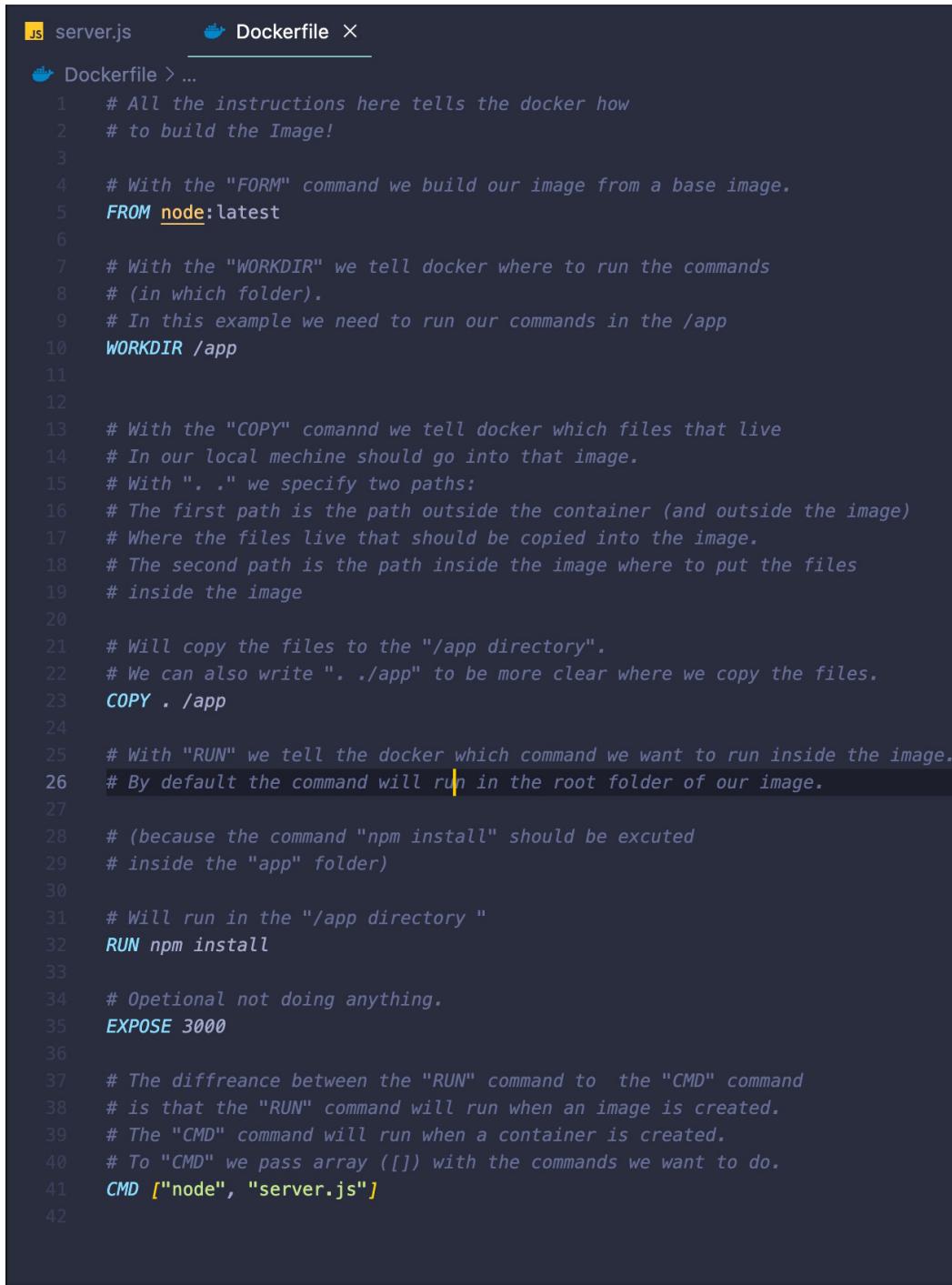


Docker Course

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

Building our own Image with a Dockerfile:



The screenshot shows a code editor with two tabs: "server.js" and "Dockerfile". The "Dockerfile" tab is active and displays the following content:

```
JS server.js Dockerfile X
Dockerfile > ...
1 # All the instructions here tells the docker how
2 # to build the Image!
3
4 # With the "FROM" command we build our image from a base image.
5 FROM node:latest
6
7 # With the "WORKDIR" we tell docker where to run the commands
8 # (in which folder).
9 # In this example we need to run our commands in the /app
10 WORKDIR /app
11
12
13 # With the "COPY" command we tell docker which files that live
14 # In our local machine should go into that image.
15 # With ". ." we specify two paths:
16 # The first path is the path outside the container (and outside the image)
17 # Where the files live that should be copied into the image.
18 # The second path is the path inside the image where to put the files
19 # inside the image
20
21 # Will copy the files to the "/app directory".
22 # We can also write ". ./app" to be more clear where we copy the files.
23 COPY . /app
24
25 # With "RUN" we tell the docker which command we want to run inside the image.
26 # By default the command will run in the root folder of our image.
27
28 # (because the command "npm install" should be executed
29 # inside the "app" folder)
30
31 # Will run in the "/app directory "
32 RUN npm install
33
34 # Optional not doing anything.
35 EXPOSE 3000
36
37 # The difference between the "RUN" command to the "CMD" command
38 # is that the "RUN" command will run when an image is created.
39 # The "CMD" command will run when a container is created.
40 # To "CMD" we pass array [[]] with the commands we want to do.
41 CMD ["node", "server.js"]
42
```

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. The 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.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
zsh + - ⊞ ⊖

A self-sufficient runtime for containers

Options:
  --config string      Location of client config files (default "/Users/remez19/.docker")
  -c, --context string Name of the context to use to connect to the daemon (overrides
                        DOCKER_HOST env var and default context set with "docker context u
  -D, --debug          Enable debug mode
  -H, --host list       Daemon socket(s) to connect to
  -l, --log-level string Set the logging level ("debug"|"info"|"warn"|"error"|"fatal")
                        (default "info")
  --tls                Use TLS; implied by --tlsverify
  --tlscacert string   Trust certs signed only by this CA (default
                        "/Users/remez19/.docker/ca.pem")
  --tlscert string     Path to TLS certificate file (default
                        "/Users/remez19/.docker/cert.pem")
  --tlskey string       Path to TLS key file (default "/Users/remez19/.docker/key.pem")
  --tlsverify          Use TLS and verify the remote
  -v, --version         Print version information and quit

Management Commands:
  builder    Manage builds
  buildx*   Docker Buildx (Docker Inc., v0.7.1)
  compose*  Docker Compose (Docker Inc., v2.2.3)
  config     Manage Docker configs
  container  Manage containers
  context    Manage contexts
  image      Manage images
  manifest   Manage Docker image manifests and manifest lists
  network   Manage networks
  node      Manage Swarm nodes
  plugin    Manage plugins
  scan*     Docker Scan (Docker Inc., v0.17.0)
  secret    Manage Docker secrets
  service   Manage services
  stack     Manage Docker stacks
  swarm     Manage Swarm
  system    Manage Docker
  trust     Manage trust on Docker images
  volume    Manage volumes

