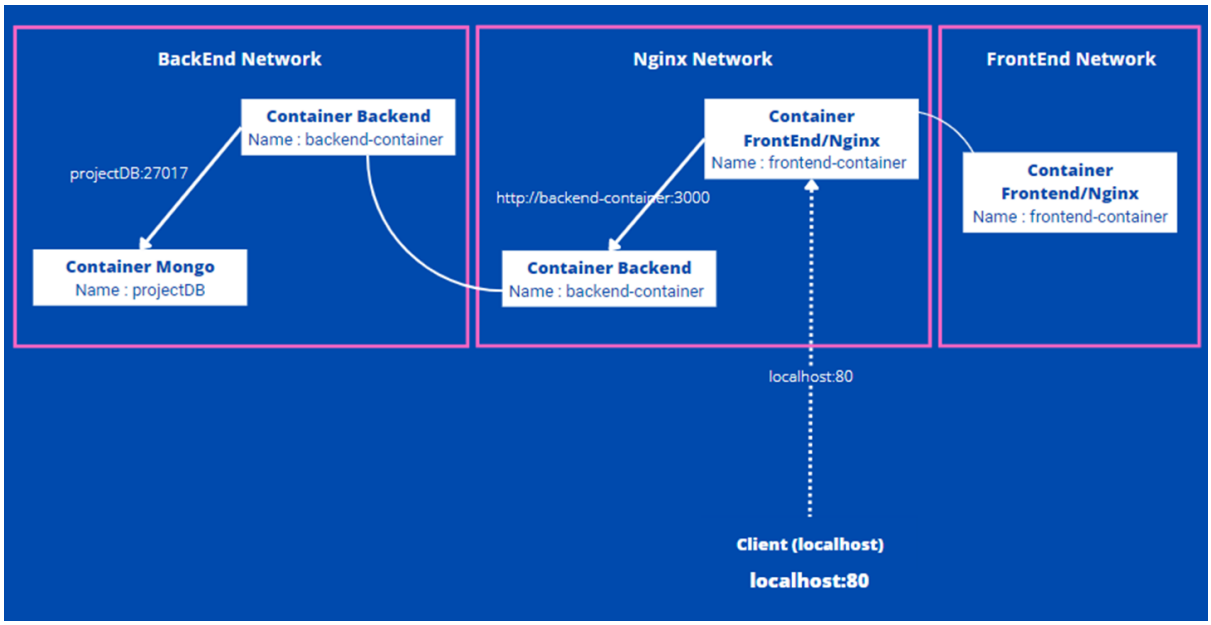
Cloud Computing Project Report

As part of our Cloud Computing course at ESILV, we have to realize a small project. Realize then dockerizer a development project. We have to realize an application of monetary conversion using Angular, NodeJS and mongoDB technologies. The deployment will be done thanks to the nginx technology as a reverse proxy.

The structure of our project will be the following:



Redaction of Dockerfile and construction of Images

Backend Image

Backend Dockerfile

The first line `FROM node:18-alpine` specifies the base image for the Docker container.

The second line `WORKDIR /usr/src/app` sets the working directory for any subsequent commands in the Dockerfile to `/usr/src/app`, which is a common convention for Node.js applications.

The third line `COPY package*.json ./` copies `package.json` and `package-lock.json` (if it exists) from the local directory to the Docker image. The `./` at the end of the command specifies that the files should be copied to the working directory.

The fourth line `RUN npm install` runs the command `npm install` inside the Docker container, which installs all of the dependencies listed in `package.json` and `package-lock.json`.

The fifth line `COPY . .` copies all of the files from the local directory to the Docker image, including the source code for the Node.js application.

The seventh line `CMD [ "node", "index.js" ]` specifies the command to run when the container is started. In this case, it runs the `node` command with `index.js` as the argument, because `index.js` is the entry point for the Node.js application.

