ESERCIZIO 1:

Obiettivo: realizzare un *Catalog*, che metta insieme le funzionalità di un *Service Catalog* e un *Resource Catalog*, ovvero un elenco di servizi web dedicati ad un mondo hardware, con l'obiettivo di registrare e di fornire, se richiesti, i dispositivi IoT disponibili. Il nostro *Catalog* però, oltre a riferirsi solo ad un mondo hardware, si collega anche al mondo software con l'obiettivo di fornire anche eventuali servizi disponibili.

Il web server è stato sviluppato sfruttando i principali metodi HTTP e le funzionalità da esso implementate sono molteplici:

```
def GET(self, *uri, **params):
    if uri[0] == "broker":
        return self.broker + self.port

    if uri[0] == "devices": #punto3
        return json.dumps(self.devices)

    elif uri[0] == "deviceID": #punto4
        item_return =""
        for item in self.devices:
            if item["deviceID"] == int(uri[1]):
                item_return = item
        if item_return == "":
            raise cherrypy.HTTPError(400, "deviceID not found")
        return json.dumps(item_return)
```

1. è possibile ricavare informazioni sull'indirizzo ip e sulla porta del message broker tramite una chiamata GET al server con la stringa "broker" come primo parametro del path;
2. permette di restituire al client, che ne fa richiesta, tutti i dispositivi memorizzati nel catalog in un file formato JSON (tramite la funzione *json.dumps()*). A differenza del caso precedente, bisogna fornire come primo parametro del path la stringa "devices";
3. terza funzionalità implementata è quella di fornire i parametri di un dispositivo tramite il proprio deviceID. Semplicemente si cerca all'interno della lista di dizionari self.devices il device con l'ID specificato nell'uri e, se trovato, viene ritornato in un file formato JSON, altrimenti viene lanciato un messaggio di errore;
4. queste identiche funzionalità sono implementate sia per la ricerca di utenti registrati al catalog sia per la ricerca di servizi che il catalog stesso offre;

```
def PUT(self,*uri,**params):
    if uri[0] == "addDevice": #punto2
        self.json = json.loads(cherrypy.request.body.read())

        new_device = {
            "deviceID": int(self.json["deviceID"]),
            "end-points": "http://localhost:8080/addDevice",
            "resources": self.json["resources"],
            "timestamp": time.time()
        }

        for item in self.devices: #per esercizio 3
            if item["deviceID"] == int(new_device["deviceID"]):
                self.devices.remove(item)

        self.devices.append(new_device)
```

5. naturalmente il catalog per poter fornire delle informazioni, deve essere anche capace di riceverle e memorizzarle. Infatti, permette di aggiungere uno o più dispositivi, identificandoli tramite un deviceID univoco, fornendo i propri end-points e le funzionalità fornite e inserendo anche l'istante in cui vengono aggiunti.
Si suppone che le informazioni sul nuovo dispositivo da inserire siamo passate al servizio REST come file JSON, che può essere convertito in un dizionario (*json.loads()*) con le informazioni sul deviceID e le resources fornite dal device, cioè una indicazione delle funzionalità fornite dal dispositivo. Inoltre, dal momento che i device, così come i servizi, possono essere eliminati (prossima funzionalità), è importante fornire anche la possibilità di aggiornare il *timestamp* dei device o dei services. Quindi, all'interno del ciclo for si cerca l'eventuale nuovo dispositivo che si vuole inserire e se già presente si elimina il precedente e si inserisce il nuovo, il quale presenterà un timestamp aggiornato all'istante in cui viene inserito.

Viene riportato solo il codice per gestire i device, ma analoghe procedure vengono svolte per gli users e services.

```
class Thread(threading.Thread):
    def __init__(self, threadID, devices, services):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.devices = devices
        self.services = services
    def run(self):
        while True:
            for item in self.devices:
                if time.time() - item["timestamp"] >= 120:
                    self.devices.remove(item)

            for item in self.services:
                if time.time() - item["timestamp"] >= 120:
                    self.services.remove(item)

            time.sleep(60)
```

Come detto precedentemente una funzionalità importante consiste nell'eliminazione di eventuali devices o services che sono registrati da oltre 2 minuti. Questa funzionalità è realizzata tramite un thread che, in maniera indipendente dal resto del codice (per definizione), ogni minuto controlla se sono presenti eventuali dispositivi o servizi che non vengono aggiornati da due minuti e, in caso affermativo, provvede all'eliminazione degli stessi.

ESERCIZIO 2:

```
    while True:
        user_input = input("""Retrieve information about:
m: message broker
rd: all the registered devices
d: device with a specific deviceID
ru: all users
u: user with a specific userID
rs: all the register services
s: service with a specific serviceID\n""")

        if user_input =="m":
            url = "http://localhost:8080/broker"
            response = requests.get(url)
            print(response.json())

        elif user_input == "rd":
            url = "http://localhost:8080/devices"
            response = requests.get(url)
            print(response.json())


        elif user_input == "d":
           deviceID = input("Inserisci deviceID: ")
           response = requests.get(f"http://localhost:8080/deviceID/{deviceID}")
           print(response.json())
```

Si vuole semplicemente sviluppare un client per invocare i metodi sviluppati nell'esercizio precedente, in modo da verificarne il corretto funzionamento.

