Docker

What is Docker?

Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers.

Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels.

All containers are run by a single operating system kernel and therefore use fewer resources than virtual machines.

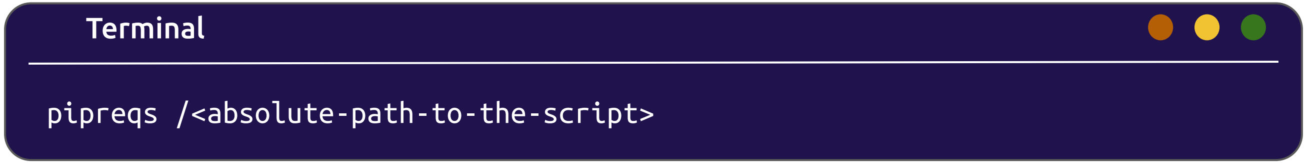
A possible Similitude (NOT LITERALLY)

vs

Run our first container

Get the requirements

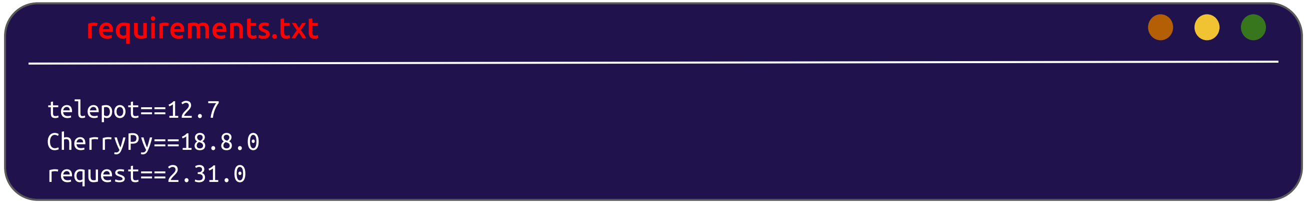
First, we need to obtain the list of the python libraries needed to run our application. We can do that in a lot of ways, the simplest one is to use `pipreqs` and run the command.



Terminal

```
pipreqs /<absolute-path-to-the-script>
```

Executing this command, will create a file called `requirements.txt` in the same folder. For example:

