Containerizing a Web Application with Vue.js and Java

Jadson Santos

Container

- Seaports before container were a mess

Container

- After container, load a ship became faster

Container

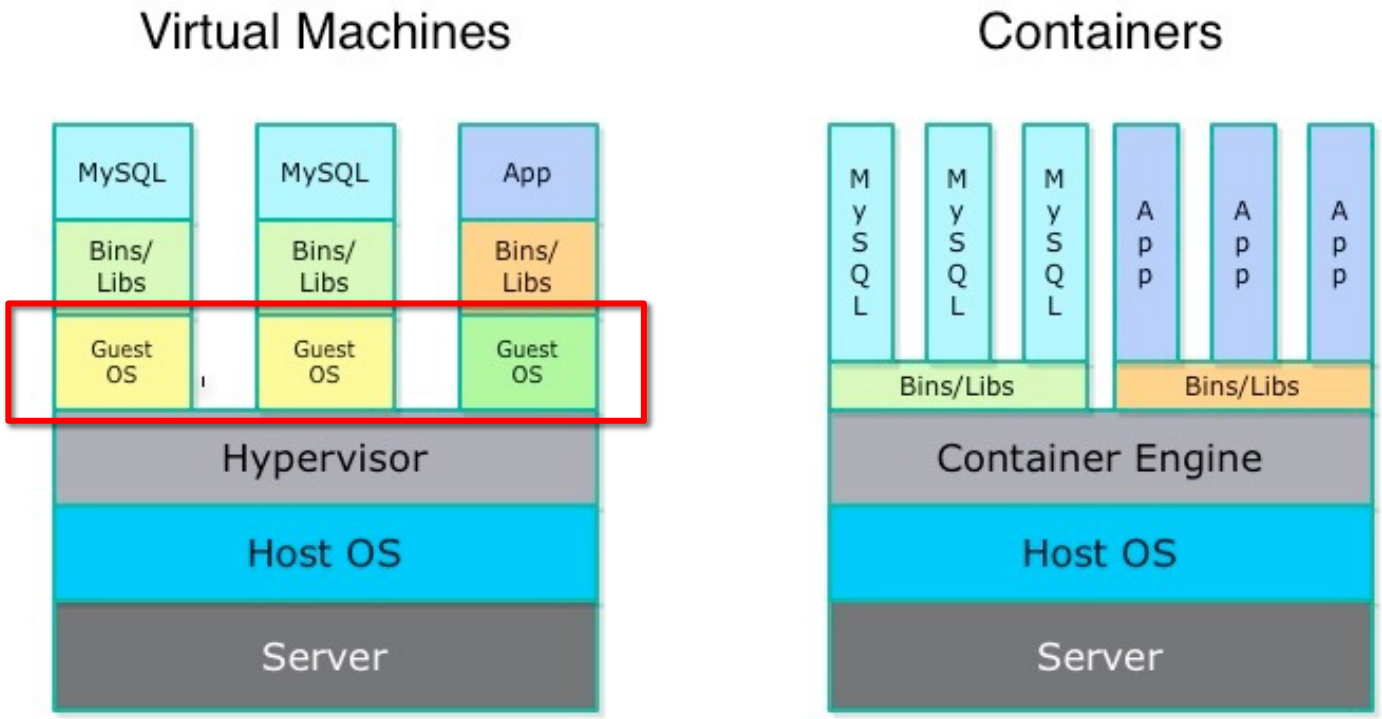
- How setup the execution environment
- 1º Install directly in SO: Very complex, several different configurations
- 2º Use virtual machines like VMWare, Virtual Box: memory and processing consumption is too high
- 3º Use a container: it's possible to run an application with all the required settings (environment variables, packages, etc.) with minimal impact

Container

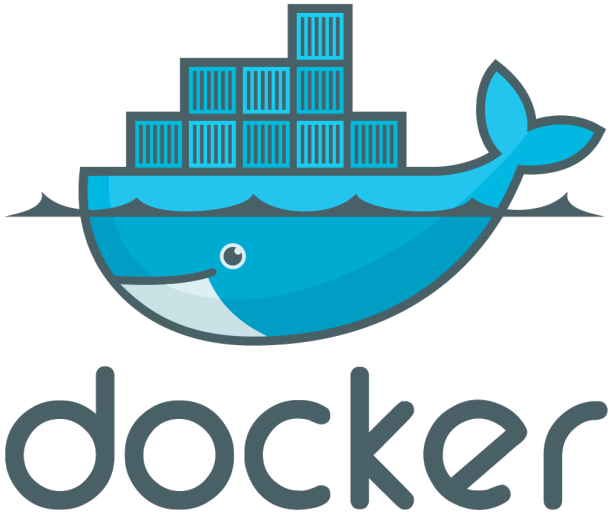
- Containers isolate microservice processes and applications into smaller instances that **utilize only the virtualized operating system rather than the full virtual machine** and the entire complement of abstracted hardware that VM includes

- For containers, they operate more as fully isolated sandboxes, with only **the minimal kernel of the operating system** present for each container.

Container

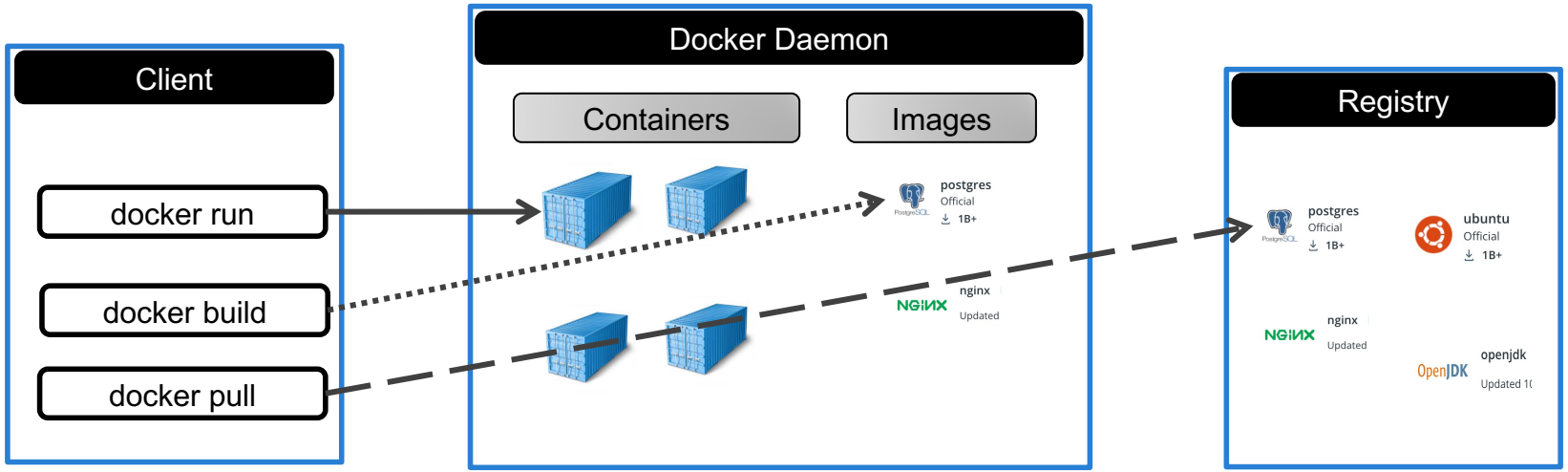

Virtual Machines
MySQL
MySQL
App
Bins/ Libs
Bins/ Libs
Bins/ Libs
Guest OS
Guest OS
Guest OS
Hypervisor
Host OS
Server
Containers
MySQL
MySQL
MySQL
App
App
App
Bins/Libs
Bins/Libs
Container Engine
Host OS
Server

Installation



Docker

- Docker is **a set** of platform-as-a-service products that use operating system-level virtualization to deliver software in packages called containers.
- Docker do not create the container technology, but was the first tool to let is popular and and relatively easy to use.



