Docker Course

Section 2: Docker Images & Containers: The Core Building Blocks:

Building our own Image with a Dockerfile:

Running a Container based on our own Image:

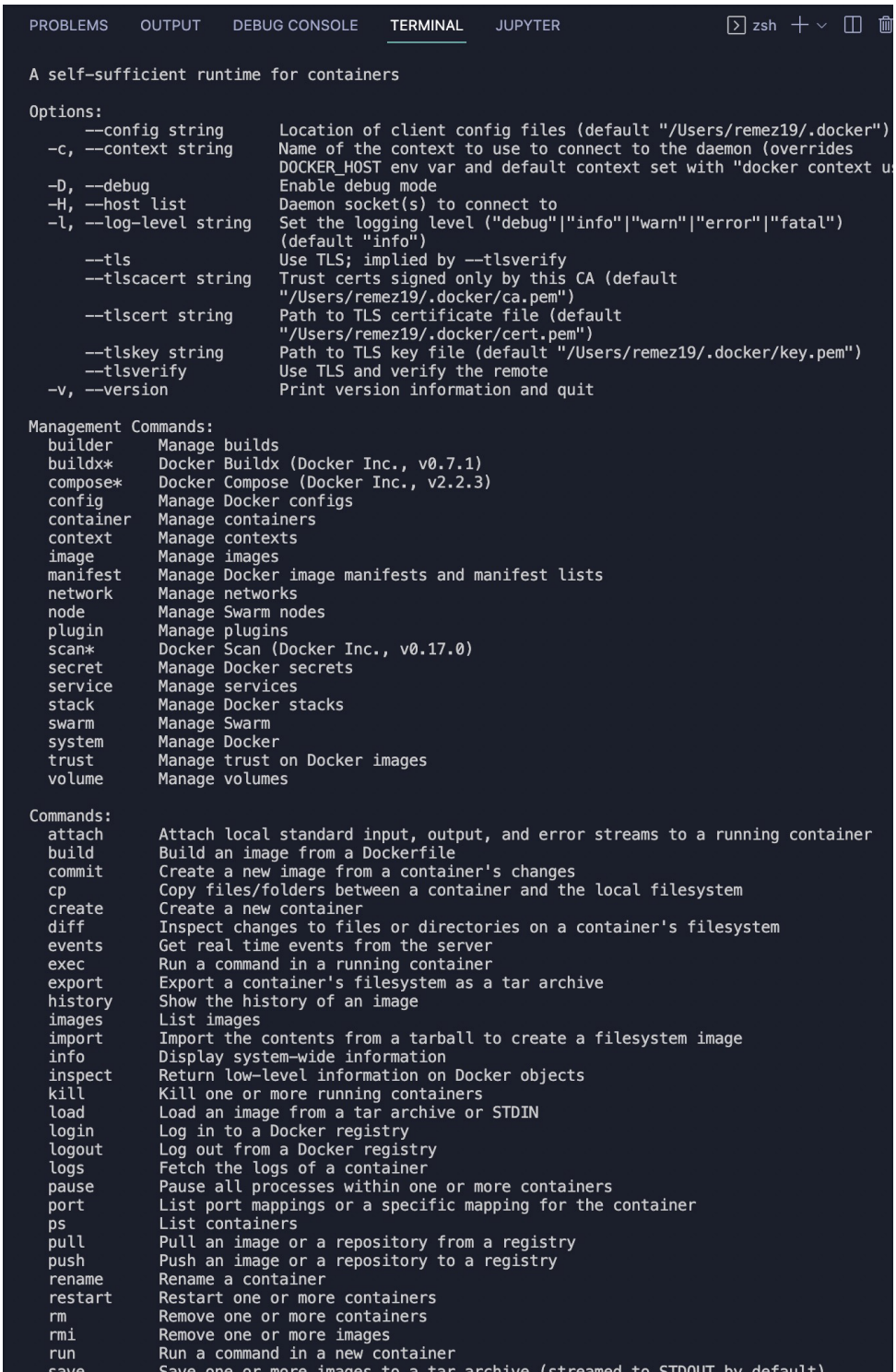
- "**docker build [path to the docker file we want to build no need for []]**" tells docker that we want to build an image base on a "Dockerfile".
- "**docker run [image id no need for []]**" will run the image.
- "**docker stop [image name no need for []]**" will stop the container.
- "**docker ps**" will show all the current running containers.

- “**docker ps -a**” will show all the created containers.
- “**docker run -p 3000:3000 [image name]**” will run the container and enable our local machine to communicate with it.

Images are Read-Only!:

Understanding Image Layers:
After every instruction in the Docker file. Thew docker engine saves a copy of that command result.
Therefore if we try to build the same image docker will get the cached command result.
In cases where docker detect a change in a result of a command all the other commands after that command will also be executed again.
In short the order in which we write the commands in the Dockerfile are important.

Managing Images & Containers:
- “docker —help” will show all the available commands of docker.



Stopping & Restarting Containers:

- "docker start [containerName/contianerID]" will start the specific container (not a new one).
- "docker container prune" will delete all the stopped containers.
- "docker image prune" will remove all unused images.

You can do a lot more (search the web for more info).

Understanding Attached & Detached Containers:

- Adding the flag "-d" in-front of the image id will run a new container in

death mode (the terminal will not be stuck).
- We can attach to a running container by using “docker attach [container name]”

Removing Stopped Containers Automatically:
- Add “—rm” to the run command when we create a container to make sure that this container will be
  deleted when stopped.

Copying Files Into & From A Container:
- “docker cp src CONTAINER_NAME/des” copy files from and into a running container.
  To copy from a container to a different place we change the command so that the container is the src.

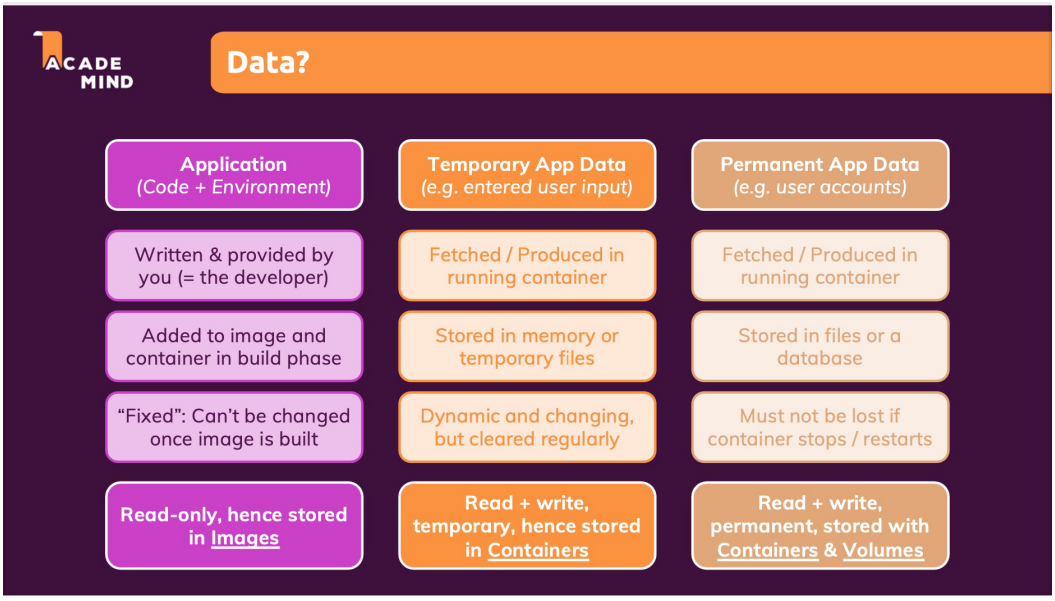
Naming & Tagging Containers and Images:
- Using “—name NAME” we can give any name to a container that we create with the run command.
  “docker run —name MY_NAME IMAGE_ID” .
- To name an image we use “docker build -t NAME:TAG .”.
- To rename a container “docker rename CONTAINER_NAME NEW_NAME”.
- For first time (when using run). “docker run -p 3000:3000 -d —name [MY_Name] —rm [Image_Id]” (-d for detached mode not have to, also the —rm).

Sharing Images - Overview:
When it comes to sharing images we have two options:
- Sharing the Dockerfile and the source code as well.
- Share a built image.

Pushing Images to DockerHub:
Docker has a built in commands for sharing images (key mechanism).
We have to options:
- Docker hub.
- Private registry.

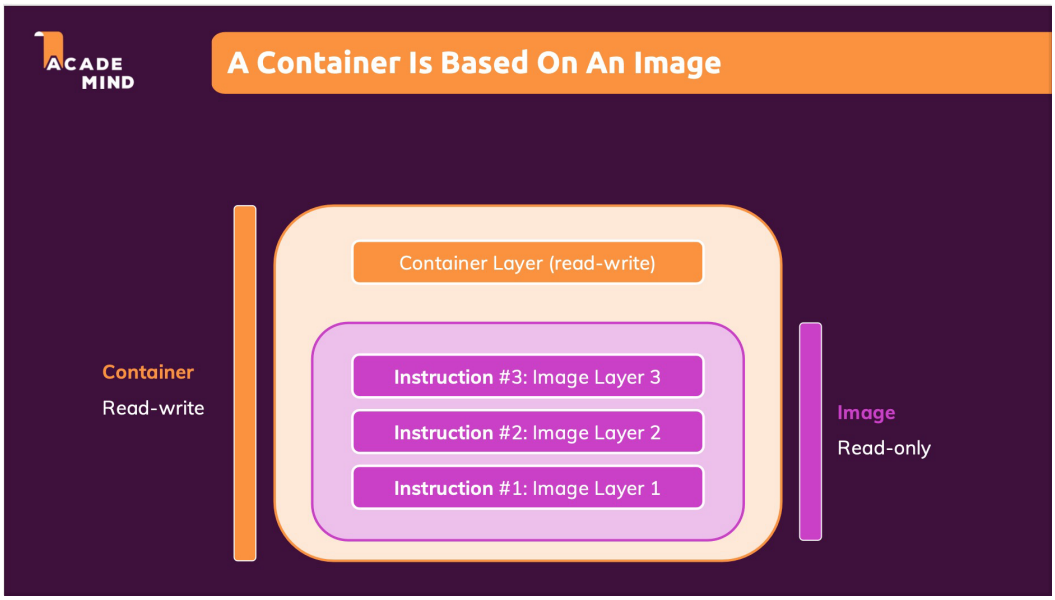
Section 3: Managing Data & Working with Volumes:

Understanding Data Categories / Different Kinds of Data:



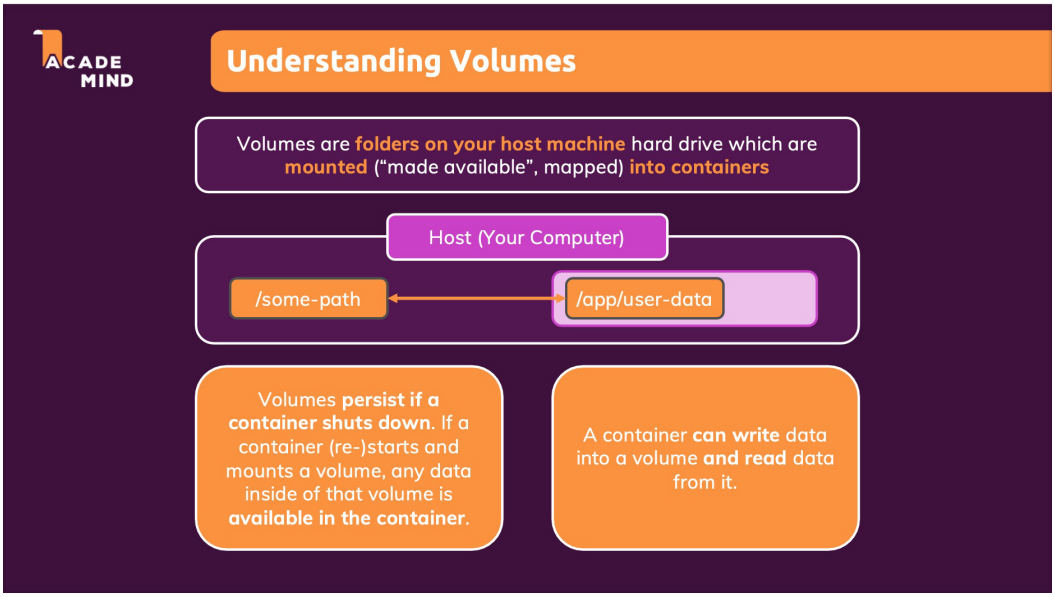
Understanding the Problem:

When we start a container the container add a layer of read-write.
The container have access to the image file system.
Once the container is removed the data that it saved removed too.
Every container is isolated and data is not shared (by default) between two containers.
Even if the two containers are based on the same
Image!



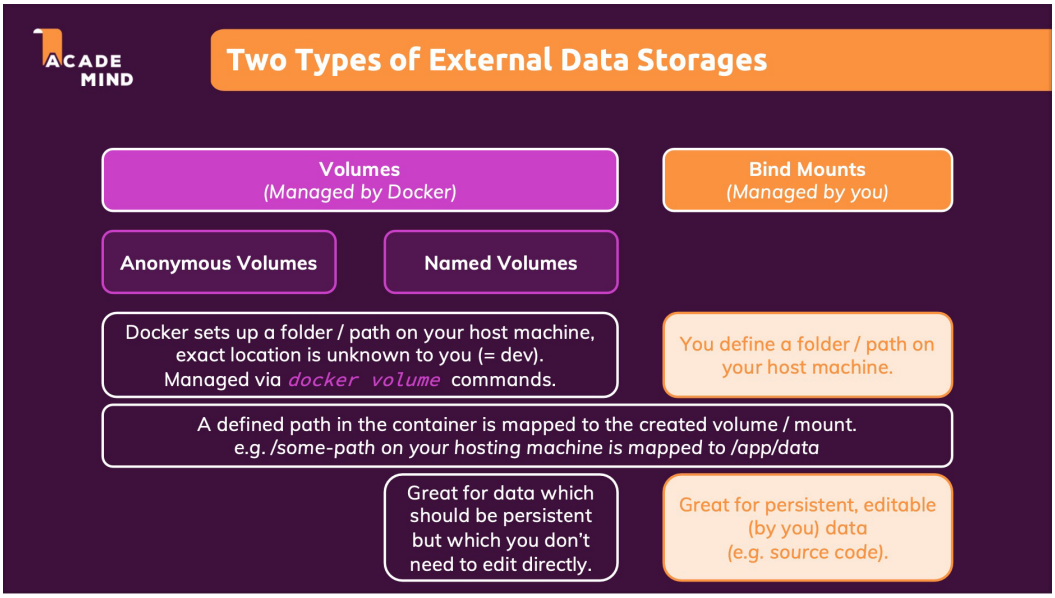
Introducing Volumes:

Volumes enable the option to save data even if a container is deleted.
Volumes are just folders in our host machine.
With volumes we give access to a container to our host machine folders.
With that we can save data. (Long term)

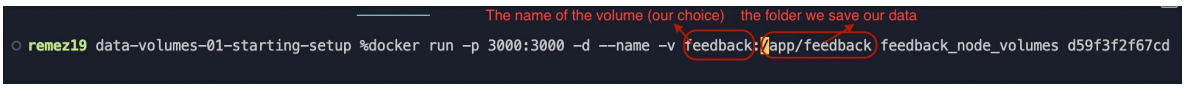


Named Volumes To The Rescue!:

Named volumes will not be deleted when a container is removed.