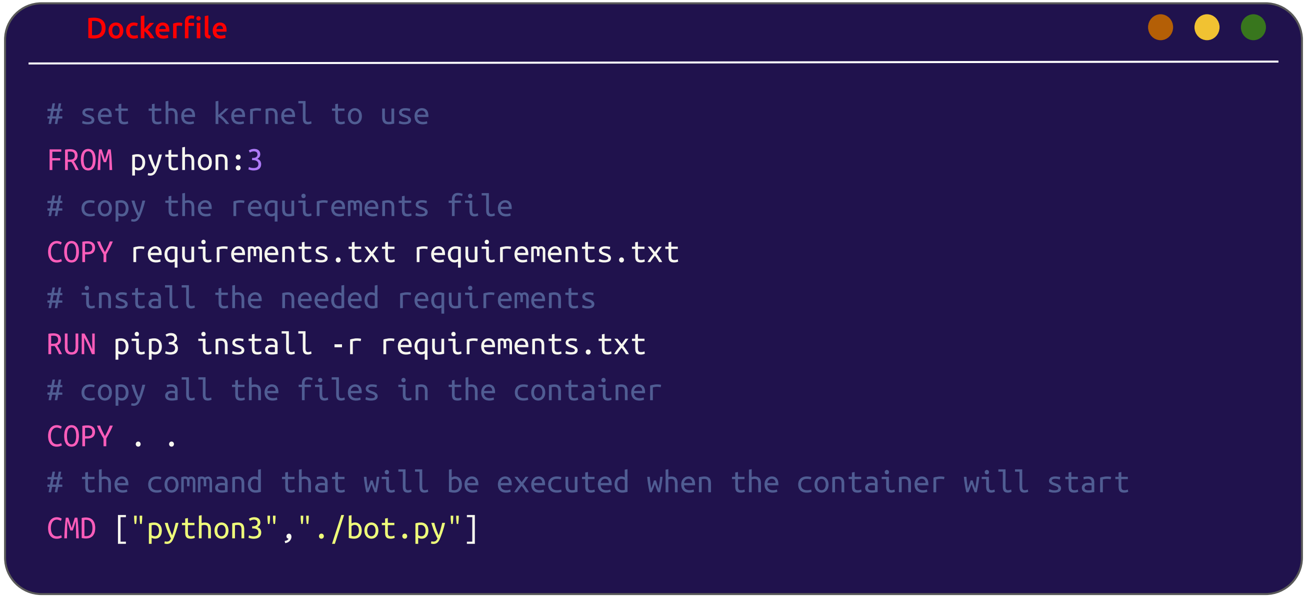


requirements.txt

```
telepot==12.7
CherryPy==18.8.0
request==2.31.0
```

Create Dockerfile

We need to create **Dockerfile** that is the crucial component needed to run a container with **Docker**.



```
# set the kernel to use
FROM python:3
# copy the requirements file
COPY requirements.txt requirements.txt
# install the needed requirements
RUN pip3 install -r requirements.txt
# copy all the files in the container
COPY . .
# the command that will be executed when the container will start
CMD ["python3","./bot.py"]
```

Build the image

To create the image of our application (something like an .exe file, we need to run a simple command:

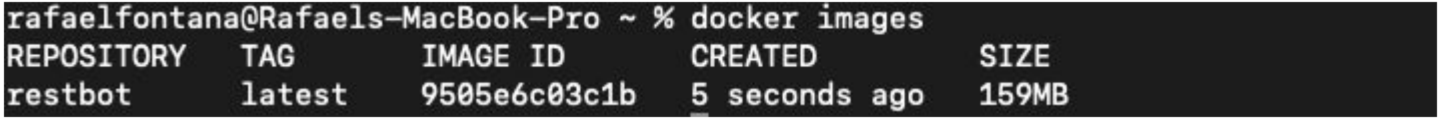
Terminal

```
docker build -t restbot .
```

After that, you can list the images using the following command:

Terminal

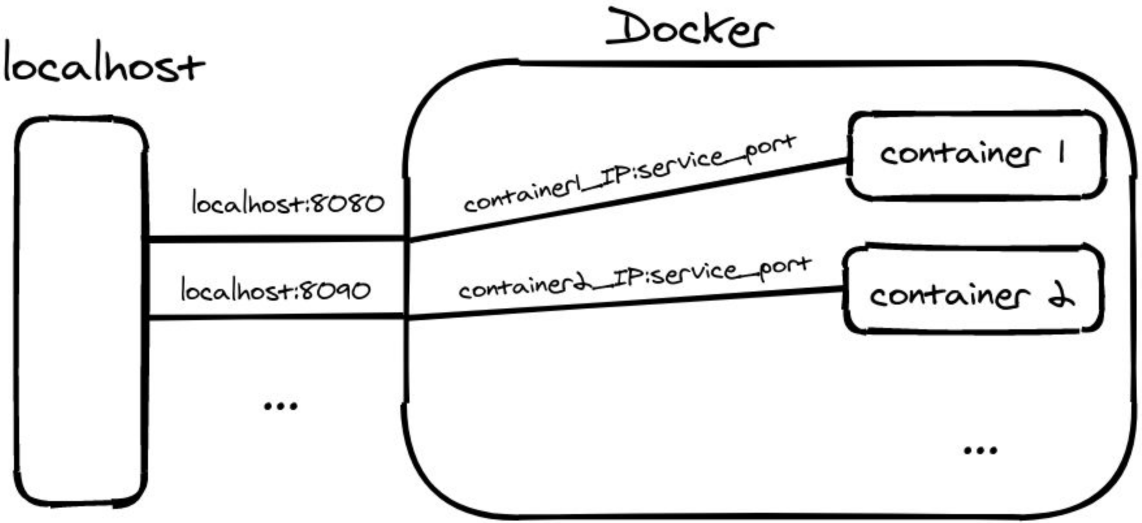
```
docker images
```



```
rafaelfontana@Rafaels-MacBook-Pro ~ % docker images
REPOSITORY    TAG       IMAGE ID      CREATED          SIZE
restbot       latest    9505e6c03c1b  5 seconds ago    159MB
```

