

Università del Piemonte Orientale

Dipartimento di Scienze e Innovazione Tecnologica

**Corso di Laurea in Informatica**

Rapporto Tecnico per

**CARCHARGER**

(PISSIR)

**Candidati**:

* Campora Emanuele 20029451
* Poletto Matteo 20029178

**Anno Accademico 2022/23**

**1. Specifiche funzionali che devono essere soddisfatte.**

Il software che abbiamo sviluppato si occupa della **gestione delle colonnine in una determinata zona circostante il client**. Le specifiche funzionali che abbiamo identificato sono quindi quelle necessarie a adempire a questo compito. In particolare, abbiamo immaginato che un eventuale committente possa richiedere le seguenti caratteristiche:

1. **Geolocalizzazione adattiva (necessaria per mostrare la zona intorno al client)**
2. **Gestione dei dati delle colonnine e salvataggio degli stessi**
3. **Gestione utenze collegate al software e memorizzazione delle stesse**
4. **Gestione prenotazioni delle colonnine e memorizzazione delle stesse**

In particolare:

* **Geolocalizzazione adattiva**
* Raccolta dati zona circostante (per facilità nel nostro esempio sfrutteremo la città di MILANO, dato che abbiamo reperito dal sito ufficiale del comune l’insieme delle colonnine nella zona)
* **Gestione dei dati delle colonnine e salvataggio degli stessi**
* Struttura del DB a più campi (memorizzare indirizzo, zona, compagnia di appartenenza, ecc)
* Consultazione dei dati relativi a una data colonnina in qualsiasi momento
* **Gestione utenze collegate al software e memorizzazione delle stesse**
* Utilizzo di una sezione apposita del DB per il record delle utenze
* **Gestione prenotazioni delle colonnine e memorizzazione delle stesse**
* Gestione dinamica stato delle colonnine (LIBERA, PRENOTATA, OCCUPATA)
* Conferma prenotazione (OTP da inserire sulla colonnina)
* TIMER tempo rimanente al tempo di carica
* TIMER per conferma prenotazione (15 min)
* Record prenotazioni passate (eliminate ogni 5 min se concluse) e della prenotazione corrente

Le specifiche appena elencate coinvolgono in parte un insieme di sensori necessari per emulare scenari non riproducibili appieno nella realtà (anche a causa di alcune assunzioni che vedremo nel corso del presente rapporto). Sfruttiamo dei **sensori** per le specifiche viste nei punti **4d e 4e**, dove troviamo il coinvolgimento di ipotetiche colonnine e veicoli che non abbiamo materialmente a disposizione.

Il software inoltre appoggia alcune sue specifiche a dei requisiti specifici di rete ma che non impongono limitazioni in merito al suo utilizzo se non dettate dal buon senso (non avrebbe senso utilizzare il software da casa propria). **Il software è quindi accessibile in qualsiasi momento e da qualsiasi client**, con alcune differenze che vedremo in seguito dettate dal fatto che l’utente possa essere loggato o meno.

In particolare, sarà necessario avere a disposizione la possibilità di **geolocalizzare il client** (che non dovrà usare una VPN di conseguenza) per soddisfare il requisito al punto **1a**, uno dei punti fondamentali tra le specifiche del sistema che altrimenti avrebbe un’utilità notevolmente ridotta. Inoltre, i punti **4a,4b e 4c** richiedono una **comunicazione Client-Server estremamente veloce e responsiva**, vedremo successivamente come soddisfare questo requisito in maniera tale da evitare problemi come prenotazioni simultanee di una stessa colonnina.

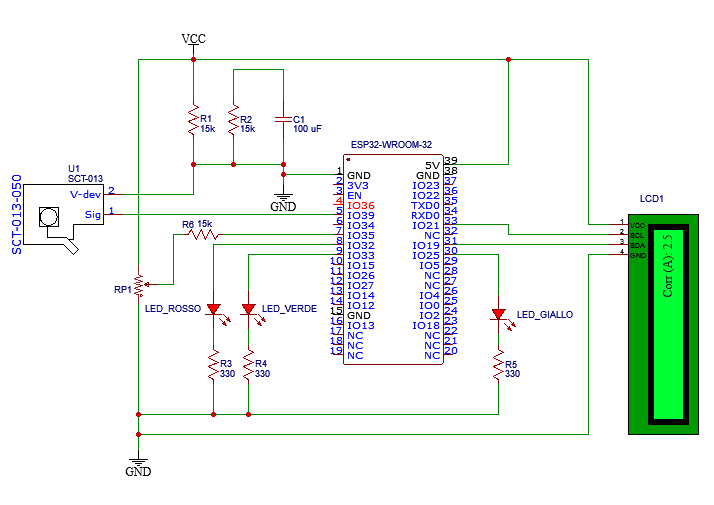
**2. Analisi delle scelte tecnologiche disponibili per realizzare le funzionalità richieste.**

Per riuscire nello sviluppo di un software funzionante e stabile abbiamo impiegato alcune componenti Hardware che vedremo nello specifico prima di affrontare i vincoli di rete che abbiamo incontrato.

