

**DIPARTIMENTO DI INGEGNERIA NAVALE, ELETTRICA, ELETTRONICA E DELLE TELECOMUNICAZIONI**

**CORSO DI STUDIO IN INGEGNERIA ELETTRONICA E TECNOLOGIE DELL’INFORMAZIONE**

Tesi di Laurea Triennale

Mese e Anno

**Titolo**

Candidato: Nome Cognome

Relatore: Prof. Nome Cognome

Correlatore: Dott. Nome Cognome

**Sommario**

A partire dal mese di giugno 2019 la tesi dovrà essere redatta in accordo alle seguenti regole. Dovrà contenere un sommario, di lunghezza compresa tra le 10 e le 15 righe. Il corpo della tesi dovrà essere scritto con un carattere di dimensione pari a 10 punti, interlinea singolo, margini up, down, top e bottom pari a 2.5 cm, e dovrà essere suddiviso in paragrafi, intitolati “Introduzione”, “Metodi e strumenti utilizzati”, “Sperimentazione e Risultati”, “Contributo personale e considerazioni conclusive” e “Riferimenti bibliografici”. La lunghezza della relazione dovrà essere compresa tra le 18 e le 22 pagine.

La presentazione da portare all’Esame di Laurea dovrà essere una sintesi della tesi e dovrà avere una struttura analoga, contenuta in al più 10 diapositive, da illustrare in non più di 10 minuti.

Per prove finali di tipo compilativo il documento dovrà avere lunghezza maggiore. Al momento non vengono definite regole per questo tipo di prova finale.

Sommario:

Sommario

[Introduzione 4](#_Toc116653436)

### [Metodi e strumenti utilizzati 4](#_Toc116653437)

[Il raspberry pi 4 4](#_Toc116653438)

[Convertitore USB – RS485 4](#_Toc116653439)

[Il protocollo ModBus 4](#_Toc116653440)

[Contatori di Energia 5](#_Toc116653441)

[L’ambiente Node-Red 5](#_Toc116653442)

[Measurify 5](#_Toc116653443)

[Flutter 5](#_Toc116653444)

[Sperimentazione e risultati 5](#_Toc116653445)

[Raspberry 5](#_Toc116653446)

[Contatori 6](#_Toc116653447)

[Node-red 6](#_Toc116653448)

[Il flow di Node-Red 7](#_Toc116653449)

[Il token di autenticazione 7](#_Toc116653450)

[La lettura dell’energy-state 8](#_Toc116653451)

[Lettura del total-energy 9](#_Toc116653452)

[L’applicazione Flutter 12](#_Toc116653453)

[Struttura del codice 12](#_Toc116653454)

[Librerie 12](#_Toc116653455)

[Widget 12](#_Toc116653456)

[Pagine 12](#_Toc116653457)

[Avvio 12](#_Toc116653458)

[Widget Homepage 13](#_Toc116653459)

[Stato Homepage Nella classe \_HomepageState che deriva da State sono definiti tutti gli aspetti che servono per visualizzare il contenuto del widget: 13](#_Toc116653460)

[Interfaccia Homepage 13](#_Toc116653461)

[Aggiornamento Dati 14](#_Toc116653462)

[Dati Measurify 14](#_Toc116653463)

[Classi TAEMeasure e TAESample 15](#_Toc116653464)

[Widget LiveData 16](#_Toc116653465)

[Grafico Lineare 16](#_Toc116653466)

[Timer aggiornamento dati 17](#_Toc116653467)

[Considerazioni conclusive 17](#_Toc116653468)

[Riferimenti bibliografici 17](#_Toc116653469)

# Introduzione

Nel presente elaborato si descrive il funzionamento di un sistema di rilevatori di consumo energetico istantaneo in standard Modbus 485 gestiti localmente con un processore raspberry pi 4.

Il processore colleziona i dati dai contatori ed invia i dati alla cloud measurify.

Sono gestite sia le potenze istantanee (energy-monitor) sia i consumi cumulativi totali (total-active-energy).  
I dati sono consultabili attraverso una pagina web e una app Android.

