/simpleweb  "docker-entrypoint.s…"   11 minutes ago   Up 11 minutes   0.0.0.0:3000->3000/tcp
elated_wilbur

C:\Users\student\Downloads\40-planned-errors\simpleweb>docker exec -it 401ed9161e97 sh
/usr/app #
```

Minimizing cache bustings and rebuilds

If you make a change in index.js, and build the image, npm will install all packages again. This is annoying as for big projects installing npm packages can take longer so to avoid such we can change our dockerfile as follows



```
# Specify a base image
FROM node:alpine

# Install some depenendencies
WORKDIR /usr/app
COPY ./package.json /usr/app
RUN npm install
COPY ./ /usr/app

# Default command
CMD ["npm", "start"]
```

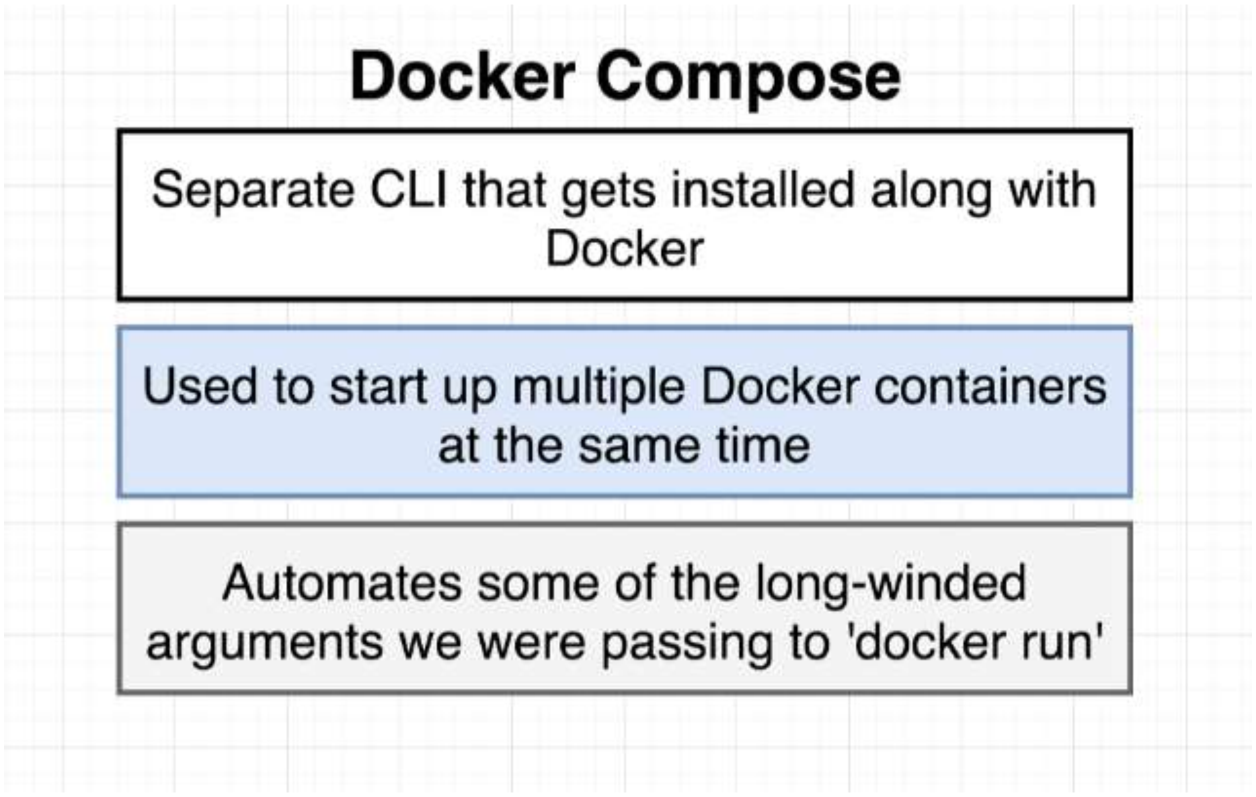
CHAPTER TWO: Docker compose (multiple containers)

Introduction

Docker Compose is a Docker tool used to define and run multi-container applications. With Compose, you use a YAML file to configure your application's services and create all the app's services from that configuration.

Think of docker-compose as an **automated multi-container workflow.** Compose is an excellent tool for development, testing, CI workflows, and staging environments. According to the Docker documentation, the most popular features of Docker Compose are:

- Multiple isolated environments on a single host
- Preserve volume data when containers are created
- Only recreate containers that have changed
- Variables and moving a composition between environments
- Orchestrate multiple containers that work together



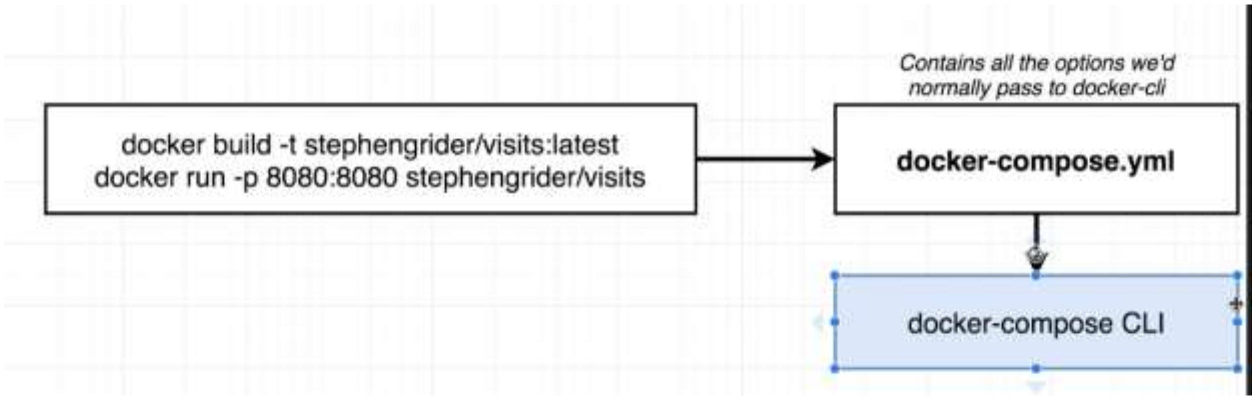
Docker Compose