Docker

- **Client**: Client receiving and executes the commands of users
- **Docker Daemon**: Who manages the containers
- **Registry**: Where the images are stored

Installation

- Linux
  - First, install some prerequisite packages that let apt use packages over HTTPS:
    - `sudo apt update`

```
sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

  - Add the GPG key to the official Docker repository on your system:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

  - Add the Docker repository to the APT sources:

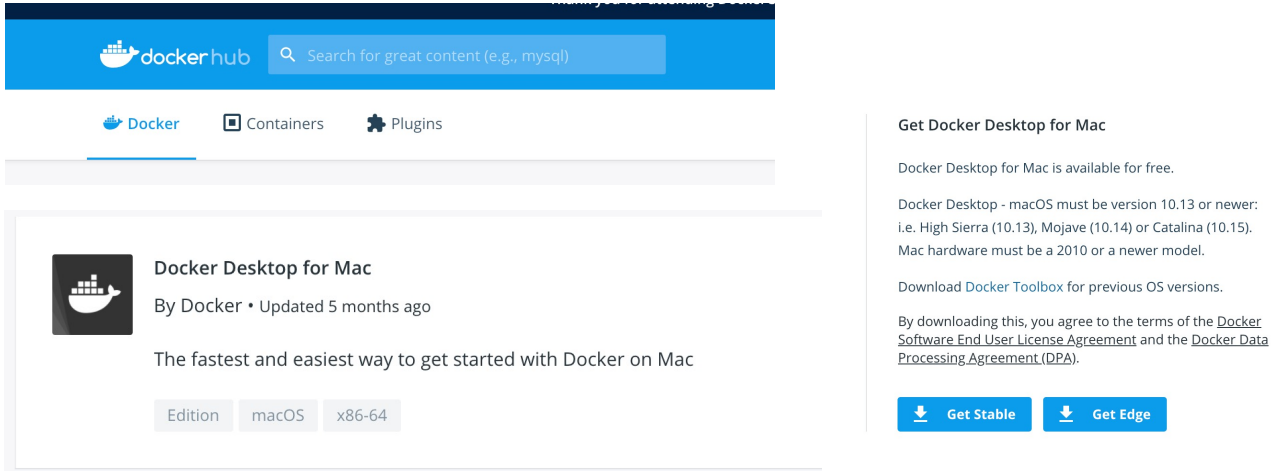
```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"
```

Installation

- Linux
  - Then update the database with the Docker packages from the newly added repository:
    - `sudo apt update`
  - Now, install the Docker
    - `sudo apt install docker-ce`
  - Add your username to the docker group:
    - `sudo usermod -aG docker "user"`

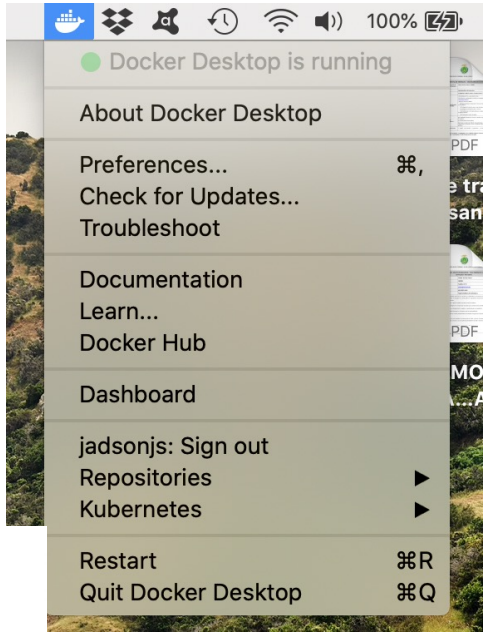
Installation

- Mac OS
  - Go to https://hub.docker.com/
  - Download **Docker Desktop** for Mac
  - Docker is native for Linux, to run on Windows and MacOS we need the Docker Desktop



Installation

- Mac OS
  - Type docker version




```
jadson@MacBook-Pro-de-Jadson ~ % docker version
Client:
 Cloud integration: 1.0.17
 Version:           20.10.8
 API version:       1.41
 Go version:        go1.16.6
 Git commit:        3967b7d
 Built:             Fri Jul 30 19:55:20 2021
 OS/Arch:           darwin/amd64
 Context:           default
 Experimental:      true
```

Docker Introduction

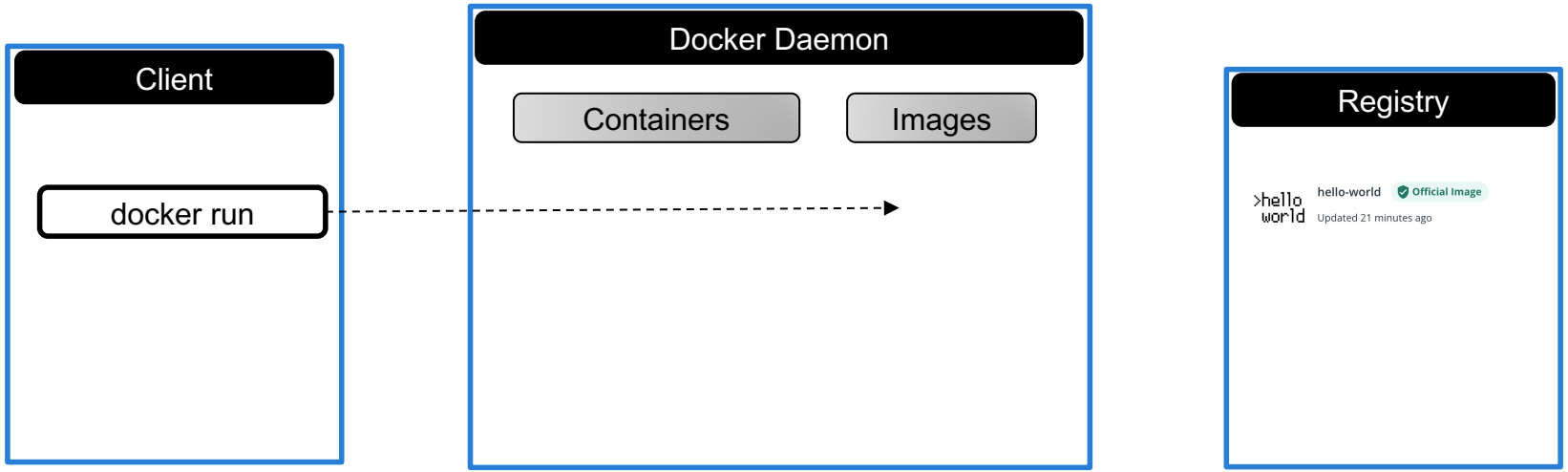
- Hello World

```
docker container run hello-world
```


```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:393b81f0ea5a98a7335d7ad44be96fe76ca8eb2eaa76950eb8c989ebf2b78ec0
Status: Downloaded newer image for hello-world:latest

