SCHEMA RAPPORTO FINALE

SERVIZIO DI PARCHEGGIO

Autore: Giulia Sonsino (matricola: 20036727)

SPECIFICHE FUNZIONALI

Il progetto prevedeva la realizzazione di un servizio per la gestione di parcheggi cittadini. Ogni parcheggio ha a disposizione tre dispositivi IoT, ovvero Ingresso, Uscita e Cassa.

L'utente deve poter visualizzare lo stato dei parcheggi, quindi posti disponibili e stato del parcheggio (attivo o disattivo). Può quindi entrare in un parcheggio attivo e avente posti a disposizione, grazie all'apertura delle transenne. Per poter uscire deve prima pagare il ticket che gli è stato consegnato all'ingresso.

Oltre alla figura dell'utente, c'è la figura dell'amministratore. Quest' ultimo può accedere a una zona riservata del sistema in cui può aggiungere ed eliminare parcheggi, o cambiarne lo stato di attività.

ANALISI DELLA TECNOLOGIA

Per realizzare il progetto ho realizzato anche una WebApp con cui l'utente può interfacciarsi e simulare quindi le funzioni che potrebbero avvenire in un reale parcheggio. Ho infatti utilizzato il protocollo HTTP (che in seguito ho trasformato in HTTPS con l'uso di un certificato) attraverso l'architettura REST. Quest'ultimo è uno stile architetturale per sistemi distribuiti in rete. L'obiettivo di REST è quello di creare un'architettura per la comunicazione tra client e server che sia semplice, scalabile ed efficiente. I concetti chiave di questo stile architetturale sono: i nomi , ovvero gli URI con cui si identificano le risorse, poi i verbi, ovvero i metodi GET POST PUT DELETE e PATCH, ed infine le rappresentazioni, ovvero linguaggi come XML, HTML, JSON.

Dunque lo stile architetturale REST si usa per applicazioni che richiedono maggiore flessibilità. E' infatti stateless, ovvero "senza stato", nel senso che non vengono salvati i dati generati in una certa sessione per poterli utilizzare in quelle successive. Infatti questo approccio, a differenza di quello stateful (in cui si stabilisce una sessione tra client e server durante la quale si memorizzano tutte le informazioni che vengono a evolversi durante il dialogo stesso), risulta più robusto in ambiente di rete, infatti cadute di rete o del server sono tollerate molto bene, ed è più semplice gestire il parallelismo. Infatti è possibile avere un thread pool, ovvero una tecnica di ottimizzazione dei thread che ne prevede la creazione di un gruppo in anticipo e quando arriva una richiesta viene assegnata ad uno di essi.

In particolare ho utilizzato HTTPS (sulla porta 8443), che è una variante sicura del protocollo HTTP che prevede infatti l'aggiunta di una connessione sicura tramite SSL/TLS per proteggere i dati trasmessi. Per far ciò ho dovuto generare un certificato che ho incluso nel progetto.

Per quanto riguarda l'autenticazione ho utilizzato OAuth2.0. L'obiettivo del protocollo è quello di fornire uno strumento che consenta a un'applicazione di accedere a determinate informazioni a seguito dell'autorizzazione dell'utente che ne ha il possesso. In particolare i passaggi che avvengono sono i seguenti:

- il client effettua una richiesta di autorizzazione all'utente

- l'utente concede l'autorizzazione

- il client richiede un token di accesso al server di autorizzazione

- quindi il server verifica le credenziali del client e la validità della richiesta

- dunque emette un token di accesso al client

-il client utilizza il token di accesso per accede alle risorse

Io in particolare ho sfruttato il login con Google. Infatti per far si che l'amministratore possa accedere alle zone riservate della web app è necessario che si autentichi. Quindi, nel momento in cui vuole autenticarsi, viene condotto a una pagina login di Google in cui può inserire l'username e la password del suo profilo Google. Dunque, se l'autenticazione ha successo, viene mandato un token (ovvero una stringa che contiene informazioni personali dell'utente) alla mia applicazione.

Per quanto riguarda la gestione dei dispositivi IoT, ovvero cassa ingresso e uscita, ho sfruttato il protocollo MQTT. Si tratta di un protocollo leggero orientato ai Message Brokers e basato sul modello publish-subscribe. E' molto utilizzato in domotica e per 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. Cosi 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.

