

Relazione sul Progetto: Servizio di Parcheggio

Specifiche Funzionali

L'obiettivo del progetto era quello di sviluppare un servizio per simulare la gestione di diversi parcheggi all'interno di un comune. In ogni parcheggio vengono simulati possibili ingressi, uscite o pagamenti tramite input degli utenti nella webApp e vengono inviati messaggi MQTT sui topic adeguati.

In particolare ci sono due classi di funzionalità coinvolte:

1. Quelle di controllo da assolvere localmente tramite sensori. Infatti nel mio sistema simulo la presenza di sensori in corrispondenza delle transenne dei parcheggi e della cassa.
2. L'accessibilità via Internet del sistema, infatti affinché il sistema funzioni correttamente è necessario essere connessi a una rete Internet.

Analisi della Tecnologia

Nello sviluppo di sistemi intelligenti, dove è necessario riuscire a mettere in comunicazione diversi dispositivi IoT, un protocollo parecchio usato per le sue caratteristiche è MQTT. E' un protocollo molto leggero che con l'ausilio di un server particolare (che ha il compito di gestire solo i messaggi) chiamato message broker, può gestire e monitorare in maniera asincrona ciò che accade nei parcheggi. In particolare usando il modello Publish-Subscribe, le 3 tipologie di dispositivi IoT (entrata, uscita, cassa) pubblicano messaggi MQTT sui topic adeguati che hanno questa struttura: *parking/idParcheggio/entrata*, *parking/idParcheggio/uscita* e *parking/idParcheggio/cassa*.

Dunque è necessario considerare le caratteristiche del traffico che il sistema dovrà gestire e i vincoli di tempo reale. Ad esempio, il sistema deve essere in grado di gestire simultaneamente una quantità significativa di messaggi MQTT generati dai dispositivi IoT; per garantire una latenza ridotta e una connessione stabile tra i componenti ho ripiegato sull'utilizzo di thread generati quando è necessario l'invio di un messaggio MQTT. Inoltre, è importante considerare i tempi di risposta desiderati per le azioni di controllo, come l'apertura o la chiusura delle barriere dei parcheggi, che devono essere eseguite in modo rapido e senza ritardi significativi: questo è stato solo simulato da stampe su terminale per confermare l'avvenuta ricezione del messaggio MQTT.

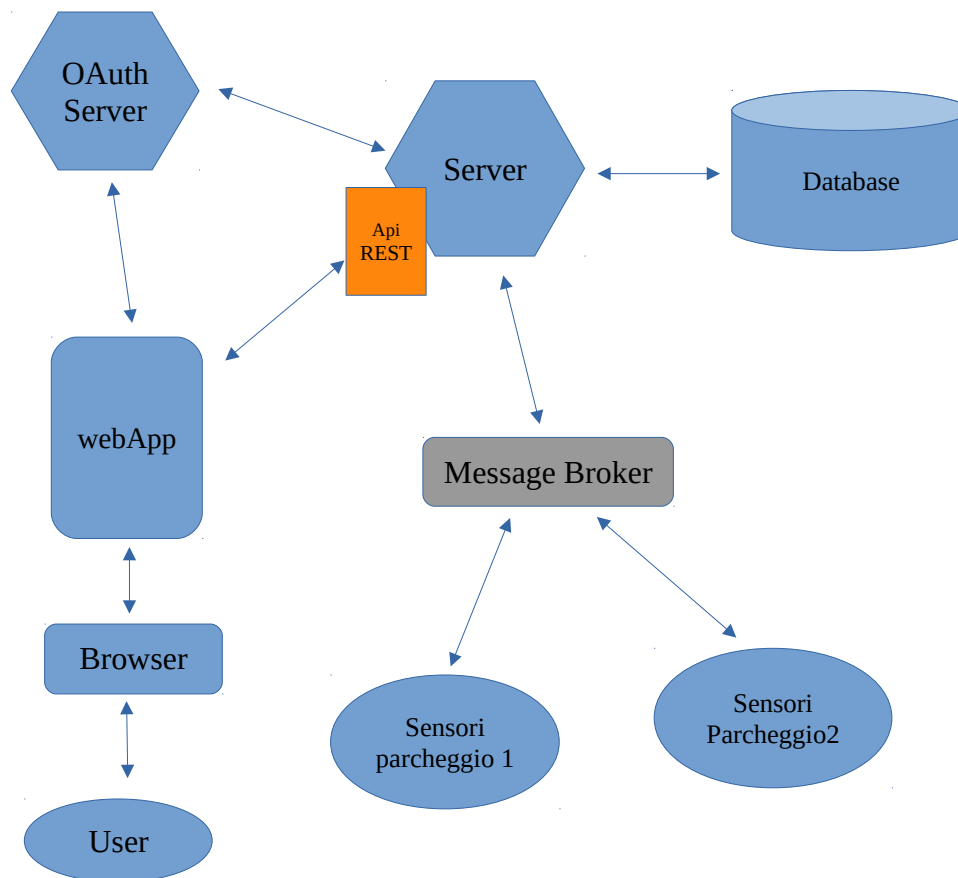
Trattandosi di un progetto a scopo educativo, non ho utilizzato dispositivi fisici o parcheggi reali ma ho realizzato una webApp che tramite l'input dell'utente può rappresentare ciò che accadrebbe nella realtà. Per fare ciò ho usato il protocollo stateless HTTP, in particolare con l'architettura REST e ho usato i verbi HTTP GET, POST, PUT, DELETE per fare operazioni sulle pagine. Il framework Spring Boot mi ha aiutato per la gestione degli url e delle pagine web: nel mio progetto ho creato 2 package (com.parking.demo.controller e com.parking.demo.controller.api) che contengono le classi java dedicate ai @Controller dell'applicazione; qui ho implementato il punto di contatto con le funzioni Javascript nelle mie pagine HTML.

