Informatica - Area scientifica
Dipartimento di Scienze matematiche, informatiche e multimediali
Università di Udine

Progetto di Internet Of Things

ParBrum: il parcheggio a portata di click

Brugnera Matteo (137370)
Parata Loris (144338)

Anno accademico 2020/2021

Indice

Capitolo 1

ParBrum

1.1 Introduzione

Questo progetto di Internet Of Things ha lo scopo di riunire tutte le nozioni apprese durante il percorso della laurea triennale. ParBrum è un progetto che ha intenzione di simulare la gestione di un parcheggio per il pubblico nel modo più smart possibile, senza la necessita per l'utente di doversi registrare su una piattaforma e creare l'ennesimo account.
Di base l'utente effettua una prenotazione tramite una Web-App per una determinata giornata, viene generata una prenotazione contenente un QrCode e l'utente lo utilizzerà per l'identificazione il giorno della prenotazione. Una volta arrivato al parcheggio l'utente andrà a far scansionare il QRCode e il sistema si occuperà di controllare l'effettiva esistenza di quella prenotazione per quel determinato giorno, in caso di esito positivo si aprirà la sbarra che permetterà l'accesso al parcheggio. Quando l'utente ha la necessità di uscire tornerà sulla Web-App contenente le informazioni della propria prenotazione e cliccherà un bottone che effettuerà la richiesta di apertura della sbarra corrispondente all'uscita dell'edificio.
In ogni istante l'utente potrà controllare varie informazioni riguardanti l'ambiente del parcheggio, come temperatura interna, umidità, posti ancora disponibili. Inoltre il parcheggio in modo autonomo controllerà i sensori riguardanti i parametri ambientali ed effettuerà delle variazioni sull'ambiente interno al parcheggio. Per esempio se viene rilevata pioggia attiverà un servomotore che andrà chiudere il tetto del parcheggio.

1.1.1 Architettura del sistema

Il sistema è composto da due elementi:
- software relativo all'applicazione di gestione del sistema, che è ospitao in locale su una unica macchina.
- modello fisico del parcheggio, che abbiamo realizzato in polistirolo, su cui abbiamo installato due Raspberry che si occupano dell'identificazione e rilevazione delle grandezze fisiche ed un microcontrollore l'Arduino che si occupa dell'apertura del tetto.

1.2 Struttura del parcheggio

Il parcheggio è stato progettato utilizzando il software di progettazione **Autodesk Inventor** con l'aiuto dello studente di ingegneria meccanica **Andrea Gattel**, successivamente ritagliato da una macchina industriale che lavora il polistirolo e infine è stato tutto assemblato manualmente.