Allora nel main del nostro programma presentiamo all'utente le funzionalità fornite dal Catalog, facendo scegliere quella desiderata. L'utente deciderà che informazione acquisire e il nostro programma, attraverso la libreria *requests* di Python, inoltrerà delle richieste GET al Catalog, fornendo nel path l'input corrispondente al servizio scelto, secondo i formati definiti nell'esercizio precedente.

ESERCIZIO 3:

```
while True:
    print("Add Device")
    deviceID = input("Device ID: ")
    resources = input("Resources: ")
    payload = {
        "deviceID": deviceID,
        "resources": resources
        }
    requests.put("http:localhost:8080/addDevice", payload)
    time.sleep(60)
```

Vogliamo sviluppare un client per emulare un device IoT che invochi i servizi offerti dal Catalog. Si sceglie di far inserire all'utente i parametri che caratterizzano il device, in conformità al formato scelto nell'implementazione del Catalog. Questi vengono memorizzati nel dizionario payload, il quale viene poi inviato come file JSON con una richiesta PUT. Il nostro Catalog ha già tutte le funzionalità per aggiungere il nuovo device e aggiornare il corrispondente "insert-timestamp".

ESERCIZIO 4:

```
#include <Bridge.h>
#include <BridgeServer.h>
#include <BridgeClient.h>
#include <ArduinoJson.h>
#include <Process.h> //comandi shell sulla CPU Linux


const int TEMP_PIN = A1;
const int LED_PIN = 12;
const float B = 4275;
const float R0 = 100000;
const float T0 = 298.15;
const int capacity = JSON_OBJECT_SIZE(2) + JSON_ARRAY_SIZE(1) + JSON_OBJECT_SIZE(4) + 40;
DynamicJsonDocument doc_snd(capacity);

void setup() {
  pinMode(LED_PIN, OUTPUT);
  pinMode(TEMP_PIN, INPUT);
  digitalWrite(LED_PIN, LOW);
  Serial.begin(9600);
  while(!Serial);
  Serial.println("Lab 2.4 SW starting");
  Bridge.begin();
  digitalWrite(LED_PIN,HIGH);
}
void loop() {

  int n = putRequest(jsonEncode());


  delay(60000);
}

int putRequest(String data){
  Process p;
  //Serial.println(data);
  p.begin("curl");
  p.addParameter("-H");
  p.addParameter("Content-Type: application/json");
  p.addParameter("-X");
  p.addParameter("PUT");
  p.addParameter("-d");
  p.addParameter(data);
  p.addParameter("http://192.168.1.24:8081/addDevice");
  p.run();
  return p.exitValue();
}

float readTemp(){
  int sig = analogRead(TEMP_PIN);
  float R = ((1023.0/(float)sig) - 1.0)*R0;
  float log_sig = log(R/R0);
  float T = 1/((log_sig/B) + (1/298.15));
  float temp = T - 273.15;
  return temp;
}


String jsonEncode(){
  doc_snd.clear();
  doc_snd["deviceID"] = 123;
  doc_snd["resources"] = "Temperature";

  String output;
  serializeJson(doc_snd, output);
  return output;
}
```

Il testo chiede di estendere le funzionalità del servizio web implementato nell'esercizio 3.2, Lab Hardware – part 3, in cui la Yùn Arduino è usata per rilevare i valori di temperatura dell'ambiente circostante e mandarli con una richiesta POST ad un certo server web. In questo caso Arduino sarà uno dei dispositivi IoT che si iscrivono al Catalog, cioè che fanno parte del nostro ambiente IoT.
Quello che vogliamo fare è quindi testare il corretto funzionamento del Catalog, permettendo a una Yùn di inviare fisicamente le proprie informazioni identificative per entrare a far parte del nostro environment.

Per farlo, semplicemente realizziamo uno sketch Arduino in cui inviamo una richiesta PUT al Catalog, con *addDevice* come primo e unico parametro del path dell'URL e con i dati della Yùn come documento JSON. Questi conterranno, conformemente al formato che il Catalog si aspetta di ricevere, le informazioni sul deviceID e sulla risorsa da lui fornita, in questo caso la temperatura. Attraverso le funzioni della libreria ArduinoJson, realizziamo il documento, che invieremo come body nella richiesta PUT, proprio come il Catalog si aspetta. Questa sottoscrizione al Catalog viene ripetuta ogni minuto, in modo da non permettergli di cancellare le sue informazioni sul device (fa parte delle sue funzionalità cancellare periodicamente elementi dalla lista di devices la cui sottoscrizione eccede un certo intervallo di tempo).

Le informazioni sulla temperatura, ovviamente, non saranno inviate al Catalog, lui registra solo la presenza di dispositivi, utenti e servizi; come reminder, abbiamo riportato le procedure con cui Arduino, usando il sensore, rileva la temperatura.

