RELAZIONE PISSIR MATTEO GEMME 20034251

SPECIFICHE FUNZIONALI

L’obbiettivo del progetto è quello gestire i diversi parcheggi all’interno di una città, nel progetto sono presenti diversi parcheggi gestiti da sensori e dispositivi IOT (Entrata, Uscita e Pagamento).

vengono simulati possibili ingressi, uscite o pagamenti tramite sensori, e vengono inviati messaggi MQTT attraverso un broker sui topic adeguati.

Vi sono poi la figura dell’utente che può visionare i parcheggi sul interfaccia web e un amministratore che può invece compiere azioni di modifica agigunta o eliminazione sul databse da pagina web.

ANALISI DELLA TECNOLOGIA

Per lo sviluppo dei dispositivi IOT e gestione dei sensori ho utilizzato un protocollo MQTT molto utilizzato, un protocollo leggero orientato ai Message Brokers e basato sul modello publish-subscribe, E’ adatto a far comunicare dispositivi IoT.

Permette infatti ai client di sottoscriversi a certi topic, che possiamo considerare come argomenti di interesse, e ad altri di pubblicare su di essi.L’idea è quella di una bacheca dove si possono pubblicare informazioni e iscriversi solo a quelle di interesse. Così facendo i client hanno modo di ricevere solo i messaggi relativi ai topic di interesse. Tutto ciò è reso possibile dal Message Broker, che filtra e distribuisce i messaggi, rendendo quindi possibile ai client sottoscritti a certi topic di ricevere i messaggi corretti.

Nella realizzazione del progetto ho creato anche un interfaccia web in cui vi sono due tipi di utente, quello base che può visionare dati e stato di un parcheggio e un amministratore.

L’amministratore dopo aver effettuato una autenticazione via GitHub con protocollo oauth2, consente di accedere a informazioni riservate e controllo dei parcheggi.

dopo aver effettuato il login verrà assegnato un authentication code e dopo l’invio del client id, client secrete e authentication code al sito github, si otterrà un token per l’accesso da utilizzare per entrare nelle aree protette del mio sito web.

Un utente loggato può gestire la modifica aggiunta o l’eliminazione di un parcheggio. Le modifiche del DB attraverso l’interfaccia web si basano sull’architettura REST uno stile architetturale per sistemi distribuiti in rete che non è uno standard ma si basa su più standard.

L’obiettivo è quello di creare un’architettura per la comunicazione tra client e server che sia semplice ed efficiente. I concetti chiave di questo stile architetturale sono: i nomi , ovvero gli URI con cui si identificano le risorse (es. Localhost:3000/amministratore), poi i verbi, ovvero i metodi GET POST PUT DELETE e PATCH, ed infine le rappresentazioni, ovvero linguaggi come XML, JSON. Ho utilizzato inizialmente il protocollo HTTP, passando succesivamente ad HTTPS avvalendomi di un certificato creato con openssl.

Un certificato TLS è un certificato digitale che autentica l’identità di un sito web e consente di instaurare una connessione crittografata tra client e server, un canale sicuro dove scambiarsi informazioni che mantenga integrità e confidenzialità. Il certificato viene utilizzato per il processo di handshaking TLS.

Per quanto riguarda il database ho usato DB Browser for SQLite. Il file per il database si trova in src/main/database/parcheggi.db.

Sono presenti 2 tabelle: una, chiamata Parcheggi, per memorizzare tutti i parcheggi che vengono creati dagli amministratori con gli opportuni attributi e un’altra, chiamata ticket, che serve per uscire da un parcheggio se lo stato del ticket è pagato(ossia = 1), si memorizza l’id del ticket generato casualmente quando si accede a un parcheggio, l’id del parcheggio e lo stato del ticket .