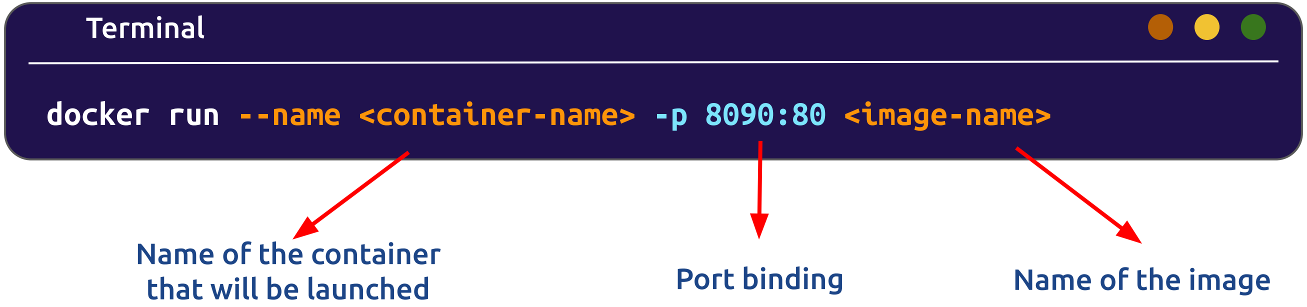
Binding ports 🚪 ↔ 🚪

Before launching our container, we need to clarify some concepts. When our containers will run, we would like to have a way to communicate with each of them for using the services they provide or even just for debugging. In order to do that we will write the `docker run` command we will need to specify the so-called port binding with the parameter `-p <host port>:<container port>`



Launch our first Container

To launch a container, we just need to execute the following command:



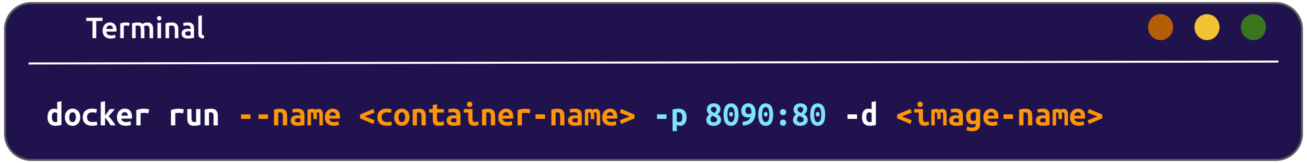
```
docker run --name <container-name> -p 8090:80 <image-name>
```

Name of the container
that will be launched

Port binding

Name of the image

Now, you should see the container starting and running in your terminal. If you want to run the container in background, you can add the parameter `-d` :



```
docker run --name <container-name> -p 8090:80 -d <image-name>
```

Upload your Image to Docker Hub

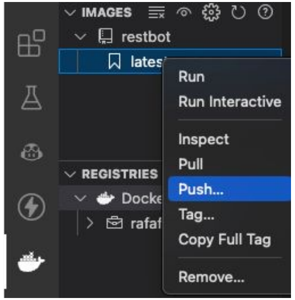
Until now, the image we built, is only available to us (local image). In case we work on a team, most probably we need to share our image with our colleagues. Docker has it’s own repository, **Docker Hub**. We just need to use the command `push` to upload our image to Docker Hub.

Terminal

```
docker push <your-docker-hub-repository>/<image-name>:<tag>
```

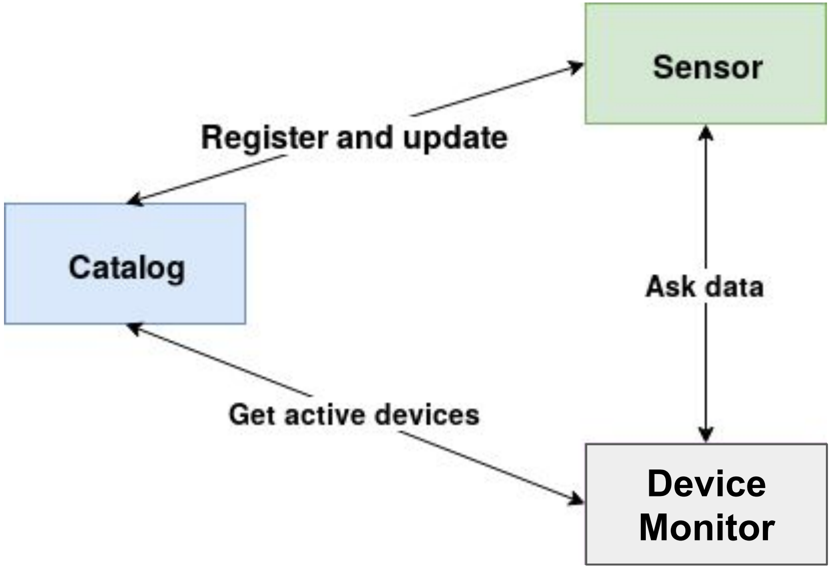
Tip

If you’re using VSCode, you can install the Docker extension, and simplify your work 😉



