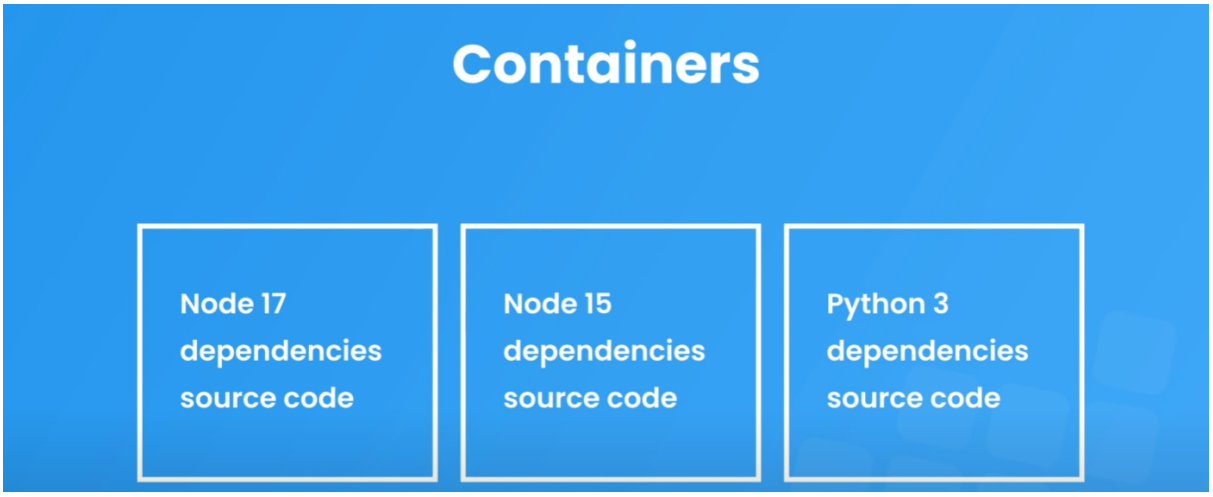
6. Docker

#1 What is Docker?

- Different applications will require → different configurations, dependencies, and libraries.
- Uses containers to run applications in isolate environment.
  - Containers → everything that our application requires in order to run.
  - Doesn’t matter what configurations we have, the application can run in isolation.
- Predictable, consistent, and isolated environment.
- Just need Docker on other systems to manage containers.



Basic idea behind Docker Containers

- Improve deployment process → pulled onto production server : all the configuration is already setup.
- But, why can't we use Virtual Machines instead of Docker? The idea seems to resembles the same!
  - **Virtual Machines** → has full operating system and typically slower!
  - **Containers** → Share the host's operating system and typically quicker!
    - Less memory - lightweight
  - Not that one is better than the other, there might be several cases where we might want to use VMs instead of containers.

#2 Images & Containers

- **Images**
  - Read-Only
  - Blueprint for containers
  - Include every single thing that our application needs to run.
    - Runtime environment, code, dependencies, additional configuration, and commands.
  - We can share the image to allow to form the container to run the application.
- When we run an image → Container is formed.
  - Container is the running instance of the image.
  - Isolated process → Like running in a box which is isolate from other processes in the computer.
- Images are made up from several *layers.*

  → We start with inclusion of Parent Image : OS and Runtime environment.

  → Source Code, Dependencies, etc.

  → Docker Hub

  → By specifying tags, we can determine the version of image along with Linux distribution.

  → Example : Alpine distribution → Lightweight

  → Doesn't have to be a node image, it could be Python or Ruby, anything.

```
docker pull node
```

#3 Docker File

- Whenever we start with docker, we will be starting with a parent image and then, we will continue to add on different layers.
- Create our own image → Docker File
  - Fundamentally, a set of instructions to develop an image → different layers that needs to be included in the image.
- Normally, if we have to run an application we will do so by installing all the dependencies and then, executing *node app*
- But, when I do so, it is going to use the node version that is installed locally on the computer and this could be different for everyone, causing issues.
- We need to run our application inside an isolated container, which has its own version of node running inside of it.
- **Docker File**
  - Each line represents layer in final image.

```
FROM node:17-alpine

// This is going to pull the image from local system - if we have downloaded it or
// from the docker hub if we haven't.

WORKDIR /app

// add a work directory where all commands will be run

COPY . /app -> COPY . .

// copy all of our soure code into the image -> first dot is the relative path of the
// directory

// We don't directly copy to the root : we copy to /app -> which is a specific directory
// for our code.

RUN npm install

// asks docker to run a command while the image is being built
// Run is executed during the build time -> when the image is being built

EXPOSE 4000

// Which Port the container should Expose. Also tells the port that is going to be
// used by the application. -> Kind of optional.
// Alternatively, we can do this by cmd : port mapping

CMD ["node", "app.js"]

// Run the application in container
```

```
// Runs the instance
// Write the commands in double quotes
```

- To run the image
  - *docker build -t my_app .*
  - T flag allows us to give a tag and the dot at the end defines where the Docker file exists.
- **Docker Ignore**
  - Sometimes the node modules folder will exists inside of our project folder and if the Docker File contains the *install* command → it will create two copies of the same folder.
  - If we create a new image → node modules will also be installed.
  - Create a docker ignore file → this will ignore files or folder when they are copied to the image.
  - Example - * . md : ignore all files related to md