Ho utilizzato quindi MQTT, anche se in realtà esistono altri protocolli Message Oriented. Uno di essi è AMQP, che usa come broker RabbitMQ. Però l'utilizzo di MQTT per il mio progetto è risultata una scelta migliore in quanto è un protocollo più leggero e più adatto alla gestione di dispositivi IoT.

SCELTA DELL'APPROCCIO

Per svolgere il progetto un punto cardine è stato sicuramente l'utilizzo del protocollo MQTT (Message Queue Telemetry Transport). Si tratta di un protocollo leggero orientato ai Message Brokers e basato sul modello Publish-Subscribe. Esistono infatti client che possono iscriversi a molteplici topics , ovvero argomenti, e attendere che messaggi arrivino su di essi. Infatti altri client possono pubblicare messaggi su determinati topics, che verranno appunto letti dai client sottoscritti ad essi. E' dunque compito del Message Broker filtrare e distribuire i messaggi, garantendo affidabilità ed efficienza.

In particolare, nel mio progetto, ho utilizzato un Message Broker posto in cloud, ho sfruttato infatti test.mosquitto.org. Ho usato la porta 8883. Questa porta infatti viene usata per la connessione sicura con il protocollo MQTT tramite SSL/TLS. Viene infatti usata per comunicazioni crittografate tra client MQTT e broker. Per poter usare questa porta mi sono avvalsa di un certificato.

Il compito del Broker è dunque quello di gestire la distribuzione dei messaggi. Ho creato dei client, uno per ogni parcheggio esistente nel sistema, che vanno a pubblicare messaggi su topic diversi, svolgendo quindi il ruolo di publisher. D'altra parte il subscriber lo ho realizzato come un client più globale che è sottoscritto a tutti i topic e che quindi, ricevendo i messaggi, si comporta di conseguenza.

Infatti, entrando più in dettaglio, quando un utente simula l'ingresso in un parcheggio, l'MqttClient relativo a quel parcheggio (caratterizzato da un identificativo) pubblica il messaggio "Richiesto ingresso" sul topic "from/idParcheggio/ingresso" . Questo verrà quindi intercettato dal client sottoscritto al topic che capisce quindi che deve fare aprire le transenne del parcheggio, caratterizzato dall'idParcheggio presente nel topic, per far entrare la macchina.

Inoltre quando si entra nel parcheggio viene consegnato un ticket, che ho rappresentato da un numero casuale che viene dato all'utente all'entrata. Infatti per pagare tale biglietto è necessario inserire il numero corrispondente al ticket che ci era stato consegnato. Nel momento del pagamento, infatti, viene pubblicato dal client relativo al parcheggio di cui l'utente desidera pagare il biglietto, il messaggio "Pagamento in corso per ticket" sul topic "from/idParcheggio/cassa". Cosi facendo il client sottoscritto ai topic, all'arrivo di esso, sa che un utente ha pagato correttamente un biglietto relativo al suo parcheggio e quindi la cassa può emettere il ticket pagato.

Infine per poter uscire dal parcheggio è necessario, appunto, aver pagato il ticket. Ho simulato ciò richiedendo all'utente, nel momento in cui desidera uscire, di scrivere il numero del ticket. Viene quindi fatto un controllo, e se il ticket risulta pagato e risulta effettivamente stato rilasciato da quel parcheggio, allora viene mandato il messaggio "Richiesta uscita approvata" sul topic "from/idParcheggio/uscita". Altrimenti, in caso di pagamento mancante, il messaggio ad essere inviato sarà "Richiesta uscita negata". Dunque il client sottoscritto ai topic, nel momento in cui vede arrivare un messaggio sul topic relativo all'uscita di uno dei parcheggi, ne analizza il contenuto. Se esso equivale a "Richiesta uscita approvata" allora permetterà al parcheggio in considerazione di aprire le transenne per poter permettere alla macchina di uscire. Nel caso del

messaggio “Richiesta uscita negata” allora capisce che l’uscita non è permessa e non apre le transenne.

ARCHITETTURA DEL SOFTWARE

Ho strutturato il progetto in cartelle distinte. Ho realizzato il package “componenti” in cui ho racchiuso le classi java che hanno diversi compiti (che dopo analizzerò più attentamente) legati al back end del progetto. Come ad esempio al recupero e modifica di dati dal Database. Poi ho la cartella main/webapp in cui ho inserito i file scritti in html e javascript dedicati quindi alla creazione di un’interfaccia utente e alla gestione delle richieste dell’utente. Infine nella cartella database ho appunto il file con estensione .db e una classe Java per la gestione di esso. Come database ho utlizzato DB Browser for SQlite.