# Metodi e strumenti utilizzati

Per la realizzazione del progetto sono stai utilizzati:

* scheda raspberry pi4 con s.o. Raspberry Pi OS Lite basato Linux Debian
* contatore di energia elettrica mod f&f ….
* Ambiente di sviluppo lato EDGE: node-red
* Cloud Measurify per l’invio dei campioni e le query al database
* Ambiente di sviluppo lato CLOUD: Flutter

## Il raspberry pi 4

Il raspberry è stato il primo processore SBC (single board computer) in grado di far girare il sistema operativo linux apparso sul mercato. Dispone di un alloggiamento per una memoria microSD da 1 a 8 gb sul quale può essere installato un sistema operativo, nel nostro caso Raspberry Pi OS Lite basato su Debian Linux. Il s.o. Pi OS Lite è senza interfaccia grafica e si utilizza da riga di comando tramite ssh.

La scheda dispone anche di una presa USB-C, usata per l’alimentazione, due porte micro hdmi per connettere due monitor ha 4 prese usb (due di tipo 2 e due di tipo 3), una porta gigabit ethernet e una scheda di rete wifi.

Infine sono disponibili decine di pin di i/o da utilizzarsi per ulteriori interfacciamenti.

## Convertitore USB – RS485

Per creare il bus modbus, si usa un convertitore USB per protocollo RS-485.

In questo modo avremo due linee + e – che formeranno il bus e che andremo a collegare, in parallelo, a tutti i contatori.

## Il protocollo ModBus

Il modbus è un protocollo industriale degli anni ‘80 ma ancor oggi ampiamente usato nel quale ogni byte è serializzato su 7 o 8 bit, con bit di start, eventuale bit di parità, eventuale bit di stop.

La velocità tipicamente è di 9600 baud, ma sovente si incontrano sistemi a velocità di 38400, 115200 o 250000 baud.

Rispetto al protocollo RS232 (che associa i valori 0 e 1 a diverse tensioni (0 e 5V)), nel modbus il dato viene espresso in maniera differenziale, +o-12V, il che rende il tutto molto più stabile ed immune ai disturbi.

Trattandosi di due soli fili, il dialogo coi dispositivi diventa half-duplex, il che necessita, per una buona gestione, di un nodo master che generi i comandi per interrogare uno alla volta tutti i nodi slave. Nessun nodo slave trasmetterà nulla sul bus se non viene interrogato espressamente.

Il modbus consente di collegare fino a 254 dispositivi in parallelo su di un unico bus.

Ogni dispositivo sul bus viene indentificato da un diverso id., distinto tra 1 e 254.

## Contatori di Energia

I contatori impiegati sono del tipo f&f LE-01. Il contatore occupa uno slot DIN ed è in grado di misurare tensione, frequenza, potenza attiva e reattiva istantanea in W e totale potenza erogata in kWh.

Secondo lo standard modbus, i dati sono disponibili su specifiche richieste che devono essere inoltrate sul bus, secondo le specifiche del costruttore questi indirizzi sono:

…..

Per la configurazione dell’indirizzo dei lettori usare l’apposito tool del costruttore

## L’ambiente Node-Red

Node-Red è uno strumento di sviluppo per collegare assieme dispositivi Hardware, API e servizi on-line in un nuovo modo, basato sul flow dell’informazione, organizzata in due elementi fondamentali: il topic (il dato che viene gestito) e il payload (il valore del dato topic).

Node-Red ha un’interfaccia web-based e si usa con un comune browser (Chrome, Firefox, …)

Node-Red è basato sul Node-Js

## Measurify

Measurify è un framework per IoT basato su cloud che offre servizi di archiviazione e interrogazione attraverso un’interfaccia API RESTful

## Flutter

E’ un ambiente di sviluppo open source che permette di sviluppare applicazioni multi piattaforma, compilate nativamente, con la stessa base di codice.   
In questo progetto è stato sviluppato il codice e compilato per un’applicazione web e per un’applicazione Android.  
Il linguaggio di programmazione utilizzato è Dart.

# Sperimentazione e risultati

## Raspberry

L’installazione del sistema operativo sul raspberry non presenta particolari difficoltà. E’ sufficiente inserire una scheda microSD in un PC e caricare l’immagine usando un software come RaspImage, che si scarica l’ultima versione del sistema operativo da internet e la carica sulla scheda SD

La configurazione del Linux è semplice, è sufficiente specificare la rete WIFi alla quale il raspberry si dovrà connettere, un eventuale indirizzo IP statico

## Contatori

Per gli scopi di questo elaborato consideriamo che i tre lettori abbiano indirizzo 1,2 e 3.