Separate CLI that gets installed along with Docker

Used to start up multiple Docker containers at the same time

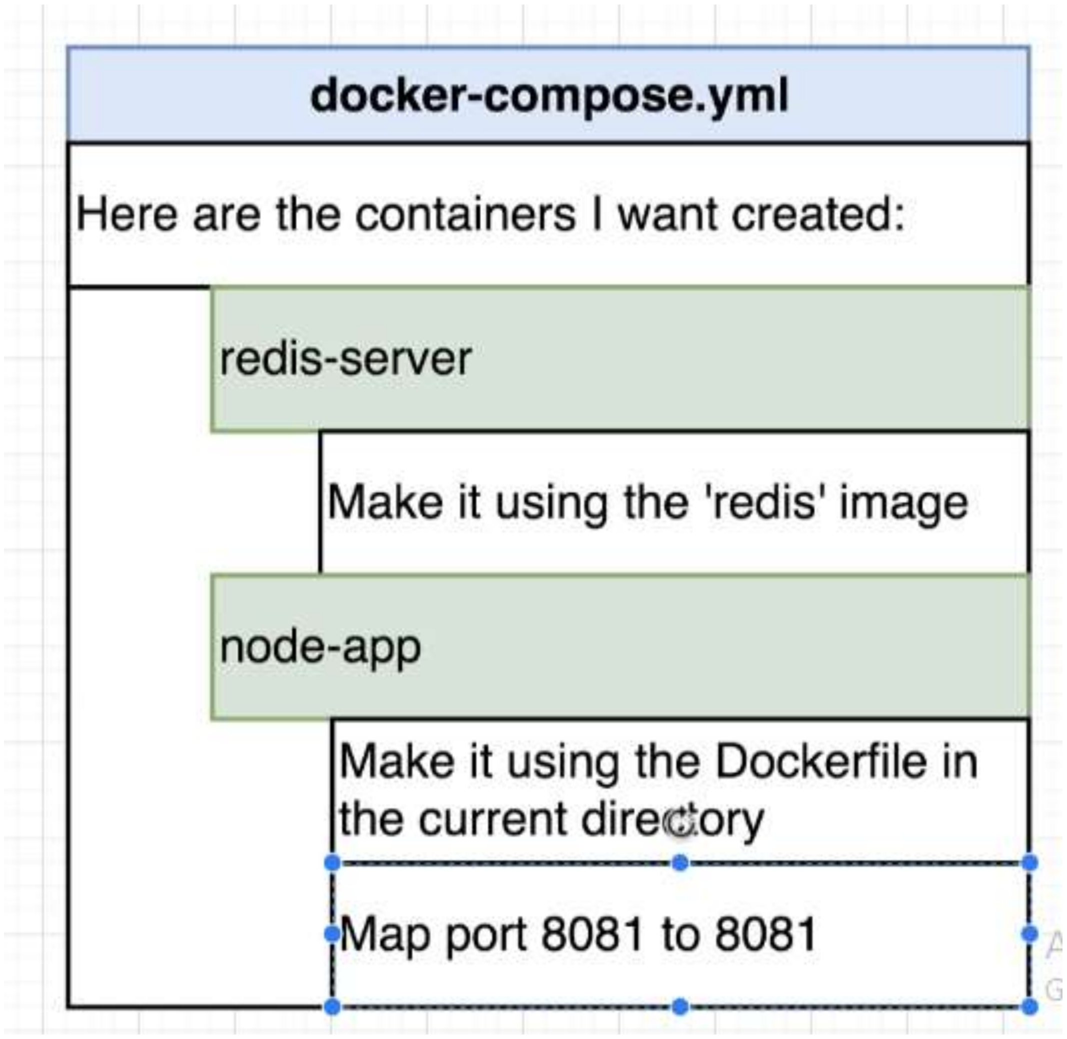
Automates some of the long-winded arguments we were passing to 'docker run'

Suppose you have an nodeApp and a redis database image, instead of running two separate docker images, we use docker compose to run the two images at the same time and allow them to share networking resources. YAML file follows a special syntax.



docker build -t stephengrider/visits:latest
docker run -p 8080:8080 stephengrider/visits
Contains all the options we'd normally pass to docker-cli
docker-compose.yml
docker-compose CLI

Docker compose file can be very complicated depending on number of containers you are binding together, considering above scenario of a node app running a redis database, below can be a step by step guide for creating its docker compose file

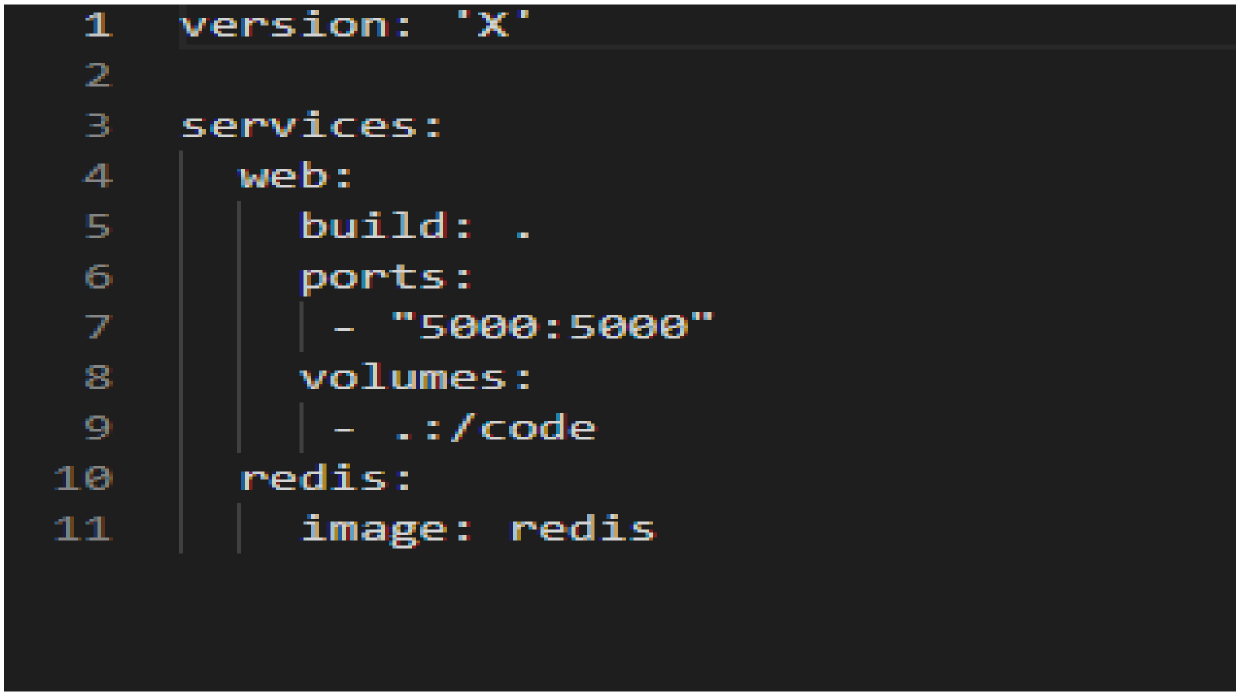

docker-compose.yml
Here are the containers I want created:
redis-server
Make it using the 'redis' image
node-app
Make it using the Dockerfile in the current directory
Map port 8081 to 8081

The docker-compose will run two images,

1) Redis-server from a redis image
2) Node-app from docker file, then map container port 8081 to physical port 8081

Docker Compose files work by applying mutiple commands that are declared within a single docker-compose.yml configuration file.

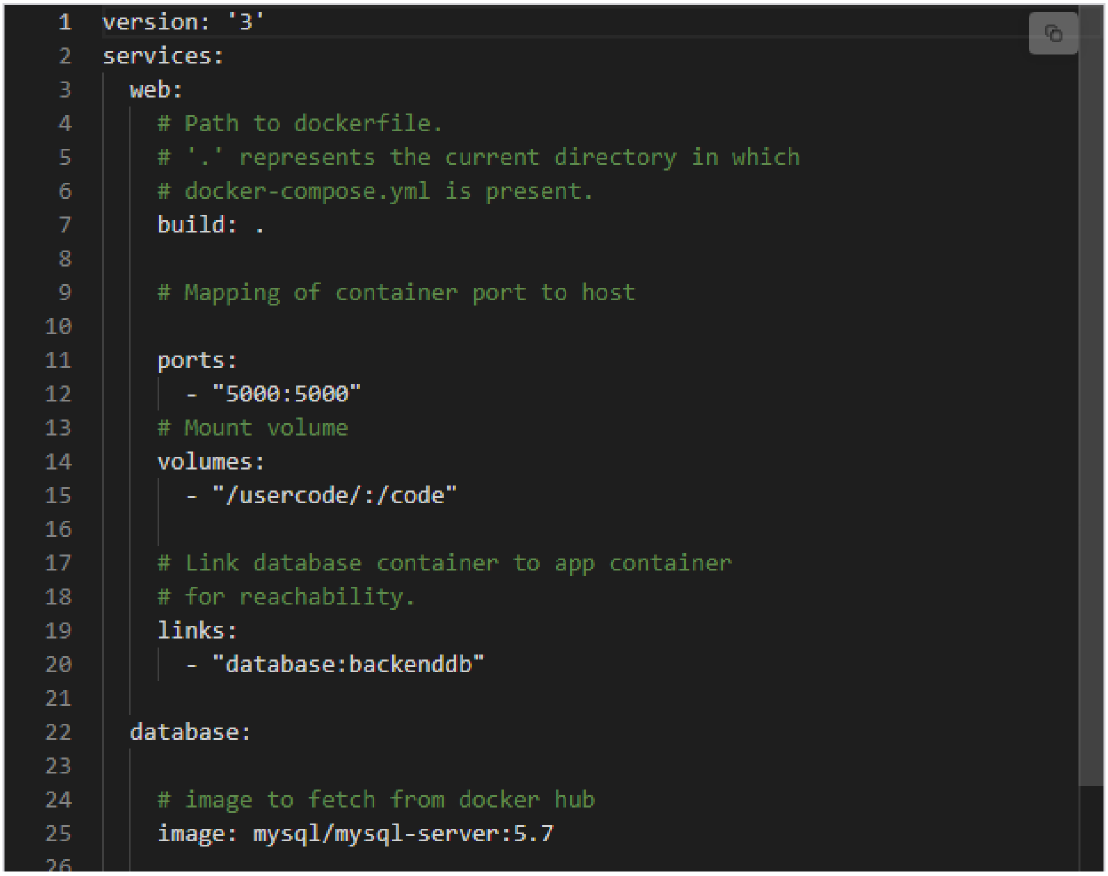
The basic structure of a Docker Compose YAML file looks like this:



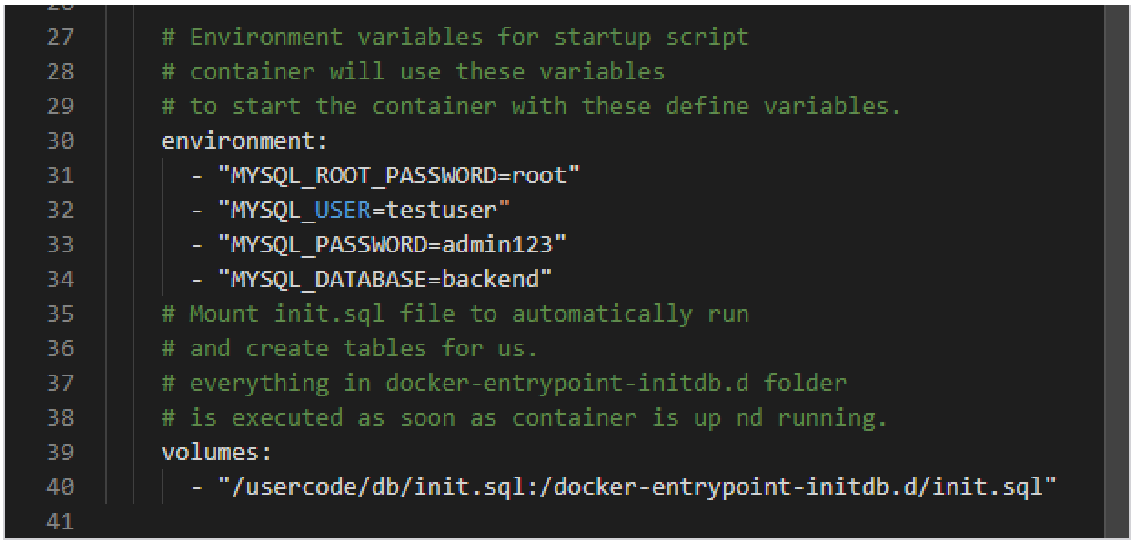
```yaml
version: 'X'