Example Simple Platform

Imagine a Platform with 3 actors: *Catalog*, *Sensor*, and *DeviceMonitor*. Each of them will run on its own container



Example Simple Platform: *Sensor*

The *Sensor* is a simple REST client for a temperature and humidity sensor. When the *sensor* is launched, it will send a **POST** request to the *Catalog* to register itself, stating which are its settings (IP address, port, accepted methods). Moreover, it will send a **PUT** request periodically (e.g. 1 minute) to the *Catalog* , to let the *Catalog* know that it is alive, and to keep it updated.

The settings of *Sensor* are stored in a `settings.json` file.

Example Simple Platform: *Catalog*

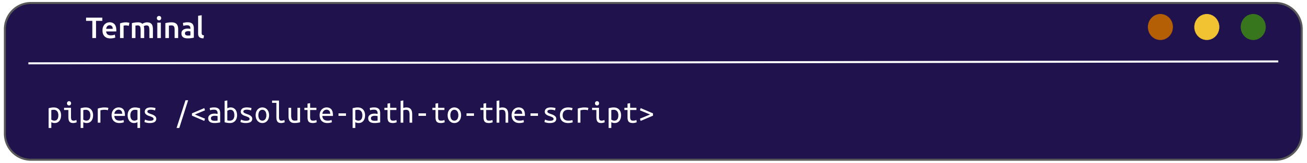
The *Catalog* is another REST client. It’s job is to keep and update the list of the available **devices** and **services** (with their settings). Moreover, it may provide this information to other entities that may need them. For example, the *DeviceMonitor* will retrieve the information from the Devices. Everytime the *Catalog* receives a request from a *Sensor* it will add it to the list of the **devices** and will store the timestamp of that request. This list is periodically controlled by the *DeviceMonitor* to check if the last timestamp of each of this devices respects a threshold, if the timestamp is too ”old” the device will be removed from the list. The settings are stored in a file called `settings.json`.

Example Simple Platform: *DeviceMonitor*

The service ***DeviceMonitor*** it’s a simple script to monitor the status of the ***Devices*** . It’s responsible for managing the status of the devices registered in the ***Catalog***.

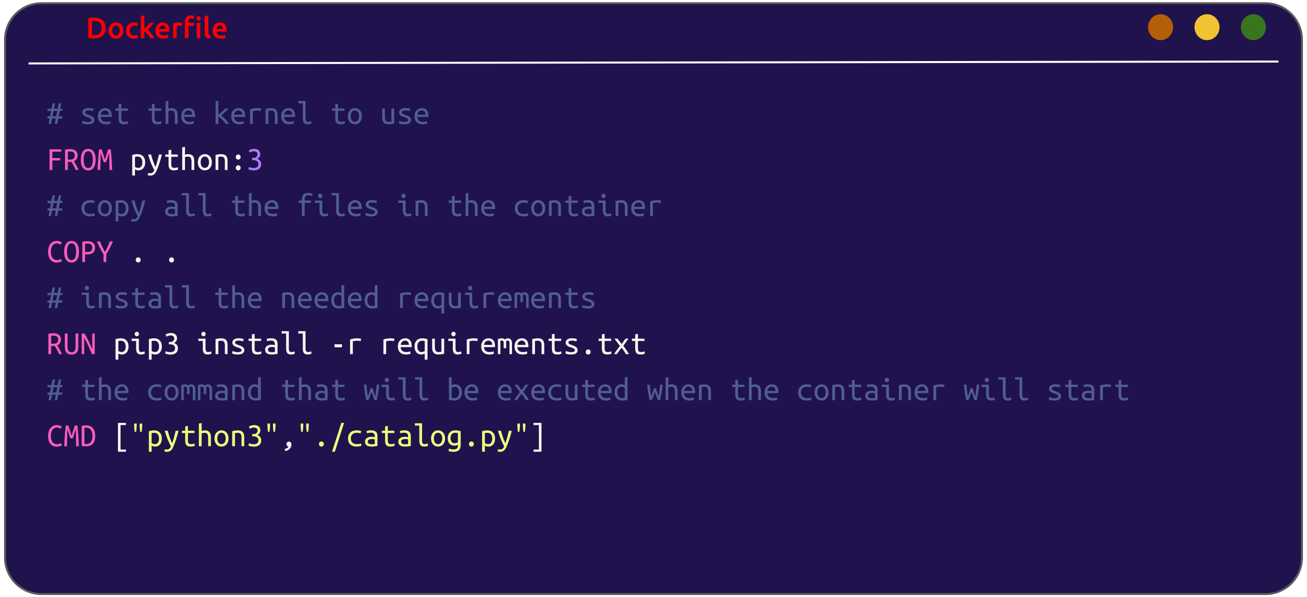
Example Simple Platform: *Requirements.txt*