Il protocollo modbus prevede inoltre che i dati siano gestiti in specifici registri del dispositivo, permettendo quindi una struttura di query che fa riferimento solo ad un indirizzo, una modalità ed un registro, indipendentemente dal prodotto o costruttore.

I registri da noi usati per l’f’ e f sono ……

Per configurare l’indirizzo all’interno del contatore, utilizzare l’appostio software disponibile sul sito f&f.

## Node-red

L’installazione di Node-Red non presenta particolari difficoltà …..

Al termine dell’installazione bisogna installare i moduli aggiuntivi necessari alla gestione del modbus, che si chiamano ………….

## Il flow di Node-Red

Il flow è stato progettato per leggere i valori di diversi contatori, per esempio “Via all’opera pia 11, primo piano”, “Via all’opera pia 11, secondo piano”, “Via all’opera pia 11, terzo piano”.

I diversi contatori devono avere degli indirizzi modbus differenziati.

Nel nodo Things viene inizializzato l’array dei contatori in questo modo

    Things = [

        { thing: "Via Opera Pia 11a - primo piano", address: 1 },

        { thing: "Via Opera Pia 11a - secondo piano", address: 2 },

        { thing: "Via Opera Pia 11a - terzo piano", address: 3 },

    ]

Il flow realizzato su node-red si divide in tre parti principali:

* Ottenimento del token
* Lettura energy-state
* Lettura total-active-energy

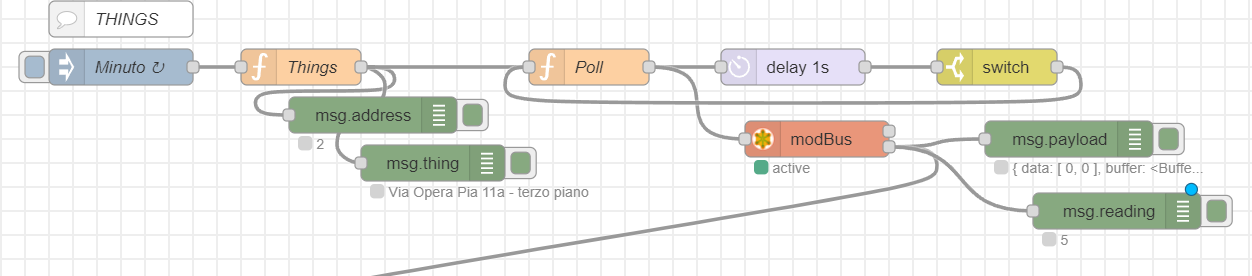
### Il token di autenticazione

Il token di autenticazione fornito da Measurify ha una scadenza di 30 minuti. Per questo motivo ogni 25 minuti viene richiesto un nuovo token.

Il token viene salvato nell’ambiente “global” di Node-Red, per essere reso disponibile a tutti i flow.

### La lettura dell’energy-state

Lo stato attuale del contatore è composto da quattro variabili, che sono la tensione, la frequenza, il consumo in W e l’assorbimento in A.



Questi dati sono contenuti in quattro registri differenti del contatore, e devono essere letti con quattro interrogazioni distinte.

Il formato della query modbus viene realizzato nel nodo “poll”

// Poll - creazione messaggio modbus

// entra msg.reading da 1 a 4

var m=msg.reading;

var ui=msg.address;

var requests = [

    { address: 304, qt: 1 },    // freq

    { address: 320, qt: 2 },    // watts

    { address: 305, qt: 1 },    // volts

    { address: 313, qt: 2 },    // ampere

]

msg.payload = {     // prepara richiesta

    value: msg.payload,

    'fc': 3, 'unitid': ui,

    'address': requests[m-1].address,

    'quantity': requests[m-1].qt

}

msg.reading = m + 1;

return msg;

Il nodo poll esce con msg.reading incrementato di uno, che entra in un nodo “switch”. Se msg.reading è minore di 4 il nodo “switch” restituisce il flusso al nodo “poll” e il ciclo si ripete, altrimenti il ciclo termina.

Per questo, nel flow è stato inserito un loop, che viene ripetuto quattro volte, che legge i quattro valori che compongono lo stato istantaneo.

I valori dello stato vengono salvati in memoria “global” per essere resi disponibili alla procedura che fa il push dei dati a measurify.