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

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

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

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

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

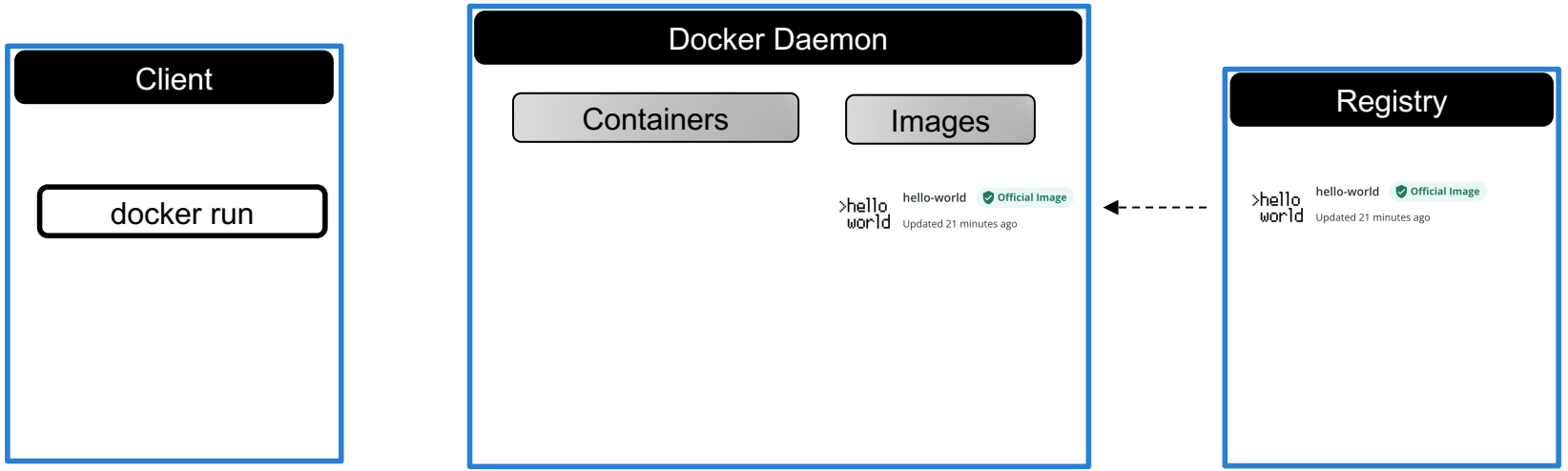
Docker Introduction

- Hello World
  - There is an image “hello-world” locally?



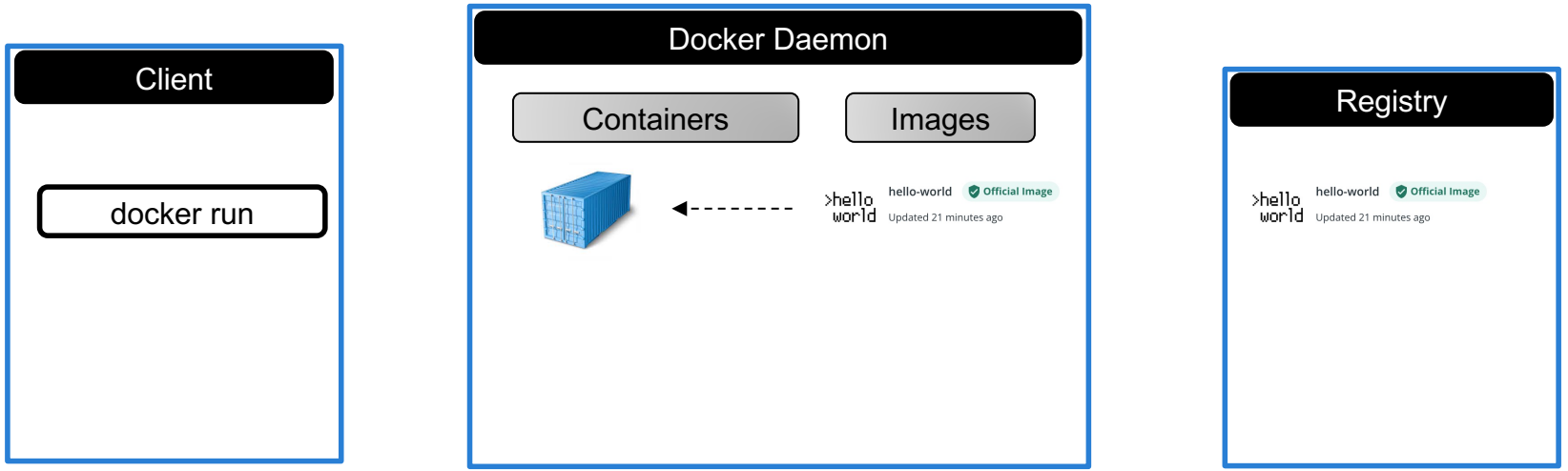
Docker Introduction

- Hello World
  - Download from registry (https://hub.docker.com/)