In particolare ci siamo serviti di:

* ***ESP32***
* ***Beaglebone***

***ESP32*** (di cui riportiamo di seguito lo schema elettrico) si è resa necessaria per rilevare il passaggio di corrente di una colonnina.



Nello schema elettrico sono collegati i seguenti componenti:

* un **SCT-030**, Pinza Amperometrica usata per misurare la corrente è collegata sul pin 39 e a un circuito elettrico;
* un **Display LCD**, usato per mostrare all’utente le varie informazioni della colonnina (corrente, percentuale della batteria dell’auto) il quale ha collegato VCC e GND ai rispettivi, e SCL collegato al pin 21 e SDA collegato al pin 19;
* **3 LED**: Rosso, Verde e Giallo collegati rispettivamente ai pin 32,33,25, che servono per rappresentare lo stato della colonnina: occupata, libera, prenotata;
* un **Trimmer** per simulare la percentuale della batteria dell’auto, collegato al pin 35;

Nella **sez4** approfondiremo meglio il suo funzionamento a livello software.

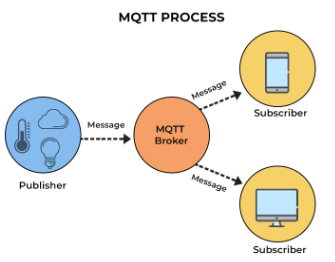
Per mantenere inalterato il software sia che vi si acceda da una rete privata che da una rete pubblica **abbiamo sfruttato un tipo di comunicazione STATELESS**, questo per fornire al sistema robustezza e soprattutto per una **corretta gestione del parallelismo**. Questo secondo aspetto è di vitale importanza per il problema menzionato precedentemente inerente alle prenotazioni simultanee di una stessa colonnina. **Ogni prenotazione** da un client (che deve essere necessariamente loggato) rende PRENOTATA una colonnina. In questo stato una colonnina rimane visibile ma non interagibile per un tempo prefissato di 15 minuti.

**3. Scelta dell’approccio tecnologico.**

I dati delle colonnine elettriche sono comunicati utilizzando ***Mosquitto Eclipse***, con cui è stato possibile coordinare i messaggi tra i diversi client.

**Il protocollo MQTT** adotta un paradigma di comunicazione basato sul modello

***publish/subscribe***. A differenza di un modello *request/response* (utilizzato ad esempio nel protocollo HTTP), il modello *publish/subscribe* **disaccoppia la produzione e la ricezione di un messaggio dal punto di vista temporale**.



Lo scambio dei messaggi nel protocollo MQTT avviene tramite un **broker**, ovvero **un server in grado di ricevere messaggi dai publisher e di consegnarli ai subscriber anche a distanza di tempo**.

Ovviamente il broker si occupa di consegnare soltanto i messaggi che fanno fede ai topic ai quali il ricevente ha fatto il subscribe.

Nel nostro caso utilizziamo **due broker**:

* uno per la gestione delle colonnine in una determinata zona (***BeagleBone***)
* uno per la gestione di arrivo dati elaborati (***macchina virtuale***)

Questi sono stati configurati in modo di garantire una connessione sicura tramite autenticazione (***BeagleBone*** usa autenticazione username/password per ARDUINO, certificati per il client; ***macchina virtuale*** effettua un’autenticazione solo tramite certificati), onde evitare connessione da parte di esterni.  
La scelta dei **topic** (categorie in cui i messaggi sono raggruppati logicamente) permetto inoltre di raggruppare i dati della colonnina e l’OTP inviato/ricevuto (vedasi dopo).

**4. Architettura del software derivante dalle scelte effettuate.**

Il software si compone di alcune classi vitali di cui approfondiremo meglio la struttura. Prima però, come accennato nella **sez2**, riprendiamo il funzionamento di ***ESP32***e ***BeagleBone*** a livello software.