Al termine del quarto ciclo, quando tutti i dati sono stati collezionati, viene avviata la procedura che fa il POST dei dati a measurify.

var t = global.get("token");

var v1 = global.get("VAR1");// freq

var v2 = global.get("VAR2");// watts

var v3 = global.get("VAR3");// volts

var v4 = global.get("VAR4"); // ampere

msg.payload = {

    "thing" : msg.thing,

    "feature": "energy-state",

    "device": "energy-monitor",

    "samples": [{ "values": [ v2,v3,v4,v1] }]

};

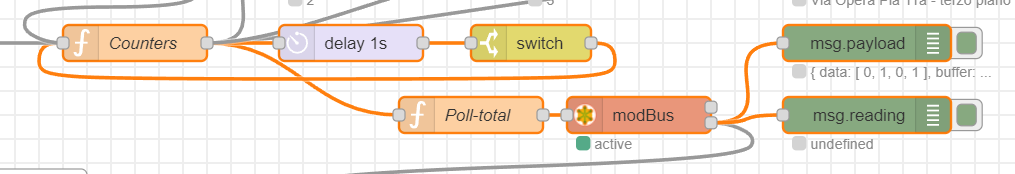
msg.headers = {};

msg.headers['Authorization'] = t;

return msg;

### Lettura del total-energy

La lettura del total-energy viene richiamata solo la sera alle ore 23:59, deve eseguire le letture di tutti e tre i contatori e deve fare tre POST diverse a measurify.



Per questo motivo, nel nodo counters vengono ciclati, uno ogni secondo, tutti i contatori (le things).

// cicla le things

var Things = global.get("Things");  // carica le things

var cur\_counter=0;

if (!isNaN(msg.cur\_counter)) {

    cur\_counter=msg.cur\_counter;

}

msg.address = Things[cur\_counter].address;

msg.thing = Things[cur\_counter].thing;

msg.cur\_counter = cur\_counter+1;

return msg;

Il msg.address preso dalle things passa al nodo che crea la richiesta modbus da inviare ai contatori

// richiesta totale contatori

var ui=msg.address;

msg.payload = { value: msg.payload,

    'fc': 3, 'unitid': ui,

    'address': 40960 , 'quantity': 4 }

return msg;

il risultato del poll al contatore viene usato per fare la post per measurify

il risultato della query modbus è valorizzato nel payload

// fa la POST della total-active-enrgy

var m=msg.reading-1;

var val = 0;

val = (msg.payload.data[0] \* 256 \* 256) + msg.payload.data[1];

val = val / 100;

var t = glob// fa la POST della total-active-enrgy

var m=msg.reading-1;

var val = 0;

val = (msg.payload.data[0] \* 256 \* 256) + msg.payload.data[1];

val = val / 100;

var t = global.get("token");

msg.payload = {

"thing": msg.thing,

"feature": "total-active-energy",

"device": "energy-monitor",

"samples": [{ "values": [val] }]

};

msg.headers = {};

msg.headers['Authorization'] = t;

msg.val=val;

return msg;al.get("token");

msg.payload = {

    "thing": msg.thing,

    "feature": "total-active-energy",

    "device": "energy-monitor",

    "samples": [{ "values": [val] }]

};

msg.headers = {};

msg.he// fa la POST della total-active-enrgy

var m=msg.reading-1;

var val = 0;

val = (msg.payload.data[0] \* 256 \* 256) + msg.payload.data[1];

val = val / 100;

var t = global.get("token");

msg.payload = {

"thing": msg.thing,

"feature": "total-active-energy",

"device": "energy-monitor",

"samples": [{ "values": [val] }]

};

msg.headers = {};

msg.headers['Authorization'] = t;

msg.val=val;

return msg;aders['Authorization'] = t;

msg.val=val;

return msg;

Timing

Mentre la lettura del total-active energy avviene esattamente alle 23:59, la lettura del energy-state avviene periodicamente, con regolarità.

Il timer che scatena il flow di lettura dell’energy-state scatta ogni minuto, per cui ogni minuto vengono letti tutti i dati real-time di un contatore.

Se i contatori gestiti sono tre, ogni contatore viene interrogato ogni tre minuti.

Onde evitare che alle 23:59 vi possa essere una sovrapposizione nelle richieste (il modbus è uno solo e due richieste contemporanee potrebbero essere gestite male dal nodo modbus), si è introdotto un delay di trenta secondi su ogni lettura eseguita ogni minuto.