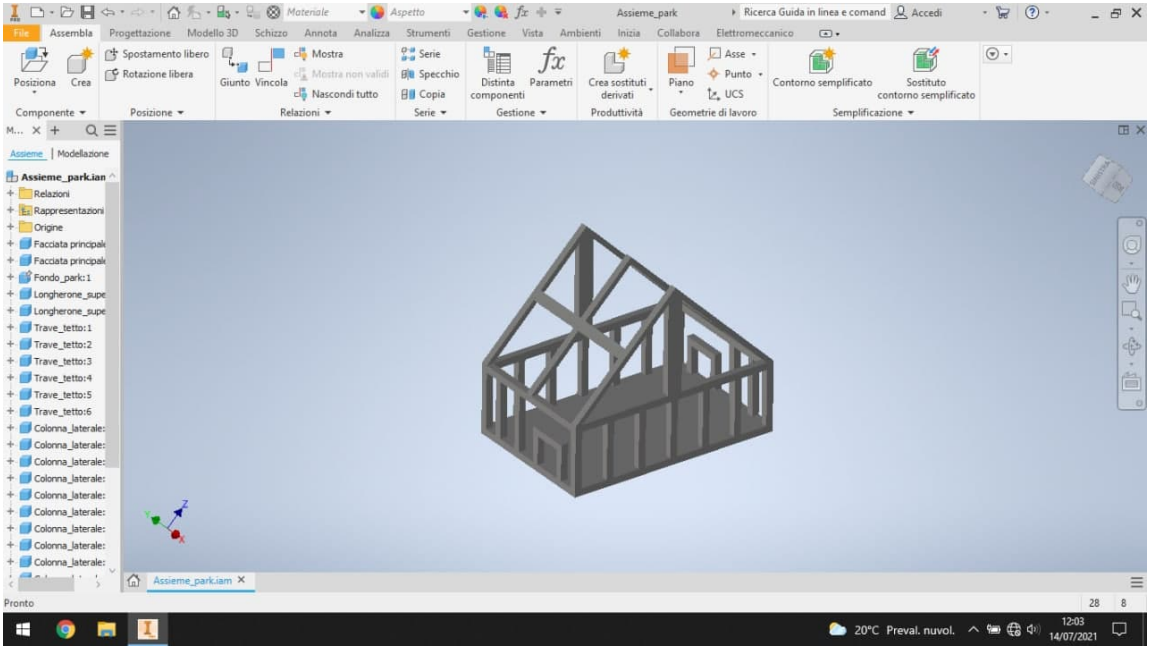

Figura 1.1: Modello vista 1

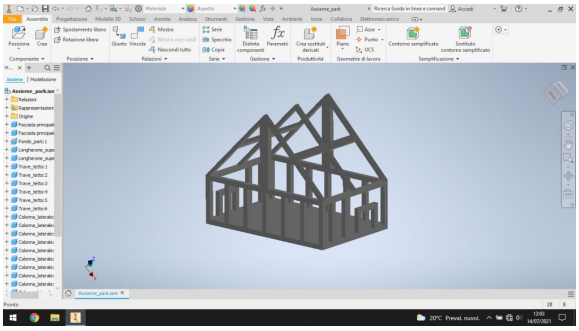

Figura 1.2: Modello vista 2

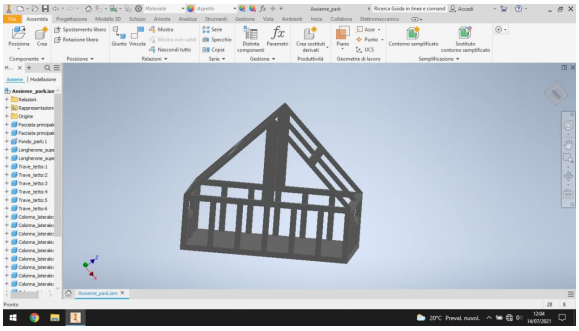

Figura 1.3: Modello vista 3

Il modello è composto da:

- 10 colonne laterali;
- 2 facciate principali;
- 1 pavimentazione;
- 4 traverse del tetto;
- 2 sostegni per il tetto;
- 6 travi del tetto;

Le immagini relative ai piani di progettazione sono allegate alla relazione.

Capitolo 2

Software

2.1 Web Application

La gestione del parcheggio avviene mediante con architettura three tier. Il presentation layer è composto da una applicazione web che si occuperà di gestire gli input dell'utente per le prenotazioni e visualizzare le informazioni riguardanti il parcheggio o lo stato della prenotazione. L'application layer si occupa di fornire tutte le funzionalità necessarie all'interfaccia grafica, mentre il data layer si occupa dell'interazione con la base di dati che conterrà tutte le informazioni relative alle prenotazioni e riguardanti il sistema fisico del parcheggio.
Abbiamo optato per la combinazione di tre tecnologie software molto utilizzate negli ultimi anni:

- **Angular 11** per gestire il frontend;
- **Django** per fornire i servizi necessari mediante delle API REST;
- **Oracle MySQL** come database management system;

2.2 Front End - Angular

Abbiamo deciso di adottare Angular nella sua versione 11, innanzitutto per imparare ad utilizzare un framework innovativo partendo da 0. Inoltre questo framework è sempre più utilizzato dalle case di sviluppo software odierne, incentivate sopratutto dal supporto di Google.
Angular con la sua struttura modulare ed alla sua facile scalabilità permette di sviluppare software di qualità e prestazionale in modo relativamente semplice. Questo framework open source è scritto in Typescript e permette uno sviluppo di applicazioni web su qualsiasi piattaforma e l'utilizzo dei servizi da ogni tipo di device.
In combinazione con Angular di base abbiamo anche integrato il framework di markup Bootstrap 4.0, per rendere l'applicazione anche completamente responsive e gradevole nel suo aspetto.
L'applicazione è strutturata nel seguente modo:

2.2.1 Home page

La schermata principale conterrà alla prima visita due bottoni, uno per visualizzare le informazioni riguardanti l'ambiente del parcheggio ed un altro per poter effettuare la prenotazione.

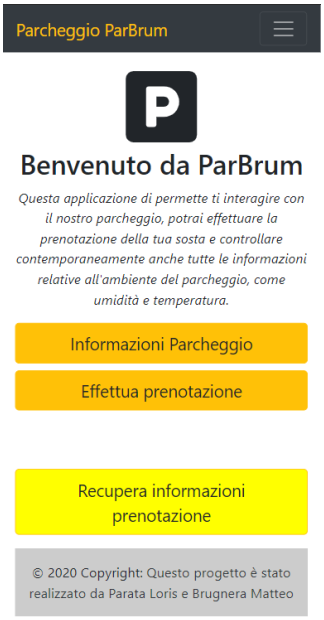

Figura 2.1: Home Page ParBrum

2.2.2 Informazioni del parcheggio

L'utente potrà in ogni istante informarsi sulle condizioni meteo riguardanti il parcheggio, inoltre potrà verificare i valori relativi alla temperatura e all'umidità presente all'interno della struttura. L'utente potrà anche verificare l'eventuale numero di parcheggi liberi in quel determinato istante all'interno del parcheggio.

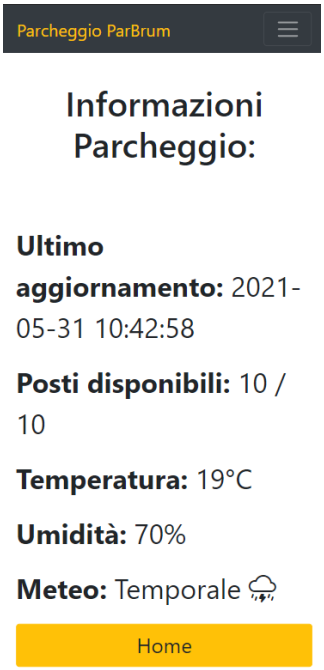

Figura 2.2: Schermata "Informazioni Parcheggio"

Condizioni meteo

Le condizioni meteo non verranno calcolate direttamente dal sistema parcheggio, ma effettuiamo una richiesta ad un API esterna fornita da **OpenWeatherMap** a cui ci siamo registrati. Date le coordinate geografiche ci restituisce tutte le informazioni riguardanti il meteo del luogo richiesto in tempo reale.

