Practice 3: **REST-API**

PTI - G12

Josep Martínez

Adrià Martínez

Index

1. Environment

For this lab session we will be using ***Visual Studio Code*** and ***PyCharm*** as our IDE, ***Python*** as the practice programming language and ***Ubuntu*** (Linux) as the host OS.
For the server, as the guide says, we will use ***Node*** and ***curl*** for the requests and server deployment.

2. Implementation of the REST-API

This section is dedicated to detail how we have implemented each endpoint of the REST-API proposed by the practice guide. You can find in the Annex the screenshots of the application output.

2.1 (Extra) Main page display

As an extra, we decided to create a main page with a simple title in order to check that the access to the root point of the service was operational.

```
JavaScript
app.get('/', (req, res) => {
 res.sendFile(path.join(__dirname, './index.html'));
})
```

2.2 Request list of all rentals

In order to implement the endpoint to list all rentals we used the following code.

```
JavaScript
app.get('/list', (req, res, next) => {
 /* Check if file exists and create it if not */
 if(!fs.existsSync('rentals.json')){
       res.send("No rentals registered yet")
 }
 else{
       /* Read file */
       rentalsFileRawData = fs.readFileSync('rentals.json');
       rentalsJSON = JSON.parse(rentalsFileRawData);
       res.send(rentalsJSON);
 }
 res.end();
})
```

The code checks if the json file exists. If not, it displays a message to warn the user. Otherwise it reads the json file, parses the file as a json object and it returns the request with that object.

2.3 Request of a new rental

In order to implement the endpoint to register a new rental order we used the following code.

```javascript
app.post('/newrental', (req, res, next) => {

  /* Check if file exists and create it if not */
  if(!fs.existsSync('rentals.json')){
       rentalsJSON = {"rentals": []};
       fs.writeFileSync("rentals.json", JSON.stringify(rentalsJSON));
  }

  /* Read file */
  rentalsFileRawData = fs.readFileSync('rentals.json');
  rentalsJSON = JSON.parse(rentalsFileRawData);

  /* Create new rental JSON and add it into the parsed array */
  new_rental = {
       "maker":req.body.maker,
       "model":req.body.model,
       "days":req.body.days,
       "units":req.body.units
  }
  rentalsJSON['rentals'].push(new_rental)

  /* Write array into the file */
  fs.writeFileSync("rentals.json", JSON.stringify(rentalsJSON));

  console.log("New rental created: ", new_rental)

  /* Return 201 HTTP status code (created) */
  res.status(201)
  res.end();
})
```

The method used to create a new rental instance was the following:

1. Check if the json file was already created (in case we found ourselves in the first execution), and create it otherwise.
2. Read the json file.
3. Create a new json object using the correct labels for the car creator, model, days of rental and units of cars to rent. Once created, push to the parsed JSON array.
4. Once the parsing is done, write it to the json file and return the http status.

3. Additional tasks

3.1 Dockerization of the Service

To Dockerize our web application we have used the proposed template by the practice guide, adding our *index.html* as the welcome page in the COPY command.

```
Unset
FROM node:14

WORKDIR /usr/src/app

COPY package*.json index.html ./

RUN npm install

COPY server.js .

CMD [ "node", "server.js" ]
```

This *Dockerfile* pulls the ***Node*** (version 14) image from DockerHub (Docker's official registry), then ***/usr/src/app*** is set as the root directory in order to work from there.

The next step is to copy all files whose name is starting with "*package*" and the *index.html* into the current working directory (*/usr/src/app*).

Then we initialize the *Node* application and copy the ***server.js*** into the folder. We copy this file later to preserve the content, as the "*npm install*" creates one by default that would overwrite the content of ours.

Finally we start up our web application by running "*node server.js*".

We build the image.

```
Unset
docker build -f Dockerfile -t carrental .
```

And finally we run the container redirecting the ports 8080 and 8443.

```
Unset
docker run --name carrental -d -p 8080:8080 -p 8443:8443 carrental
```

3.2 CI/CD implementation

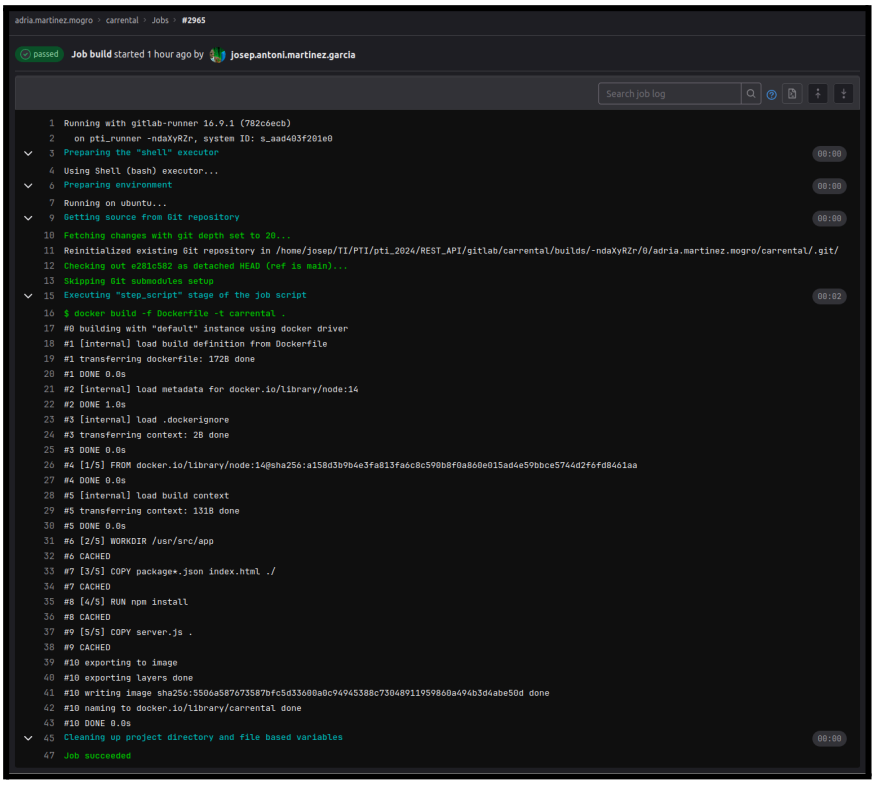
The first thing we have done is to create a repository at FIB's GitLab hosted in the URL: https://repo.fib.upc.es/adria.martinez.mogro/carrental
Next, we have copied our ***carrental*** project developed in the previous section. Then we have written the *.gitignore* file to prevent git from adding the *node_modules/* to our commits.

3.2.1 Installing, registering and running a GitLab Runner

As the guide says, the first thing to do is to install **gitlab-runner** and add the *gitlab-runner* user into the docker group. The next step is to define a CI/CD pipeline by creating our ***gitlab-ci.yml*** file. This mechanism triggers the pipeline and every time we push changes to our repository the defined jobs in this file are executed sequentially. We have defined three jobs:

Job 1: build
We build our image.

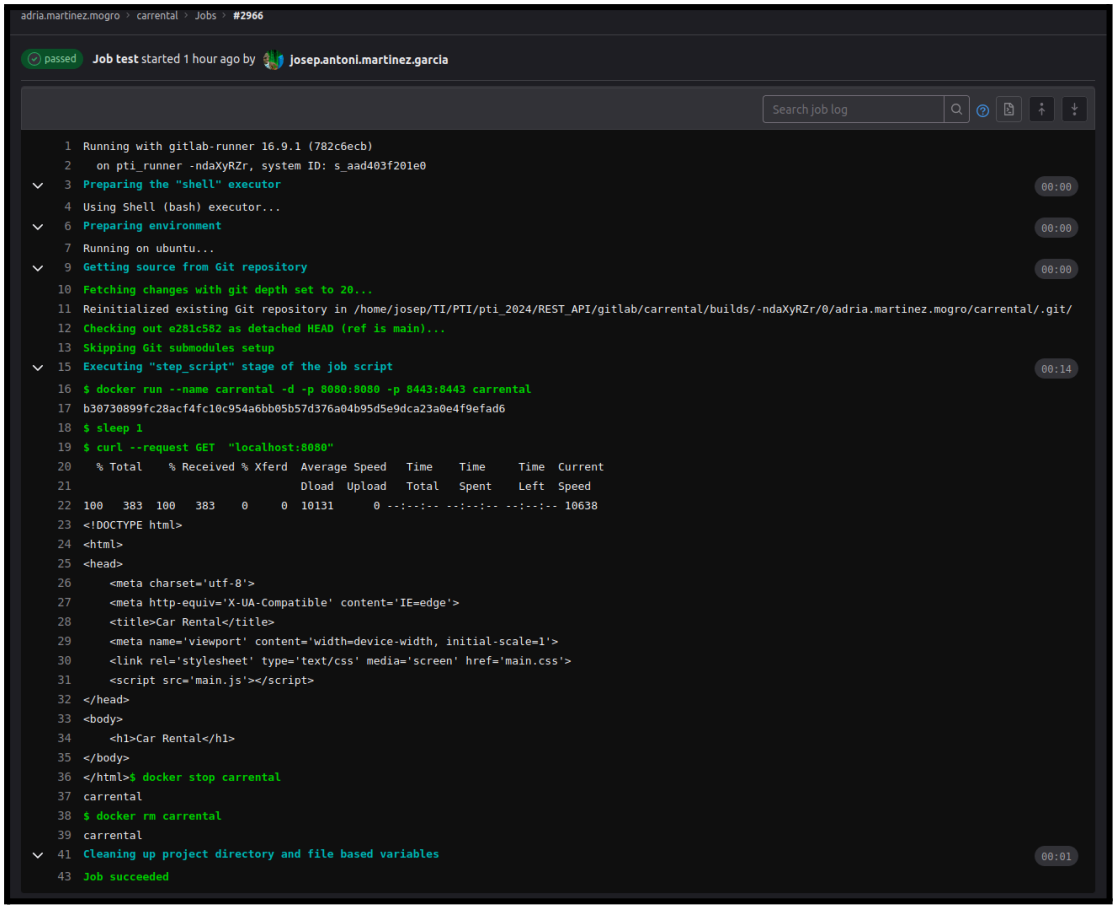
```
Unset
build:
 script:
 - docker build -f Dockerfile -t carrental .
```



Job 2: test

We run the container with the previous image, and then we forge a request to check that the container is running correctly. Finally we stop and remove the container.

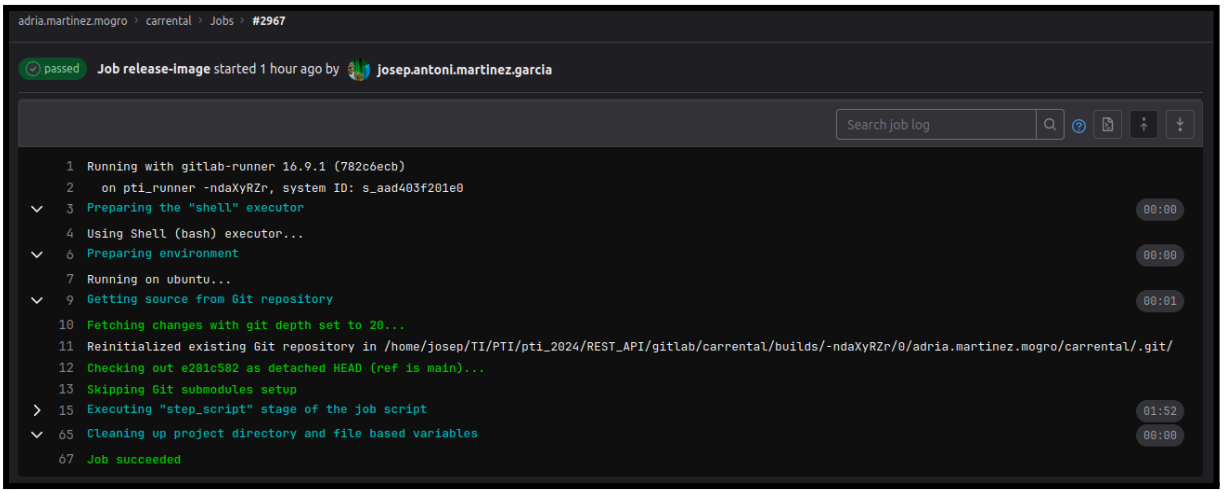
```
Unset
test:
 script:
  - docker run --name carrental -d -p 8080:8080 -p 8443:8443 carrental
  - sleep 1
  - curl --request GET "localhost:8080"
  - docker stop carrental
  - docker rm carrental
```



Job 3: release-image

This final test, rebuilds our updated image and pushes it into our local registry.

```
Unset
release-image:
 script:
  - docker tag carrental:latest localhost:5000/carrental
  - docker push localhost:5000/carrental
  - docker image remove localhost:5000/carrental
  - docker pull localhost:5000/carrental
```



We have created our local registry with the *registry* image running at port 5000.

```
Unset
docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

3.2.2 Considerations

In the first place, we had some troubles with running jobs due to permission issues. The problem was that the **gitlab-runner** was unauthorized to run docker commands, which requires privileged permissions. Following this page, we found the solution by creating a docker group and adding our user into it. This way, running docker commands with sudo is not required anymore.

4. Conclusions

While working on this project we have learned a lot of useful things. While implementing our web service we could learn how to properly work with endpoints and how the json files are being used in Javascript, being capable of using a json file, parsing and pushing its parameters in order to get the correct information we need.

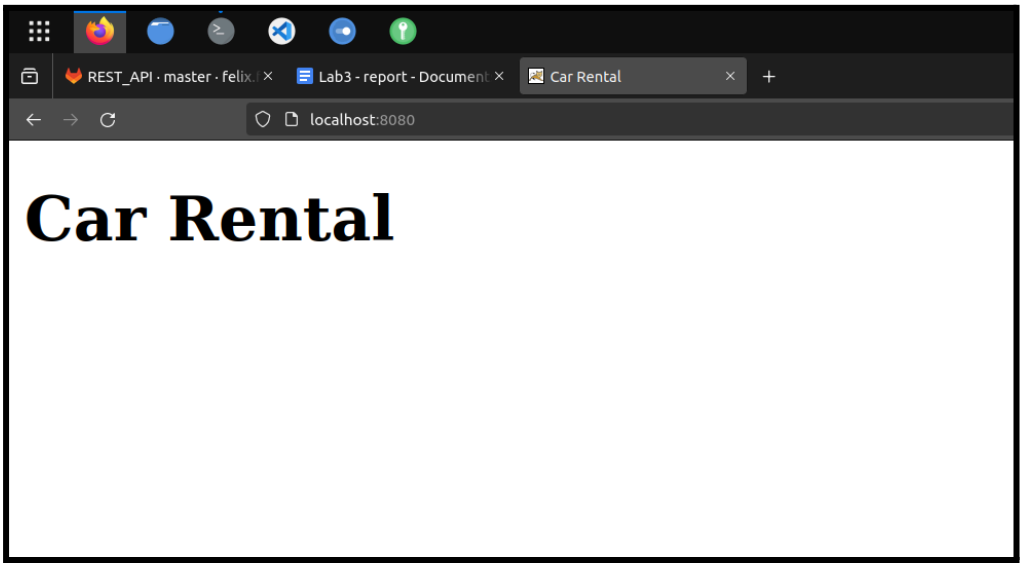
We also reinforced the knowledge of Docker that we had learnt in previous projects, dockerizing our server to be able to build and deploy a service image.

Finally we embarked in a deep learning of CI/CDs in order to make our repository consistency better. A knowledge which we are willing to use in our further projects and specially in our PTI main project.

5. Annex

---

index.html



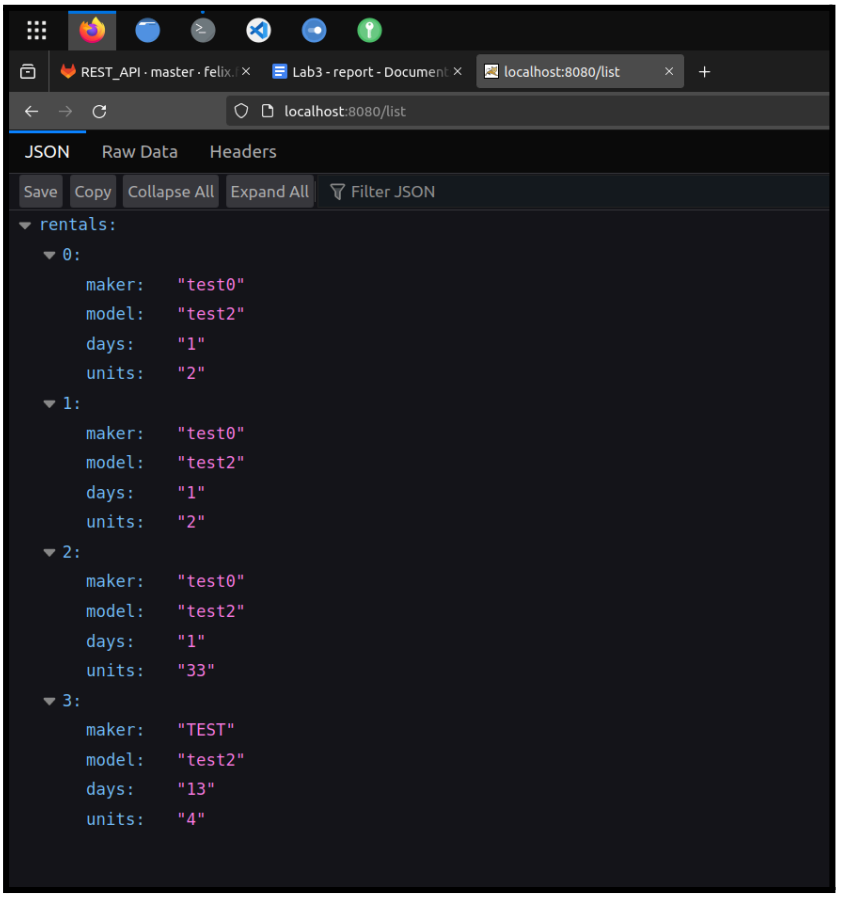
New car rental

By running:

```
Unset
curl -X POST -H "Content-Type: application/json" -d '{"maker":"TEST",
"model":"test2", "days":"13", "units":"4"}' http://localhost:8080/newrental
```

We receive the request in our server's log.

Car rental list



If we check the content of *rentals.json*.

6. References

---

https://phoenixnap.com/kb/docker-permission-denied