***SCELTA DELL’APPROCCIO***

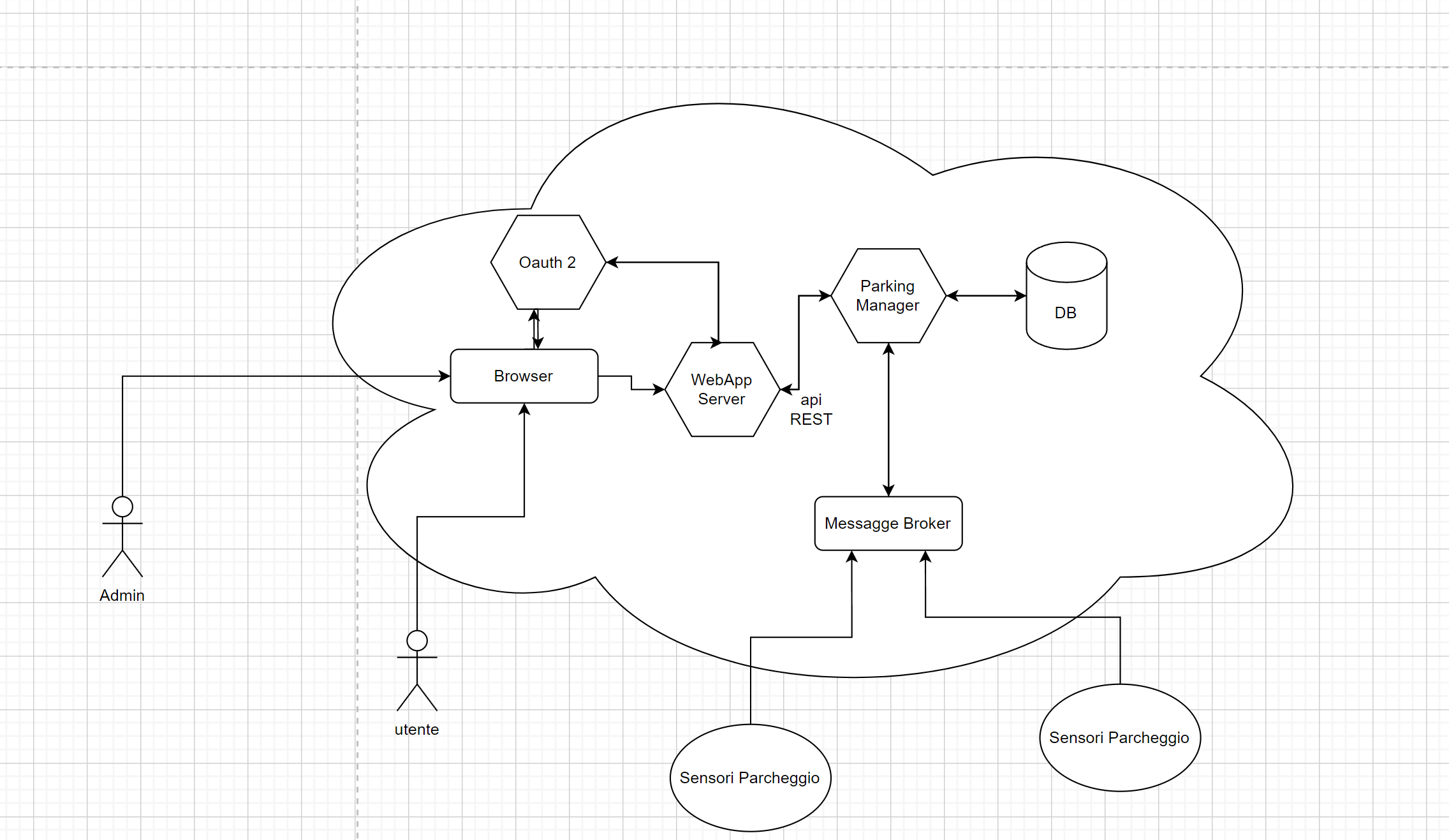
Nella realizzazione del progetto era necessario l’utilizzo di un message broker come mosquitto o rabbitMQ, nella soluzione ho utilizzato MQTT e message broker mosquitto in quanto era più coerente alle linee guida del progetto.