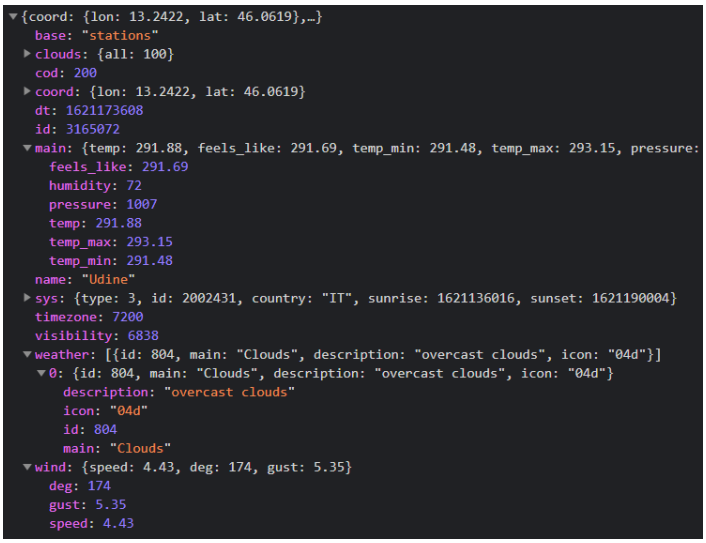

Figura 2.3: Json servizio WeatherMap

2.2.3 Richiesta di prenotazione

L'utente potrà decidere di effettuare una richiesta di prenotazione per una determinata data cliccando sul bottone **Effettua Prenotazione**.
Comparirà una schermata in cui dovrà inserire i dati necessari alla prenotazione, come Nome, Cognome, Email e Data di prenotazione. Una volta effettuata la richiesta il sistema verificherà la disponibilità di un posto nella data selezionata. Nel caso di esito positivo l'utente riceverà una notifica di conferma e verrà reindirizzato su una schermata contenente le informazioni riguardante la prenotazione.

Figura 2.4: Schermata "Richiesta Prenotazione"

Email di Conferma

E' stata implementata anche una funzionalità aggiuntiva che nel momento in cui viene creata una prenotazione si andrà a generare una email contenente un riepilogo dei dati ed il codice QR generato dal sistema, in modo da poterlo utilizzare in alternativa alla sezione dedicata dell'applicazione web.

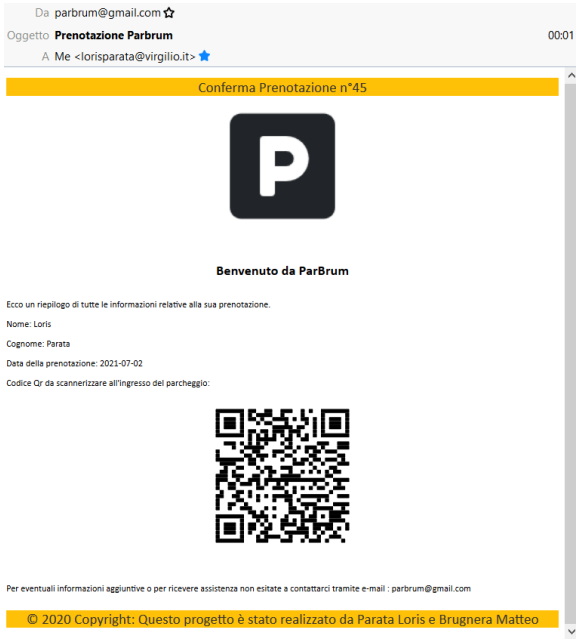

Figura 2.5: Esempio email di conferma della prenotazione

2.2.4 Prenotazione

Questa schermata dell'applicazione conterrà tutte le informazioni riguardanti la prenotazione inseriti durante la fase di Richiesta di prenotazione, ed inoltre conterrà anche un QrCode che permetterà all'utente di accedere fisicamente al parcheggio. Questo QrCode verrà letto da una camera, verificato e successivamente permetterà all'utente di accedere al parcheggio.

Figura 2.6: Schermata "Prenotazione"

Nella schermata è presente anche il bottone che permetterà all'utente di richiedere al sistema di uscire facendo aprire la sbarra.

Recupero della prenotazione

E' stata implementata anche una pagina che permette all'utente di recuperare i dati riguardanti una determinata prenotazione. In questo modo può accedere al servizio anche da un dispositivo differente, questo è possibile mediante un form. Le informazioni richieste sono il numero di prenotazione e la data corrispondente alla stessa, l'applicazione va a verificare i dati e rimanda l'utente alla pagina **Prenotazione** contenente i dati della prenotazione richiesta.

2.3 Back End - Django

Per il backend abbiamo deciso di utilizzare Django, un framework scritto in Python. Esso è strutturato in modo tale da interagire un database MySQL e di fornire i servizi REST necessari al sistema di gestione del parcheggio. Tutti i servizi sono forniti mediante delle chiamate HTTP che restituiscono i dati richiesti sottoforma di JSON. Nel caso si tratti di chiamate che alterano lo stato del parcheggio, come la richiesta di uscire da esso, verranno restituiti dei JSON contenenti l'esito delle richieste.