Affinchè fosse sicuro navigare sulla webApp ho creato un certificato per poter navigare con HTTPS sulla porta 8443 (il certificato si chiama parkings.p12 e si trova in *src/main/resources/keystore/parkings.p12*).

Nella webApp era richiesto di implementare 2 tipi di utenti: l'utente comune che può visualizzare lo stato dei parcheggi con le informazioni relative e l'amministratore che deve essere autenticato per poter svolgere funzioni di modifica o cancellazione. L'amministratore si può autenticare con Facebook usando il protocollo OAuth2.0: questo consente di effettuare l'accesso utilizzando le proprie credenziali da un provider di identità esterno, cioè Facebook.

Scelta dell'Approccio

Per la realizzazione di questo sistema era necessario utilizzare un Message Broker. A lezione abbiamo analizzato diversi message broker come Mosquitto o RabbitMQ. Ho scelto di usare il protocollo MQTT e il message broker Mosquitto in quanto era più adatto al progetto commissionato dal professore: è un protocollo molto leggero e più diretto all'IoT e a dispositivi con poca potenza come, nel nostro caso, i sensori nei parcheggi. Un punto che era richiesto nelle specifiche del progetto e che mi ha spinto ad usare Mosquitto sta nel fatto che RabbitMQ, al contrario di Mosquitto, non supporta la crittatura dei messaggi con TLS. Personalmente, ho iniziato a sviluppare il progetto usando come message broker la mia macchina locale e la porta 1883; dopo, per implementare la crittatura dei messaggi, ho deciso di usare il broker libero e gratuito *test.mosquitto.org* con la porta 8883 in quanto era anche necessario disporre di un message broker in cloud (per usare la porta 8883 ho dovuto scaricare un certificato dal sito di mosquitto). Inoltre dato che l'unico compito che dovevano svolgere i device IoT era quello di mandare messaggi, ho pensato di dover simulare device con una potenza molto bassa: questo significa che Mosquitto era più che sufficiente; invece con RabbitMQ avrei avuto più funzionalità come gestione di code di messaggi, ma allo stesso tempo avrei dovuto simulare dispositivi più potenti. In generale l'uso di un message broker permette di avere un server dedicato esclusivamente per distribuire i messaggi, occupandosi lui stesso di definire le strutture necessarie e le rotte di instradamento opportune.



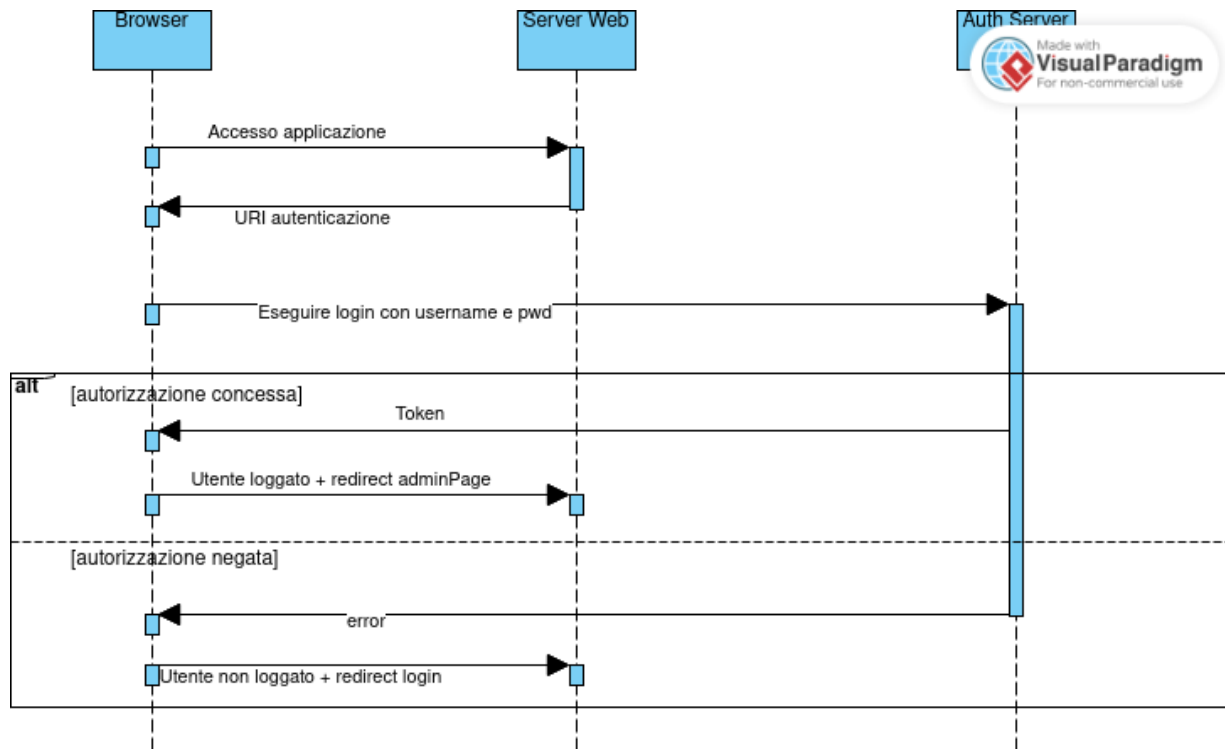
Descrivo i componenti:

1 – User: l'utente rappresenta il cliente che accede al servizio di parcheggi pubblici. E' il punto di accesso principale al sistema attraverso un browser web.

2 – webApp: è l'interfaccia utente scritta in HTML e JavaScript che l'utente utilizza per interagire con il servizio di parcheggio. Questa invia richieste HTTP al server Tomcat integrato nel framework

Spring. Ora il server riceve la richiesta HTTP, la elabora e genera una risposta appropriata che viene inoltrata alla webApp e mostrata graficamente.

3 – Server OAuth: è il server che gestisce il processo di autenticazione degli utenti tramite Facebook. Si occupa di rilasciare il token di accesso affinché l'utente possa accedere alle risorse protette.



4 – Server: La classe ParkingService riveste un ruolo fondamentale all'interno del server. Questa classe rappresenta la logica di business del sistema di parcheggi. Gestisce operazioni complesse come l'accesso, l'uscita, o il pagamento nei parcheggi, verificandone la disponibilità. La classe interagisce con la classe del database per recuperare e aggiornare i dati relativi ai parcheggi, garantendo una gestione delle informazioni.

Il server espone due classi con annotazioni @Controller che svolgono un ruolo chiave nell'interfacciamento con gli utenti attraverso pagine web. Questi controller gestiscono le richieste HTTP provenienti dai browser, reindirizzando l'utente alle pagine corrette e consentendo interazioni. Attraverso questi controller, gli utenti e gli amministratori possono visualizzare le informazioni sui parcheggi, modificarli, aggiungerne o eliminarli.

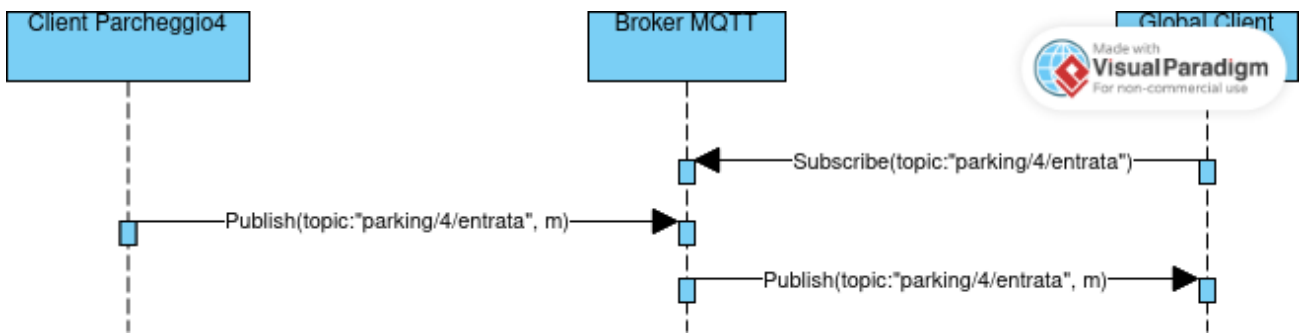
Le classi annotante @RestController sono responsabili della gestione delle REST API che forniscono funzionalità come il recupero dei dati sui parcheggi e la visualizzazione o la modifica.

5 – Database: è il componente in cui vengono archiviate e gestite le informazioni dei parcheggi; la classe che gestisce i compiti del database è DbCreate.java e viene richiamata da ParkingService per leggere, aggiungere o modificare dati nel database.

