Report Laboratori Tecnologie per IOT

Gruppo: 14

Claudio Raccomandato 245488, Giovanni Gaddi 246574, Federico Giuranno 234437

Laboratorio Software Parte 1

Esercizio 1

Per questo esercizio era richiesto di realizzare un convertitore di temperatura tra diverse unità di misura (°C, °K, °F). Doveva essere utilizzata un'architettura REST e programmazione ad oggetti con gestione degli errori.
Il codice che abbiamo realizzato si basa su due classi:

- ErrorManager, contiene un dizionario con tutti i possibili errori e una funzione che permette di richiamarli
- Converter, contiene la GET con tutta la parte di gestione degli errori e una funzione, ConvertValue, che permette la conversione vera e propria attraverso la funzione matematica adeguata

Come richiesto dall'esercizio i parametri vengono passati dall'utente attraverso uri e params, i risultati della conversione vengono scritti su un documento JSON che rappresenta l'output:

```
{
    "value": "10.0",
    "originalUnit": "C",
    "convertedValue": "283.15",
    "targetUnit": "K"
}
```

Figura 1 - JSON output

Esercizio 2

Modificando l'esercizio precedente, era richiesto di realizzare un convertitore di temperatura questa volta passando i dati separati da slash.
La soluzione che abbiamo pensato è stata quella di modificare la classe Converter in modo da utilizzare solo l'uri e non più i params. È stato quindi sufficiente modificare la gestione degli errori nella GET ed i messaggi di errore della classe ErrorManager.

Esercizio 3

Anche in questo esercizio era richiesto di realizzare un convertitore di temperatura, ma questa volta il passaggio dei dati da parte dell'utente doveva avvenire tramite PUT. Una volta inviato un documento JSON, contenente la lista di valori da convertire, il codice doveva restituire un secondo JSON contenente la lista di valori da convertire e quella di valori convertiti.
La nostra soluzione è stata quella di riutilizzare la classe Converter degli esercizi precedenti, in modo da riutilizzare la funzione di conversione, ma sostituendo la GET con una PUT. In quest'ultima viene implementata una try-except che ci permette di controllare la formattazione del JSON e allo stesso tempo di generare il nuovo andando ad aggiungere la lista dei valori convertiti nel documento.


```
{
    "values": [10, 9, 8, 7, 6, 5, 3, 2, 1],
    "originalUnit": "C",
    "convertedValues": [283.15, 282.15, 281.15, 280.15, 279.15, 278.15, 276.15,
275.15, 274.15],
    "targetUnit": "K"
}
```


Figura 2 - JSON output

Esercizio 4

In questo esercizio era richiesto di realizzare una dashboard che doveva essere aperta attraverso metodo GET ed una volta salvata i valori di configurazione dovevano essere ricevuti tramite POST e salvati in un JSON.
Abbiamo quindi realizzato una classe freeboard che tramite GET aprisse il file "freeboard/index.html". Poi abbiamo aggiunto una POST che quando in uri era contenuto il dato "saveDashboard" scrivesse il JSON "freeboard/dashboard/dashboard.json".

Laboratorio Software Parte 2

Esercizio 1

Per questo esercizio era richiesto di realizzare un Catalog che permettesse di registrare e gestire dispositivi, servizi e utenti. Il tutto doveva essere realizzato tramite un'architettura REST e le informazioni dovevano essere memorizzate in dei file.
La soluzione che abbiamo pensato si basa su tre classi:

- DeviceManager, per la registrazione dei dispositivi e la loro ricerca nel JSON
- ServiceManager, per la registrazione dei servizi e la loro ricerca nel JSON
- UserManager, per la registrazione degli utenti e la loro ricerca nel JSON