2.3.1 Models

Il framework Django ci permette di creare dei modelli, paragonabili alle classi, che permettono di modellare il nostro sistema e le entità con cui andremo ad interagire. In base a come vengono definiti i dati, il framework stesso si occuperà di creare le tabelle definite all'interno del codice. Ogni classe corrisponde ad una determinata tabella, di cui definiamo i vari attributi e le loro caratteristiche, le chiavi primarie e le chiavi esterne che definiscono le relazioni tra le tabelle.

```python
from django.db import models

You, seconds ago | 1 author (You)
class Prenotazione(models.Model):
    """ il modello generico di una prenotazione """
    nome = models.CharField(max_length=30)
    cognome = models.CharField(max_length=30)
    data = models.DateField(auto_now_add=False)
    qrCode = models.CharField(max_length=256)

    def __str__(self):
        return self.nome

You, seconds ago | 1 author (You)
class InfoParcheggio(models.Model):
    """modello generico di un oggetto contenente le informazioni del parcheggio """
    data = models.DateTimeField()
    temperatura = models.IntegerField()
    umidita = models.IntegerField()
    meteo = models.CharField(max_length=30)

    def __str__(self):
        return self.data
```

Figura 2.7: Models backend

2.3.2 Views

Questo framework ci permette di creare la logica applicativa dal punto di vista del backend mediante delle viste. Le viste sono un meccanismo del tipo "Request - Response", vengono richiamate tramite un URL specifico stabilito nel file urls.py. Ogni vista permette di attivare delle specifiche funzioni da noi definite e all'interno delle funzioni è possibile anche richiamare funzioni esterne o funzioni definite all'interno della logica del server. Questo sistema ci permette di riutilizzare il codice. Di specifico abbiamo utilizzato le **RedirectView** che ci permettono di interagire con le funzioni mediante delle chiamate HTTP.

```
urlpatterns = [
    #PATH PER LA GESTIONE DELLE API
    path("getPrenotazione/<int:id_Prenotazione>/", par_brum_views.getPrenotazione),
```

Figura 2.8: Urls backend

```
def getPrenotazione(request,id_Prenotazione):        You, 2 days ago • Backend
    try:
        response = checkHeader(request.headers['Authorization'])
    except:
        response= False
    if(response == True):
        prenotazione = Prenotazione.objects.get(pk=id_Prenotazione)
        o = {}
        o['id'] = prenotazione.pk
        o["data"] = str(prenotazione.data)
        o["nome"] = prenotazione.nome
        o["cognome"] = prenotazione.cognome
        o["email"] = prenotazione.email
        o["qrCode"] = prenotazione.qrCode
        response = o
    else:
        response= {'status': 'Unauthorizated'}
    return HttpResponse(json.dumps(response))
```

Figura 2.9: Views backend

QrCode

Il qrCode viene generato utilizzando la libreria **PyQrcode** a cui forniamo alcuni dati relativi alla prenotazione in maniera tale da renderli univoci per ogni prenotazione. Componiamo una stringa utilizzando id, nome, cognome e data della prenotazione. A questa stringa applichiamo una funzione di Hash MD5 in modo da aumentare la confidenzialità dei dati. La stringa hashata verrà utilizzata per generare il QrCode che verrà memorizzato sottoforma di SVG sul server e la sua posizione nel campo apposito QrCode.

```
#genera il codice qr, lo salva sul server e ne restituisce il percorso
def creazione_qrcode(id,nome, cognome, data):
    toHash=f"{id}_{nome}_{cognome}_{data}"
    file_name = hashlib.md5(str.encode(toHash)).hexdigest()
    qrcode = pyqrcode.create(f"{file_name}")
    qrcode.svg("media/"+f"{file_name}.svg", scale=8)
    path = "media/"+f"{file_name}.svg"
    return path

#controlla l'esistenza di un determinato qrCode valido per il giorno stesso della verifica.
def qrCodeChecker(request):
    today = date.today()
    now = today.strftime("%Y-%m-%d")
    data = json.loads(request.body)
    toCheck= "media/"+data['qr_code']+".svg"
    trovato = False
    try:
        trovato = Prenotazione.objects.get(qrCode=toCheck)
    except:
        response = {'status': 'false'}
    if(str(trovato.data) == str(now)):
        response = {'status': 'true'}
    else:
        response = {'status': 'false'}
    return HttpResponse(json.dumps(response))
```

Figura 2.10: QrCode Backend ParBrum

Utilizzeremo la stessa stringa hashata per effettuare la convalida della prenotazione nel momento della lettura del qrCode da parte della camera.

2.3.3 JWT JSON Web Tokens

Il JSON Web Token (JWT) è uno standard open (RFC 7519) che definisce uno schema in formato JSON per lo scambio di informazioni tra vari servizi. Il token generato può essere firmato (con una chiave segreta che solo chi genera il token conosce) tramite l'algoritmo di HMAC, oppure utilizzando una coppia di chiavi (pubblica / privata) utilizzando gli standard RSA o ECDSA. I JWT sono molto utilizzati per autenticare le richieste nei Web Services e nei meccanismi di autenticazione OAuth 2.0 dove il client invia una richiesta di autenticazione al server, il server genera un token firmato e lo restituisce al client che, da quel momento in poi, lo utilizzerà per autenticare le successive richieste.

```
145 def checkHeader(header):
146     try:
147         toCheck = jwt.decode(header, "secret", algorithms="HS256")
148         token_prenotazione = Prenotazione.objects.get(pk= toCheck['id'])
149         if(token_prenotazione):
150             status = True
151     except:
152         print("Errore di autenticazione, operazione negata.")
153         status = False
154     return status
```

Figura 2.11: JWT ParBrum

Noi abbiamo abbiamo implementato questo standard per autenticare le richieste riguardanti la visualizzazione delle informazioni della prenotazione effettuata e nella fase di richiesta di uscita, andando a verificare chi effettivamente effettua la richiesta. L'autenticazione della richiesta avviene analizzando l'header della richiesta HTTP configurato a livello di front-end. Abbiamo installato la libreria **PyJWT** apposita per Python.