The command is:

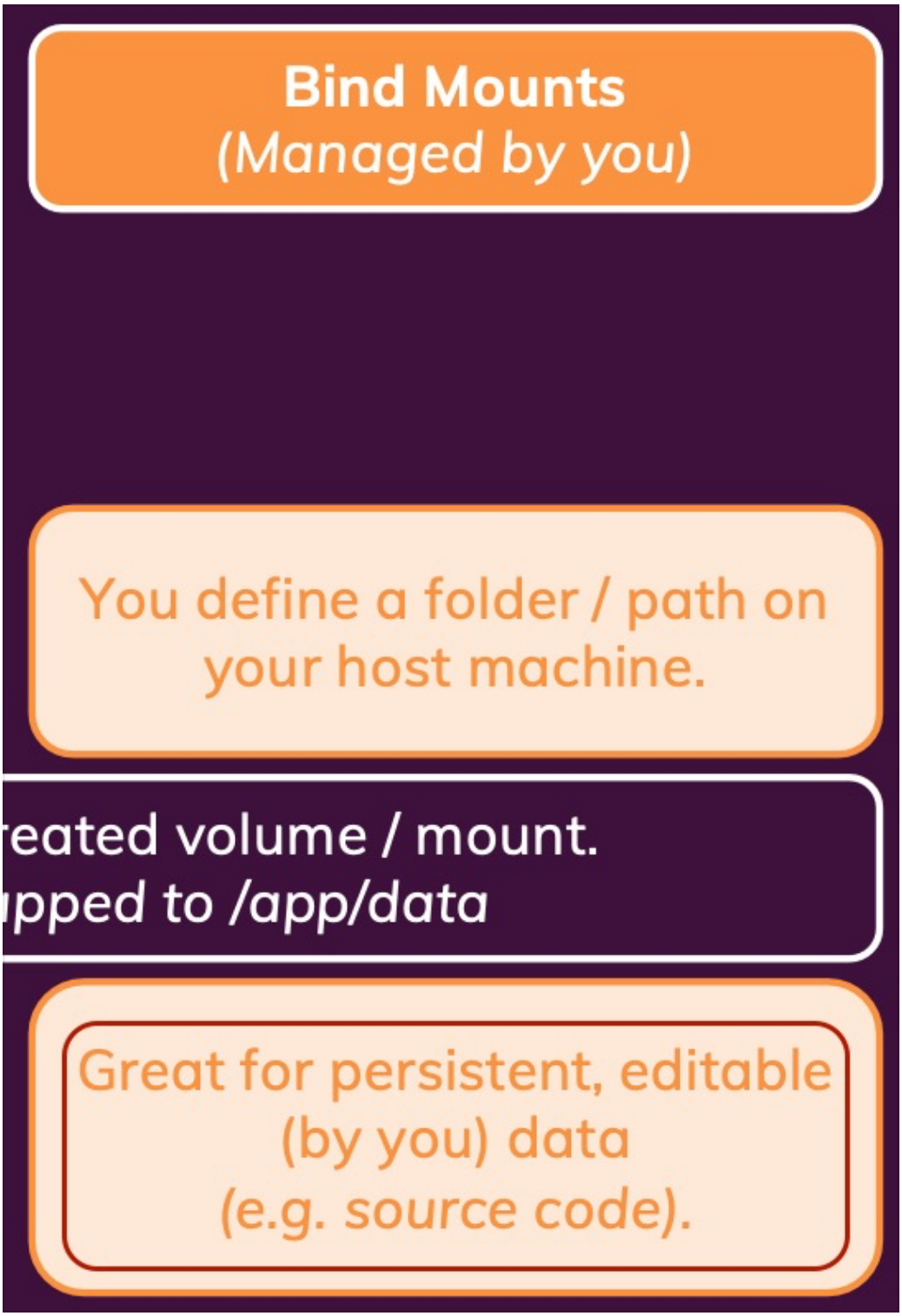


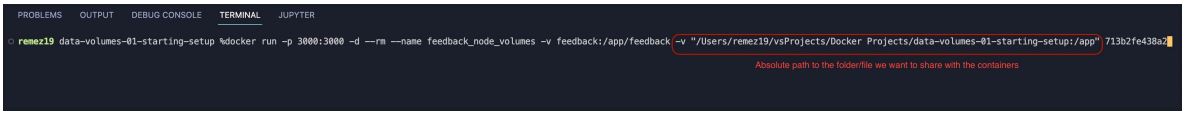
If we want to get access to the volume after a remove (run another container) we need to use the same we gave the volume!
To remove all volumes: "docker volume prune".
To remove a specific volume: "docker volume rm VOL_NAME"

Getting Started With Bind Mounts (Code Sharing):

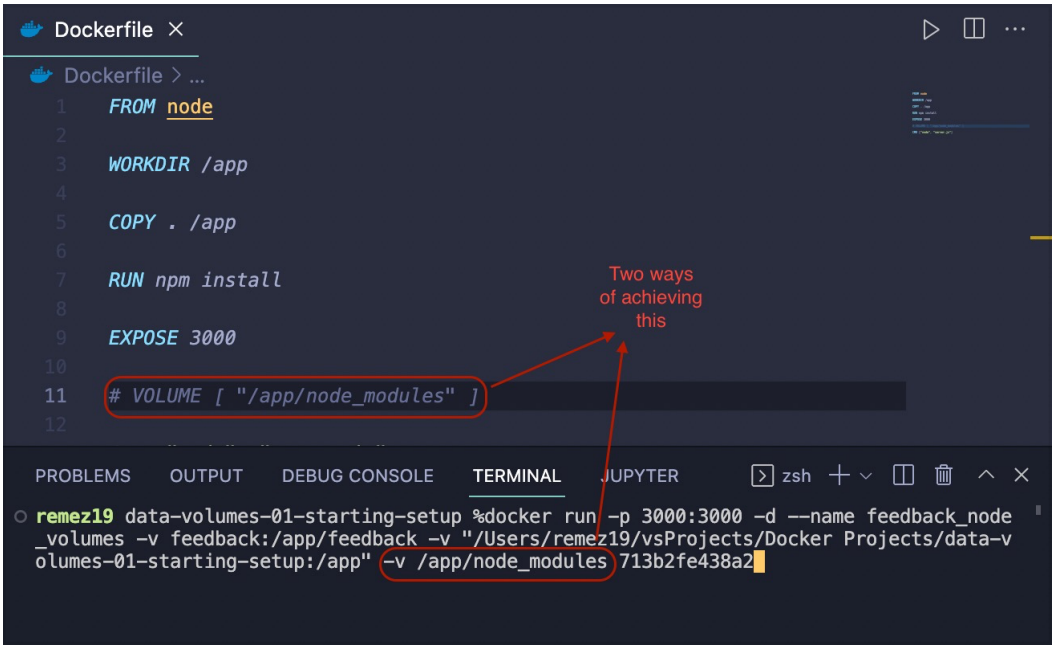
Great for source code during development.
Unlike volumes that managed by docker, Bind Mounts managed by us. We can configure which files in our local machine can be accessed by containers (read-

write privilege)

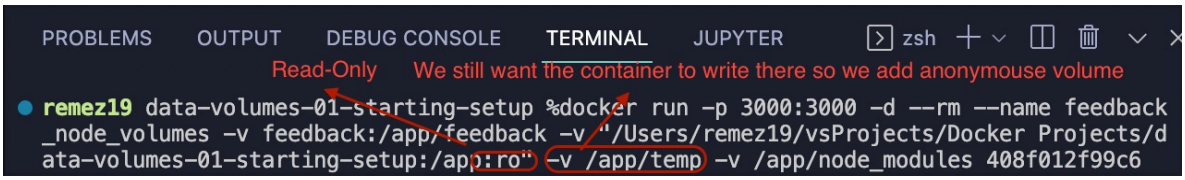




With this we actually delete (override) our app folder in the container but, this folder has “node_modules” folder (in this case) that we need.
In cases like this we can:



feedback_node_volumes:
In cases where we want to make volumes read-only by containers:



Managing Docker Volumes:

```
Commands:
  create      Create a volume
  inspect     Display detailed information on one or more volumes
  ls          List volumes
  prune       Remove all unused local volumes
  rm          Remove one or more volumes
```

Using "COPY" vs Bind Mounts:
When using Bind mounts in development we can give up the copy command in the docker file.
In production we have to have the copy command !

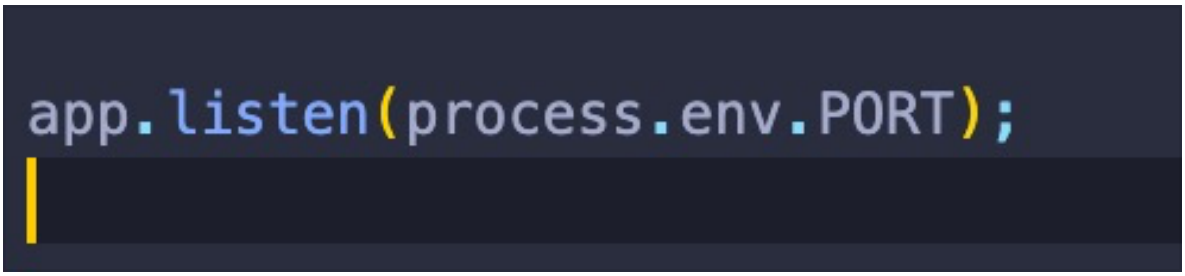
Don't COPY Everything: Using "dockerignore" Files:
We can add a docker ignore file to not include files - https://docs.docker.com/engine/reference/builder/#dockerignore-file (More Information!)
You can add more "to-be-ignored" files and folders to your .dockerignore file.
For example, consider adding the following to entries:

- Dockerfile
- .git

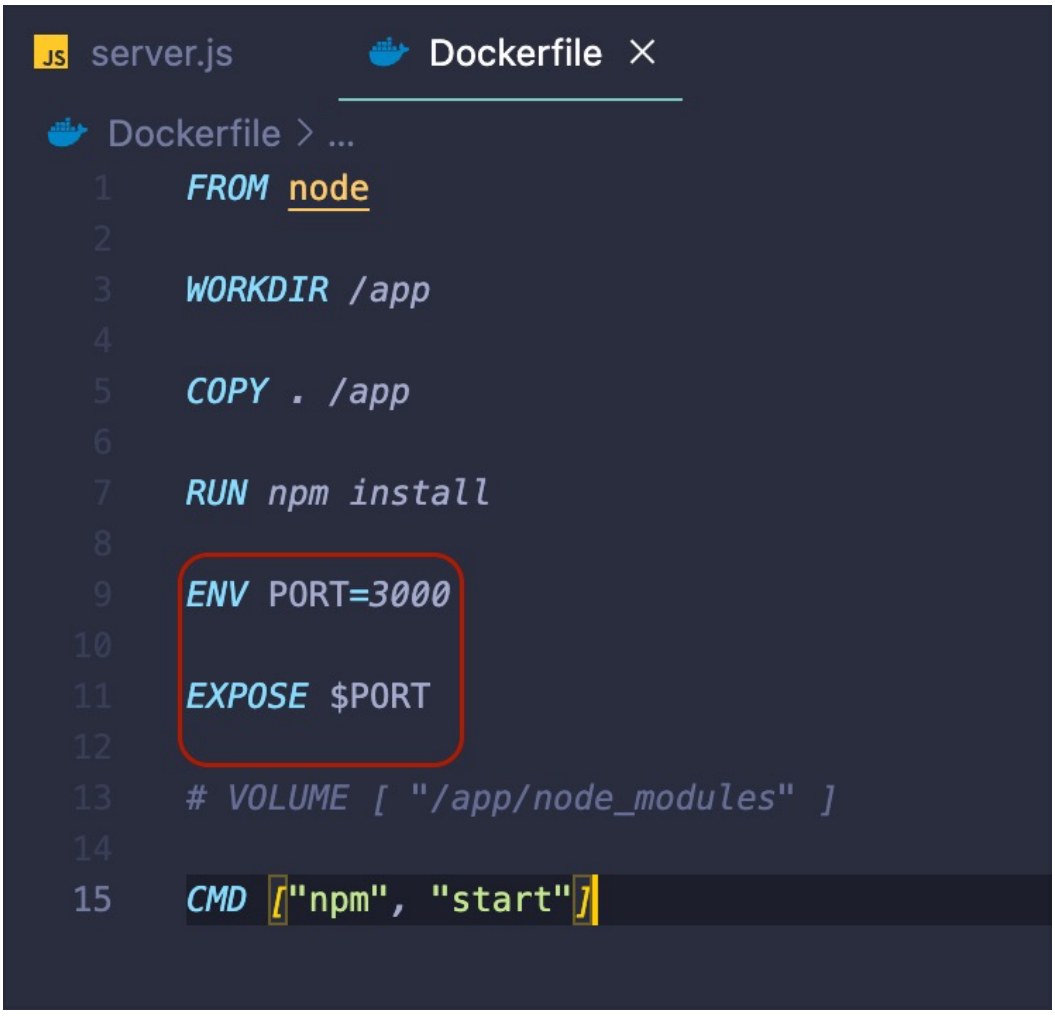
This would ignore the Dockerfile itself as well as a potentially existing .git folder

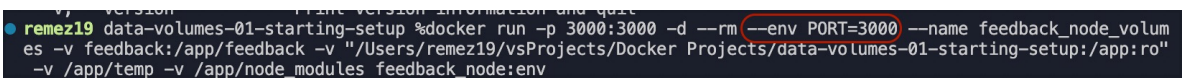
(if you are using Git in your project).
In general, you want to add anything which isn't required by your application to execute correctly.

Getting access to environment variables in NodeJs.