Per gestire i dispositivi IoT, per ognuno dei quali ho creato una classe, ho sfruttato i thread. In particolare ho creato un pool di 10 thread, e nel momento in cui se ne ha bisogno vengono dati ad essi dei compiti. In particolare nel momento in cui un utente desidera entrare in un parcheggio e quindi l’MqttClient relativo a quel parcheggio deve pubblicare un messaggio sul topic “from/idParcheggio/ingresso”, questo compito viene svolto da un thread in quel momento disponibile. Analogamente per la funzione di uscita e del pagamento. Questo approccio dovrebbe rendere più semplice la gestione del parallelismo, perché si possono gestire più richieste contemporaneamente senza dover per forza creare sempre nuovi thread, aumentando efficienza e scalabilità del sistema.

DESCRIZIONE

In questa sezione vado ad analizzare le classi del mio progetto:

All’interno del package “componenti” troviamo:

- WebServer.java : si tratta di una classe principale del mio progetto che si occupa di configurare e avviare il server HTTPS sulla porta 8443. Infatti, una volta avviato il programma, sarà possibile accedere ad https://localhost:8443 e iniziare a visionare la mia web app. Inoltre la classe WebServer, all’interno di MyHttpHandler, si occupa della gestione delle richieste HTTP. In particolare la funzione handle() controlla il percorso richiesto e il metodo HTTP utilizzato (GET o POST). Ad esempio, nel caso del percorso “/”, corrispondente alla home della web app, ci sono le funzioni handleGetRequest() e handlePostRequest() che gestiscono rispettivamente le richieste GET e POST. Come per “/”, anche per tutti gli altri percorsi (quali “/opzioni Amministratore”, “/area Utente”, “/pagamento”, “/uscita Parcheggio” etc…) troviamo delle funzioni che gestiscono le richieste HTTP.

- MqttClients.java : è una classe in cui vado a far sottoscrivere a tutti i topic un MqttClient globale, che ha appunto il compito di ricevere i messaggi e gestirli nel modo corretto. Questa classe viene infatti chiamata, all'inizio del programma, nella precedente classe WebServer. Infatti , all'inizio, si fa si che questo client vada a sottoscriversi ai topic "from/idParcheggio/ingresso", "from/idParcheggio/uscita" e "from/idParcheggio/cassa" per ogni parcheggio presente nel database (e caratterizzato quindi da un suo idParcheggio). Poi anche quando un amministratore crea un nuovo parcheggio, allora viene richiamata la classe MqttClients per far sottoscrivere il client ai nuovi topics per il nuovo parcheggio appena creato. In questa classe grazie alla funzione messageArrived() si gestisce la ricezione dei messaggi sui topics.

- ParkingManager.java: questa è una classe che ho utilizzato semplicemente per richiamare alcune funzioni che sono presenti nella classe CreateTables.java, presente nel package database. L'ho creata per aiutarmi nel richiamo di alcune funzioni e avere un po' più di ordine nel codice.

- Parcheggio.java : utilizzo questa classe per definire l'oggetto Parcheggio e i suoi parametri, ovvero "quartiere", "via", "posti Totali", "posti Liberi", "attivo" e "idMQTT". Questi rappresentano infatti il quartiere della città e via in cui si trova il parcheggio, il numero di posti totali e liberi, lo stato di attività (attivo o disattivo) e l'id che identifica il parcheggio. Uso questa classe, ad esempio, nel momento in cui vado a creare un nuovo Parcheggio e per accedere ai campi di esso.

- Ingresso.java, Uscita.java, Cassa.java : analizzo queste tre classi contemporaneamente in quanto hanno una struttura molto simile. Infatti tutte e tre implementano la classe Runnable() e hanno quindi il metodo run(). Hanno tutte una struttura simile, in quanto si occupano di far si che l'MqttClient relativo al parcheggio in considerazione pubblichi un messaggio sul topic corretto ("from/IdParcheggio/ingresso" o "from/IdParcheggio/uscita" o"from/IdParcheggio/cassa" a seconda della classe). Queste classi vengono chiamate nella classe WebServer nel momento in cui un utente entra, esce o paga un ticket. Tutte e tre necessitano di un Thread.