Commands:
  attach     Attach local standard input, output, and error streams to a running container
  build      Build an image from a Dockerfile
  commit    Create a new image from a container's changes
  cp        Copy files/folders between a container and the local filesystem
  create    Create a new container
  diff      Inspect changes to files or directories on a container's filesystem
  events   Get real time events from the server
  exec     Run a command in a running container
  export   Export a container's filesystem as a tar archive
  history  Show the history of an image
  images   List images
  import   Import the contents from a tarball to create a filesystem image
  info     Display system-wide information
  inspect  Return low-level information on Docker objects
  kill     Kill one or more running containers
  load     Load an image from a tar archive or STDIN
  login   Log in to a Docker registry
  logout  Log out from a Docker registry
  logs    Fetch the logs of a container
  pause   Pause all processes within one or more containers
  port    List port mappings or a specific mapping for the container
  ps      List containers
  pull    Pull an image or a repository from a registry
  push    Push an image or a repository to a registry
  rename  Rename a container
  restart Restart one or more containers
  rm      Remove one or more containers
  rmi    Remove one or more images
  run     Run a command in a new container
  save   Save one or more images to a tar archive (streamed to STDOUT by default)
  search  Search the Docker Hub for images

```

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:

Data?

Application (Code + Environment)	Temporary App Data (e.g. entered user input)	Permanent App Data (e.g. user accounts)
Written & provided by you (= the developer)	Fetched / Produced in running container	Fetched / Produced in running container
Added to image and container in build phase	Stored in memory or temporary files	Stored in files or a database
"Fixed": Can't be changed once image is built	Dynamic and changing, but cleared regularly	Must not be lost if container stops / restarts
Read-only, hence stored in <u>Images</u>	Read + write, temporary, hence stored in <u>Containers</u>	Read + write, permanent, stored with <u>Containers & Volumes</u>

Understanding the Problem:

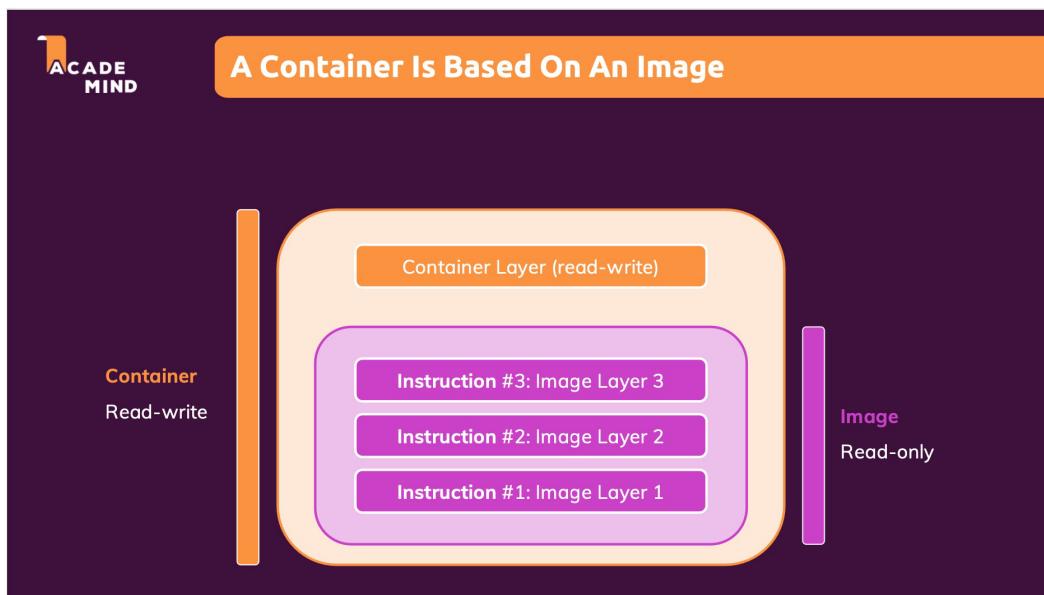
When we start a container the container add a layer of read-write.

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.

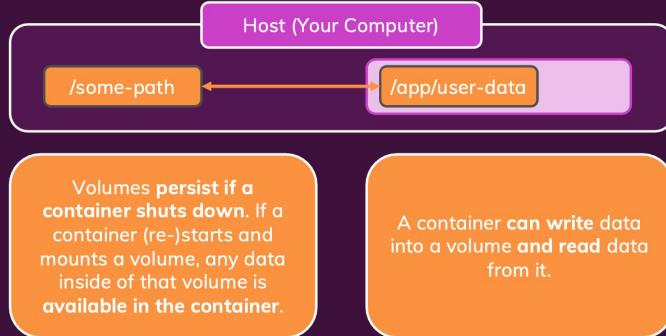
Volumes are just folders in our host machine.

With volumes we give access to a container to our host machine folders.

With that we can save data. (Long term)

Understanding Volumes

Volumes are **folders on your host machine** hard drive which are **mounted** ("made available", mapped) **into containers**



Volumes persist if a container shuts down. If a container (re-)starts and mounts a volume, any data inside of that volume is available in the container.

A container can write data into a volume and read data from it.

Named Volumes To The Rescue!:

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

Two Types of External Data Storages

Volumes
(Managed by Docker)

Bind Mounts
(Managed by you)

Anonymous Volumes

Named Volumes

Docker sets up a folder / path on your host machine, exact location is unknown to you (= dev).
Managed via `docker volume` commands.

You define a folder / path on your host machine.

A defined path in the container is mapped to the created volume / mount.
e.g. `/some-path` on your hosting machine is mapped to `/app/data`

Great for data which should be persistent but which you don't need to edit directly.

Great for persistent, editable (by you) data (e.g. source code).

The command is:

```
remez19 data-volumes-01-starting-setup % docker run -p 3000:3000 -d --name feedback -v feedback:/app/feedback feedback_node_volumes d59f3f2f67cd
```

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)

Bind Mounts (Managed by you)

You define a folder / path on
your host machine.

Created volume / mount.
Mapped to /app/data

Great for persistent, editable
(by you) data
(e.g. source code).

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
○ remez19 data-volumes-01-starting-setup % docker run -p 3000:3000 -d --rm --name feedback_node_volumes -v feedback:/app/feedback -v "/Users/remez19/vsProjects/Docker Projects/data-volumes-01-starting-setup:/app" 713b2fe438a2
Absolute path to the folder/file we want to share with the containers
```

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:

```
FROM node
WORKDIR /app
COPY . /app
RUN npm install
EXPOSE 3000
# VOLUME [ "/app/node_modules" ]
```

Two ways
of achieving
this

```
remez19 data-volumes-01-starting-setup % docker run -p 3000:3000 -d --name feedback_node_volumes -v feedback:/app/feedback -v "/Users/remez19/vsProjects/Docker Projects/data-volumes-01-starting-setup:/app" -v /app/node_modules 713b2fe438a2
```

feedback_node_volumes:

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

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER zsh

Read-Only We still want the container to write there so we add anonymous volume

- `remez19 data-volumes-01-starting-setup % docker run -p 3000:3000 -d --rm --name feedback_node_volumes -v feedback:/app/feedback -v "/Users/remez19/vsProjects/Docker Projects/data-volumes-01-starting-setup:/app:ro" -v /app/temp -v /app/node_modules 408f012f99c6`

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.

```
app.listen(process.env.PORT);
```

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

```
server.js Dockerfile X
Dockerfile > ...
FROM node
WORKDIR /app
COPY . /app
RUN npm install
ENV PORT=3000
EXPOSE $PORT
# VOLUME [ "/app/node_modules" ]
CMD ["npm", "start"]
```

```
● remez19 data-volumes-01-starting-setup %docker run -p 3000:3000 -d --rm --env PORT=3000 --name feedback_node_volum
es -v feedback:/app/feedback -v "/Users/remez19/vsProjects/Docker Projects/data-volumes-01-starting-setup:/app:ro"
-v /app/temp -v /app/node_modules feedback node:env
```

Using an environment file.

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

```

PORT=3000
  
```

```

remez19 data-volumes-01-starting-setup % docker run -p 3000:3000 -d --rm --env-file ./ .env -name feedback_node_volumes -v feedback:/app/feedback -v "/Users/remez19/vsProjects/Docker Projects/data-volumes-01-starting-setup:/app:ro" -v /app/temp -v /app/node_modules feedback_node:env
  
```

Using Build Arguments (ARG):

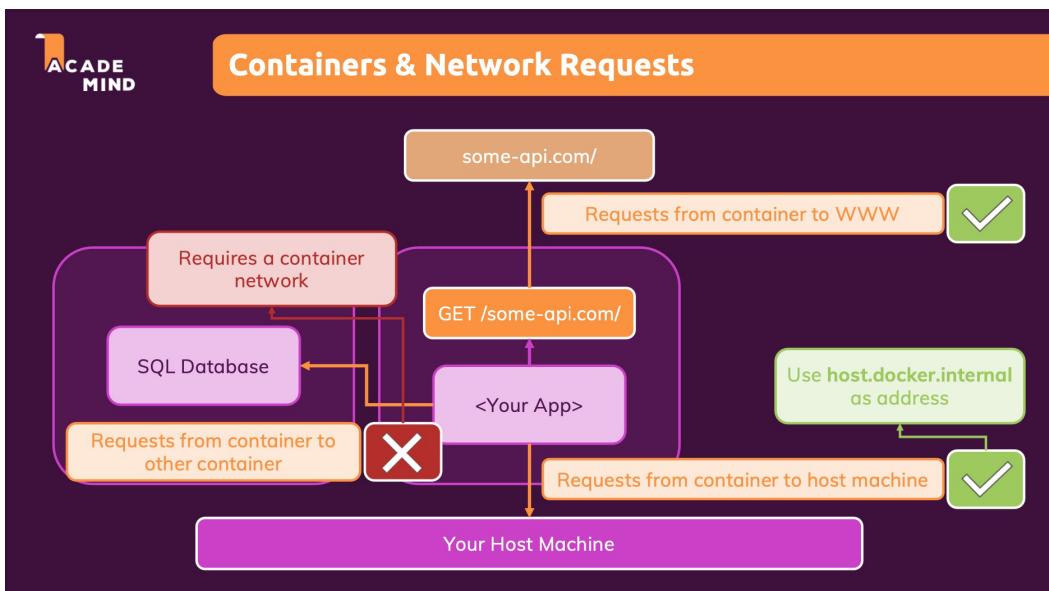
Setup arg in the docker file with ARG

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:

```
mongoose.connect([
  "mongodb://host.docker.internal:27017/swfavorites",
  { useNewUrlParser: true },
  (err) => {
    if (err) {
      console.log(err);  This special syntax is understood
    } else {           by docker
      app.listen(3000);
    }
  }
]);
```