In order to create the container for each of the actors, we need to create a **Dockerfile** for each of them. Remember, that before defining the **Dockerfile**, we need to create the `requirements.txt` file for each of the actors.



```
pipreqs /<absolute-path-to-the-script>
```

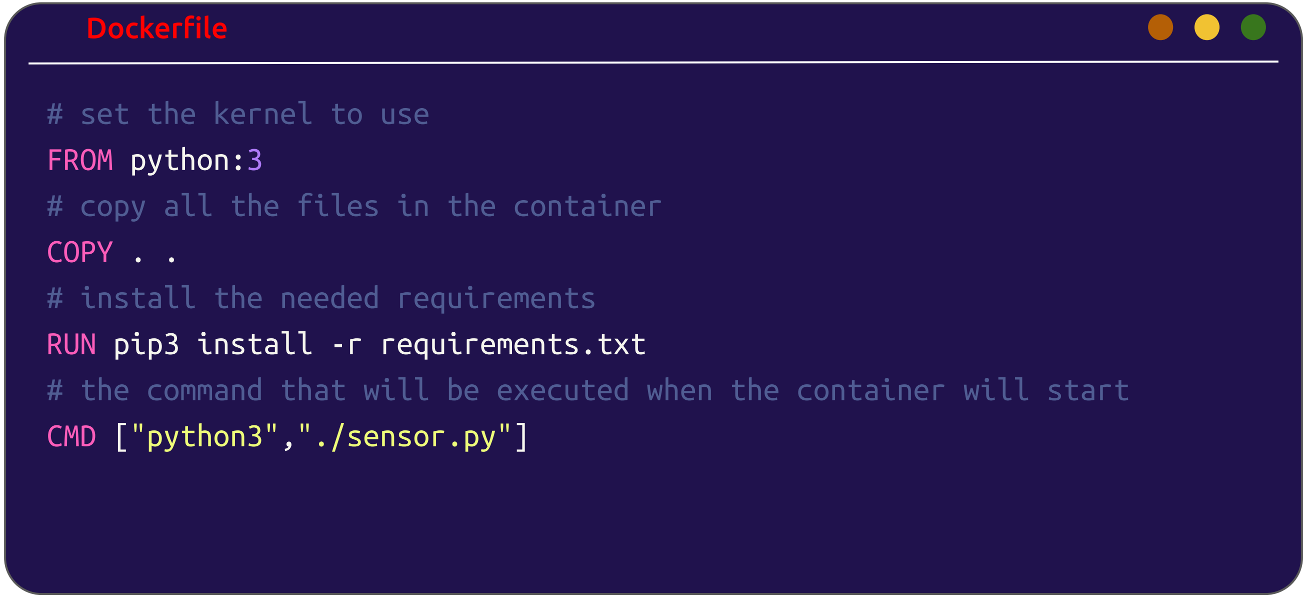
Example Simple Platform: *Catalog Dockerfile*



Dockerfile

```dockerfile
# set the kernel to use
FROM python:3
# copy all the files in the container
COPY . .
# install the needed requirements
RUN pip3 install -r requirements.txt
# the command that will be executed when the container will start
CMD ["python3","./catalog.py"]
```