Nel package "database" troviamo invece:

- CreateTables.java: questa classe si occupa della creazione delle tabelle nel database. In particolare le tabelle "parcheggi" e "tickets". Inoltre nella classe sono presenti diverse funzioni che hanno il compito di modificare, eliminare o selezionare valori dal database.

- DatabaseParcheggio.db: questo è effettivamente il database del progetto. In particolare io ho usato DB Browser for SQLite. Questo file viene appunto richiamato in CreateTables.java.

DESCRIZIONE DELL'INTERFACCIA UTENTE

Infine ho la cartella “main/webapp” in cui gestisco l’interfaccia con l’utente. Analizzo brevemente i file in essa presenti:

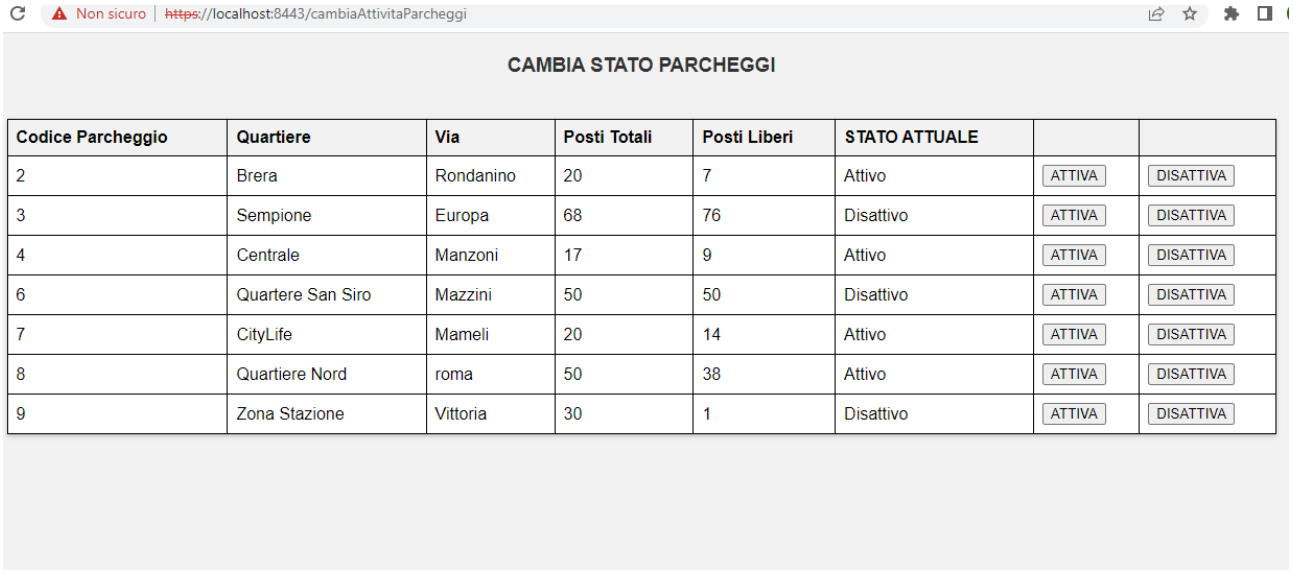
- **home.html**

è la pagina principale in cui si ha la possibilità di andare nell’area amministratori oppure nella sezione parcheggi, ovvero quella aperta a tutti gli utenti.

- **opzioniAmministratore.html**

qui si viene ricondotti se dalla home si decide appunto di andare nell’area degli amministratori. Per accedere qui è richiesto il login (con OAuth2.0). Nella pagina opzioniAmministratore è quindi possibile scegliere tra: aggiungi/elimina parcheggio o la possibilità di cambiare stato ai parcheggi. E’ inoltre presente il bottone LOGOUT.

