ChargeAndTrack

Applicazioni e Servizi Web

Bedeschi Federica - 0001175655 {federica.bedeschi4@studio.unibo.it}
Pracucci Filippo - 0001183555 {filippo.pracucci@studio.unibo.it}

05 Febbraio 2026

Chapter 1

Introduzione

Il progetto ha l'obiettivo di realizzare un'applicazione web per gestire e usufruire di parcheggi dotati di colonnine per la ricarica di auto elettriche. L'obiettivo è quello di fornire la possibilità di visualizzare tutte le colonnine presenti sul territorio e la loro disponibilità. Inoltre, si permette di selezionare una colonnina disponibile per avviare la ricarica di una propria auto e di tracciarne il progresso di caricamento.

Chapter 2

Requisiti

2.1 Requisiti funzionali

- Login di un utente registrato;
- aggiungere/modificare/rimuovere un'auto;
- cercare le colonnine vicine, o la più vicina, ad un determinato indirizzo;
- effettuare una richiesta ad un LLM per trovare le colonnine più vicine, o la più vicina, ad un determinato indirizzo; con la possibilità di specificare anche dei parametri.
- ricarica di un'auto presso una colonnina scelta;
- monitoraggio dello stato di carica delle proprie auto in ricarica.

Funzionalità aggiuntive per utenti **admin**:

- aggiungere/modificare/rimuovere una colonnina di ricarica;
- abilitare/disabilitare una colonnina;

2.2 Requisiti non funzionali

- Persistenza: l'utente rimane loggato e le ricariche in corso rimangono monitorate anche dopo aver ricaricato la pagina;
- gestione errori: feedback con messaggio esplicativo in caso di successo o fallimento di un'azione;
- interfaccia grafica intuitiva e che si adatta efficacemente alla dimensione dello schermo.

Chapter 3

Design

Per quanto riguarda la comunicazione tra backend e frontend, assumono un ruolo fondamentale gli eventi, i quali vengono inviati dal backend tramite **socket** riguardo specifici aggiornamenti dello stato interno; questi vengono ascoltati dal frontend di interesse per aggiornare la relativa interfaccia grafica.

3.1 Backend

L'architettura del backend, come in Figura 3.1, si compone di due container che comunicano tramite una rete interna: uno per il database **MongoDB** e l'altro per l'applicazione **Node.js**. Quest'ultima si suddivide in:

- `models`: contiene le interfacce che modellano il dominio applicativo e la struttura dei dati;
- `controllers`: gestisce la logica delle chiamate per ogni rotta;
- `routes`: contiene le rotte alle quali sono esposte le risorse dell'applicazione.

Il database è formato da due collezioni: `chargingStations` e `users`; quest'ultima viene fornita già comprendendo alcuni utenti in quanto non si realizza la funzionalità di registrazione.
L'applicazione permette la comunicazione con l'esterno tramite l'esposizione di **REST API**, mostrate in Figura 3.2 e Figura 3.3, che contengono come collezioni principali `/charging-stations` e `/cars`, le quali forniscono operazioni CRUD per ottenere e manipolare i documenti. I principali controllers sono `/login`, `/charging-stations/{id}/start-recharge`, `/charging-stations/{id}/stop-recharge`, `/charging-stations/near`, `/charging-stations/closest` e `/llm/search`, i quali permettono di effettuare specifiche azioni all'interno dell'applicazione.