Per quanto concerne ESP32, all’interno del codice si usano le varie librerie e metodi per effettuare i seguenti passaggi per avere un corretto funzionamento del microcontrollore:

**Setup**

1. Si inizializza il Display LCD e il Serial Monitor

2. Si connette al WiFi usando le credenziali segnate nelle variabili apposite (stampando anche su Display i vari passaggi)

3. Una volta connesso al WiFi, si assegna al client MQTT il client WiFi, il server MQTT che sarà assegnato a 0.0.0.0 con porta 1883, e si assegna la funzione di callback.

4. Si impostano in OUTPUT i 3 pin a cui sono collegati i LED

5. Si assegna un'attenuazione ai pin in cui sono collegati la pinza e il trimmer, per ridurre i disturbi

6. Si effettua una prima lettura di corrente, di calibrazione

Siccome il server MQTT è impostato a 0.0.0.0, si effettua una ricerca all’interno della rete per trovare il broker presente nella BeagleBone, utilizzando una funzione apposita. Difatti all’interno del **loop**:

1. Se l’IP dell’MQTT server è 0.0.0.0, dato l’hostname, username e password si trova l’IP con la funzione menzionata precedentemente. Una volta trovato, si imposta il server con l’IP effettivo.
2. Una volta collegato al broker, si esegue il loop effettivo dell’ESP32: calcolo la media di corrente, leggo il valore del trimmer, dichiaro il valore della batteria (costante di 30 kWhr), creo il messaggio JSON da inviare alla BeagleBone, invio il messaggio sul Topic “event/nome\_dispositivo/nome\_auto/pin\_sct/dati” (dove nome\_dispositivo è l’ID della colonnina, pin\_sct è il pin a cui è collegata la pinza) e stampo la media di corrente e la percentuale di batteria sul display
3. Fatto questo, leggo se l’utente ha inserito un OTP: se l’ha inserito, creo il messaggio e lo pubblico sulla BeagleBone sul Topic “event/nome\_dispositivo/pin\_trimmer/otp”

Oltre a inviare i messaggi MQTT al broker, l’**ESP32 riceve i messaggi dalla BeagleBone**, che verranno gestiti nella funzione **callback**:

1. La funzione viene richiamata ogni volta che riceve un messaggio, scritto in byte sulla variabile payload, e il topic viene scritto sulla variabile topic
2. Si converte da byte in char e:

* Se il messaggio ricevuto è sul topic /stato, vado a verificare che stato è (libera, prenotata, occupata), e accendo il led corrispondente.
* Se il messaggio ricevuto è sul topic /OTP, stampo il risultato della verifica dell’OTP sia sul Serial Monitor, sia su Display (true, false).

In merito alla **BeagleBone** invece troviamo al suo interno una cartella con nome “BeagleBone\_DataController”. All’interno è presente un server in JS, ma la parte che interessa a noi è il file **mqtt.js**, del quale approfondiremo una sezione**.**

All’interno di questo file abbiamo scritto in codice una inizializzazione dei client MQTT presente nella BeagleBone e presente all’interno della macchina virtuale (EC2), collegandosi usando i certificati appositi.

Una volta inizializzati i client, quando è stabilita una connessione si stampa un messaggio di successo, e si iscrivono ai seguenti topic (dove i ... indicano qualsiasi valore):

**Client della BeagleBone:**

* + event/.../.../.../dati: si ricevono i dati dalla colonnina (corrente, percentuale di batteria)
  + event/.../.../otp: dove si riceve l’otp inserito alla colonnina

**Client della Macchina Virtuale:**

* + event/EC2/OTPresult: dove si riceve il risultato della verifica dell’OTP, effettuata nella macchina virtuale (true, false)
  + event/EC2/stato: dove si riceve lo stato della colonnina (libera, prenotata, occupata)

In entrambi i client, abbiamo che ogni volta che si riceve un messaggio, si verifica se il messaggio ricevuto sia effettivamente in formato JSON poi, a seconda del client:

**BeagleBone**

1. Se il messaggio ricevuto è sul topic /dati, si effettua il seguente calcolo:
   * si calcola i kWhr passanti: valore della corrente \* tensione della zona)
   * si calcola la capacità attuale della batteria: capacità \* (percentuale / 100)
   * si calcola la differenza (in kWhr) tra la batteria all’80% e la capacità attuale e si arrotonda per eccesso (Math.ceil)
   * se la differenza è maggiore di 0, faccio la differenza diviso i kilowattora passanti, moltiplico per 60 e arrotondo per eccesso, così da avere il tempo di ricarica in minuti
   * se la differenza è minore di 0, vuol dire che la batteria è già oltre all’80%, quindi non ho bisogno di caricarla.
   * mi segno l’ora in cui è arrivato il dato con new Date()
   * Se il tempo di carica è maggiore di 0, vuol dire che ho la colonnina occupata, altrimenti libera
   * creo il messaggio da inviare al broker della macchina virtuale sul topic event/BB/tempoCarica
2. Se il messaggio ricevuto è sul topic /otp ,
   * creo il messaggio con l’otp e l’id della colonnina, che ho ricevuto, per la macchina virtuale sul topic event/BB/otpSent
3. Eseguo una publish sul client della macchina virtuale, poiché devo pubblicare sulla macchina virtuale il messaggio e non sulla BeagleBone

**EC2**

1. Se il messaggio ricevuto è sul topic /stato, creo il messaggio da pubblicare sulla BeagleBone sul topic event/id/stato, dove id è l’id della colonnina
2. Se il messaggio ricevuto è sul topic /OTPresult, creo il messaggio da pubblicare sulla BeagleBone sul topic event/id/OTPresult
3. Pubblico il messaggio sul broker della BeagleBone

Poiché il codice funziona correttamente, **all’interno della BeagleBone abbiamo scaricato e installato pm2. Esso è un process manager** **per rendere automatizzato l’esecuzione del codice**, difatti ogni volta che si accenderà la BeagleBone, il codice partirà automaticamente senza interazioni da parte del programmatore. [Qui](https://pm2.keymetrics.io/docs/usage/quick-start/) possiamo trovare la documentazione, ma a noi interessa soltanto i comandi:

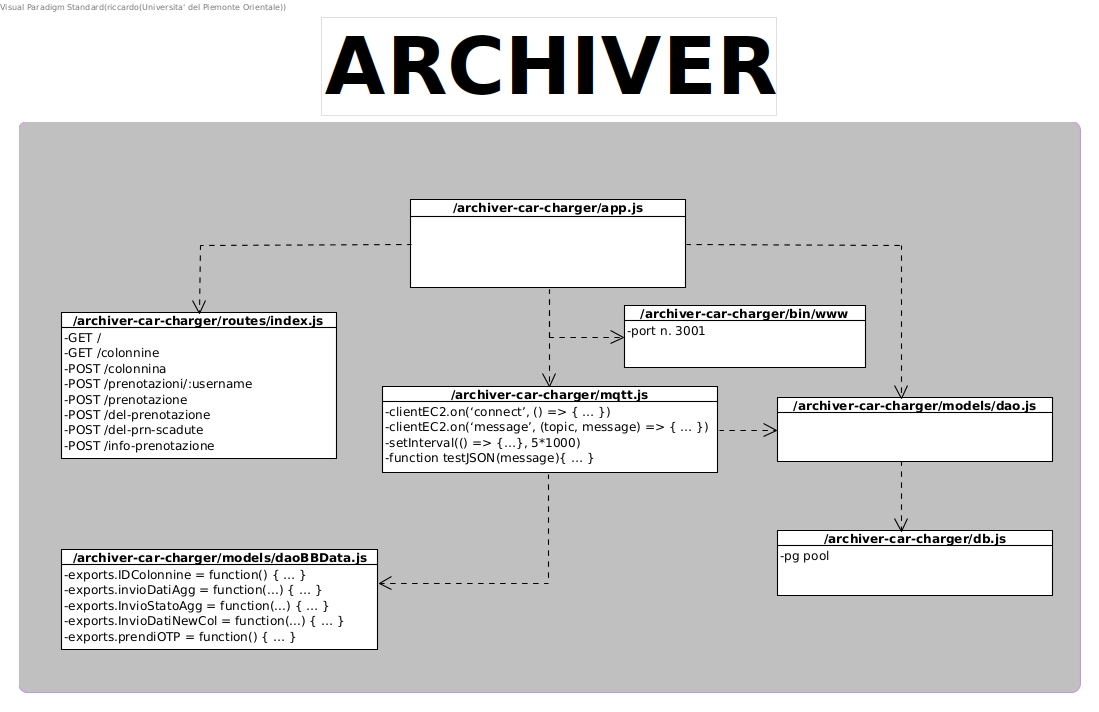
* pm2 list: per vedere i processi in esecuzione
* pm2 logs id\_processo: per vedere le varie stampe che ha effettuato il codice
* pm2 start/stop/restart id\_processo: per avviare, fermare, riavviare il processo che sta eseguendo il codice.