services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
  redis:
    image: redis
```

Now, let’s look at real-world example of a Docker Compose file and break it down step-by-step to understand all of this better. Note that all the clauses and keywords in this example are commonly used keywords and industry standard.

With just these, you can start a development workflow. There are some more advanced keywords that you can use in production, but for now, let’s just get started with the necessary clauses.



```
version: '3'
services:
  web:
    # Path to dockerfile.
    # '.' represents the current directory in which
    # docker-compose.yml is present.
    build: .

    # Mapping of container port to host

    ports:
      - "5000:5000"
    # Mount volume
    volumes:
      - "/usercode/:/code"

    # Link database container to app container
    # for reachability.
    links:
      - "database:backenddb"

  database:

    # image to fetch from docker hub
    image: mysql/mysql-server:5.7
```



```

    # Environment variables for startup script
    # container will use these variables
    # to start the container with these define variables.
    environment:
      - "MYSQL_ROOT_PASSWORD=root"
      - "MYSQL_USER=testuser"
      - "MYSQL_PASSWORD=admin123"
      - "MYSQL_DATABASE=backend"
    # Mount init.sql file to automatically run
    # and create tables for us.
    # everything in docker-entrypoint-initdb.d folder
    # is executed as soon as container is up nd running.
    volumes:
      - "/usercode/db/init.sql:/docker-entrypoint-initdb.d/init.sql"

```

- **version ‘3’:** This denotes that we are using version 3 of Docker Compose, and Docker will provide the appropriate features. At the time of writing this article, version 3.7 is latest version of Compose. To know the compose version corresponding to docker release you may always refer to docker hub https://docs.docker.com/compose/compose-file/compose-versioning/

- **services:** This section defines all the different containers we will create. In our example, we have two services, web and database.
- **web:** This is the name of our node app service. Docker Compose will create containers with the name we provide.
- **build:** This specifies the location of our Dockerfile, and . represents the directory where the docker-compose.yml file is located.
- **ports:** This is used to map the container's ports to the host machine.
- **volumes:** This is just like the -v option for mounting disks in Docker. In this example, we attach our code files directory to the containers' ./code directory. This way, we won't have to rebuild the images if changes are made.
- **links:** This will link one service to another. For the bridge network, we must specify which container should be accessible to which container using links.
- **image:** If we don't have a Dockerfile and want to run a service using a pre-built image, we specify the image location using the image clause. Compose will fork a container from that image.
- **environment:** The clause allows us to set up an environment variable in the container. This is the same as the -e argument in Docker when running a container.

Some common Docker Compose commands

Now that we know how to create a docker-compose file, let's go over the most common Docker Compose commands that we can use with our files. Keep in mind that we will only be discussing the most frequently-used commands.

docker-compose: Every Compose command starts with this command. You can also use docker-compose <command> --help to provide additional information about arguments and implementation details.

```
$ docker-compose --help
Define and run multi-container applications with Docker.
```

docker-compose build: This command builds images in the docker-compose.yml file. The job of the build command is to get the images ready to create containers, so if a service is using the prebuilt image, it will skip this service.

```
$ docker-compose build
database uses an image, skipping
Building web
Step 1/11 : FROM python:3.9-rc-buster
 ---> 2e0edf7d3a8a
Step 2/11 : RUN apt-get update && apt-get install -y docker.io
```

docker-compose images: This command will list the images you've built using the current docker-compose file.

```
$ docker-compose images
          Container                  Repository        Tag       Image Id       Size
--------------------------------------------------------------------------------------
7001788f31a9_docker_database_1   mysql/mysql-server   5.7      2a6c84ecfcb2   333.9 MB
```

```
docker_database_1               mysql/mysql-server   5.7      2a6c84ecfcb2   333.9 MB
docker_web_1                   <none>               <none>   d986d824dae4   953 MB
```

docker-compose stop: This command stops the running containers of specified services.

```
$ docker-compose stop
Stopping docker_web_1      ... done
Stopping docker_database_1 ... done
```

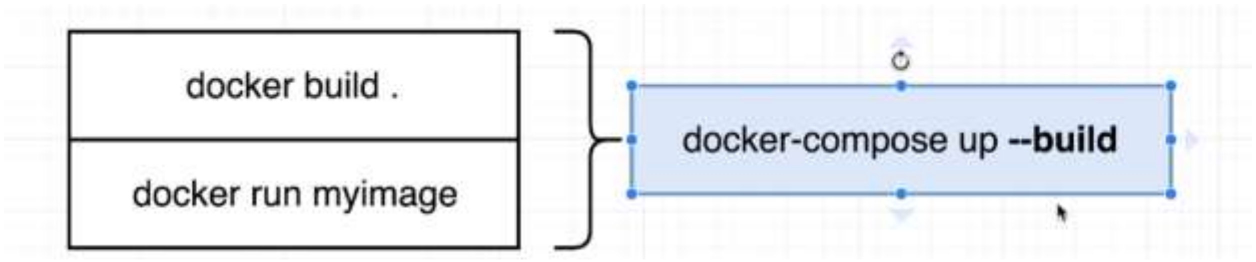
docker-compose run: This is similar to the docker run command. It will create containers from images built for the services mentioned in the compose file.

```
$ docker-compose run web
Starting 7001788f31a9_docker_database_1 ... done
 * Serving Flask app "app.py" (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 116-917-688
```

docker-compose up: This command does the work of the docker-compose build and docker-compose run commands. It builds the images if they are not located locally and starts the containers. If images are already built, it will fork the container directly.

```
$ docker-compose up
Creating docker_database_1 ... done
Creating docker_web_1      ... done
Attaching to docker_database_1, docker_web_1
```

Incase you want to force a rebuild you can use up –build



docker-compose ps: This command list all the containers in the current docker-compose file. They can then either be running or stopped.

```
$ docker-compose ps
    Name              Command          State            Ports
--------------------------------------------------------------------------------
docker_database_1   /entrypoint.sh mysqld   Up (healthy)   3306/tcp, 33060/tcp
docker_web_1        flask run               Up             0.0.0.0:5000->5000/tcp

$ docker-compose ps
    Name              Command          State    Ports
------------------------------------------------------
docker_database_1   /entrypoint.sh mysqld   Exit 0
docker_web_1        flask run               Exit 0
```

docker-compose down: This command is similar to the docker system prune command. However, in Compose, it stops all the services and cleans up the containers, networks, and images.

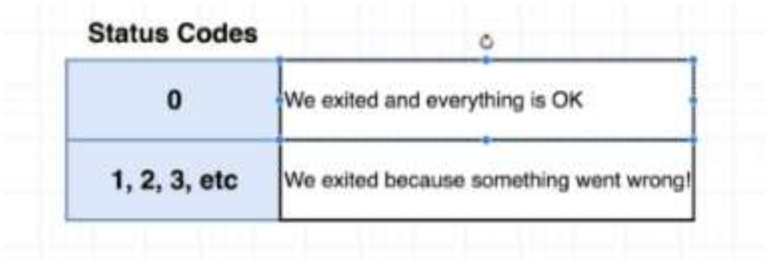
```
$ docker-compose down
Removing docker_web_1      ... done
Removing docker_database_1 ... done
Removing network docker_default
(django-tuts) Venkateshs-MacBook-Air:Docker venkateshachintalwar$ docker-compose images
Container   Repository   Tag   Image Id   Size
----------------------------------------------
(django-tuts) Venkateshs-MacBook-Air:Docker venkateshachintalwar$ docker-compose ps
Name   Command   State   Ports
------------------------------
```

Container maintenance with compose

Let us use process.exit to imitate app crushing, we add

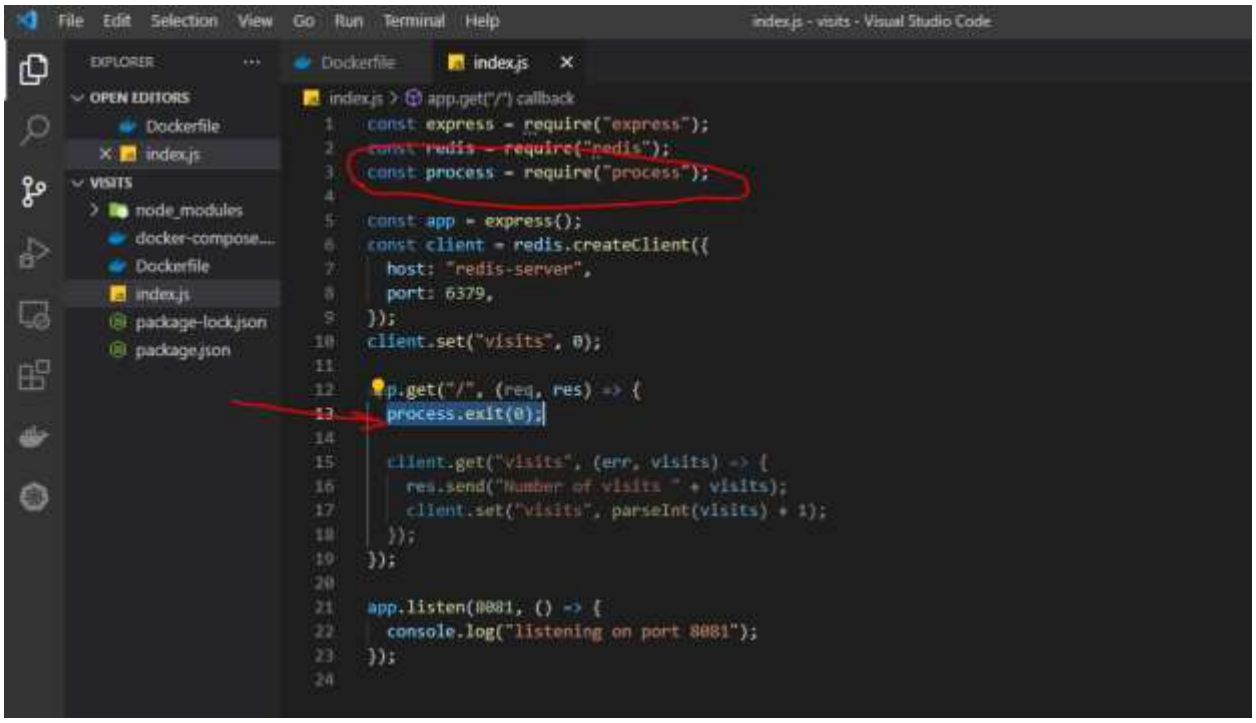
```
process.exit(0);
```

this causes the app to exit intentionally as you know, below diagram reminds us the meaning of different process.exit parameters



Status Codes

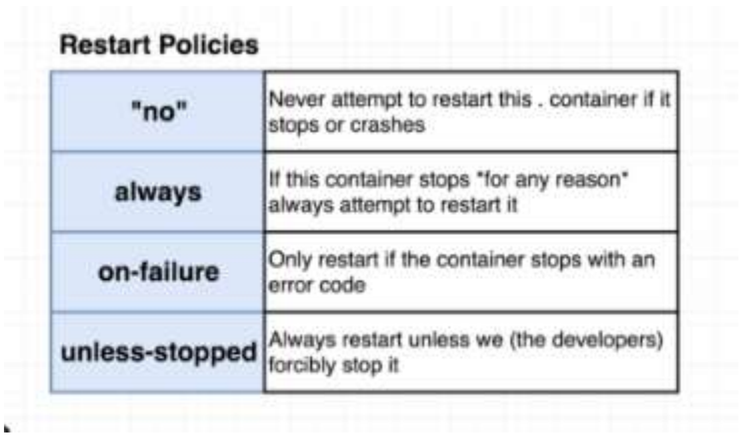
0	We exited and everything is OK
1, 2, 3, etc	We exited because something went wrong!