Construction of Backend Image

```
docker build -t timoth37/currencyapp:backend-image .
```

This command will build a Docker container with the tag "Dagabwa/backend-container" using the Dockerfile in the current directory. The period at the end of the command specifies that the build context is the current directory.

```
docker login
```

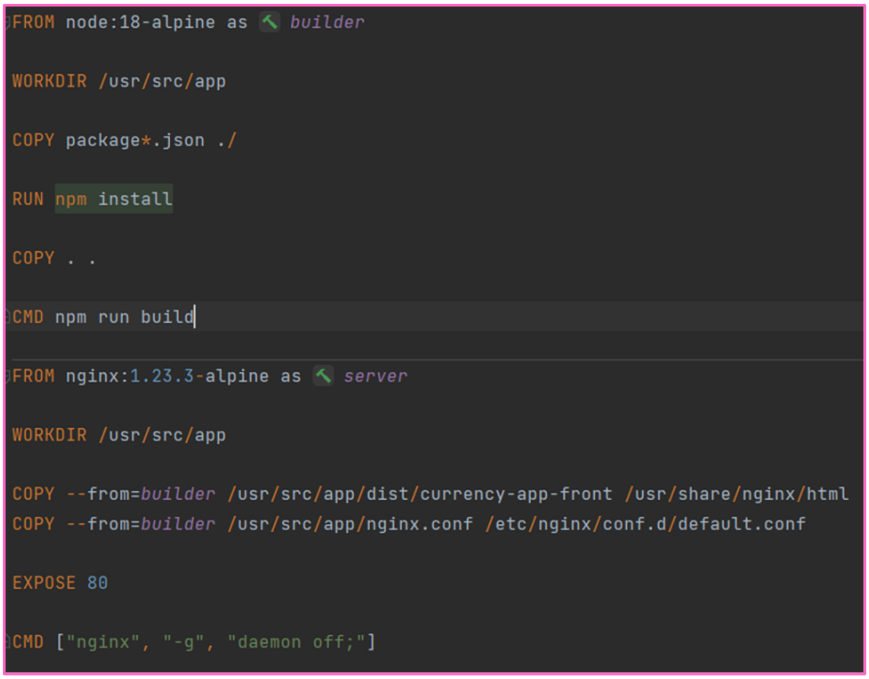
The user is logged in into his dockerhub account

```
docker push timoth37/currencyapp:backend-image
```

The user is pushing his image to his dockerhub account. Everyone can access this image.

Frontend Image

Frontend Dockerfile



The first line `FROM node:18-alpine` specifies the base image for the Docker container. Builder will be the name of this part of the Dockerfile.

The second line `WORKDIR /usr/src/app` sets the working directory for any subsequent commands in the Dockerfile to `/usr/src/app`, which is a common convention for Node.js applications.

The third line `COPY package*.json ./` copies `package.json` and `package-lock.json` (if it exists) from the local directory to the Docker image. The `./` at the end of the command specifies that the files should be copied to the working directory.

The fourth line `RUN npm install` runs the command `npm install` inside the Docker container, which installs all of the dependencies listed in `package.json` and `package-lock.json`.

The fifth line `COPY . .` copies all of the files from the local directory to the Docker image, including the source code for the Node.js application.

The seventh line is `CMD npm run build` It will build the node application. The end result of running this command is a set of files that can be served by a web server and used to run the application in production mode. These files are placed in a “dist” directory

The second part of the Dockerfile starts a new stage for building the final image that will be used to run the application.

The first line `FROM nginx:1.23.3-alpine` specifies the base image for the Docker container.

The second line `WORKDIR /usr/src/app` sets the working directory for any subsequent commands in the Dockerfile to `/usr/src/app.`

The third line `COPY --from=builder /usr/src/app/dist/currency-app-front /usr/share/nginx/html` command copies the built files from the previous stage to the Nginx default document root, which will serve as the entry point for the application.

The fourth line `COPY --from=builder /usr/src/app/nginx.config /etc/nginx/conf.d/default.conf` command copies the nginx.conf file from the next stage to the container's Nginx configuration directory. This file will be redacted in next part.

The fifth line `EXPOSE 80` exposes port 80, which is the port that the application will be served on.

Finally, the `CMD nginx -g daemon off` starts the Nginx server in the foreground and keeps it running thanks to the parameter `-g dameon off`.

Redaction of Nginx config file

The Nginx reverse proxy serves as an entry point for all requests sent to your application. It intercepts requests and redirects them to the back-end if needed. This secures our application by preventing clients from accessing the back-end directly and providing an additional layer of protection against attacks.

```
server {
  listen 80;
  listen [::]:80;
  server_name  localhost;

   location / {
     add_header Access-Control-Allow-Origin *;
     index  /index.html;
     root /usr/share/nginx/html/;
   }

  location /api/ {
    proxy_pass http://backend-container:3000/;
  }
}
```

`server` block is a basic server block for the nginx server.

`listen` directives specify the port and IP address (IPV4 and IPv6 in this case) that the server block listens on.