ESERCIZIO 5:

```
class MySubscriber:
    def __init__(self, clientID,topic,broker):
        self.clientID = clientID
        self._paho_mqtt = PahoMQTT.Client(clientID, False)
        self._paho_mqtt.on_connect = self.myOnConnect
        self._paho_mqtt.on_message = self.myOnMessageReceived
        self.topic = topic
        self.messageBroker =broker

    def start (self):
        self._paho_mqtt.connect(self.messageBroker, 1883)
        self._paho_mqtt.loop_start()
        self._paho_mqtt.subscribe(self.topic, 2)

    def stop (self):
        self._paho_mqtt.unsubscribe(self.topic)
        self._paho_mqtt.loop_stop()
        self._paho_mqtt.disconnect()

    def myOnConnect (self, paho_mqtt, userdata, flags, rc):
        print ("Connected to %s with result code: %d" % (self.messageBroker, rc))

    def myOnMessageReceived (self, paho_mqtt , userdata, msg):
        print ("Topic:'" + msg.topic+"', QoS: '"+str(msg.qos)+"' Message: '"+str(msg.payload) + "'")
        payload = json.loads(msg.payload)



        new_device = {
                "deviceID": payload["deviceID"],
                "end-points": msg.topic,
                "resources": payload["resources"],
                "timestamp": time.time()
            }
        DEVICES.append(new_device)
```

In questo esercizio vogliamo estendere le funzionalità del Catalog per implementare anche il paradigma di comunicazione publish/subscribe; si sceglie di aggiungere la nuova feature lasciando inalterate quelle precedenti, cioè il Catalog potrà funzionare sia come server http sia come Subscriber MQTT.

Viene definita la classe MySubscriber con le caratteristiche e le procedure viste a lezione, che implementa un Subscriber MQTT grazie alle funzioni della classe paho.mqtt.client. La funzione significativa è la *myOnMessageReceived(),* cioè la funzione di callback richiamata dal sistema quando viene rilevato l'invio, da parte del Message Broker, di un'informazione utile, che riporterà lo stesso topic al quale mi sono iscritto; in questo caso il topic è tiot/6/device, passato alla classe MySubscriber nella sua definizione nel main. La *myOnMessageReceived()* si aspetta le informazioni sul nuovo device nella solita forma, come documento JSON con il deviceID e le sue risorse. Viene allora creato un nuovo dizionario in locale, che raccoglie le informazioni sul nuovo dispositivo, aggiungendo il topic come endpoint e il timestamp corrente, e le salva come dizionario in una lista di dizionari definita come variabile globale. È in questo modo che il Catalog tiene traccia dei dispositivi, utenti e servizi.

Come specificato nel testo, con MQTT si gestisce solo la sottoscrizione di nuovi dispositivi IoT, mentre i servizi e gli utenti vengono gestiti con il paradigna RESTful discusso negli esercizi precedenti (non riportato), che ora, a differenza di prima, salva le informazioni di cui deve tener traccia nelle variabili globali DEVICES, USERS e SERVICES.

ESERCIZIO 6

```
if __name__=="__main__":
#Standard configuration to serve the url "localhost:8080"

    conf={
        '/':{
                'request.dispatch':cherrypy.dispatch.MethodDispatcher(),
                'tool.session.on':True
                }
        }
    cherrypy.tree.mount(Lab2SW(),'/',conf)
    #cherrypy.config.update({"server.socket_host":'0.0.0.0'})
    cherrypy.config.update({'server.socket_port': 8080})
    test=MySubscriber("MySubscriber", "tiot/6/device", "test.mosquitto.org")
    test.start()
    cherrypy.engine.start()
    cherrypy.engine.block()
```

```
if __name__ == "__main__":
    test = MyPublisher("MyPublisher", "tiot/6/device")
    test.start()

    while True:
            deviceID = input("Device ID: ")
            resources = input("Resources: ")
            subscription = {
                    "deviceID": deviceID,
                    "resources": resources
            }
            print(subscription)
            test.myPublish(json.dumps(subscription))
            time.sleep(60)
```

Vogliamo, similmente agli esercizi precedenti, testare il corretto funzionamento della nuova modalità in cui abbiamo realizzato il Catalog, simulando un dispositivo IoT che agisce da Publisher, mandando le sue informazioni al Catalog per registrarsi e ripetendo l'operazione ogni minuto.

Definiamo il Publisher con la classe MyPublisher, con le stesse caratteristiche standard del codice discusso a lezione; il Message Broker a quale invierà le sue informazioni sarà definito a priori, e ovviamente deve essere lo stesso a cui fa riferimento il Catalog come Subscriber. Nel main definiamo il Publisher, dandogli il topic dei messaggi che dovrà pubblicare, lo stesso al quale è iscritto il Catalog. Chiediamo all'utente di inserire le informazioni del device, che saranno salvate in un dizionario nella solita modalità, convertite con *json.dumps()* in un documento Json e inviate al Broker tramite la funzione *myPublish()* del Publisher, che usa a sua volta la funzione *publish()* della libreria paho.mqtt.client (classe Client) per mandarla al Broker a cui è collegato. Lui invierà il documento al Catalog, il quale, ricevendolo, lo processerà come già sappiamo e registrerà l'informazione sul nuovo device.