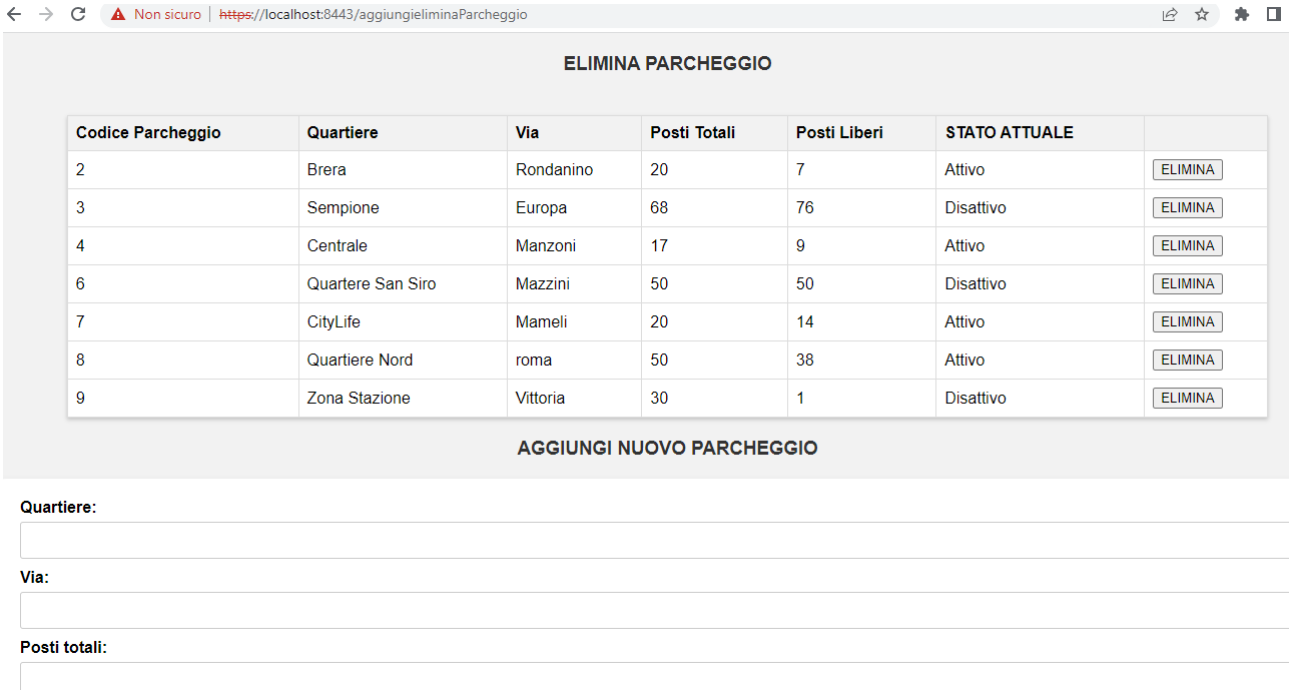
- **cambiaAttivitaParcheggi.html**



Codice Parcheggio	Quartiere	Via	Posti Totali	Posti Liberi	STATO ATTUALE		
2	Brera	Rondanino	20	7	Attivo	ATTIVA	DISATTIVA
3	Sempione	Europa	68	76	Disattivo	ATTIVA	DISATTIVA
4	Centrale	Manzoni	17	9	Attivo	ATTIVA	DISATTIVA
6	Quartere San Siro	Mazzini	50	50	Disattivo	ATTIVA	DISATTIVA
7	CityLife	Mameli	20	14	Attivo	ATTIVA	DISATTIVA
8	Quartiere Nord	roma	50	38	Attivo	ATTIVA	DISATTIVA
9	Zona Stazione	Vittoria	30	1	Disattivo	ATTIVA	DISATTIVA

qui l'amministratore ha modo di cambiare lo stato dei parcheggi, da attivo a disattivo e viceversa.