Under the hood this will
be translated to our host
local machine ip address

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 mongodb server on our 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 not 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", // IP address of the container
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "MacAddress": "02:42:ac:11:00:02",
        "Networks": {
          "bridge": {
            "IPAMConfig": null,
            "Links": null,
            "Aliases": null,
            "NetworkID": "fa08e9f3b19444b6efb9df098286fffeec2fee7f2f4f4dfb4a"
          }
        }
      }
    }
  }
}

```

With this the two containers communicate between them.

Introducing Docker Networks: Elegant Container to Container Communication:

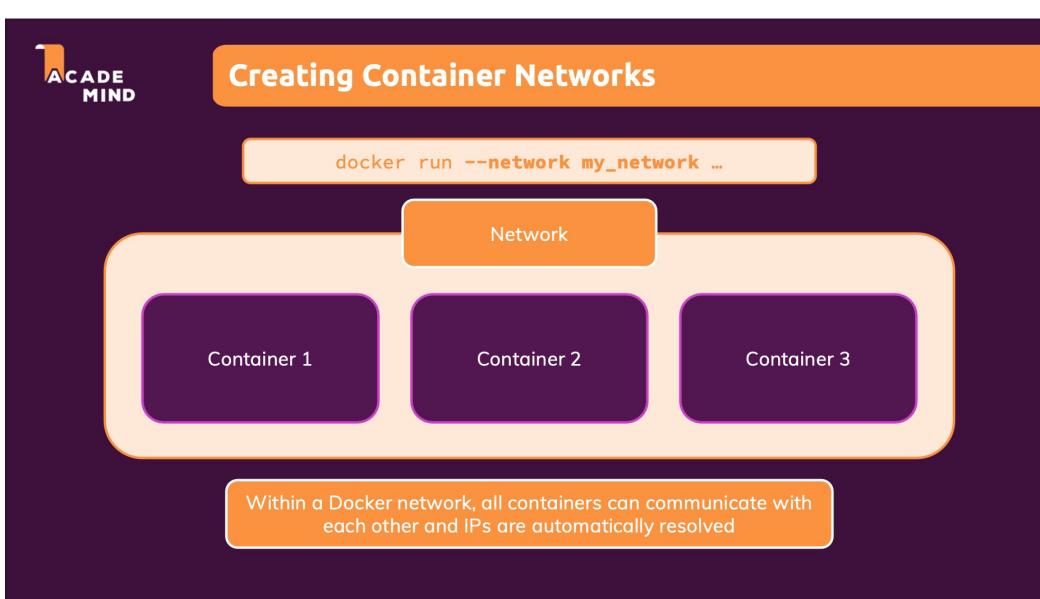
With docker we can setup container network.

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.

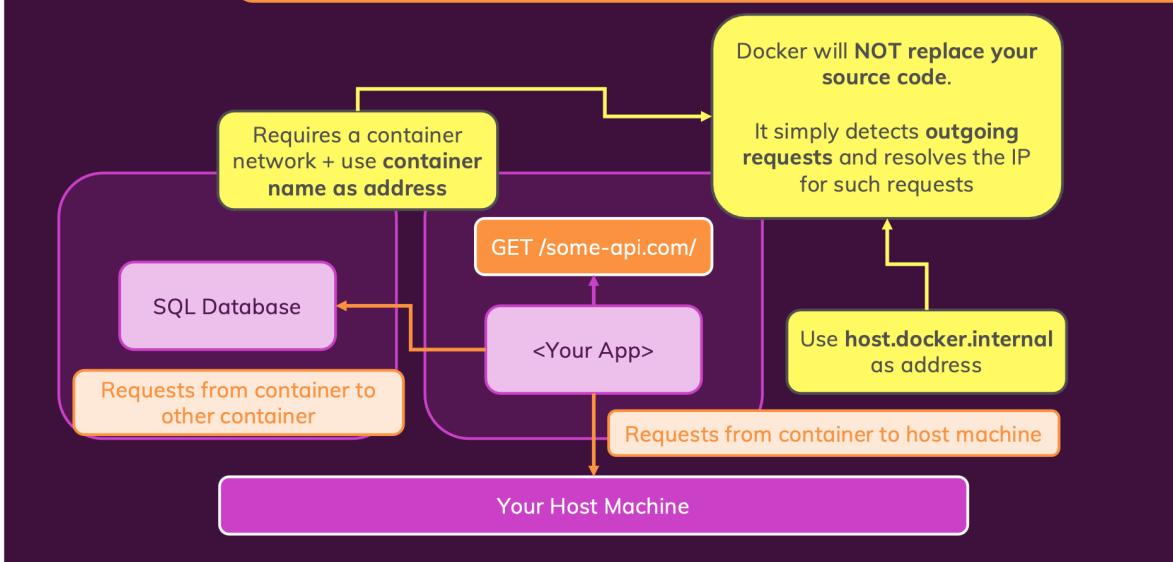
The name of the container will be translated by docker to ip.

And this will work if the containers are in the same network !

```
mongoose.connect(  
  "mongodb://mongodb:27017/swfavorites",  
  { useNewUrlParser: true },  
  (err) => {  
    if (err) {  
      console.log(err);  
    } else {  
      app.listen(3000);  
    }  
  }  
);
```

How Docker Resolves IP Addresses:

Understanding Docker Network IP Resolving

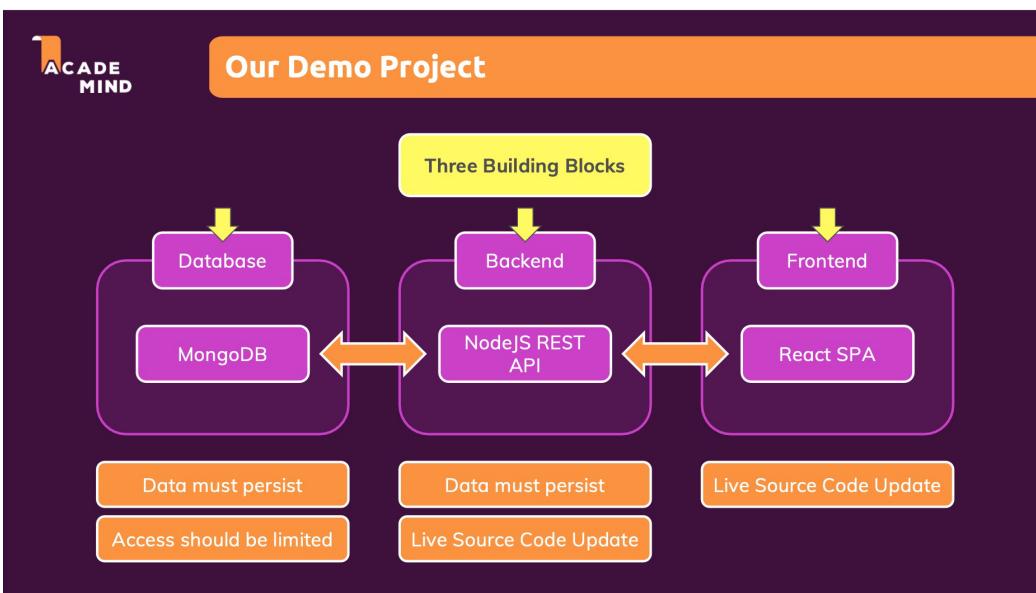


Section 5: Building Multi-Container Application with Docker:

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/backend:/app:ro" -v /app/node_modules --network full-stack-net backend-node
```

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

The screenshot shows a code editor with two files open: `Dockerfile` and `package.json`.

Dockerfile:

```
FROM node
WORKDIR /backend
COPY package.json .
RUN npm install
EXPOSE 80
COPY . .
CMD [ "npm", "start" ]
```

package.json:

```
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon app.js"
  },
  "author": "Maximilian Schwarzmüller",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.19.0",
    "express": "^4.17.1",
    "mongoose": "^5.10.3",
    "morgan": "^1.10.0"
  },
  "devDependencies": {
    "nodemon": "^2.0.4"
  }
}
```

Red boxes highlight the `start` script in the `scripts` section of `package.json` and the `CMD` command in the `Dockerfile`, indicating they are the key components for live code reload.

