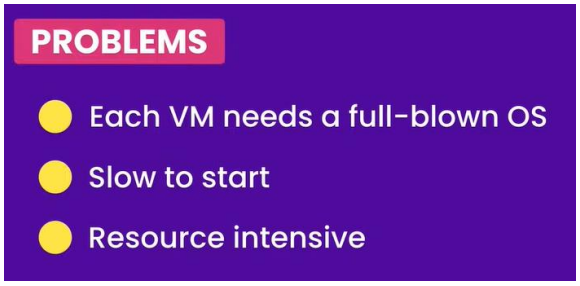
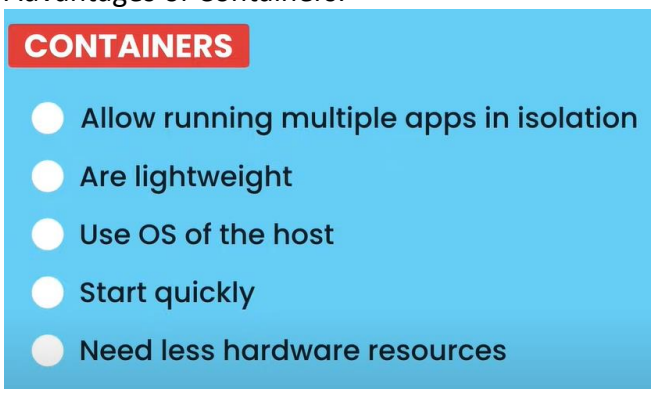
DOCKER NOTES:

1- What is Docker?
A platform for building, running, and shipping applications

2- What is a container:
Container vs Virtual Machine:
Problems with VMs:



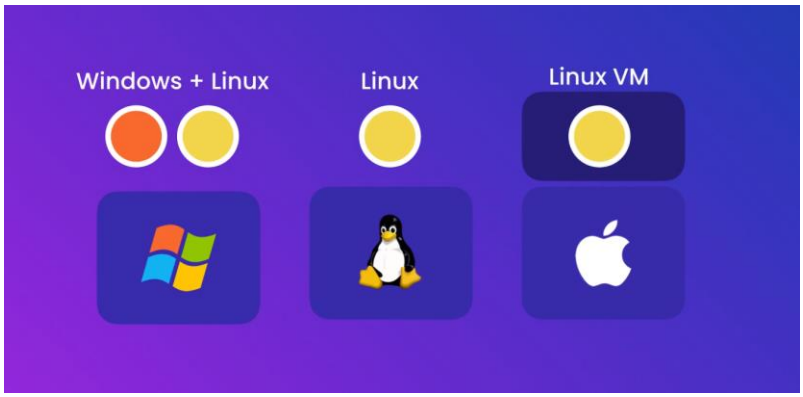
Advantages of Containers:



3- The architecture of Docker:

Docker uses a client server architecture: it has a client component that talks to a server component through a REST API. This server is also called Docker engine which takes care of building a Docker container.

4- Container:
A container is a process (like a normal computer process). Containers share the kernel of the host. A kernel manages applications and hardware resources.
Kernels in different OS:



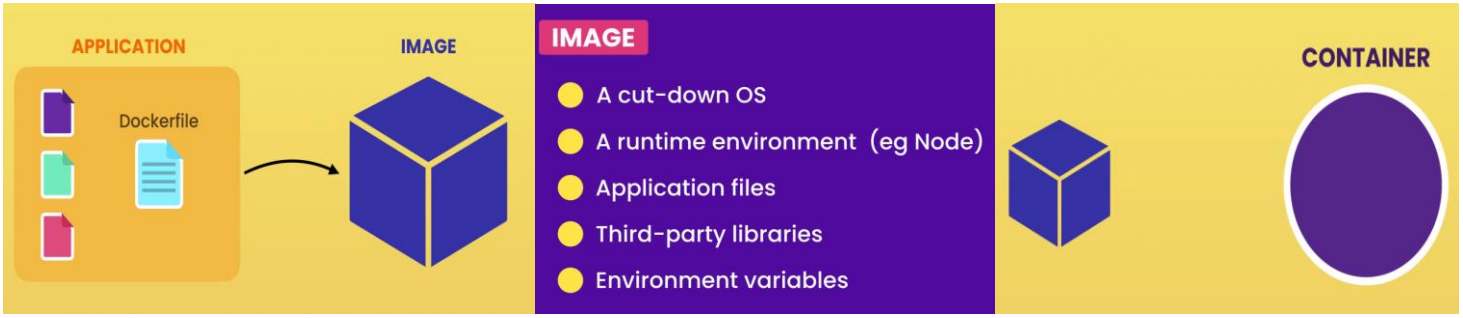
5- Installing Docker:

Enable Hyper-V and Containers on Windows features

Docker version -----→ 20.10.7

6- Development Workflow:

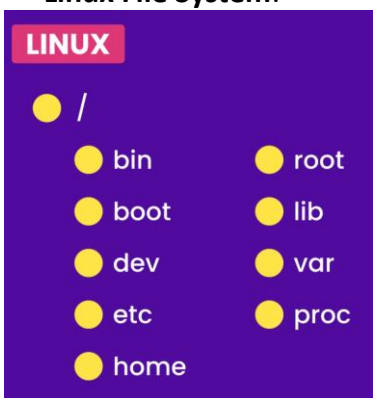
Dockerize the application make an image put the image in a container



7- Linux Command Line:
- Linux Distributions: Ubuntu, Debian, Alpine, Fedora, Centos….

- Choosing Ubuntu for the rest of the tutorial:
*Running Linux:
- Pulling Ubuntu image from Docker Hub – usually we use *docker pull ubuntu* but we can use *docker run ubuntu*
- Start the ubuntu image from a container: *docker run -it ubuntu*
- Linux is case sensitive.
- command *history* shows all the commands we used
- command *!2* executes the second command return by the history command.
Docker package manager: **apt**
- Updating the packages list: ***apt update***
- to install a package: *apt install <package name>*

To clear the command line: **ctrl + L**
- to remove a package: *apt remove nano*
** **Linux File System**:



LINUX COMMANDS:

- **pwd**: print working directory
- **ls**: lists, change layout results: **ls -1** and **ls -l and ls -a** (show all files – even hidden ones)
- **cd**: change directory – **cd ..** go back – **cd ../..** go to root directory – **cd ~** home directory
- **mkdir:** create directory
- **mv**: rename directory: example: **mv test docker** - or move files/directories to another directory
- **touch**: create a file
- **rm**: remove files and directories – **rm file*** - remove all files that start with file – **rm -r docker/** - remove docker directory
- **nano**: file editor in Linux, it is not included by default (**apt install nano**)
- **nano file1.txt**: create a new file and open it in text editor to write
- **cat**: concatenate files or see the content of a file (if the file is short)

- **more**: see the content of bigger files (only supports scrolling down)
- **less**: see the content of a file (using the up and down arrows) (**apt install less**)
- **Q**: to exit when inside another window
- **head**: show the first few lines of the content of a file (**head -n 5 /etc/adduser.conf** shows the first 5 lines of the file named adduser.conf stored in the directory etc)
- **tail**: show the last few lines of the content of a file (**tail -n 5 /etc/adduser.conf)**
- Redirection: using the "**>**" operator: showing the output elsewhere

Example: **cat file1.txt > file2.txt**

- **grep**: search files for strings – **grep -i hello file*** : search for the word hello in all files that start with name file and case insensitive (-i) – **grep -i -r hello .** : search current directory for the word hello
- combining options: **grep -i -r hello .** is equivalent to **grep -ir hello .**
- **find**: finding all files and directories in the current directory – **find -type d** (list all directories) – **find -type f** (list all files) **– find -type f -name "f*"** (find all files that start with f) - **find -type f -iname "f*":** same but case incensitive
-
```
root@510431ff8a44:~# find / -type f -iname "*.py" > python.txt
```

