ESERCIZIO 1

Obiettivo: realizzare uno sketch http sulla yun per rispondere a richieste GET provenienti dalla rete locale.

Componenti software utilizzati: Libreria Bridge.

Componenti hardware utilizzati: LED, sensore di temperatura, interfaccia seriale, interfaccia WiFi.

```
void process(BridgeClient client){
  String command = client.readStringUntil('/');
  command.trim();

  if (command == "led"){
    int val = client.parseInt();
    if (val == 0 || val == 1){
      digitalWrite(LED_PIN, val);
      printResponse(client, 200, senMLEncode(F("led"), val, F(""))); //F è una macro

    }
    else{
      printResponse(client,404, "");
    }
  }
  else if (command == "temperature") {
    printResponse(client, 200, senMLEncode(F("Temperature"), readTemp(), F("")));
  }
  else {
    printResponse(client,400, "");
  }
}
```

Sketch: nella funzione *setup()*, oltre ai comandi per settare i pin, abbiamo configurato un http server sulla yun. Questo server risponde a tutti i client che inviano delle richieste e, per ogni client, viene avviato un processo tramite la funzione *process()*. L'obiettivo della funzione *process()* è quello di "scomporre" l'URL in due elementi e, sulla base di questi elementi, settare lo stato del led oppure leggere il valore corrente di temperatura. All'interno di essa grande rilevanza assume la funzione *printResponse()*, che permette di stampare il body della risposta http. Inoltre, quest'ultimo viene codificato come file con formato SenML JSON tramite la funzione *senMlEncode()*, nella quale abbiamo creato manualmente un file "stile JSON" che poi, attraverso la funzione *serializeJson()*, viene trasformato in stringa JSON e usato nel body della risposta http.

ESERCIZIO 2

Obiettivo: realizzare uno sketch per effettuare richieste POST dalla Yun verso un server riportando nel body i dati ricavati dal sensore di temperatura.

Componenti software utilizzati: libreria Process.

Componenti hardware utilizzati: sensore di temperatura.

```
void loop() {
  int n = postRequest(senMLEncode("temperature", readTemp(), "Cel"));

 if (n == 200){
    Serial.println("Tutto ok: " + String(n));
  }
  else{
    Serial.println("Errore: " + String(n));
  }

  delay(1000);
}

int postRequest(String data){
  Process p;
  Serial.println(data);
  p.begin("curl");
  p.addParameter("-H");
  p.addParameter("Content-Type: application/json");
  p.addParameter("-X");
  p.addParameter("POST");
  p.addParameter("-d");
  p.addParameter(data);
  p.addParameter("http://192.168.1.24:8080/log");
  p.run();

  return p.exitValue();
}
```

```
 def GET(self, *uri, **params):
     j = self.dict_output #dict

     for item in j:
         if item["bn"] != "Yun":
             raise cherrypy.HTTPError(400, "Campo bn errato")
         elif item["e"][0]["n"] != "temperature":
             raise cherrypy.HTTPError(400, "Campo temperatura errato")

     return json.dumps(j)


 def POST(self,*uri,**params):
     if uri[0] == "log":
         self.json = json.loads(cherrypy.request.body.read())
         self.dict_output.append(self.json)
```

Sketch: una delle funzioni principali del codice è la *postRequest()*. Questa funzione permette di far funzionare la Yun come un "client" che manda ogni secondo delle informazioni al server Cherrypy sul valore di temperatura misurato. Questa informazione viene inviata nel body della richiesta, il quale, come nell'esercizio precedente, viene serializzato in un file JSON all'interno della funzione *senMLEncode()*. Inoltre, la *postRequest()* dichiara al suo interno un Process p utile per lanciare comandi Linux da Arduino. Noi useremo questo per inviare richieste POST tramite un tool Linux chiamato curl.

Server Cherrypy:

Lato server vengono gestite tutte le richieste GET e POST. Le prime ritornano l'intera lista formattata in formato JSON; mentre le seconde permettono di memorizzare il contenuto del body nella lista dict_output.

ESERCIZIO 3:

Obiettivo: realizzare uno sketch che consenta alla Yùn di comunicare via MQTT, agendo sia come publisher, pubblicando i dati di temperatura rilevati ogni 10 secondi, che come subscriber, ricevendo dati per controllare il led sulla breadboard.

