Docker

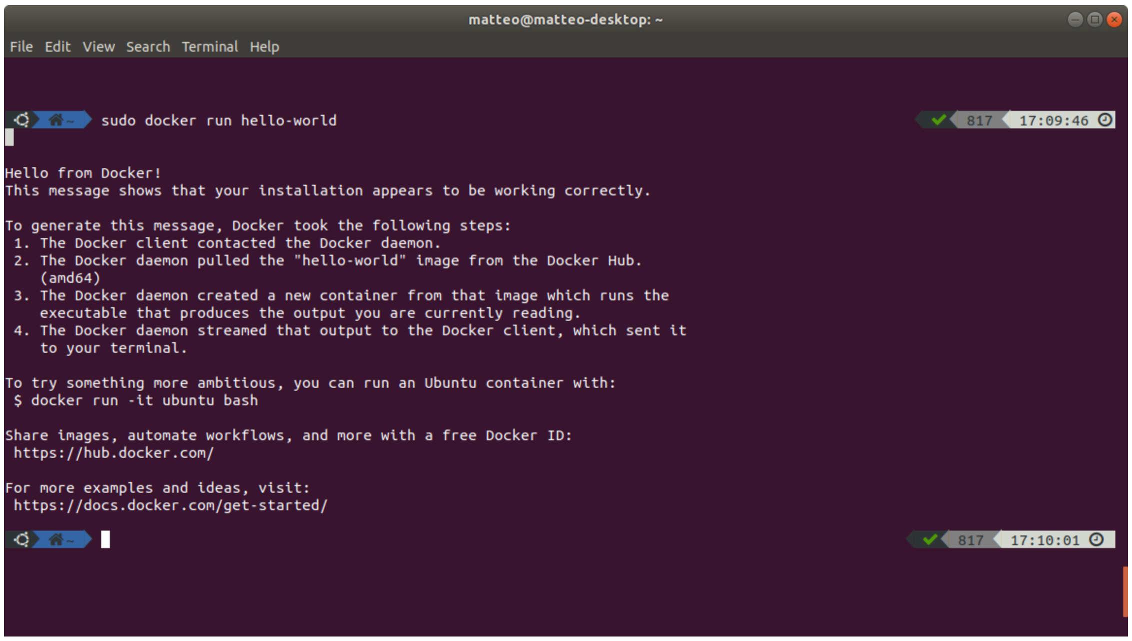
December 17, 2019

1 Get started with Docker

1.1 Installation

To install docker you can simply follow the instruction you can find at this website https://docs.docker.com/v17.09/engine/installation/

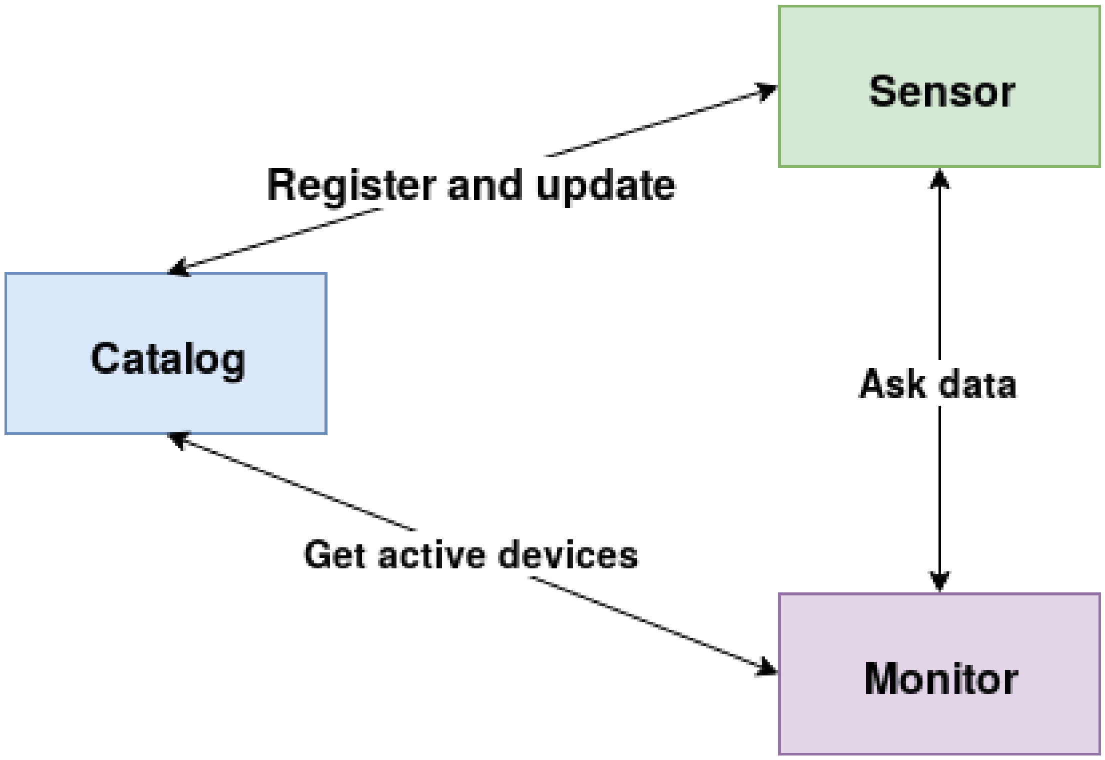
When you finished you can check that everything works correctly by opening a terminal and typing ”docker run hello-world” (you may need to add sudo for macOS and Linux), then you should see something like this



```
sudo docker run hello-world

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/
```