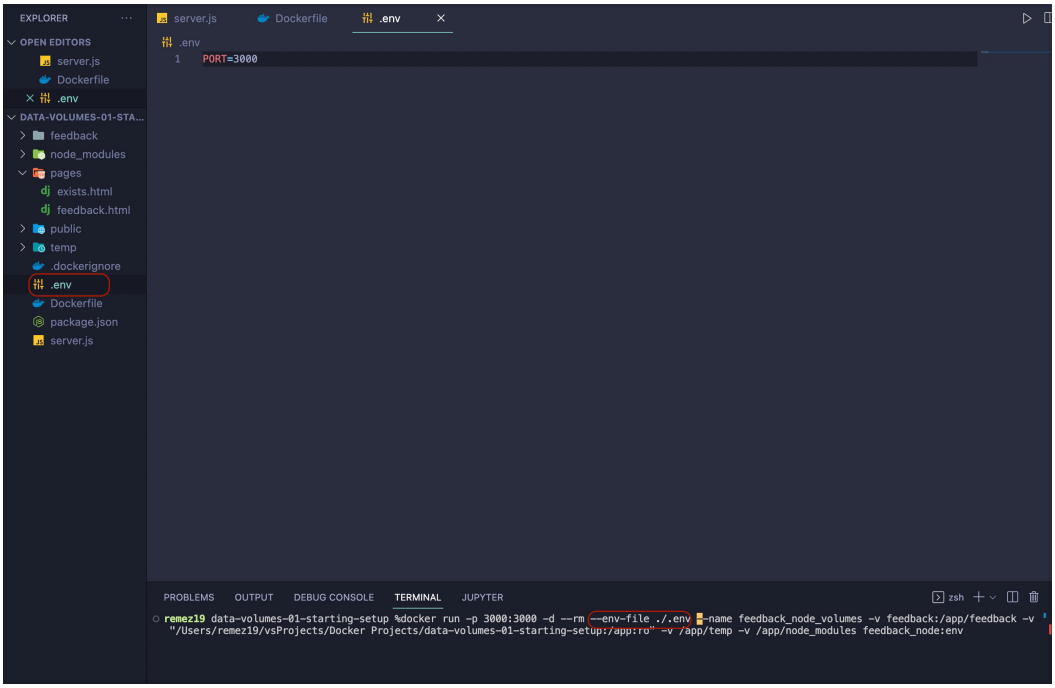


Providing the environment variable to the the app via the docker file





Using an environment file.
If we want docker to get our environment variables from a file. (Key-Value pairs)



Using Build Arguments (ARG):

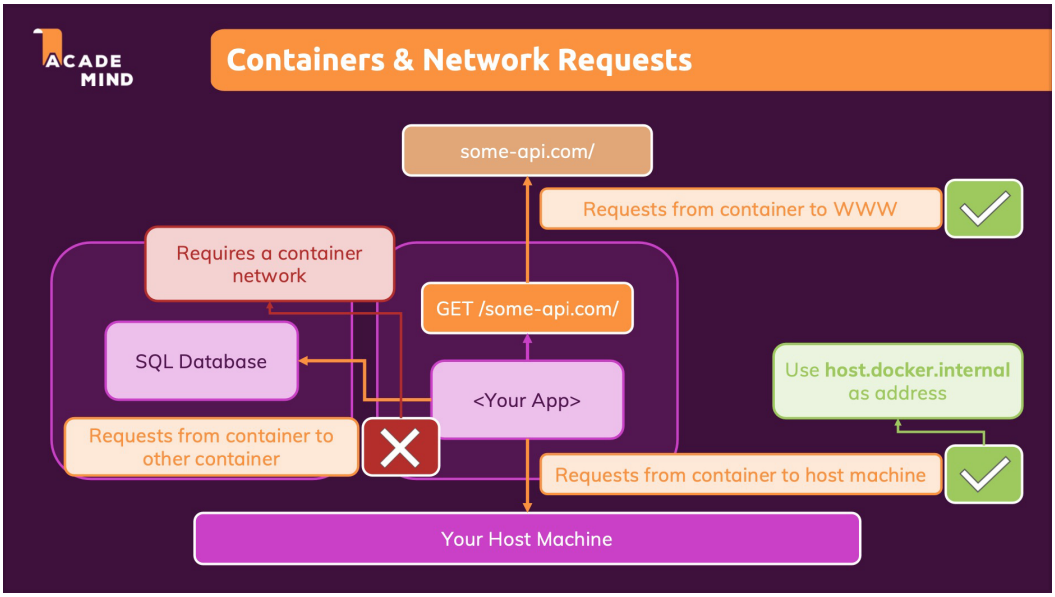
Setup arg in the docker file with **ARG**

Section4: Networking: (Cross-)Container Communication:

Types of communication:

There are three types of communication:

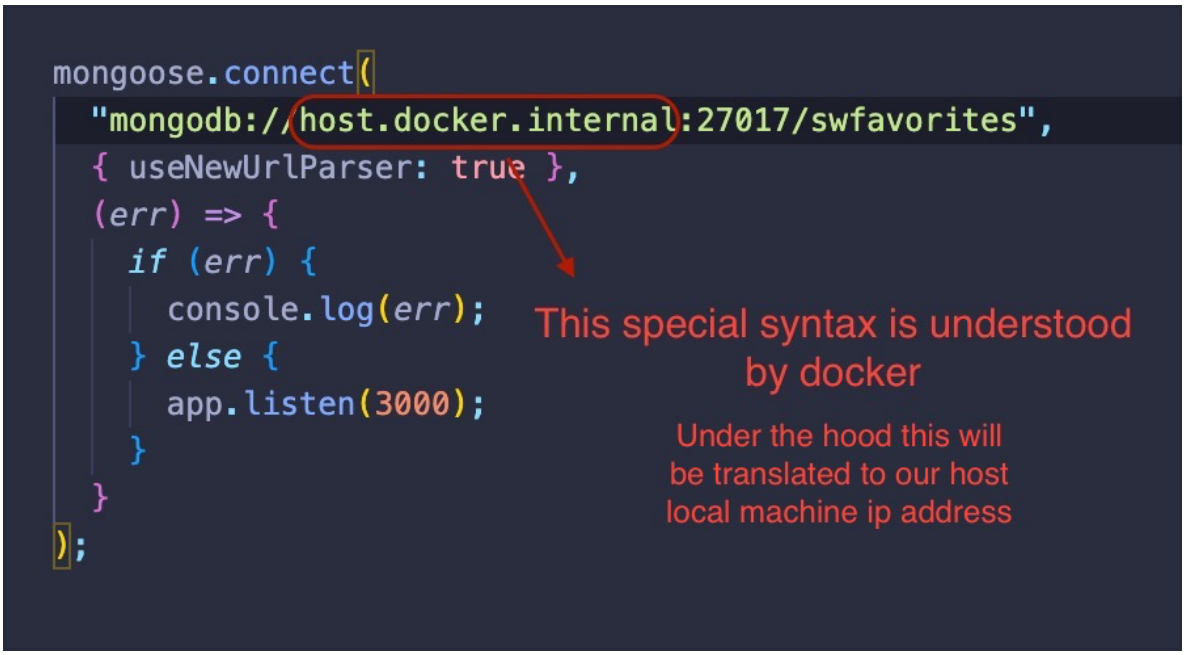
- Container to WWW communication (Like http request to api to fetch data).
- Container to Local Host Machine Communication (data base).
- Container to Container Communication.



Creating a Container & Communicating to the Web (WWW):

Out of the box dockerised apps (containers) can communicate with the web. Therefore sending http requests from a container to a web api is available.

Making Container to Host Communication Work:



Container to Container Communication: A Basic Solution:

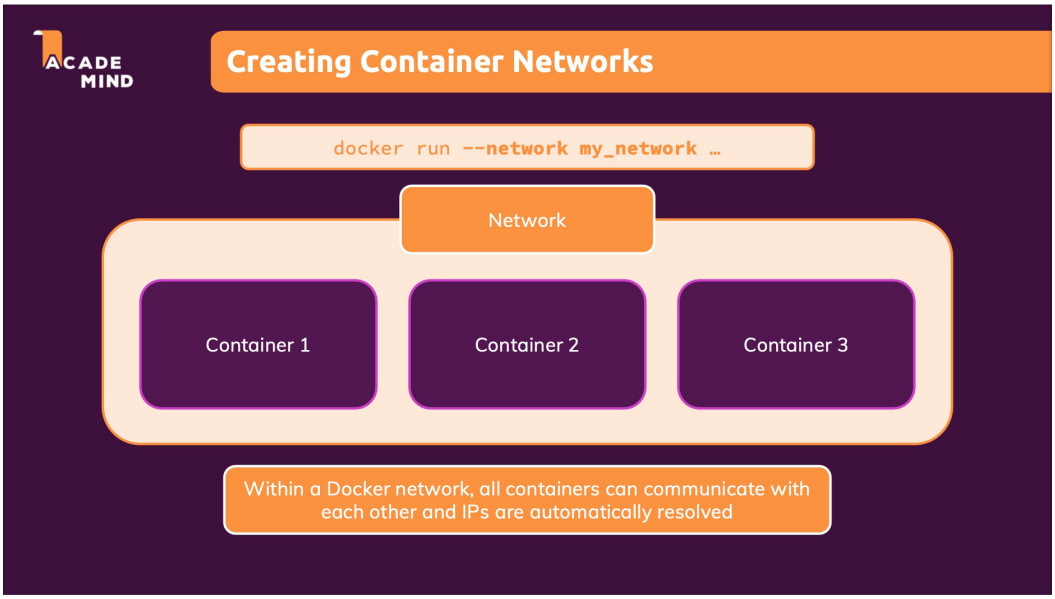
For the example of container to container communication we use the mongo image from docker hub.
This image will run a mongoldb server on are local machine inside a container of course.
To use the image we run - **“docker run mongo”**
Now the “host.docker.internal” will not work.
We need the ip of the running mongoldb container and nor the host machine ip.
With “docker inspect CONTAINER_NAME” we can get information about the running container.

```
            "Labels": {}
        },
        "NetworkSettings": {
            "Bridge": "",
            "SandboxID": "d03e96096f1730065ead8a6f4e9463bb5b7a6e10bf647bad02b597539
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {
                "27017/tcp": null
            },
            "SandboxKey": "/var/run/docker/netns/d03e96096f17",
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "EndpointID": "14b6b74a952d448ea6bae8b2d02e071e0ba162813525ec3a0f40839f
            "Gateway": "172.17.0.1",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "172.17.0.2",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "MacAddress": "02:42:ac:11:00:02",
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID": "fa08e9f3b19444b6efb9df098286ffeec2fee7f2f4f4dfb4a
```


With this the two containers communicate between them.

Introducing Docker Networks: Elegant Container to Container Communication:

With docker we can setup container network.
With docker we can put containers in the same network.
Using the —network.
This then creates a network in which all containers can talk to each other.
Docker automatically get the ip address.



In order to use this docker feature we need to create this network.
Using: “docker network create NETWORK_NAME” we create a new docker network

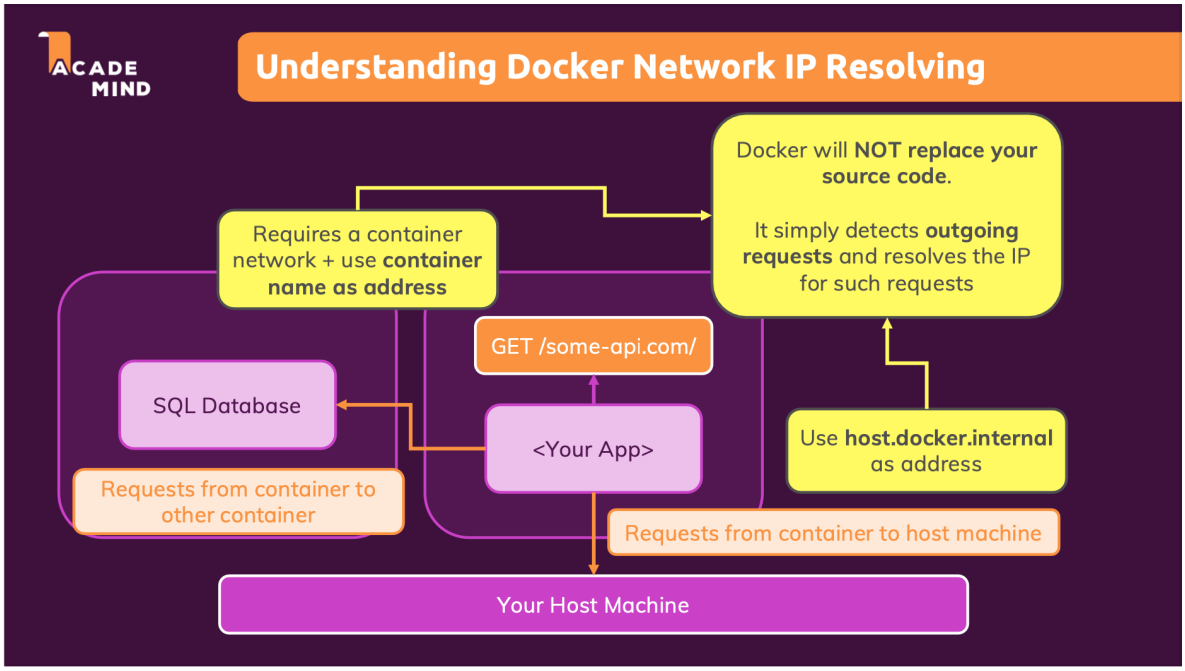
```
remez19 networks-starting-setup %docker network create favorites-net
```

It’s a docker internal network which we then can use on docker containers to let them talk to each other.
Creating a container that uses our network “favorites-net”.
Any containers with the same network can talk to each-other .

```
remez19 networks-starting-setup %docker run -d --name mongodb --network favorites-net mongo
```

Now we just need to put the container name that we want to talk to.
The name of the container will be translated by docker to ip.
And this will work if the containers are in the same network !

How Docker Resolves IP Addresses:

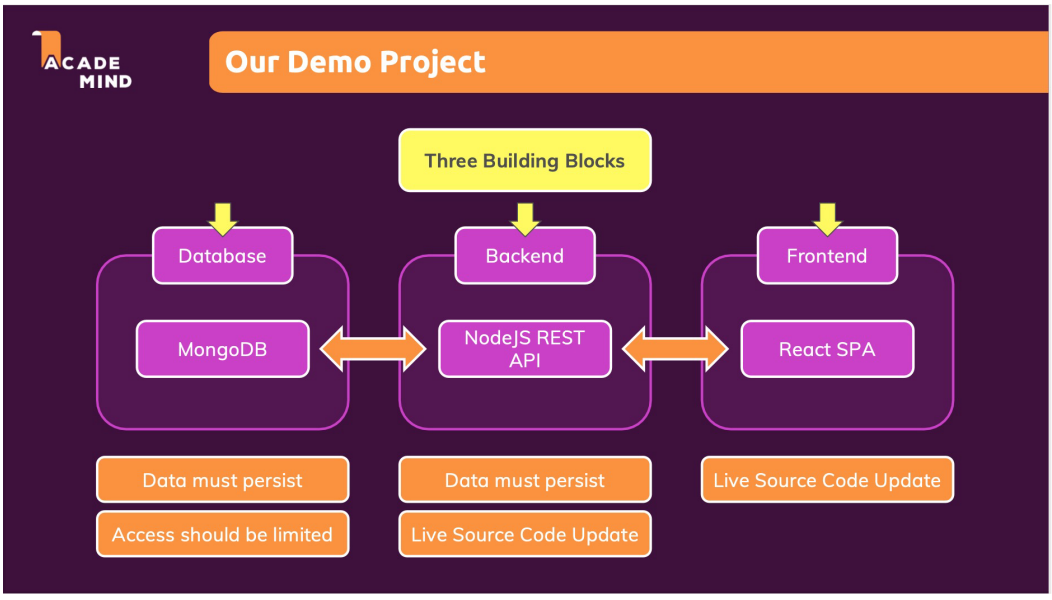


Section 5: Building Multi-Container Application with Docker:

Our Target App & Setup:

Three parts of our application:

- Backend - Nodejs.
- Frontend - React.
- Database - MongoDb.