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:

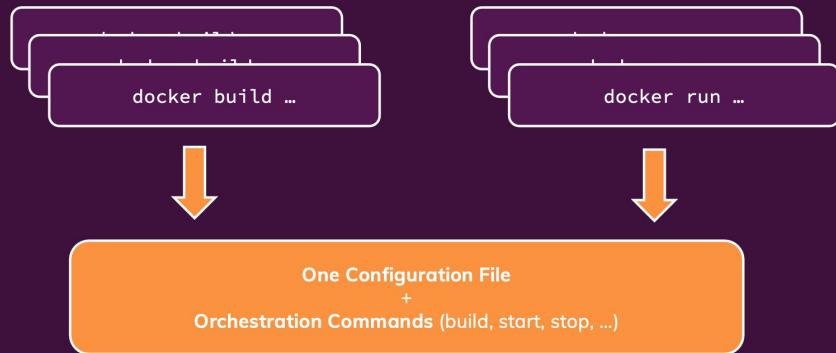
Using Docker compose Makes the managing of a multi-container setup easier !

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)

What is Docker Compose?



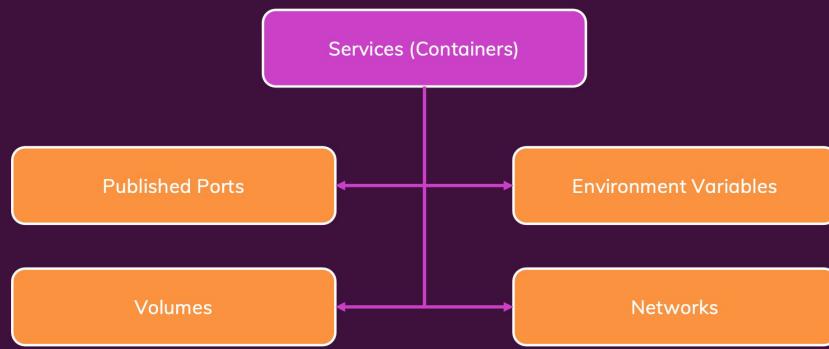
Docker compose is great for managing multiple containers on the same host!

What Docker Compose is NOT

Docker Compose does **NOT** replace Dockerfiles for custom Images

Docker Compose does **NOT** replace Images or Containers

Docker Compose is **NOT** suited for managing multiple containers on different hosts (machines)



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.yaml ● docker-commands.txt

  docker-compose.yaml
  1  # The version of the docker compose we want to use.
  2  # The version we define here has an effect of the
  3  # features we can use in this docker-compose file.
  4  version: "3.8"
  5  # When using services we need to indent the next
  6  # line (children) by two.
  7  # Below services we write our containers names
  8  services:
  9    mongodb:
 10      # Can also be a custom image.
 11      # by default when we use docker compose when a container.
 12      # is stopped he will be removed.
 13      # also by default we start containers in detached mode.
 14      # no need for (--rm, -d).
 15      image: "mongo"
 16      volumes:
 17        # We add volumes as we learned.
 18        # At the end of the volume info we can add ":".
 19        # To specify more options such as "ro" = read only.
 20        - data:/data/db
 21
 22      # alternatively we can use "environment" file:
 23      # env_file:
 24      #   - ./env/mongo.env
 25      environment:
 26        - MONGO_INITDB_ROOT_USERNAME=max
 27        - MONGO_INITDB_ROOT_PASSWORD=secret
 28
 29      # In many cases we don't need to specify the networks
 30      # Because when we use docker compose docker will
 31      # automatically create a new environment (network) for all
 32      # the containers specified in this compose file and add them
 33      # networks:
 34      #   - networkName
 35
 36      # When using a NAMED volumes we need to add the names as such:
 37      # volumes:
 38      #   data:
 39
 40
 41      # backend:
 42
 43      # frontend:
 44
 45

```

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:



The screenshot shows a terminal window with two tabs: "docker-compose.yaml" and "docker-commands.txt". The "docker-compose.yaml" tab is active and displays the following YAML configuration:

```
29  | # In many cases we dont need to specify the networks
30  | # Because when we use docker compose docker will
31  | # automatically create a new environment (network) for all
32  | # the containers specified in this compose file and add them
33  | # networks:
34  | #   - networkName
35  | backend:
36  | # we can use "build" to tell docker compose to run
37  | # this container on image built from docker file.
38  | # We need to specify the path to docker file.
39  | build: ./backend/
40  | # Another way:
41  | # build:
42  | #   context: ./backend
43  | #   dockerfile: Dockerfile
44  | #   args:
45  | #     - some-arg=value
46
47  # Allows us to sepcify a published ports
48  ports:
49  | # "host:container_internal_port"
50  | - "80:80"
51  volumes:
52  | - logs:/app/logs
53  | - /app/node_modules
54  | - "./backend:/app"
55  environment:
56  | - MONGODB_USERNAME=max
57  | - MONGODB_PASSWORD=secret
58  # depends_on - available only in docker compose
59  # When a conatiner depends on another container runing
60  # In our case backend depends on mongodb
61  # we can tell docker compose to first run all the dependencies
62  # (Containers) and then this one!
63  depends_on:
64  | - mongodb
```

Adding Another Container:

Adding the frontend app setup.

```

  docker-compose.yaml X dockerCommands.txt
  docker-compose.yaml

  28
  29      # In many cases we dont need to specify the networks
  30      # Because when we use docker compose docker will
  31      # automatically create a new environment (network) for all
  32      # the containers specified in this compose file and add them
  33      # networks:
  34      #   - networkName
  35
  36      backend:
  37          # we can use "build" to tell docker compose to run
  38          # this container on image built from docker file.
  39          # We need to specify the path to docker file.
  40          build: ./backend/
  41          # Another way:
  42          # build:
  43          #   context: ./backend
  44          #   dockerfile: Dockerfile
  45          #   args:
  46          #     - some-arg=value
  47
  48          # Allows us to sepcify a published ports
  49          ports:
  50              # "host:container_internal_port"
  51              - "80:80"
  52
  53          volumes:
  54              - logs:/app/logs
  55              - /app/node_modules
  56              - "./backend:/app"
  57
  58          environment:
  59              - MONGODB_USERNAME=max
  60              - MONGODB_PASSWORD=secret
  61
  62          # depends_on - available only in docker compose
  63          # When a conatiner depends on another container runing
  64          # In our case backend depends on mongodb
  65          # we can tell docker compose to first run all the dependencies
  66          # (Containers) and then this one!
  67          depends_on:
  68              - mongodb
  69
  70
  71      frontend:
  72          build: ./frontend/
  73          ports:
  74              - "3000:3000"
  75          volumes:
  76              - ./frontend/src:/app/src
  77          # Letting docker know that this service needs an open connection
  78          stdin_open: true
  79          tty: true
  80
  81          depends_on:
  82              - backend
  83
  84
  85      # When using a NAMED volumes we need to add the names as such:
  86      volumes:
  87          data:
  88          logs:
  89
  90

```

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.

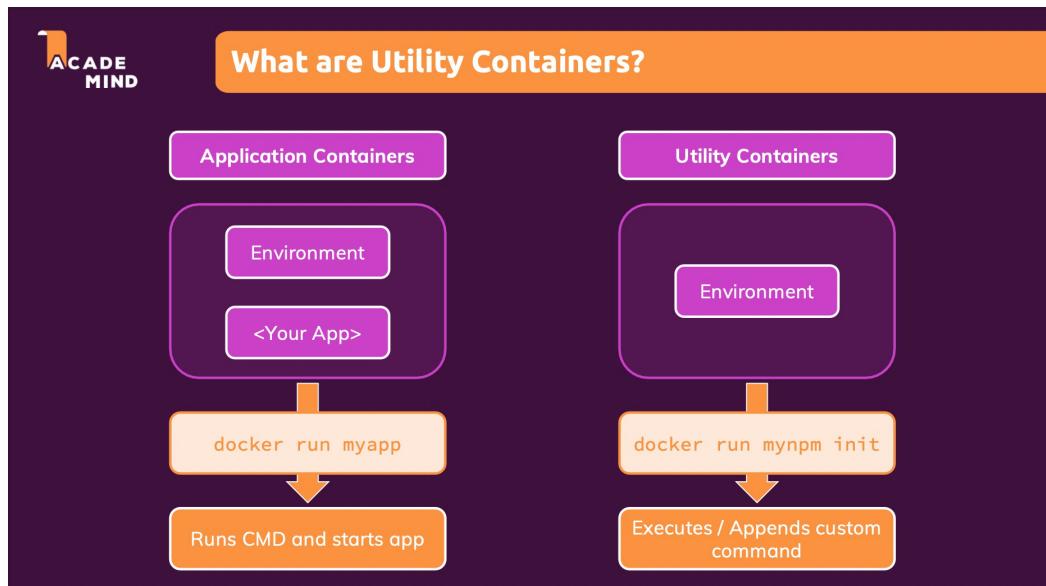
Else docker will always use the already existing image (from the first up command).

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`"

The screenshot shows the VS Code interface with the Terminal tab selected. The terminal window displays the following text:

```
remez19 Utility_Containers % docker run -it -v "/Users/remez19/vsProjects/Docker Projects/Utility_Containers:/app" node -u  
til npm init  
This utility will walk you through creating a package.json file.  
It only covers the most common items, and tries to guess sensible defaults.  
See 'npm help init' for definitive documentation on these fields  
and exactly what they do.  
  