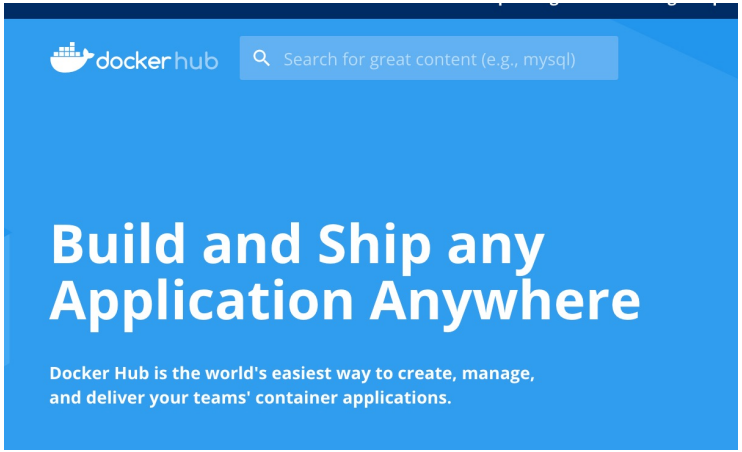
Docker Introduction

- Hello World
  - Run the container that is an instance of “hello-world” image



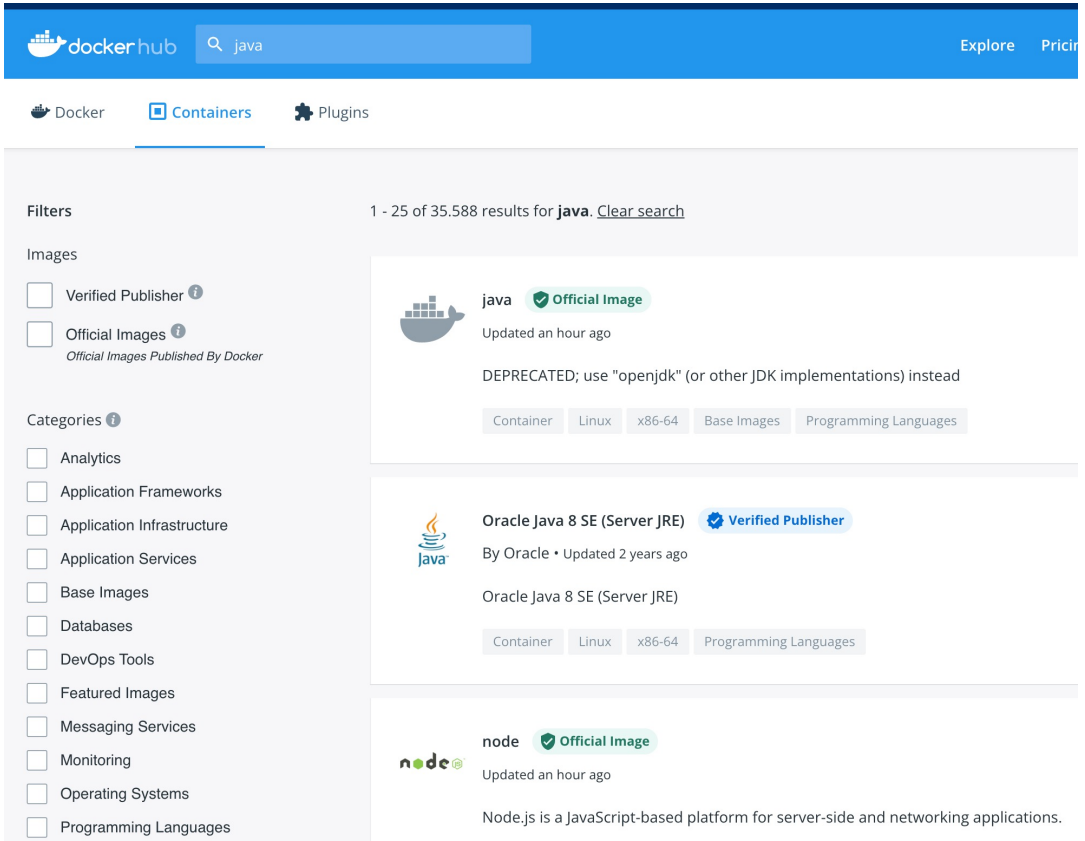
Docker Introduction

- Docker hub
  - Docker hub is the default docker registry
    - It has unlimited public repositories



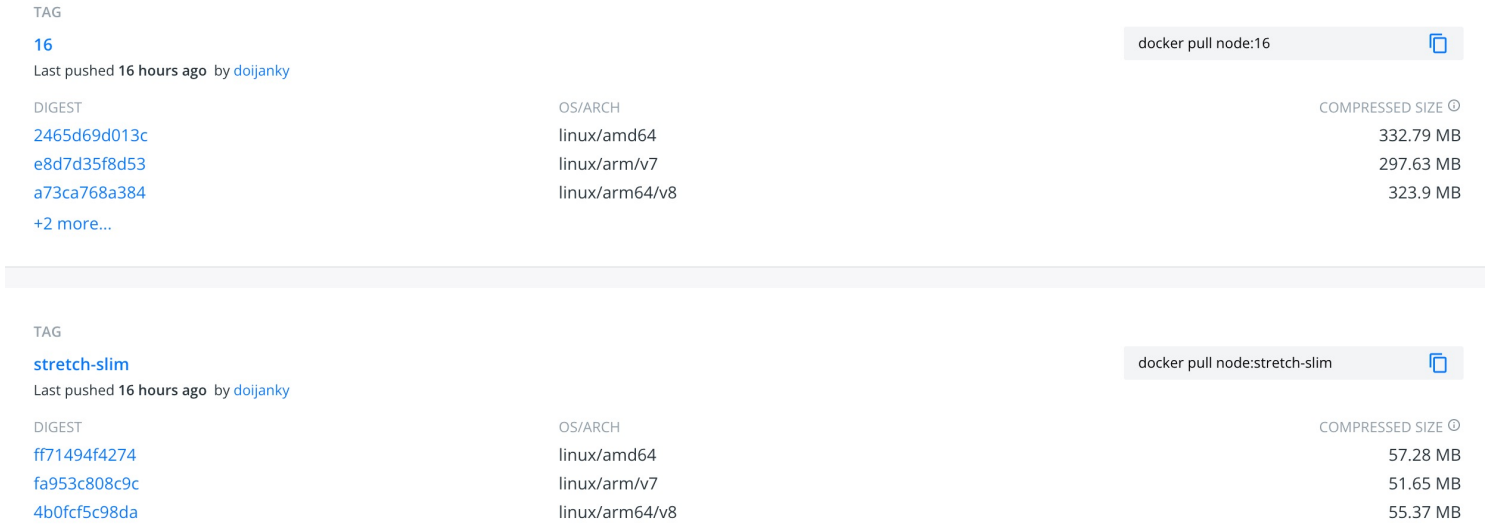
Docker Introduction

- Docker hub
  - You can search for images
  - Give preference to official images



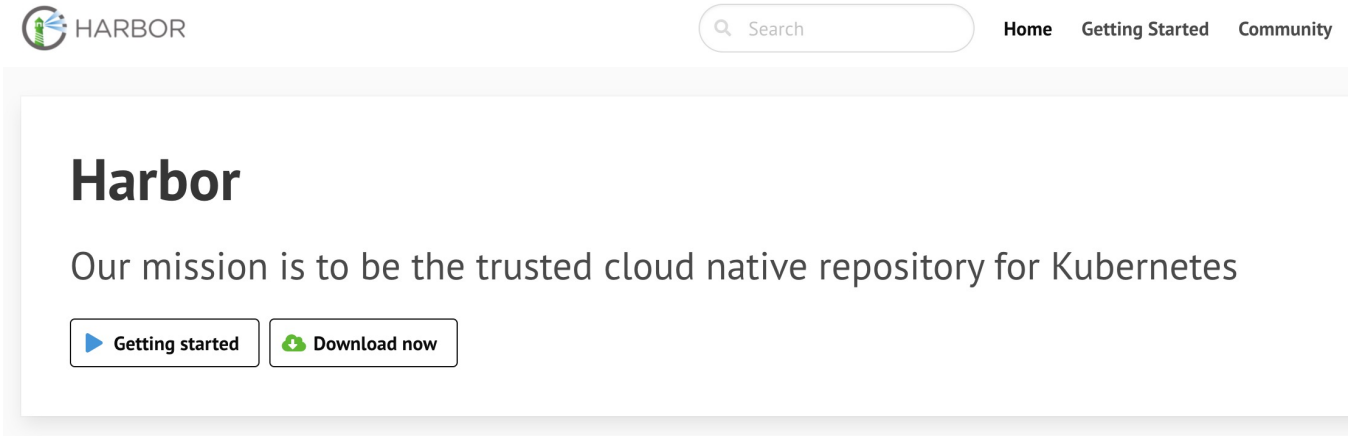
Docker Introduction

- Docker hub
  - Each image has several tags
  - Choose the image that best fits your application requirements



Docker Introduction

- Docker hub
  - You can install local registry to your company.



Docker Introduction

- List the containers running
  - `docker container ls`

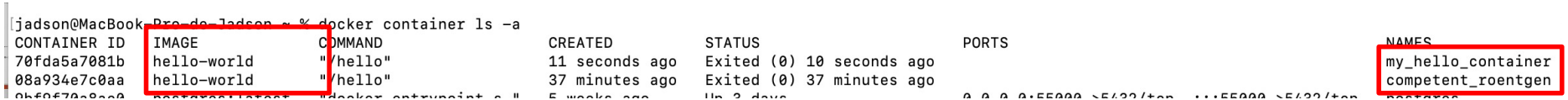
```
jadson@MacBook-Pro-de-Jadson ~ % docker container ls
CONTAINER ID   IMAGE             COMMAND                  CREATED       STATUS      PORTS                                       NAMES
9bf9f70a8ae0   postgres:latest   "docker-entrypoint.s…"   5 weeks ago   Up 3 days   0.0.0.0:55000->5432/tcp, :::55000->5432/tcp   postgres
```