Capitolo 3

Hardware

Dal punto di vista hardware abbiamo utilizzato un Raspberry che gestisce il display LCD, i sensori e la mobilità del tetto. Inoltre abbiamo utilizzato anche un altro Raspberry Pi in combo con un microcontrollore Arduino, che si occupano dell'identificazione degli utenti tramite cam Pi con la conseguente gestione degli accessi alla struttura.

3.1 Raspberry e Arduino

L'apertura della sbarra d'accesso al parcheggio è stata effettuata tramite l'utilizzo di un Raspberry Pi in combinazione con un Arduino Uno. Questo microcontrollore è collegato a due led (uno rosso e uno verde), un micro servo sg90 e un buzzer. L'utilizzo della scheda Arduino avviene tramite il protocollo pyFirmata. Infatti, è stato caricato lo sketch StandardFirmata all'interno dell'Arduino, in modo tale da non utilizzare il suo IDE e la programmazione a basso livello della scheda, che è stata sostituita dal linguaggio di programmazione Python. In questo modo non serve più caricare e compilare altri sketch su Arduino.

3.1.1 PyFirmata

Il firmware pyFirmata è un protocollo per la comunicazione su porta seriale fra microcontrollore e computer host. E' possibile sfruttare questa protocollo andando a caricare uno sketch predefinito sul microcontrollore Arduino e di gestire la comunicazione con il dispositivo direttamente ad alto livello mediante delle librerie apposite utilizzabili in Python. Questo pacchetto dà l'accesso completo ai pin presenti su Arduino, con conseguente possibilità di lettura e scrittura di su un qualsiasi pin presente.

```
import sbarra, qr_code
import requests
import multiprocessing
import time
import pyfirmata
from pyfirmata import Arduino
from RPi import GPIO

"""
Inizializzazione variabili per Arduino:
-arduino: stringa che identifica l'Arduino collegato alla porta usb del raspberry;
-pin_*: mi indica in quale pin di Arduino sono collegati i vari led/buzzer/servo
"""
arduino = Arduino("/dev/ttyACM0")
pin_led_verde = 4
pin_led_rosso = 2
pin_buzzer = 6
arduino.digital[pin_led_rosso].write(1)
iteratore = pyfirmata.util.Iterator(arduino)
iteratore.start()
pin_servo = arduino.get_pin('d:9:s')
```

Figura 3.1: Inizializzazione Arduino con PyFirmata

Nella parte di script presente nella figura 3.1 vengono importate le librerie necessarie all'utilizzo dei dispositivi. Inoltre vengono inizializzati tutti gli attuatori, assegnando i vari pin su cui verranno effettuate le scritture dei valori da parte delle funzioni.

3.1.2 Apertura sbarra

```
'''
Apre la sbarra ed attiva il led verde se valore_passato_qrcode è true, altrimenti attiva il buzzer che segnala l'errore
'''
def apertura_sbarra(valore_passato_qrcode, arduino, pin_led_verde, pin_led_rosso, pin_buzzer, pin_servo):

    if valore_passato_qrcode:
        arduino.digital[pin_led_rosso].write(0)
        arduino.digital[pin_led_verde].write(1)
        pin_servo.write(100)
        arduino.pass_time(7)
        pin_servo.write(0)
    else:
        tempo = 0
        while tempo < 800:
            arduino.digital[pin_buzzer].write(1)
            arduino.digital[pin_buzzer].write(0)
            tempo += 1

    arduino.digital[pin_led_verde].write(0)
    arduino.digital[pin_led_rosso].write(1)
```

Figura 3.2: Script apertura della sbarra

Lo script presente in figura 3.2 viene utilizzato per l'apertura della sbarra mediante l'attivazione del servomotore corrispondente per l'ingresso e l'uscita dal parcheggio. La funzione riceve in ingresso un valore booleano, risultato ottenuto dalla scansione e verifica del qrcode. Nel caso di esito positivo, allora viene mandato il comando ad Arduino per aprire la sbarra e richiuderla dopo 7 secondi. Se il riconoscimento fallisce viene attivato il buzzer che emetterà un suono.