Use 'npm install <pkg>' afterwards to install a package and  
save it as a dependency in the package.json file.  
  
Press ^C at any time to quit.  
package name: (app) test  
Sorry, name cannot contain leading or trailing spaces and name can only contain URL-friendly characters.  
package name: (app) test  
version: (1.0.0)  
description:  
entry point: (index.js)  
test command:  
git repository:  
keywords:  
author:  
license: (ISC)  
About to write to /app/package.json:  
  
{  
  "name": "test",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC"  
}  
  
Is this OK? (yes)  
o remez19 Utility_Containers %
```

Utilizing ENTRYPPOINT:

```
 Dockerfile X  
 Dockerfile > ...  
1  FROM node:14-alpine  
2  
3  WORKDIR /app  
4  
5  # With this we can make sure that every command we run  
6  # when using utility containers will be appended to "npm".  
7  # So now what ever comes after the image name (in the run command)  
8  # will be apended to npm therefore if we type  
9  # "install" -> "npm install".  
10 ENTRYPOINT [ "npm" ]  
11  
12 # docker build -t node-util .
```

Using Docker Compose:

Using docker compose to achieve the same result:

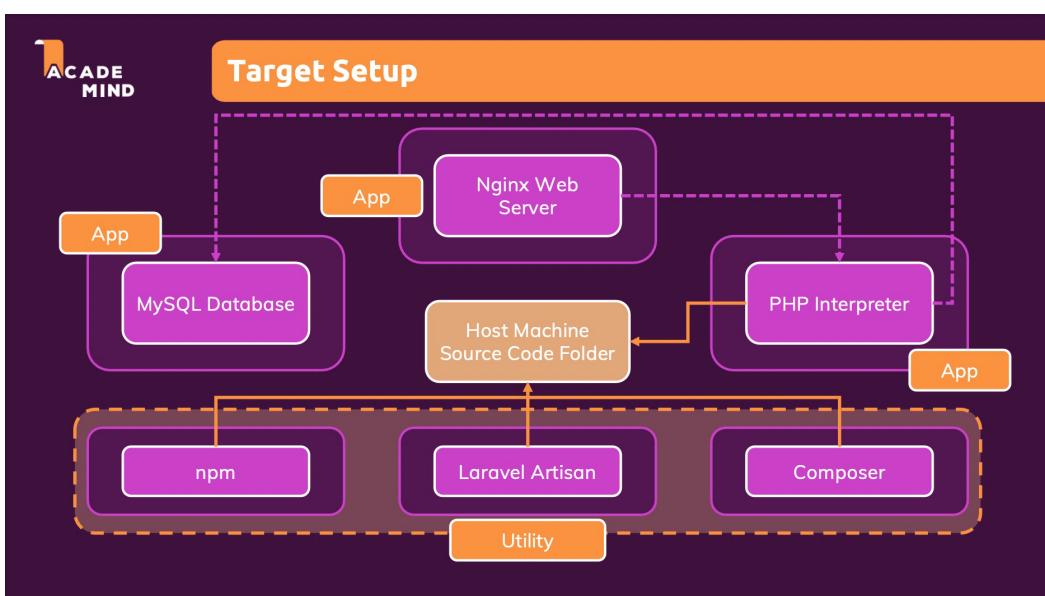
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER > docker + - X
```

remez19 Utility_Containers %docker-compose run npm init
[+] Running 1/0
 :: Network utility_containers_default Created 0.0s
[+] Building 2.4s (7/7) FINISHED
=> [internal] load build definition from Dockerfile 0.0s
=> [internal] load .dockerignore 0.0s

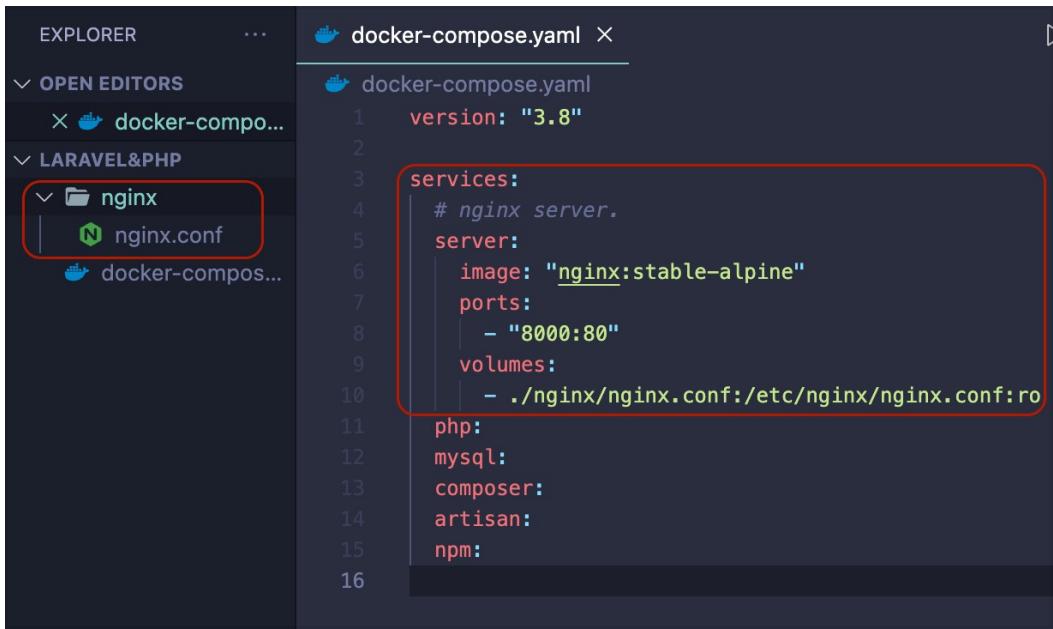
```
 docker-compose.yaml X  
  
 docker-compose.yaml  
1   version: "3.8"  
2   services:  
3     npm:  
4       build: ./  
5       stdin_open: true  
6       tty: true  
7       volumes:  
8         - ./:/app  
9
```

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

The Target Setup:



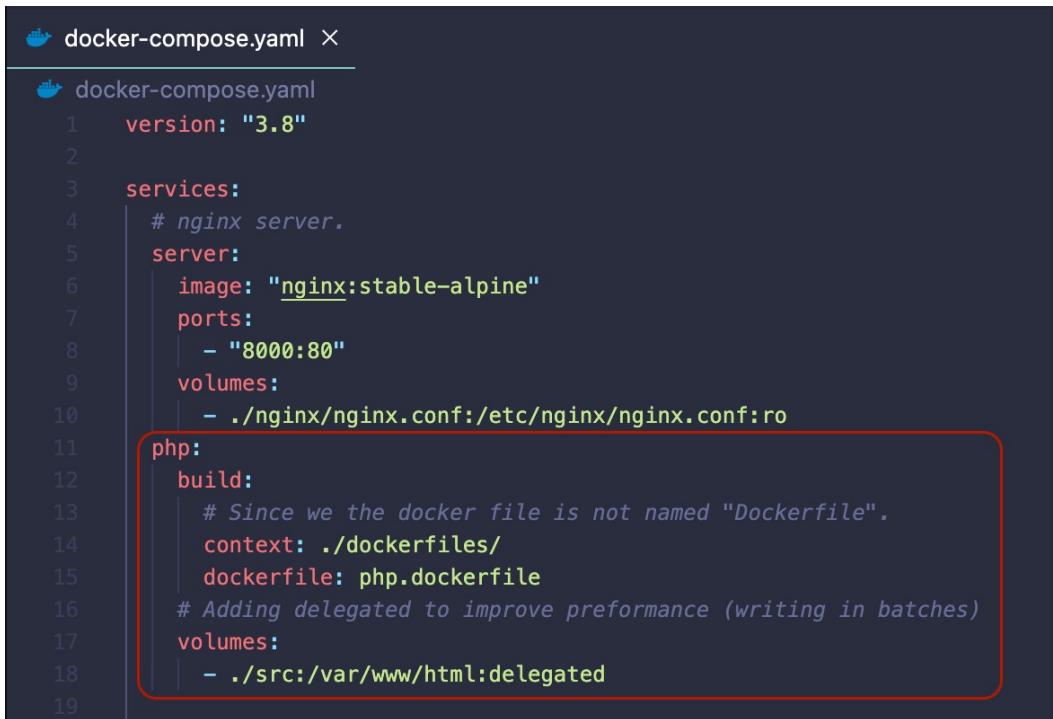
Adding a Nginx (Web Server) Container:



The screenshot shows the VS Code interface with the Explorer and Editor panes. In the Explorer pane, there's a folder named 'LARAVEL&PHP' containing an 'nginx' folder with an 'nginx.conf' file. The Editor pane displays the 'docker-compose.yaml' file. A red box highlights the configuration for the 'server' section of the 'nginx' service:

```
version: "3.8"
services:
  # nginx server.
  server:
    image: "nginx:stable-alpine"
    ports:
      - "8000:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
```

Adding a PHP Container:



The screenshot shows the VS Code interface with the Editor pane displaying the 'docker-compose.yaml' file. A red box highlights the configuration for the 'php' service:

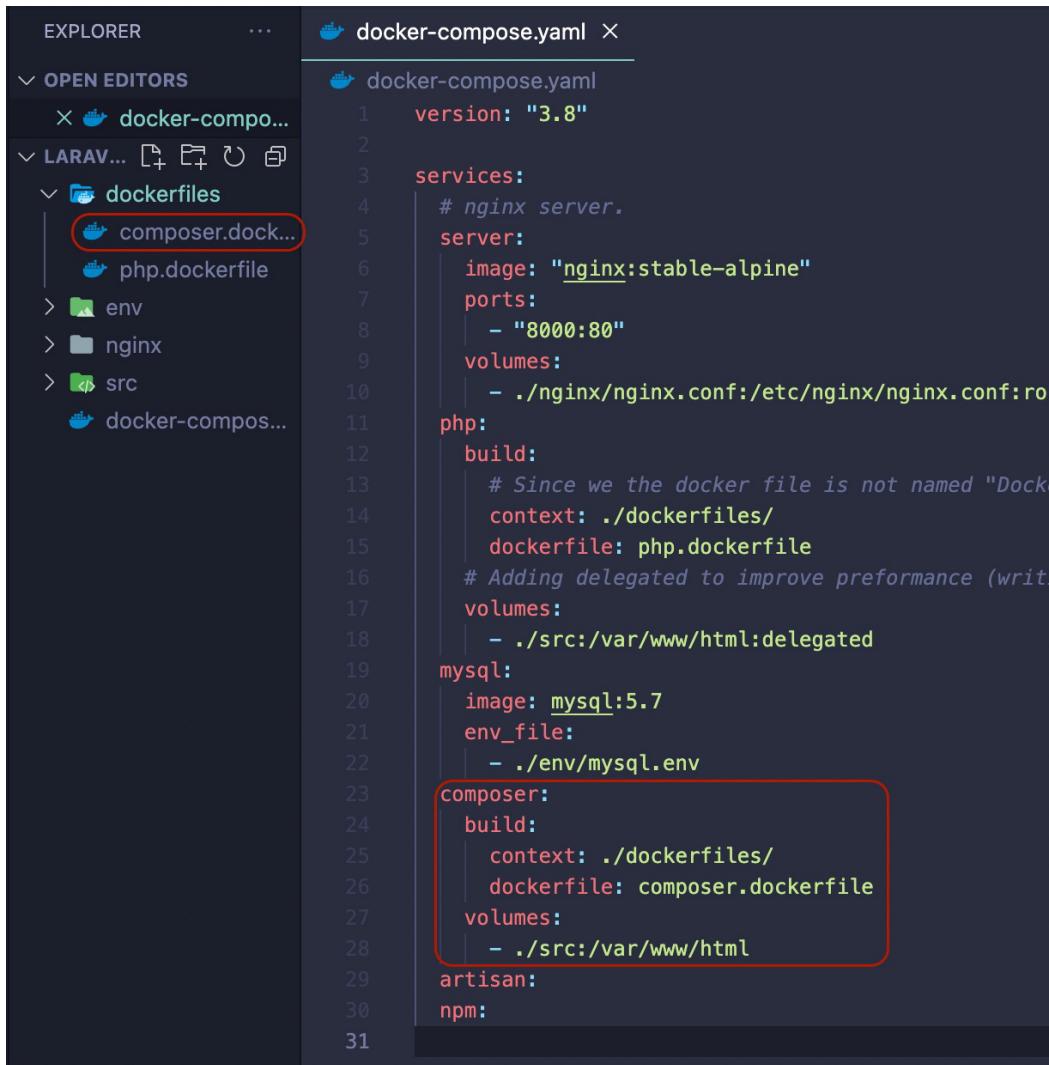
```
version: "3.8"
services:
  # nginx server.
  server:
    image: "nginx:stable-alpine"
    ports:
      - "8000:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
  php:
    build:
      # Since we the docker file is not named "Dockerfile".
      context: ./dockerfiles/
      dockerfile: php.dockerfile
      # Adding delegated to improve preformance (writing in batches)
    volumes:
      - ./src:/var/www/html:delegated
```

Adding a MySQL Container:

The screenshot shows the VS Code interface with the Explorer and Editor panes. In the Explorer pane, there is a folder named 'LARAV...' containing 'dockerfiles' and 'env'. Inside 'env', there is a file named 'mysql.env'. In the Editor pane, the 'docker-compose.yaml' file is open, showing the following configuration:

```
version: "3.8"
services:
  # nginx server.
  server:
    image: "nginx:stable-alpine"
    ports:
      - "8000:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
  php:
    build:
      # Since we the docker file is not named "Dockerfile"
      context: ./dockerfiles/
      dockerfile: php.dockerfile
    # Adding delegated to improve preformance (writing)
    volumes:
      - ./src:/var/www/html:delegated
  mysql:
    image: mysql:5.7
    env_file:
      - ./env/mysql.env
  composer:
  artisan:
  npm:
```

Adding a Composer Utility Container:



```
version: "3.8"
services:
  # nginx server.
  server:
    image: "nginx:stable-alpine"
    ports:
      - "8000:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
  php:
    build:
      # Since we the docker file is not named "Dockerfile"
      context: ./dockerfiles/
      dockerfile: php.dockerfile
    # Adding delegated to improve preformance (writing)
    volumes:
      - ./src:/var/www/html:delegated
  mysql:
    image: mysql:5.7
    env_file:
      - ./env/mysql.env
  composer:
    build:
      context: ./dockerfiles/
      dockerfile: composer.dockerfile
    volumes:
      - ./src:/var/www/html
  artisan:
  npm:
```

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:



Containers Are Always Great!

In Development

In Production

Isolated, standalone environment

Reproducible environment, easy to share and use

Isolated, standalone environment

Reproducible environment, easy to share and use

No surprises!

What works on your machine (in a container) will also work after deployment



Development to Production: Things To Watch Out For

Bind Mounts **shouldn't** be used in Production!

Containerized apps **might need a build step** (e.g. React apps)

Multi-Container projects might need to be **split** (or should be split) across multiple hosts / remote machines

Trade-offs between **control** and **responsibility** might be worth it!

Deployment Process & Providers:

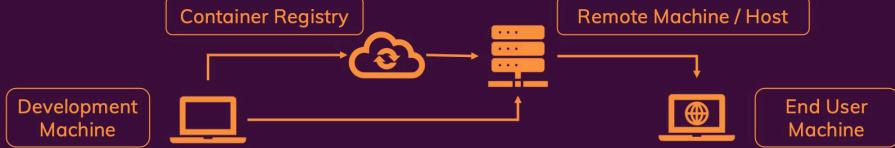
A Basic First Example: Standalone NodeJS App

Just NodeJS, no database, nothing else

1 Image & Container

Possible Deployment Approach

Install Docker on a remote host (e.g. via SSH), push and pull image, run container based on image on remote host



Hosting Providers

There are hundreds and thousands of Docker-supporting hosting providers out there!