La registrazione avviene per tutti e tre tramite POST e i JSON ricevuti vengono salvati nei rispettivi file Devices.txt, Services.txt e Users.txt, che in caso non fossero presenti all'avvio vengono generati automaticamente. Quando viene ricevuta una GET agli end-point "/devices", "/services" e "/users", il file corrispondente viene letto e restituisce il dizionario contenente la risorsa.
Una quarta classe chiamata BrokerInfo si occupa di restituire ip e port del broker MQTT quando viene invocata una GET con end-point "/broker".
Infine, altre due classi chiamate Rem_ExpDev e Rem_ExpServ si occupano della cancellazione di dispositivi e servizi, una volta trascorsi due minuti dalla loro registrazione. Entrambe le classi lavorano su thread separati per valutare il tempo trascorso dalla registrazione, ecco un estratto del codice che ci permette di gestire la cancellazione:

```
flag = False
for dev in temp_data.keys():
        if((time.time() - temp_data[dev]['insert-timestamp']) > 120): # 2 minuti
                del dev_data[dev] # cancello la entry
                with open('Devices.txt', 'w') as outfile:
                        json.dump(dev_data, outfile)
                print(time.strftime("%H:%M:%S") + " - [Dev]INFO: Deleted \'" + dev
                      + "\' entry from Devices (added " + time.strftime("%H:%M:%S",
                      time.localtime(temp_data[dev]['insert-timestamp'])) + ").\n")
                flag = True
        if(not flag): print(time.strftime("%H:%M:%S") + " - [Dev]INFO: No entries
                            removed.\n")
        time.sleep(60) # aspetto 1 minuto e poi mando il prossimo loop
```

Figura 3 - Cancellazione Devices

Esercizio 2

In questo esercizio era richiesto di realizzare un Client per ricevere informazioni dal Catalog dell'esercizio precedente.
La nostra idea è stata quella di creare una semplice interfaccia che permettesse di visualizzare e cercare per nome tutti i dispositivi, i servizi e gli utenti registrati sul Catalog. Per le richieste abbiamo creato una classe chiamata GetRequest contenente la libreria request e quattro funzioni getter: getBroker, getDevice, getUser e getService.
Ecco come si presenta l'interfaccia:

```
1 - Visualizza l'indirizzo del Message Broker in uso
2 - Visualizza tutti i Device
3 - Cerca un Device
4 - Visualizza tutti gli User
5 - Cerca un User
6 - Visualizza tutti i Service
7 - Cerca un Service
8 - Esci
Digita il numero dell'operazione da svolgere:
```

Figura 4 - Interfaccia prompt

Esercizio 3

Anche in questo esercizio era richiesto di realizzare un Client, però stavolta doveva simulare un dispositivo IOT che periodicamente si registrava al Catalog.
La nostra idea è stata quella di creare un ciclo while in cui il dispositivo faceva una POST dei dati per la sua registrazione ogni 60 secondi.

Esercizio 4

Per questo esercizio era richiesto modificare l'esercizio 2 del laboratorio Hardware parte 3, in modo da permettere all'Arduino di registrarsi periodicamente al Catalog.
Abbiamo quindi aggiunto una nuova funzione chiamata RegisterDevice dove generiamo il JSON contenente i dati di registrazione. Successivamente abbiamo creato una condizione in più nel loop che ci permette di fare periodicamente il POST del JSON di registrazione al Catalog. Anche nel postRequest è stata aggiunta una condizione che ci permette di cambiare l'indirizzo a cui fare il POST se stiamo registrando il dispositivo. Di seguito riportiamo la nuova funzione di registrazione:

```
String RegisterDevice()
{
  doc_snd.clear();
  doc_snd["deviceID"] = "Yun";
  doc_snd["endpoints"][0] = "/devices/Yun/temperature";
  doc_snd["resources"][0] = "Temperature";

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

Figura 5 - Generazione del JSON di registrazione

Esercizio 5

In questo esercizio era richiesto modificare il Catalog dell'esercizio 1, in modo da permettergli di registrare nuovi dispositivi tramite uno specifico topic MQTT.
La soluzione che abbiamo pensato è stata quella di creare un nuovo client MQTT all'interno del Catalog, che appena avviato fa una subscribe a "tiot/14/deviceadd". Successivamente abbiamo aggiunto due callback, on_connect ed on_message, nel DeviceManager.py. Il modo in cui viene registrato il dispositivo nell'on_message è praticamente analogo al modo in cui viene registrato con la POST.

Esercizio 6

Per questo esercizio veniva richiesto di creare un publisher MQTT che simulasse un dispositivo IOT che periodicamente si registra al Catalog dell'esercizio precedente.
La nostra idea è stata quella di creare un ciclo while in cui il dispositivo faceva una publish dei dati per la sua registrazione al topic "tiot/14/deviceadd" ogni 5 secondi.

Laboratorio Software Parte 3

Esercizio 2

Per questo esercizio era richiesto di realizzare un MQTT subscriber che ricevesse i valori di temperatura inviati dall'Arduino nell'esercizio 3 del laboratorio Hardware parte 3.
La nostra idea è stata quella di ricevere tramite linea di comando il nome del dispositivo in modo da poter accedere ai suoi end-point e visualizzare la temperatura. Per le richieste abbiamo creato una classe chiamata GetRequest contenente la libreria request, due funzioni getter, getBroker e getDevice, e una funzione addService, che tramite POST registra il servizio nel Catalog.
Una volta registrato il servizio e ottenuto l'ip del broker, viene creato un client MQTT che fa una subscribe a tutti gli end-point del dispositivo, in questo modo tramite la callback on_message è possibile ottenere i valori di temperatura inviati dall'Arduino.
In fine abbiamo utilizzato una try-except per uscire dal programma premendo CTRL+C:

```
try:
        while True:
                time.sleep(1)
except KeyboardInterrupt: # premendo CTRL+C puoi interrompere lo script
        print("INFO: Disconnessione in corso...")
        client.disconnect()
        client.loop_stop()
```

Figura 6 - Interruzione del programma con CTRL+C

Esercizio 3

In questo esercizio era richiesto di realizzare un MQTT publisher che controllasse il led dell'Arduino nell'esercizio 3 del laboratorio Hardware parte 3.
Abbiamo quindi modificato il codice dell'esercizio precedente sostituendo la callback on_message con la on_publish, grazie alla quale mostriamo il JSON pubblicato e modificando la callback on_connect in modo che controlli se il topic a cui vogliamo pubblicare è presente tra gli end-point del dispositivo.
Dopo che il client MQTT viene creato, è possibile settare il led tramite linea di comando. In fine viene generato il JSON da pubblicare al topic "tiot/14/led".

Esercizio 4

Per questo esercizio era richiesto di modificare l'esercizio 1 del laboratorio Hardware parte 2, in modo da permettere all'Arduino di registrarsi al Catalog, comunicando tramite MQTT i dati

dei sensori. Inoltre, veniva richiesto di realizzare un servizio che permettesse, sempre tramite MQTT, di pilotare gli attuatori di Arduino.
Abbiamo iniziato modificando il codice dell'Arduino, aggiungendo la libreria MQTTclient e rimuovendo le funzioni che gestivano gli attuatori lasciando le funzioni di PIRManager e NoiseSensorManager, per il controllo della presenza, e GetTemp, per il controllo della temperatura.
Quando il programma inizia l'Arduino fa il subscribe a 3 topic per la gestione del led, della ventola e dell'lcd. Per ognuno di questi abbiamo creato una funzione (setLedValue, setFanValue e setLcdMessage) che riceve in input il JSON e pilota l'attuatore scelto di conseguenza.
Come richiesto dall'esercizio abbiamo aggiunto nel loop una condizione che in base al PUBLISH_TIME fa il publish di temperatura, presenza e rumore periodicamente.
In fine abbiamo realizzato una funzione chiamata RegisterDevice che riutilizza lo stesso documento JSON con cui viene fatto l'invio dei dati per la registrazione del dispositivo al Catalog. Per permettergli di inviare tutte le informazioni necessarie alla registrazione, abbiamo scelto di aumentare la dimensione del documento JSON e di non inviare anche il base topic negli end-point ma di aggiungerlo successivamente nel nuovo servizio:

```
for topic in thisService['endpoints'][0]:
        getEndpoints['getvalues'].append(my_base_topic + topic)
for topic in thisService['endpoints'][1]:
        getEndpoints['setvalues'].append(my_base_topic + topic)
```

Figura 7 - Aggiunta del base topic agli end-points del dispositivo

In questo modo ci siamo assicurati di inviare più informazioni senza occupare tutta la memoria dinamica dell'Arduino.

Per la parte del servizio, la nostra idea è stata quella di ricevere tramite linea di comando il nome del dispositivo in modo da poter accedere ai suoi end-point tramite la classe GetRequest utilizzata nell'esercizio 2. Tramite questa classe è stato possibile registrare il servizio al Catalog tramite POST come richiesto dall'esercizio. Successivamente abbiamo realizzato un piccolo menu che permette di impostare i setpoint di ventola e led tramite linea di comando e di mandare messaggi all'lcd tramite publish.

```
1 - Inserisci manualmente dei setpoints
2 - Invia un messaggio sul display
3 - Esci

Digita il numero dell'operazione da svolgere:
```

Figura 8 - Interfaccia prompt del servizio

Tramite la callback on_connect il servizio fa una subscribe a tutti gli end-points a cui l'Arduino invia informazioni mentre con la callback on_message memorizza i dati ricevuti e setta la ventola e il led di conseguenza, grazie alle funzioni set_FanLed e linProp. In particolare, quest'ultima funzione, fa una proporzione lineare proprio come faceva la funzione di Arduino mappa dell'esercizio 1 del laboratorio Hardware parte 2.

Laboratorio Hardware Parte 1

Esercizio 1

In questo esercizio era richiesto di pilotare due led, uno in polling tramite delay ed uno tramite ISR.
Abbiamo quindi creato una nuova funzione chiamata blinkGreen e nel setup l'abbiamo associata all'interrupt, mentre nel loop facciamo il toggle del led periodicamente in base al valore assegnato alla costante R_HALF_PERIOD.

Esercizio 2

In questo esercizio era richiesto di pilotare due led in modo da accenderli e spegnerli tramite seriale. Gli stati dei due led dovevano essere mostrati in schermo.
La nostra idea è stata quella di controllare la seriale tramite Serial.available e poi tramite Serial.read() memorizzare il valore ricevuto in una char e settare il led corretto tramite uno switch-case.

Esercizio 3

Per questo esercizio era richiesto di utilizzare il sensore PIR come contatore di persone.
Abbiamo quindi creato una nuova funzione chiamata checkPresence e nel setup l'abbiamo associata all'interrupt. Ogni volta che viene chiamato l'interrupt la variabile tot_count incrementa. Ogni 30 secondi viene mostrata nel monitor seriale tramite un Serial.println nel loop.

Esercizio 4

Per questo esercizio era richiesto di impostare la velocità della ventola con dei comandi inviati tramite seriale.
Abbiamo creato una nuova funzione chiamata serialFanControl che viene chiamata in polling nel loop e serve a cambiare il valore della variabile current_speed nel caso arrivano comandi dalla seriale. La seriale viene controllata tramite un Serial.available e una volta salvato il valore lo si elabora con uno switch-case, nel caso in cui il valore di current_speed è maggiore di 255 o minore di 0 il valore viene bloccato.

Esercizio 5

Per questo esercizio era richiesto di leggere ogni 30 secondi il valore del sensore di temperatura e mostrarlo nel monitor seriale.
Abbiamo quindi applicato la formula di conversione per ottenere dal valore di tensione letto dal pin A1 il valore della temperatura corrispondente e poi abbiamo stampato quel valore con una Serial.println. Il tutto viene fatto in polling con un delay di 30 secondi.

Esercizio 6

In questo esercizio era richiesto di modificare l'esercizio precedente in modo da mostrare il valore di temperatura su schermo lcd.
Abbiamo quindi aggiunto la libreria LiquidCrystal_PCF8574 e tramite il comando print della libreria mostrato il valore sull'lcd.

Laboratorio Hardware Parte 2

Esercizio 1

In questo esercizio era richiesto di realizzare un sistema di riscaldamento e condizionamento per smart home. Il dispositivo doveva pilotare la ventola emulando un condizionatore e un led rosso emulando un riscaldatore resistivo. PIR e sensore di rumore dovevano accettarsi se era presente gente all'interno della stanza in modo da cambiare automaticamente i 4 set-point di temperatura. Lcd doveva mostrare le informazioni rilevanti e i set-points dovevano essere impostati tramite seriale. In fine il codice doveva essere modificato per permettere l'accensione di un secondo led tramite doppio battito di mani.

La nostra idea è stata quella di realizzare diverse funzioni che fungessero da Manager per i diversi sensori e poi mettere tutto in polling nel loop.
Il primo Manager che abbiamo realizzato è stato il ConditioningManager, che serviva a pilotare ventola e led in base alla temperatura registrata. Essendo però che i valori di massimo e minimo variavano in base alla presenza di persone nella stanza, abbiamo scelto di realizzare una piccola struct che memorizzasse i due valori di massimo e minimo per sensore:

```
typedef struct
{
  int Min[2];
  int Max[2];
} set_point;

set_point cooler = {{40, 28}, {45, 30}};
set_point heater = {{26, 15}, {30, 20}};
```

Figura 9 - Struct di valori minimi e massimi in base alla presenza

Per fare la proporzione tra i set-points ed il valore su 8 bit accettato dall'analogWrite, abbiamo realizzato una funzione chiamata mappa che funziona in modo simile al map di Arduino, ma tra i parametri vi sono degli interi e ritorna un intero.
In questo modo la funzione del ConditioningManager si riduce a due analogWrite.

```
analogWrite(LED_PIN, mappa(temp, heater.Min[isAnyoneIn_PIR or isAnyoneIn_Noise],
                           heater.Max[isAnyoneIn_PIR or isAnyoneIn_Noise], 255, 0));
analogWrite(FAN_PIN, mappa(temp, cooler.Min[isAnyoneIn_PIR or isAnyoneIn_Noise],
                           cooler.Max[isAnyoneIn_PIR or isAnyoneIn_Noise], 0, 255));
```

Figura 10 - Pilotaggio led e ventola

Successivamente abbiamo realizzato il PIRManager che nel caso in cui viene rilevato movimento nella stanza imposta isAnyoneIn_PIR = true ed il NoiseSensorManager che nel caso un cui gli eventi di rumore sono più di 49 in 10 minuti imposta isAnyoneIn_Noise = true. Per accertarci che gli eventi di rumore fossero più di 49 in un intervallo di 10 minuti qualsiasi, abbiamo creato un vettore chiamato singleSoundsStart nel quale vengono memorizzati i time-stamp degli ultimi 50 eventi di rumore. Una volta che il vettore è stato riempito la variabile arrayFull diventa true e da quel momento il NoiseSensorManager va a controllare il

primo e l'ultimo valore memorizzato all'interno del vettore. Se la differenza dei due time-stamp è minore di 10 minuti allora imposta isAnyoneIn_Noise = true ed inizia i 60 minuti di attesa.