Adding Data Persistence to MongoDB with Volumes:

```
remez19 multi-01-starting-setup %docker run --name mongodb -v data:/data/db -d --net
work full-stack-net mongo
```

Volumes, Bind Mounts & Polishing for the NodeJS Container:

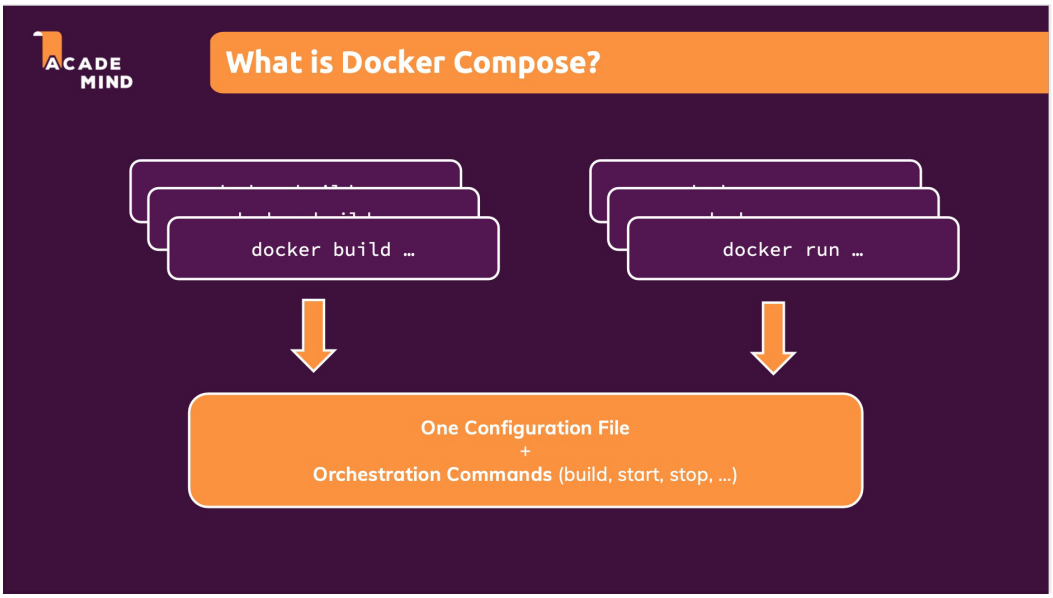
```
remez19 multi-01-starting-setup %docker run --name backend-app -d -p 80:80 -v logs:/
app/logs -v "/Users/remez19/vsProjects/Docker Projects/multi-01-starting-setup/backe
nd:/app:ro" -v /app/node_modules --network full-stack-net backend-node
```

For live changes in our code we use nodemon to help us !

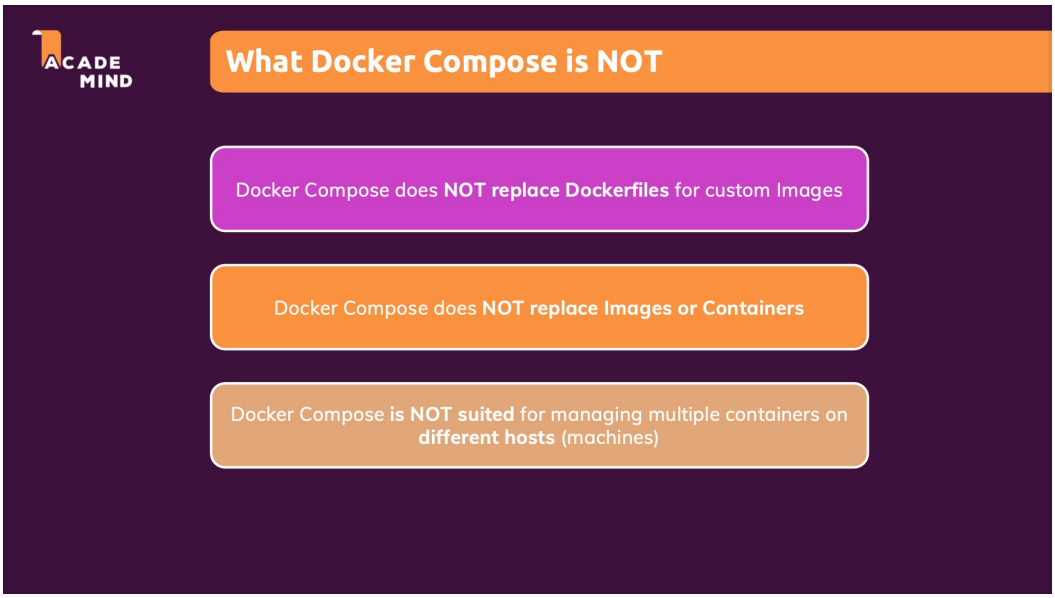
Live Source Code Updates for the React Container (with Bind Mounts):
React container command:
docker run --name frontend-app -it -p 3000:3000 -v"/Users/remez19/
vsProjects/Docker Projects/multi-01-starting-setup/frontend/src:/frontend" -v /
frontend/node_
modules frontend-react

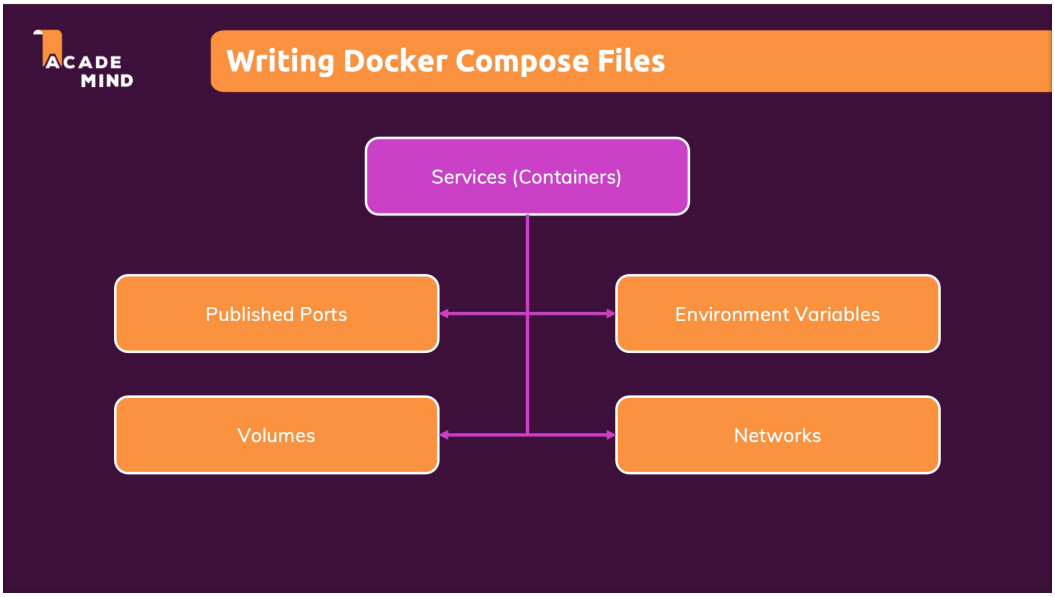
Section 6: Docker Compose: Elegant Multi-Container Orchestration:
Using Docker compose Makes the managing of a multi-container setup easier !

Docker-Compose: What & Why?
Docker Compose enable us to use one configuration file that can holds all the docker basic commands such as “build” and “run”.
Eventually we will use one command to start our multi container app. (Time consuming)



Docker compose is great for managing multiple containers on the same host!





Creating a Compose File:

We can create a docker compose file by, creating a file with the name “docker-compose.yaml” we end the file with “.yaml” or “.yml” .
In this file we describe our multi container environment - our project setup.
All the specifications of a docker compose file:
https://docs.docker.com/compose/compose-file/

Diving into the Compose File Configuration:

Docker Compose Up & Down:

To use the docker-compose file we created we need to run in the terminal (**To start all services-containers**):

"**docker-compose up**" we can also use "docker-compose up -d" to start in detach mode .

To stop all services and containers:

"Docker-compose down". Stops and remove all containers, services and, networks but, not the volumes.

To stop all services and containers and remove volumes:

“Docker-compose down -v”

Working with Multiple Containers:

Adding Another Container:

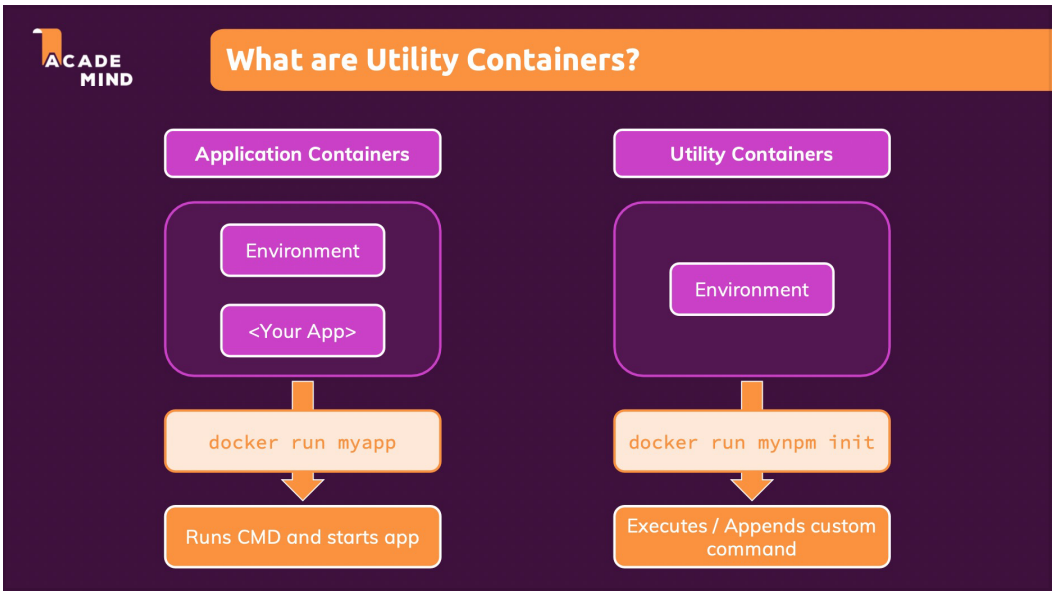
Adding the frontend app setup.

Building Images & Understanding Container Names:

By adding the "**—build**" option to the "docker-compose up" command ("**docker-compose up —build**") you force that images are rebuilt.
Else docker will always use the already existing image (from the first up command).
If we just want to build any custom image defined in the docker-compose file we can:
"**docker-compose build**" - will only build the images and not run containers on them.

To name a container we can use the “**container_name: name**” in the docker-compose file.

Module Introduction & What are "Utility Containers”?:



Utility Containers: Why would you use them?:

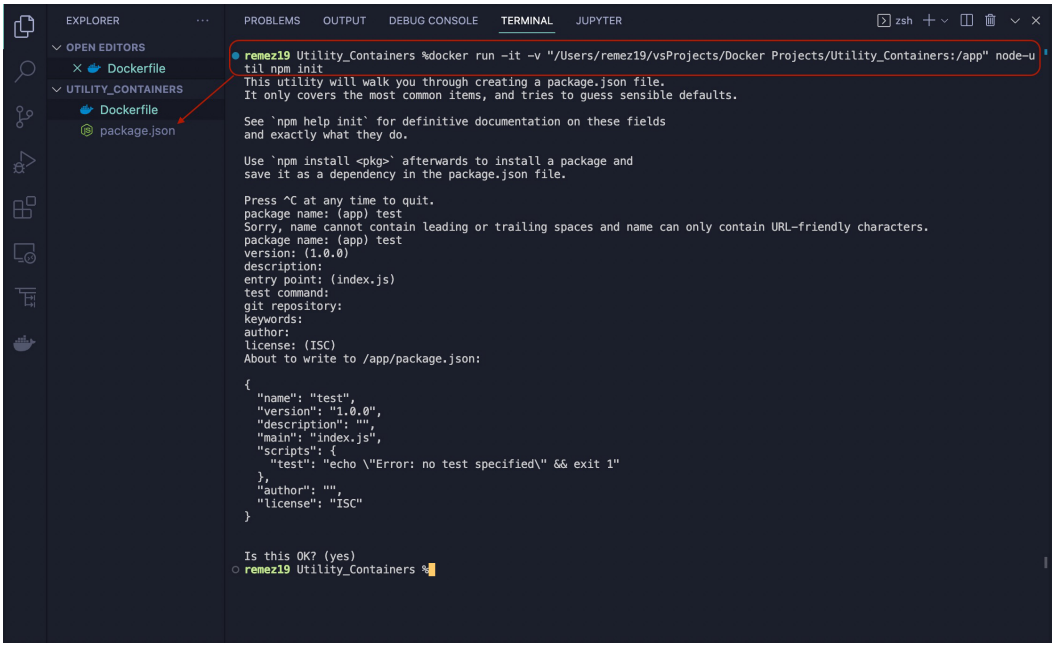
Somer projects have dependencies which require us to download third party liberties or frameworks to even start creating projects
For example Nodejs.
Utility containers helps us in those situations.