Amazon Web Services



Microsoft Azure



Google Cloud

Getting Started With An Example:

Example: Deploy to AWS EC2

AWS EC2 is a service that allows you to spin up and manage your own remote machines

1

Create and launch EC2 instance, VPC and security group

2

Configure security group to expose all required ports to WWW

3

Connect to instance (SSH), install Docker and run container

Bind Mounts In Production:

Bind Mounts, Volumes & COPY

In Development

Containers should encapsulate the runtime environment but not necessarily the code

Use "Bind Mounts" to provide your local host project files to the running container

Allows for instant updates without restarting the container

In Production

Image / Container is the "single source of truth"

A container should really work standalone, you should NOT have source code on your remote machine

Use COPY to copy a code snapshot into the image

Ensures that every image runs without any extra, surrounding configuration or code

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.

Connect to instance [Info](#)

Connect to your instance i-0fe3b06770b45ade3 (Docker_Containers_Deployment_Exercise) using any of these options

[EC2 Instance Connect](#) | [Session Manager](#) | **SSH client** | [EC2 serial console](#)

Instance ID
 [i-0fe3b06770b45ade3 \(Docker_Containers_Deployment_Exercise\)](#)

1. Open an SSH client.
2. Locate your private key file. The key used to launch this instance is example-1.pem
3. Run this command, if necessary, to ensure your key is not publicly viewable.
 chmod 400 example-1.pem
4. Connect to your instance using its Public DNS:
 [ec2-43-206-105-78.ap-northeast-1.compute.amazonaws.com](#)

Example:
 ssh -i "example-1.pem" ec2-user@ec2-43-206-105-78.ap-northeast-1.compute.amazonaws.com

Note: In most cases, the guessed user name is correct. However, read your AMI usage instructions to check if the AMI owner has changed the default AMI user name.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
ssh + - ×

● remez19 deployment-01-starting-setup %chmod 400 example-1.pem
○ remez19 deployment-01-starting-setup %ssh -i "example-1.pem" ec2-user@ec2-43-206-105-78.ap-northeast-1.compute.amazonaws.com
The authenticity of host 'ec2-43-206-105-78.ap-northeast-1.compute.amazonaws.com (43.206.105.78)' can't be established.
ED25519 key fingerprint is SHA256:Ix0F/eFnkEcea9x6g089YmpiRwlJqu+JHhSD1JihZ94.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? y
Please type 'yes', 'no' or the fingerprint: yes
Warning: Permanently added 'ec2-43-206-105-78.ap-northeast-1.compute.amazonaws.com' (ED25519) to the list of known hosts.

-| ( --|_) _|_ Amazon Linux 2 AMI
-| \__|_|_ Connecting to the host machine.
https://aws.amazon.com/amazon-linux-2/ Now we are inside the host machine terminal!
[ec2-user@ip-172-31-13-210 ~]$ █

```

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**

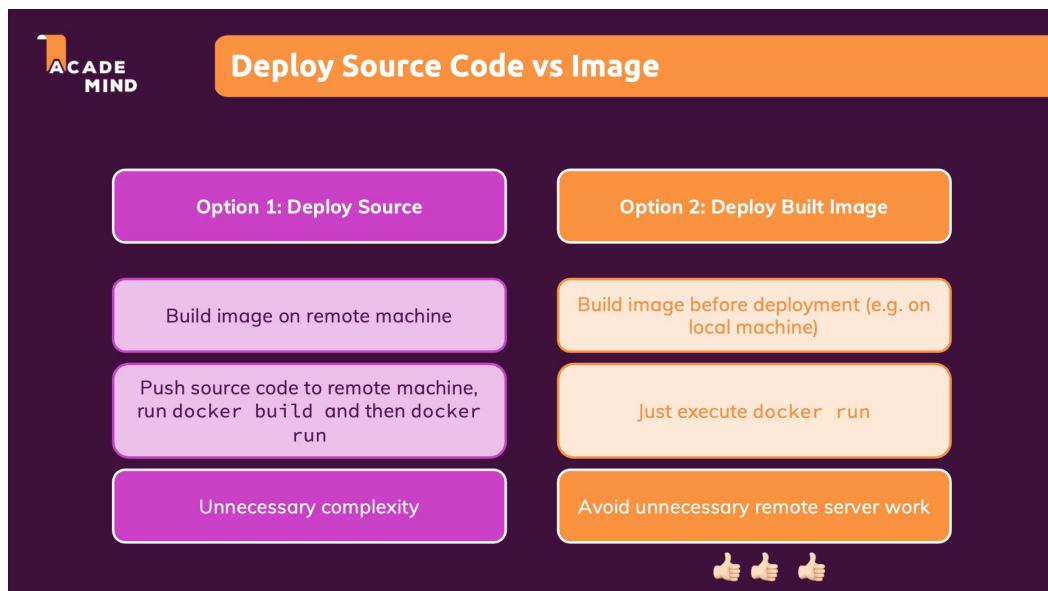
Of course you might not always want to install it on a AWS EC2 instance though - maybe you are using a different provider.

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-example-1-docker-deployment
```

Managing & Updating the Container / Image:

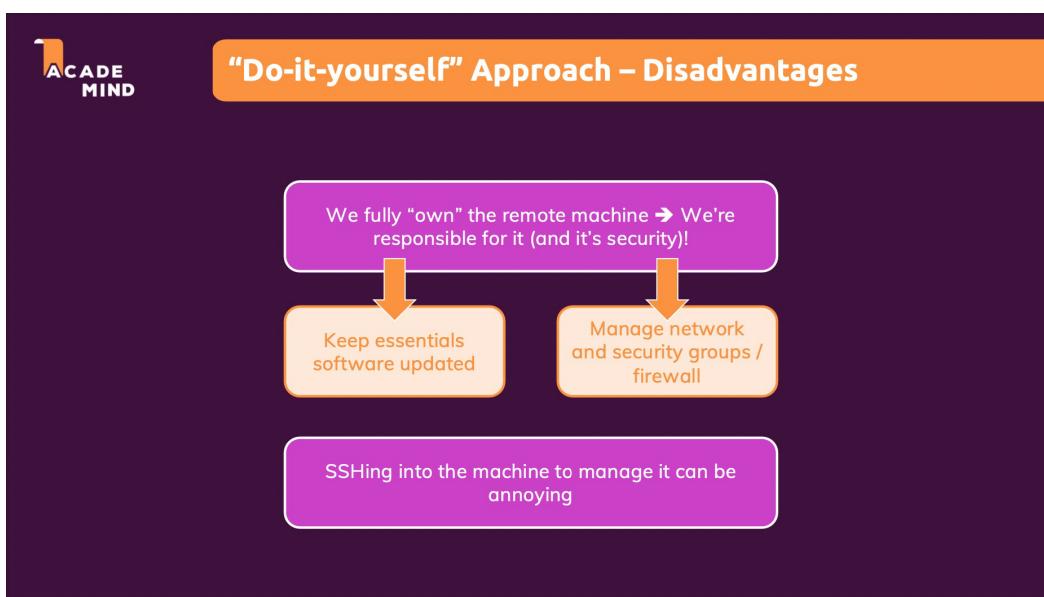
To Make sure that we see the changes we making in our src code we need to go through all the steps again:

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:

A Managed / Automated Approach



Your Own Remote Machines
e.g. AWS EC2



Managed Remote Machines
e.g. AWS ECS

You need to create them, manage them, keep them updated, monitor them, scale them etc.

Great if you're an experienced admin / cloud expert

Creation, management, updating is handled automatically, monitoring and scaling is simplified

Great if you simply want to deploy your app / containers

Databases & Containers: An Important Consideration:

A Note about Databases

You can absolutely manage your own Database containers

but ...



Scaling & managing availability can be challenging



Performance (also during traffic spikes) could be bad



Taking care about backups and security can be challenging

Consider using a **managed Database service** (e.g. AWS RDS, MongoDB Atlas, ...)