#4 - Spin a Container

- We can run image to create a container → that will run our application.
- From Docker Desktop -
  - We can specify extra information → such as ports that will allow us to reach the application within the container from our system.
  - Map Local Host Port to Container's Port : These don't have to be same and can be different. However, we keep them same to avoid any sort of confusion.
  - Stopping doesn't delete the container.
- CMD
  - To perform any action → we need either ID or name of Image or Container
  - List all images : docker images
  - Command : docker run —name *container-name image-name*
    - This will create a new container and run it.
    - However, when we will try to run this command → our app will start but we will not be able to access it from our browser.
    - Reason ? → We can't access the port without performing the mapping → as we did in the desktop.

    - To fix this, open a new CMD, and first stop the running container
    - docker ps → running processes i.e. container
    - docker stop *name / id*
    - Now we can run -

```
docker run --name mynewimage_c3 -p 4000:4000 -d mynewimage

// -p flag allows to map the ports
// -d helps in detaching the process i.e. running the process without blocking container
```

- List all existing containers (either running or stopped) → docker ps -a
- To re-run or restart a container : docker start

#5 - Layer Caching

- Every line in Docker File is a layer and adds something to the image.
- For each layer in the file, Docker tends to take some time.
- Now imagine, if we make a change to the app → we will need to make a new image to pick up that change.
- Every time docker tries to build an image : it leverages Cache to build new images faster.
- We can move the RUN npm install command above COPY
    - This would run before copying of the source code → However, it would not find the Package JSON file.
    - Solution : Create a COPY command for Package JSON.

#6 - Managing Images and Container

```
docker image rm image-name

-----------------------------------------------------------------------------------------

- Force delete the image

docker image rm testimage3 -f

-----------------------------------------------------------------------------------------

- To delete a container

docker container rm myapp
```

```
-------------------------------------------------------------------------------------------

- To remove all containers and images at once

docker system prune -a
```

```
PS D:\Docker\api> docker image rm testimage3
Untagged: testimage3:latest
PS D:\Docker\api> docker image rm testimage2
Error response from daemon: conflict: unable to remove repository reference "testimage2" (must force)
 - container b8d19205e4d1 is using its referenced image 6dd3ab4e4df7
```

Versioning

- After the image name → colon : can be added and that allows us to specify the version

```
docker build -t myimage:v1 .

// v1 tag will be added to the image

-------------------------------------------------------------------------------------------

docker run --name mycontainer -p 4000:4000 myimage:v1

// run container for a specific version of the image
```

#7 - Volumes

- Once an image is made → it becomes *read-only* and we will have to create new image to ensure the changes are observed.
- Solution → Use volumes.
  - Allows us to specify folders on host computer that can be made available to running containers.
  - Any change on local system would be reflected in the container.

```
FROM node:17-alpine

RUN npm install -g nodemon // new line added

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 4000
```

```
#CMD ["node", "app.js"]

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

- Write a command for this in the package JSON file and we need -L flag for it.
    - We need extra flag in order for it be executed in the container environment.

```
docker run --name myapp_c_nodemon -p 4000:4000 --rm  myapp:nodemon

# if we add the rm flag in the command, it is going to delete the container once it
# stops. That way we are not going to store any container when not in use!
```

- However, even with nodemon - the changes are not observable live. It’s because the file in the container haven’t undergone any change - only our files in the local system have changed.
- Map our API folder from localhost to /app work directory in container.

```
docker run --name myapp_c_nodemon -p 4000:4000 --rm -v D:\Docker\api:/app  myapp:nodemon
```

- But if we delete node modules from API, it would also delete it from APP work directory.
    - Solution : Anonymous volume

```
docker run --name test_c -p 4000:4000 --rm -v D:\Docker\api:/app -v /app/node_modules test
```

#8 - Docker Compose

- If we have multiple components in project, it will be difficult to write commands for each of them every time we have to spin up a container.
- By using Docker Compose, we can mention all the configuration in one single file which will allow us to spin up all the containers at once.
- Always make the compose file in root folder.
- In YAML, we always have to take the indentation in account.

```
version: "3.8"
services:
  api:
    build: ./api -> Docker Compose will build the image and run the container also.
    container_name: api_c
    ports:
      - '4000:4000'
    volumes:
```

```
      - ./api:/app
      - ./app/node_modules
```

- CMD
  - docker-compose up : To run the compose file.
  - docker-compose down —rmi all -v

#9 - Docker React

- Start with Docker File

```
FROM node:16-alpine

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 3000
# required for docker desktop port mapping

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

- Docker Compose

```
version: "3.8"
services:
  api:
    build: ./api
    container_name: api_c
    ports:
      - "4000:4000"
    volumes:
      - ./api:/app
      - /app/node_modules
  myblog:
    build: ./myblog
    container_name: myblog_c
    ports:
      - "3000:3000"
    volumes:
      - ./myblog:/app
      - /app/node_modules
    stdin_open: true
    tty: true
```