## L’applicazione Flutter

L’applicazione client che serve per visualizzare i dati dei contatori è stata sviluppata con l’ambiente Flutter in codice Dart.

L’applicazione è composta da un file principale “main.dart” che contiene tutto il codice necessario sia per il backend che per il frontend, questo codice viene poi compilato per la piattaforma di destinazione.  
In questo progetto le due piattaforme previste sono: web e Android.

Per l’applicazione web la compilazione produce una cartella che contiene tutti i file necessari da installare su un server web (es. IIS per Windows)

Per l’applicazione Android la compilazione produce un file eseguibile .apk pronto per essere installato su un dispositivo mobile Android

### Struttura del codice

### Librerie

Nella parte iniziale del file main.dart ci sono i riferimenti alle librerie utilizzate nel codice:

**import 'dart:async'**;  
**import 'dart:convert'**;  
**import 'dart:io'**;  
**import 'dart:developer' as** developer;  
**import 'package:flutter/material.dart'**;  
**import 'package:http/http.dart' as** http;  
**import 'package:date\_time\_picker/date\_time\_picker.dart'**;  
**import 'package:intl/intl.dart'**;  
**import 'package:charts\_flutter/flutter.dart' as** charts;

Le diverse librerie aggiungono funzionalità all’applicazione, es.

* http.dart serve per poter eseguire le chiamate http POST e GET al cloud di Measurify
* date\_time\_picker.dart serve per poter utilizzare l’interfaccia di selezione data e ora nel frontend

### Widget

In Flutter l’interfaccia utente è composta da Widget. I Widget descrivono il loro aspetto, la loro configurazione e il loro stato. Ci sono due tipi di Widget, quelli senza stato StatelessWidget e quelli con stato StatefulWidget.

I primi servono per quelle parti di interfaccia dove non occorre modificare dinamicamente l’aspetto e il contenuto, mentre i secondi servono quando occorre aggiornare l’interfaccia dinamicamente sulla base di un certo timer o di uno stato di sistema.

### Pagine

L’applicazione è composta da due pagine:

* La pagina iniziale **Homepage** visualizza i dati Total-Active-Energy in formato tabellare, è possibile selezionare il periodo e il piano
* La seconda pagina **LiveData** visualizza i dati live Energy-State in formato grafico lineare. E’ possibile selezionare il piano e la grandezza. La pagina si aggiorna automaticamente ogni 15”

### Avvio

L’esecuzione del codice parte dalla funzione main() al suo interno la funzione runApp() riceve in ingresso un widget e lo imposta come widget principale da visualizzare a schermo.

In questo caso viene eseguito il widget Homepage() all’interno del widget MaterialApp che server per impostare lo stile dell’app in modalità Material Design.

**void** main() {  
 runApp(**const** MaterialApp(home: Homepage(),));  
}

## Widget Homepage

Il widget Homepage() è definito nell’omonima classe che deriva da StatefulWidget e al suo interno è definito lo stato attraverso il metodo createState()

*//classe per prima pagina con dati total-active-energy***class** Homepage **extends** StatefulWidget {  
 **const** Homepage({**super**.key});  
  
 @override  
 State<Homepage> createState() => \_HomepageState();  
}

### Stato Homepage Nella classe \_HomepageState che deriva da State sono definiti tutti gli aspetti che servono per visualizzare il contenuto del widget:

*//classe per stato della prima pagina con dati total-active-energy***class** \_HomepageState **extends** State<Homepage> {  
 *//definizione della lista di misure di tipo Future* **late** Future<List<TAEMeasure>> **futureMeasures**;  
  
 *//inizializzo le date e la dropdown dei piani* String **cDateI** = DateFormat(**'yyyy'**).format(DateTime.now()) + **'-'** + DateFormat(**'MM'**).format(DateTime.now()) + **'-01'**;  
 String **cDateF** = DateFormat(**'yyyy'**).format(DateTime.now()) + **'-'** + DateFormat(**'MM'**).format(DateTime.now()) + **'-'** + DateFormat(**'dd'**).format(DateTime.now());  
 String **cPiano** = **'Scegli il Piano'**;  
 **var Piani** = [**'Scegli il Piano'**,**'Via Opera Pia 11a - primo piano'**,**'Via Opera Pia 11a - secondo piano'**,**'Via Opera Pia 11a - terzo piano'**];  
  
 @override  
 **void** initState() {  
 **super**.initState();  
 **futureMeasures** = getMeasuresList(**cDateI**,**cDateF**,**cPiano**);  
 }

In particolare si definisce la variabile futureMeasures che è una lista di elementi (la classe TAEMeasure definisce le misure) ed è di tipo **Future.**

Il tipo Future caratterizza un elemento come il risultato di una operazione asincrona per cui il valore non è disponibile subito ma solo quando l’operazione è completata.