```
const express = require("express");
const redis = require("redis");
const process = require("process");

const app = express();
const client = redis.createClient({
  host: "redis-server",
  port: 6379,
});
client.set("visits", 0);

app.get("/", (req, res) => {
  process.exit(0);

  client.get("visits", (err, visits) => {
    res.send("Number of visits " + visits);
    client.set("visits", parseInt(visits) + 1);
  });
});

app.listen(8081, () => {
  console.log("listening on port 8081");
});
```

Depending on the exit status/parameter docker can know what to do when the app crushes.

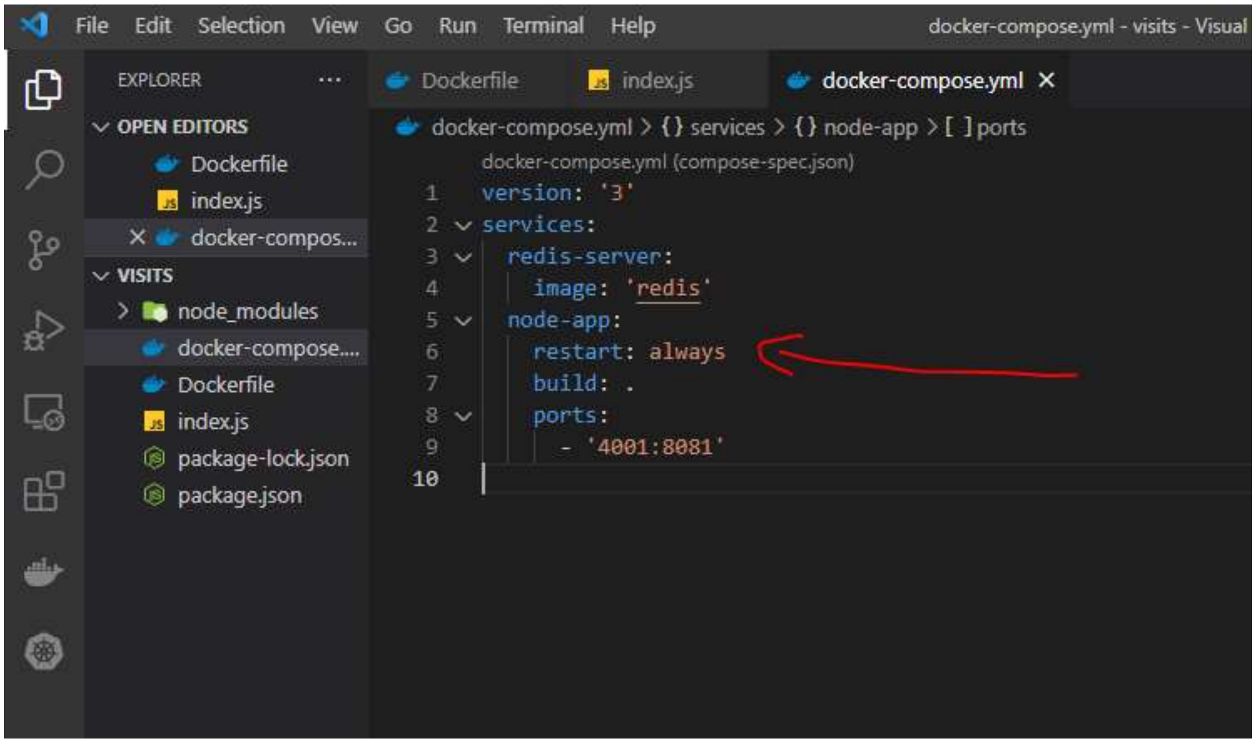
Generally docker compose has below restart policies



Restart Policies

"no"	Never attempt to restart this . container if it stops or crashes
always	If this container stops "for any reason" always attempt to restart it
on-failure	Only restart if the container stops with an error code
unless-stopped	Always restart unless we (the developers) forcibly stop it

If we want to tell docker-compose to always try to restart our container if it crushes, we can modify docker-compose file as follows:

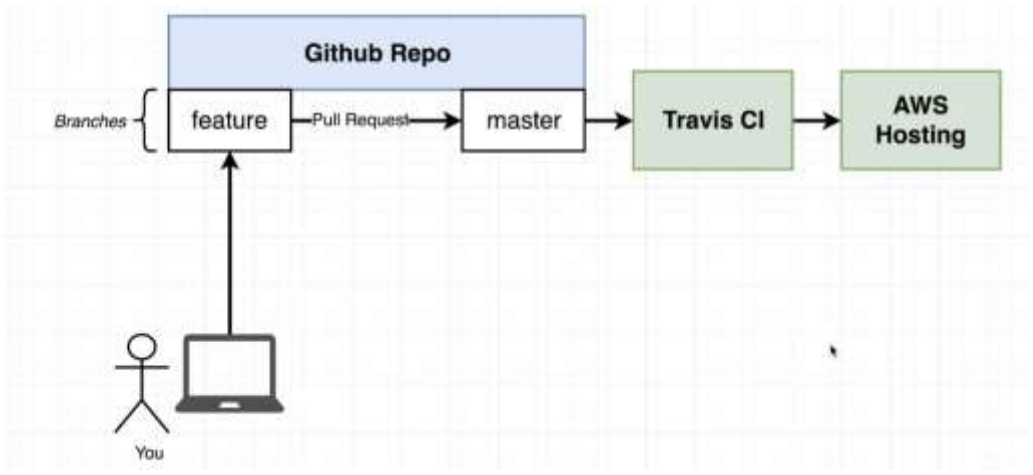

File Edit Selection View Go Run Terminal Help
docker-compose.yml - visits - Visual
EXPLORER
OPEN EDITORS
Dockerfile
index.js
docker-compos...
VISITS
node_modules
docker-compose....
Dockerfile
index.js
package-lock.json
package.json
docker-compose.yml > {} services > {} node-app > [ ] ports
docker-compose.yml (compose-spec.json)