Understanding a Common Problem:

Apps with Development Servers & Build Steps

Some apps / projects require a build step

e.g. optimization script that needs to be executed **AFTER** development but **BEFORE** deployment

Development Setup

IS NOT EQUAL TO

Production Setup

(not entirely)

e.g. ReactJS App

Uses live-reloading
development server, uses
unoptimized / unsupported
JS features

Build Step / Script

No attached server,
optimized, fully browser-
compatible code

Introducing Multi-Stage Builds:

Introducing Multi-Stage Builds

One Dockerfile, Multiple Build / Setup Steps ("Stages")

Stages can copy results (created files and folders) from each other



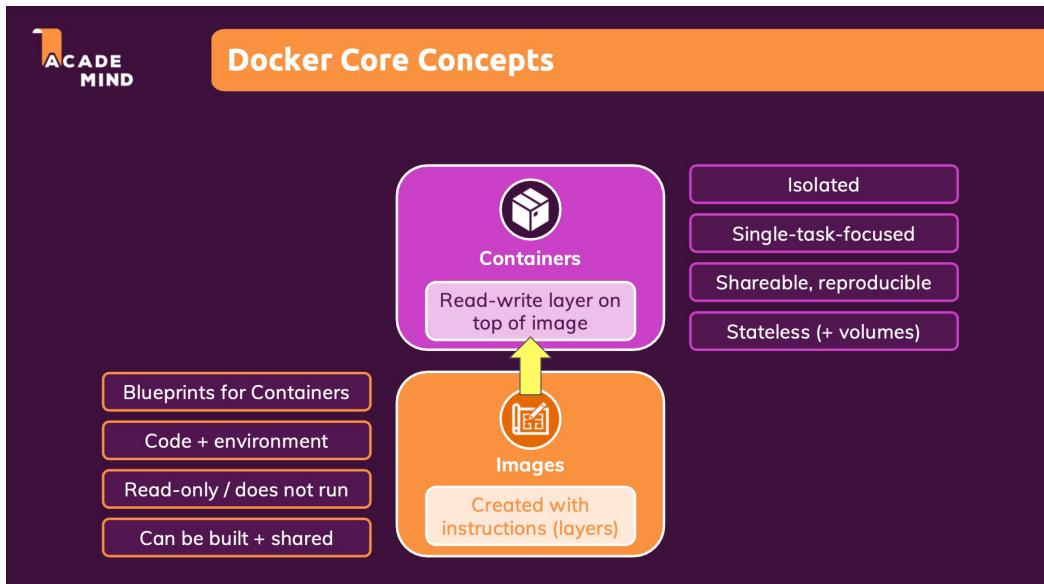
You can either build the complete image or select individual stages



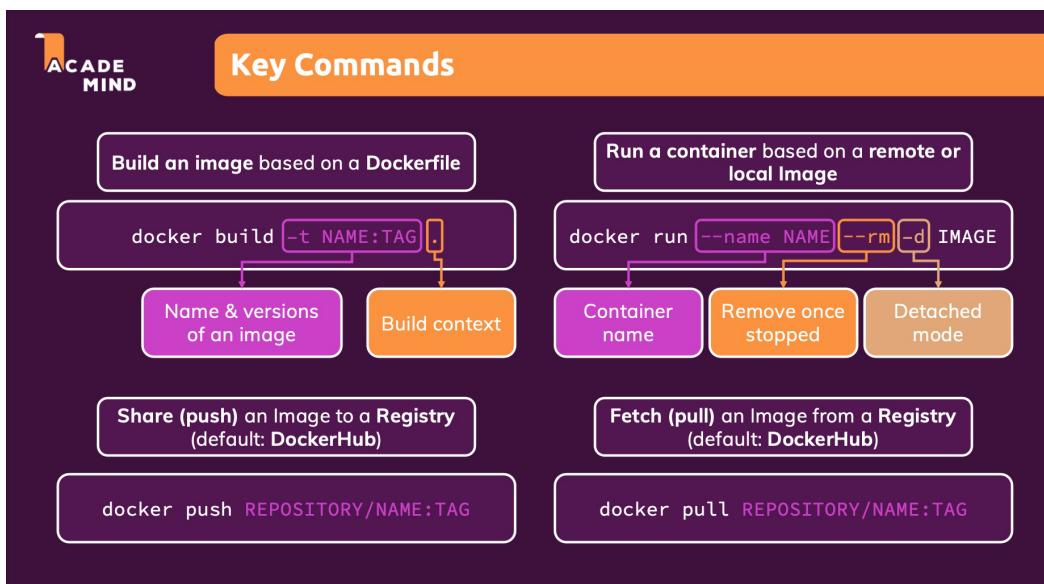
Section 10: Docker & Containers - A Summary:

Images & Containers:

Docker Core Concepts

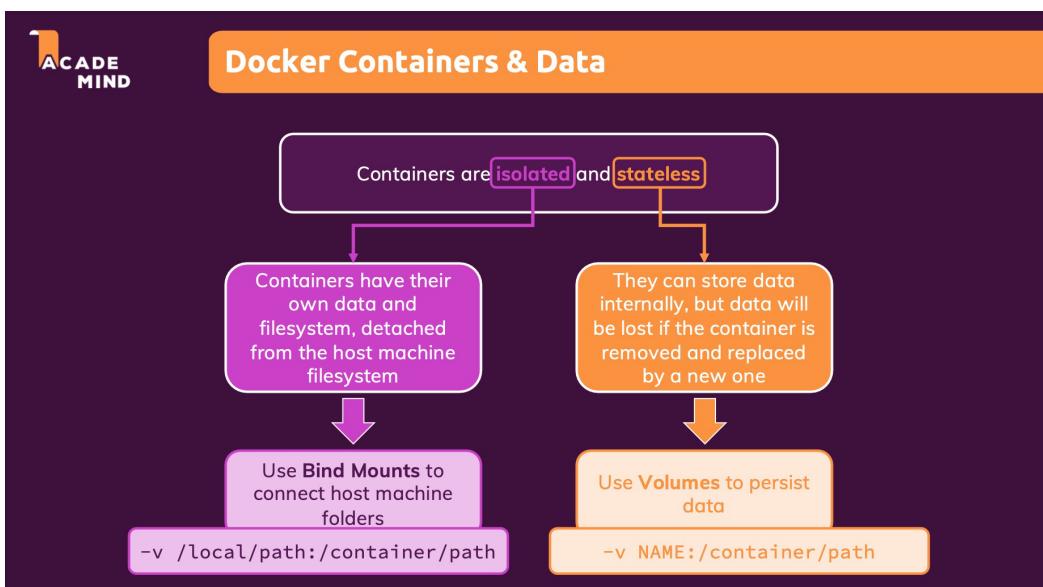


Key Commands:

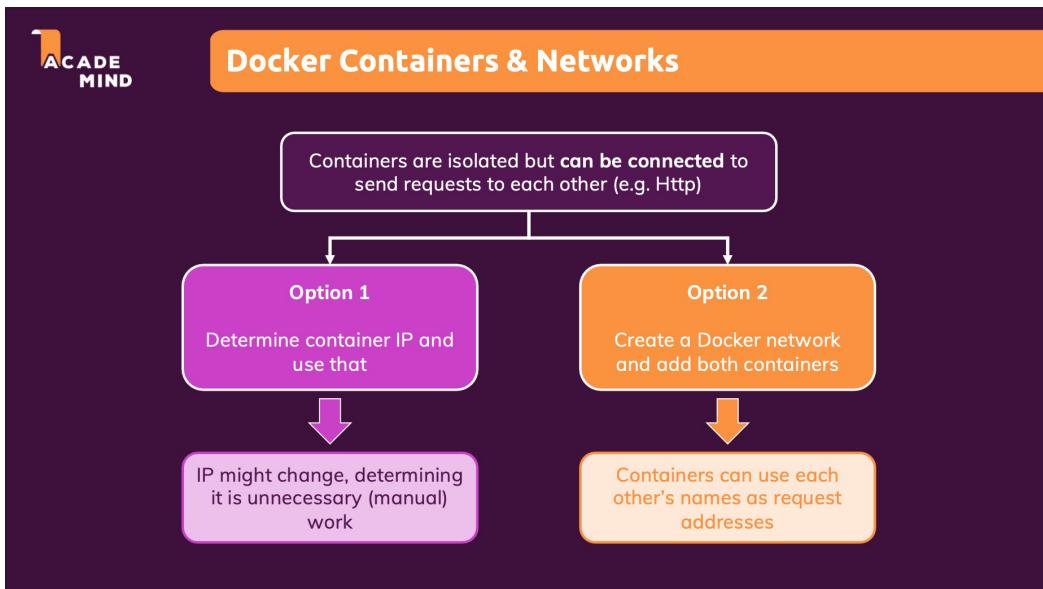


Data, Volumes & Networking:

Docker Containers & Data



Docker Containers & Networks



Docker Compose:

Docker vs Docker Compose

Repeating long `docker build` and `docker run` commands gets annoying – especially when working with multiple containers

Docker Compose allows you to pre-define build and run configuration in a .yaml file

`docker-compose up`

Build missing images and start all containers

`docker-compose down`

Stop all started containers

Local vs Remote:

Local Host (Development) vs Remote Host (Production)

Local Host / Development

Remote Host / Production

Isolated, encapsulated, reproducible development environments

No dependency or software clashes

Isolated, encapsulated, reproducible environments

Easy updates: Simply replace a running container with an updated one

Develop your application in the same environment you'll run it in after deployment

Deployment:

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

You can use a managed service instead

Full control but you also need to
manage everything

Less control and extra knowledge
required but easier to use, less
responsibility