Lettura e controllo dati contenuti nel QrCode

```
"""
Attiva la webcam e passa alla funzione di richiesta dell'autenticazione la stringa rilevata
"""
def cattura_webcam():
    codice_identificativo = qr_code.readQrCode()
    check_QrCode_string(codice_identificativo)

"""
Effettua una richiesta HTTP al servizio di autenticazione del backend
e passa il valore ottenuto alla funzione conferma_entrata()
"""
def check_QrCode_string(codice_identificativo):
    url = 'http://172.16.150.165:8000/parcheggio/reciveRasp/'
    payload = {"qr_code": codice_identificativo}
    req = requests.post(url, json=payload)
    response = req.json()
    if(response['status'] == "true"):
        conferma_entrata(True)
    else:
        conferma_entrata(False)
```

Figura 3.3: Cattura del codice QrCode

La funzione **cattura_webcam()** della figura 3.3 richiama la funzione **readQrCode()** che attiva la camera che legge i QrCode e restituisce la stringa letta. Questa stringa viene successivamente passata a **check_QrCode_string()** che invierà una richiesta al servizio responsabile dell'identificazione dell'utente. In base alla risposta del servizio verrà aperta o meno la sbarra.

3.2 Raspberry

Il secondo Raspberry Pi esegue lo script che gestisce le informazioni rilevate dai sensori, le invia alla base di dati e contemporaneamente le visualizza sullo schermo che a rotazione visualizza le informazioni e un messaggio di benvenuto. Inoltre lo script legge le informazioni integrative relative al meteo attuale dalla base di dati ed effettua una eventuale modifica sullo stato di apertura del tetto.

RPi GPIO

E' un pacchetto di librerie che permette di controllare i pin GPIO (general-purpose input/ouput) di Raspberry Pi. Con la conseguente possibilità di controllare moltissimi dispositivi come lcd, servi e sensori.

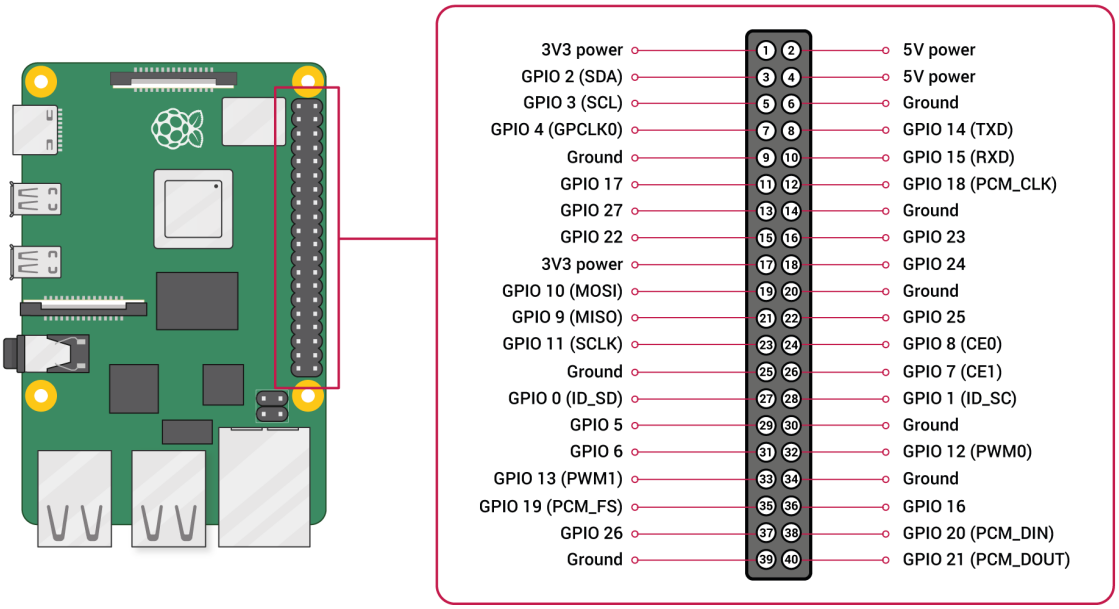

Figura 3.4: Schema pin Raspberry Pi

3.2.1 Calcolo della temperatura

```python
parametri_rilevati = []
"""
Funzione che restituisce la temperatura e l'umidità rilevata dal sensore dht22
param dht: parametro che rappresenta il dht22.
return: ritorna un vettore [temperatura, umidità]
"""
def calcolo_temperatura(dht):
    parametri_rilevati.clear()
    temperatura_gradi = dht.temperature
    umidita = dht.humidity
    parametri_rilevati.append("Temp: {:.1f} C ".format(temperatura_gradi))
    parametri_rilevati.append("Umidita': {}%".format(umidita))
    return parametri_rilevati

"""
Funzione che rileva la temperatura e l'umidità che venogno memorizzati in un JSON
param dht: parametro che rappresenta il dht22
return: json contenente temperatura ed umidità
"""
def value_sensore(dht):
    parametri_rilevati.clear()
    temperatura_gradi = dht.temperature
    umidita = dht.humidity
    response = {"temp": temperatura_gradi, "umidita": umidita}
    return response
```

Figura 3.5: Funzione rilevazione temperatura e umidità

Lo script in figura 3.5 ha il compito di calcolare la temperatura e l'umidità tramite l'utilizzo del sensore dht22 e della libreria adafruit_dht. Sono state formulate due funzioni differenti per fare ciò:

- 1. **calcolo_temperatura()** si crea la stringa che verrà poi utilizzata dal programma incaricato a mostrare a display il risultato;
- 2. **value_sensore()** i dati rilevati dal sensore vengono salvati in formato JSON. Questa funzione verrà richiamata per salvare all'interno del database tutte le rilevazioni effettuate per mantenere uno storico.