- List all containers
  - `docker container ls -a`

```
jadson@MacBook-Pro-de-Jadson ~ % docker container ls -a
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS                      PORTS                                       NAMES
08a934e7c0aa   hello-world       "/hello"                 30 minutes ago   Exited (0) 30 minutes ago                                               competent_roentgen
9bf9f70a8ae0   postgres:latest   "docker-entrypoint.s…"   5 weeks ago      Up 3 days                   0.0.0.0:55000->5432/tcp, :::55000->5432/tcp   postgres
```

Docker Introduction

- Give a name to the container
  - `docker container run` **`--name my_hello_container`** `hello-world`



Docker Introduction

- Removing a container
  - `docker container` **`rm <ID or NAME>`**
- Restarting the container
  - `docker container` **`restart <ID or NAME>`**
- Stopping the container
  - `docker container` **`stop <ID or NAME>`**

Docker Introduction

- Accessing the container
  - Download the “ubuntu” image, run the container using this image and give me access the **/bin/bash** application inside de container

    - `docker container run -it ubuntu /bin/bash`

In this case, the application argument stay after image name, in the other cases, image name is always the last argument of command

  - Accessing the container when it is running

    - `docker container exec -it ubuntu /bin/bash`

Docker Port Binding

- To have access an application running inside the container, it is necessary to define a port from the port bind. In the format **HOST : CONTAINER**

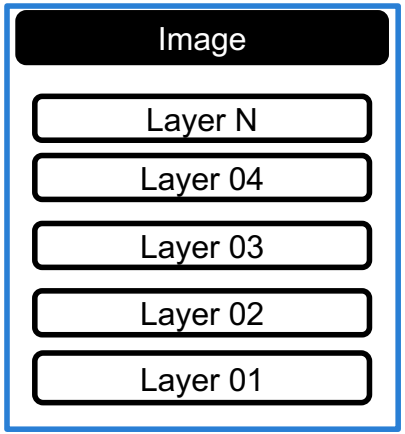
HOST : Mongo DB

- `docker container run -p 27017:27017 mongo`

Bing the mongo DB 27017 port *(mongo db goes up inside de container)* to the 27017 HOST port

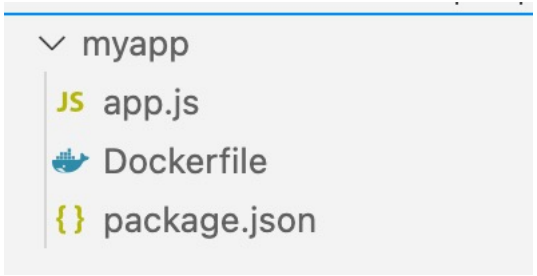
Docker Images

- Docker images act as a template to build a Docker container.
  - Docker images contains layers that install libraries, tools, dependences, files needed to make your application run.
  - Docker images layers increase reusability, decrease disk usage, and speed up docker build by allowing each step to be **cached**.
  - We need to create an image containing our application to run it on a Docker



Docker Images

- The easiest way to create an image is via the Dockerfile file
  - Example of Dockerfile of a simple Node application



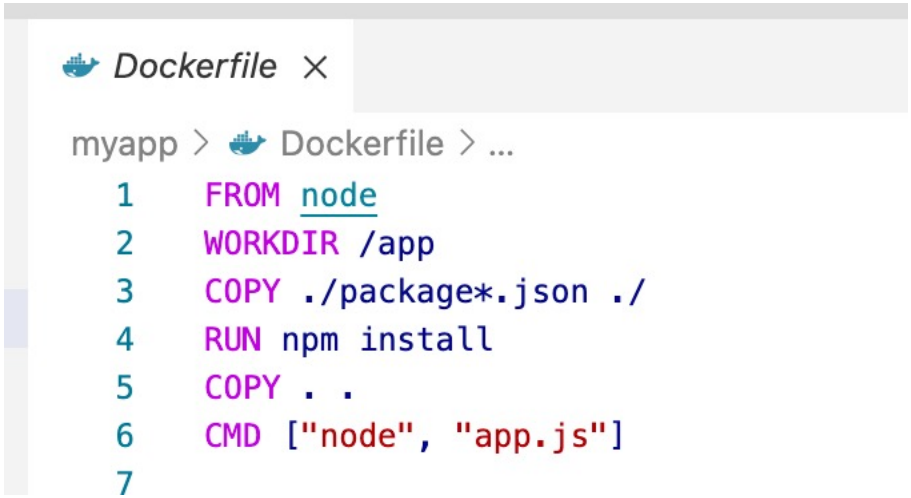
app.js

myapp > app.js > ...

```
1  var http = require("http");
2
3  http.createServer(function (request, response) {
4     // Send the HTTP header
5     // HTTP Status: 200 : OK
6     // Content Type: text/plain
7     response.writeHead(200, {'Content-Type': 'text/plain'});
8
9     // Send the response body as "Hello World"
10    response.end('Hello World\n');
11 }).listen(8081);
12
13 // Console will print the message
14 console.log('Server running at http://127.0.0.1:8081/');
15
```

Docker Images

- The easiest way to create an image is via the Dockerfile file
  - Example of Dockerfile of a simple Node application



Docker Images

- The easiest way to create an image is via the Dockerfile file
  - Starting downloading a **node** image
  - Create a working directory *(WORKDIR == "mkdir /app" and "cd /app")*
  - Copies the .package.json and .package-lock.json file
  - Run ***"npm install"*** to download and install node dependencies
  - Copy the other files of project
  - When the container goes up, execute the command ***"node app.js"***

Docker Images

- Build a image with your node application
  - `docker build . -t jadsonjs/appnode:v1`

Docker Images

- List the images

```
docker image ls
```

```
REPOSITORY          TAG       IMAGE ID       CREATED                  SIZE
jadsonjs/appnode    v1        0f1ecd47832f   About a minute ago       908MB
hello-world         latest    feb5d9fea6a5   25 hours ago             13.3kB
postgres            latest    29dd0a82ea20   5 weeks ago              315MB
```

Docker Images

- Execute your node application inside the docker

```
docker container run -d -p 8080:8081 --name my_appnode_container jadsonjs/appnode:v1
```

Run in background

Bind the **8081** NODE application port to **8080** HOST port

Container name

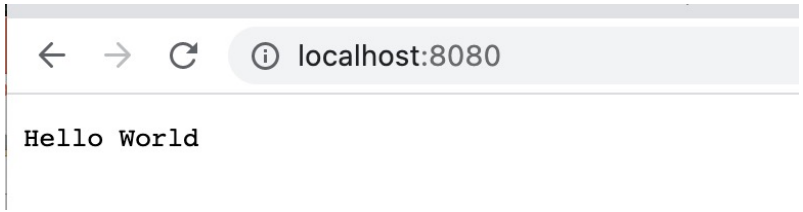
Image name

```
docker container ls
```

```
CONTAINER ID   IMAGE                COMMAND                  CREATED          STATUS          PORTS                                       NAMES
11c82454015b   jadsonjs/appnode:v1  "docker-entrypoint.s…"   51 seconds ago   Up 50 seconds   0.0.0.0:8080->8081/tcp, :::8080->8081/tcp   my_appnode_container
9bf9f70a8ae0   postgres:latest      "docker-entrypoint.s…"   5 weeks ago      Up 4 days       0.0.0.0:55000->5432/tcp, :::55000->5432/tcp postgres
```