Nel nostro caso la lista di misure viene recuperata tramite la funzione getMeasuresList() che effettua una chiamata http GET al cloud Measurify per cui sarà pronta solo dopo la risposta http.

*Nota*  
In questa parte di codice si potrebbe apportare un miglioramento, la definizione dei piani attualmente è hard-coded con un array statico. Una soluzione più raffinata prevede di recuperare i piani direttamente da Measurify con l’apposita API.

### Interfaccia Homepage

Sempre nella classe \_HomepageState si descrive la parte di interfaccia utente rappresentata dal widget Homepage. Attraverso la direttiva Widget build() di definiscono i diversi widget che appariranno sullo schermo, viene creata una barra per il titolo (AppBar) e un contenitore (Column) con diversi widget figli:

* pulsante per navigare alla pagina LiveData (ElevatedButton)
* menu a tendina per la scelta del piano (DropDownButton)
* 2 caselle per la scelta delle date (DateTimePicker)
* una lista che conterrà le misure (ListView). Essendo le misure un dato **Future** il widget ListView deve essere all’interno di un FutureBuilder

@override  
Widget build(BuildContext context) {  
 **return** MaterialApp(  
 title: **'Energy Manager'**,  
 theme: ThemeData(primarySwatch: Colors.*blue*,),  
 home: Scaffold(  
 appBar: AppBar(title: **const** Text(**'Energy Manager'**),),  
 body: SingleChildScrollView(padding: EdgeInsets.only(left: 20,

right: 20, top: 10),  
 child: Form(  
 child: Column(  
 children: <Widget>[  
 ElevatedButton(…),  
 DropdownButton(…),  
 DateTimePicker(…),  
 DateTimePicker(…),  
 FutureBuilder<List<TAEMeasure>>(future: **futureMeasures**,

builder: (context, snapshot) {

**if** (snapshot.**hasData**) {  
 List<TAEMeasure> data = snapshot.**data**!;  
 **return** ListView.builder(…);

**)**] ), ), ), ) );}

## Aggiornamento Dati

Quando l’utente sceglie il piano e le date, automaticamente vengono recuperati i dati da Measurify e mostrati a video. Per intercettare l’azione dell’utente occorre aggiungere nei vari widget interessati la gestione dell’evento:

onChanged: (String? newValue) {  
 **cPiano** = newValue!;  
 filterResult();  
},

In questo modo quando cambia il valore (del piano o delle date), viene aggiornata la variabile locale e poi viene chiamata la funzione filterResult() che aggiorna la lista di misure sulla base dei nuovi valori di piano e date.

**void** filterResult(){  
 setState(() {  
 **futureMeasures** = getMeasuresList(**cDateI**,**cDateF**,**cPiano**);  
 });  
}

Nella funzione filterResult() viene chiamata la funzione setState del framework Dart. La funzione setState ha il compito di informare il framework che è cambiato lo stato del widget e aggiornare l’interfaccia utente. La funzione setState riceve in ingresso il nuovo valore e la relativa funzione che lo aggiorna.

### Dati Measurify

I dati provenienti dal cloud Measurify sono recuperati con la funzione getMeasuresList(), dove entrano il piano e le date e esce una lista di misure di tipio Future:

Future<List<TAEMeasure>> getMeasuresList(String sDate,String eDate, String cPiano) **async** {  
  
 *//prima viene fatta una chiamata POST di login* **final** response1 = **await** http.post(  
 Uri.*parse*(**'https://students.measurify.org/v1/login'**),  
 headers: <String, String>{**'Content-Type'**: **'application/json;**

**charset=UTF-8'**,  
 }, body: jsonEncode (<String, String> {  
 **'username'** : **'xxx'**, **'password'** : **'xxx'**, **'tenant'**: **'xxx'** }),);  
  