If you get that you’re ready to proceed with the example below

2 How to use Docker container



We want to create the code for the 3 different actors above and we want assign *Catalog* and *Sensor* to two different container, while instead *Monitor* will run in the "usual" way. Before creating the container let's see how the different actors behave in order to have a better understanding.

2.1 Sensor

The script *sensor.py* is a simple REST client for a temperature and umidity sensor. When the script is launched a put request will be sent to the catalog in order to the *Catalog* to notify that the sensor is alive and which are his settings (IP address, port, accepted methods). This notification will happen periodically to ensure that the *Catalog* stays updated. The settings are stored in a file called *settings.json*. The code is written below

[ ]:
```python
import cherrypy
import requests
import json
import random
class SensorREST(object):
    exposed=True
    def __init__(self):
        self.settings=json.load(open('settings.json'))
        self.settings['commands']=['hum','temp']
        requests.put(self.settings['catalog'],data=json.dumps(self.settings))
    def GET(self,*uri,**params):
        if len(uri)!=0:
            if uri[0]=='hum':
                value=random.randint(60,80)
            if uri[0]=='temp':
                value=random.randint(10,25)
            output={'deviceID':self.settings['ID'],str(uri[0]):value}
            return json.dumps(output)
        else:
            return json.dumps(self.settings)
    def pingCatalog(self):
        requests.put(self.settings['catalog'],data=json.dumps(self.settings))

if __name__ == '__main__':
    conf={
        '/':{
                'request.dispatch':cherrypy.dispatch.MethodDispatcher(),
                'tool.session.on':True
        }
    }
    s=SensorREST()
    cherrypy.config.update({'server.socket_host': '0.0.0.0','server.
 ↪socket_port': s.settings['port']})
    cherrypy.tree.mount(s,'/',conf)
    cherrypy.engine.start()
    while True:
        print('sleeping')
        s.pingCatalog()
        time.sleep(10)
    cherrypy.engine.exit()
```

2.2 Catalog

The script *catalog.py* is another REST client, his job is to keep and updated list of the available devices and their setting and to trovide it to toher actor that may need it (in our case *Monitor*). 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 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*. The code for the *Catalog* is below