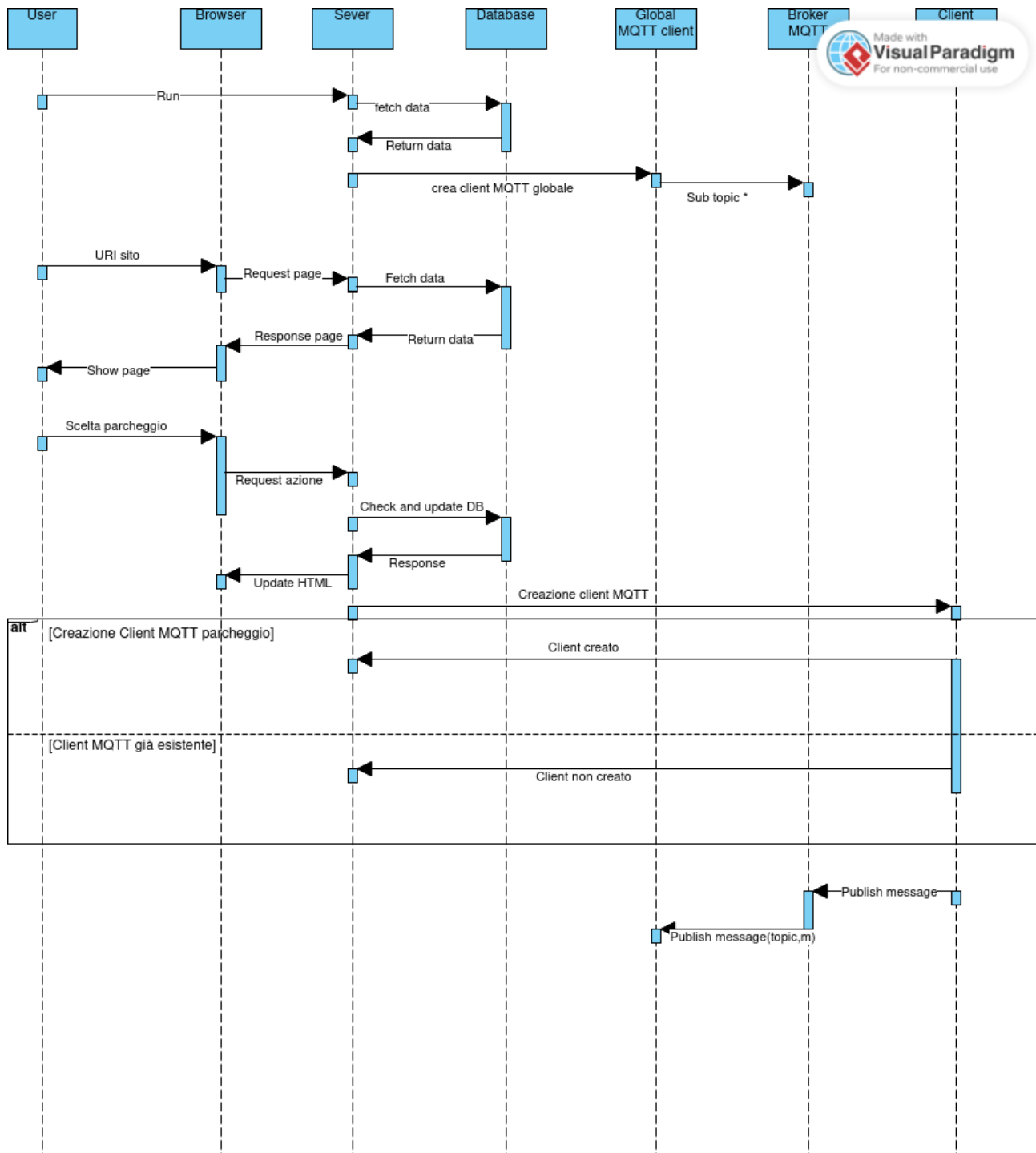
6 – Message Broker: è l'intermediario tra il mio web server e ogni parcheggio. Gestisce i messaggi che vengono pubblicati dai vari parcheggi nel momento in cui un utente desidera entrare, uscire o pagare il parcheggio.

7 – Sensori: rilevano gli avvenimenti nei parcheggi segnalando al message broker l'entrata, uscita o pagamento nel giusto parcheggio. Li ho implementati creando una classe per ogni tipo di sensore:

da qui si richiama un metodo nella classe ParkingService per creare un Thread e mandare il messaggio MQTT.



Architettura del Software



Il sistema è basato su un'architettura distribuita composta da diversi componenti interconnessi; questa permette di avere una gestione efficiente dei parcheggi attraverso l'utilizzo di dispositivi IoT, un broker MQTT, una webApp e un Database.