 **if** (response1.**statusCode** == 200) {  
 *//se il login è positivo viene salvato il token per la chiamata*

*successiva* **dynamic** jsonres=jsonDecode(response1.**body**);  
 \_logintoken =jsonres[**'token'**].toString();  
  
 List<TAEMeasure> samples\_list;  
 **if** (cPiano!=**'Scegli il Piano'**) {  
 String add\_sDate=**''**;  
 **if** (sDate!=**''**){  
 add\_sDate=**', "startDate": { "\$gt":"**$sDate**", "\$lt":"**$eDate**\T23:59:59"}'**;  
 }  
   
 *//chiamata GET per recuperare le misure di un piano/periodo* **final** response = **await** http.get(  
Uri.*parse*(**'https://students.measurify.org/v1/measurements?filter={"feature":"total-active-energy", "thing":"**$cPiano**"**$add\_sDate**}'**),  
 headers: {  
 HttpHeaders.*authorizationHeader*: \_logintoken,  
 },  
 );

Per prima cosa si effettua una chiamata http POST inviando le credenziali di accesso a Measurify e ottenendo in risposta il token per le chiamate successive.

Successivamente, se la chiamata di login ha avuto successo (http 200), si effettua una chiamata http GET con il token di autenticazione e passando i parametri relativi al piano e alle date.

**if** (response.**statusCode** == 200) {  
 **dynamic** jsonres=jsonDecode(response.**body**);  
 **final** parsed = jsonres[**'docs'**].cast<Map<String, **dynamic**>>();  
 List<TAEMeasure> samples\_list = parsed.map<TAEMeasure>((json) => TAEMeasure.fromJson(json)).toList();  
  
 **for** (TAEMeasure isample **in** samples\_list) {  
 List samples = isample.**samples**.toList();  
  
 List<TAESample> sample\_list = isample.**samples**.map<TAESample>((json) => TAESample.fromJson(json)).toList();  
}  
 **return** samples\_list;

Se anche la chiamata GET ha avuto successo (http 200) allora si effettua il parsing della risposta JSON e si mappa la struttura del JSON con gli oggetti appositamente creati TAEMeasure e TAESample.  
Il risultato in uscita sarà una lista di misure pronte per essere visualizzate nell’interfaccia utente.   
  
*Nota*

Si potrebbe apportare un miglioramento, in quanto il token di autenticazione ha validità 30 minuti, si potrebbe quindi scorporare la chiamata login che recupera il token dalla funzione getMeasuresList() e ripetere il login solo quando il token è scaduto.

### Classi TAEMeasure e TAESample

Al fine di poter gestire con semplicità i dati ottenuti dal cloud, sono state create due classi che riflettono la struttura del JSON ottenuto da Measurify. La classe TAEMeasure definisce la struttura di una misura:

*//classe che definisce una misura, i valori sono una lista di TAESample***class** TAEMeasure {  
 **final** String **id**;  
 **final** String **thing**;  
 **final** String **feature**;  
 **final** String **device**;  
 **final** List **samples**;  
 **final** String **startdate**;  
  
 **const** TAEMeasure({  
 **required this**.**id**,  
 **required this**.**thing**,  
 **required this**.**feature**,  
 **required this**.**device**,  
 **required this**.**samples**,  
 **required this**.**startdate** });  
  
 **factory** TAEMeasure.fromJson(Map<String, **dynamic**> json) {  
 **return** TAEMeasure(  
 id: json[**'\_id'**] **as** String,  
 thing: json[**'thing'**] **as** String,  
 feature: json[**'feature'**] **as** String,  
 device: json[**'device'**] **as** String,  
 samples: json[**'samples'**] **as** List,  
 startdate: json[**'startDate'**] **as** String  
 );  
 }  
}

In particolare la proprietà sample è una lista e in fase di mappatura viene riempita con oggetti TAESample che definiscono i valori di una misura:

*//classe che definisce i valori di una misura, campo TAEMeasure.samples***class** TAESample {  
 **final** List **values**;  
  
 **const** TAESample({  
 **required this**.**values** });  
  
 **factory** TAESample.fromJson(Map<String, **dynamic**> json) {  
 **return** TAESample(values: json[**'values'**] **as** List);  
 }  
}

## Widget LiveData

La seconda pagina dell’applicazione, accessibile dal pulsante “LiveData” in Homepage, ha una struttura molto simile alla pagina Homepage.