**Inoltre, all’interno della Beaglebone,** più precisamente all’interno del file .bashrc, abbiamo scritto due righe di codice in cui fermiamo mosquitto e lo facciamo ripartire con il file di configurazione personalizzato.

**5. Descrizione dell’implementazione effettuata.**

In questa sezione andremo ad analizzare la struttura del software nelle sue due macroaree:

* ***ARCHIVER***
* ***INTERFACCIA***



**Porta**: 3001

***File routes:*** */archiver-car-charger/routes/index.js*

* **GET** ‘/’ fa una query al database per prendere i dati delle prime 10 colonnine (per semplificare), se la query non va a buon fine, restituisce un array vuoto come array delle colonnine con anche un messaggio di errore nella variabile message
* **Valori ritornati:** variabile message (stringa vuota in caso di esito positivo, stringa piena in caso di esito negativo) e array contenente i valori delle colonnine (array vuoto in caso di errore)
* **GET** ‘/colonnine’ fa una query al database per prendere i dati delle prime 10 colonnine (per semplificare), se la query non va a buon fine, restituisce un array vuoto come array delle colonnine con anche un messaggio di errore nella variabile message

**Valori ritornati:** variabile message (stringa vuota in caso di esito positivo, stringa piena in caso di esito negativo) e array contenente i valori delle colonnine (array vuoto in caso di errore)

* **POST** ‘/colonnina’ riceve l’id della colonnina con una fetch fatta dall’interfaccia-progetto.

Utilizza l’id colonnina ricevuto, per fare una query al database per prendere i dati della colonnina in questione

**Valori ritornati:** oggetto con i valori della singola colonnina

* **POST** ‘/prenotazioni/:username’ (:username viene sostituito dall’id dell’utente loggato in quel momento). Controlla se il token dell’utente è valido. Se l’utente ha effettuato il login, viene visualizzata la lista delle prenotazioni da lui effettuate, altrimenti viene restituito il messaggio di errore

**Valori ritornati:** elenco delle prenotazioni dell’utente loggato

* **POST** ‘/prenotazione’ riceve i dati della nuova prenotazione fatta dall’utente

Controlla se il token dell’utente loggato, se è valido, fa una query al database per salvare i dati della prenotazione appena fatta. Se il token dell’utente non è valido, viene visualizzato il messaggio di errore

**Valori ritornati:** nessuno

* **POST** ‘/del-prenotazione’ riceve i dati della prenotazione da eliminare

Controlla se il token dell’utente loggato, se è valido, fa una query al database per eliminare i dati della prenotazione ricevuta. Se il token dell’utente non è valido, viene visualizzato il messaggio di errore

**Valori ritornati:** variabile contenente il messaggio solo se è andata a buon fine l’eliminazione

* **POST** ‘/del-prn-scadute’ route che serve al solo scopo di eliminare periodicamente le prenotazioni scadute. (Viene richiamata solo nel file app.js)

**Valori ritornati:** variabile contenente il messaggio con lo stato dell’eliminazione delle prenotazioni scadute

* **POST** ‘/info-prenotazione’ riceve id, nome, inizio, cerchia, id colonnina e username della prenotazione e visualizza tutte le informazioni della prenotazione, compreso il codice OTP per sbloccare la colonnina

Controlla se il token dell’utente loggato, se è valido, fa una query al database per richiedere i dati della prenotazione richiesta dall’utente. Se il token dell’utente non è valido, viene visualizzato il messaggio di errore

**Valori ritornati:** tutti i valori della singola prenotazione richiesta

***File:*** */archiver-car-charger/mqtt.js*

**Pacchetti Richiesti:** [mqtt.js](https://github.com/mqttjs/MQTT.js#api), [fs](https://nodejs.org/api/fs.html), [path](https://nodejs.org/api/path.html)

**File Richiesti:** daoBBData.js, dao.js

**Variabili**:

**url = 'mqtts://carcharger.uniupo.click'**

**KEY =** fs.readFileSync(path.join(**‘path-to-dir’**. **‘client.key’**))

**CERT =** fs.readFileSync(path.join(**‘path-to-dir’**. **‘client.crt’**))

**TRUSTED\_CA\_LIST =** fs.readFileSync(path.join(**‘path-to-dir’**. **‘ca.crt’**))

**CLIENT\_ID = ‘MQTT\_Archiver’**

Il pacchetto MQTT.js, utilizzato per stabilire e gestire una connessione MQTT, rende disponibili le seguenti funzioni, di cui ho adattato al mio codice:

mqtt.connect(url, {params}) →

**const** **clientEC2** = mqtt.connect(url,{  
 port: **'8883'**,  
 clientId: CLIENT\_ID,  
 key: KEY,  
 cert: CERT,  
 ca: TRUSTED\_CA\_LIST,  
 rejectUnauthorized: false,  
 protocol: **'mqtts'**  
**}) →** Stabilisco una connessione MQTTs su url con port:**'8883'**

clientId: nome del client,  
key: chiave del client,

cert: certificato del client,  
ca: certificato dell’autorità, self-signed

rejectUnauthorized: false perché sto usando un certificato firmato dal gruppo. protocol:**'mqtts'** perché è la versione "TLS-secure" di MQTT

* **clientEC2.on**(**‘connect’**, () => { … })

**Input:** Nessuno

**Descrizione:** Al verificarsi di un evento (in questo caso **‘connect’**) il client esegue quello presente all’interno dell’arrow function.

**Valori ritornati:** Stampa di connessione stabilita con il broker dell’archiver, **SUBSCRIBE** ai topic event/BB/tempoCarica, event/BB/otpSent

* **clientEC2.on**(**‘message’**, (topic, message) => { … })

**Input:** topic, message

**Descizione:** A seconda del valore del topic, elaboro message.

Se topic è /tempoCarica, prendo i dati di message, salvo su DB, cambio stato della colonnina.

Se topic è /otpSent, prendo l’OTP inserito nella colonnina tramite message, verifico la correttezza dell’OTP, **PUBLISH** del risultato.

**Valori ritornati:** Nessuno.

* **setInterval(**() => {…}, 5\*1000**)**

**Input:** Nessuno.

**Descrizione:** Esegue una query per prendere tutti gli stati delle colonnine dal database. Per ogni stato, creo il topic, creo il messaggio, **PUBLISH** sul client, **ogni 5 secondi.**

**Output:** Nessuno.

* **function testJSON(message)**{ … }

**Input:** message

**Descrizione:** funzione per verificare se message è in formato JSON.

**Output:** true semessage è in formato JSON, false altrimenti.

***File:*** */archiver-car-charger/models/daoBBData.js*

**Pacchetti Richiesti:** Nessuno

**File Richiesti:** db.js

**Variabili**: Nessuno

* **exports.IDColonnine** = function() { … }

**Input:** Nessuno

**Descrizione:** Prende gli id delle colonnine presenti nella tabella dati\_colonnine

**Output:** Array contenente tutti gli id delle colonnine

* **exports.invioDatiAgg** = function(...) { … }

**Input:** id\_colonnina, tempoCarica, timestamp

**Descrizione:** Aggiorna la riga di id\_colonnina nella tabella dati\_colonnine con i valori messi in input

**Output:** Nessuno

* **exports.InvioStatoAgg** = function(...) { … }

**Input:** id\_colonnina, stato

**Descrizione:** Aggiorna lo stato della riga di id\_colonnina nella tabella colonnine

**Output:** Nessuno

* **exports.InvioDatiNewCol** = function(...) { … }

**Input:** id\_colonnina, tempoCarica, timestamp

**Descrizione:** Inserisce una nuova colonnina all’interno della tabella datI\_colonnine

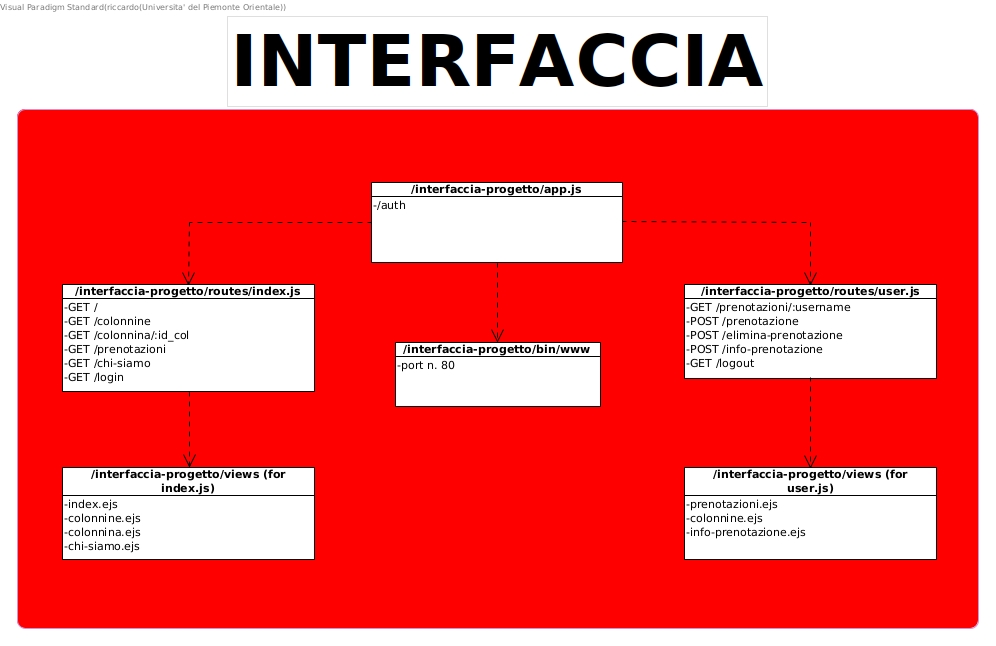
**Output:** Restituisce un messaggio d’errore o un messaggio di inserimento corretto