Componenti hardware utilizzate: breadboard, sensore di temperatura, led rosso.

Componenti software utilizzate: libreria MQTTClient, libreria Bridge

Sketch:

```
const int capacity = JSON_OBJECT_SIZE(2) + JSON_ARRAY_SIZE(1) + JSON_OBJECT_SIZE(4) + 40;
DynamicJsonDocument doc_snd(capacity);
DynamicJsonDocument doc_rec(capacity);

 void setup() {
   pinMode(LED_PIN, OUTPUT);
   pinMode(TEMP_PIN, INPUT);
   digitalWrite(LED_PIN, LOW);
   Serial.begin(9600);
   while(!Serial);
   Serial.println("Lab 3.2 Starting");
   Bridge.begin();
   digitalWrite(LED_PIN,HIGH);
   mqtt.begin("test.mosquitto.org", 1883);
   mqtt.subscribe("/tiot/0/led", setLedValue);

 }

 void loop() {
   mqtt.monitor();

   //Temperatura
   float temp = readTemp();
   String message = senMLEncode("temperature", temp, "Cel");
   mqtt.publish("/tiot/0/temperature", message);
   Serial.println(temp);
   delay(1000);

void setLedValue(const String& topic, const String& subtopic, const String& message){

  DeserializationError err = deserializeJson(doc_rec, message);
  if(err){
    Serial.println(F("DeserializedJson() failed with code "));
    Serial.println(err.c_str());
  }
  //controllare formato doc_rec

  if(doc_rec["e"][0]["n"] == "led"){
    if(doc_rec["e"][0]["v"] == 1 || doc_rec["e"][0]["v"] == 0){
      digitalWrite(LED_PIN, doc_rec["e"][0]["v"]);
    }

  Serial.print("topic: ");
  Serial.println(topic);
  Serial.print("message: ");
  Serial.println(message);
  }
}
```

Nella *setup()*, come negli esercizi precedenti, verifichiamo il corretto funzionamento della funzione *Bridge.begin()* attraverso il led (si accende se l'operazione va a buon fine); usiamo allora la funzione *begin()* per indicare alla Yùn quale sarà il Message Broker a cui fare riferimento per l'invio dei dati come publisher o la ricezione dei dati come subscriber. Attraverso la funzione *subscribe()* ci iscriviamo allora a un topic nel broker, indicando anche la funzione di callback *setLedValue()* da richiamare quando un eventuale publisher esterno invierà dati al nostro Broker sotto quel topic.

Nella *loop()* mettiamo la Yùn in attesa di messaggi mandati al topic a cui si è iscritta, che come detto richiameranno la *setLedValue(),* e lo facciamo con la funzione *monitor()* di mqtt. Dall'altro lato pubblichiamo, a intervalli di 10 secondi, i dati di temperatura rilevati dal sensore ottenuti dalla funzione *readTemp(),* già discussa negli esercizi precedenti, tramite la funzione *publish()*, specificando il topic di pubblicazione (diverso da quello a cui abbiamo iscritto la Yùn nella loop, altrimenti il dato sarebbe rimandato indietro dal Broker). Tutti i dati, che siano inviati o ricevuti dalla Yùn, sono formattati in senML: quindi il messaggio che passiamo con la publish e quello che ci aspettiamo nella *setLedValue()* dovrà essere serializzato (lo si fa nella *senMLEncode*) e deserializzato, rispettivamente.

La funzione di callback *setLedValue()* associata a una comunicazione col Broker sul topic a cui ci siamo iscritti riporta tra i suoi parametri il messaggio ricevuto (pubblicato da qualcuno su quel topic), ovviamente in senML; allora dobbiamo solo occuparci di deserializzarlo con la *deserializeJson()*, ottenendo un dizionario con le chiavi tipiche del senML. La funzione ritorna un codice di errore nel caso la deserializzazione non sia andata a buon fine. A questo punto possiamo semplicemente analizzare il dato ricevuto e agire di conseguenza, scrivendo sul pin del led il valore digitale ricevuto nel messaggio.