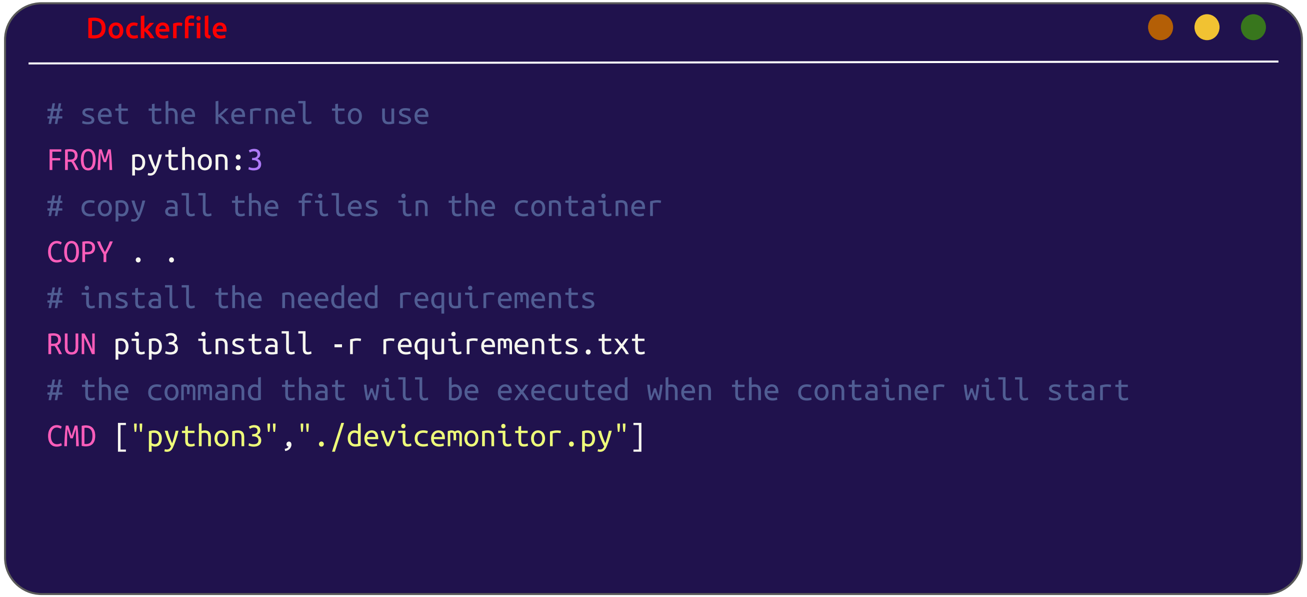
Example Simple Platform: *Sensor Dockerfile*



Dockerfile

```
# set the kernel to use
FROM python:3
# copy all the files in the container
COPY . .
# install the needed requirements
RUN pip3 install -r requirements.txt
# the command that will be executed when the container will start
CMD ["python3","./sensor.py"]
```

Example Simple Platform: *Monitor Dockerfile*



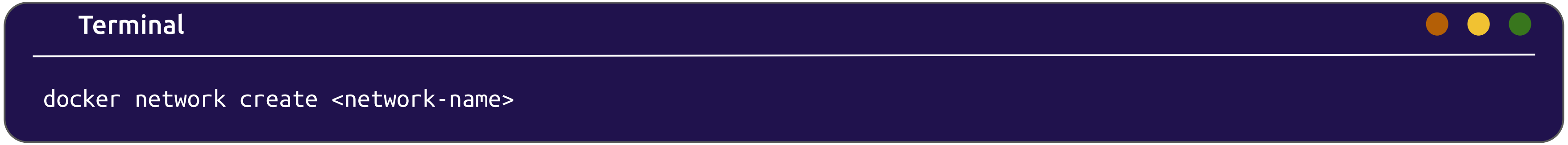
Dockerfile

```
# set the kernel to use
FROM python:3
# copy all the files in the container
COPY . .
# install the needed requirements
RUN pip3 install -r requirements.txt
# the command that will be executed when the container will start
CMD ["python3","./devicemonitor.py"]
```

Example Simple Platform

After creating each **Dockerfile**, we need to build the **Images** of each actor. In our case, we need to create 3 containers (one per actor), so that they can communicate between them and to “the world”. We first need to launch the ***Catalog***, and later ***Sensor*** and ***Monitor***.

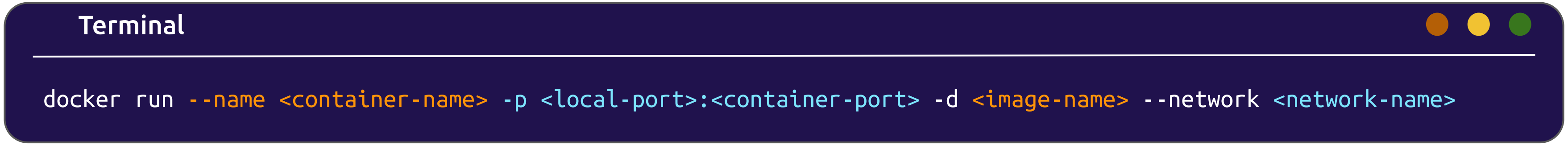
Before we run our containers, we will define a `docker network`, that will be used by the containers to communicate between them.