`server_name` directive sets the server name for this server block. In this case, it is set to "localhost".

`location /` defines the behavior of requests to the root URL "/". The `add_header` directive adds a response header to allow cross-origin resource sharing (CORS), while the `index` directive specifies the default file to serve when a directory is requested. In this case, it is set to "index.html". The `root` directive sets the root directory from which to serve files.

`location /api/` defines the behavior of requests to URLs starting with "/api/". The `proxy_pass` directive forwards these requests to the backend container listening on port 3000.

In summary, this nginx configuration file listens on port 80, serves files from the `/usr/share/nginx/html` directory for requests to the root URL ("/"), and forwards requests to the backend container listening on port 3000 for URLs starting with "/api/".

Construction of Frontend Image

We will not create an image for this part. It will be created when running the docker compose file.

Creation of volume and networks

We now need to create a basic volume. We will name it “project-volume”.

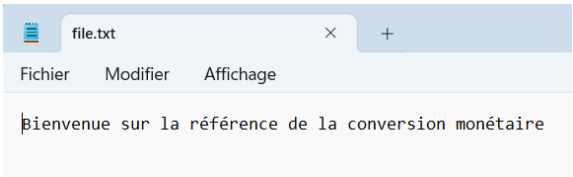
```
docker volume create project-volume
```

We also need to create our three bridge networks.

```
docker network create --driver bridge frontend-network
docker network create --driver bridge backend-network
docker network create --driver bridge nginx-network
```

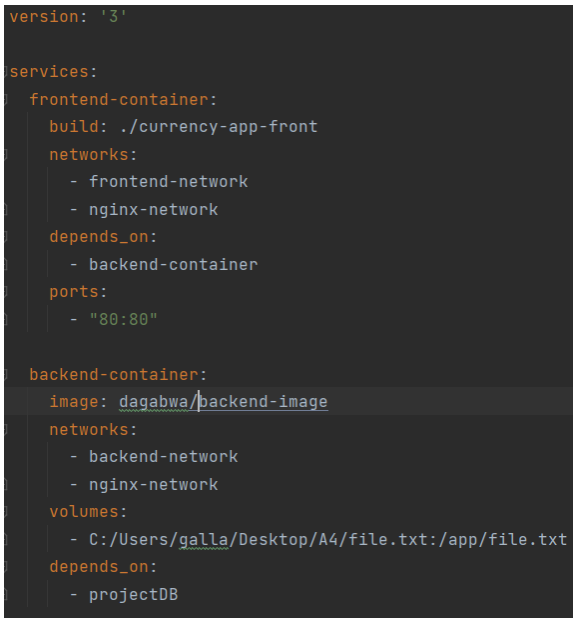
The `--driver` parameter allows us to choose a bridge driver. The "bridge" driver is a built-in driver that creates a private network for containers on a single Docker host.

Creation of the file to be bind mounted



We are going to write a simple text in the file.txt file. This file will be stored locally and bind mounted to the backend container.

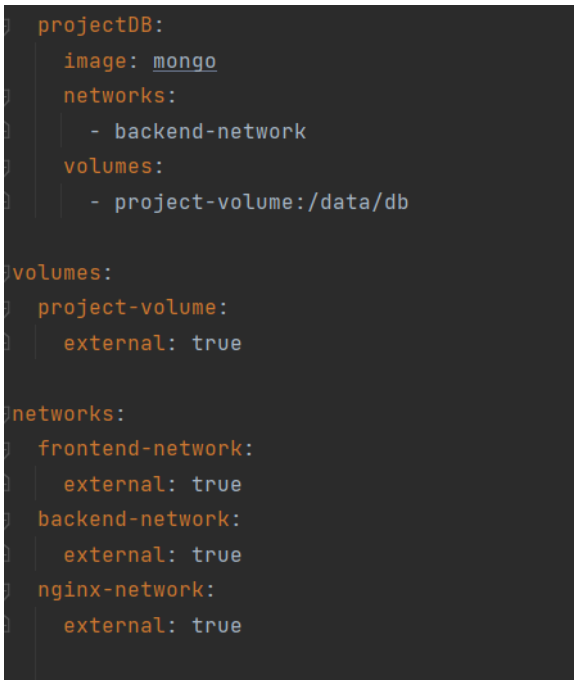
Redaction of docker compose file



```yaml
version: '3'

services:
  frontend-container:
    build: ./currency-app-front
    networks:
      - frontend-network
      - nginx-network
    depends_on:
      - backend-container
    ports:
      - "80:80"

  backend-container:
    image: dagabwa/backend-image
    networks:
      - backend-network
      - nginx-network
    volumes:
      - C:/Users/galla/Desktop/A4/file.txt:/app/file.txt
    depends_on:
      - projectDB
```



```yaml
  projectDB:
    image: mongo
    networks:
      - backend-network
    volumes:
      - project-volume:/data/db

volumes:
  project-volume:
    external: true

networks:
  frontend-network:
    external: true
  backend-network:
    external: true
  nginx-network:
    external: true
```

This file is written in version 3 of the docker-compose format. It defines three services, each of which runs in a separate container.