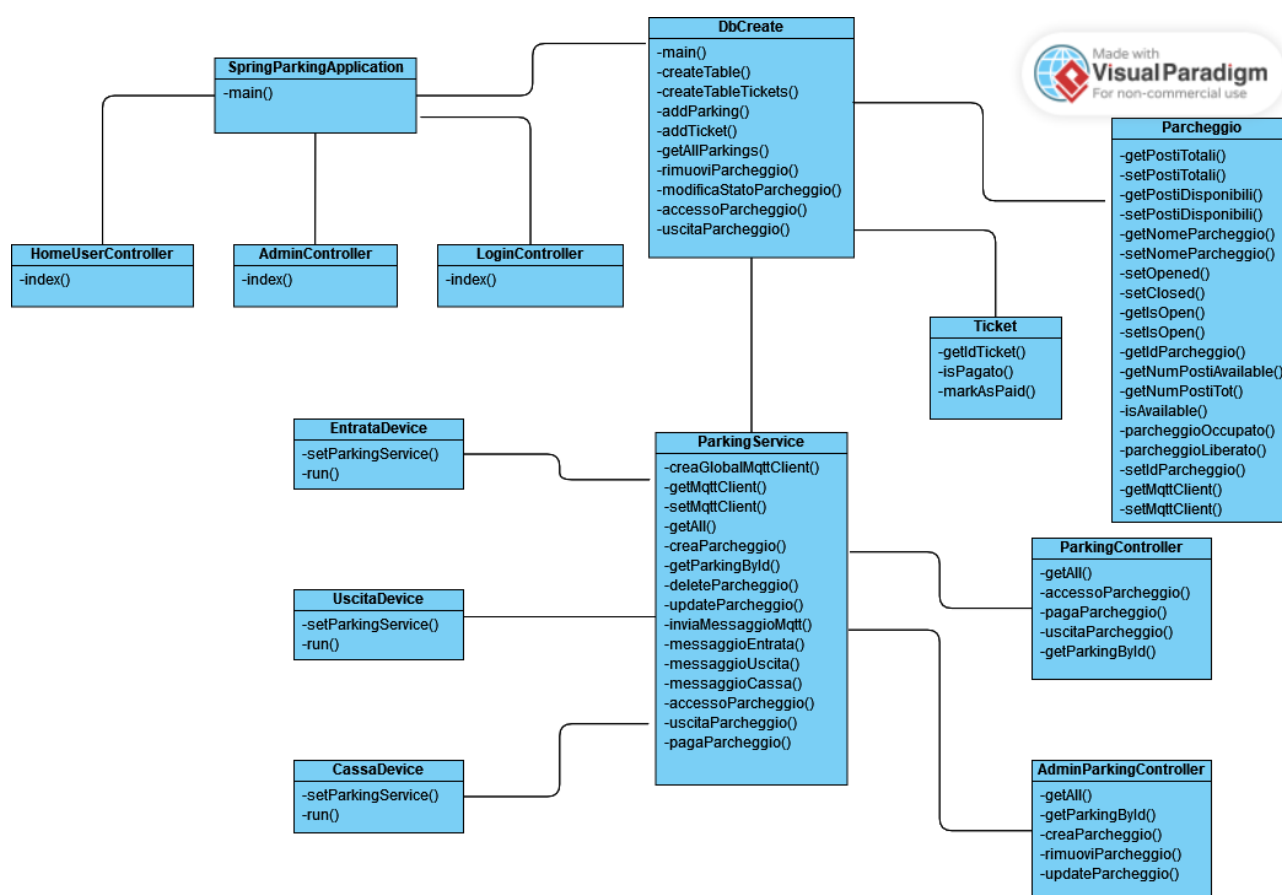
-Dispositivi IoT: ho utilizzato tre tipi di dispositivi IoT: entrata, uscita e cassa. La simulazione di questi dispositivi è stata realizzata con la creazione di 3 classi separate (EntrataDevice, UscitaDevice e CassaDevice) che sfruttando la creazione di thread, inviano messaggi al broker. In base alle vetture che entrano, escono o pagano il parcheggio vengono pubblicati messaggi sui topic adatti che hanno una struttura di questo tipo: *parking/5/entrata* (nel caso del topic di entrata del parcheggio con id 5). Questi dispositivi simulano la presenza di sensori in ogni parcheggio in grado di comunicare al message broker tutti gli avvenimenti tramite messaggi MQTT.

-Broker MQTT: Il broker MQTT in cloud funge da punto di comunicazione centrale per i dispositivi IoT. In particolare ho utilizzato il broker MQTT pubblico *test.mosquitto.org*. Questo broker MQTT in cloud consente la comunicazione affidabile e asincrona tra i dispositivi IoT attraverso il protocollo MQTT. I dispositivi IoT si connettono al broker e scambiano messaggi per inviare informazioni sull'ingresso delle vetture, sull'uscita e sul pagamento. Per quanto riguarda la porta ho scelto di usare la porta 8883 che permette una connessione criptata TLS.