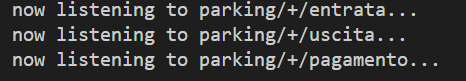
E’ dunque compito del Message Broker filtrare e distribuire i messaggi, garantendo affidabilità ed efficienza, in questo caso Mosquitto è un protocollo molto leggere diretto alle tecnologie IOT e efficiente per dispositivi con poca potenza come il nostro.

Ho creato un client globale che rimanga in ascolto sottoscritto a tutti e 3 i topic di interesse entrata uscita e pagamento, d’altra parte dei singoli client per ogni parcheggio disponibile che pubblichino sui topic di interesse Infatti, in modo più specifico quando un utente simula l’ingresso in un parcheggio attraverso Sensori parcheggio, l’MqttClient relativo a quel parcheggio con il suo specifico id pubblica il messaggio in base al bottone cliccato sul topic es“parking/1/entrata” . Questo verrà quindi intercettato dal client sottoscritto al topic che capisce quindi che deve fare aprire le transenne del parcheggio, e modifica nel database quel parcheggio, così da tenere sotto controllo il numero dei parcheggi disponibili.

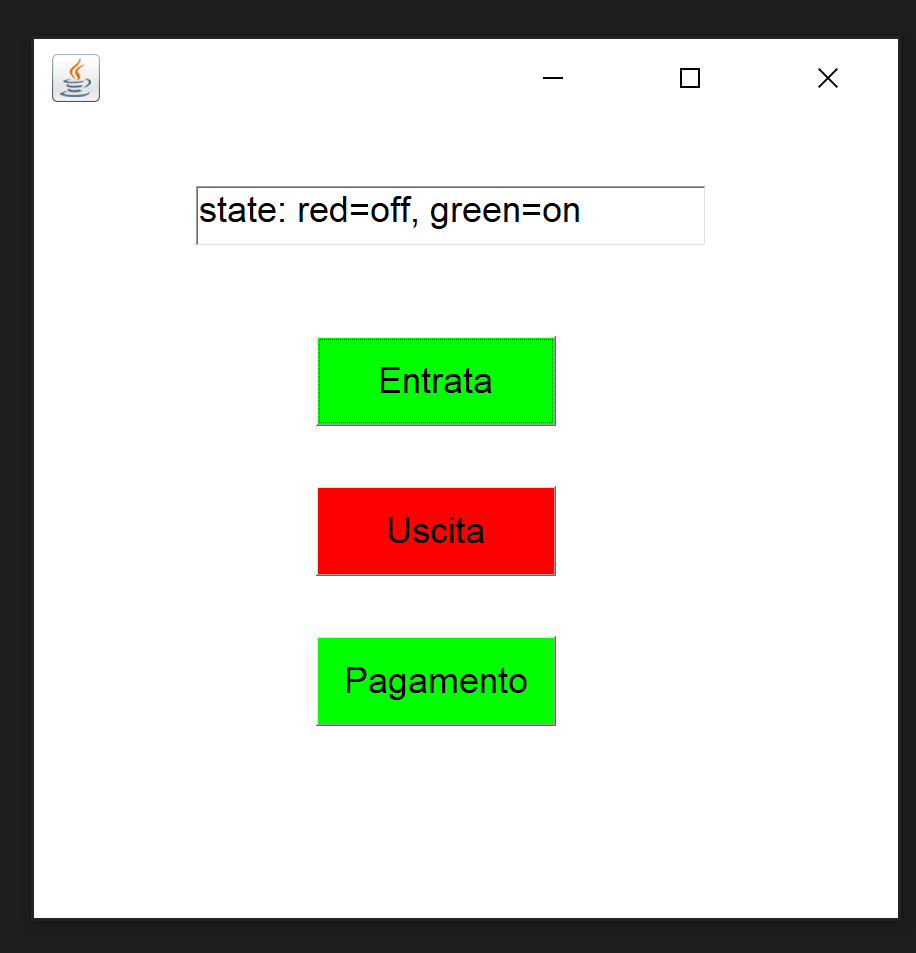
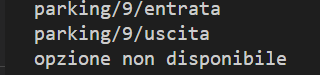
Inoltre all’ingresso del parcheggio verrebbe generato un ticket che dopo essere stato pagato attraverso il bottone specifico per quel topic, abilità l’uscita di una macchina da quel parcheggio, a quel punto si potrà pagare il ticket inviando messaggio sul topic parking/1/pagamento, questo permetterà alle transenne di aprirsi e far uscire una macchina dal parcheggio pubblicando sul topic parking/1/uscita. Questo è il ciclo completo di una macchina all’interno di un parcheggio, ipotizzando che abbia posti disponibili e lo stato del parcheggio sia aperto.