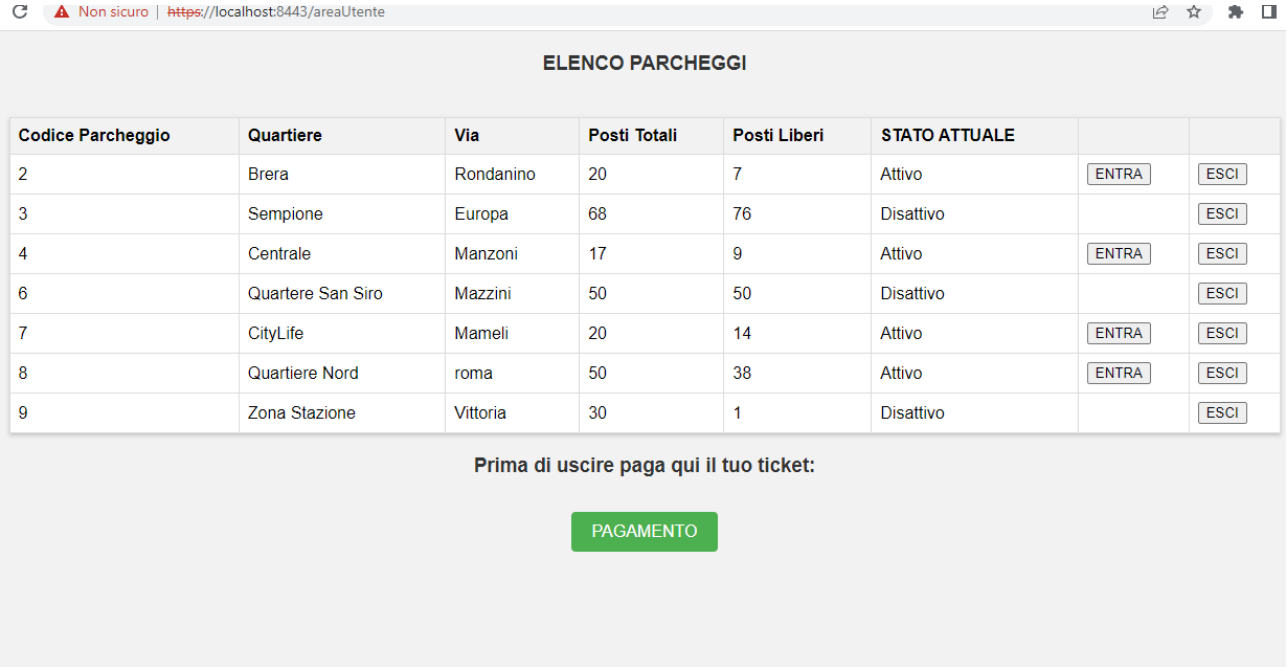
- **aggiungieliminaParcheggio.html**



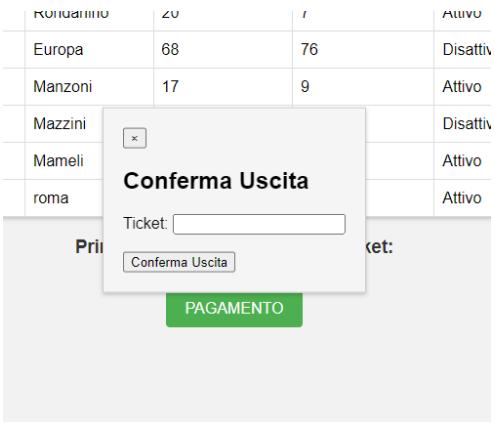
Codice Parcheggio	Quartiere	Via	Posti Totali	Posti Liberi	STATO ATTUALE	
2	Brera	Rondanino	20	7	Attivo	ELIMINA
3	Sempione	Europa	68	76	Disattivo	ELIMINA
4	Centrale	Manzoni	17	9	Attivo	ELIMINA
6	Quartere San Siro	Mazzini	50	50	Disattivo	ELIMINA
7	CityLife	Mameli	20	14	Attivo	ELIMINA
8	Quartiere Nord	roma	50	38	Attivo	ELIMINA
9	Zona Stazione	Vittoria	30	1	Disattivo	ELIMINA

in cui è appunto possibile eliminare un parcheggio o crearne uno nuovo, inserendo tutti i parametri necessari (quartiere, via, posti totali, posti liberi e stato di attività). Cliccando quindi su salva avviene un controllo relativo al corretto inserimento di tutti i campi, e quindi in caso affermativo possiamo vedere la tabella aggiornata con tutti i parcheggi.