-WebApp: La webApp è una semplice interfaccia che consente agli amministratori di gestire i parcheggi in modo intuitivo e agli utenti comuni di visualizzare lo stato dei parcheggi. Gli amministratori possono accedere alla pagina *Admin* tramite la pagina di *login* che utilizza l'autenticazione OAuth2 con Facebook. Una volta autenticati, gli amministratori possono visualizzare i parcheggi esistenti, aggiungere nuovi parcheggi o modificare lo stato di quelli esistenti. Invece gli utenti comuni possono solo visualizzare in una tabella i parcheggi disponibili e entrare, uscire o pagare. La webApp utilizza connessioni SSL e HTTPS sulla porta 8443 per garantire la sicurezza delle comunicazioni.

-Database: Per quanto riguarda il database ho usato DB Browser for SQLite. Il file per il database si trova in *src/databasePark.db*. Nel database sono presenti 2 tabelle: una, chiamata *parkings*, per memorizzare tutti i parcheggi che vengono creati dagli amministratori con gli opportuni attributi e un'altra, chiamata *tickets*, che serve solo per memorizzare i l'id del ticket generato casualmente quando si accede a un parcheggio.

Descrizione delle classi



Per garantire una struttura organizzata e modulare nel mio progetto, ho adottato il framework Spring Boot. Spring Boot è un framework Java che semplifica lo sviluppo delle applicazioni fornendo convenzioni predefinite e una configurazione automatica. Ho organizzato le classi Java e i package seguendo le best practice consigliate da Spring Boot. Ho utilizzato le annotazioni fornite dal framework per identificare e configurare le diverse componenti del sistema. L'utilizzo di queste annotazioni permette di sfruttare le funzionalità offerte da Spring Boot, come l'iniezione delle dipendenze e la gestione dei cicli di vita delle componenti, semplificando lo sviluppo e garantendo una struttura coerente.

Nel package com.parking.demo ci sono 3 classi:

- AppConfig.java che è una classe di configurazione per Spring Boot, infatti è caratterizzata dall'annotation @Configuration.
- DbCreate.java che è la classe che si occupa di gestire il database: qui c'è il collegamento alla posizione del file databasePark.db dove si trova effettivamente il database. Ci sono le funzioni per la creazione delle tabelle e le funzioni per l'aggiunta, rimozione, modifica del parcheggio e le funzioni per aggiornare i valori nel database nel caso di entrata o uscita dal parcheggio.
- SpringParkingApplication.java è la classe di partenza del sistema; è stata generata da SpringBoot alla creazione del progetto ed è annotata con @SpringBootApplication affinché il framework possa capire che è la classe principale.

Ho creato il package com.parking.demo.controller per ospitare le classi che fungono da controller nella mia webApp. Queste classi gestiscono le richieste HTTP e si occupano solo della logica di routing: all'interno infatti ci sono solo le 3 classi che permettono di visualizzare nella webApp le uniche 3 pagine html. Tutte queste classi sono annotate con @Controller e inoltre hanno l'annotation @RequestMapping() che specifica a SpringBoot l'url per raggiungere quella pagina html. Le classi java sono:

- AdminController.java è la classe che permette di aprire la pagina adminPage.html
- HomeController.java è la classe che permette di aprire la pagina index.html.
- LoginController.java che è la classe che permette di aprire la pagina login.html

Ho deciso inoltre di creare il package com.parking.demo.controller.api per distinguere le classi che si occupano di gestire le API RESTful; qui infatti uso l'annotation di SpringBoot @RestController per dichiarare il controller come un controller RESTful in grado di gestire le richieste e le risposte HTTP in formato JSON (che ho implementato nelle pagine html e javascript). Quindi in questo package ho 2 classi java (per le 2 pagine HTML admin e user) che implementano metodi mappati a diverse url che gestiscono metodi CRUD (create, read, update, delete). Le 2 classi sono:

- AdminParkingController.java è la classe che gestisce ciò che succede nella pagina Admin. Ad esempio nel primo metodo *getAll()*, l'url *admin/api/parkings* restituisce tutti i parcheggi disponibili, dunque ho un'operazione di lettura; invece il metodo *creaParcheggio()* si occupa di una operazione di creazione, infatti specificando *RequestMethod.POST* all'url */admin/api/parkings* che è lo stesso nella funzione javascript *addParking* (in *adminPage.html*) verrà creato un nuovo parcheggio (passando per la classe *ParkingService*) e aggiunto al database.

-ParkingController.java, allo stesso modo, è l'altra classe che si occupa di gestire le entrate, uscite e pagamenti ai parcheggi nella pagina html index.html. Anche questa classe è annotata come @RestController in quanto gestisce le richieste HTTP specificate nelle annotation @RequestMapping().