Different Ways of Running Commands in Containers:

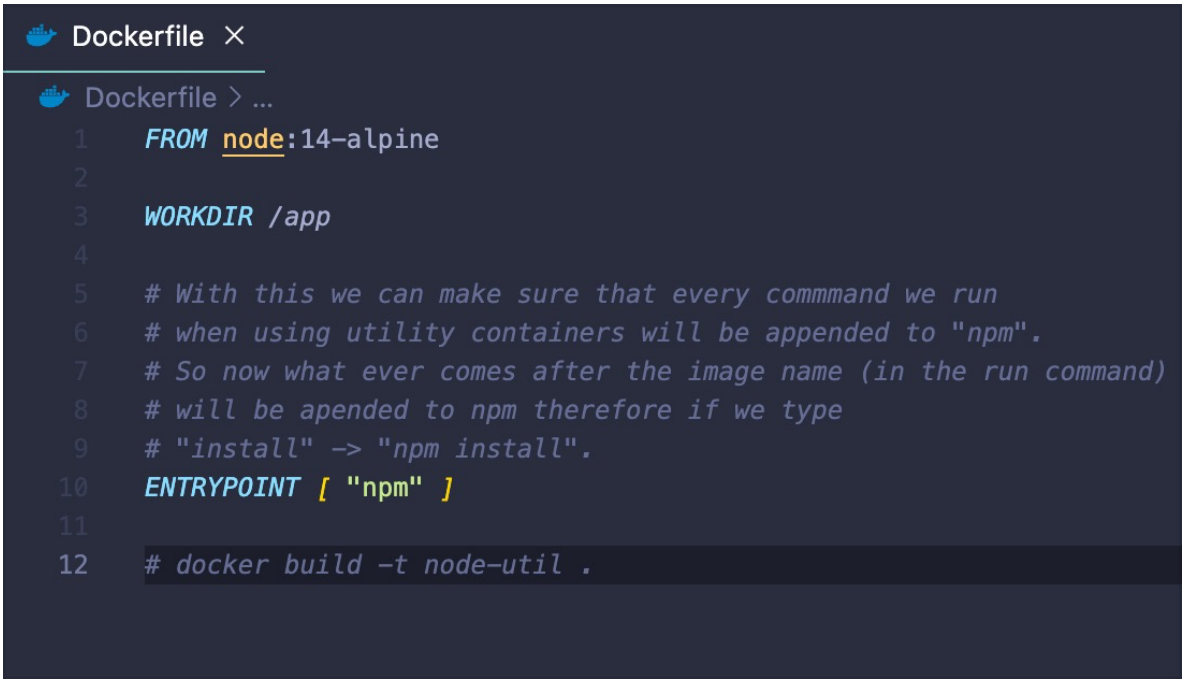
“docker exec” - let us execute certain commands inside a running container besides the default command the container execute.
“**docker exec -it agitated_archimedes npm init**”
Overriding a default command of a container - “run -it node npm init
“ (Overriding the default command of the node image with npm init)

Building a First Utility Container:

In this example we run node in a utility container and using the container as environment for project in our local machine.
By making a bind mount to a folder in our local machine and to the working directory inside the container.
“**docker run -it -v "/Users/remez19/vsProjects/Docker Projects/Utility_Containers:/app" node-util npm init**”

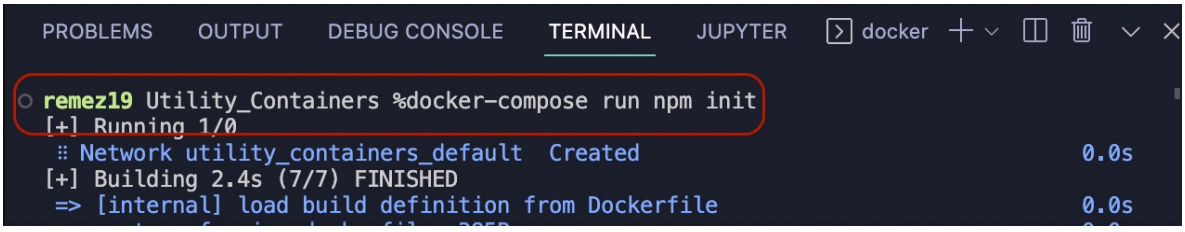


Utilizing ENTRYPOINT:



Using Docker Compose:

Using docker compose to achieve the same result:

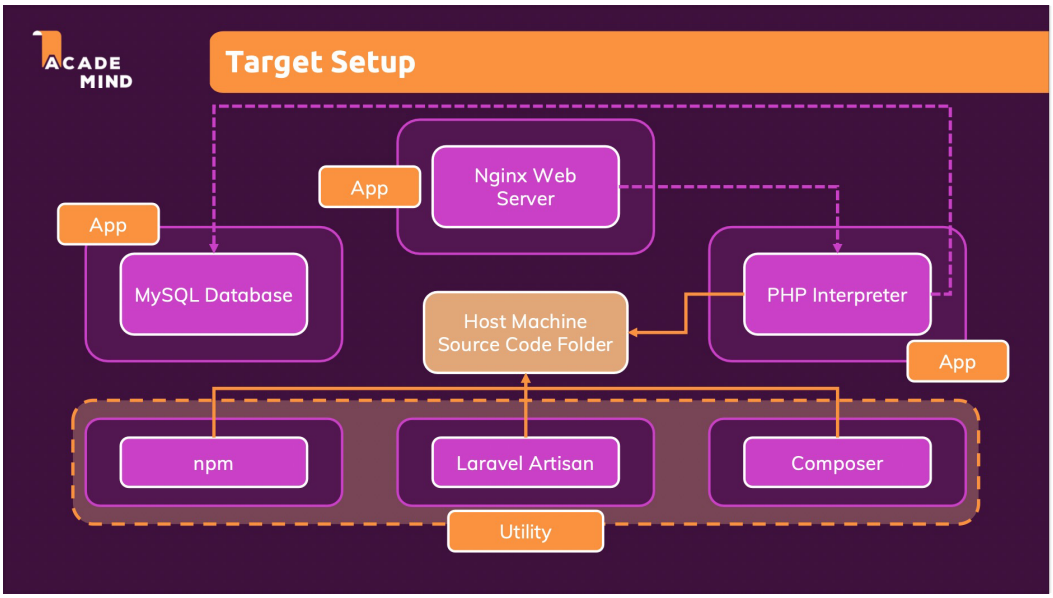


docker-compose.yaml

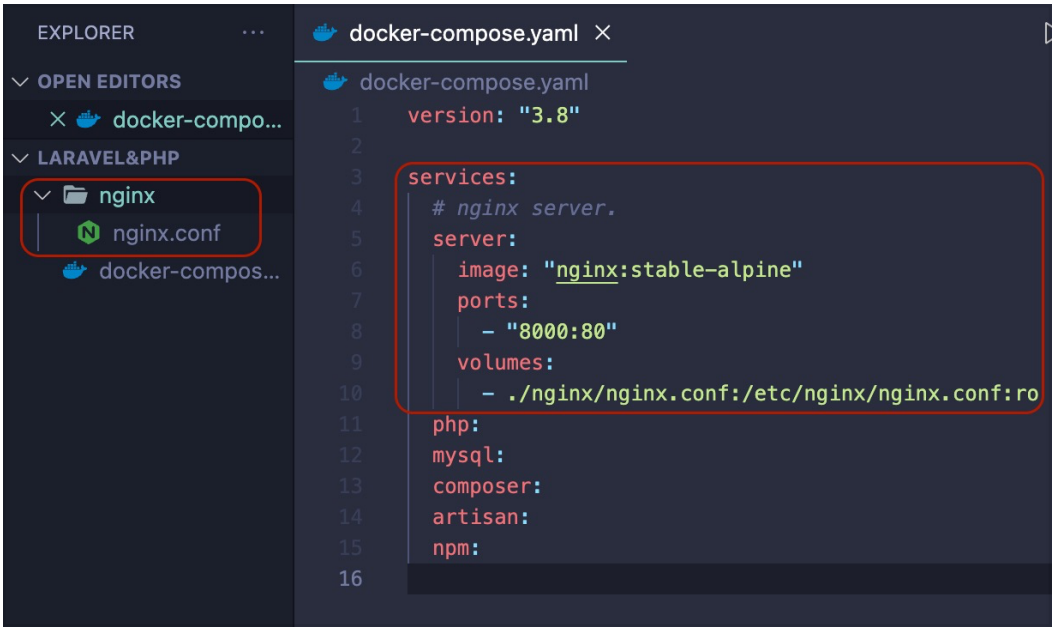
```yaml
version: "3.8"
services:
  npm:
    build: ./
    stdin_open: true
    tty: true
    volumes:
      - ./:/app
```

Section 8: A More Complex Setup: A Laravel & PHP Dockerized Project:

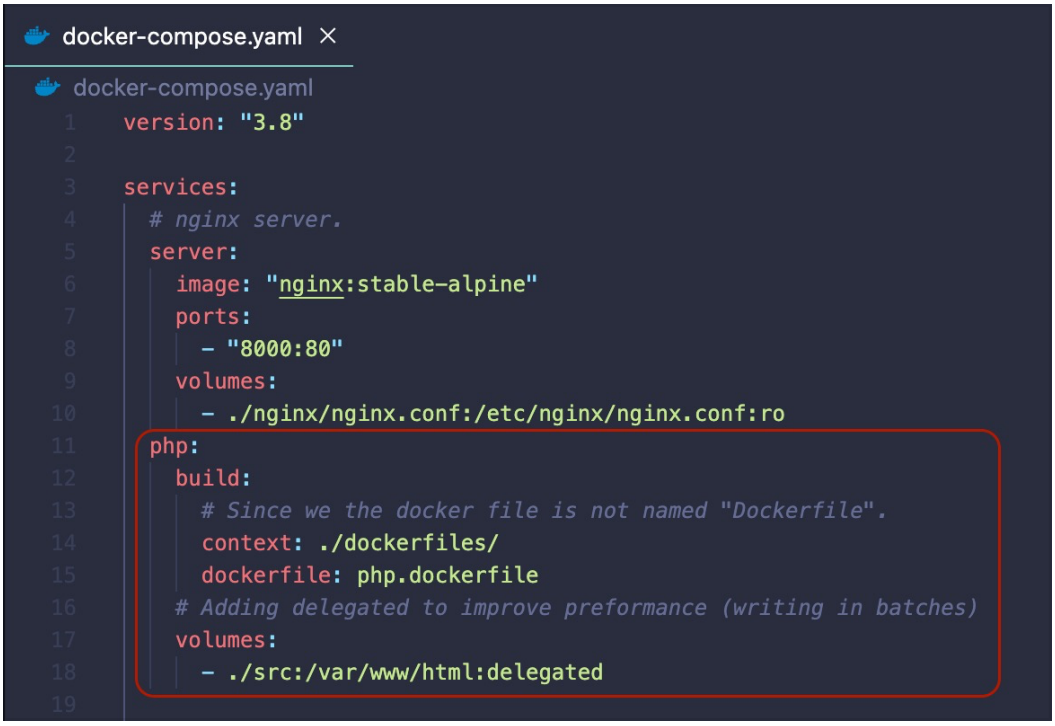
The Target Setup:



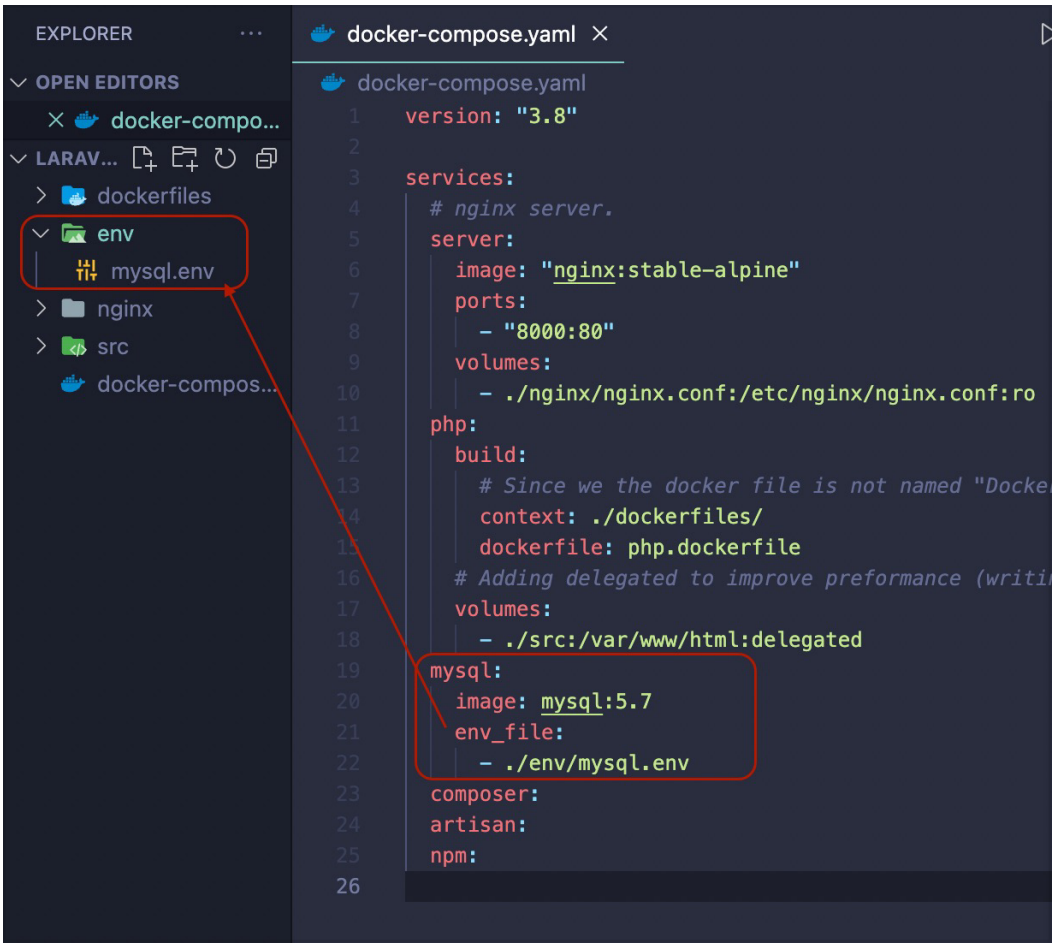
Adding a Nginx (Web Server) Container:



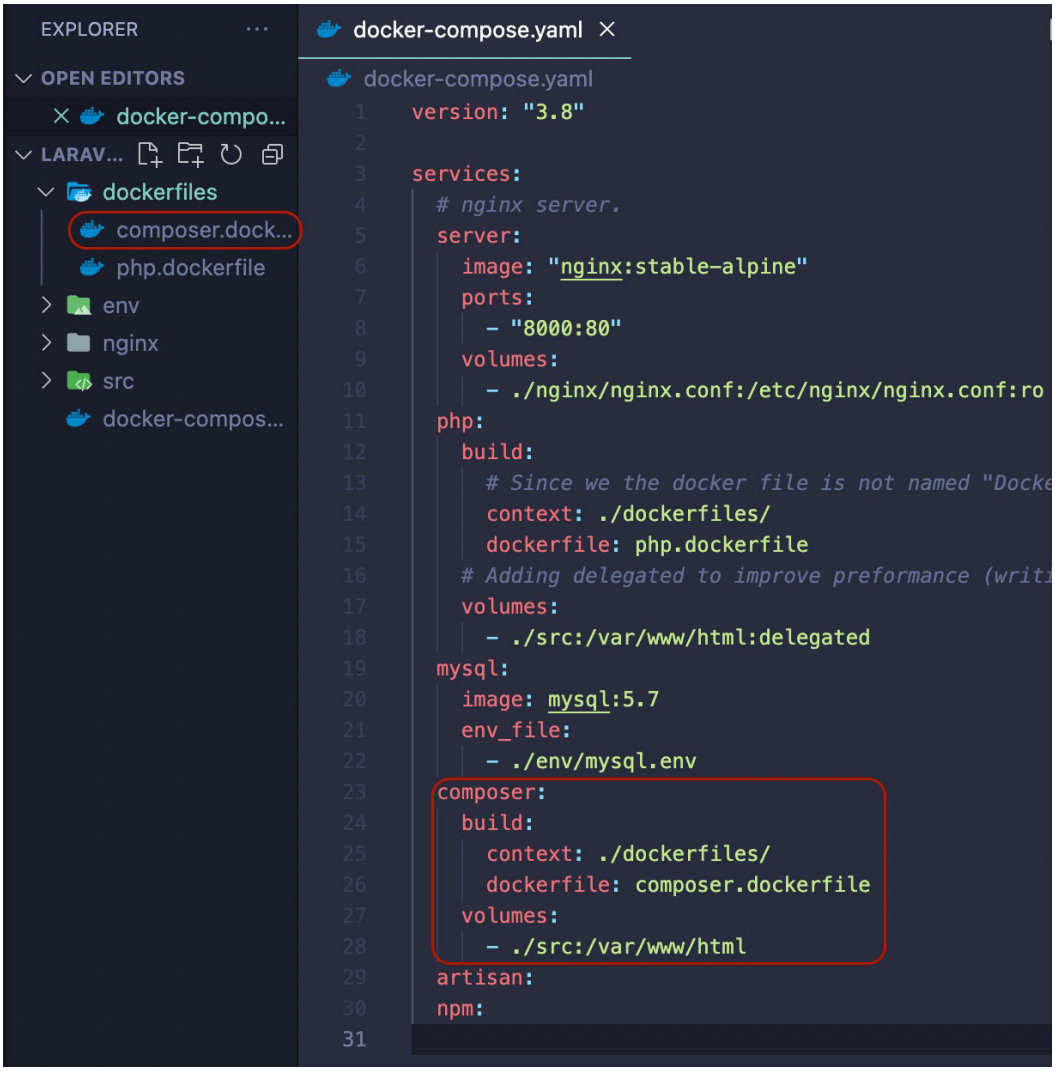
Adding a PHP Container:



Adding a MySQL Container:



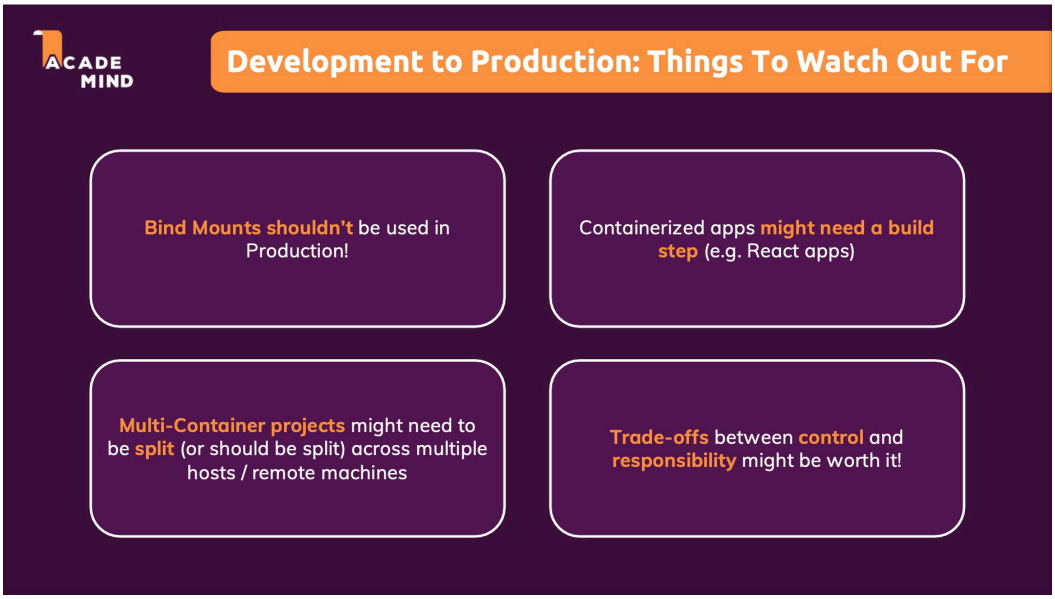
Adding a Composer Utility Container:



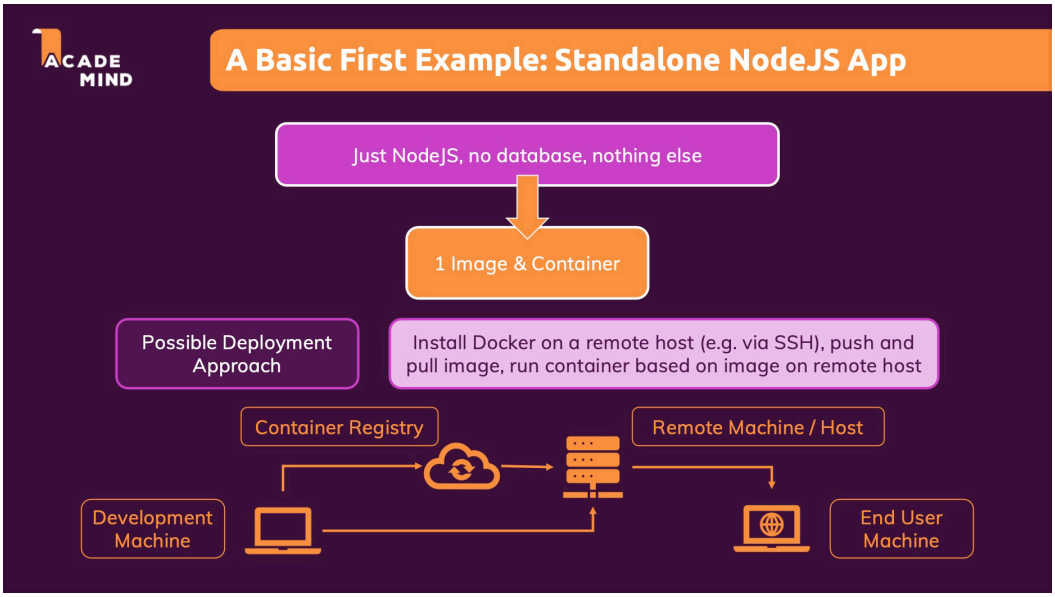
Creating a Laravel App via the Composer Utility Container:
Command - “docker-compose run --rm composer create-project --prefer-dist laravel/laravel .”

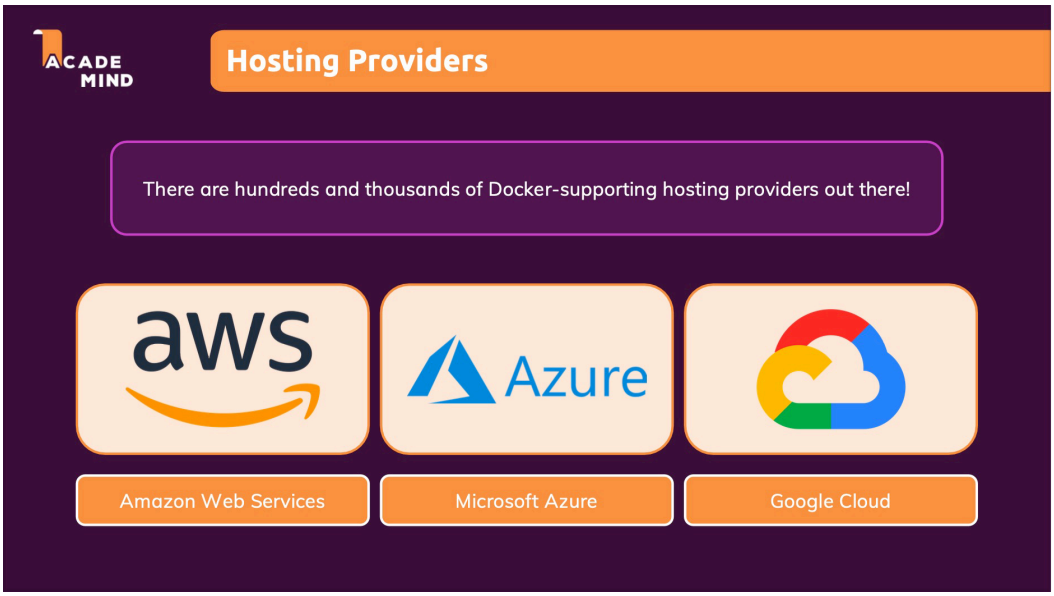
Section 9: Deploying Docker Containers:

From Development To Production:

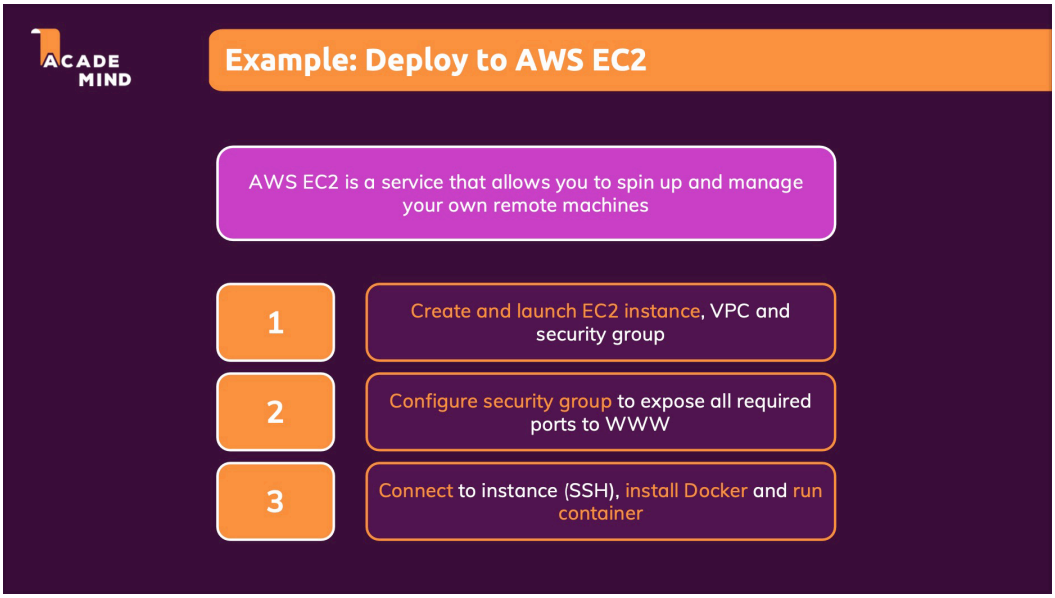


Deployment Process & Providers:

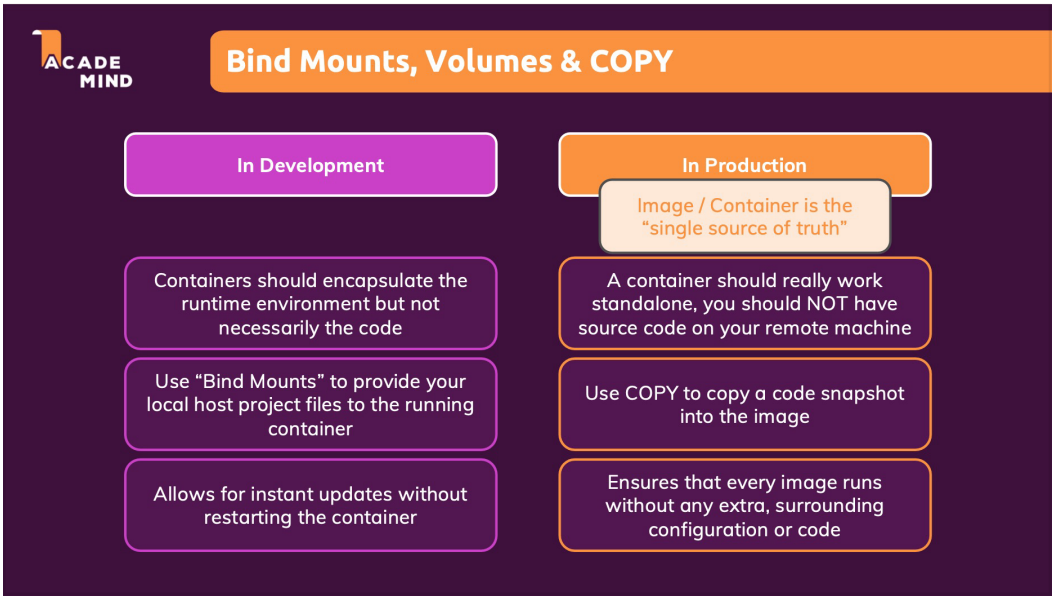




Getting Started With An Example:



Bind Mounts In Production:

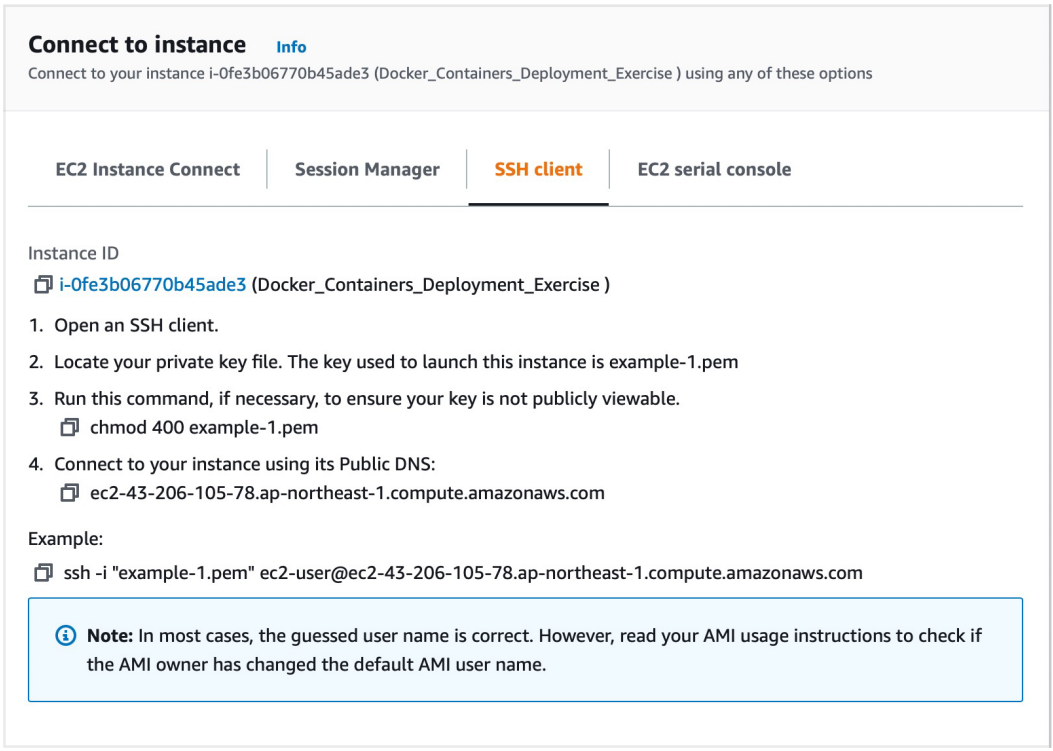


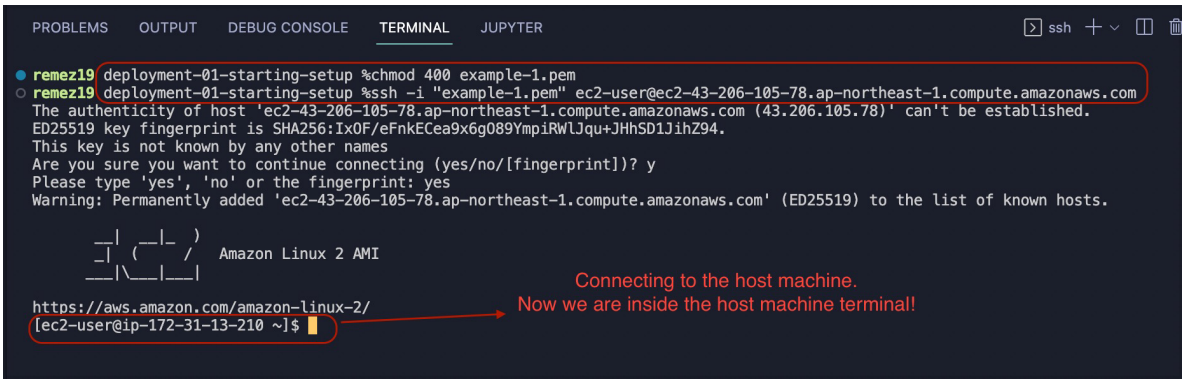
Introducing AWS & EC2:

All about this section in this links:
Lecture 129:
https://www.udemy.com/course/docker-kubernetes-the-practical-guide/learn/lecture/22626283#overview

Connecting to an EC2 Instance:

Connecting to our AWS instance by SSH (Secure shell).
It's a protocol for connecting from our local machine via the terminal of our local machine.