[ ]:
```
import cherrypy
import requests
import json
import time
class Catalog(object):
    def __init__(self):
        self.devices=[]
        self.actualTime=time.time()
    def addDevice(self,devicesInfo):
        self.devices.append(devicesInfo)
    def updateDevice(self,deviceID,devicesInfo):
        for i in range(len(self.devices)):
            device=self.devices[i]
            if device['ID']==deviceID:
                self.devices[i]=devicesInfo
    def removeDevices(self,deviceID):
        for i in range(len(self.devices)):
            device=self.devices[i]
            if device['ID']==deviceID:
                self.devices.pop(i)
    def removeInactive(self):
        self.actualTime=time.time()
        for device in self.devices:
            if self.actualTime-device['last_update']>10:
                self.devices.remove(device)

class CatalogREST(object):
    exposed=True
    def __init__(self,clientID):
        self.ID=clientID
        self.catalog=Catalog()
    def GET(self,*uri,**params):
        output={'devices':self.catalog.devices,"updated":self.catalog.
↪actualTime}
        return json.dumps(output)
    def PUT(self):
        body=cherrypy.request.body.read()
        json_body=json.loads(body.decode('utf-8'))
        if not any(d['ID']==json_body['ID'] for d in self.catalog.devices):
            last_update=time.time()
            json_body['last_update']=last_update
            self.catalog.addDevice(json_body)
            output=f"Device with ID {json_body['ID']} has been added"
            print (output)
            return output
        else:
            last_update=time.time()
            json_body['last_update']=last_update
            self.catalog.updateDevice(json_body['ID'],json_body)
            return json_body

    def DELETE(self,*uri):
        self.catalog.removeDevices(uri[0])
        output=f"Device with ID {uri[0]} has been removed"
        print (output)


if __name__ == '__main__':
    catalogClient=CatalogREST('Catalog')
    conf={
        '/':{
                'request.dispatch':cherrypy.dispatch.MethodDispatcher(),
                'tool.session.on':True
        }
    }
    cherrypy.config.update({'server.socket_host': '0.0.0.0','server.
 ↪socket_port': 80})
    cherrypy.tree.mount(catalogClient,'/',conf)
    cherrypy.engine.start()
    while True:
        catalogClient.catalog.removeInactive()
        time.sleep(2)
    cherrypy.engine.exit()
```

2.3 Monitor

The script *monitor.py* it's a simple script to monitor retrieve the value of the available devices. It's able to get the list of available devices from the *Catalog* and it will show the data coming from the chosen devices. The code is below

[ ]:
```
import requests
import json
import time
class Viewer(object):
    """docstring for Viewer"""
    def __init__(self):
        self.catalogInfo=json.load(open("settings.json"))
        self.devices=self.getDevices()

    def getDevices(self):
        response=requests.get(self.catalogInfo['catalogURL']).json()
        print('List of available devices obtained')
        return  response['devices']

    def listDevices(self):
        print('This are the available devices:')
        for device in self.devices:
            print(device['ID'])
        idSelected=input("Which device to you want to monitor (r to refresh  q␣
↪to quit)\n")
        if idSelected!='q':
            if idSelected=='r':
                self.getDevices()
                self.listDevices()
            else:
                idSelected=int(idSelected)
                self.monitor([x for x in self.devices if␣
↪x['ID']==idSelected][0])
        else:
            exit()

    def monitor(self,device):
        print("Temp (C)\tHum(%)")
        end_time=time.time()+4
        while time.time()<end_time:
            temp=requests.get(device["IP"]+':'+str(device['port'])+'/temp').
↪json()['temp']
            hum=requests.get(device["IP"]+':'+str(device['port'])+'/hum').
↪json()['hum']
            print(str(temp)+'\t\t'+str(hum),end="\r")
            time.sleep(0.5)
        self.listDevices()

if __name__ == '__main__':
    v=Viewer()
    v.listDevices()
```

3 Dockerfile creation

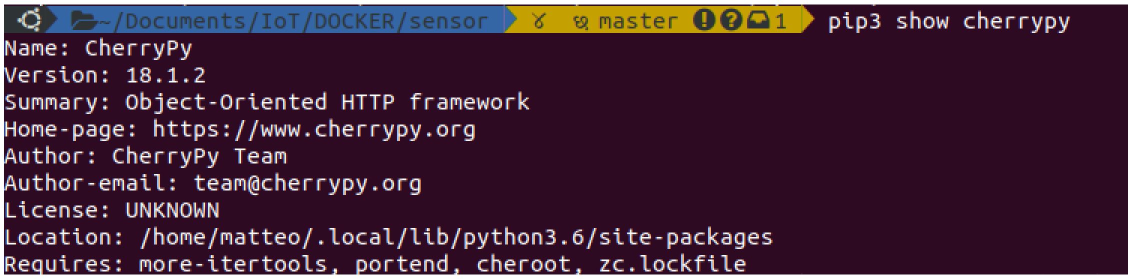
In order to create a container for *Catalog* and *Monitor* we need two define two *Dockerfile* that specify their settings: the file to use, the library to install, the command to execute etc. As we did before with the code we will analyze each actor separately