```yaml
version: '3'
services:
  redis-server:
    image: 'redis'
  node-app:
    restart: always
    build: .
    ports:
      - '4001:8081'
```

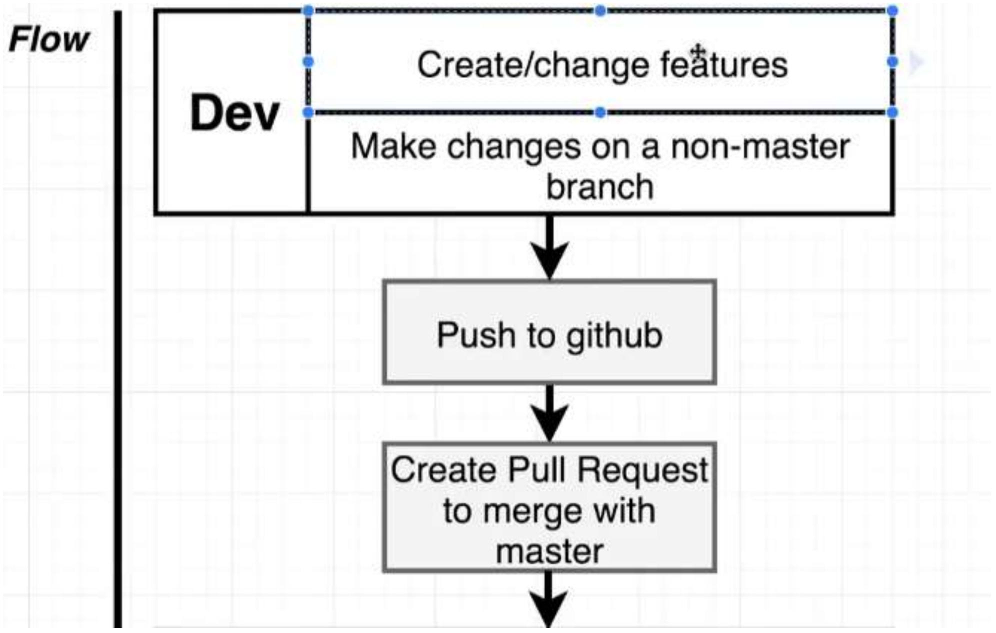
If on-failure is used, you'd need to change process.exit parameter from 0 to another number since 0 is regarded as intentional exit no failure at all.

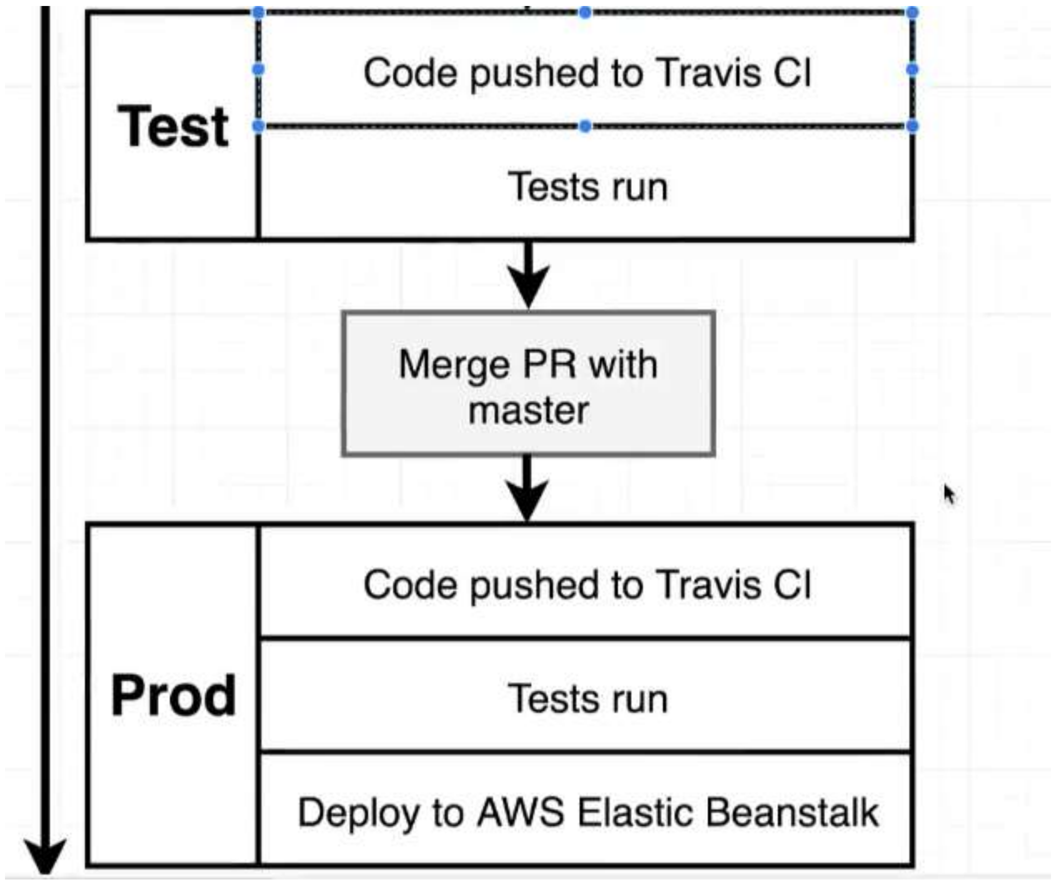
Publishing and deploying dockerized application

In ordinary application the flow to work for a developer evolves around development, testing and deployment, different tools can be used at each step. Diagram below shows minimal CI/CD flow



In a more detailed way, the above flow can be also represented as follows


Flow
Dev
Create/change features
Make changes on a non-master branch
Push to github
Create Pull Request to merge with master



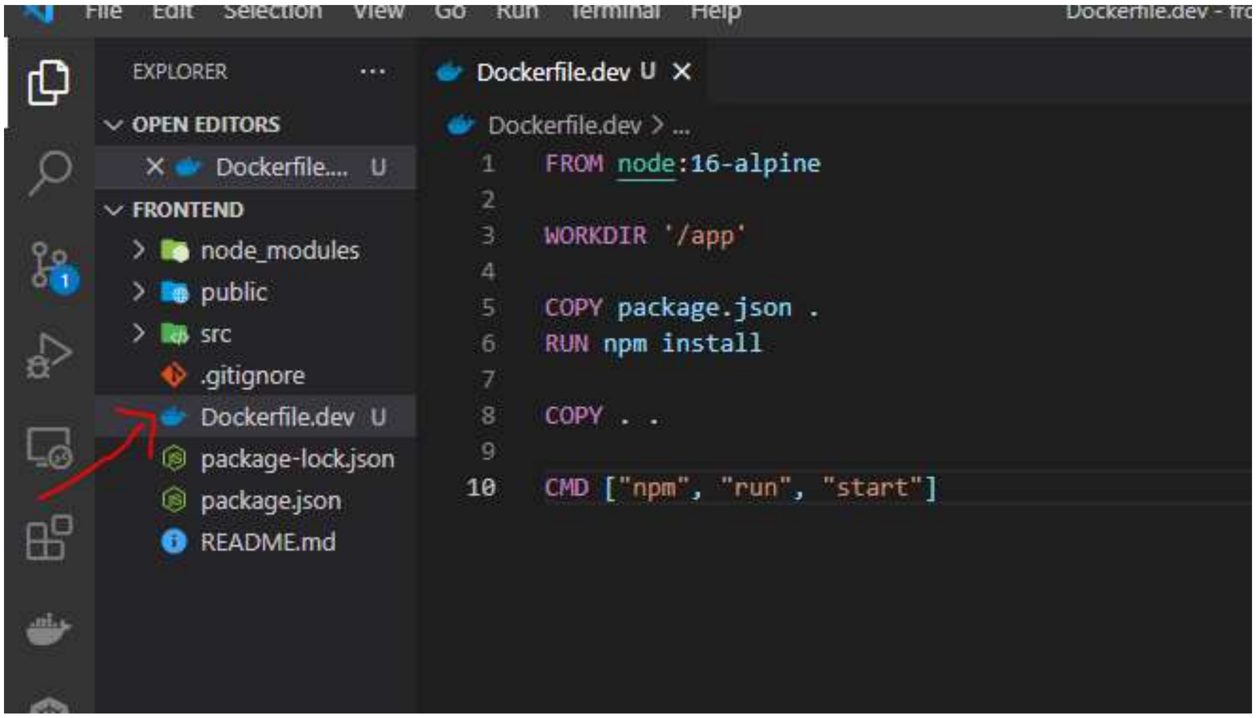
So in above flow where does docker come in? Docker is a tool in normal development flow and using it makes the work a lot easier.

Exampe:

Let us create a react app and go through all flows

Step1 : create react app `npx create-react-app frontend`

Step 2: Create a dev docker file `Dockerfile.dev`

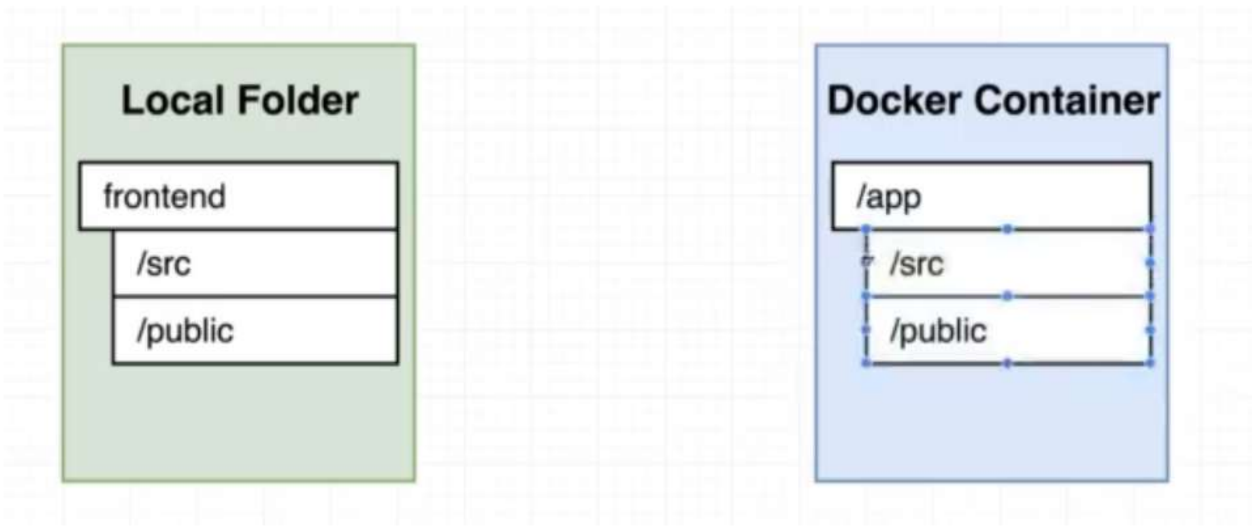


```
FROM node:16-alpine

WORKDIR '/app'

COPY package.json .
RUN npm install

COPY . .

CMD ["npm", "run", "start"]
```

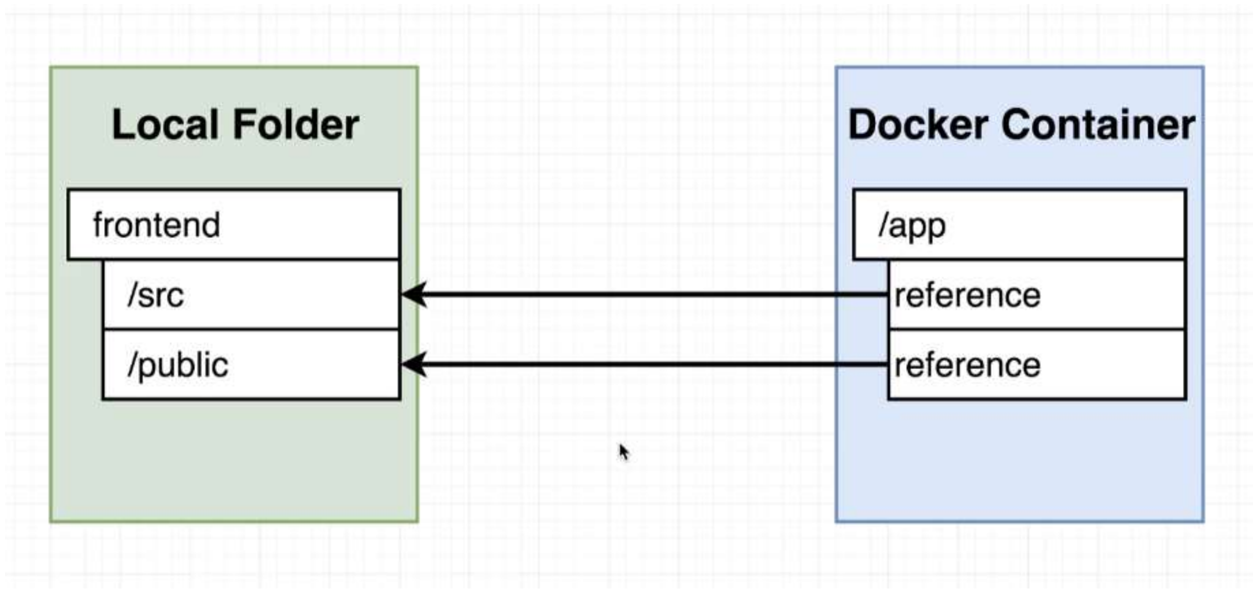
Step3: start the application container now docker run -p 3000:3000 16181dc65e84

To make sure that all changes made in our code are automatically detected we need to change our docker command by using volumes as follows:

- Normally during container creation from image we saw that fs is copied and pasted just like below diagram



- When we use volume, we keep a reference of local fs in our container fs, it just works like port apping we have seen before.

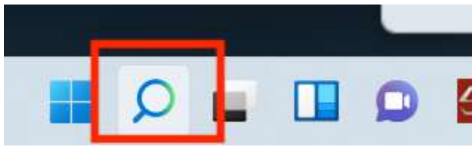