3.2.2 Visualizzazione informazioni LCD

```
import calcolo_temperatura
import time, requests, json
from RPi import GPIO
import board
import digitalio
import adafruit_character_lcd.character_lcd as characterlcd
import adafruit_dht

"""
Inizializzazione variabili per sensore DHT22 e LCD1602:
-lcd_*: indica quale GPIO viene assegnato ad ogni elemento del lcd,il quale vie
        in modalità 4 bit.
"""
lcd_columns = 16
lcd_rows = 2
lcd_rs = digitalio.DigitalInOut(board.D26)
lcd_en = digitalio.DigitalInOut(board.D19)
lcd_d4 = digitalio.DigitalInOut(board.D13)
lcd_d5 = digitalio.DigitalInOut(board.D6)
lcd_d6 = digitalio.DigitalInOut(board.D5)
lcd_d7 = digitalio.DigitalInOut(board.D11)

# Inizializzazione della classe lcd
lcd = characterlcd.Character_LCD_Mono(lcd_rs, lcd_en, lcd_d4, lcd_d5, lcd_d6,
                                      lcd_d7, lcd_columns, lcd_rows)

# Inizializzazione della classe dht tramite l'utilizzo della libreria adafruit
dht = adafruit_dht.DHT22(board.D4, use_pulseio=False)
```

Figura 3.6: Import librerie, inizializzazione LCD e DHT22

Nell'immagine 3.6 si può notare come viene importato lo script "**calcolo_temperatura**", che ci permette di ottenere i dati relativi alla temperatura utilizzando lo script analizzato nella figura 3.5. Utilizziamo le librerie **adafruit_character** e **digitalio** per l'utilizzo del display, **adafruit_dht** per l'utilizzo del sensore dht22 e per la gestione dei servomotori utilizziamo la libreria **GPIO**. Nella seconda parte del codice viene inizializzato sia il DHT22 che l'LCD. Per quanto riguarda quest'ultimo, si indicano quante colonne e righe vengono utilizzate e subito dopo si elencano i pin occupati all'interno del Raspberry Pi, utilizzati per inviare i segnali al display.

```
"""
Funzione che mostra sul lcd un messaggio di benvenuto al parcheggio alternato alla temperatura e umidità.
I due valori vengono presi dalla funzione calcolo_temperatura() passando il dht inizializzato precedentemente.
:return:
"""
def display_temperatura():

    try:
        lcd.clear()
        frase_benvenuto = "Benvenuto al\n" + "ParBrum!"
        lcd.message = frase_benvenuto
        time.sleep(4)
        risultato_finale = calcolo_temperatura.calcolo_temperatura(dht)
        lcd.message = risultato_finale[0] + "\n" + risultato_finale[1]
        time.sleep(4)
    except Exception as e:
        print(e)
```

Figura 3.7: Funzione che visualizza le informazioni sul display

Nella figura 3.7 è presente la funzione **display_temperatura()**, che visualizza sullo schermo lcd un messaggio di benvenuto al parcheggio, seguito da una seconda schermata contenente tutte le informazione legate alla temperatura e all'umidità rilevante nel parcheggio.

3.2.3 Controllo del meteo, temperatura e umidità

```
"""
Funzione che richiede al sito informazioni inerenti al meteo. Qualora esse siano -temporale-, -pioggia- oppure
la temperatura è maggiore di 25 gradi, allora il tetto viene chiuso, altrimenti il tetto viene aperto.
"""
def getMeteo():
    url = 'http://172.16.150.165:8000/parcheggio/getInfoMeteoRasp'
    req = requests.get(url)
    response = req.json()
    if (response['meteo'] == "Thunderstorm" or response['meteo'] == "Rain" or int(response['temperatura']) > 25):
        chiudi_tetto()
    else:
        apri_tetto()

"""
Funzione che inserisce all'interno del database informazioni inerenti alla temperatura e all'umidità.
"""
def setTempAndHum():
    sensore = calcolo_temperatura.value_sensore(dht)
    temperatura = sensore["temp"]
    umidita = sensore["umidita"]
    payload = {"temperatura": temperatura, "umidita": umidita}
    url = 'http://172.16.150.165:8000/parcheggio/insertInfoParking/'
    req = requests.post(url, json=payload)
```

Figura 3.8: Funzione che verifica le condizioni meteo

La funzione **getMeteo()** permette di reperire le informazioni legate alle condizioni meteo direttamente dalla base di dati del sistema parcheggio effettuando una richiesta all'API corrispondente.

Inserimento informazioni sensori nel database

La seconda funzione in figura 3.8 **setTempAndHum()** permette di inserire all'interno del database le informazioni relative al temperatura e umidità rilevate dal raspberry. La funzione invia in formato JSON le informazioni mediante una richiesta http del tipo POST ad una API apposita del sistema parcheggio, in modo tale da creare uno storico. In particolare, qualora il contenuto del JSON restituito nel campo [**'meteo'**] risulti essere uguale a 'thunderstorm', 'rain' oppure la temperatura è maggiore di 25 gradi, allora viene chiamata la funzione **chiudi_tetto()**, in caso contrario viene chiamata la procedura **apri_tetto()**.

3.2.4 Apertura e chiusura del tetto

```
"""
Funzione che apre il tetto del parcheggio utilizzando due servi tramite la libreria GPIO.
"""
def apri_tetto():
    servoPIN = 10 #pin utilizzato dal primo servo
    servoPIN2 = 9 #pin utilizzato dal secondo servo

    GPIO.setmode(GPIO.BCM)
    GPIO.setup(servoPIN, GPIO.OUT)
    GPIO.setup(servoPIN2, GPIO.OUT)
    p = GPIO.PWM(servoPIN, 50)  # GPIO 10 PWM con 50Hz
    p2 = GPIO.PWM(servoPIN2, 50)# GPIO9 PWM con 50Hz

    #Inizializzazione
    p.start(2.5)
    p2.start(2.5)

    #Movimento servi per aprire
    p.ChangeDutyCycle(7.5)
    p2.ChangeDutyCycle(7.5)
    time.sleep(2)

    #Interruzione movimento servi
    p.stop()
    p2.stop()
```

Figura 3.9: Funzione apertura tetto

```
"""
Funzione che apre il tetto del parcheggio utilizzando due servi tramite la libreria GPIO.
"""
def chiudi_tetto():
    servoPIN = 10 #pin utilizzato dal primo servo
    servoPIN2 = 9 #pin utilizzato dal secondo servo

    GPIO.setmode(GPIO.BCM)
    GPIO.setup(servoPIN, GPIO.OUT)
    GPIO.setup(servoPIN2, GPIO.OUT)
    p = GPIO.PWM(servoPIN, 50)  # GPIO 10 PWM con 50Hz
    p2 = GPIO.PWM(servoPIN2, 50)# GPIO 9 PWM con 50Hz

    #Inizializzazione
    p.start(2.5)
    p2.start(2.5)

    #Movimento servi per chiudere
    p.ChangeDutyCycle(12.5)
    p2.ChangeDutyCycle(12.5)
    time.sleep(2)

    #Interruzione movimento servomotori
    p.stop()
    p2.stop()
```

Figura 3.10: Funzione chiusura tetto

Le due funzioni che troviamo nella figura 3.9 e 3.10 servono per aprire e chiudere il tetto del parcheggio. Inizialmente vengono indicati i pin di riferimento per l'utilizzo dei due servo motori. Dopo aver fatto il setup(), a cui vengono passati i pin di input e output per inizializzarli, viene utilizzata la funzione **ChangeDutyCycle()**, a cui viene passato l'impulso adatto per ottenere l'angolo di rotazione

desiderato. Questo permette di effettuare i movimenti riguardanti l'apertura e la chiusura del tetto. Al termine della funzione, i due servo motori vengono fermati grazie alla chiamata **stop()**.

3.2.5 Main

Tutte le funzioni descritte precedentemente vengono richiamate in un ciclo while, che presenta all'interno un tempo di attesa arbitrario, per esempio 60 secondi, in modo da eseguirle ciclicamente.

3.2.6 Uscita dal parcheggio

L'uscita del parcheggio è effettuata mediante un piccolo Web server,in esecuzione sul Raspberry collegato alla sbarra d'ingresso, che mette a disposizione una API REST. Quando viene richiamato il servizio viene eseguita la funzione **apertura_sbarra()** della figura 3.2. Questo servizio viene richiamato dall'applicazione lato client.

```python
from flask import Flask
import sbarra, json, pyfirmata
from pyfirmata import Arduino
"""
Inizializzazione variabili per Arduino:
-arduino: stringa che identifica l'Arduino collegato alla porta usb del raspberry;
-pin_*: mi indica in quale pin di Arduino sono collegati i vari led/buzzer/servo
"""
arduino = Arduino("/dev/ttyACM0")
pin_led_verde = 4
pin_led_rosso = 2
pin_buzzer = 6
arduino.digital[pin_led_rosso].write(1)
iteratore = pyfirmata.util.Iterator(arduino)
iteratore.start()
pin_servo = arduino.get_pin('d:9:s')

def conferma_uscita(booleano):
    """
    Funzione che viene usata nel momento in cui il valore di entrata risulta
    essere TRUE/FALSE. Andrà quindi ad eseguire l'apertura della sbarra su arduino, oppure ritornerà errore.
    :param booleano: valore TRUE/FALSE
    """
    sbarra.apertura_sbarra(booleano, arduino, pin_led_verde, pin_led_rosso, pin_buzzer, pin_servo)

app = Flask(__name__)
@app.route('/')
def index():
    conferma_uscita(True)
    status = {
        "status":"True"}
    return json.dumps(status)

if __name__ == '__main__':
    app.run(debug=False, port=80, host='172.16.151.203')
```

Figura 3.11: Web server che gestisce le richieste di uscita dal parcheggio

Capitolo 4

Conclusioni

4.1 Possibili sviluppi futuri

Questo prototipo potrebbe essere sviluppato creando una rete di parcheggi appartenenti ad uno stesso ente, implementando di conseguenza una gestione dei vari parcheggi a livello di software amministrativo ed implementare la gestione degli utenti, in maniera tale da permettere anche uno storico delle varie prenotazioni effettuate nei vari parcheggi nel tempo.
Si potrebbe anche implementare un sistema di riconoscimento delle targhe che permetterebbe all'utente di evitare la scansione del codiceQR, mantenendolo soltanto come sistema alternativo di autenticazione.
Nel caso in cui un'azienda volesse implementare questo sistema, rendendolo un servizio privato per gli utenti, sarebbe necessario implementare la gestione dei pagamenti in base al tempo trascorso all'interno del parcheggio.

4.2 Informazioni conclusive

Alleghiamo alla relazione del progetto tutto il codice scritto per la realizzazione e un video che ne visualizza il funzionamento completo.