* **exports.prendiOTP** = function() { … }

**Input:** Nessuno

**Descrizione:** Prende tutti gli OTP presenti nella tabella prenotazioni del DB

**Output:** Array contenente tutti gli id e OTP delle colonnine



**Porta:** 80

***File routes:*** */interfaccia-progetto/routes/index.js*

* **GET** ‘/’ fa una fetch all’ARCHIVER (<http://archiver:3001>) per avere i dati delle colonnine (per semplificare vengono prese solo le prime 10). In caso di errore, manda un array vuoto come array delle colonnine e una stringa message con il messaggio di errore

**Valori mandati al frontend:** array con i valori delle colonnine (in caso di errore array vuoto), stringa message (vuota in caso di esito positivo, piena in caso di errore), eventuale utente loggato

**Pagina visualizzata**: index.ejs

* **GET** ‘/colonnine’ fa una fetch all’ARCHIVER (<http://archiver:3001/colonnine>) per avere i dati delle colonnine (per semplificare vengono prese solo le prime 10). In caso di errore manda un array vuoto come array delle colonnine e una stringa message con il messaggio di errore

**Valori mandati al frontend:** array con i valori delle colonnine (in caso di errore array vuoto), stringa message (vuota in caso di esito positivo, piena in caso di errore), eventuale utente loggato

**Pagina visualizzata:** colonnine.ejs

* **GET** ‘/colonnina/:id\_col’ (:id\_col viene sostituito dall’id colonnina della colonnina che si vuole vedere). Fa una fetch all’ARCHIVER (<http://archiver:3001/colonnina>) mandando l’id della colonnina da visualizzate. Vengono richieste le informazioni relative alla singola colonnina visitata.

**Valori mandati al frontend:** oggetto con i dati della colonnina della quale si è ricevuto l’id da parte dell’interfaccia-progetto, eventuale utente loggato.

**Pagina visualizzata:** colonnina.ejs

* **GET** ‘/prenotazioni’ route prenotazioni che serve solo quando l’utente non è loggato. Rimanda alla pagina di login

**Valori mandati al frontend:** nessuno

**Pagina visualizzata:** pagina di login di Cognito

* **GET** ‘/chi-siamo’ visualizza la pagina con le informazioni del gruppo 15 (nomi, cognomi e relativi indirizzi email)

**Valori mandati al frontend:** eventuale utente loggato per cambiare la barra di navigazione con i menu per l’utente loggato

**Pagina visualizzata:** chi-siamo.ejs

* **GET** ‘/login’ reindirizza alla pagina di AWS Cognito che permette di effettuare il login come utente inserendo i dati (email e password) nel relativo form

Bisogna essere stati registrati su AWS Cognito dalla pagina di amministratore per poter avere un account attivo

(NOTA: Al momento gli account registrati nel servizio AWS Cognito sono due)

**Valori ritornati:** auth code ritornato alla route ‘/auth’. L’auth code viene mandato alla route auth per continuare la procedura di login (parte non visibile siccome la generazione dell’auth code viene fatta dal servizio Cognito di amazon)

**Pagina visualizzata:** pagina di login di Cognito

***File routes:*** */interfaccia-progetto/routes/user.js*

* **GET** ‘/prenotazioni/:username’ (:username viene sostituito dall’id dell’utente loggato in quel momento).