Docker Images

- Execute your node application inside the docker



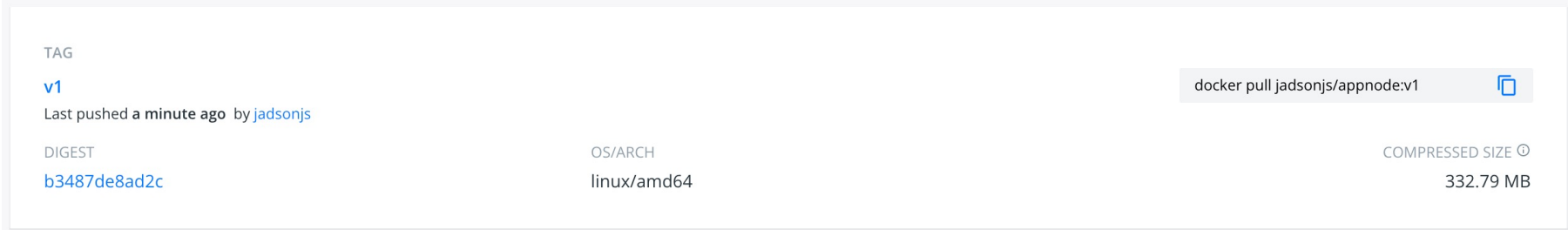
Docker Images

- Publish image to docker hub
  - Publish the image to my account do docker hub as a public image that other people can use.

```
docker login
docker push jadsonjs/appnode:v1
```

Docker Images

- Publish image to docker hub
  - Publish the image to my account do docker hub as a public image that other people can use.



Docker Networks

- If you are going to run 2 or more applications in 2 or more separate containers and need them to communicate. Must create a network between these containers

```
docker network create my-app-net
```

```
docker container run --name api -d -p 8080:8080 --network=my-app-net
jadsonjs/appnode:v1
```

```
docker container run --name mongodb -d -p 27017:27017 -e USER=user -e
PASS=1234 --network=my-app-net mongo:5.0.3
```

Docker Volumes

- Used to save data if you kill and upload the container again. Example: Container of a database.
  - Link container file system with the host file system
  - This can be done via directory bind. In the format **HOST : CONTAINER**, similar with port bind

```
docker container run --name bancoDeDadosMongo -d -p 27017:27017 -e
USER=user -e PASS=1234 --network=api-net -v "/usr/local/mongodb:/data/db"
                                                mongo:5.0.3
```

Bind the **/data/db** mongo db directory with **/usr/local/mongodb** HOST directory
In other words, the data will be save in **/usr/local/mongodb** HOST directory

Docker Volumes

- Creating a volume

```
docker volume create mongodb_vol
docker container run --name mongodb -d -p 27017:27017 -e USER=user -e
PASS=1234 --network=my-app-net -v "mongodb_vol:/data/db" mongo:5.0.3
```

Bind the **/data/db** mongo db directory with a **mongodb_vol** create by the container
mongodb_vol is a directory in HOST machine created and managed by container

Docker Volumes

- Creating a volume

```
docker volume ls
```

```
jadson@MacBook-Pro-de-Jadson myapp % docker volume ls
DRIVER    VOLUME NAME
local     3fef66ee9da691192a945984c6a76d182719238f8cd197642ba3ac885f34733d
local     dfb4a159a2371454812231723195 26e41f294d0b7b82769a94b2ccf3d4d28fda
local     mongo_db_vol
```

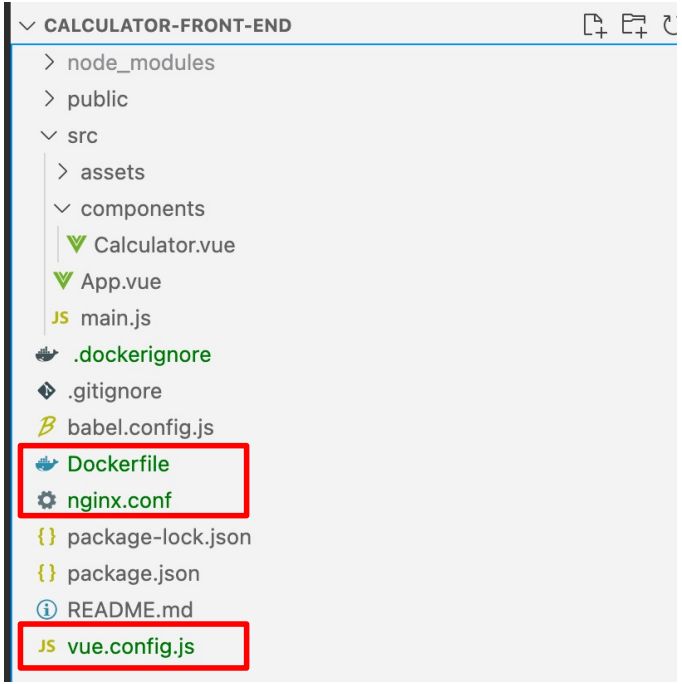
```
docker volume inspect mongo_db_vol
```

```
jadson@MacBook-Pro-de-Jadson myapp % docker volume inspect mongo_db_vol
[
    {
        "CreatedAt": "2021-09-25T01:46:55Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/mongo_db_vol/_data",
        "Name": "mongo_db_vol",
        "Options": {},
        "Scope": "local"
    }
]
```

Vue JS Application on Docker

Vue JS Application on Docker

- Let's create a simple VUE calculator application with de command
  - `vue create calculator-front-end`
- The application will be a simple calculator that will call a back-end application written in Java.
- The front end application will be deployed using nginx server.
- The back end application will be deployed in another Docker container.



Vue JS Application on Docker

- Let's create a simple VUE calculator application
  - We create a **nginx.conf** file in the project root that configure your "calculator" application
    - When somebody type *"/calculator"* on url, the nginx serve should execute the /calculator/index.html of a VUE Single Page Application project
  - This file will be copy into the container file system.

```
server {

  listen       8080;
  server_name  localhost;

  root /usr/share/nginx/html;

  location /calculator {
    try_files $uri $uri/ /calculator/index.html;
  }

}
}
```

Vue JS Application on Docker

- Let’s create a simple VUE calculator application
  - For the vue app work inside nginx server, we have to define a context to VUE application when execute on Docker, we make this in the **vue.config.js** file.

JS vue.config.js U ×

JS vue.config.js > ...

```
1
2
3   module.exports = {
4       publicPath: process.env.NODE_ENV === 'production'
5         ? '/calculator/'
6         : '/'
7     }
8
```

Vue JS Application on Docker

- In the dockerfile file, we define some steps to build our image
  - Build Steps:
    - Download a “node” image
    - Create a temp directory “app” and copy package*.json files to it
    - Install all Vue dependences
    - Copy the other files and make the build of project
  - Execution Steps:
    - Now, download a image of nginx
    - Copy the vue **“dist”** directory content to nginx default directory: **/usr/share/nginx** renaming it.
    - And copy the **nginx.conf** file (in the root of the project) to nginx default configuration directory: **/etc/nginx/**

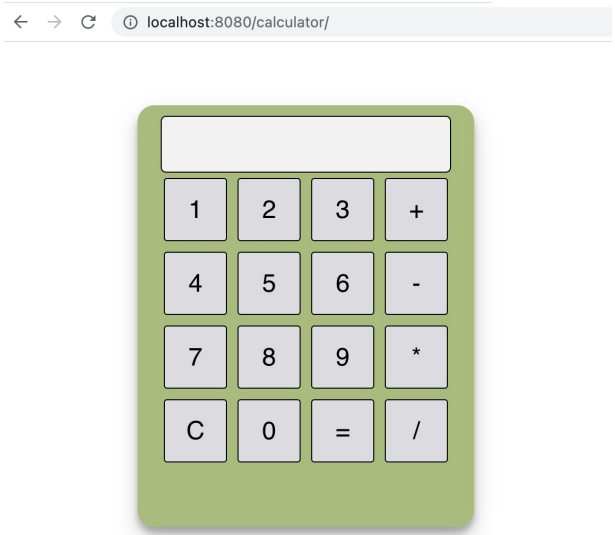
```
FROM amd64/node:14.17.5-alpine3.14 as build-stage
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY ./ .
RUN npm run build

FROM nginx:mainline-alpine as production-stage
COPY --from=build-stage /app/dist /usr/share/nginx/html/calculator/
COPY nginx.conf /etc/nginx/nginx.conf
```