Finding all python files in the root directory (on the image) and writing the result to a file called python.txt

- chaining/combining commands: **mkdir test; cd test; echo done;** note the semi colon! If your commands are too long, you can divide your commands into different lines as follows:

mkdir test;
cd test;
echo done;

we also can use logic operators here to make sure all commands get executed together or not:

mkdir test && cd test && echo done or **mkdir test || echo "directory exists"**

Note: the ***echo*** command is equivalent to the **print** command in python

- piping: **ls /bin | less**

- environment variables:
- **printenv**: shows all environment variables on the machine – **printenv PATH (case sensitive):** print the value of the variable PATH
- **echo $PATH**: Also shows the value of the PATH variable (similar to **printenv PATH**)

Setting a variable:

- **export DB_USER=houssem**: setting a variable named DB_USER to a value of Houssem, however, this variable is available only on the open session, once the session is closed, we lose that variable.
- **docker ps -a**: to see all docker containers
- **exit**: exit a session
- **docker start -i 2f7**: to start a container – Note: 2f7 are the first 3 characters of the container id. The option -i is included here to enable interaction with the container
- **echo DB_USER=Houssem >> .bashrc** to create a permanent environment variable (that it is not deleted when session is closed). **Note**: bashrc file is where we have to write our permanent environment variables. Also, we use the ">>" instead of ">" to avoid overwriting the content of bashrc, instead we want just to append to it.
- **source .bashrc**: to see the variable change, we need to reload the bashrc file from the ***home*** directory.

Managing processes: a process is an instance of a running program

- **ps**: see all running processes
- **sleep 3 &**: create a process and put it in the background
- **kill 37**: kill the process with id 37

Managing users:

- **useradd -m John**: add a user John to **/stc/passwd**-
- **usermod**: modify user params -for example – change shell to bash: **usermod -s /bin/bash john**
- **cat /etc/shadow**: where passwords are saved (this file is only accessible to the root user)
- log in with a different user:
```
C:\Users\Owner>docker exec -it -u john 510431ff8a44 bash
```

Note: **-it**: to interact with docker, **-u**: to specify user, **510431ff8a44**: docker container id, **bash**: open a bash session for user john. John has less permission than the root user:

```
john@510431ff8a44:/$ cat /etc/shadow
cat: /etc/shadow: Permission denied
```

- **userdel john**: delete user
- **adduser**: more interactive than **useradd,** in general use useradd with docker.

Managing groups:

- **groupadd**: create a group – **groupadd developers**
- **usermod -G developers john**: add a user to a group
- **grep john /etc/passwd**:

```
root@510431ff8a44:~# grep john /etc/passwd
john:x:1000:1000::/home/john:/bin/bash
```

- **groups john**: shows all the groups that john is a part of

```
root@510431ff8a44:~# groups john
john : john developers
```

Here john is part of 2 groups: john and developers: john is a primary group and developers is a supplementary group.

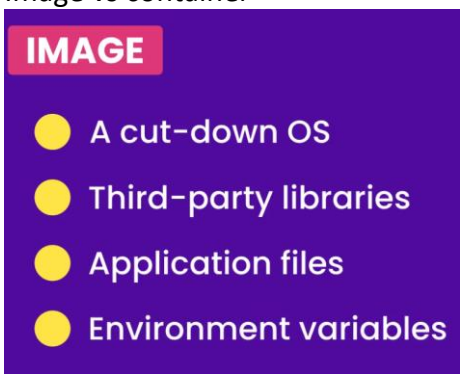
- **usermod --append john -G artists**: add john to a third group without removing an older group.