Installing Docker on a Virtual Machine:

Starting by typing: "**sudo yum update -y**"
Will make sure that all the essentials packages are updated (on the remote machine) and we are using the latest version.
Than we need to run: "**sudo amazon-linux-extras install docker**"
This will install docker on the remote machine.
To start docker on the remote machine: "**sudo service docker start**"
Now we will be able to run the docker commands such as "run"

Installing Docker on Linux in General:

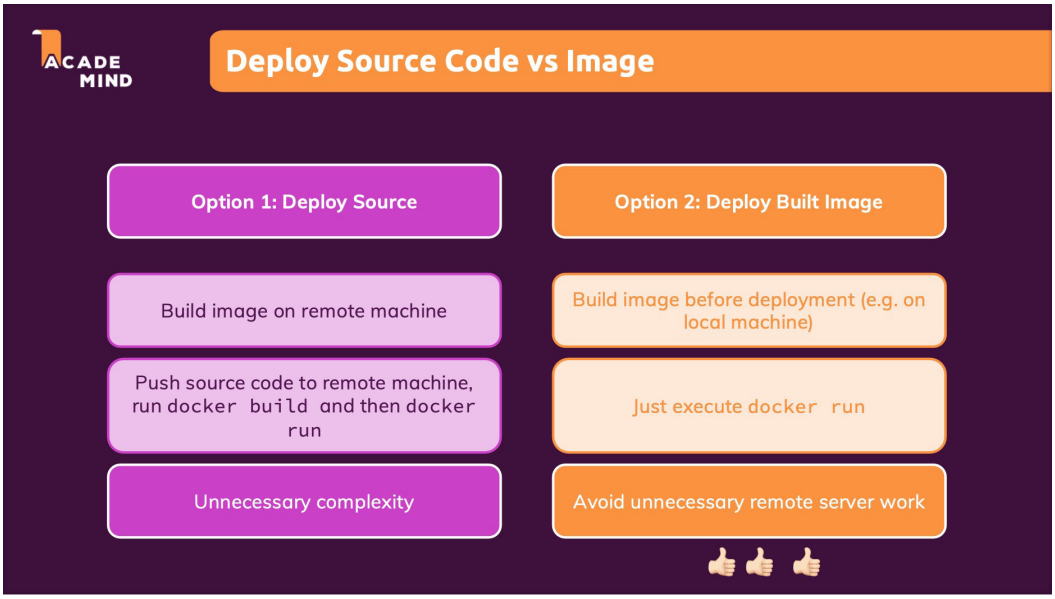
In the last lecture, you saw the AWS-specific command for installing Docker on a Linux machine:

- amazon-linux-extras install docker

Of course you might not always want to install it on a AWS EC2 instance though - maybe you are using a different provider.
In that case, you can always follow the Linux setup instructions you find on the official Docker page: https://docs.docker.com/engine/install/ (under "Server").

Pushing our local Image to the Cloud:

Two main options:
1. We finding a way to copy our source code to the host machine and, then we build it there.
2. We build our image a head of time, for example, on our local machine and, then we deploy that image on the host machine.



Running & Publishing the App (on EC2):

Building image correctly on the M1 - "**docker buildx build --platform linux/amd64 -t node-dep-example-1 .**"
This will make sure that our image will run on the host machine.
After we push our image to docker hub we can run it on the host machine:

```
[ec2-user@ip-172-31-13-210 ~]$ sudo docker run -d --rm -p 80:80 remez19/node-exam
ple-1-docker-deployment
```

Managing & Updating the Container / Image:

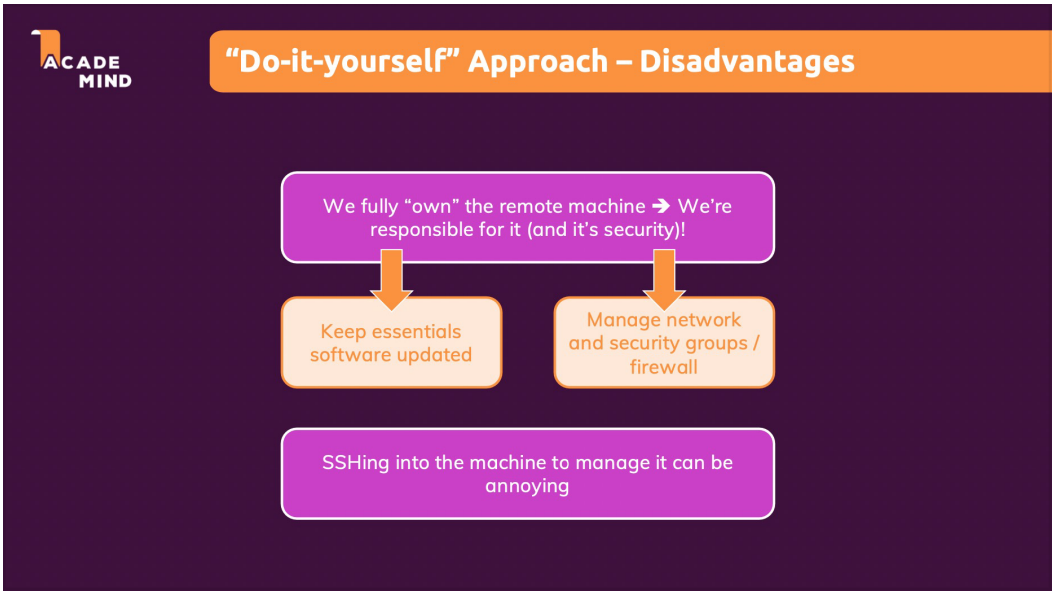
To Make sure that we see the changes we making in our src code we need to go through all the steps again:

1. Create/Build an image on our local machine - "docker build -t MY_IMAGE_NAME."
2. Tag the image correctly ignorer to push it to docker hub - "docker tag stam remez19/node-example-1-docker-deployment:stam "
3. Push the image to Docker Hub - "docker push remez19/node-example-1-docker-deployment:stam".
4. Than we pull the image in the host machine - "docker pull remez19/node-example-1-docker-deployment:stam"
5. Finally we run a container inside the host machine based on the image

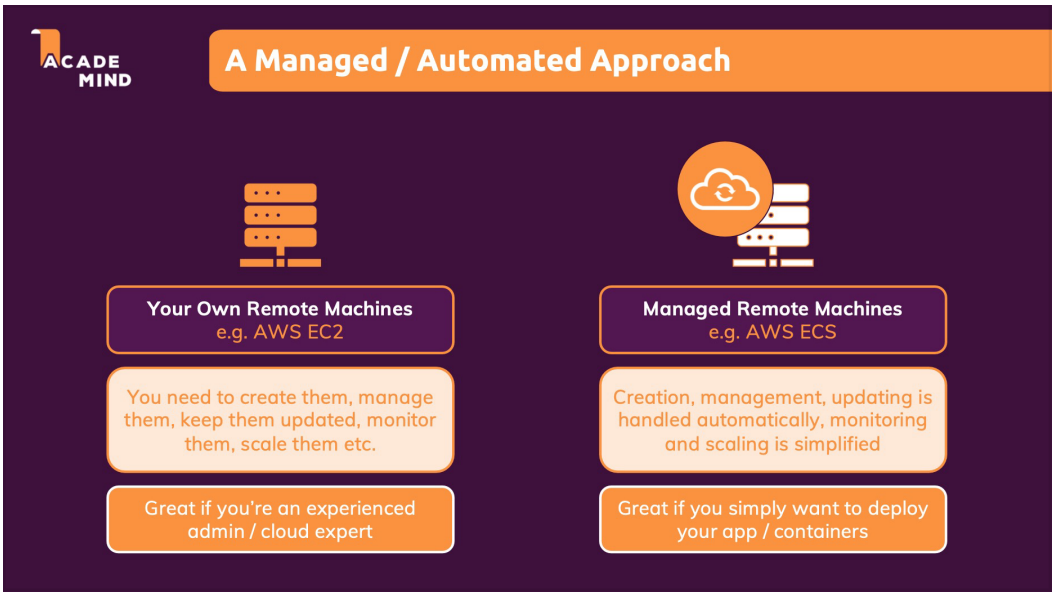
we pulled.

Because I'm using M1 - "docker buildx build --platform linux/amd64 "

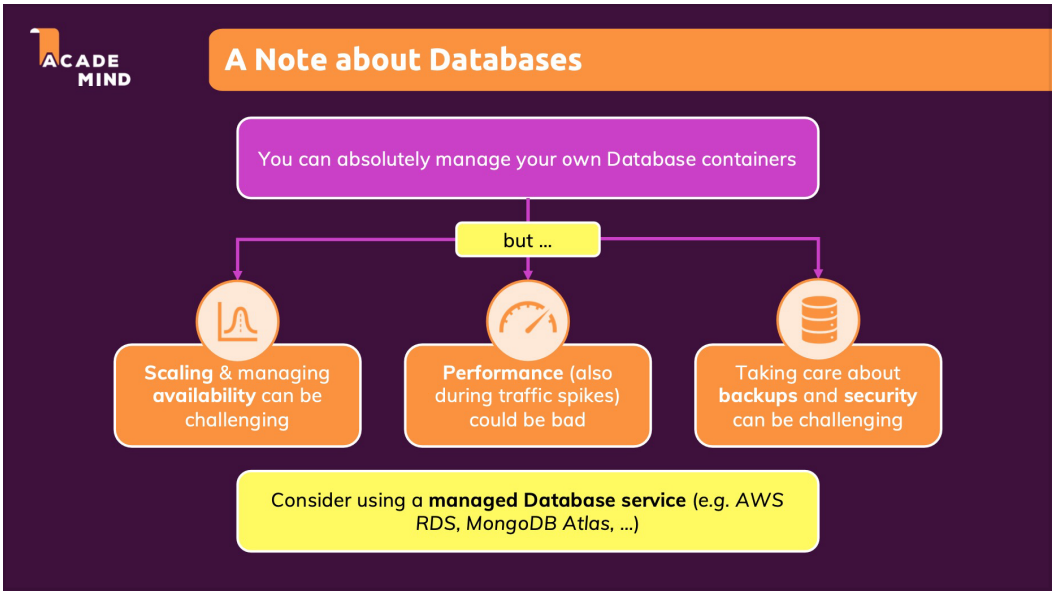
Disadvantages of our Current Approach:



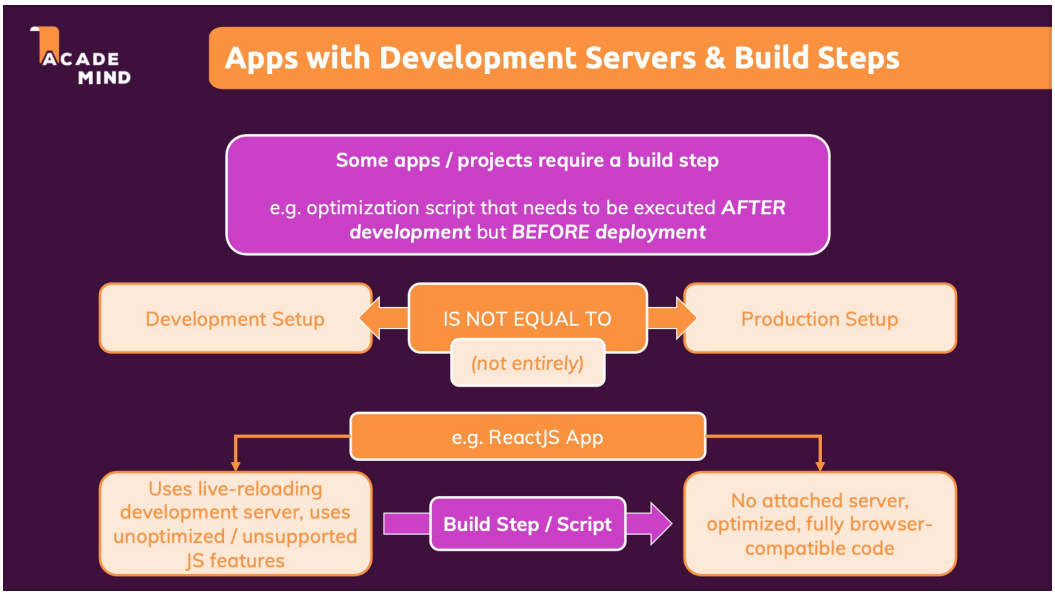
From Manual Deployment to Managed Services:



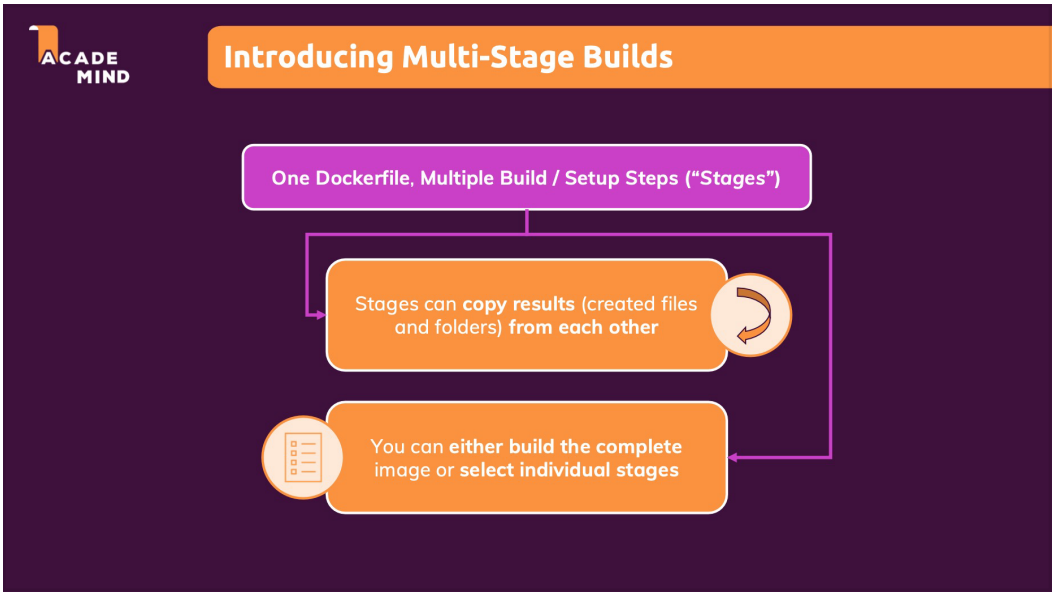
Databases & Containers: An Important Consideration:



Understanding a Common Problem:

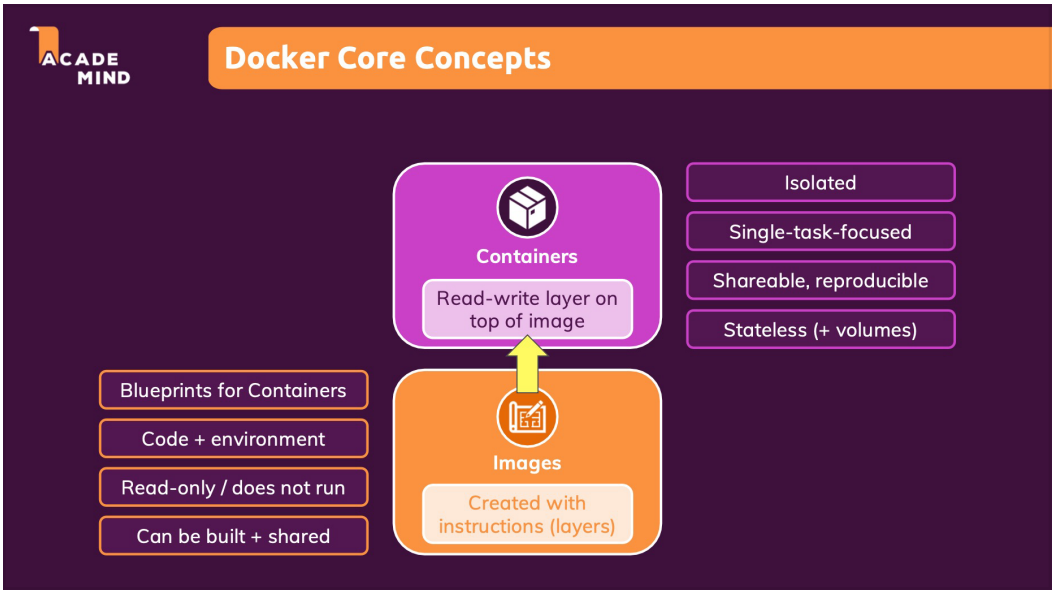


Introducing Multi-Stage Builds:

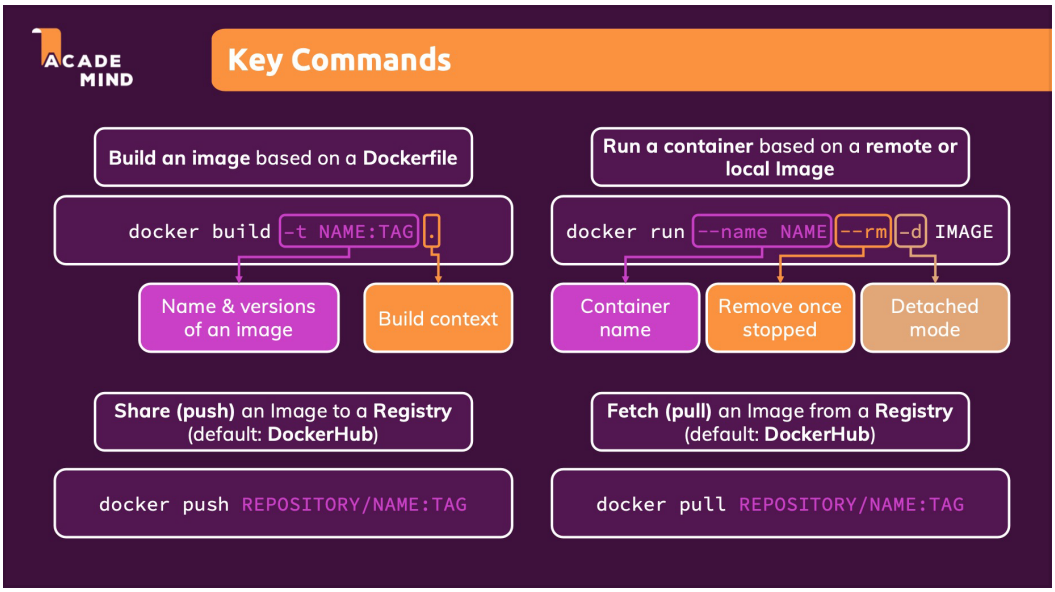


Section 10: Docker & Containers - A Summary:

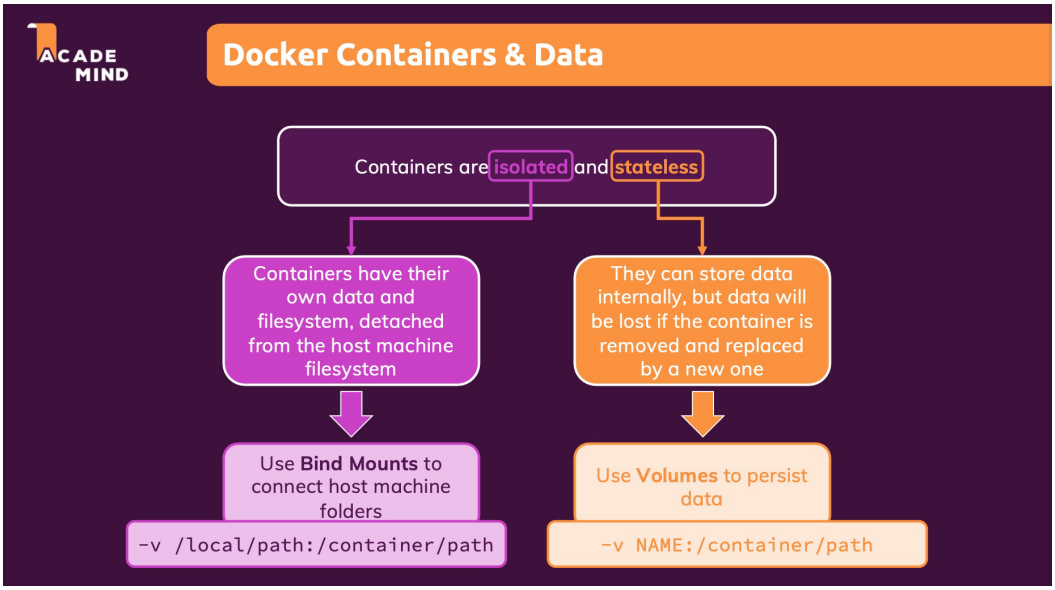
Images & Containers:

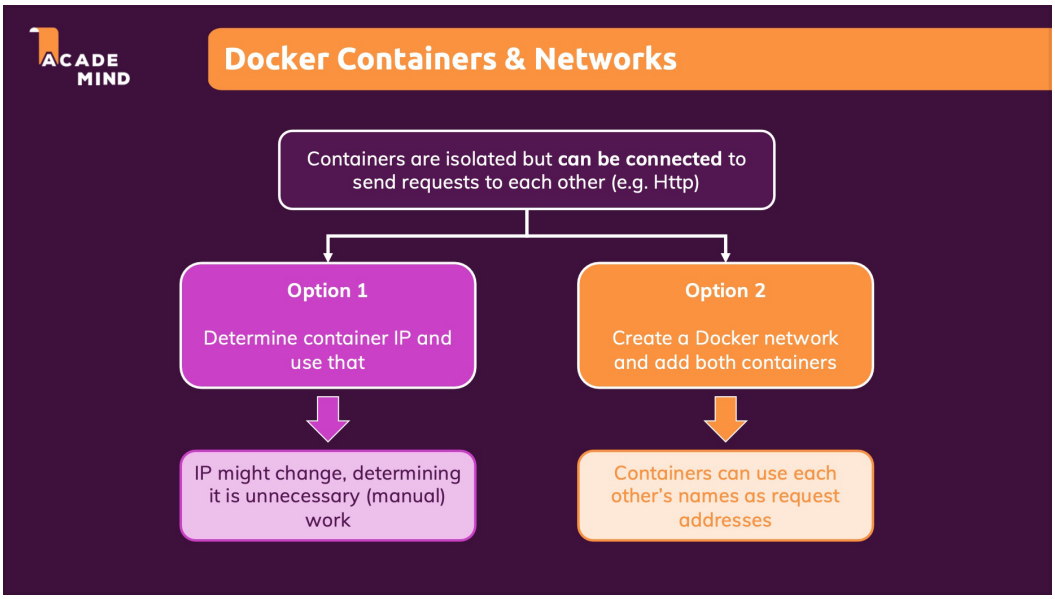


Key Commands:

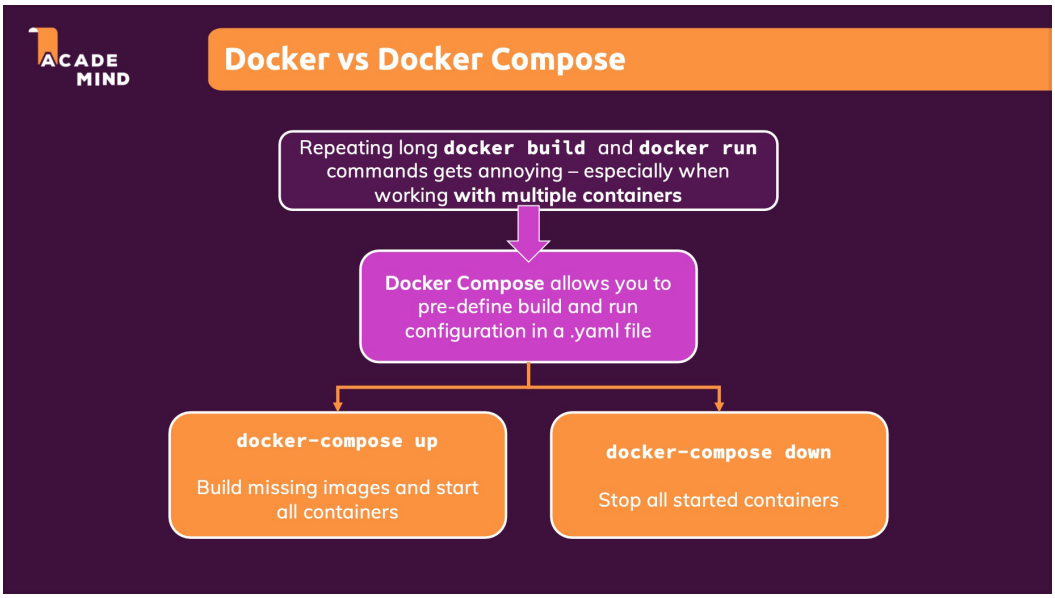


Data, Volumes & Networking:

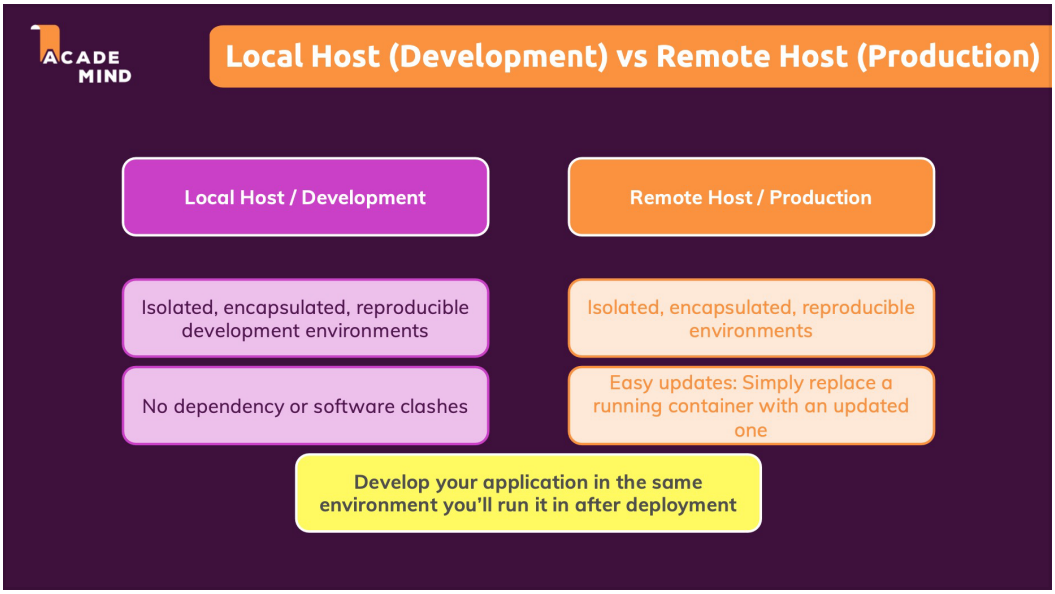




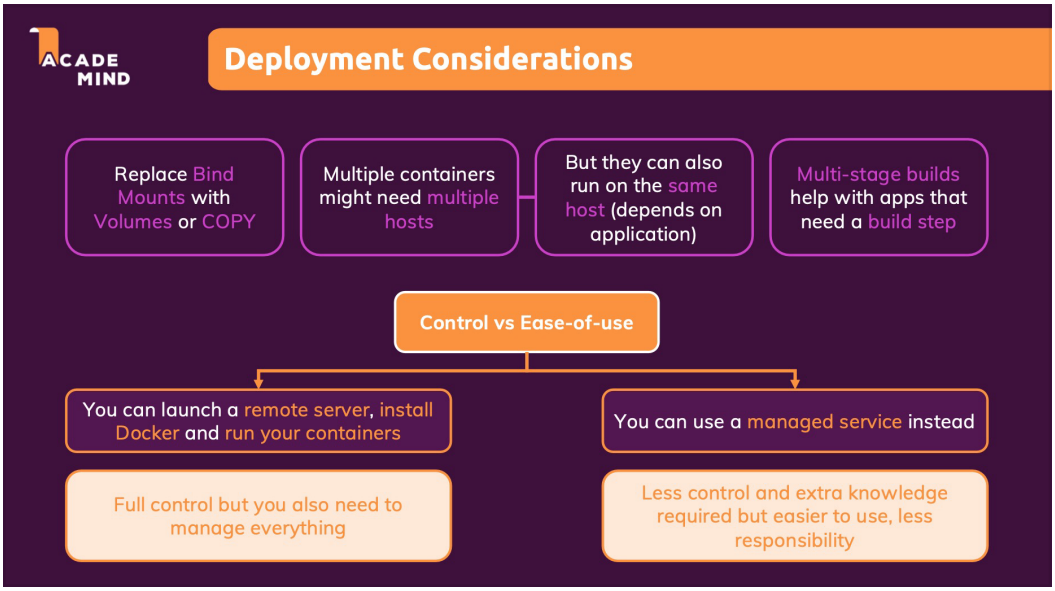
Docker Compose:



Local vs Remote:



Deployment:


ACADEMIND
Deployment Considerations
Replace Bind Mounts with Volumes or COPY
Multiple containers might need multiple hosts
But they can also run on the same host (depends on application)
Multi-stage builds help with apps that need a build step
Control vs Ease-of-use
You can launch a remote server, install Docker and run your containers
Full control but you also need to manage everything
You can use a managed service instead
Less control and extra knowledge required but easier to use, less responsibility