```
if (arrayFull) {  // singleSoundsStart riempito
  if ((singleSoundsStart[sound_count] - singleSoundsStart[(sound_count + 1) %
       n_sound_events]) < sound_interval) {
    isAnyoneIn_Noise = true;
    timeout_sound_start = millis(); // Inizio timeout 60 min
  }
}
```

Figura 11 - Controllo degli eventi di rumore nei 10 minuti

A questo punto invece di svuotare il vettore dei singleSoundsStart semplicemente settiamo false la variabile arrayFull, in questo modo terminati i 60 minuti il vettore verrà nuovamente riscritto prima che venga iniziato il controllo.
Abbiamo scelto di non fare il conteggio degli eventi di rumore in polling ma bensì di creargli una ISR chiamata NoiseSensorDetector, nella quale viene incrementato il valore del sound_count. Essendo però che il sensore di rumore generava più interrupt quando riceveva un rumore, abbiamo scelto di settargli un threshold di 50 ms tramite la variabile singleSoundThreshold in modo che il conteggio degli eventi fosse più preciso.

L' LCDManager si occupa di mostrare i risultati nello schermo lcd e di alternare le due schermate tramite il toggle della variabile LCD_sel ogni 5 secondi. Per mostrare i dati abbiamo preferito usare la funzione di setCursor invece di fare il clear dello schermo e riscrive tutto ogni volta.

Infine, il SerialManager si occupa di ricevere tramite seriale i nuovi valori dei set-points.
I comandi accettati sono del tipo -ABC N

- A può essere 'H' per selezionare il riscaldamento o 'C' per selezionare il condizionatore
- B può essere 0 o 1 rispettivamente se vi è la presenza o meno di persone nella stanza
- C può essere 'M' per selezionare il massimo o 'm' per selezionare il minimo
- N è il valore da sovrascrivere nel setpoint

Bonus:

Per rilevare il doppio battito di mani abbiamo pensato di utilizzare il sound_count, però era necessario impostare anche un limite di tempo molto breve entro cui questi due eventi di rumore dovevano essere registrati. Abbiamo quindi creato una nuova variabile chiamata timeout_LED_bonus che definisse la velocità con cui con cui i due applausi doveveano essere eseguiti. Successivamente abbiamo modificato il NoiseSensorManager in modo che contasse quanto tempo fosse passato tra il primo ed il secondo applauso e nel caso in cui fosse minore del timeout_LED_bonus faceva il toggle del led e settava la variabile clap_detected = true. Infine, per conservare il valore del sound_count abbiamo creato una variabile di appoggio chiamata clap_checker e modificato il NoiseSensorDetector in modo che facesse partire il conteggio non appena il sound_count era uguale al clap_checker + 1, ovvero non appena ci fosse stato il primo applauso.

Laboratorio Hardware Parte 3

Esercizio 1

In questo esercizio era richiesto di usare due metodi GET per settare lo stato di un led collegato ad Arduino e ricevere i valori di temperatura letti dal sensore.
L'idea è stata quella di generare, tramite la libreria Bridge, un BridgeClient che tramite il comando readStringUntil potesse leggere le richieste che gli venivano fatte. Nel caso in cui nella richiesta era contenuto il comando "led", tramite la funzione process veniva settato il led nello stato voluto e veniva mandato un JSON con formattazione senML contenente lo stato del led, mentre se fosse stato contenuto il comando "temperature" sarebbe stato mandato un JSON con formattazione senML contenente la temperatura letta dal sensore.

Esercizio 2

Per questo esercizio era richiesto di inviare tramite POST dall'Arduino i dati del sensore di temperatura. Un server doveva ricevere il JSON contenente i dati e facendo una GET all'end-point "/log" doveva essere mostrata la lista dei JSON inviati dall'Arduino.
Per il server REST abbiamo creato una classe chiamata ArduinoService e nel costruttore gli abbiamo inizializzato la lista che avrebbe contenuto i JSON. Quando viene ricevuto il metodo GET all'end-point "/log", la classe restituisce la lista mentre quando viene fatto il POST aggiunge il JSON ricevuto alla lista.
Per il codice dell'Arduino abbiamo creato una funzione postRequest che tramite curl permette di fare il POST del JSON formattato come senML, grazie alla funzione senMlEncode.
L'Arduino quindi esegue la POST in polling nel loop mentre il tempo di invio può essere impostato tramite la constante REQUEST_TIME.

Esercizio 3

In questo esercizio era richiesto di utilizzare MQTT per pilotare un led e ricevere in dati del sensore di temperatura da remoto.
Per ricevere lo stato del led abbiamo fatto nel setup una subscribe al topic "tiot/14/led" e creato una callback chiamata setLedValue. Tramite la funzione deserializeJson l'Arduino è poi in grado di accedere alle informazioni del JSON inviatogli e di conseguenza ad accendere o spegnere il led.
Nel frattempo, tramite polling, l'Arduino genera periodicamente una publish al topic "tiot/14/temperature" contente il JSON formattato come senML con i dati del sensore di temperatura. L'intervallo di tempo tra una publish e l'altra può essere impostata tramite la constante PUBLISH_TIME.