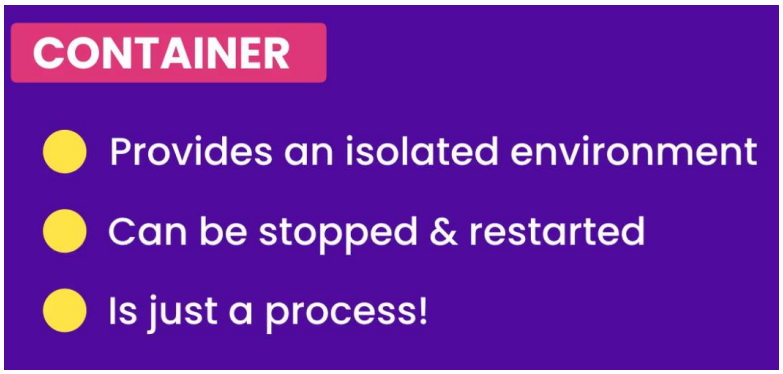
File Permissions:

- **chmod u+x deploy.sh**: Give permission to the user to execute deply.sh file – **chmod o+x**: give permission to others – **chmod g+u**: give permission to group owner – **chmod og+x+w-r *.sh**
- **chmod u-x**: remove permission

Docker Building Images:

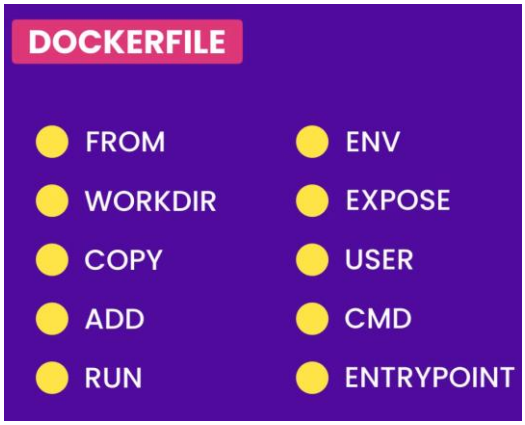
Image vs container





Dockerize an application:

Create a dockerfile: a dockerfile contains instructions for building an image



Create a new file in the working directory of your project (in VS code), this file is called **Dockerfile** (no extension)

1- Choosing the right base Image:

For JavaScript, we need a node image, example: ***16.4-alpine3.14***

```
FROM node:16.4-alpine3.14
```

1-0- Building the image:
docker build -t react-app . : -t for tagging the image and "." For current directory.
1-1- **docker image ls**: to view all built images
1-2- **docker run -it react-app**: we will enter a node environment.
1-3- **docker run -it react-app bash**: will enter a bash environment
1-4- **docker run -it react-app sh**: will enter a shell environment

2- copying application files and directories into the image:
-Copying multiple files into the image: `COPY package.json README.md /app/`
Note: /app/ is the image directory(if it doesn't exist, docker will create it)
-Using patterns: `COPY package*.json /app/` : coping all files that start with package and ends with .json
- copying everything in the current directory to the image: `COPY . /app/`
- Destinations: /app/ (this is an absolute path because it start with "/"), we can also use a relative path if we set the **WORKDIR** first:

```
WORKDIR /app
COPY . .
```

here we have set the WORKDIR to /app, so in COPY, we replace /app/ to "."
Note: if we are trying to copy a file that has a space in its name, then we ca use an array of files to copy like this:

```
COPY ["hello world.text", "."]
```

ADD: similar to COPY, but has few extra features: we can add files from web links – also, ADD can unzip (zip) files.