Terminal

```
docker network create <network-name>
```

After that, we can run launch container:

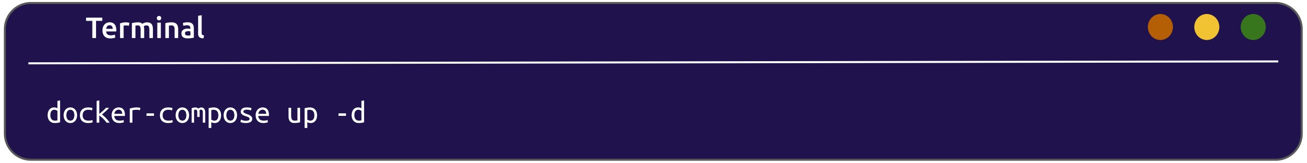


Terminal

```
docker run --name <container-name> -p <local-port>:<container-port> -d <image-name> --network <network-name>
```

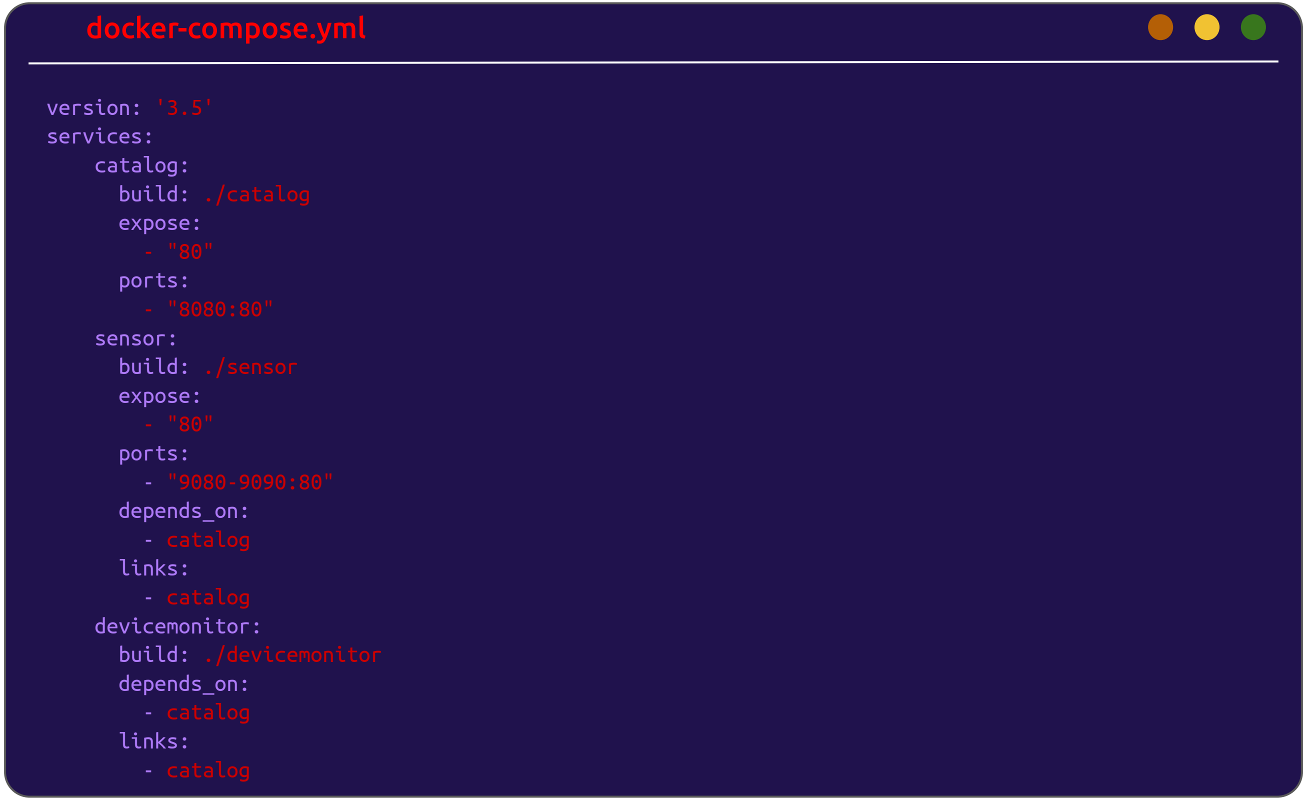
Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. With Docker Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration:



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

Docker Compose

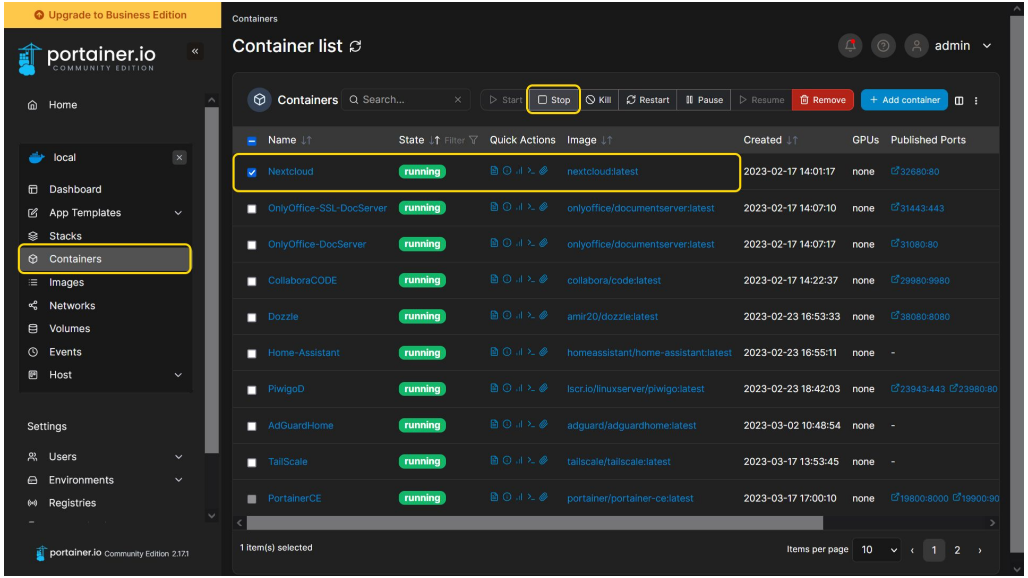


docker-compose.yml

```yaml
version: '3.5'
services:
    catalog:
      build: ./catalog
      expose:
        - "80"
      ports:
        - "8080:80"
    sensor:
      build: ./sensor
      expose:
        - "80"
      ports:
        - "9080-9090:80"
      depends_on:
        - catalog
      links:
        - catalog
    devicemonitor:
      build: ./devicemonitor
      depends_on:
        - catalog
      links:
        - catalog
```

Portainer IO

Portainer IO is a simple and lightweight management UI that allows users to easily manage Docker environments. It is a simple container that can run in Docker and allows you to manage Docker Images, Containers, Volumes, Networks, etc.



	Name	State	Quick Actions	Image	Created	GPUs	Published Ports
☑	Nextcloud	running		nextcloud:latest	2023-02-17 14:01:17	none	32680:80
□	OnlyOffice-SSL-DocServer	running		onlyoffice/documentserver:latest	2023-02-17 14:07:10	none	31443:443
□	OnlyOffice-DocServer	running		onlyoffice/documentserver:latest	2023-02-17 14:07:17	none	31080:80
□	CollaboraCODE	running		collabora/code:latest	2023-02-17 14:22:37	none	29980:9980
□	Dozzle	running		amir20/dozzle:latest	2023-02-23 16:53:33	none	38080:8080
□	Home-Assistant	running		homeassistant/home-assistant:latest	2023-02-23 16:55:11	none	-
□	PiwigoD	running		lscr.io/linuxserver/piwigo:latest	2023-02-23 18:42:03	none	23943:443 23980:80
□	AdGuardHome	running		adguard/adguardhome:latest	2023-03-02 10:48:54	none	-
□	TailScale	running		tailscale/tailscale:latest	2023-03-17 13:53:45	none	-
□	PortainerCE	running		portainer/portainer-ce:latest	2023-03-17 17:00:10	none	19800:8000 19900:90

1 item(s) selected

Items per page 10 ‹ 1 2 ›

Exercise

Let’s imagine you are an Administrator of the Simple platform that we have seen before. We will create a TelegramBot to receive an **alert** every time a Device is not working or disconnected. This alert will be sent by the DeviceMonitor every time a Device has been removed (due to inactivity). The Bot must register itself as a service to the Catalog.

Once you have developed the Bot, Dockerized it.