Il package com.parking.demo.model è il package che ospita le classi per definire gli oggetti di base nel mio sistema come ad esempio il parcheggio, il ticket del parcheggio o i dispositivi IoT.

-La classe Parcheggio.java si occupa di definire l'entità parcheggio con le sue caratteristiche; qui sono implementati solo gli attributi del parcheggio come id, nome, posti totali, posti disponibili e un valore di tipo String chiamato *isOpen* che gestisce il dato booleano di apertura o chiusura del parcheggio (nel database è impostato come INT in quando SQLite non dispone del tipo BOOLEAN).

-La classe Ticket.java definisce l'oggetto Ticket che è solo caratterizzato da un ID casuale e salvato nella tabella *tickets* del database ogni volta che si esegue un accesso in un parcheggio.

I 3 dispositivi IoT li ho implementati come 3 classi separate ma sono molto simili. Tutti e 3 implementano l'interfaccia Runnable che consente di eseguire il device IoT come un thread separato. In ogni classe vi è la variabile *idParcheggio* che rappresenta l'id del parcheggio a cui è associato il device IoT (di entrata, uscita o cassa). Il metodo run viene eseguito quando viene avviato come thread nella classe ParkingService (nei 3 metodi accessoParcheggio(), uscitaParcheggio() e pagaParcheggio()) e si occupa di inviare il messaggio MQTT adeguato. Ogni messaggio MQTT verrà poi ricevuto dal *global-client* in ParkingService.

Infine l'ultimo package è com.parking.demo.service che ospita solo una classe:

ParkingService.java è la classe annotata come @Service cioè quella classe che secondo Spring Boot, si occupa di gestire la business logic, infatti qui vengono richiamati tutti i metodi per gestire i parcheggi. L'obiettivo della suddivisione tra classe @Service e classi @Controller è che nei controller non devono essere implementati i metodi di gestione dei parcheggi ma devono solo essere richiamati; sarà nella classe service che vengono creati quei metodi e sarà la classe service che accederà al database per svolgere le operazioni di lettura e modifica ecc. In questa classe ho dichiarato url del broker, cioè *broker = "ssl://test.mosquitto.org:8883"*: il message broker ha il compito di gestire i messaggi MQTT che vengono mandati dai vari client mqtt (un client mqtt per ogni parcheggio dove l'id del parcheggio è anche l'id del client mqtt) nei giusti topic in base a quello che viene monitorato nel parcheggio. Avendo scelto un broker con connessione TLS sicura ho prima scaricato il certificato mosquitto.org.crt e dopo nel codice ho dovuto specificare delle MqttConnectOptions ogni volta che andavo a creare un client mqtt (cosa non necessaria se si usa la porta 1883) e allo stesso modo ho creato un client globale che è sottoscritto a tutti i topic di tutti i parcheggi: questo, tramite il metodo *messageArrived()*, permette di visualizzare con stampe su console i messaggi che arrivano sui terminali. All'interno di questa classe sono implementati i metodi che vanno a richiamare la classe DbCreate.java per andare a modificare il database in seguito a creazioni, modifiche o rimozioni di nuovi parcheggi. E' presente il metodo *inviaMessaggioMqtt()* che ha il compito di inviare il messaggio MQTT sul topic corretto creando un nuovo mqttClient (per quel parcheggio) solo se non esiste ancora nella lista parcheggi. Questo metodo verrà richiamato dai metodi

messaggioEntrata(), *messaggioUscita()*, *messaggioCassa()* che specificano il corretto testo del messaggio e questi metodi verranno richiamati dai Device IoT. Infine i metodi *accessoParcheggio()*, *uscitaParcheggio()* e *pagaParcheggio()* andranno a richiamare i Device IoT.

Descrizione dell'interfaccia utente

Ho realizzato una semplice interfaccia utente creando 3 pagine HTML che, seguendo le indicazioni sulla struttura di un progetto Spring, ho posizionato in `src/main/resources/templates`.

La pagina User è quella pagina che mostra una tabella con l'elenco dei parcheggi e le loro caratteristiche; è possibile entrare o uscire dai parcheggi tramite i pulsanti ENTRA o ESCI ed è possibile pagare il ticket del parcheggio con il pulsante PAGA. Questa pagina è visitabile da tutti gli utenti. Ci sono 4 funzioni javascript che vengono richiamate nella classe `ParkingController` tramite le annotation Spring `@RequestMapping`.

Cliccando su "Vai alla pagina Admin" si può raggiungere la pagina di Amministratore ma prima bisogna effettuare il login tramite OAuth2.0 nella pagina `login.html`. Nella pagina Admin è possibile modificare lo stato del parcheggio tramite il pulsante ATTIVA/DISATTIVA ed eliminare un parcheggio con il pulsante RIMUOVI; come nel caso precedente, ogni click richiama la funzione javascript associata che richiama il metodo java opportuno in `AdminParkingController` (osservando la `@RequestMapping`). Infine scorrendo in basso, è possibile aggiungere un nuovo parcheggio con le opportune caratteristiche. Cliccando in alto su Logout si torna alla pagina User.