Il widget principale è sempre di tipo StatefulWidget, sono presenti dei menu a tendina per la selezione del piano e della grandezza da visualizzare.

La principale differenza è che i dati sono visualizzati in forma di grafico lineare con aggiornamento automatico ogni 15”.

### Grafico Lineare

Per visualizzare il grafico viene usata la libreria:

**import 'package:charts\_flutter/flutter.dart' as** charts;

E nella direttiva Widget build() si utilizza il widget charts.TimeSeriesChart sempre all’interno di un widget FutureBuilder:

FutureBuilder<List<TAEMeasure>>(  
 future: **liveMeasures**, builder: (context, snapshot) {  
 **if** (snapshot.**hasData**) {  
 List<TAEMeasure> data = snapshot.**data**!;  
 **return** Expanded(child: **new** charts.TimeSeriesChart(

Il widget TimeSeriesChart utilizza una serie per la creazione del grafico, la serie viene popolata partendo dai dati ricevuti dal cloud Measurify in maniera analoga a quando descritto in precedenza.

### Timer aggiornamento dati

Nella classe che definisce lo stato del widget LiveData è stato aggiunto un oggetto Timer e relativa funzione filterLiveT che occorre allo scattare del timer:

*//timer per aggiornamento dati***late** Timer **tmr1**;

**tmr1** = Timer.periodic(Duration(seconds: 15), filterLiveT);

*//funzione che aggiorna le misure, viene chiamata dal timer***void** filterLiveT(Timer tmr1){  
 setState(() {  
 **liveMeasures** = getLiveList(**cPiano**);  
 **cTime** = **'Ultimo aggiornamento: '** + DateFormat(**'dd/MM/yyyy HH:mm:ss'**).format(DateTime.now());  
 });  
}

La funzione filterLiveT, utilizza la direttiva setState per informare il framework del cambio di stato e di conseguenza aggiornare l’interfaccia utente.

# Considerazioni conclusive

Il sistema sopra descritto può funzionare come base per analisi più complesse utili alla realizzazione di un sistema di gestione semi-automatizzato.

Se installato in un B&B è in grado di dare indicazioni circa la presenza o assenza degli ospiti, e combinando assieme i consumi dei singoli apparecchi è alle volte possibile stabilire quali apparecchi sono accesi e quali no (es. frigorifero).

Un sistema come quello descritto in questo elaborato può costituire un mattone di base per lo sviluppo di sistemi di domotica più avanzati, in grado anche di azionare le singole utenze come frigorifero, aria condizionata, ecc…

Contenuto chiarito dal titolo

# Riferimenti bibliografici

[1] N. Cognome, “Titolo”, rivista, volume, numero, pagine, mese, anno

[2] N. Cognome, Titolo, Casa Editrice, anno, edizione, luogo di pubblicazione

Oggetto:

Lettori gestiti

Software Raspberry

Il raspberry è equipaggiato con sistema operativo …. e monta l’ambiente … node-red.

Node-red è un ambiente di sviluppo molto comodo e flessibile per sviluppare prodotti custom o che si inetrfacciano a diversi ambienti, sia hardware (come modbus, can-bus, bluetooth) che  software (come web, tcp, social, email ecc.)

Applicazione Node-Red

L’applicazione Node Red interroga ciclicamente tutti i lettori e invia i dati al server measurify.

Sono distinte due funzioni:

A

Acquisizione dati live, real time, (frequenza, tensione, potenza e …)

Per far ciò un timer è impostato ad un minuto, e il sistema cicla ogni minuto ciascun lettore, leggendo tutti e quattro i dati richiesti.

B

Acquisizione dei totali, eseguita una volta al giorno, alle 23:59

Dashboard locale

La Dashboard locale è gestita dal plug-in di node-red e fornisce informazioni sulla

* uso della cpu
* consumo istantaneo di ciascun lettore
* totale
* grafici storici giornalieri, settimanali e mensili.

L’ambiente Measurify

In MEasurify sono stati definite tre things

things e festures

La APP flutter

Il prototipo

riferimenti

documentazione lettori f&f

Riferimenti:

Debian Linux

Node-Red

Node-Js

F&F  - [www.fif.com.pl](http://www.fif.com.pl)

\_