The `frontend-container` service is built from the Dockerfile located in the `./currency-app-front` directory. Docker compose will automatically build the image thanks to the dockerfile provided. It is connected to the `frontend-network` and `nginx-network` networks, allowing it to communicate with other services running on those networks. This service depends on the `backend-container` service, so the backend must be running before the frontend can start. The service exposes port 80 of the container to port 80 on the host machine.

The `backend-container` service is created from the `dagabwa/backend-image` image and will be pulled from dockerhub. It is connected to the `backend-network` and `nginx-network` networks. It mounts the `C:/Users/galla/Desktop/A4/file.txt` file on the host machine to the `/app/file.txt` path inside the container. This service depends on the `projectDB` service, so the database must be running before the backend can start.

The `projectDB` service is created from the `mongo` image. It is connected to the `backend-network` network. It mounts the `project-volume` volume to the `/data/db` path inside the container.

Finally, the `volumes` section defines a named volume named `project-volume`, which is marked as external. This means that the volume is created outside of this `docker-`

`compose.yml` file and must be managed separately.

The `networks` section defines three named networks: `frontend-network`, `backend-network`, and `nginx-network`, all of which are marked as external.

We now need to start the docker compose and its services :

```
docker compose up -d --build
```

The `-d` parameter allows to avoid printing all the logs in the terminal, and to run our docker compose in the background. The `--build` parameter will force docker compose to rebuild the image even if they exist.

Fill the database

Now that our docker compose services are up, we can fill the database with the needed currencies. To complete that, we will use a python script that we will integrate into a container.

Let’s see our python script :

```python
import pymongo

# Connexion Informations to Existing MongoDB container
HOST = "project-projectDB-1"
PORT = 27017
DB_NAME = "projectdb"

# Connexion to container using pymongo
mongo_client = pymongo.MongoClient(HOST, PORT)
mongo_db = mongo_client[DB_NAME]

#Currencies collection
mongo_collection = mongo_db['currencies']
currencies = [
    {"iso": "EUR", "name": "Euro", "rate": 0.94153},
    {"iso": "USD", "name": "Dollars Américain", "rate": 1},
    {"iso": "GBP", "name": "Livre Britannique", "rate": 0.83230},
    {"iso": "JPY", "name": "Yen Japonnais", "rate": 136.13},
    {"iso": "CAD", "name": "Dollars Canadien", "rate": 1.3606},
    {"iso": "AUD", "name": "Dollars Australien", "rate": 1.4802},
    {"iso": "CHF", "name": "Franc Suisse", "rate": 0.93815},
    {"iso": "INR", "name": "Roupie Indienne", "rate": 81.690}
]

# Insert datas into collection
mongo_collection.insert_many(currencies)
```

With the pymongo library, we will fill the database hosted on the project-projectDB-1 container. We will choose the HOST to be project-projectDB-1, the port to be 27017, the default port on MongoDB, and the database to be projectdb. Thanks to the insert_many() function provided by pymongo, we can insert as many as items we want.

We can now build an image with a Dockerfile. This Dockerfile is pretty much the same as the one we did during the TPs.

```
FROM python

WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt && rm requirements.txt
COPY script.py .

ENTRYPOINT ["python", "script.py"]
```

We are taking the latent (by default) python image as our based image. We are working inside the app directory in the container.

We then copy the requirements.txt (only containing pymongo as it is our only dependency) in the current directory. We copy the requirements.txt file before the rest so it will be cached if we re-build the image several time.

Following this, we are running the command `pip install -r requirements.txt && rm requirements.txt`. It will install the dependencies we need and then erase the file requirements.txt because we don't need it anymore.

We can now copy the script.py file into the directory.

We can then run it with the command `ENTRYPOINT ["python", "script.py"]` which will run the script.