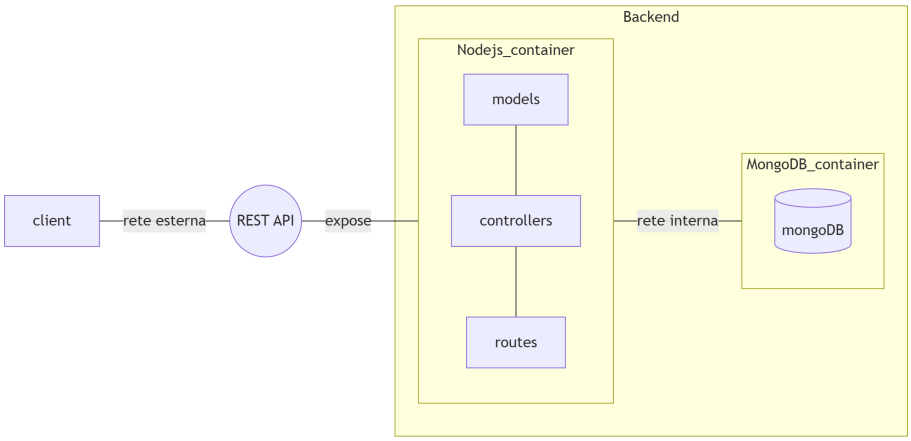


Figure 3.1: Design

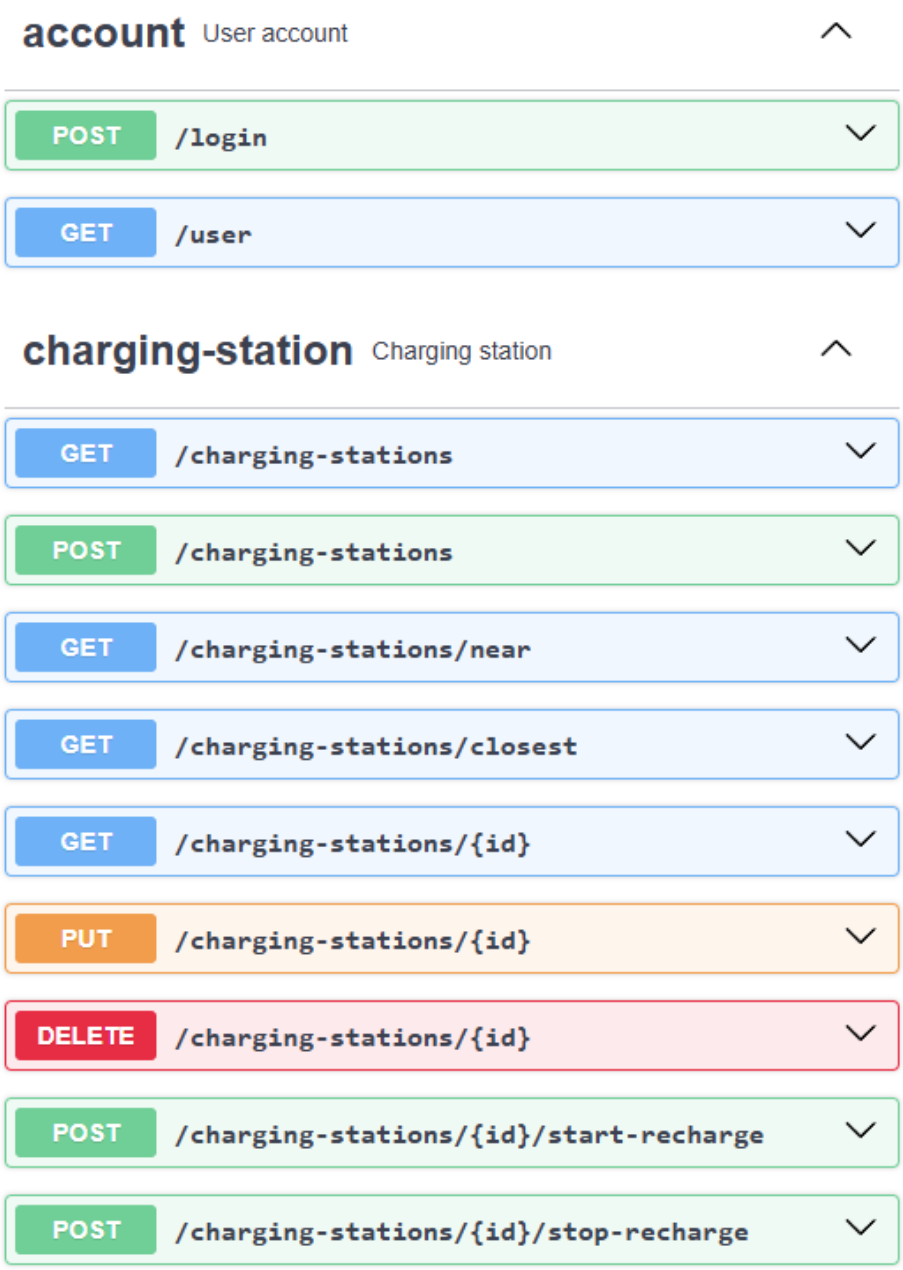


Figure 3.2: REST API account e charging station

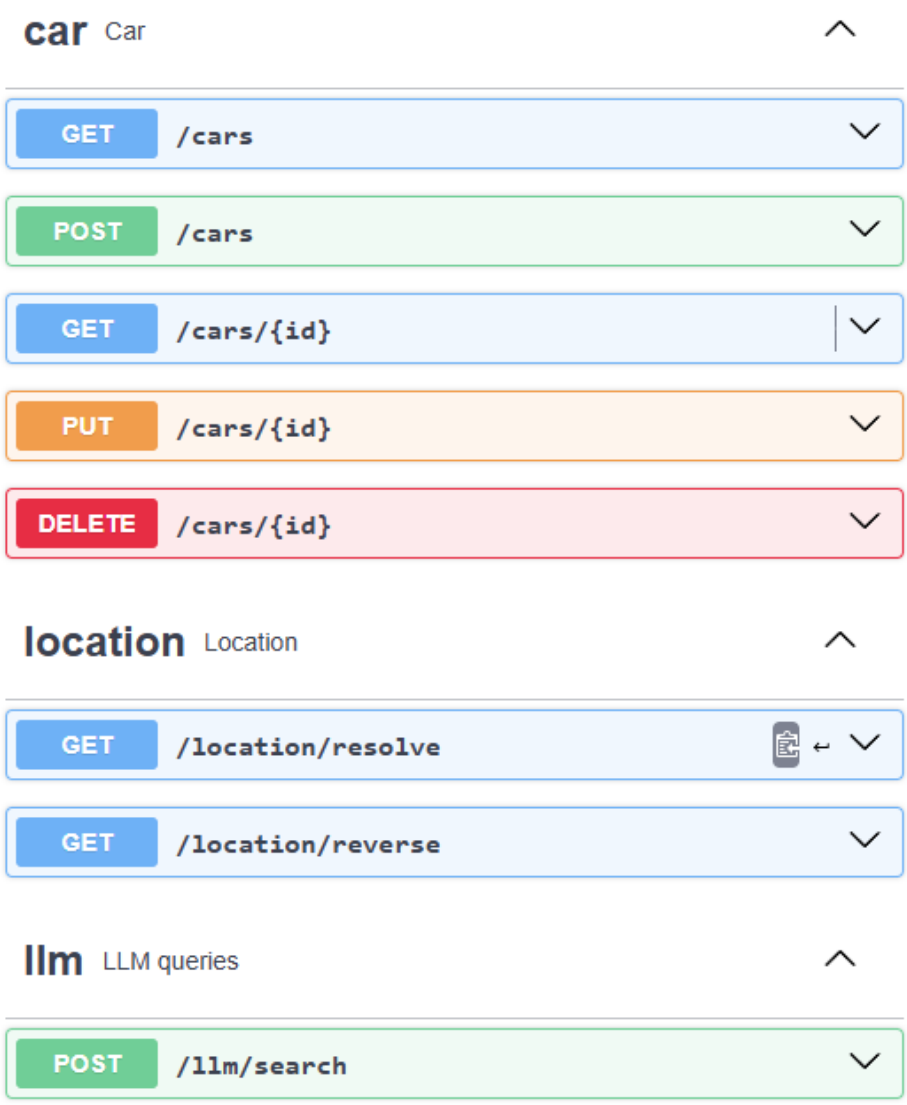


Figure 3.3: REST API car, location e llm

3.2 Frontend

Sono stati realizzati alcuni Mockup in fase di analisi, seguendo un approccio **Mobile First**, per rappresentare le interfacce delle varie pagine che compongono l'applicazione. Questi design rappresentano una versione iniziale che poi ha subito alcuni miglioramenti a seguito di feedback da parte di utenti in fase di test.

Le pagine vengono navigate tramite una `Navbar`, la quale in modalità *mobile* è posizionata nella parte inferiore dello schermo, con il solo nome dell'applicazione nella parte superiore; mentre in modalità *desktop* è tutto posizionato nella parte superiore. Inoltre, i componenti si adattano in maniera fluida alla dimensione attuale dello schermo.

Login Page

La schermata per l'accesso al sistema è quella mostrata in Figura 3.4a.

Profile Page

La pagina in Figura 3.4b mostra una sezione per la visualizzazione delle proprie informazioni utente ed una sezione per visualizzare e gestire le proprie auto.

Home Page

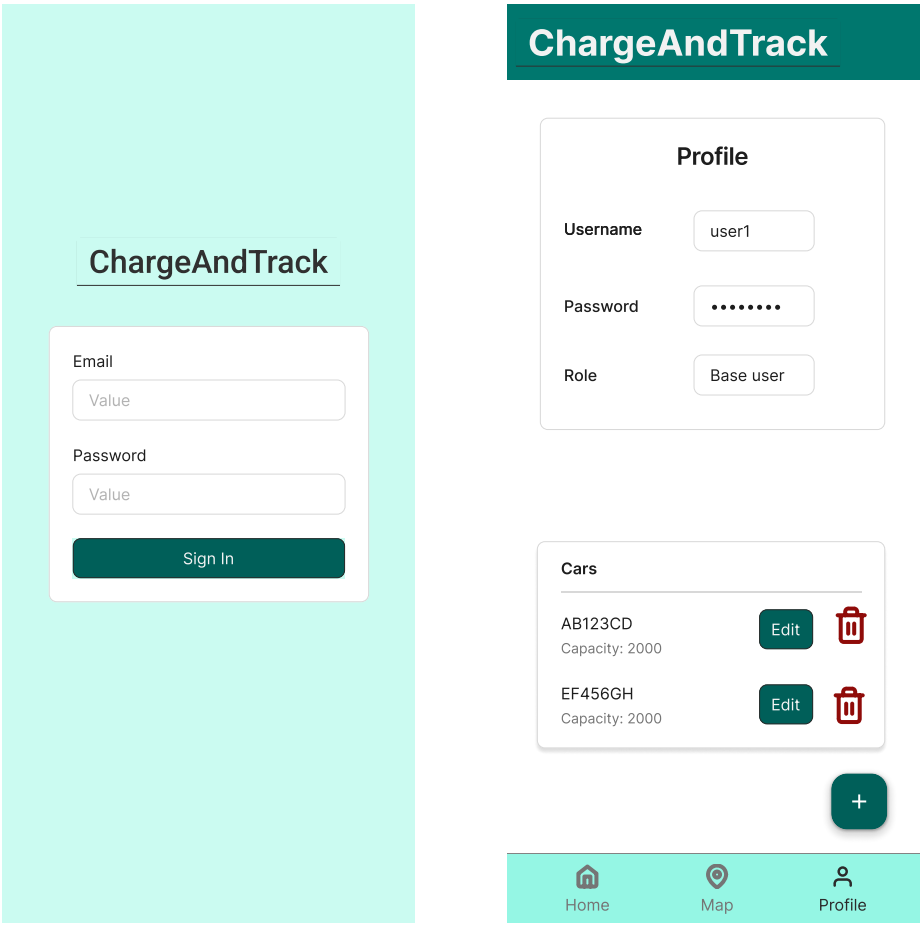
La pagina in Figura 3.5a mostra la schermata principale, la quale permette di effettuare richieste ad un LLM per cercare le colonnine di ricarica che verranno poi visualizzate in una lista sottostante. Inoltre, in caso l'utente abbia delle auto in fase di ricarica, queste vengono visualizzate mostrando dinamicamente la percentuale di batteria e con la possibilità di interrompere la ricarica.

Map Page

La pagina in Figura 3.5b permette di effettuare la ricerca delle colonnine in base ad un indirizzo inserito, visualizzandole all'interno di una mappa, la quale può essere navigata liberamente dall'utente. Inoltre, l'utente ha la possibilità di richiedere quale colonnina è la più vicina all'indirizzo cercato. Le colonnine presenti nella mappa sono rappresentate secondo il loro stato in tempo reale e mostrando le relative informazioni, con la possibilità di avviare una ricarica se la colonnina è disponibile.

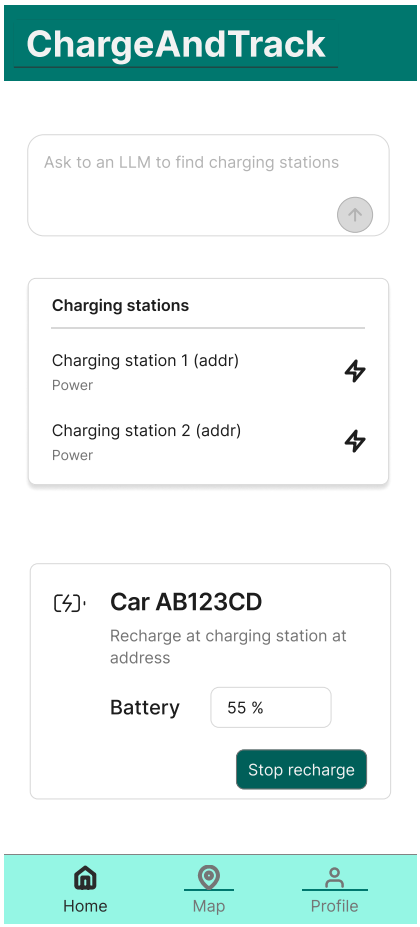
Manage Page

La pagina in Figura 3.6 rappresenta una schermata disponibile solo per gli utenti con ruolo di *admin*, la quale permette di gestire le colonnine di ricarica aggiornando le relative informazioni, rimuovendone una esistente oppure aggiungendone una nuova.

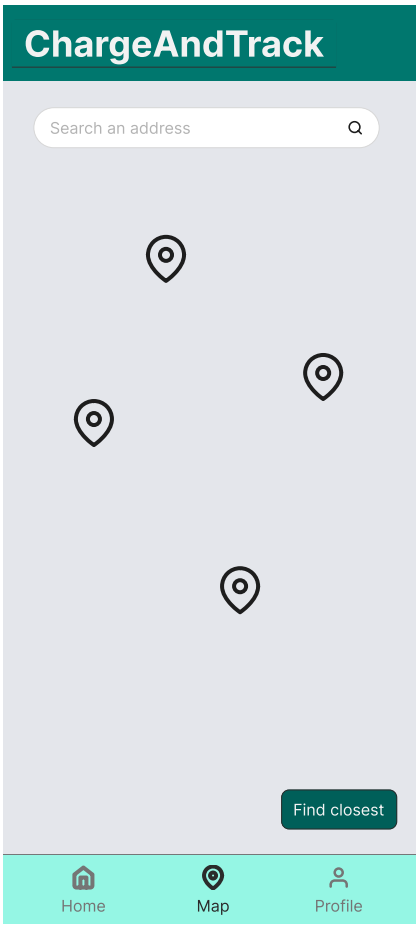


(a) Login page (b) Profile page

Figure 3.4: Login e Profile pages



(a) Home page



(b) Map page

Figure 3.5: Home e Map pages

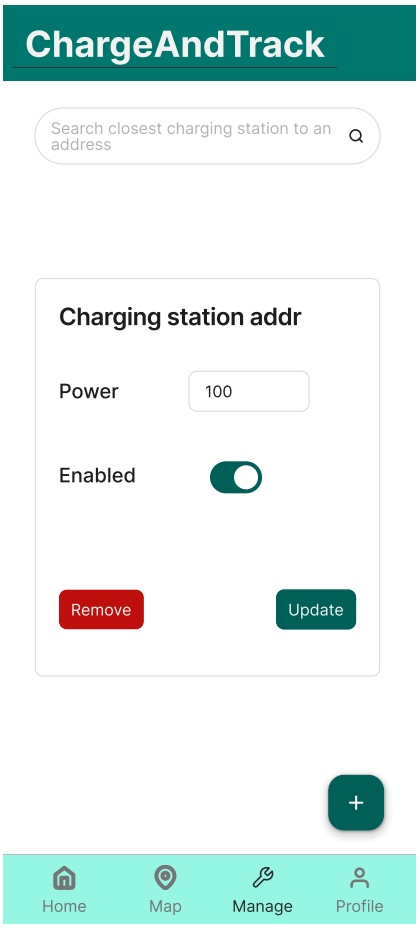


Figure 3.6: Manage page

3.2.1 Design di dettaglio

Le pagine consistono in un insieme di componenti, che sono realizzati in maniera modulare, facilitandone il riutilizzo in più pagine. Si utilizzano gli *store* per condividere facilmente tra le pagine alcune informazioni comuni. Inoltre, per ottenere persistenza della sessione anche a seguito di una ricarica della pagina, si utilizza una forma di archiviazione delle informazioni principali relative alla sessione.

Chapter 4

Tecnologie

Il sistema aderisce allo stack **MEVN**, che prevede l'utilizzo di:

- **MongoDB**: database documentale (NoSQL);
- **Express.js**: framework server-side per applicazioni web;
- **Vue.js**: framework per interfacce web lato utente;
- **Node.js**: JavaScript Runtime Environment.

Al posto di JavaScript abbiamo utilizzato **TypeScript** per facilitare lo sviluppo grazie alla tipizzazione.

4.1 Backend

4.1.1 Mongoose

Per interfacciarci con MongoDB abbiamo utilizzato **Mongoose**, che permette di definire gli schemi dei dati, facilitando la gestione e l'interazione col database. Inoltre, consente la validazione dei dati.

4.1.2 JWT

Per gestire l'autenticazione degli utenti abbiamo utilizzato **JWT (Json Web Token)** che permette di associare in fase di accesso ad ogni utente un token (con un periodo di validità limitato), che poi dovrà essere fornito nelle future richieste per verificare l'autenticazione, altrimenti la richiesta fallirà.

4.1.3 Zod

Abbiamo utilizzato **Zod** per definire schemi in modo da validare i dati relativi alle richieste. In particolare, li utilizziamo per validare i dati ricevuti in ingresso, controllando che siano conformi alle specifiche delle API.

4.1.4 BullMQ e Redis

Per simulare il processo di ricarica di un'auto presso una colonnina abbiamo utilizzato la libreria **BullMQ** che consente l'esecuzione di job in maniera periodica. In particolare, ogni job consiste nell'incremento dell'1% della batteria, e il periodo viene calcolato in base alla potenza della colonnina e alla capacità dell'auto in carica. Inoltre, è stato necessario utilizzare **Redis**, in quanto BullMQ è direttamente collegato ad esso. Per fare ciò è stato aggiunto un container Docker costruito su un'immagine esistente di Redis.

4.1.5 Socket.io

Per inviare eventi al frontend riguardanti incrementi della batteria durante le ricariche e cambiamenti di disponibilità delle colonnine, abbiamo utilizzato **Socket.io**, che ci permette di creare un server che rimane in ascolto di eventuali connessioni e successivi eventi in ingresso. Gli eventi inviati vengono separati in *room* in modo da essere ricevuti solo dai client interessati.

4.2 Frontend

Per realizzare l'interfaccia grafica abbiamo utilizzato come framework CSS **Bootstrap**, mentre per effettuare richieste HTTP ci siamo appoggiati alla libreria **Axios**. Dato l'obiettivo di realizzare una SPA è stato fondamentale l'utilizzo di **Vue-Router** che gestisce il routing lato client. Inoltre, abbiamo definito una *navigation guard* per impedire e reindirizzare un tentativo di navigazione sia in caso di utente non autenticato che non autorizzato.

4.2.1 Pinia

Data la necessità di accedere ad alcune informazioni in ogni punto dell'applicazione abbiamo utilizzato **Pinia** per la creazione di due store centralizzati, uno per l'autenticazione ed uno per tenere traccia delle ricariche in corso. Dato che gli store non sono persistenti abbiamo salvato le stesse informazioni nel *Local Storage* così da recuperarle in seguito ad un ricaricamento della pagina.

4.2.2 Leaflet

Abbiamo utilizzato la libreria **Leaflet**, che, col supporto delle API di **OpenStreetMap**, ci ha permesso di gestire e visualizzare la mappa in maniera interattiva. Inoltre, abbiamo potuto aggiungere alla mappa marker corrispondenti alle colonnine di ricarica coi quali è possibile interagire.

4.2.3 Socket.io

Per ricevere gli eventi inviati dal backend e per aggiornare l'interfaccia di conseguenza, utilizziamo la libreria **Socket.io** lato client. Inoltre, la utilizziamo per inviare al backend la volontà di unirsi o abbandonare una determinata room.

Chapter 5

Codice

5.1 Backend

5.1.1 Indici e query geospaziali con MongoDB

Per gestire le informazioni riguardanti le posizioni delle colonnine di ricarica abbiamo deciso di memorizzare le coordinate di quest'ultime in modo da sfruttare MongoDB per effettuare query geospaziali. Nello snippet 5.1 è mostrata la creazione del'indice geospaziale e nello snippet 5.2 è presente la definizione dell'interfaccia della rappresentazione GeoJSON relativa. Infine, nello snippet 5.3 si può vedere un esempio di query per cercare la colonnine di ricarica vicine a certe coordinate.

Listing 5.1: Location index

```
db.chargingStations.createIndex({ location: "2dsphere" });
```

Listing 5.2: GeoPoint interface

```
export interface GeoPoint {
    type: Point;
    coordinates: [Longitude, Latitude];
}
```

Listing 5.3: getNearbyCS function

```
export async function getNearbyCS(role: Role, data:
    NearChargingStationsDTO, llmFilters: LlmFiltersSchema =
    {})
        : Promise<ChargingStation[]> {
    const EARTH_RADIUS_METERS = 6378137;
    const onlyEnabled = getOnlyEnabledIfAllowed(data.
        onlyEnabled, role);
    return await chargingStationModel.find(
```

```
        {
            location: {
                $geoWithin: {
                    $centerSphere: [[data.lng, data.lat],
                        data.radius / EARTH_RADIUS_METERS]
                }
            },
            ...(onlyEnabled ? { enabled: true } : {}),
            ...(llmFilters.minPowerKw ? { power: { $gte:
                llmFilters.minPowerKw } } : {}),
        },
        { ...(onlyEnabled ? { enabled: 0  } : {}) }
    );
}
```

5.1.2 Integrazione LLM

Per quanto riguarda l'integrazione con un LLM ci siamo appoggiati alla piattaforma **HuggingFace**, effettuando richieste a `https://router.huggingface.co/v1/chat/completions` ed utilizzando il modello `Qwen/Qwen2.5-7B-Instruct:together`. L'LLM si occupa di ricevere la richiesta dell'utente e generare un JSON che rispetti lo schema 5.4, così che possa essere interpretato in modo da effettuare la corretta richiesta (tramite la funzione riportata nello snippet 5.5).

Listing 5.4: LLM response schema

```
export const llmFiltersSchema = z.object({
    minPowerKw: z.number().positive().optional()
});
export const llmResponseSchema = z.object({
    intent: z.enum(["NEAR", "CLOSEST"]),
    address: z.string().min(3),
    filters: llmFiltersSchema.optional()
});
```

Listing 5.5: makeRequest function

```
async function makeRequest(res: Response, role: Role, data:
    LlmResponseSchema): Promise<Response> {
    const location: LatitudeLongitudeDTO = await
        resolveAddress(data.address);
    console.log("Location: lat " + location.lat + " lng " +
        location.lng);
    switch (data.intent) {
        case "NEAR":
            const stations = await getNearbyCS(
                role,
```

```
                { lat: location.lat, lng: location.lng,
                    radius: DEFAULT_RADIUS },
                data.filters
            );
            return res.status(200).json(stations);
        case "CLOSEST":
            const chargingStations = await getClosestCS(role
                , location, data.filters);
            if (chargingStations.length === 0) {
                return res.status(404).json({ message: "No
                    charging stations found" });
            }
            return res.status(200).json(chargingStations[0])
                ;
    }
}
```

5.1.3 Recharge Worker

Si crea un worker di BullMQ che processa i job presenti nella coda, dove un job nel nostro caso consiste nell'incremento della batteria di un'auto in fase di ricarica dell'1%. Se questo aggiornamento va a buon fine si invia alla room specifica per quell'auto un evento `recharge-update`. Inoltre, nel caso in cui la batteria abbia raggiunto il 100% di carica si invia alla room specifica della colonnina un evento `charging-station-updated` e si interrompe la carica rimuovendo anche il job periodico dalla coda. Si definisce anche la connessione verso Redis per poter iniziare a processare i job.

Listing 5.6: Recharge Worker

```
export const rechargeWorker = () => {
    worker = new Worker('recharge-queue', async (job: Job)
        => {
        const { userId, carId, chargingStationId } = job.
            data;
        const userWithCar = await updateCarLogic(
            userId,
            carId,
            UpdateCarMethod.Inc,
            { "cars.$.currentBattery": 1 }
        );
        if (!userWithCar) {
            throw new Error("Car not found");
        }
        const currentBattery: number | undefined =
            userWithCar.cars[0]!.currentBattery;
        if (currentBattery) {
            io.to(`car:${carId}`).emit('recharge-update', {
                id: carId, level: currentBattery });
```

```
            console.log("Battery update to " +
                currentBattery);
            if (currentBattery >= 100) {
                const chargingStation = await
                    chargingStationModel.findByIdAndUpdate(
                    chargingStationId,
                    { $set: { "available": true }, $unset: {
                        currentCarId: "" } },
                    { new: true, runValidators: true }
                );
                if (!chargingStation) {
                    throw new Error("Charging station not
                        found");
                }
                await updateCarLogic(userId, carId,
                    UpdateCarMethod.Unset, { "cars.$.
                    currentChargingStationId": "" });
                io.to(`chargingStation:${chargingStation._id
                    }`)
                    .emit("charging-station-updated", { id:
                        chargingStation._id });
                job.repeatJobKey ?
                    await rechargeQueue.removeJobScheduler(
                        job.repeatJobKey) :
                    await rechargeQueue.removeJobScheduler(
                        carId);
                return { status: 'Recharge complete' };
            }
            return { status: 'In charge' };
        } else {
            throw new Error("Car has no currentBattery value
                ");
        }
    }, {
        connection: {
            host: config.redisHost,
            port: config.redisPort,
            maxRetriesPerRequest: null
        }
    });
    return worker;
};
```

5.2 Frontend

5.2.1 Axios e gestione errori

Abbiamo utilizzato gli *interceptors* di Axios, come mostrato nello snippet 5.7, per gestire aspetti generali riguardanti sia le richieste sia le risposte. Per quanto riguarda le richieste, prima di inviarle, aggiungiamo nell'header il token JWT, se presente nello store. Mentre per le risposte, aggiungiamo una gestione degli errori comune. Questa prende il messaggio della risposta (quindi dal backend) per mostrarlo all'utente, e nel caso non sia presente ne usa uno di default in base allo status code, come mostrato nello snippet 5.8.

Listing 5.7: Axios

```
export const api = axios.create({
    baseURL: "http://localhost:3000/api/v1",
    headers: { "Content-Type": "application/json" }
});

const { handleError } = useErrorHandler();

api.interceptors.request.use(
    (config) => {
        const authenticationStore = useAuthenticationStore()
            ;
        if (authenticationStore.token) {
            config.headers.Authorization =
                authenticationStore.token;
        }
        return config;
    },
    (error) => {
        if (error.response?.status === 403) {
            router.push({ name: 'Forbidden' });
        }
        return Promise.reject(error);
    }
);

api.interceptors.response.use(
    (response) => response,
    (error) => {
        const status = error.response?.status;
        if (status === 403) {
            router.push({ name: 'Forbidden' });
        }
        handleError(error);
        return Promise.reject(error);
    }
);
```

Listing 5.8: Gestione errori

```
const handleError = (error: any) => {
    message.show = true;
    const status = error.response?.status;
    const backendMessage = error.response?.data?.message;
    message.text = backendMessage ?? '';

    if (message.text === '') {
        switch (status) {
            case 400:
                message.text = "Invalid request. Please
                    check your input.";
                break;
            case 404:
                message.text = "The requested resource was
                    not found.";
                break;
            case 500:
                message.text = "Server error. We're working
                    on it!";
                break;
            default:
                message.text = "An unexpected error occurred
                    .";
        }
    }
    message.type = MessageType.Error;
    setTimeout(() => (message.show = false), 4000);
};
```

5.2.2 Routing

I path si dividono principalmente in `'/login'` e `'/'`, con quest'ultimo che ha come figli tutte le rotte delle altre pagine; in questo modo possiamo richiedere per accedervi l'autenticazione. Come detto in precedenza prima di ogni navigazione si controlla tramite la *navigation guard* 5.9 se la rotta di destinazione richiede l'autenticazione, e se non si è in possesso di un token valido si viene reindirizzati alla pagina di login. Allo stesso modo un utente non autorizzato viene bloccato se la rotta di destinazione richiede permessi da admin.

Listing 5.9: Routing

```
const routes = [
    { path: '/login', name: "Login", component: Login },
    {
        path: '/',
        meta: { requiresAuthentication: true },
        children: [
            { path: '', name: "Home", component: Home },
```

```
            { path: 'map', name: "Map", component: Map },
            { path: 'manage', name: "Manage", component:
                Manage, meta: { role: "admin" } },
            { path: 'profile', name: "Profile", component:
                Profile },
            { path: 'forbidden', name: "Forbidden",
                component: Forbidden },
            { path: ':pathMatch(.*)*', component: NotFound }
        ]
    }
];

const router = createRouter({
    history: createWebHistory(),
    routes
});

router.beforeEach((to) => {
    const authenticationStore = useAuthenticationStore();
    const isAuthenticated: boolean = !!authenticationStore.
        token;

    if (to.meta.requiresAuthentication && !isAuthenticated)
        {
        return { name: 'Login' };
    }

    if (to.meta.role === "admin" && !authenticationStore.
        isAdmin()) {
        return { name: 'Forbidden' };
    }
});
```

Chapter 6

Test

6.1 Backend

Abbiamo utilizzato le librerie **Node.js Test Runner (node:test)** e **SuperTest**, quest'ultima per effettuare richieste verso le REST API esposte. I test end-to-end consistono nel verificare che le risposte ottenute in base agli input forniti corrispondano a quelle attese, sia in caso di successo che di fallimento.

6.2 Frontend

Sono stati effettuati degli **User Acceptance Tests** con due utenti admin e quattro utenti base. Da questi, come si può notare dalle differenze tra i mockup e l'interfaccia grafica finale, sono emerse alcune organizzazioni non sufficientemente chiare:

- visibilità della password: aggiunta di un pulsante per visualizzare/nascondere la password durante il login e nel profilo dell'utente;
- auto dell'utente: spostato il pulsante di aggiunta di un'auto all'interno della lista per definire con chiarezza il suo contesto, e spostati i pulsanti di cancellazione sulla parte sinistra per allontanarli da quelli di modifica per evitare click involontari;
- home page: aggiunta di un pulsante per rimuovere la lista di colonnine ottenuta tramite la richiesta all'LLM per lasciare più spazio alla visualizzazione delle auto in carica;
- messaggi di successo: mostrare tramite un *toast*, non solo i messaggi d'errore, ma anche quelli di successo, per fornire un feedback sull'esito delle azioni dell'utente.

Chapter 7

Deployment

Il deployment del backend è stato effettuato tramite **Docker** e **Docker Compose**.
Il backend può essere avviato tramite **Docker Compose** seguendo i seguenti passi:

- clonare il repo oppure scaricare l'ultima release;
- posizionarsi nella root (`backend-asw`);
- eseguire lo script di setup tramite `./setup_env.sh`;
- rimpiazzare l'`HF_SECRET` presente nel file `nodejs/.env` con il proprio token di Hugging Face;
- eseguire i comandi `docker compose build` e `docker compose up`.

Il frontend può essere avviato seguendo i seguenti passi:

- clonare il repo oppure scaricare l'ultima release;
- posizionarsi nella root (`frontend`);
- eseguire il comando `npm run build`;
- eseguire il comando `npm run preview`.

Chapter 8

Conclusioni

In conclusione, il progetto è stato svolto interamente rispetto alle funzionalità e ai requisiti che ci eravamo posti inizialmente in fase di proposta; lasciando spazio a future integrazioni per ampliare il suo utilizzo. Rispetto ai lavori passati, la possibilità di utilizzare lo stack **MEVN** ci ha aiutato nello sviluppo, grazie principalmente alla grande modularità, per esempio tramite l'utilizzo di componenti **Vue** molto facilmente riutilizzabili, ed una grande disponibilità di librerie all'interno del package manager **npm**. Inoltre, l'adozione di un database documentale come **MongoDB** ha reso lo sviluppo più veloce grazie alla sua flessibilità e l'interazione più immediata grazie all'uso del formato JSON.
Abbiamo notato grossi vantaggi nell'utilizzo di **Typescript** sia nel backend che nel frontend, grazie alla sua tipizzazione forte e alla possibilità di modellazione tramite interfacce.
Infine, l'aver utilizzato API esterne per la gestione della mappa e l'interazione con un **LLM** ha consentito di aumentare le possibilità dell'applicazione, senza aumentare notevolmente la complessità; lo stesso è avvenuto nell'implementazione del processo di ricarica, la cui gestione è stata grandemente facilitata dall'uso di **BullMQ** in collaborazione con **Redis**.