- **areaUtente.html**

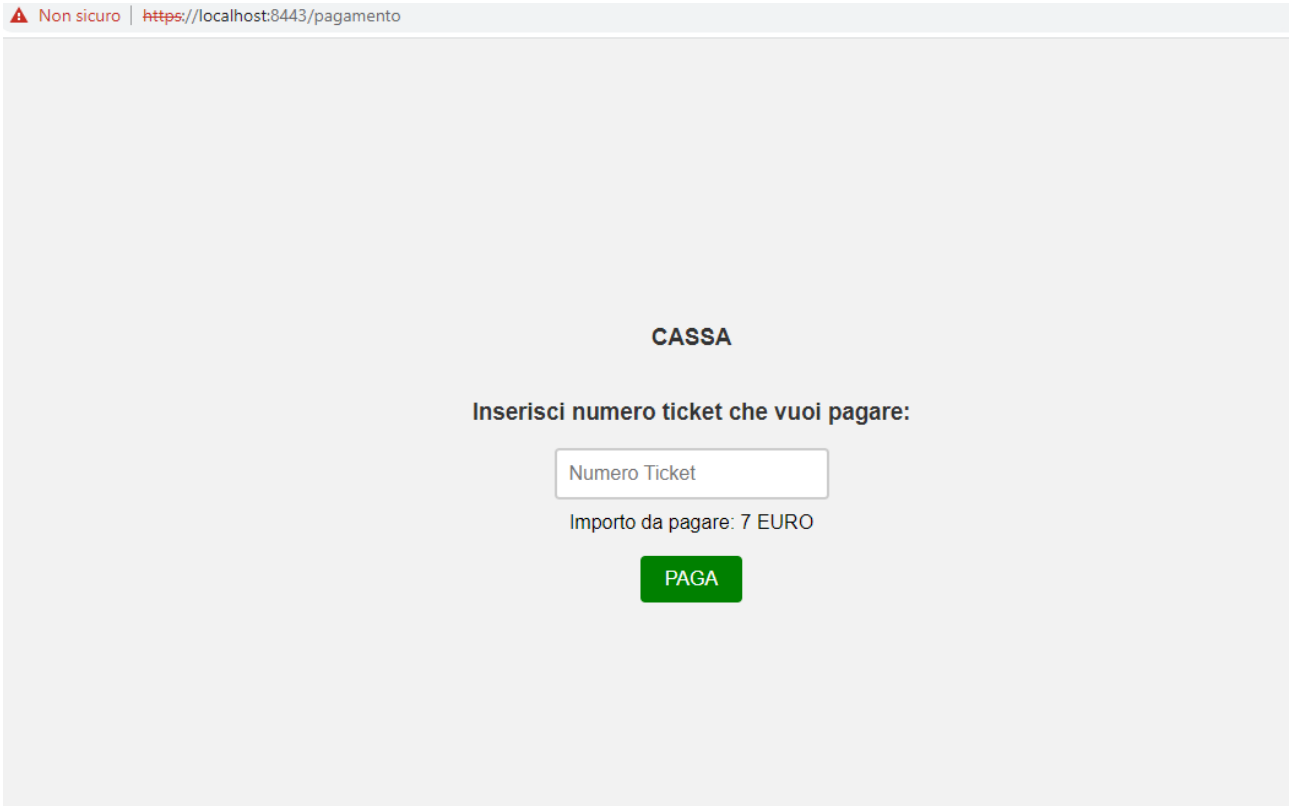


qui ci si arriva se dalla schermata home, invece di andare nella sezione dedicata agli amministratori, si va nella zona riservata al consulto dei parcheggi. Nell'area utente è possibile infatti vedere lo stato dei parcheggi e i posti in essi disponibili. Quindi , se c'è disponibilità, si può simulare l'ingresso premendo sul bottone "entra". A questo punto verrà dato un numero rappresentante il ticket da pagare. Per pagarlo bisogna cliccare su PAGAMENTO (e si viene ricondotti alla pagina /pagamento che analizzo successivamente). Dunque si può poi uscire, cliccando analogamente su "esci". Al momento dell'uscita viene chiesto in un dialog di inserire il codice del ticket che si ha pagato. Questo è il dialog che si apre:



Cliccando su "Conferma uscita", se effettivamente si aveva pagato il ticket, si può quindi uscire dal parcheggio.

- **pagamento.html**



qui è possibile inserire il numero del ticket ricevuto e pagarlo. Dunque una volta pagato il ticket che è stato consegnato all'ingresso sarà possibile uscire dal parcheggio (infatti non si può usare il ticket rilasciato all'ingresso di un certo parcheggio per uscire da un altro parcheggio).

Tutti questi file html presentano ovviamente del codice javascript per gestire l'interazione con gli utenti. Queste parti di codice si trovano all'interno di <script>.

VALIDAZIONE

Si tratta dunque di una web app che simula un sistema di parcheggi. Ovviamente essendo una simulazione di esso, alcune azioni, quali l'apertura o chiusura delle transenne sono solo simulate.