3.1 Requirements definition

The first thing to do before creating the dockerfile is to create a file calle *requirements.txt* containing all the library that we need for our script. To do this there are 2 main ways:

First method

For each of the library we import in our script we must run in the terminal the following command *"pip3 show "* (in some cases you can use pip instead of pip3) and we should see something like this



```
~/Documents/IoT/DOCKER/sensor  master 1  pip3 show cherrypy
Name: CherryPy
Version: 18.1.2
Summary: Object-Oriented HTTP framework
Home-page: https://www.cherrypy.org
Author: CherryPy Team
Author-email: team@cherrypy.org
License: UNKNOWN
Location: /home/matteo/.local/lib/python3.6/site-packages
Requires: more-itertools, portend, cheroot, zc.lockfile
```

form this result we need the *Name* and the *Version*, which we will write in the file *requirements.txt*, in this case it would be

```
CherryPy==18.1.2
```

We must do this operation for all the library that are not "standard" adding a new line in the file for each of them (for example we can avoid it for the library *json* and *time*)

Second method

The second method is the easiest to use but it require to install pipreqs by running on the terminal the comand *"pip3 install pipreqs"* (also in this case sometimes pip is enough). Once the installation is finished you can create the file *requirements.txt* buy running the following command *"pipreqs "*

3.2 Sensor Dockerfile

So the first thing to do is creating *requirements.txt*. Once we've done that we can can create a new file called *Dockerfile* with no extension (no .txt no .py etc). The first thing to write in the file is which language we're going to use, in our case it's Python 3 so we're going to write

```
FROM python:3
```

The next thing to do is to *ADD* each file we're going to use specifing the path of origin and the destination. In the case of the sensor the file used are *sensor.py*, *settings.json* and *requirements.txt* so we will write

```
#ADD <origin> <destination>
ADD sensor.py /
ADD settings.json /
ADD requirements.txt /
```

After that we want to install all the library specified in the file *requirements.txt*. To do this we write

```
RUN pip3 install -r requirements.txt
```

The last thing we need to do is specify which comand to execute to start our script, so in this case

```
#CMD <comma-separated list of command>
CMD ["python3","./sensor.py"]
```

So at the end our dockerfile for *Sensor* will look like this

```
FROM python:3
```

```
ADD sensor.py /
ADD settings.json /
ADD requirements.txt /
RUN pip3 install -r requirements.txt
CMD ["python3","./sensor.py"]
```

3.3 Catalog Dockerfile

The process is similar to the one explained above, at the end we should have something like

```
FROM python:3
ADD catalog.py /
ADD requirements.txt /
RUN pip3 install -r requirements.txt
CMD ["python3","./catalog.py"]
```

4 Build and run and image inside a container

4.1 Build

To build the image of the docker we need to:

- Open the terminal
- go in the folder where the Dockerfile is located
- type "docker build . -t "

After doing that docker will start to download the needed version of the python and the libraries we specified in the file *requirements.txt*.
For our case we need to do that bot for the catalog and the sensor.

4.2 Run

To run a container we simply need to type on the terminal this

```
docker run  --name <NameOfContainer> <imageTag>
```

If you want to run the container in background you can add the option -d before the image tag.

In most of the cases we want containers to communicate among each other and we want also external application to communicate with these, in order to achievethis we must ***publish*** the port of the containers. This can be done in different ways but we will just see the simplest one

In our case we need to run 2 container *catalog* and *sensor* so we want this two to be able to communicate among them and to the world. To do this we need to launch *catalog* first and then *sensor*. So before starting the containers we need to create a docker network

```
docker network create <networkName>
```

Once we've done that we can launch each image like this:

```
docker run  --name <NameOfContainer>  -p <localhostPort>:<dockerPort> --network <networkName>

Example:
docker run --name sensor -p 9090:80 --network myNetwork sensor
```

After doing this for both of the images we can launch monitor.py or use the browser to get the information of the catalog or the measurements of the sensor

[ ]: