Smart Parking
Sistemi Distribuiti

Boldini Davide, 0001075697
davide.boldini@studio.unibo.it

Bucchieri Davide, 0001075283
davide.bucchieri@studio.unibo.it

1 dicembre 2023

Indice

0.1 Introduzione

Oggigiorno sempre più frequentemente viene alla luce il problema della ricerca di parcheggi per la sosta di auto e moto. Ciò accade maggiormente nelle aree urbane più densamente popolate. Nasce da qui l'idea di creare una piattaforma che venga incontro alle necessità degli utenti che si ritrovano spesso con questa problematica da affrontare: una mappa che segnali lo stato al momento attuale per ogni punto sosta (a strisce blu) e che permetta di prenotarne l'occupazione in base alle esigenze dell'utente.
Nel corso di questa relazione, useremo i seguenti termini come sinonimi:

- Parcheggio
- Area di sosta
- Punto di sosta
- Posto auto

Indicano tutti una zona delimitata da strisce blu adibita al deposito di un autoveicolo (nel progetto non saranno fatte distinzioni fra auto e moto). Con Aree, invece, si indicherà un insieme di parcheggi con lo stesso prezzo (o meglio vincolati nello stesso prezzo, visto che nulla vieta di avere aree con prezzi uguali).

0.2 Requisiti

Smart Parking nasce come soluzione per permettere agli utenti di avere una visione d'insieme sullo stato attuale delle aree di sosta, a pagamento, messe a disposizione dalla sovrintendenza comunale. Il tutto facendo uso di viste satellitari ed indicatori che evidenziano le aree supportate dal servizio.

Ogni punto di sosta sarà raffigurata in base al suo stato:

- Disponibile
- Occupata da altri
- Occupata da te
- Prenotata da altri
- Prenotata da te

Per ogni posto auto, inoltre, sarà presente:

- Foto rappresentativa
- Id associato
- Stato
- Indirizzo

- Prezzo per ora
- Note (facoltativo)

Nel caso il parcheggio sia disponibile allora l'utente potrà procedere alla prenotazione.

Ogni utente avrà a disposizione un proprio profilo in cui potrà inserire, oltre che i dati anagrafici ed una foto profilo (facoltativa), anche i veicoli che intenderà utilizzare per le prenotazioni inserendo:

- Foto veicolo (facoltativo)
- Tipologia
  - Auto
  - Moto
- Targa
- Marca
- Modello
- Anno immatricolazione
- Colore
- Note (facoltativo)

Ed anche diversi metodi di pagamento tra cui:

- PayPal
- Carta di credito

Ogni profilo conterrà inoltre una sezione *"Prenotazioni"* con:

- Prenotazioni attive
- Archivio prenotazioni

In fase di prenotazione di una sosta l'utente avrà una fase di *Riepilogo* in cui potrà ricontrollare tutte le opzioni scelte per la sosta: orario di ingresso, se definire o no l'orario di uscita, il veicolo da parcheggiare e il metodo di pagamento da usare.

Sarà inoltre presente una sezione dedicata puramente agli *Amministratori* da cui si potrà visualizzare lo stato attuale del sistema nonché grafici dedicati per l'analisi e lo studio dell'occupazione dei parcheggi, la fidelizzazione cliente e le entrate giornaliere.

Gli Amministratori potranno inoltre lavorare direttamente con la banca dati per quanto riguarda:

- Aggiungere, modificare, eliminare punti di sosta
- Aggiungere, modificare, eliminare aree
- Nel caso di eliminazione area, indicare quale area (fra quelle esistenti) associare ai parcheggi rimasti orfani

Tutto ciò può essere più semplicemente riassunto utilizzando il seguente diagramma dei casi d'uso.

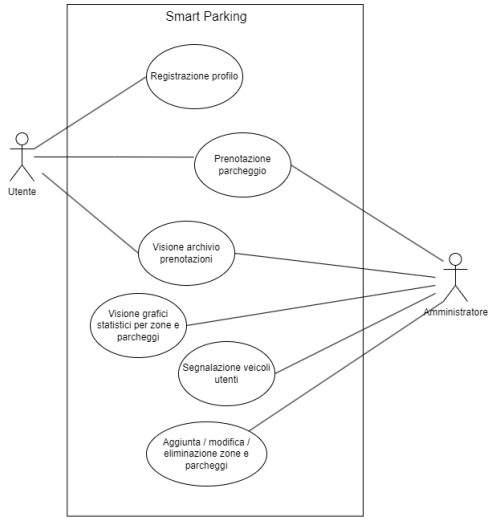

Diagramma dei casi d'uso

In tutto questo riveste un ruolo chiave garantire la consistenza in ogni pagina del sito, con un continuo aggiornamento dei dati la cui frequenza sia calibrata in base al contesto, cosi da evitare di sovraccaricare di richieste i server: si passerà, quindi, da aggiornamenti a cadenza lenta nella dashboard amministratori (dove per l'analisi e la modifica delle informazioni non servono gli ultimi dati disponibili) alla pagina prenotazioni in cui l'utente sarà avvertito in tempo reale dei cambiamenti di stato del parcheggio di interesse.

0.3 Design

Si è deciso di adottare una architettura client-server avendo scelto come focus lo sviluppo di una Web App da eseguire tramite browser. L'entità principali saranno:

- Client
- Server NodeJS
- Server Socket.IO

- Database MongoDB

Questo design permette di avere a che fare con una architettura modulare, rendendo più semplice la gestione di errori e future implementazioni mirate.

0.3.1 Struttura

Andando più nei dettagli della struttura del sistema, possiamo schematizzarla col seguente diagramma di deployment

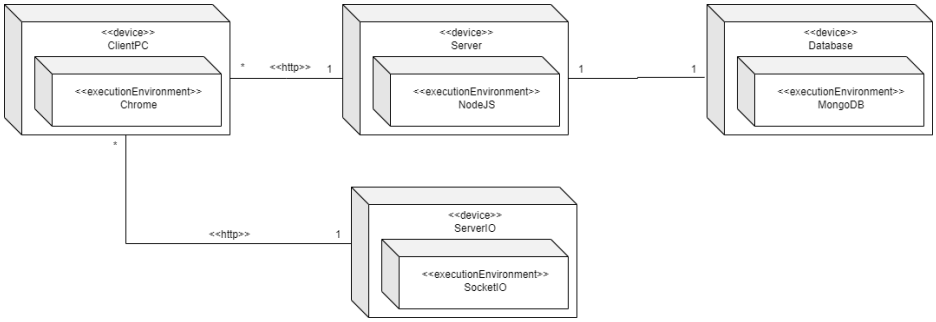

Diagramma di deployment

Come è evidenziato abbiamo che il client ha collegamenti con il server Socket.IO e con il server NodeJS tramite messaggi HTTP che siano GET o POST. A sua volta il server NodeJS ha un collegamento diretto col database di MongoDB per la gestione dei dati del sistema. Come già detto, tale struttura rende più modulare il sistema riuscendo a gestire separatamente il comportamento del server NodeJS ed il server Socket.IO. Andando più nei dettagli strutturali a livello di package, possiamo schematizzare i suddetti col seguente diagramma dei package.

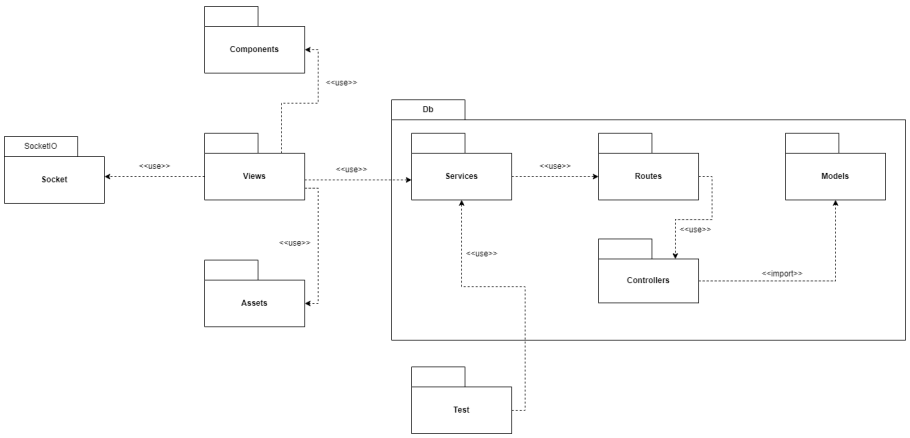

Diagramma dei package

Partiamo analizzando quest'ultimo diagramma andando a spiegare package per package cosa rappresentano.

Views

Si tratta del package principale dedicato al front-end del sistema. Al suo interno sono presenti i file Vue con la struttura grafica ed i comportamenti per gestire le interazioni con l'utente. Tali file saranno in grado di comunicare anche con i package di Socket.IO e quelli dedicati alle comunicazioni col database.

Components

Semplice package con all'interno file Vue creati per andare a modularizzare alcuni elementi grafici del sistema. Si è adottata tale filosofia sia per rendere più snelli i singoli file Vue nel package *Views* sia per una politica di riutilizzabilità di tali componenti in diverse zone del front-end.

Assets

Package con all'interno gli assets per le immagini, le icone ed altri dettagli grafici del front-end.

Socket

Package con all'interno il file contenente la struttura del server Socket.IO dedicato alla gestione real-time delle prenotazioni, per permettere una maggiore reattività e consistenza della fase di prenotazione di una sosta.

Services

Entriamo ora nel macro-package dedicato al server NodeJS con annesso collegamento al database. Il package *Services* contiene i file utili a gestire le chiamate HTTP GET e POST usati per ottenere o inviare informazioni al database. Tali file sono stati creati principalmente per andare a semplificare le chiamate utilizzando un sistema a classi.

Routes

Package dedicato alla gestione delle rotte delle chiamate GET e POST per indirizzarle verso il corretto metodo del database.

Controllers

Package con all'interno il vero cuore di comunicazione col database. Contiene diversi file per la gestione di ogni collezione del database, ognuno dei quali riceverà in ingresso la chiamata HTTP reindirizzata dalle route, definite nel package *Routes*, interrogherà il database e restituirà il risultato dell'interrogazione.

Models

Package con all'interno file per la schematizzazione delle collezioni del database. Strutture utili per una migliore gestione dei documenti all'interno del database.

Test

Package contenente i file necessari per l'esecuzione di test automatizzati per il controllo di alcune operazioni basilari.

0.3.2 Comportamento

Analizzando il comportamento del sistema andiamo a focalizzarci su come avvengono le due operazioni principali ossia:

1. Registrazione utente
2. Prenotazione parcheggio

I passaggi principali della registrazione di un nuovo utente possono essere rappresentati dal seguente diagramma delle attività.

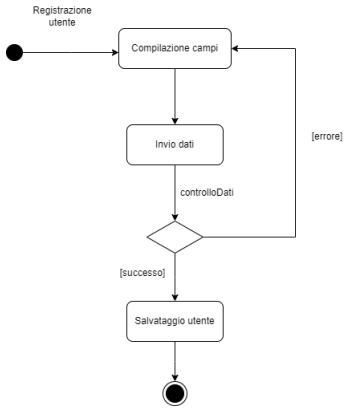

Diagramma di attività registrazione utente

Il client andrà a compilare i campi della form presentata nel front-end. Tali dati saranno passati tramite messaggi HTTP al database passando per le varie routes. Lato database avverranno i controlli, ossia la verifica sulla correttezza dei dati e su eventuali conflitti o incompatibilità con gli utenti attualmente salvati nel database. Se tali controlli avranno successo allora la registrazione andrà a buon fine, altrimenti sarà restituito un errore al client che ritornerà alla fase di compilazione dei campi.

Per quanto riguarda la gestione della prenotazione, essa risulta essere un pò più complessa.

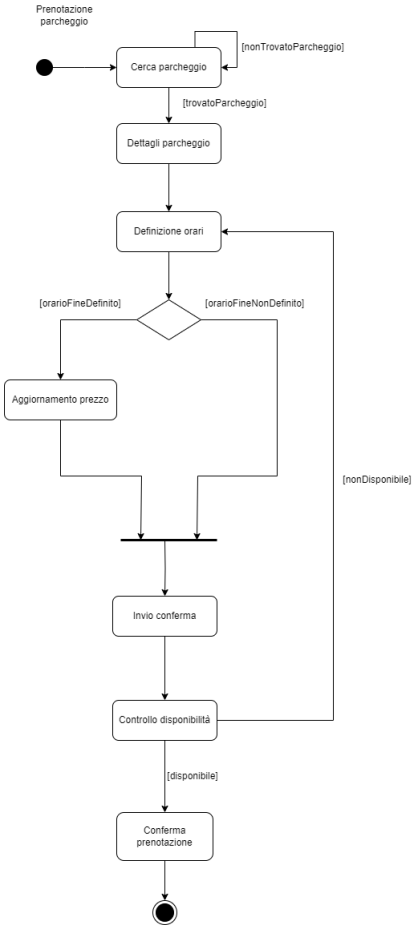

Diagramma di attività prenotazione parcheggio

Restando ad un livello più astratto per la spiegazione, possiamo mostrarla come segue (sarà meglio descritto nelle sotto sezioni 0.3.3 e 0.5.1). L'utente ricerca un parcheggio, una volta trovato entra nella sezione per effettuare la prenotazione. Tale sezione avrà un collegamento real-time con gli altri utenti in quel momento collegati alla medesima sezione del medesimo parcheggio. L'utente definisce gli orari, in tempo reale tutti gli altri utenti collegati allo stesso parcheggio riceveranno una notifica che li avvisa di questa operazione effettuata dall'utente. Una volta inviata la conferma della prenotazione avverrà la verifica dell'effettiva disponibilità. Nel caso che la verifica abbia esito positivo allora si procederà in base alla scelta della soluzione dell'orario di fine.

- Nel caso in cui l'utente avesse definito l'orario di termine della prenotazione, sarà calcolato il prezzo totale da pagare.
- In caso contrario allora la prenotazione terminerà con successo.

0.3.3 Interazioni

Continuando l'analisi dei due casi di studio precedenti, andremo ora ad analizzare più nel dettaglio i passaggi prima schematizzati astrattamente nei diagrammi delle attività.

Registrazione utenti

Partendo dal diagramma di sequenza relativo alle registrazioni di nuovi utenti.

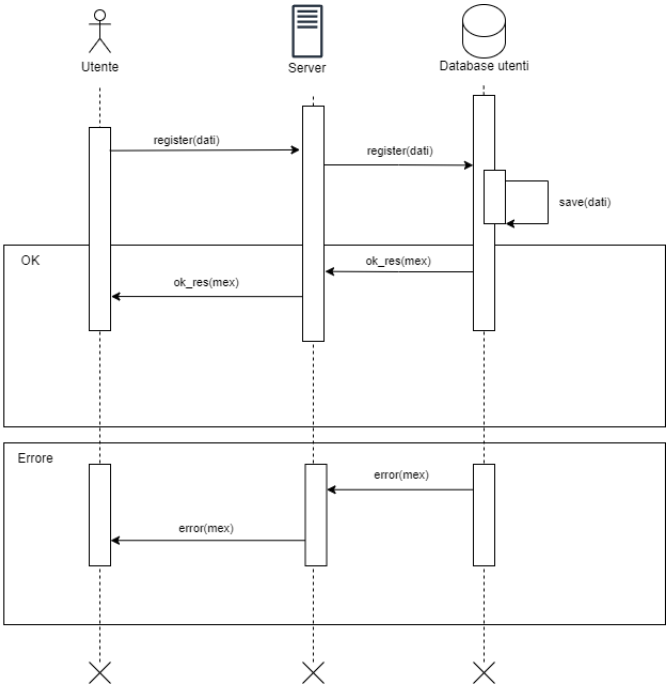

Diagramma di sequenza registrazione utente

Da questo diagramma notiamo che l'utente invia una richiesta di registrazione al server ponendo come argomenti i dati del form. Il server poi li reindirizza al database senza effettuare elaborazioni intermedie, ed è il database che col metodo *save(dati)* salva i dati nella collezione *users*. Delle domande potrebbero sorgere spontanee, ossia: *vengono effettuati dei controlli lato database per la registrazione dell'utente? Viene controllato che non vi sia già un utente registrato coi medesimi dati?* La risposta a queste domande è stata una nostra scelta implementativa; abbiamo deciso di porre come chiave univoca della collezione *users* l'email fornita al form. Pertanto sarà l'email che andrà a definire l'unicità di un utente. Sarà possibile avere n utenti omonimi, con la stessa data di nascita, stessa residenza ecc... Ma dovranno avere tutti email distinte, altrimenti in fase di registrazione verrà restituito un errore.

Prenotazione parcheggio

Anche in questo caso mostriamo il diagramma di sequenza relativo al processo di prenotazione di un parcheggio.

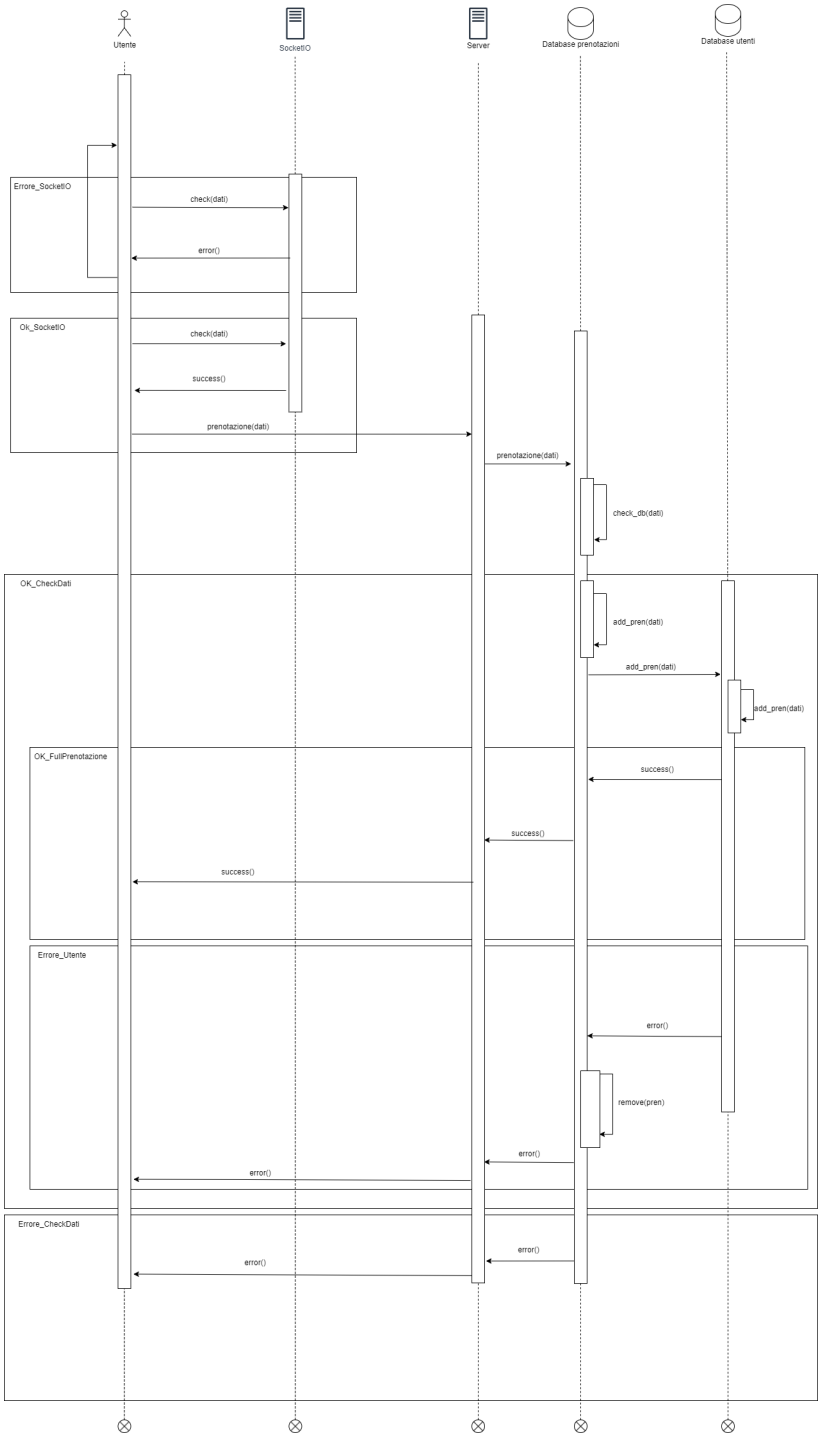

Diagramma di sequenza prenotazione parcheggio

Possiamo dividere tale passaggio in due macro settori:

- Controllo Socket.IO
- Controllo Database

Partendo dal primo settore, l'utente si trova nella pagina di prenotazione di un certo parcheggio. In questa pagina avviene una comunicazione real-time tramite Socket.IO con tutti gli utenti connessi al medesimo parcheggio. Ogni volta che un utente collegato definirà dei nuovi orari per la prenotazione, sarà inviata una notifica a tutti gli altri utenti. La verifica effettiva della disponibilità avverrà alla conferma della prenotazione. Se il parcheggio non è disponibile in tale orario, l'utente è tenuto a modificarlo, altrimenti può proseguire con la prenotazione, tale dettaglio è approfondito nella sezione 0.5.1.
Nel secondo settore l'utente invia la sua richiesta di prenotazione al server che viene reindirizzata al database. Sarà qui lato database delle prenotazioni che saranno effettuati ulteriori controlli come ad esempio:

- L'ID del parcheggio effettivamente esiste.
- Se è già presente una prenotazione per quel parcheggio con quegli orari.
- Se il veicolo selezionato è già stato utilizzato per un altro parcheggio nei medesimi orari.

In caso di successo di tutti i controlli allora la prenotazione sarà salvata nella collezione *Reservations* del database e, per una scelta implementativa dettata dalla necessità di consistenza ed integrità dei dati, anche nel documento relativo all'utente. Pertanto sarà inoltrata una richiesta di salvataggio di tale prenotazione anche alla collezione *Users*. Avendoli effettuato già lato Reservations, non saranno effettuati ulteriori controlli lato *Users*. Soltanto nel caso di errori critici durante l'aggiornamento dell'utente lato database sarà fatto un rollback anche nella collezione *Reservations* andando ad annullare tale prenotazione e restituendo un messaggio di errore lato front-end.

0.4 Tecnologie

Per quanto riguarda le tecnologie utilizzate è stato adottato lo stack MEAN, sostituendo però Angular con Vue, preferendolo per la sua maggiore semplicità e scalabilità. Per il resto si è fatto largo uso soprattutto della piattaforma maptiler per la gestione della mappa e PrimeVue per velocizzare lo sviluppo delle componenti (e ridurre la quantità di codice).
Più in generale si avranno:

- **Mongoose**: per semplificare l'interazione con MongoDB.
- **Maptiler**: scelto per la sua compatibilità con vue3 che permette di utilizzarlo come libreria nativa. Poi ci si limita ad importare dinamicamente

alcuni fogli di stile e script aggiuntivi e ad utilizzare l'apikey gratuito per richiedere il css della mappa e il reverse geocoding delle coordinate (cosi da ottenere dinamicamente gli indirizzi dei vari punti di sosta).

- **PrimeVue**: set di components open source per Vue, che ci è stato molto utile nello sviluppo del sito, permettendoci di velocizzare la creazione di buttons, charts, pannels, cards, sidebars, calendars, dialogs etc. Si può dire che l' 80% del front-end derivi da tale set. In questo modo ci si è potuti concentrare sul back-end e sullo sviluppo delle meccaniche davvero importanti.

- **Twitter Bootstrap**: framework front-end molto utile per organizzare il layout di una pagina, ragionando coi concetti di container, colonne e righe. Molto utile anche per quanto riguarda semplici componenti come Bottoni, Form, Menù, etc. Insieme a PrimeVue è il principale mezzo utilizzato lato design Front-end per garantire un layout responsive.

- **Datiopen.it**: dataset strutturato utile per ricavare id e coordinate geografiche di tutti i parcheggi chiusi di Italia. Quindi per la simulazione si utilizza il nostro paese come ipotetica smart-city e i parcheggi coperti come punti sosta.

- **Socket.IO**: essenziale per poter gestire le comunicazioni client e server e viceversa nella sezione prenotazione. Un approfondimento si avrà nella sezione 0.5.4

- **Docker**: strumento nato per la gestione della fase di deployment, fondamentale nel nostro caso per permettere un testing in uno scenario ipoteticamente multi-piattaforma. Ulteriori dettagli nel capitolo 0.7.

- **Jest**: strumento utilizzato per un basilare meccanismo di testing automatizzato. Scelto per la sua immediatezza e il suo basso coefficiente di complessità.

Per quanto riguarda tecnologie secondarie, una rapida carrellata:

- **Swiper**: libreria front-end utile per creare gallerie di slide "swipeabili" sia da desktop che da mobile. Scelta per migliorare la user-experience.

- **VueEx**: libreria per la gestione e l'utilizzo di dati persistenti all'interno di un progetto. In Smart Parking utilizzato per migliorare la gestione di dati persistenti relativi all'utente loggato.

- **Emailjs**: libreria che fornisce le meccaniche per la creazione di un template e l'invio di email, utile per le segnalazioni di inconvenienti, avvisi e suggerimenti da e verso gli utenti. Utilizzata per migliorare il rapporto e le interazioni con gli utenti.

0.5 Dettagli distribuiti

Come è stato riportato sopra, per costruzione del progetto si è deciso di optare per una soluzione client-server centralizzata. Più entità quindi potranno accedere e modificare dati comuni e per questo è essenziale che tutti vedano sempre dati consistenti e aggiornati. Sfortunatamente le tecnologie scelte non garantiscono sempre il rispetto di questi vincoli, quindi si è dovuti intervenire più volte per compensare le diverse mancanze riscontrate.

0.5.1 Prenotazioni

Uno degli aspetti più critici lo abbiamo nella gestione delle prenotazioni. L'uso di un server centrale permette di canalizzare le attività e creare una struttura sequenziale, ma lavorando sulla rete c'è sempre il rischio di prendere decisioni su dati non aggiornati all'ultima versione. In particolare la permanenza dell'utente nella pagina di prenotazioni potrebbe creare delle incongruenze nel caso in cui altre entità (che siano utenti o amministratori) intervengano sul medesimo parcheggio. In questa pagina, infatti, i dati vengono passati solo in fase di apertura, alcuni attraverso le props (id parcheggio, indirizzo e l'orario di inizio scelto) e altri con tre diverse richieste axios al backend:

- Con le prime due si ottengono prima l'area e poi il prezzo per ora del parcheggio.
- Con la terza, invece, si verifica se il parcheggio è veramente libero nell'orario indicato dall'utente e si stabilisce l'intervallo di tempo entro cui si potrà prenotare. Infatti, anche se nella home il parcheggio è indicato come libero non c'è alcuna garanzia che rimanga tale durante il passaggio alla pagina di prenotazione. Per evitare di far perdere tempo all'utente, quindi, si preferisce controllare fin da subito se la situazione non è cambiata. In ogni caso sarà fatto un ulteriore check dal server di backend in fase di creazione della prenotazione.

Successivamente i dati non verranno più aggiornati e l'utente si troverà in una sorta di bolla statica. In particolare saranno tre gli scenari problematici che deriveranno da ciò:

Un amministratore cambia il prezzo del parcheggio di interesse

Ogni area avrà il suo prezzo. Di conseguenza cambiando l'area di un parcheggio se ne modifica anche il costo. In questo caso si possono verificare due situazioni diverse:

- Se l'area è cambiata in fase di apertura della pagina di prenotazioni, quest'ultima riporterà l'aggiornamento
- Se, invece, il cambiamento avverrà in una fase successiva si procederà con la prenotazione con il vecchio prezzo.

Il motivo di questa differenziazione si deve alla presenza in *reservationView* di una casella che riporta la stima del prezzo finale. Tale costo si aggiornerà ogni qual volta l'utente modificherà l'intervallo di tempo della prenotazione permettendo di tenere traccia di quanto dovrà successivamente pagare. Per evitare di vanificare la cosa si è deciso di limitare il più possibile l'influenza di agenti esterni sul preventivo finale se non nei casi indispensabili, ovvero quello della prenotazione del parcheggio da parte di terzi nelle stesse fasce orarie a cui la stima si riferisce. In quel caso l'intervallo indicato dall'utente sarà automaticamente modificato e con questo anche il costo riportato nella sopracitata cella. Anche dalla conferma della prenotazione in poi anche andando a cambiare l'area del parcheggio, l'utente si troverà a pagare il prezzo concordato in fase di prenotazione.

Un amministratore elimina il parcheggio di interesse

Nel caso in cui un amministratore elimini un parcheggio su cui un utente sta ultimando una prenotazione, quest'ultimo non sarà avvertito se non alla conferma della stessa, quando sarà invocato il metodo **addNewReservation()** del **ReservationDataService** e la richiesta giungerà al backend. Si è, infatti, deciso di seguire un ragionamento ancora più stringente al caso precedente visto che la situazione presa in esame dovrebbe verificarsi molto raramente: un parcheggio non dovrebbe essere quasi mai eliminato se non in casi eccezionali. Di conseguenza la mancanza di ulteriori sistemi di sicurezza non dovrebbe incidere più di tanto sulla user-experince.

Più utenti che prenotano sullo stesso parcheggio

A differenza delle precedenti due situazioni questo è il caso che si dovrebbe verificare con più frequenza: dato un certo parcheggio libero a partire da un certo orario c'è una certa probabilità che più utenti siano interessati alla sua prenotazione. Come detto già sopra, una volta entrati nella pagina delle prenotazioni si viene isolati e non si hanno più informazioni sul mondo esterno. Nel caso in cui due utenti siano interessati allo stesso parcheggio nello stesso orario (o comunque a prenotare in intervalli con intersezione diversa da 0) chi prenota per secondo riceverà un messaggio di errore, in quanto il parcheggio sarà già stato prenotato dal primo. Sebbene questo sia inevitabile, si è voluto creare qualcosa di più interattivo, in cui i singoli utenti fossero avvisati in tempo reale dei cambiamenti di stato del punto sosta di interesse. E' proprio in questo contesto che entra in gioco la libreria Socket.IO: grazie ad essa un server può contattare un client senza che questo debba prima inviare una richiesta specifica. Nel caso in esame si avrà il seguente walkthrough:

1. Un utente individua il parcheggio di interesse X nella home (muovendosi liberamente o sfruttando la ricerca rapida), verifica il suo stato in base al colore del marker e alle informazioni del menu laterale e, nel caso in cui sia libero, può premere il tasto "prenota"

2. Si apre quindi la pagina prenotazioni. Per prima cosa si ricavano le informazioni di X e si verifica se sia ancora disponibile.

3. Se è tutto okay, si crea un client socket e ci si connette al server socket (che sarà approfondito nella sottosezione 0.5.4).

4. il client socket invia un messaggio di init al server, con la quale si "registra" all'id del parcheggio X.

5. ogni volta che un altro utente occupa o libera (sia nel presente che in futuro) X, il client socket sarà avvertito di tale cambiamento.

6. Ora, prima di cliccare nel tasto prenota e di passare dalla pagina home a quella di prenotazione, viene sempre indicato un orario specifico. Questo sarà inizialmente uguale al tempo attuale, ma potrà essere manualmente spostato in avanti per verificare la disponibilità dei punti sosta in un certo futuro. In ogni caso tale tempo, che chiameremo T e che potrà quindi essere pari o al tempo attuale o ad un orario futuro indicato dall'utente, sarà trasmesso anche alla pagina *reservationView*: all'apertura di questa, come già detto, si andrà a verificare la presenza o meno di altre prenotazioni usando proprio T. Questo serve non solo a indicare all'utente se nel frattempo X è stato occupato, ma anche a delimitare l'intervallo di tempo in cui potrà muoversi. In particolare partendo da T l'utente potrà indicare un orario che va dal tempo massimo fra il tempo attuale e il tempo di liberazione della prenotazione passata più vicina, al tempo di inizio della prenotazione successiva, se è presente una prenotazione successiva su X, o indeterminato in caso contrario. Si avranno quindi due intervalli: uno che indica i limiti di tempo entro cui muoversi e uno che definisce gli orario di ingresso e uscita della prenotazione. Ogni volta che un altro utente prenota o libera X, il client socket invierà gli orari della liberazione o dell'occupazione, che definiranno un nuovo intervallo I. Fatto questo si proseguirà nel seguente modo:

    - Se l'intersezione fra I e l'intervallo massimo in cui l'utente può muoversi è pari a 0, quindi non vi sono sovrapposizioni, si ignorerà semplicemente l'evento.
    - Se, invece, è diversa da 0 si avvertirà l'utente con un toast di warning nel caso di una prenotazione e di informazione nel caso di liberazione. Inoltre si dovranno aggiornare le date selezionabili dall'utente e disattivare o attivare la possibilità di inserire la fine indefinita in base alla presenza o meno di una prenotazione futura.
    - Se, inoltre, non solo l'intersezione con l'intervallo massimo è diversa da 0, ma anche quella fra T e l'intervallo di tempo indicato dall'utente, trattandosi per forza di un'occupazione su X si cambierà la **severiry** del toast in error. Similmente a come è avvenuto nel caso precedente, inoltre, si dovrà cambierà in automatico l'intervallo di tempo definito dell'utente, adattandolo alla nuova disponibilità.

7. Dopo aver indicato sia il veicolo che il metodo di pagamento da utilizzare l'utente potrà cliccare sul tasto conferma. Sarà controllato, quindi, se l'intervallo di tempo definito è di almeno un minuto e se la targa inserita non sia già usata in una prenotazione con sovrapposizione di orario.

8. Se è ancora tutto okay si procederà con l'invio della richiesta di prenotazione al sever socket e ci si metterà in attesa.

9. Durante questa attesa se dovesse capitare che un altro utente nel frattempo richieda il parcheggio X in una fascia d'orario sovrapposta (e la richiesta di quest'ultimo giunge per prima al server socket), il client sarà informato e si andranno ad aggiornare le date indicabili. Successivamente, quando, sarà presa in carico la richiesta di prenotazione del nostro utente dal server essa sarà declinata. L'utente sarà quindi informato e dovrà decidere se prenotare comunque con i nuovi tempi (sempre se l'intervallo di tempo rimasto sia maggiore di un minuto) o abbandonare la pagina.

10. Se, invece, l'intervallo di tempo indicato dall'utente non sarà toccato e la sua richiesta andrà a buon fine, sarà inviato un messaggio di conferma al socket client associato e uno di occupazione a tutti gli altri registrati sull'id di X.

11. A questo punto si passerà all'invocazione del metodo *addNewReservation* del *ReservationDataService* . Qui entra in gioco Express, che attraverso un meccanismo di routing, indirizza la chiamata POST generata dal suddetto metodo ad un controller (*reservation.controller*) che sfruttando i servizi forniti da mongoose come model (rappresentazione della struttura del documento in MongoDB) e metodi essenziali quali (save, findOne, findOneAndUpdate, ecc..) ci permettono di interagire col DB.

12. Se la richiesta va a buon fine l'utente sarà spostato alla pagina di prenotazioni attive. In caso contrario (i motivi del fallimento si vedranno dopo nella sezione 0.5.3), sarà invece inviata una richiesta di liberazione al server socket cosi da avvertire tutti gli altri utenti interessati a X.

Un'ultima cosa da segnalare è che il backend al momento del ricevimento della richiesta della prenotazione farà nuovamente una serie di controlli. Di questi molti sono ridondanti (si fanno solo per sicurezza), ma uno in particolare è importante: un nuovo controllo della targa. Infatti fra il primo check del veicolo e questa fase non c'è alcuna garanzia che nel frattempo la stessa targa non sia stata usata per un'altra prenotazione sovrapposta. Bisogna considerare che non c'è alcun controllo né sulla registrazione dello stesso veicolo da parte di più utenti, né sulla presenza di più sessioni contemporanee loggate con lo stesso account, quindi è fondamentale eseguire questa seconda verifica per garantire l'integrità.

Come si è visto il client prima fa una richiesta al server socket per verificare la possibilità di occupare un determinato punto sosta, e successivamente, dopo aver ricevuto l'okay, genera una richiesta POST al backend per ultimare la

prenotazione. Anche se si sarebbe potuto far fare quest'ultimo passaggio al server socket stesso, cosi da ridurre le richieste fatte dal client e avere meno rimpalli, non si è fatto per scelte puramente progettuali: nonostante alle fine non sia stato implementato, sarebbe stato interessante far apparire una dialog al momento della risposta positiva da parte del server socket, cosi che l'utente avesse potuto confermare per un'ultima volta la volontà di prenotare il parcheggio. Se, infatti, alla prima conferma non è garantito che la transazione vada a buon termine, dopo che è stata ricevuto l'okay dal server socket, salvo nei casi particolari in cui si usa la stessa targa per prenotazioni sovrapposte, quel determinato parcheggio sarà già stato riservato a quell'utente in quel determinato intervallo. Passando quindi da una fase di possibile prenotazione (in cui si fa una sorta di proposta) alla certezza di avere quel posto, sarebbe utile far esprimere all'utente un'ultima volontà. A quel punto, invece che mandare una seconda richiesta al server socket meglio bypassarla e lanciare direttamente la richiesta post.

0.5.2 Altri dati aggiornati nel tempo

Oltre quella di prenotazione, nella piattaforma sono presenti altre due pagine i cui dati riportati vengono aggiornati in "tempo reale" durante la visualizzazione: la dashboard amministratore e la Home. In questo caso non si fa uso di websocket, ma si preferisce eseguire richieste client-server cicliche con diversa granularità (si è scelto un timer pari a cinque secondi nella home e a un minuto nella sezione amministratore). In entrambi i casi la logica è la stessa: si fa partire un timer asincrono che allo scoccare avvia una richiesta GET al backend e si riavvia. Di volta in volta viene anche definito l'handler attraverso cui, una volta ottenuti, i dati vengono passati alle componenti figlie per essere visualizzati. Il timer sarà fatto partire nella fase *mounted* e sarà bloccato in *unmounted* (in pratica si definisce un booleano che all'uscita della pagina sarà settato a true. Una volta scaduto un timer, quello successivo sarà generato solo se tale booleano è su false). La scelta di non usare anche qui delle socket, ma un vero e proprio polling ciclico si deve al tentativo di evitare congestioni del server e alla non necessità di avere sempre le ultime informazioni aggiornate. Cosi facendo inoltre è possibile avere una stima del carico di lavoro del server per ogni utente connesso e regolarsi di conseguenza. I contro dipendono dal fatto che i dati cambino o meno:

- Nel caso in cui non cambino si fa lavorare il server e si consuma banda inutilmente
- Nel caso in cui ci siano aggiornamenti, invece, potrebbero verificarsi dei problemi di integrità per il delay dato dai singoli timer.

Per capire il motivo per cui si è comunque deciso di lavorare in questo modo, conviene analizzare singolarmente le due pagine:

- La home è la pagina in cui ci si aspetta che l'utente passi più tempo. Se si dovesse aggiornare ogni qual volta cambi lo stato di un parcheggio si avrebbe un grosso vantaggio nei momenti in cui gli utenti si limitano a

monitorare gli stati dei vari parcheggi e si rischia la congestione del caso opposto. Con l'adozione dell'aggiornamento ciclico, come detto sopra, si ha il pro di avere un carico di lavoro costante e, quindi, anche stimabile. In aggiunta anche nel caso in cui un utente dovesse provare a prenotare un parcheggio durante l'intervallo di tempo fra un aggiornamento e un altro, non vi sarebbero problemi visto che una volta aperta la pagina di prenotazione vengono richiesti gli ultimi dati relativi a quel parcheggio. Quindi nel peggiore dei casi scoprirà solo in un secondo momento che il parcheggio risulta già occupato, ma non verrà minimamente intaccata l'integrità del sistema (anche se si perde la consistenza).

- Per la dashboard amministratore la situazione è diversa. In questo caso si prevede che gli utenti siano pochi, se non solo uno. Come anticipato essa si divide in due porzioni: la prima è puramente statistica e lavora sullo storico dei dati, quindi non è necessario che siano aggiornati in tempo reale; la seconda serve alla modifica dei dati o alla segnalazione degli utenti, tutte funzioni che dovrebbero essere usate raramente considerando la natura del progetto. Per questo motivo si sarebbe anche potuto evitare il timer, ma si è preferito comunque inserire un minimo di dinamicità e da qui tempo di aggiornamento abbastanza lungo e pari a un minuto. Se più amministratori dovessero modificare, aggiungere o eliminare aree o parcheggi in contemporanea si potrebbero avere problemi di consistenza fra i dati visualizzati dai singoli, ma ancora una volta, grazie alla struttura sequenziale del server, non si creeranno incongruenze all'interno di MongoDB. Nel peggiore dei casi un secondo admin non vedrebbe un parcheggio/area già aggiunto (se prova ad aggiungerlo nuovamente riceverà un errore per l'id duplicato), ne eliminerebbe uno già eliminato (che non sarebbe trovato nel database, che a sua volta non farebbe nulla) o modificherebbe un parcheggio o un'area basandosi su dati vecchi (se si sono messi d'accordo su cosa cambiare alla fine il risultato sarà lo stesso e si farebbe solo un update di troppo).

Quindi, tirando le somme, per tutti questi motivi si è visto nella scelta del pooling la soluzione migliore. Per concludere è importante segnalare che ci sarebbero altre pagine che nonostante siano statiche mostrano dati che potrebbero cambiare durante la visualizzazione. Un esempio è la pagina di prenotazioni attive: ipotizzando di lavorare con lo stesso utente su due sessioni diverse, se da una parte ci si trova nella pagina prenotazioni e dall'altra in quella delle prenotazioni attive, ci si aspetterebbe che al momento del click del tasto conferma nella prima, la nuova prenotazione appaia nella seconda. Cosi non sarà (a meno che non si refresha manualmente) per il semplice fatto che pur non impedendo l'uso dello stesso account per più sessioni, non è qualcosa che si vuole incentivare e quindi non è stato dedicato del tempo per la realizzazione di meccanismi ad hoc. Allo stesso modo se si aprono due pagine prenotazioni attive, con lo stesso utente, e si cancella o termina una prenotazione da una parte non lo si vedrà dall'altra se non dopo un refresh manuale.

0.5.3 MongoDB

Per quanto utile e versatile, in un database NoSQL come MongoDB mancano due aspetti che sarebbero stati abbastanza utili al nostro progetto: chiavi esterne e trigger temporali. Di conseguenza per garantire che non si lavori su dati vecchi e/o inesatti si è realizzato del codice ad hoc.

"Chiavi esterne"

Per quanto riguarda l'adattamento del concetto di "chiavi esterne", si è scelto di limitarsi a fare i controlli lato client, tranne in tre casi:

- *Creazione di una nuova prenotazione*: come si è visto nella sezione 0.3.3, si verifica prima se esiste ancora il parcheggio a cui fa riferimento.
- *Associazione parcheggio - area*: Durante la creazione si verifica che l'area associata sia effettivamente presente nel database (magari un altro admin l'ha eliminata fra l'invio della richiesta di creazione e la sua ricevuta). Se cosi non fosse non si potrebbero ricavare i prezzi dei nuovi parcheggi. La stessa verifica verrà fatta anche al momento della modifica dell'area associata ad un certo parcheggio, cosa che può avvenire sia quando si vuole aggiornare un singolo parcheggio, sia quando si elimina del tutto un'area. In quest'ultimo caso, infatti, si chiede all'admin di indicare una nuova area (fra quelle esistenti) a cui associare i parcheggi rimasti senza. Quando successivamente sarà eseguita la richiesta, sempre per questioni di integrità, prima si aggiorneranno tutti i parcheggi appartenenti alla vecchia area e poi, se non sono sorti errori, quest'ultima sarà eliminata.
- *Cancellazione di un punto sosta*: al momento della cancellazione di un parcheggio saranno impedite prenotazioni future e terminate quelle attive (ovviamente sempre se presenti). Sarà quindi restituita la lista di tutte le *reservations* modificate, cosi da poter informare, tramite email, tutti gli utenti coinvolti.

Per tutte le altre "chiavi esterne" si è deciso di non dedicare troppo tempo alla creazione di controlli di sicurezza, visto che una mancanza di integrità non provocherebbe danni economici. E' lo stesso motivo per cui non vi saranno verifiche (questa volta nemmeno lato client) sulle date di nascita indicate o sull'anno del veicolo: la conseguenza è che si potrebbero registrare dati falsi, ma non ci sarebbero ripercussioni sul corretto funzionamento della piattaforma. L'unico caso borderline è il controllo sulla targa durante la creazione di un nuovo parcheggio: si controlla solo se non sia già stata usata per soste prenotate nello stesso intervallo di tempo di quello della prenotazione attuale, non che esista davvero. Lato client l'utente potrà selezionare solo i veicoli già registrati e caricati all'apertura della pagina dal database, ma nulla vieta di eliminarli fra questa fase e la conferma della prenotazione. Allo stesso modo quando sarà eliminato un veicolo non saranno toccate le prenotazioni ad esso associate. Alla fine, infatti, l'indicazione della targa è solo un modo per limitare il numero di prenotazioni

sovrapposte fatte da uno stesso utente (che non avrà di certo macchine infinite) e una stessa macchina/moto, ma a conti fatti ci si immagina che si dovrà guardare il veicolo che fisicamente occuperà il parcheggio allo scoccare dell'ora di inizio della prenotazione più che quello indicato nella prenotazione. Magari in futuro si potrebbe approfondire meglio quest'aspetto.

Creazione nuova prenotazione

Come già discusso precedentemente al momento della creazione di una nuova prenotazione saranno fatti una serie di controlli sequenziali per verificare che sia tutto apposto. La gestione delle prenotazioni, infatti, riveste un ruolo chiave, rappresentando il cuore del progetto.

1. Per prima cosa si verifica che il parcheggio esista, in caso negativo si ritorna errore.
2. Successivamente si fa il sopracitato controllo sulla targa.
3. Per ultimo si cercano altre prenotazioni sullo stesso parcheggio in orari sovrapponibili a quello richiesto.

Nel caso, in pratica certo grazie al server socket, che la prenotazione richiesta non si sovrapponga con altre occupazioni si può finalmente procedere al salvataggio del nuovo record. Per poter accedere con più facilità alle informazioni si è deciso di creare una ridondanza, mettendo i dati relativi alle prenotazioni sia in un array nei singoli utenti che in una collezione MongoDB apposita. E' quindi fondamentale che il salvataggio riesca in entrambe le posizioni o in nessuna delle due. Per questo si lavorerà ancora una volta sequenzialmente, salvando prima la nuova prenotazione "nell'omonima" collezione e poi nell'array utente. Se durante il caricamento in quest'ultimo si manifestano dei problemi, il nuovo record sarà eliminato anche dalla collezione *reservations* cosi da garantire ancora una volta l'integrità.

Conclusione prenotazioni

Le prenotazioni possono concludersi in due modi:

1. Scadenza automatica causa orario
2. Interruzione da parte dell'utente

Partiamo dal punto 1. Sappiamo che MongoDB non ha alcuna funzione di trigger temporale che permetta una verifica periodica dei dati. Pertanto si è deciso di adottare una politica efficiente dal punto di vista computazionale a discapito di una organizzazione migliore lato codice. L'idea è la seguente: ogni volta che si richiedono le informazioni dalla collezione *reservations* viene fatto un controllo sulle singole prenotazioni. Le casistiche sono:

- Se la prenotazione non è ancora scaduta allora non viene modificata e permane allo stato attuale.

- Se la prenotazione è scaduta viene rimossa dalla collezione *reservation*, ma non dalla lista di prenotazioni attive dell'utente.

Il secondo caso per forza di cose andrà a rompere il concetto di ridondanza che abbiamo descritto più volte in precedenza, ma perchè abbiamo preso questa decisione?
Si è deciso di preferire il rendere cosciente l'utente mostrandogli nella relativa pagina quali prenotazioni deve ancora terminare. Cosi facendo si potrebbe anche imporre un prezzo maggiorato per non aver rispettato l'orario di uscita o evitare pagamenti non riusciti nel caso di richiesta di autorizzazioni da parte dei gestori delle carte (o PayPal). Starà poi all'utente manualmente rimuovere la prenotazione scaduta dalla pagina delle prenotazioni attive, andando ad avviare il meccanismo di messa in archivio.

Passiamo al punto 2. Consideriamo il caso che un utente decida autonomamente di interrompere una prenotazione attiva, che si tratti di fine definita o indefinita. In questo caso ci si limiterà a trasferirla dalla lista delle prenotazioni attive all'archivio che ogni utente ha. Sarebbe superfluo rimuovere la prenotazione dalla collezione *reservation*, visto che si aggiornerà in automatico non appena qualcuno farà una richiesta GET.

0.5.4 Socket server

Dopo averlo nominato più volte è importante parlare del socket server. Esso si configura come un modulo aggiuntivo, con cui entrano in connessione i client nella pagina di prenotazione e in quella di prenotazioni attive (ogni volta che l'utente interrompe una prenotazione sarà informato il server socket che propagherà la notizia a tutti i client socket interessati a quel punto sosta).

Reservation_data

Il cuore del server è una struttura dati *reservation_data* che mappa sull'id dei singoli parcheggi una lista di clients e di prenotazioni. La prima tiene gli id di tutte le client socket attive registrate su quel punto sosta, la seconda le prenotazioni attive fatte su quel parcheggio fino a quel momento. Per evitare che la mappa si estenda all'infinito ogni volta che la lista delle sockets di un determinato parcheggio si svuota, il campo di quel punto sosta viene eliminato andando a cancellare anche le prenotazioni ancora presenti. Il server socket, infatti, ha come unico scopo quello di aggiornare i clients che, avendo caricato la pagina di prenotazione, sono fermi alle informazioni acquisite in fase di apertura della stessa. Di conseguenza è sufficiente che siano salvate solo le prenotazioni fatte dopo che il primo utente abbia aperto tale pagina, visto che tutte quelle precedenti saranno già state acquisite (andando a modificare l'intervallo di tempo massimo in cui un'utente può muoversi). Quando, poi, tutti i clients interessati ad un certo punto sosta si sono scollegati dalla socket, non ha più senso continuare a tenere i dati in memoria e si può procedere all'eliminazione del campo.

Funzionamento

Come con il backend anche in questo caso si avrà una struttura sequenziale: un thread unico che si occuperà di gestire tutte le richieste dei singoli client (alla fine sono tutti lavori computazionalmente leggeri, quindi non si dovrebbe avere un collo di bottiglia). Gli eventi gestibili dalla socket sono i seguenti:

- '*connection*': si verifica quando un client socket si connette portando alla definizione degli handler degli eventi '*delete*' e '*init*'
- '*delete*': permette alla client socket di trasmettere al server una prenotazione da eliminare. Dopo l'eliminazione, si propagherà un broadcast a tutte le altre clients socket registrate al punto sosta che si è appena liberato. Una cosa importante è che, nel caso di interruzione di una prenotazione, i client genereranno l'evento '*delete*' solo dopo che il backend abbia correttamente archiviato la prenotazione associata.
- '*init*': serve al server per ricevere dal client appena connesso l'id del parcheggio a cui è interessato. Avvenuto ciò l'id della socket client sarà aggiunta alla lista delle sockets di quel determinato punto sosta e sarà definito un handler per l'evento '*parkingId*' (dove parkingId è uguale all'id del parcheggio sopracitato).
- '*parkingId*': generato dalla socket client quando l'utente clicca sul pulsante di conferma prenotazione, porta con se l'orario di inizio e di fine. In questo modo è possibile confrontare questo nuovo intervallo di tempo con quelli già salvati in *reservation_data* (relativi allo stesso punto sosta) e verificare se ci siano sovrapposizioni o no. In entrambi i casi la socket client generatrice sarà informata con l'evento 'reservationResponse' dell'esito di tale controllo, mentre sarà mandato un broadcast a tutte le altre solo nel secondo caso.
- '*disconnect*': si attiva quando una client socket si disconnette. Nell'handler associato sarà controllato se per il parcheggio di interesse siano presenti altre sockets connesse, in caso contrario il campo relativo al suo id (del punto sosta) sarà eliminato.

0.5.5 Maptiler

Come è già stato scritto per la visualizzazione della mappa è stato scelto Maptiler, una libreria open source utilizzatissima in questo campo. L'accesso ai dati avviene attraverso REST API con apikey: ogni volta che servono verrà fatta la richiesta e sarà mostrata a schermo la risposta. Considerando la diffusione e l'importanza del servizio si è dato per scontato che tutte le richieste inviate ottengano una risposta positiva e per questo non si sono creati meccanismi di sicurezza ad hoc. Nel caso di malfunzionamento si avrebbero, però, due problemi:

1. Non avendo una cache interna, nella home non sarebbe mostrata nessuna mappa, ma si avrebbe un caricamento infinito. Nonostante questo il resto

del sito continuerebbe a funzionare senza problemi. La mappa, però, è una cosa puramente grafica, visto che tutti i dati sui parcheggi, le aree e le prenotazioni sono salvate su MongoDB: in futuro si potrebbe, quindi, pensare ad una visualizzazione di emergenza ad hoc per l'accesso e la scelta dei singoli parcheggi (magari anche solo una semplice tabella con ricerca rapida, che riporti le informazioni principali e permetta di accedere alla pagina di prenotazioni).

2. Non sarebbe più possibile accedere agli indirizzi dei singoli parcheggi che poi, tra l'altro, vengono salvati anche in una campo specifico dei documenti prenotazione. Da questo punto di vista il fastidio è più che altro per l'utenza che perderebbe il riferimento sulla posizione esatta del posto auto prenotato, ma non influenzerebbe minimamente il corretto funzionamento della piattaforma.

0.5.6 Registrazione utente e ActiveReservationView

Prima di chiudere questa sezione ci si vuole soffermare velocemente su due piccoli aspetti comunque importanti.

- *Registrazione utente*: nel caso in cui due utenti dovessero registrarsi contemporaneamente con la stessa email, essendo il server sequenziale e visto che MongoDB non ammette chiavi duplicate (e le chiavi degli account utente sono le email appunto) sarà sempre respinta la richiesta presa in carico per seconda. La stessa cosa avverrà per tutte le altre richieste simultanee in conflitto.

- *ActiveReservationView*: nel sistema, oltre i due timer di cui si è parlato nella sotto-sezione 0.5.2, ne è presente anche un terzo che, però, non provoca la generazione delle richieste cicliche al backend: nella pagine di prenotazione attive ogni cards avrà un proprio timer che, ogni 60 secondi, aggiornerà il contatore dei minuti rimasti. Dopo che il contatore sarà sceso a 0, si marcherà la prenotazione come "scaduta". Si tratta di una cosa puramente grafica che non provoca alcuna ripercussione sul database (ne sulle scelte dell'utente che al massimo saranno portati a concludere una prenotazione qualche secondo prima o dopo la reale terminazione), quindi non è stato necessario definire un sistema di sincronizzazione fra il tempo del client e quello del server.

0.6 Test

Per quanto riguarda la fase di testing si sono adottare due politiche:

1. Un blocco di testing automatizzato

2. L'analisi di un particolare caso di studio

Trattandosi di una applicazione necessitante di un vasto bacino d'utenza per verificarne appieno le funzionalità su alti carichi di lavoro, si è deciso di bypassare il testing su larga scala causa impossibilità di mezzi. Il blocco di testing automatizzato è stato effettuato utilizzando la libreria Jest sviluppata appositamente per lo sviluppo di test automatizzati in ambito Javascript.
Nello specifico si è deciso di verificare il corretto funzionamento del meccanismo di registrazione di un utente e della prenotazione di un parcheggio a carico di tale utente. I dati utilizzati per la definizione di queste due entità sono puramente sperimentali, ma consoni con l'ambiente di utilizzo dell'applicazione. I dettagli implementativi sono visualizzabili all'interno del package *src/test*. Tale test avviene automaticamente all'avvio dello script docker all'interno di un container apposito come indicato nel capitolo 0.7. Il risultato di successo o meno di tale test è visualizzabile sia dalla console del container oppure visualizzando manualmente le collezioni MongoDB di *users* e *reservation*.
Piccola nota: all'interno del package *test* è stata inserita una versione modificata del *http-common* riguardante le comunicazioni con Axios. Tale modifica è stata necessaria per permettere la connessione tra i container Docker di test e backend.

L'analisi del caso di studio si è deciso di dedicarla allo scenario in cui un Utente decide di effettuare il logout mentre è nella schermata di Prenotazione e poi tenti di ritornarci scalando uno step indietro nella history del browser. In fase di caricamento di ogni pagina, nella fase *beforeCreate* viene effettuato un controllo sulla situazione di accesso o meno dell'utente. Se un utente non è loggato non potrà ovviamente accedere a determinate pagine quali ad esempio quelle delle Prenotazioni attive/Archivio. Il medesimo controllo avviene quindi anche quando si cerca di effettuare l'azione back descritta precedentemente. Concludendo, ha creato notevoli difficoltà il testing della correttezza dell'algoritmo che gestisce i grafici della sezione amministratore (la dashboard), essendo necessario la generazione di un certo quantitativo di dati.

0.7 Deployment

Il sistema possiede anche un meccanismo di deploy basato su Docker. Docker nasce proprio con lo scopo di automatizzare la distribuzione di applicazioni andando ad inventare il concetto di **container**, una sorta di contenitori leggeri e portabili che possono essere eseguiti su cloud o in locale. Entrando nei dettagli del nostro utilizzo di Docker possiamo identificare cinque container:

1. *smart_parking_db*
2. *smart_parking_backend*
3. *smart_parking_socket_io*
4. *smart_parking_frontend*
5. *smart_parking_test*

Come indicato dai nomi, essi riguardano rispettivamente:

1. Container per MongoDB
2. Container per Express
3. Container per Socket.IO
4. Container per VueJS
5. Container per testing Jest

Tutti in grado di comunicare ed interagire tra di loro grazie alle funzionalità fornite da Docker, permettendo quindi, avendo container separati, di avere maggiore modularità nel sistema.

Per avviare il sistema partendo da un sistema vergine i passaggi sono i seguenti:

1. Installare ed avviare sul proprio sistema il software Docker (non è necessario cambiare le impostazioni da quelle di default)
   `https://www.docker.com/`
2. Installare sul proprio sistema il software Git per poter effettuare la clone della repository in locale.
   `https://git-scm.com/`
3. Clonare la repository del progetto sul proprio sistema, utilizzando il terminale di Windows/Linux/Mac inserendo il comando *git clone* `https://dvcs.apice.unibo.it/pika-lab/courses/ds/projects/ds-project-bucchieri-bfini-git`
4. Posizionarsi nella cartella radice del progetto e sempre da terminale digitare il comando *docker-compose up -d –build*
5. Attendere finchè tutti i container non saranno avviati e *smart_parking_frontend* completi l'inizializzazione delle le librerie utilizzate.
6. Al termine sarà possibile collegarsi a *Smart Parking* digitando *localhost:8080* nella barra di ricerca di un qualsiasi browser.

Le informazioni di deployment sono presenti anche all'interno del file *README.md*.

0.8 Esempi di utilizzo

Immaginando uno scenario diverso da quello didattico, ossia l'ipotesi in cui Smart Parking sia effettivamente in produzione con il server in esecuzione h24, a cui l'utente può accedere banalmente attraverso un URL da browser; i possibili scenari di utilizzo sono svariati.

Un utente sapendo di doversi recare in una certa località il giorno X all'orario

Y può visualizzare la zona di suo interesse dalla mappa in Home, e dopo aver effettuato la registrazione / login potrà procedere con la fase di prenotazione indicando gli orari, il veicolo e il metodo di pagamento.

Un utente che riceverà un contenzioso potrà accedere alla sezione *Archivio* per ritrovare la prenotazione del parcheggio preso in atto ed effettuare un possibile reclamo.

Un utente che possiede due veicoli: V1 e V2, sa che il giorno X sarà tutta la giornata in una certa località con V1 e un'altra giornata sarà da altre parti con V2; Smart Parking gli permette di organizzare in anticipo la gestione della sosta.

Questi sono soltanto alcuni dei possibili esempi di utilizzo di Smart Parking, un sistema nato per semplificare scenari di vita quotidiana in cui la viabilità si pone come fonte di stress e alterazione.

0.9 Conclusioni

Traendo le conclusioni Smart Parking nasce come progetto didattico per scoprire ed imparare a maneggiare tecnologie come VueJS, Express, MongoDB, Socket.IO, Docker e Rest Api.
Inoltre, ha aiutato a gestire ed implementare scenari tipici degli ambienti distribuiti, andando a ragionare su come adottare politiche di scelte implementative ragionate appositamente per il nostro scenario di utilizzo. La decisione di applicare una ridondanza delle prenotazioni sia con una apposita collezione che con un campo dedicato per ogni utente per mantenere robustezza ed integrità, ne è un esempio.
La creazione di un sistema gestionale per un argomento molto attuale quale i parcheggi e la loro semplificazione. Aiutare gli automobilisti di oggi ad ottimizzare il loro tempo, sgombrare le menti dalle noie della continua ricerca di parcheggio nelle grandi aree urbane, semplificare i metodi di pagamento sempre più smart e veloci. Smart Parking partendo da questa versione embrionale, nel caso si proseguisse lo sviluppo, potrebbe portare alla creazione di una vera e propria piattaforma nazionale distribuibile su larga scala.

0.9.1 Future implementazioni

Immaginando la sopracitata situazione, si è pensato anche a possibili implementazioni future per arricchire e fornire un sistema in grado di gestire a 360 gradi il mondo delle soste. Alcune di esse possono essere:

1. Gestione di parcheggi a strisce bianche e autorimesse e differenziazione posti automobili e moto;

2. Usare i dati raccolti dal bacino di utenti per perfezionare il suggerimento del parcheggio in base all'orario e la destinazione scelta (magari indicando

quando sarebbe preferibile arrivare);

3. Possibilità di modificare l'orario definito nella prenotazione di un parcheggio;
4. Calcolo dei parcheggi più vicini utilizzando la distanza reale e non quella aerea.
5. Prezzi diversi in base alla fascia oraria
6. Possibilità per gli amministratori di lanciare eventi sconti a tempo
7. Mostrare tutti le possibili date in cui il parcheggio è disponibili nella pagina prenotazioni (e non solo un singolo intervallo di tempo)
8. Approfondire discorso sulle targhe delle auto
9. Vincolare la registrazione è l'utilizzo dei singoli mezzi
10. Visualizzazione di emergenza in caso di problemi con mapTiler
11. Dialog per ulteriore conferma della prenotazione, dopo l'okay della server socket

0.9.2 Cosa abbiamo imparato

Questo progetto ha arricchito senza dubbio il nostro bagaglio di competenze tecniche. L'aver sviluppato una Web App (Single Page Application) con sistemi quali Vue, Express, MongoDB e Socket.IO ha innalzato il nostro livello di competenze in tale ambito. Ci ha mostrato come alcune soluzioni sviluppate con questi strumenti risultino molto più efficienti e semplici rispetto alle implementazioni "classiche" viste in precedenza. Anche le implementazioni e le politiche da noi scelte autonomamente, errate o meno col senno di poi, ci hanno aiutato a formarci per imparare a migliorare nel caso vi ci ritrovassimo in progetti futuri. Vedere come cambia il pensiero dall'implementare una classica applicazione web con HTML, CSS e JS, al creare una Single Page Application vedendo pro e contro l'uno dell'altro è stato anch'esso motivo di stimolo e arricchimento formativo per progetti futuri. Per non parlare poi di tutti i nuovi dettagli e i casi limite a cui abbiamo dovuto fare attenzione per cercare di creare qualcosa di consistenze e interattivo.
Anche lato puramente organizzativo e gestionale è stato un progetto molto formativo, l'imparare a creare un'applicazione partendo dalla base (Obiettivi, analisi dei casi d'uso, scenari applicativi ecc...) passando poi al lato tecnico vero e proprio, alla fase di testing e deploy, concludendo con la stesura di un report finale riassuntivo di tutto ciò.