Vue JS Application on Docker

- Now, we need to build the image
  - `docker build . -t calculator-front-end`
- After that, we create a network and run the docker container
  - This same network will be used by the back end container
    - `docker network create calculator-net`
    - `docker run -d -p 8080:8080 –network=calculator-net calculator-front-end`

Vue JS Application on Docker

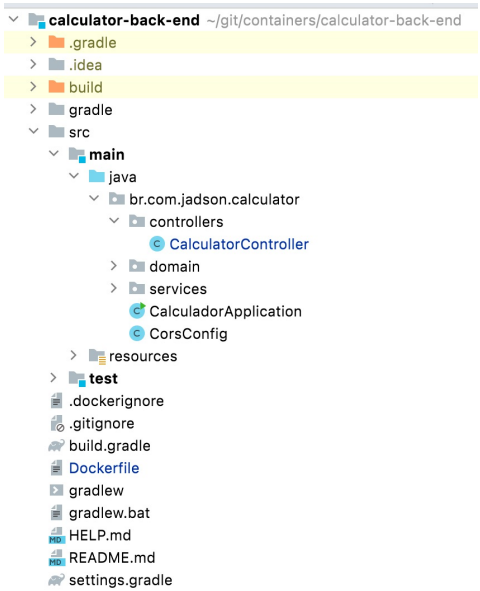
- When the container goes up, the ningx server will goes up also and make deploy of our application following the nginx.conf files
- We can access the application on address: **http://localhost:8080/calculator/**



Java Application on Docker

Java Application on Docker

- We will create a simple **Java spring boot** back-end application for our calculator
  - We will use Gradle as a build tool.




```java
@RestController
@RequestMapping(path = "/calculator")
public class CalculatorController {

    @Autowired
    CalculatorService calculatorService;

    /**
     * Sum of 2 numbers
     *
     * http://localhost:8081/calculator/sum?a=1&b=1
     *
     * @param a
     * @param b
     * @return
     */
    @GetMapping("/sum")
    public ResponseEntity<Integer> sum(
            @RequestParam(value = "a", required = true, defaultValue = "0") int a,
            @RequestParam( value = "b", required = true, defaultValue = "0") int b) {

        int sum = calculatorService.sum(a, b);

        return new ResponseEntity<>(sum, HttpStatus.OK);
    }
```

Java Application on Docker

- Now we will configure dockerfile file to create a docker image
  - Build Steps:
    - Download the gradle image
    - Create a workdir
    - Copy all files to this workdir
    - Run the gradle build
  - Executions steps:
    - Download JDK 11 image
    - Copy and rename the application jar
    - Expose the back end port
    - Execute the Java application when container goes up

```
# build #
FROM gradle:6.4.1-jdk11 as builder
WORKDIR /app
COPY . .
RUN ./gradlew clean build

# execution #
FROM adoptopenjdk:11-jdk-openj9
COPY --from=builder /app/build/libs/calculator*.jar calculator.jar
EXPOSE 8081
ENTRYPOINT ["java","-jar","/calculator.jar"]
```

Java Application on Docker

- We can build the Java back end image with this command
  - `docker build . -t calculator-back-end`

```
jadson@MacBook-Pro-de-Jadson calculator-back-end % docker image ls
REPOSITORY              TAG       IMAGE ID       CREATED          SIZE
calculator-back-end     latest    86071dc66b4c   56 seconds ago   489MB
<none>                  <none>    ab864db90d4e   3 minutes ago    489MB
calculator-front-end    latest    5849fadabe56   22 hours ago     23.7MB
<none>                  <none>    65600cb377c7   22 hours ago     23.7MB
<none>                  <none>    57df53734587   22 hours ago     23.7MB
jadsonjs/appnode        v1        0f1ecd47832f   46 hours ago     908MB
hello-world             latest    feb5d9fea6a5   2 days ago       13.3kB
postgres                latest    29dd0a82ea20   5 weeks ago      315MB
```

Java Application on Docker

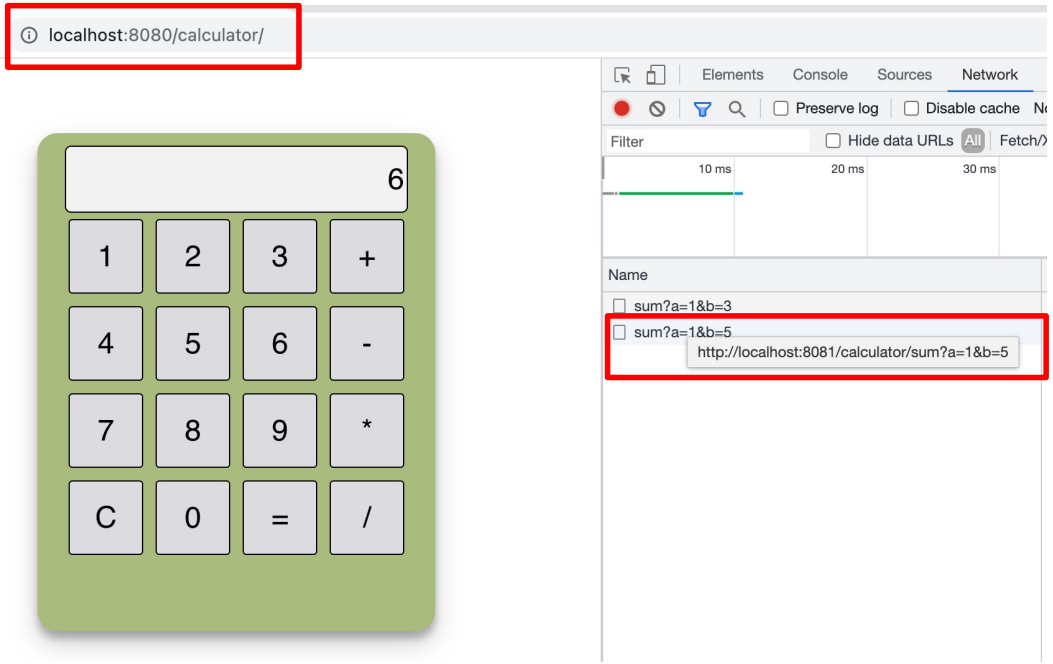
- Now we can run the back end Java application with this command
  - PS.: The front end application should be already running (previous slides)

  - `docker run -d -p 8081:8081 –network=calculator-net calculator-back-end`

```
jadson@MacBook-Pro-de-Jadson calculator-front-end % docker container ls
CONTAINER ID   IMAGE                  COMMAND                  CREATED          STATUS          PORTS                                                NAMES
9f761be11425   calculator-back-end    "java -jar /calculat…"   17 seconds ago   Up 17 seconds   0.0.0.0:8081->8081/tcp, :::8081->8081/tcp            elastic_fermi
a1c43838df88   calculator-front-end   "/docker-entrypoint.…"   43 seconds ago   Up 42 seconds   80/tcp, 0.0.0.0:8080->8080/tcp, :::8080->8080/tcp    dazzling_shirley
```