3- Excluding files and directories:
Similar to git: create a file named: **.dockerignore**

4- Running commands:
RUN: `RUN npm install`

5- Setting environment variables:
ENV: `ENV API_URL=http://api.myapp.com/`

6- Exposing Ports:
EXPOSE: `EXPOSE 3000`: specify the port where the application will be running

7- Setting the User:
USER:

```
RUN addgroup app && adduser -S -G app app
USER app
```

setting the USER to app (not root)

8- Defining Entry Points:
CMD: `CMD npm start` or CMD: `CMD ["npm", "start"]` (this one is favored)
Difference between **CMD** and **RUN**: RUN gets executed when we build the image, but CMD gets executed when we run the container.

Note: **ENTRYPOINT** is similar to CMD (use it only when we are certain about the command that we want to start when running the container) – syntax is similar to CMD

FINAL DOCKERFILE FORM:

```dockerfile
FROM node:16.4-alpine3.14
RUN addgroup app && adduser -S -G app app
USER app
WORKDIR /app
COPY . .
RUN npm install
ENV API_URL=http://api.myapp.com/
EXPOSE 3000
#CMD npm start
CMD ["npm", "start"]
```

9- Speeding up builds:
Optimize building the image:

```dockerfile
FROM node:16.4-alpine3.14
RUN addgroup app && adduser -S -G app app
USER app
WORKDIR /app
COPY package*.json .
RUN npm install
COPY . .
ENV API_URL=http://api.myapp.com/
EXPOSE 3000
CMD ["npm", "start"]
```

In general:

-- Removing Images:
Using these 2 commands to remove dangling images:
docker image prune
docker container prune

to delete an actual image: **rm image <image name or image id>**

-- Tagging images:
`docker build -t react-app:1 .`: tagging image while building it
Or
`docker image tag react-app:latest react-app:1`

Remove tag:
`docker image remove react-app:1`

-- Sharing Images: uploading image to docker hub

1- create a repository on docker hub
2- run: `docker image tag fce zoug86/react-app:2` Note: ***fce*** are the first 3 characters of the image id
3- run: **docker login**
4- run: `docker push zoug86/react-app:2`

-- Saving and Loading images:
- save: **docker image save -o react-app.zip react-app:3**
- load: **docker image load -i react-app.zip**

------### WORKING with CONTAINERS###------

-- Starting containers:
To view running contianers: **docker ps**
To run a container: **docker run -d –name blue-sky react-app**: we named this container blue-sky. Note: -d for detached, which allows the container to run in the background and gives us back the terminal to write more commands.

-- Viewing the logs:
To view log: **docker logs <container id>** (to see all options run docker logs - - help)

-- Publishing Ports:
To allow the container to open the web page on port 3000 of the host:
```
docker run -d -p 3000:3000 --name new react-app
```

-- Executing commands in running containers:
Use the **exec** command: `docker exec new ls`
Open a shell command: `docker exec -it new sh`

-- Stopping and Starting Containers:
Stopping: `docker stop new`: stopping a container called new
Starting: `docker start new`: restarting the container stopped above

-- Removing Containers:
Removing with force: **docker rm -f <container name>**
docker ps -a | grep <container name>: to see if a stopped container still exists on a long list of containers
docker container prune: delete all stopped containers

-- Persisting data using Volumes:
Create a new volume: **docker volume create app-data**
Inspect the volume: **docker volume inspect app-data**
Mapping the volume to a directory in the file system of the container:
```
docker run -d -p 4000:3000 -v app-data:/app/data react-app
```

Note: a normal user cannot write to a normal directory (data directory above), unless we add the data directory manually on the dockerfile.
```
USER app
WORKDIR /app
RUN mkdir data
```

Then build the image again: **docker build -t react-app .**

Now start a new container: `docker run -d -p 5000:3000 -v app-data:/app/data react-app`

-- Copying files between Host and containers:

- Copying from container to host: `docker cp 453a2277457c:/app/log.txt .`
- Copying files from host to container: `docker cp secret.txt 453a2277457c:/app`

-- Sharing the Source Code with a container:

In Windows:

```
docker run -d -p 5001:3000 -v "C:\Users\Owner\Desktop\coding training\Docker\Section 4- Images\section4-react-app":/app react-app
```

In Linux:

```
docker run -d -p 5001:3000 -v $(pwd):/app react-app
```

------### Running multi- CONTAINER Applications###------

using docker compose

With docker compose, there is no need to install frontend and backend (and database) dependencies separately, all we need run is this:

```
docker-compose up
```

The ***docker-compose.yml*** file: **YAML** language format:

Here is how a typical YAML file looks like:

```yaml
---
name: The Ultimate Docker Course
price: 149
is_published: true
tags:
  - software
  - devops
author:
  first_name: Houssem
  last_name: Marzougui
```

Note the three hyphens up top. YAML works by indentation (similar to python)

-- Creating a docker compose YAML file:

1- create a new file named: **docker-compose.yml**

docker-compose.yml

```yaml
version: "3.8"

services:
  frontend:
    depends_on:
      - backend
    build: ./frontend
    ports:
      - 3000:3000

  backend:
    depends_on:
      - db
    build: ./backend
    ports:
      - 3001:3001
    environment:
      DB_URL: mongodb://db/vidly
    command: ./docker-entrypoint.sh

  db:
    image: mongo:4.0-xenial
    ports:
      - 27017:27017
    volumes:
      - vidly:/data/db

volumes:
  vidly:
```

2- build the image: **docker-compose build**
Force a new rebuild: **docker-compose build –no-cache**

3- start the application: **docker-compose up -d**
Note: we can combine 1 and 2 in one go: **docker-compose up --build**
Stop the application: **docker-compose down**

-- Docker Networking:
View all docker networks: **docker network**
Ping one container form another: **docker exec -it -u root 8c6 sh**, then: **ping backend**
ifconfig: to see IP addresses of containers

-- Viewing logs:
docker-compose logs – to see logs for one container: **docker logs <container_id>**

-- Publishing changes:
Sharing source code between containers and host
Added**: volumes: - ./backend:/app and - ./frontend:/app**

```
  backend:
    depends_on:
      - db
    build: ./backend
    ports:
      - 3001:3001
    environment:
      DB_URL: mongodb://db/vidly
    #command: ./docker-entrypoint.sh
    volumes:
      - ./backend:/app
```

Problem: could not get **nodemon** to work with docker compose

--Migrating the database:
- create a docker entrypoint shell file and call it: ***docker-entrypoint.sh***

```
#!/bin/sh

echo "Waiting for MongoDB to start..."
./wait-for db:27017

echo "Migrating the databse..."
npm run db:up

echo "Starting the server..."
npm start
```

Then in the ***docker-compose.yml*** file include the command line as follows:

```
environment:
    DB_URL: mongodb://db/vidly
command: ./docker-entrypoint.sh
volumes:
    - ./backend:/app
```

-- Running Tests:
Can do them separate or include them inside the container:

```
frontend-test:
    image: vidly_frontend
    volumes:
      - ./frontend:/app
    command: npm test
```

------### Deploying Applications###------

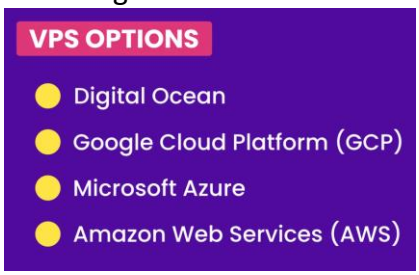
-- Deployment Options:

- Single-host deployment
- Cluster deployment

For cluster deployment solutions:

- Docker Swarm
- Kubernetes

-- Getting a Virtual Private Server:



-- Install docker machine:

```
$ base=https://github.com/docker/machine/releases/download/v0.16.0 \
>   && mkdir -p "$HOME/bin" \
>   && curl -L $base/docker-machine-Windows-x86_64.exe > "$HOME/bin/docker-machine.exe" \
>   && chmod +x "$HOME/bin/docker-machine.exe"
```

```
base=https://github.com/docker/machine/releases/download/v0.16.0 \
  && mkdir -p "$HOME/bin" \
  && curl -L $base/docker-machine-Windows-x86_64.exe > "$HOME/bin/docker-machine.exe" \
  && chmod +x "$HOME/bin/docker-machine.exe"
```

-- Provisioning a Host:

1- `$ export DOTOKEN=your-api-token`

$export DOTOKEN=88087eba9c18895e82573f8d287850d3aec81a211f530e209116471295d0bbe1

2-

```
$ docker-machine create --driver digitalocean --digitalocean-access-token $DOTOKEN --digitalocean-image ubuntu-18-04-x64 --engine-install-url "https://releases.rancher.com/install-docker/19.03.9.sh" vidly
```

docker-machine create --driver digitalocean --digitalocean-access-token $DOTOKEN --digitalocean-image ubuntu-18-04-x64 --engine-install-url "https://releases.rancher.com/install-docker/19.03.9.sh" vidly

--Connecting to the Host:

Run: `$ docker-machine ssh vidly`

-- Defining the Production Configuration:

We need to create a new ***docker-compose.yml*** file for production phase:

docker-compose.prod.yml

```yaml
version: "3.8"

services:
    frontend:
        depends_on:
            - backend
        build: ./frontend
        ports:
            - 80:3000
        restart: unless-stopped
    backend:
        depends_on:
            - db
        build: ./backend
        ports:
            - 3001:3001
        environment:
            DB_URL: mongodb://db/vidly
        command: ./docker-entrypoint.sh
        restart: unless-stopped

    db:
        image: mongo:4.0-xenial
        ports:
            - 27017:27017
        volumes:
            - vidly:/data/db
        restart: unless-stopped

volumes:
        vidly:
```

-- Reducing Image size:

Create a production Dockerfile: ***Dockerfile.prod***

frontend > Dockerfile.prod > ...

```dockerfile
# step 1: build stage
FROM node:14.16.0-alpine3.13 AS build-stage
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# step 1: production stage
FROM nginx:1.12-alpine AS production-stage
RUN addgroup app && adduser -S -G app app
USER app
COPY --from=build-stage /app/build /usr/share/nginx/html
EXPOSE 80
ENTRYPOINT [ "nginx", "-g", "deamon off;" ]
```

Then run the following:

```
$ docker build -t vidly_web_opt -f Dockerfile.prod .
```

Then modify the ***docker-compose.prod.yml*** file as follows:

docker-compose.prod.yml

```yaml
version: "3.8"

services:
    frontend:
        depends_on:
            - backend
        build:
            context: ./frontend
            dockerfile: Dockerfile.prod
        ports:
            - 80:80
        restart: unless-stopped
```

Then run:
$ docker-compose -f docker-compose.prod.yml build

-- Deploying the application:

1- run: **$ docker-compose -f docker-compose.prod.yml build**
2- run: **@FOR /f "tokens=*" %i IN ('"C:\Users\Owner\bin\docker-machine.exe" env vidly') DO @%i** (run it on CMD not bash)
3- **docker-compose -f docker-compose.prod.yml up -d**
Then we got error (permission denied)
To fix this, add this line in the backend Dockerfile:

```
RUN addgroup app && adduser -S -G app app
RUN mkdir /app && chown app:app /app
USER app
```

4- Now run this: ***docker-compose -f docker-compose.prod.yml up -d –build (on CMD)***

5- add the ENV variables*:*

```
COPY . .
ENV REACT_APP_API_URL=http://159.89.37.140:3001/api
RUN npm run build
```

6-Rebuild: ***docker-compose -f docker-compose.prod.yml up -d –build (on CMD)***

-- Publishing changes:
Add tags to images:

```
    context: ./frontend
    dockerfile: Dockerfile.prod
image: vidly_frontend:1
ports:
    - 80:80
```