1) **Start a new project called practice1. You're going to create an API. You can use Flask, FastAPI (recommended), Django, or any other python framework you're familiar with… or any other API framework from other language (Golang? Node.JS?) Put it inside a docker**

For the P1 project, we have used **FastAPI** to build the API, as it was the recommended framework for the task. FastAPI's automatic interactive documentation and support for asynchronous programming made it a perfect choice for the project.

practice1.py ● seminar1.py

C: > Users > claud > OneDrive > Documentos > CUARTO > 1trim

```python
from fastapi import FastAPI


app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello, World!"}

```

The application was containerized using **Docker Desktop**, which enabled easy deployment and environment isolation. We used PowerShell to run the necessary commands for creating the Docker container for the FastAPI application.

```
PS C:\Users\claud> cd OneDrive\Documentos\CUARTO\1trim\Codificació\practice1
PS C:\Users\claud\OneDrive\Documentos\CUARTO\1trim\Codificació\practice1> docker run -d -p 8000:8000 --name fastapi-cont
ainer practice1-fastapi-app
```

docker run -d -p 8000:8000 --name fastapi-container practice1-fastapi-app, this will give us an ID of the container, which will also appear at the docker desktop.

To check if our container was running, we used the following command: *docker ps*

By checking the localFastAPI website with the corresponding port, we can see that there is our designed message "Hello World!" which indicates that it is correctly executed.

2) **Put ffmpeg inside a Docker**

The jrottenberg/ffmpeg image already contains a prebuilt version of ffmpeg, so we did not need to install ffmpeg manually.

After pulling the image, we run it as a container with the following command:

docker run --rm jrottenberg/ffmpeg -version

```
PS C:\Users\claud\OneDrive\Documentos\CUARTO\1trim\Codificació\ffmpeg> docker run --rm jrottenberg/ffmpeg -version
ffmpeg version 4.1 Copyright (c) 2000-2018 the FFmpeg developers
built with gcc 5.4.0 (Ubuntu 5.4.0-6ubuntu1~16.04.11) 20160609
configuration: --disable-debug --disable-doc --disable-ffplay --enable-shared --enable-avresample --enable-libopencore-a
mrnb --enable-libopencore-amrwb --enable-gpl --enable-libass --enable-libfreetype --enable-libvidstab --enable-libmp3lam
e --enable-libopenjpeg --enable-libopus --enable-libtheora --enable-libvorbis --enable-libvpx --enable-libx265 --enable-
libxvid --enable-libx264 --enable-nonfree --enable-openssl --enable-libfdk_aac --enable-libkvazaar --enable-libaom --ext
ra-libs=-lpthread --enable-postproc --enable-small --enable-version3 --extra-cflags=-I/opt/ffmpeg/include --extra-ldflag
s=-L/opt/ffmpeg/lib --extra-libs=-ldl --prefix=/opt/ffmpeg
libavutil      56. 22.100 / 56. 22.100
libavcodec     58. 35.100 / 58. 35.100
libavformat    58. 20.100 / 58. 20.100
libavdevice    58.  5.100 / 58.  5.100
libavfilter     7. 40.101 /  7. 40.101
libavresample   4.  0.  0 /  4.  0.  0
libswscale      5.  3.100 /  5.  3.100
libswresample   3.  3.100 /  3.  3.100
libpostproc    55.  3.100 / 55.  3.100
PS C:\Users\claud\OneDrive\Documentos\CUARTO\1trim\Codificació\ffmpeg>
```

ffmpeg/5.0.1-ubuntu2004

3) **Include all your previous work inside the new API. Use the help of any AI tool to adapt the code and the unit tests**

To include everything from our seminar 1 in this API, we include our previous work in the same folder as the folder of the API. This way we can call all of our classes with the python command as if it was a library.

from seminar1 import colorconversor, DCT_coding, wavelet_coding

This way we can call all of our functions inside the practice1 file which contains our API.

4) **Create at least 2 endpoints which will process some actions from the previous S1**

Following the structure of exercise 1, we create various endpoints to try in our localFastAPI web. For example, for the converter from RGB_to_YUV and YUV_to_RGB, we created the following endpoints:

```
# Color Conversion Endpoints
@app.get("/rgb_to_yuv/")
async def rgb_to_yuv(R: int, G: int, B: int):
    Y, U, V = colorconversor.rgb_to_yuv(R, G, B)
    return {"Y": Y, "U": U, "V": V}

@app.get("/yuv_to_rgb/")
async def yuv_to_rgb(Y: float, U: float, V: float):
    R, G, B = colorconversor.yuv_to_rgb(Y, U, V)
    return {"R": R, "G": G, "B": B}
```

After doing the following exercises we found that we needed to change some of our previous endpoints to make it more interactive and/or so it could return the expected result. Since some of the endpoints had some plots to show, we have decided that the easiest and more practical way to show the results is by creating an URL where it would show the results. We create a new output direction and mount it to the original directory such that:

```
# Directory to save generated images
OUTPUT_DIR = "output_images"
os.makedirs(OUTPUT_DIR, exist_ok=True)# Create the
directory if it doesn't exist
# Mount the directory to serve static files
app.mount("/output_images",
StaticFiles(directory=OUTPUT_DIR),
name="output_images")
```

This way we can now make the DCT_encoding endpoints and wavelet endpoint more interactive where any image file from the user can be used to be tested (thanks to the command line `image = Image.open(io.BytesIO(await file.read()))`)

We call the method necessary (normalizing the image due to problems in the computation output) and add the result to a URL direction as:

```
output_file = os.path.join(OUTPUT_DIR, "example.jpeg")
plt.imsave(output_file, decoded_normalized)
# Generate the URL for the saved image
host_url = "http://localhost:8000"
image_url = f"{host_url}/output_images/example.jpeg"
```

```
return {"message": "DCT encoded image generated successfully", "image_url": image_url}
```

Then by just searching the link we will see the result. Since our images are all jpeg and are already compressed, the results may not be returned correctly since by the normalization and/or the method by itself we can have information loss and could not restore the image correctly.

5) **Use docker-compose to launch both and make them interact (i.e., you have a method for conversion, you launch your API and it will call the FFMPEG docker)**

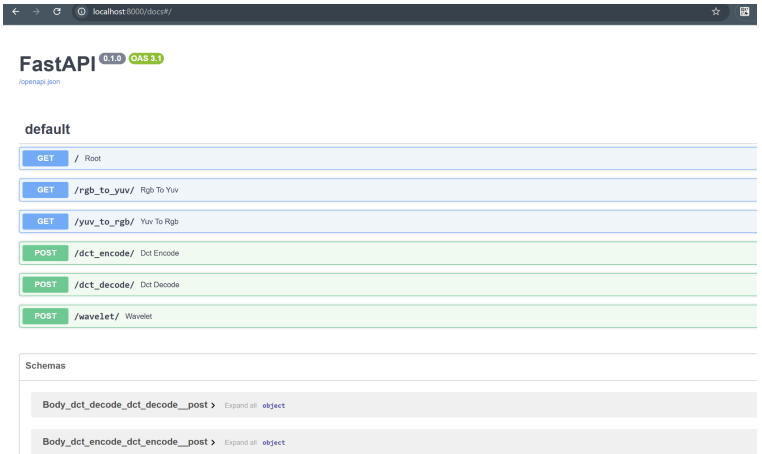
Once done all the previous steps, we called the command *docker-compose up* to make our fastAPI code be launched to the localhost where we can try all possible methods. To make this docker compose up we needed first to create a requirements file to install all necessary libraries or else errors would pop up during the building

```
PS C:\Users\claud\OneDrive\Documentos\CUARTO\1trim\Codificació\practice1> docker-compose up
[+] Building 76.9s (12/12) FINISHED                                                    docker:desktop-linux
 => [fastapi-app internal] load build definition from Dockerfile                                    0.0s
 => => transferring dockerfile: 627B                                                                0.0s
 => [fastapi-app internal] load metadata for docker.io/library/python:3.9-slim                      1.3s
 => [fastapi-app internal] load .dockerignore                                                       0.0s
 => => transferring context: 2B                                                                     0.0s
 => [fastapi-app 1/6] FROM docker.io/library/python:3.9-slim@sha256:6250eb7983c08b3cf5a7db9309f8630d3ca03dd152158  0.0s
 => [fastapi-app internal] load build context                                                       0.0s
 => => transferring context: 350B                                                                   0.0s
 => CACHED [fastapi-app 2/6] WORKDIR /app                                                           0.0s
 => [fastapi-app 3/6] COPY requirements.txt .                                                       0.0s
 => [fastapi-app 4/6] RUN apt-get update && apt-get install ffmpeg libsm6 libxext6 -y              48.0s
 => [fastapi-app 5/6] RUN pip install --no-cache-dir -r requirements.txt                           26.0s
 => [fastapi-app 6/6] COPY . .                                                                      0.0s
```

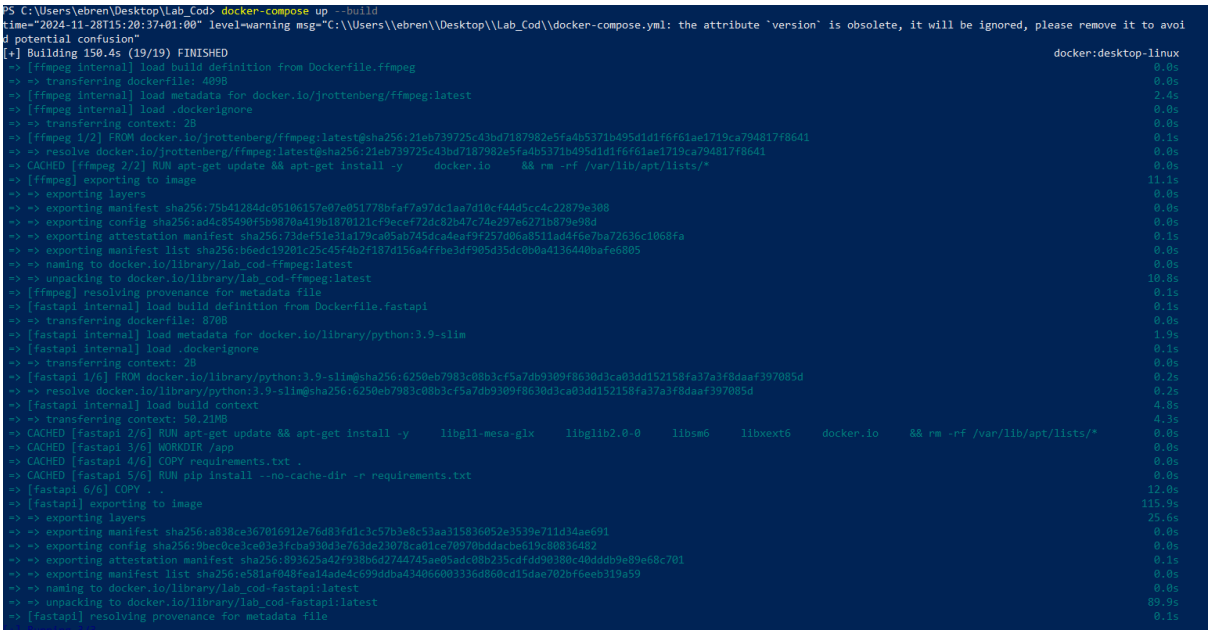
Once the command ended running we saw that the network and the container where created and then the application started as seen in the following screenshot:

```
 => => naming to docker.io/library/practice1-fastapi-app                                            0.0s
 => [fastapi-app] resolving provenance for metadata file                                            0.0s
[+] Running 2/2
 ✔Network practice1_default          Created                                                        0.0s
 ✔Container practice1-fastapi-app-1  Created                                                        0.1s
Attaching to fastapi-app-1
fastapi-app-1  | INFO:     Started server process [1]
fastapi-app-1  | INFO:     Waiting for application startup.
fastapi-app-1  | INFO:     Application startup complete.
fastapi-app-1  | INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
fastapi-app-1  | INFO:     172.18.0.1:41838 - "GET /docs HTTP/1.1" 200 OK
fastapi-app-1  | INFO:     172.18.0.1:41838 - "GET /openapi.json HTTP/1.1" 200 OK
fastapi-app-1  | INFO:     172.18.0.1:52698 - "GET /rgb_to_yuv/?R=246&G=38&B=129 HTTP/1.1" 200 OK
fastapi-app-1  | INFO:     172.18.0.1:41242 - "GET /yuv_to_rgb/?Y=110.566&U=9.07333999999999&V=118.81946999999998 HTTP/1
.1" 200 OK
```

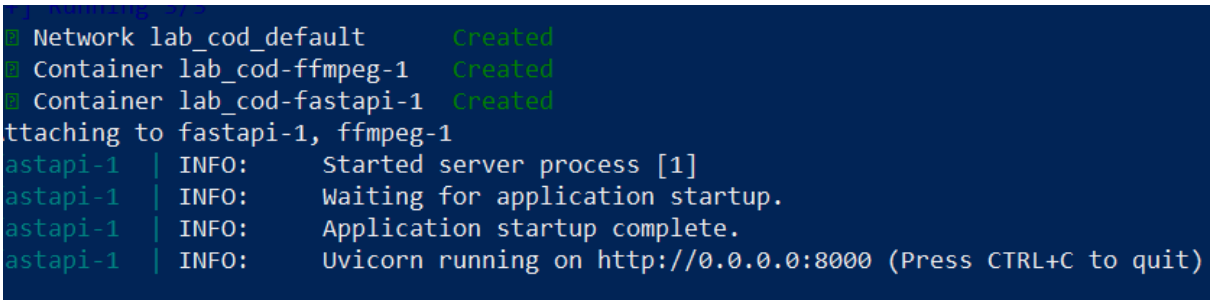
Then by entering the localhost website after the command ended we saw that all of our endpoints were on the localhost and also that they worked out fine. We made the necessary modifications (as specified in previous exercise) so the interaction was more comfortable.



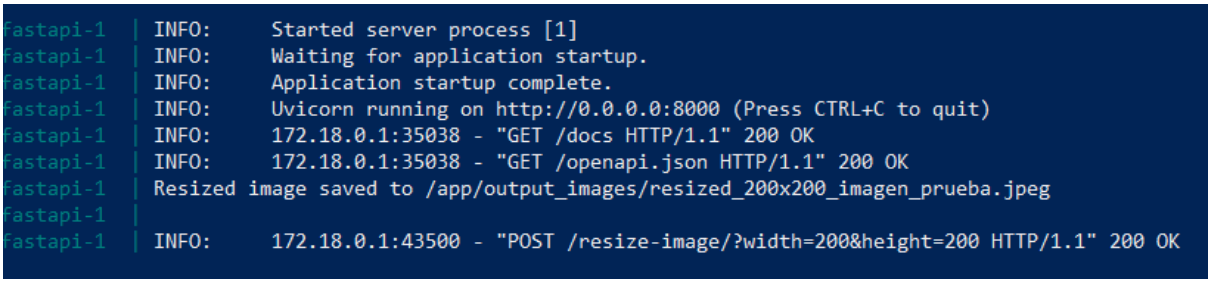
To ensure proper functionality, we created two separate Dockerfiles: Dockerfile.fastapi for the FastAPI application and Dockerfile.ffmpeg for FFMPEG. We then executed the following command:



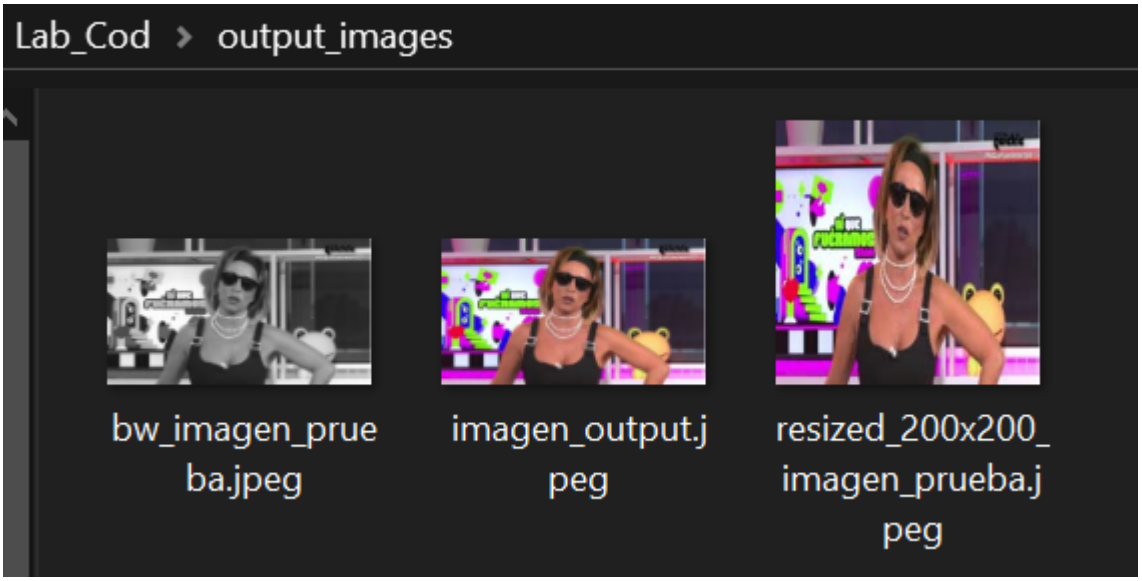
After doing docker-compose up –build:



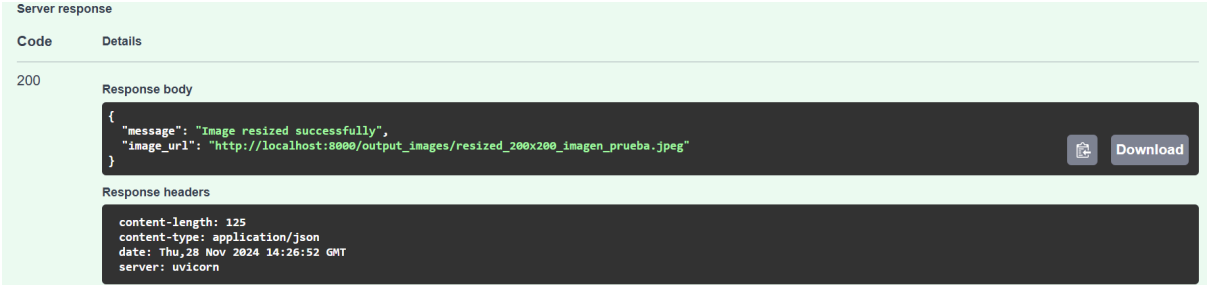
After doing the ffmpeg resize call:



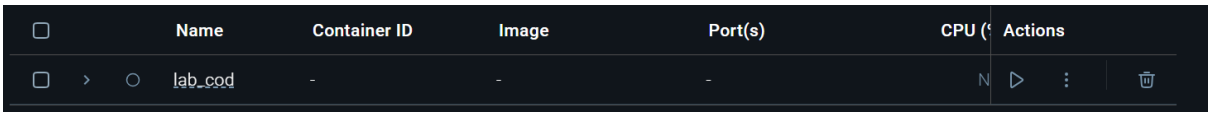
To test the integration, we made a conversion call to FFMPEG through our API. The resulting outputs were saved in a shared folder (output_images), which could be accessed via dynamically generated URLs. This setup made it easy to visualize the results in a web browser.



For example, after performing a resizing operation with FFMPEG, the output was saved and made available at:



Inside Docker Desktop, we confirmed that the single container was running as expected. This container hosted both the FastAPI application and FFMPEG processes.



Additionally, the "Images" section in Docker Desktop displayed the respective container images for FastAPI and FFMPEG, confirming that they had been built and deployed correctly. This interface provided a clear overview of the system's status and allowed us to monitor

performance during testing.