Per validare il software è necessario fare andare il programma partendo appunto dalla classe WebServer. Infatti essa ci permetterà di poter andare all'url https://localhost:8443. Una volta fatto accesso a questo url è possibile navigare nella web app. Andando nell'area riservata agli amministratori verrà chiesto di fare il login con Google. Fatto ciò si viene indirizzati alla pagina /opzioni Amministratore. Qui è possibile decidere di aggiungere o eliminare parcheggi o cambiare lo stato di attività di essi. E' possibile fare ciò e andare anche a verificare nel database la presenza dei parcheggi corretti. Quindi facendo LOGOUT si viene reindirizzati nella home (ovvero "/"). Qui dunque, oltre all'area amministratori, è possibile accedere tramite il bottone PARCHEGGI all'area utente. Qui si possono simulare le azioni di entrata ed uscita da un parcheggio ed è interessante

vedere come i messaggi vengono mandati sui vari topics. Infatti per validare ciò è possibile aprire un terminale e sottoscriversi a un certo topic, ad esempio con : “mosquitto_sub –h test.mosquitto.org -t from/4/ingresso” . Così facendo siamo sottoscritti al topic relativo all’ingresso del parcheggio 4, ed è interessante provare dunque a cliccare sul bottone “entra” del parcheggio 4 e vedere apparire il messaggio sul nostro terminale. Inoltre, una volta fatto ciò, l’MqttClient globale che è in ascolto su tutti i topic lo intercetta e capisce quindi che dovrebbe aprire le transenne del parcheggio 4. Ciò lo ho testato sfruttando la console di Eclipse, in cui ho infatti sviluppato il progetto. Infatti ho fatto si che nel momento in cui il client riceve il messaggio in considerazione stampi sul terminale un messaggio, grazie al codice “System.out.println(“Apri transenne del parcheggio: “ + id);”.

Analogamente è dunque interessante iscriversi al topic “from/4/cassa” per vedere la ricezione del messaggio nel momento in cui si andrà a pagare. Infatti quando si entra nel parcheggio, ad esempio supponiamo il parcheggio numero 4, viene dato all’utente un codice del ticket (che consiglio di copiarsi, altrimenti è facilmente reperibile dal database!) che dovrà quindi usare cliccando sul bottone Pagamento. Si viene quindi reindirizzato alla pagina di pagamento dove basta inserire il codice ricevuto all’ingresso. Se effettivamente si va a pagare un ticket che è stato generato entrando, allora l’MqttClient del parcheggio da cui è stato rilasciato il ticket (ovvero il 4 nel nostro esempio) andrà a pubblicare il messaggio “Pagamento in corso per ticket” sul topic from/4/cassa (che possiamo dunque vedere sul nostro terminale). E quindi il client che andrà a ricevere questo messaggio stamperà su console (“pagamento in corso, emettere ticket pagato nel parcheggio: “+id).

Infine si può uscire dal parcheggio cliccando sul bottone “esci”. Si aprirà quindi un dialog in cui l’utente deve inserire il numero del ticket che ha pagato. A questo punto avverrà un controllo: ovvero se il ticket risulta pagato e se è davvero un ticket che è stato rilasciato dal parcheggio da cui si vuole uscire, allora l’MqttClient andrà a pubblicare il messaggio “Richiesta uscita approvata” sul topic “from/4/uscita” (sempre che stiamo uscendo sempre dal parcheggio 4. Altrimenti ci sarà l’id del parcheggio in considerazione). Altrimenti in caso di pagamento non avvenuto o non corrispondenza con i ticket rilasciati dal parcheggio in questione, il messaggio sul topic sarà “Richiesta uscita negata”. Dunque aprendo il terminale e mettendolo questa volta in ascolto sul topic legato all’uscita, risulta interessante intercettare questi messaggi. Inoltre l’MqttClient globale, anche per quanto riguarda l’uscita, intercetta i messaggi e si comporta di conseguenza. Se legge sul topic di interesse il messaggio “Richiesta uscita approvata” allora stamperà sulla console (di Eclipse) il messaggio (“Uscita approvata. Apri transenne del parcheggio: “ + id”). Altrimenti, nel caso del messaggio “Richiesta uscita negata”, stamperà (“Uscita negata per parcheggio: “ +id”).

Ho fatto in questo modo per simulare quella che nella realtà sarebbe stata la reale apertura delle transenne di un parcheggio.