```
docker run -it -p 3000:3000 -v /home/node/app/node_modules -v ~/frontend:/home/node/app
<imgID>
```

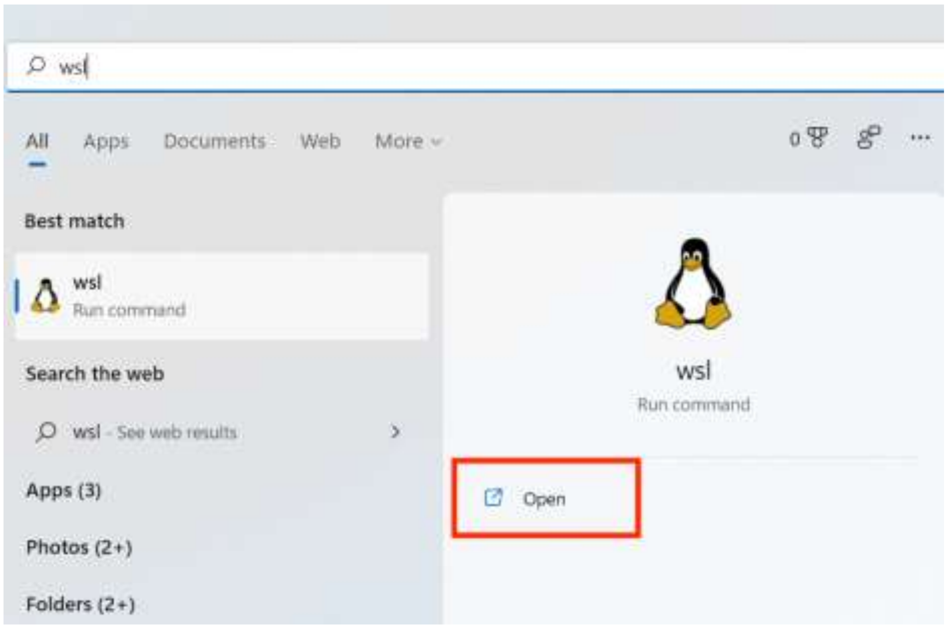
-v /home/node/app/node_modules is called bookmarking a folder, and it is not mapped to any folder on local PC. This allows container to create it and install dependencies instead of referencing one in local machine.

Note: Windows users All docker commands should be run within WSL2 and not on Windows.

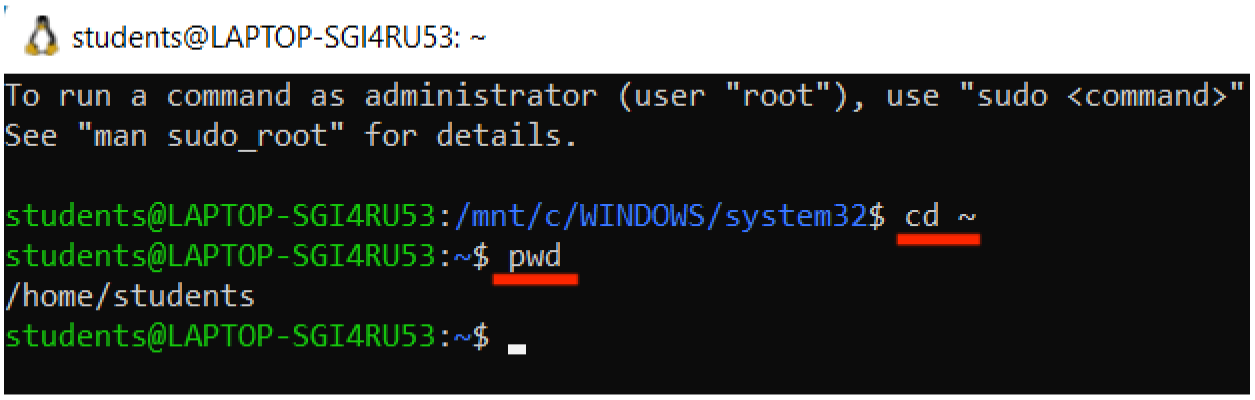
1. To open your WSL2 operating system use the search / magnifying glass in the bottom system tray:



2. type `wsl` in the Search Bar and click **Open** under the **wsl Run command** option:

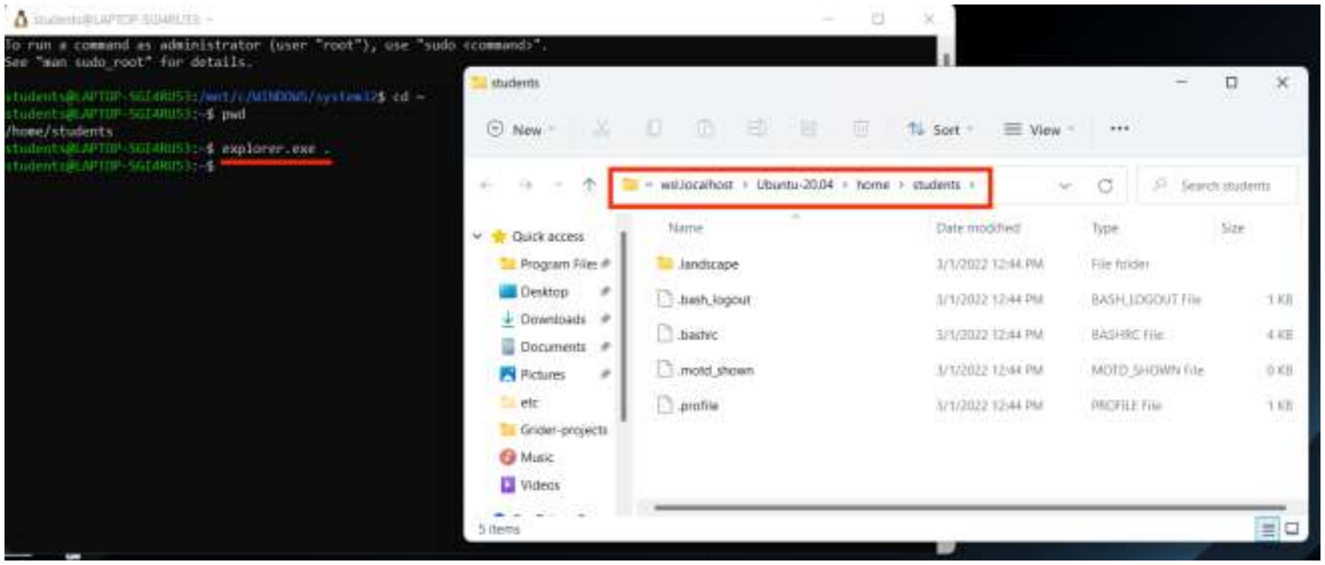


3. When the WSL terminal launches, change into the WSL user directory by running the `cd ~` command. Verify that you are in the correct location by running `pwd`. This should now printout /home/USERNAME. If it still shows /mnt/c/WINDOWS/system32, you are in the wrong location:



```
students@LAPTOP-SGI4RU53: ~
To run a command as administrator (user "root"), use "sudo <command>"
See "man sudo_root" for details.

students@LAPTOP-SGI4RU53:/mnt/c/WINDOWS/system32$ cd ~
students@LAPTOP-SGI4RU53:~$ pwd
/home/students
students@LAPTOP-SGI4RU53:~$ 
```

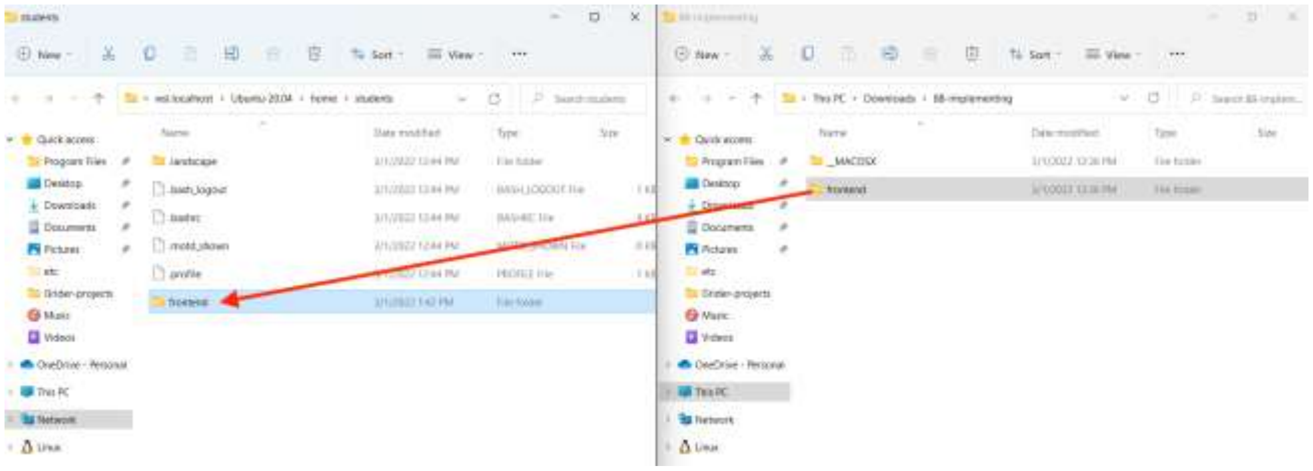
4. Run `explorer.exe .` to open up the file explorer at /home/students within WSL:



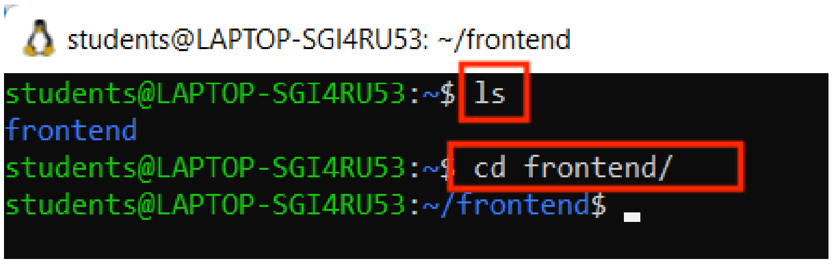
```
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

students@LAPTOP-5GI4RU53:/mnt/c/WINDOWS/system32$ cd ~
students@LAPTOP-5GI4RU53:~$ pwd
/home/students
students@LAPTOP-5GI4RU53:~$ explorer.exe .
students@LAPTOP-5GI4RU53:~$
```

Name	Date modified	Type	Size
.landscape	3/1/2022 12:44 PM	File folder	
.bash_logout	3/1/2022 12:44 PM	BASH_LOGOUT File	1 KB
.bashrc	3/1/2022 12:44 PM	BASHRC File	4 KB
.motd_shown	3/1/2022 12:44 PM	MOTD_SHOWN File	0 KB
.profile	3/1/2022 12:44 PM	PROFILE File	1 KB

5. Move the frontend project directory into the WSL file browser window:



6. Your project path should now look like **/home/USERNAME/frontend.** Run `ls` to confirm that you are in the correct location. Then, run `cd frontend` to change into the project directory.