***Architettura del Software***



-IOT Sensori Parcheggio: Ogni parcheggio è gestito tramite tre devices IoT (Ingresso, Uscita, Cassa), e il Parking manager (gestore parcheggio) che rimarrà in ascolto su tutti e 3 i topic e invierà messaggi mqtt al topi parking/id/log per conferma avvenuta modifica nel db

La simulazione di questi dispositivi è stata realizzata attraverso l'uso del frame aEventsense(top-level window con titolo e bord), nel frame sono presenti 3 bottoni per le operazioni nel parcheggio. i bottoni sono disabilitati se il background è rosso verrà stampato un messaggio di errore, opzione non disponibile. Il controllo delle funzionalità dei bottoni è gestito da una funzione GestisceAE() che controlla ad ogni operazione se vi ancora sono posti disponibili per l'entrata nel parcheggio, altrimenti se il ticket è pagato per abilitare il bottone per uscita.

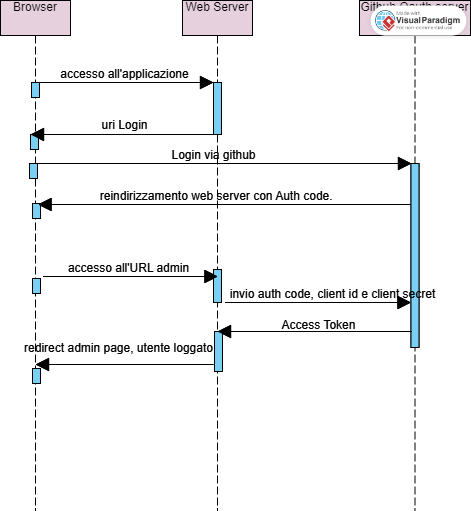
- Messagge Broker Mosquitto: è l’intermediario tra i sensori dei parcheggi e il parking manager che andrà ad effettuare le operazioni nel DB. questo broker permette la comunicazione affidabile e asincrona tra i dispositivi IOT attraverso il protocollo MQTT. I dispositivi IOT sensori, si connettono al broker e scambiano messaggi per inviaire informazioni a proposito dell’entrata, uscita di una macchina da un parcheggio o al pagamento di un ticket.

-Database: vengono gestite e salvate le informazioni relative ai parcheggi sul db attraverso parking manager che andrà a richiamare le funzioni nella classe DB in base ai topic dei messaggi mqtt in arrivo. funzione db.insertUpdateparcheggio() e ugualmente per pagamento e l’uscita con le rispettive funzioni pagaParcheggio() e UscitaParcheggio().

-WebApp: La webapp permette agli utenti base di visualizzare una tabella dei parcheggi avendo a disposizione le informazioni sul nome, stato(aperto, chiuso) e info sul numero posti disponibili

E gli Amministratori possono invece registrare Nuovi parcheggi, sospendere il servizio di un parceggio, attivare il servizio e disattivare permanentemente un parcheggio esistente. Un utente può accedere come amministratore soltando dopo essersi loggato attraverso Oauth2, nel mio caso utilizzando github.

La webApp utilizza connessioni SSL e HTTPS grazie ad un certificato per garantire la sicurezza delle comunicazioni, sulla porta 3000.

-Oauth2: 

***DESCRIZIONE DELLE CLASSI***

Per gestire meglio la struttura del progetto mi sono avvalso dell’uso di Spring Boot.