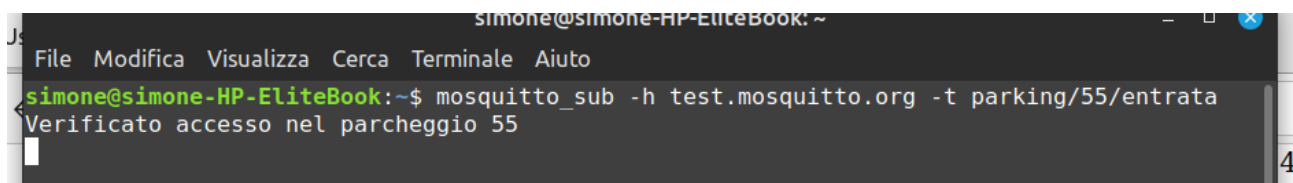
Validazione

Per poter testare la webApp integralmente e quindi poter svolgere anche le azioni dell'amministratore bisogna effettuare il login tramite OAuth2.0 tramite Facebook; nell'app Facebook che ho dovuto creare per abilitare il login ho dato l'accesso a un utente di prova. Per testare, ecco le credenziali:

email= 20034352@studenti.uniupo.it

password= ^LW6W"4B"~F+7m+

Nella prima fase di sviluppo testavo gli url delle pagine con Postman e controllavo che il server rispondesse con 200 OK; dopo, una volta che le pagine funzionavano correttamente, non ho più usato Postman ma controllavo direttamente sul browser, al massimo ispezionando la pagina web. Ho creato un metodo per testare l'effettivo invio dei messaggi MQTT sui diversi topic: nella classe `ParkingService` ho creato un mqtt client globale che tramite il metodo *messageArrived()* si accorge se arriva il messaggio (in quanto sempre in ascolto), su che topic arriva e il contenuto mqtt; ora, a seconda del tipo di messaggio ricevuto (entrata, uscita o cassa), va a eseguire la stampa adeguata sulla console. Per esempio, lanciando il programma e creando un terminale in ascolto sul topic `parking/55/entrata`, possiamo raggiungere la pagina User (<https://localhost:8443/>) e cliccare sul pulsante "ENTRA" del parcheggio 55: in seguito all'accesso, se si torna sul terminale si può notare il messaggio MQTT che viene stampato.

A screenshot of a terminal window titled "simone@simone-HP-EliteBook: ~". The terminal shows a command prompt where the user has entered `mosquitto_sub -h test.mosquitto.org -t parking/55/entrata`. Below the command, the terminal displays the output "Verificato accesso nel parcheggio 55". The terminal window has a menu bar with "File", "Modifica", "Visualizza", "Cerca", "Terminale", and "Aiuto".

Inoltre tornando sulla console del nostro IDE possiamo vedere che è stata eseguita una stampa in quanto è stato ricevuto dal client globale un messaggio MQTT:

```
2023-07-12T10:26:48.717+02:00 DEBUG 3415 --- [nio-8443-exec-8] o.s.web.servlet.DispatcherServlet : Completed 200
2023-07-12T10:28:03.832+02:00 DEBUG 3415 --- [nio-8443-exec-4] o.s.web.servlet.DispatcherServlet : PUT "/user/ap
2023-07-12T10:28:03.833+02:00 DEBUG 3415 --- [nio-8443-exec-4] s.w.s.m.a.RequestMappingHandlerMapping : Mapped to con
Ticket generato per l'ingresso: 530569933
2023-07-12T10:28:03.914+02:00 DEBUG 3415 --- [nio-8443-exec-4] m.m.a.ResponseBodyMethodProcessor : Using 'applic
2023-07-12T10:28:03.915+02:00 DEBUG 3415 --- [nio-8443-exec-4] m.m.a.ResponseBodyMethodProcessor : Nothing to wr
2023-07-12T10:28:03.915+02:00 DEBUG 3415 --- [nio-8443-exec-4] o.s.web.servlet.DispatcherServlet : Completed 200
Messaggio ricevuto. Topic: parking/55/entrata, Messaggio MQTT: Verificato accesso nel parcheggio 55
Apertura transenna per l'entrata del parcheggio 55
```

La stampa eseguita ci notifica che il messaggio è stato ricevuto: ci specifica il topic *parking/55/entrata* e il contenuto del messaggio; infine, solo dopo la ricezione del messaggio, viene eseguita la corretta apertura delle transenne con la stampa “Apertura transenna per l’entrata del parcheggio 55”.

Infine, per quanto riguarda i test sul corretto funzionamento del database andavo direttamente a controllare gli aggiornamenti delle tabelle in *databasePark.db* aprendo DB Browser for SQLite.