```
students@LAPTOP-SGI4RU53: ~/frontend
students@LAPTOP-SGI4RU53:~$ ls
frontend
students@LAPTOP-SGI4RU53:~$ cd frontend/
students@LAPTOP-SGI4RU53:~/frontend$
```

7. Delete any **node_modules** or **package-lock.json** files that may exist in the project directory. If these were generated on the Windows file system and were copied over, they will conflict.
8. Using the WSL2 terminal build your Docker image as you typically would:

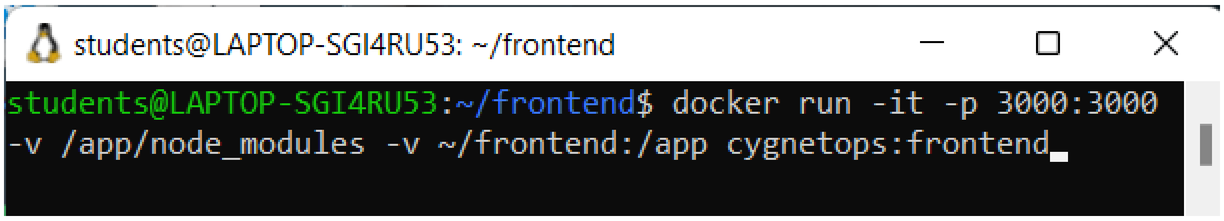
   ```
   docker build -f Dockerfile.dev -t USERNAME:frontend .
   ```

9. Using the WSL2 terminal, start and run a container. It is very important that you do not use a PWD variable as shown in the lecture video. Use the `~` alias for the home directory or type out the full path:

   ```
   docker run -it -p 3000:3000 -v /app/node_modules -v ~/frontend:/app
   USERNAME:frontend
   ```

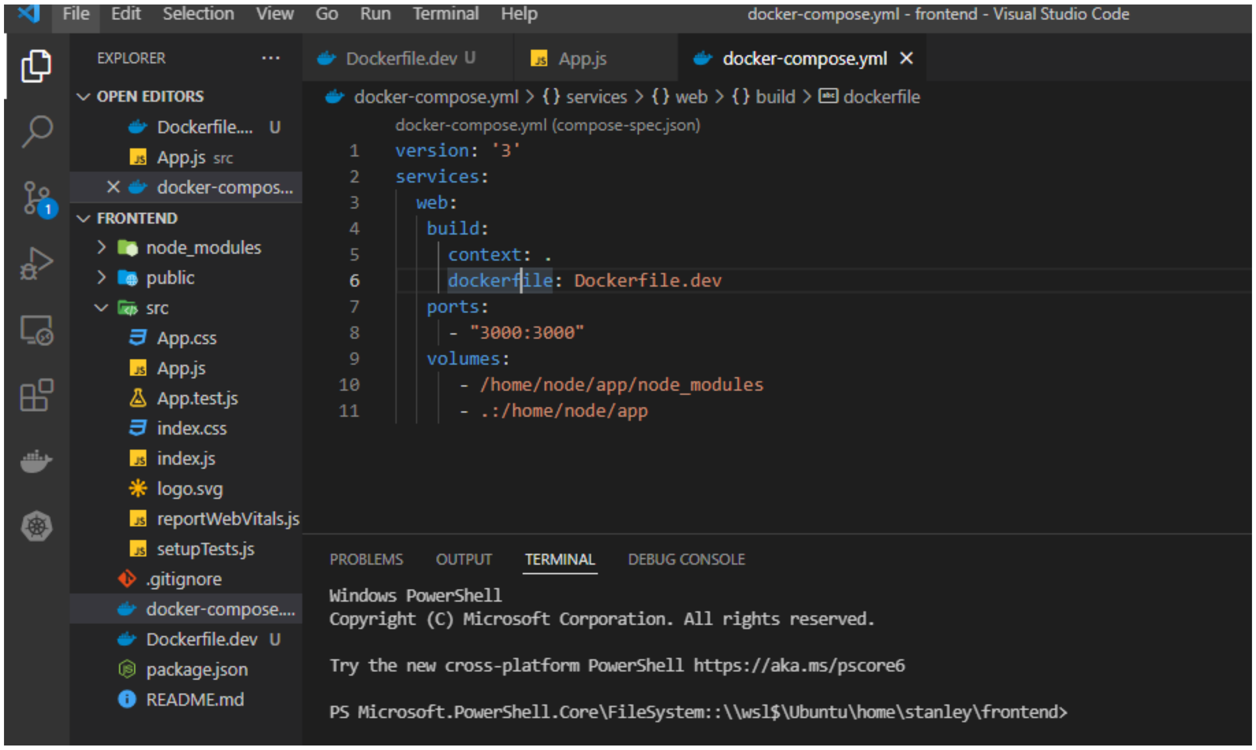
   or

   ```
   docker run -it -p 3000:3000 -v /app/node_modules -v /home/USER/frontend:/app
   USERNAME:frontend
   ```


```
students@LAPTOP-SGI4RU53: ~/frontend
students@LAPTOP-SGI4RU53:~/frontend$ docker run -it -p 3000:3000
-v /app/node_modules -v ~/frontend:/app cygnetops:frontend
```

10. Going forward, all Docker commands and projects should be run within WSL2 and not Windows.

Above command on step 9 can be simplified by using docker compose again and simply run docker-compose up

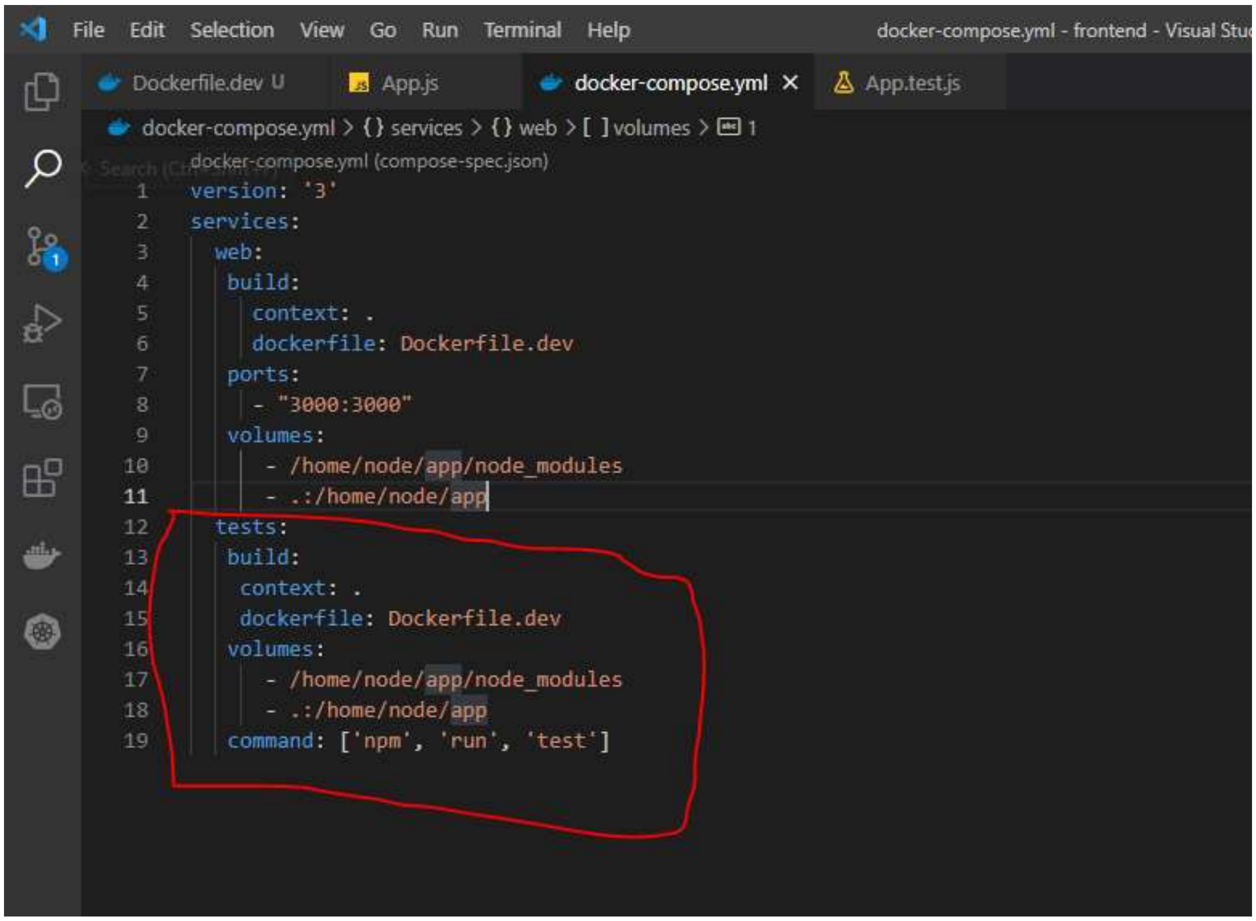