Java Spring Boot è uno strumento che semplifica e velocizza lo sviluppo di applicazioni web e microservizi con Spring Framework tramite diverse funzionalità come: Configurazione automatica ossia applicazione viene inizializzata con dipendenze preimpostate che non hai bisogno di configurare manualmente

E in più consente di creare applicazioni autonome che vengono eseguite autonomamente integrando un server web

Classi presenti nel package spring.httpparking sono httpparking application che viene generata automaticamente annotata con @springbootapplication così che il framework la rilevi come classe principale

securityconfig presente nel package config, è classe di configurazione di springboot con l’annotazione @Configuration

Package controller, sono i controller di springboot per la gestione di richieste http dalla webapp sono annotate con @controller e gestiscono con l’annotazione @request mapping l’url per raggiungere la pagina html.

I controller sono homecontroller per aprire la pagine index e amministratoreController per aprire la pagina amministratore.html (dopo essersi autenticati)

Poi vi sono due controller API per gestire le richieste http e le risposte http in formato JSON attraverso gestorePercheggi.

utenteAPI serve per gestire la get dei parcheggi attraverso il path /user/api/parkings che viene richiamato dentro una funzione create nell’index.html in javascript fetchParcheggi. Otteniamo così i parcheggi da mostrare sulla pagina html in modo dinamico

amministratoreAPI invece gestisce sia richieste di get “getAll()” che anche per i metodi delete “rimuvoiParcheggio()” Post “creaParchegio()” e put “UpdateParcheggio” tutti con path admin/api/parkings/{id} (eventuale id del parcheggio) per la modifica o l’eliminazione.

É presente poi il package database contente il database parcheggi.db e classe Database con tutte le funzioni per gestire le query nel database select insert update che sono gestite da un unica funzione db.insertUpdateparcheggio() che rileva se l’id è =-1 quindi non è ancora presente quell’oggetto nel db allora lo inserisce, altrimente avviene una modifica dell’oggetto parcheggio già creato precendentemente e infine delete. Le stesse funzioni sono presenti anche per i ticket.

Inoltre vi sono i package Sensori e manager, sensori è package che contiente Sensori parcheggio, che simula attraverso l'uso del frame aEventsensel'entrata e uscita o il pagamento di un ticket.

Il package manager contiene invece la classe gestorePercheggio che viene utilizzata per rimanere in ascolto su tutti i topic di interesse e per gestire tutte le funzioni utili nel parcheggio.

attraverso controllo del topic se il messaggio è in arrivo nel topic entrata allora per l’entrata in un parcheggio verrà richiamata la funzione entrataParcheggio() che attiverà il thread iotentrata per inviare un messaggio di conferma e verrà poi inserito il parcheggio nel db utilizzando la funzione db.insertUpdateparcheggio() e ugualmente per pagamento e l’uscita con le rispettive funzioni pagaParcheggio() e UscitaParcheggio().

inoltre vi è una funzione per gestire la creazione di un parcheggio generato da un amministratore, transformando l’oggetto json in un parcheggio

***Validazione***

Si basa dunque su una web app e dei sensori che simulano alcune azioni come apertura o chiusare delle transenne di un parcheggio. Per validare la soluzione si deve far partire HttpParkingApplication che aprirà L’interfaccia utente che troviamo alla porta <https://localhost:3000>.

l’utente che può solo visionare i vari parcheggi poi sempre nella pagina di home si può accedere alla pagina amministratori /amministratori Dove loggandosi con l’account github(le credenziali sono a piè di pagina) si può eliminare modificare e aggiungere un parcheggi.

Per poter gestire e attivare i sensori bisognerà andare nel package sensori e far partire la classe sensori parcheggio che attiverà un interfaccia con i 3 bottoni per ogni topic per gestire entrata uscita e pagamento.

Infine servirà far partire la classe gestore parcheggio nel package manager che rimarrà in ascolto sui topic e gestirà le funzioni nel database.