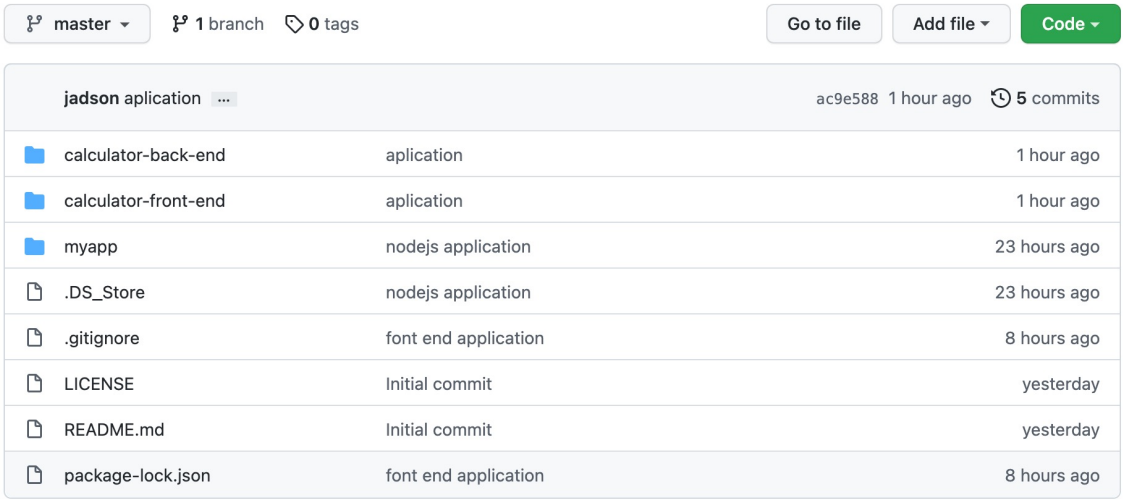
Java Application on Docker

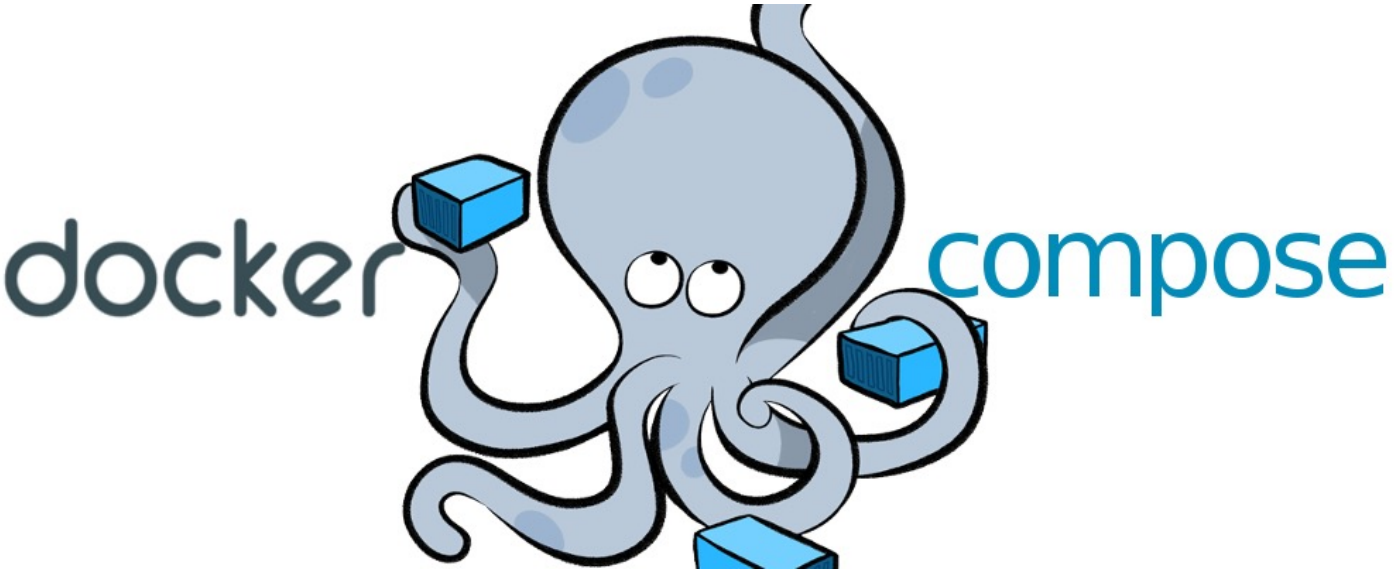
- If everything was ok, now we have a front end VUE.js application run with nginx server on a Docker calling a back end Java application run on Docker also.



Source Code

- https://github.com/jadsonjs/containers




docker
compose

Docker Compose

- Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, you can use a YAML file to configure and start several docker applications.
- So, create a ***docker-compose.yaml*** file with the follow content

docker-compose.yaml U

docker-compose.yaml

```yaml
version: "1.0"

# define the networks that need to be created
# to our services (containers)
networks:
  calculator-net2:
    driver: bridge

# definer our services (containers)
services:
  # the first is the front end application container
  # It uses the calculator-front-end in port 8080:8080
  # Using a calculator-net2 network
  # and should starts after back-end service
  front-end:
    image: calculator-front-end
    ports:
      - "8080:8080"
    networks:
      - calculator-net2
    depends_on:
      - back-end
  # now up a container using the calculator-back-end image
  # on port 8081:8081
  # and networks calculator-net2
  back-end:
    image: calculator-back-end
    ports:
      - "8081:8081"
    networks:
      - calculator-net2

```

Docker Compose

- First we define the version and the networks of our containers
  - We define a new network to the docker compose create it

```
version: "1.0"

# define the networks that need to be created
# to our services (containers)
networks:
  calculator-net2:
    driver: bridge
```

Docker Compose

- Now, we need to define our services (containers)
  - The first one is the VUE js front end application.
  - It uses the ***calculator-front-end*** image in port ***8080:8080***, using the ***calculator-net2*** network and should starts after back-end container

```
services:
  front-end:
    image: calculator-front-end
    ports:
      - "8080:8080"
    networks:
      - calculator-net2
    depends_on:
      - back-end
```

Docker Compose

- Now, we need to define our services (containers)
  - The second one is the Java back end application.
  - It uses the ***calculator-back-end*** image in port ***8081:8081***, using the same ***calculator-net2*** network.

```
back-end:
  image: calculator-back-end
  ports:
    - "8081:8081"
  networks:
    - calculator-net2
```

Docker Compose

- Now, just execute the follow command:
  - `docker compose up -d`
- This command will replace the follow two commands and simplify the docker execution
  - `docker run -d -p 8080:8080 –network=calculator-net calculator-front-end`
  - `docker run -d -p 8081:8081 –network=calculator-net calculator-back-end`
- To stop the execution of containers, type:
  - `docker compose down`

References

- https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04
- https://cli.vuejs.org/guide/deployment.html#docker-nginx
- https://medium.com/bb-tutorials-and-thoughts/how-to-serve-vue-js-application-with-nginx-and-docker-d8a872a02ea8
- https://www.baeldung.com/dockerizing-spring-boot-application
- https://docs.docker.com/language/java/build-images/

References

- https://medium.com/swlh/spring-boot-with-docker-2467db187fa2
- https://dzone.com/articles/docker-with-spring-boot-how-to
- https://docs.docker.com/compose/