Viene eseguito un controllo per sapere se il valore del token dell’utente loggato è valido. In caso affermativo, viene fatta una fetch all’ARCHIVER (<http://archiver:3001/prenotazioni> /${utente}, dove utente è l’id utente) per avere tutte le prenotazioni di quell’utente. Con la fetch viene inviato il token per effettuare il controllo per l’accesso anche sull’ARCHIVER. In caso negativo, viene restituito un messaggio di errore

In caso di errore della fetch l’array delle prenotazioni viene creato vuoto e passato alla pagina con il relativo messaggio di errore nella variabile message

**Valori mandati al frontend:** elenco di tutte le prenotazioni dell’utente loggato (in caso di errore array vuoto), messaggio di errore (stringa vuota in caso di successo), informazioni utente loggato

**Pagina visualizzata:** prenotazioni.ejs

* **POST** ‘/prenotazione’ riceve dal frontend i dati della prenotazione da creare

Viene creata la stringa con l’ora e i minuti della prenotazione, viene generato il codice OTP che dovrà essere memorizzato insieme ai dati della prenotazione e viene inviato il tutto con una fetch all’ARCHIVER (<http://archiver:3001/prenotazione>). Insieme ai dati della prenotazione viene anche inviato il token dell’utente loggato per la verifica

**Valori mandati al frontend:** nessuno

**Pagina visualizzata:** colonnine.ejs (viene fatto il redirect alla pagina colonnine una volta fatta la prenotazione)

* **POST** ‘/elimina-prenotazione’ riceve dal frontend i dati della prenotazione da eliminare

Fetch all’ARCHIVER (<http://archiver:3001/del-prenotazione>) per inviare i dati della prenotazione da eliminare e il token dell’utente loggato per fare il controllo

**Valori mandati al frontend:** nessuno

**Pagina visualizzata:** info-prenotazioni.ejs (viene fatto il redirect, una volta eliminata la prenotazione, a ‘/prenotazioni/${utente}, dove ${utente} è l’utente loggato)

* **POST** ‘/info-prenotazione’ riceve dal frontend alcuni dati che servono per identificare la prenotazione che si vuole chiedere all’ARCHIVER insieme al token dell’utente loggato per la verifica

Fetch all’ARCHIVER (<http://archiver:3001/info-prenotazione>) con i dati ricevuti dal frontend per avere tutte le informazioni della singola prenotazione

**Valori mandati al frontend:** informazioni complete singola prenotazione richiesta, dati dell’utente loggato

**Pagina visualizzata:** info-prenotazione.ejs

* **GET** ‘/logout’ viene eliminato l’utente dall’elenco degli utenti loggati nella parte del progetto  
  Nella parte di cognito, viene poi richiesto il logout dell’utente dal servizio. Vengono poi eliminati i cookie creati in precedenza

**Valori mandati al frontend:** nessuno

**Pagina visualizzata:** index.js (viene fatto il redirect da Cognito, quindi il reindirizzamento non figura nel codice e non è visibile visto che viene fatto dal servizio di AMAZON)

**6. Validazione del software sviluppato.**

Per validare il software abbiamo utilizzato i dati delle colonnine del **comune di Milano**, reperiti [qui.](https://dati.comune.milano.it/dataset/ds540_colonnine-di-ricarica)

Ci siamo avvalsi del **circuito** presentato al **punto 2 per simulare il veicolo elettrico** (percentuale di batteria e capacità della stessa),parametri modificabili con l’uso del **trimmer apposito.**

Le pinze, invece, si occupano di monitorare il passaggio di corrente.

Una volta collegato il tutto alla corrente, si fa partire il container nella macchina virtuale tramite **docker-compose-up build.**

Dopo una breve attesa, avremo i primi segnali di funzionamento grazie ai **led** presenti nella **breadboard.**

Successivamente, effettuando il **login** (unica modalità in cui è possibile effettuare prenotazioni) con le credenziali di default da inserire su **Cognito** (**admin ; admin123**) possiamo effettivamente iniziare la validazione vera e propria.

Nel nostro esempio abbiamo prenotato la **colonnina 1**, che dopo pochi secondi successivi alla prenotazione porta all’accensione del **led giallo** e allo spegnimento degli altri (**stato PRENOTATA**).

Avviando **l’IDE di ARDUINO** e aprendo il **Serial Monitor** abbiamo la possibilità di **inserire l’OTP** che, se corretto, renderà la colonnina **OCCUPATA** e comporterà l’accensione del **led rosso.**

Tutte le altre funzionalità non richiedono un vero e proprio test dell’utente ma saranno visibili a schermo o in tempo reale (es. tempo rimanente di carica) o attendendo qualche minuto (es. cancellazione prenotazioni scadute).