Let's now build the image and run the container.

```
docker build -t timoth37/currencyapp:populate-image .
docker login
docker push timoth37/currencyapp:populate-image
docker run --name populate-container --network backend-network timoth37/populate-image
```

We will build the image, and we will take the habit to push it on our dockerhub because we want it to be accessible by everyone.

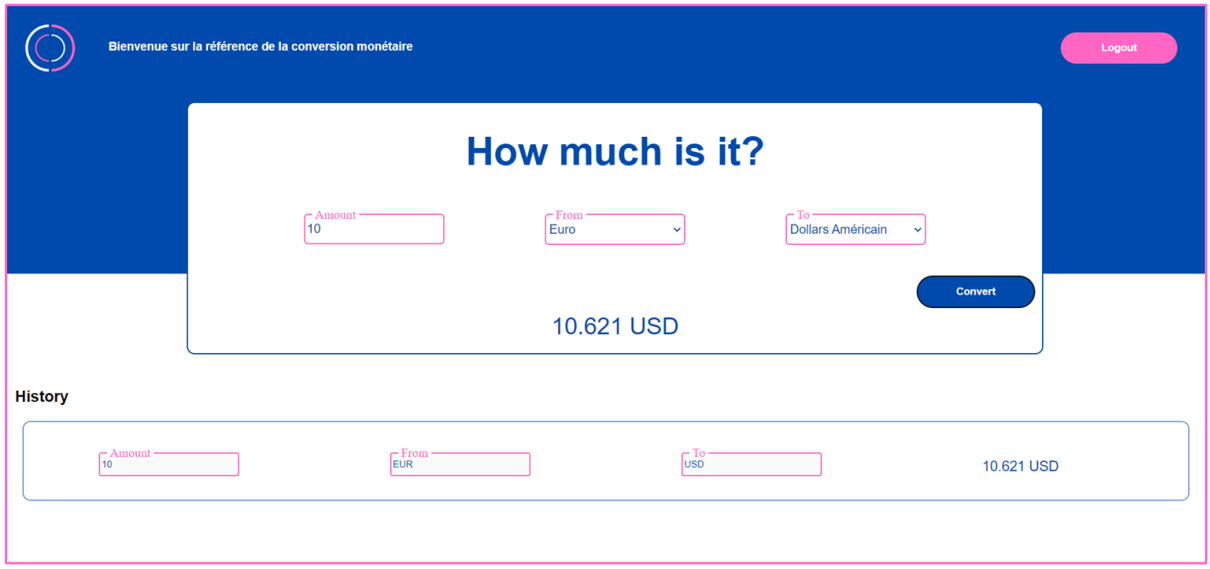
We can then run our container. We will specify a name and the same network as our project-projectDB-1 container, which is backend-network. Our based image will be the one we built previously.

Let’s use our Application

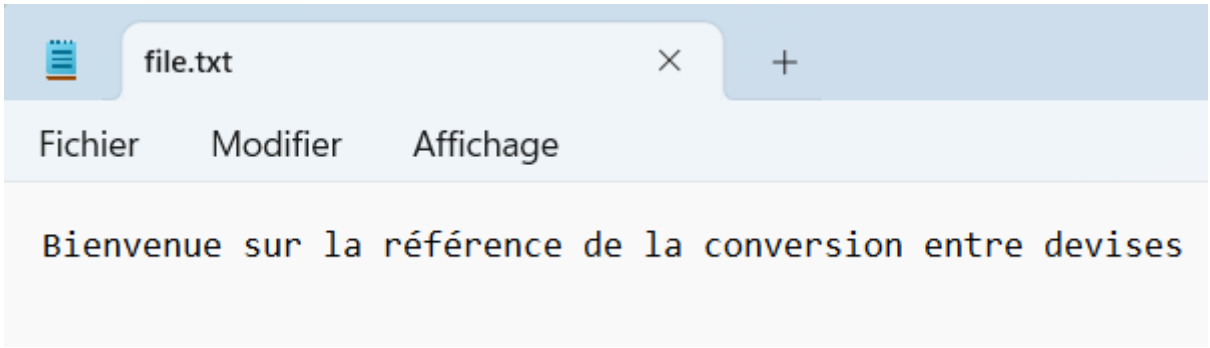
We can use our application.

We can create user thanks to the register page, login as a user thanks to login page.

We can obviously convert some amount of money from a currency to another one.



We can try to change the text in our bind mounted file :



And finally reload the web page, the text has been changed, our bind mount is working fine :