```yaml
version: '3'
services:
  web:
   build:
     context: .
     dockerfile: Dockerfile.dev
   ports:
     - "3000:3000"
   volumes:
      - /home/node/app/node_modules
      - .:/home/node/app
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS Microsoft.PowerShell.Core\FileSystem::\\wsl$\Ubuntu\home\stanley\frontend>
```

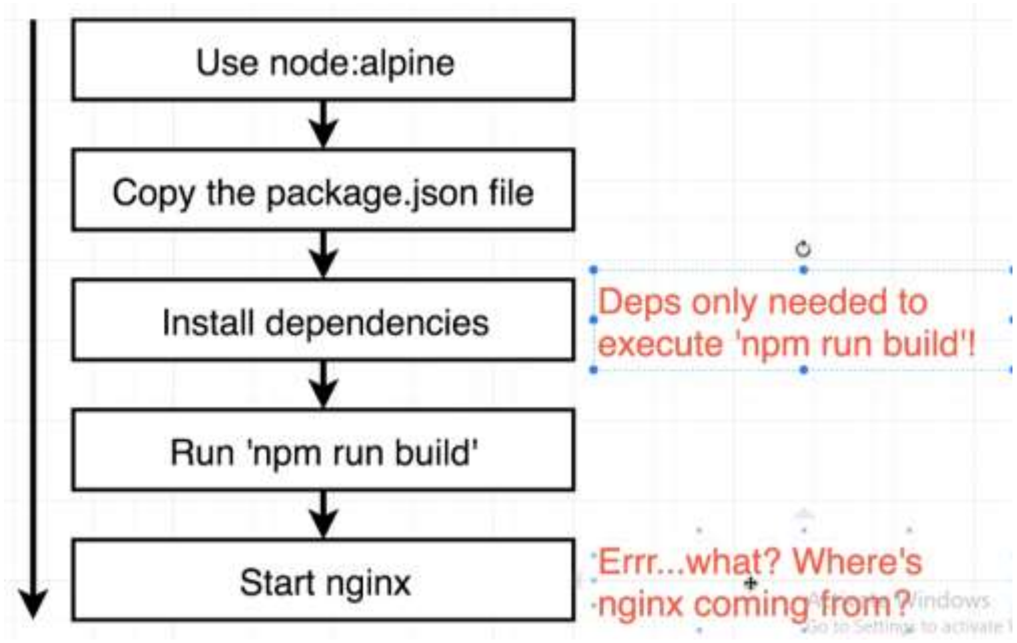
For testing part:

We can add another service in our docker-compose file as figure below show


File Edit Selection View Go Run Terminal Help
docker-compose.yml - frontend - Visual Stu
Dockerfile.dev U
App.js
docker-compose.yml
App.test.js
docker-compose.yml > {} services > {} web > [ ] volumes > 1
docker-compose.yml (compose-spec.json)

```yaml
version: '3'
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile.dev
    ports:
      - "3000:3000"
    volumes:
      - /home/node/app/node_modules
      - .:/home/node/app
  tests:
    build:
     context: .
     dockerfile: Dockerfile.dev
    volumes:
      - /home/node/app/node_modules
      - .:/home/node/app
    command: ['npm', 'run', 'test']
```

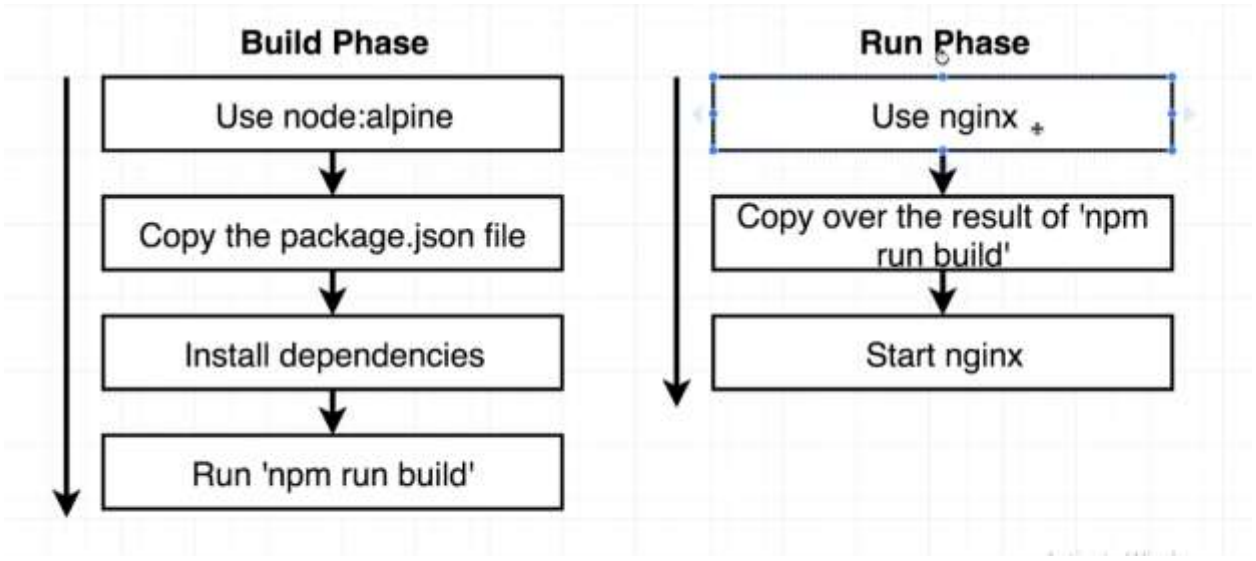
Prod Building phase – Implementing multi step docker builds


Use node:alpine
Copy the package.json file
Install dependencies
Run 'npm run build'
Start nginx
Deps only needed to execute 'npm run build'!
Errr...what? Where's nginx coming from?

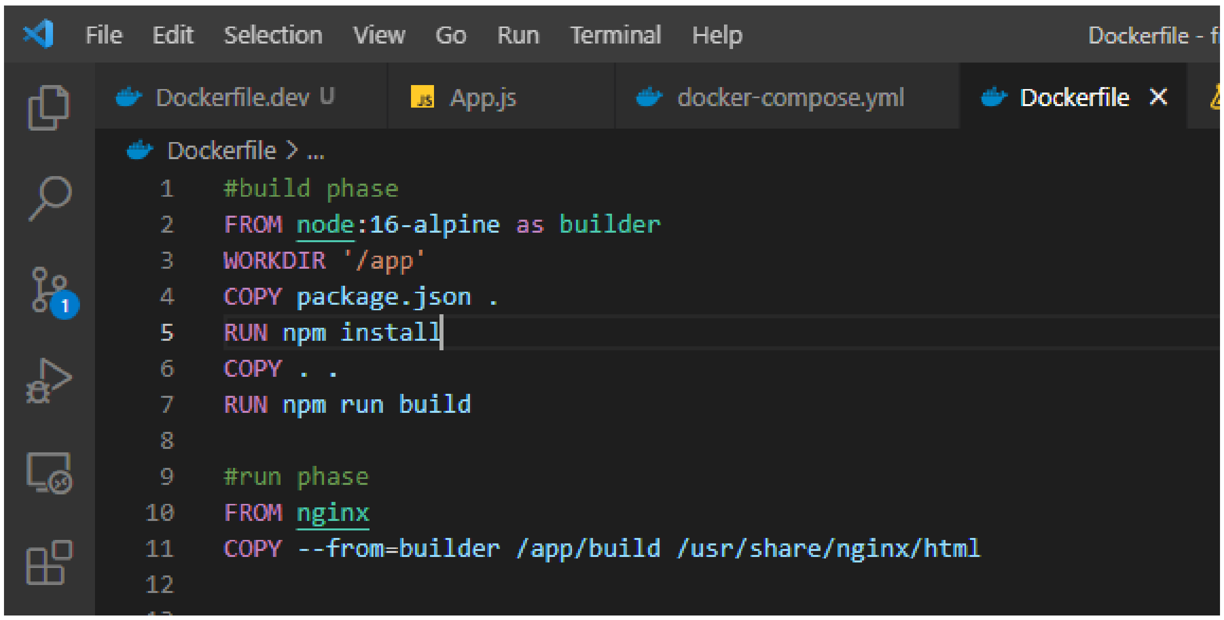
Here we have two big issues:

1) When you run npm run build, production assets/files are generated and stored in build folder, but npm has to install dependencies before building(it is a hard requirement that must be followed). These dependencies makes the project very huge, they can add many MBs to the folder size, it is then advisable that you do not ship them to production
2) Where are we getting nginx, how are going to integrate it in our flow. So we have an issue of nginx server set.

Above issues brings us to have a multi step docker build process as diagram below shows.



Our final dockerfile would look like the diagram below



```
#build phase
FROM node:16-alpine as builder
WORKDIR '/app'
COPY package.json .
RUN npm install
COPY . .
RUN npm run build

#run phase
FROM nginx
COPY --from=builder /app/build /usr/share/nginx/html
```

Note: no step is required to run nginx, it starts itself and by default uses port 80,

Now build image and run it but remember to map port 80 to machine port

```
$ Docker build .

$ docker run -p 8080:80 4f6db854d4b7
```

Deploying the app on AWS using github action flow

GitHub Actions are a powerful way to automate workflows helping you build, test and deploy your code. The marketplace is rapidly growing with new actions being added every day, doing everything from posting to Slack to deploying to AWS.

Refer to this article for some actions to help you deploy your code to Amazon AWS.

https://akashsingh.blog/complete-guide-on-deploying-a-docker-application-react-to-aws-elastic-beanstalk-using-docker-hub-and-github-actions

https://github.com/features/actions

https://github.com/marketplace?type=actions

The final github action yml file(docker-image.yml) would look like the one below

```yaml
name: Docker Image CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:

  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
      with:
       lfs: 'true'
```

```yaml
- name: Build the Docker image
  run: docker build -t mwizerwa77/docker-app-demo -f Dockerfile .

- name: Generate Deployment Package
  run: zip -r deploy.zip *

- name: Get timestamp
  uses: gerred/actions/current-time@master
  id: current-time

- name: Run string replace
  uses: frabert/replace-string-action@master
  id: format-time
  with:
    pattern: '[:\.]+'
    string: "${{ steps.current-time.outputs.time }}"
    replace-with: '-'
    flags: 'g'

- name: Deploy to EB
  uses: einaregilsson/beanstalk-deploy@v14
  with:
    aws_access_key: ${{ secrets.AWS_ACCESS_KEY_ID }}
    aws_secret_key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    application_name: docker-demo-app
    environment_name: Dockerdemoapp-env
    version_label: "docker-demo-app-${{ steps.format-time.outputs.replaced }}"
    region: us-east-1
    deployment_package: deploy.zip
```