



e-go

Applicazioni e Servizi Web

Matteo Ragazzini - 0000978948 {matteo.ragazzini5@studio.unibo.it}

16 Gennaio 2022

Indice

| | | |
|----------|--|-----------|
| 1 | Introduzione | 2 |
| 2 | Requisiti | 3 |
| 2.1 | Requisiti Funzionali | 4 |
| 2.2 | Requisiti non Funzionali | 6 |
| 3 | Design | 7 |
| 3.1 | Design Architettuale | 7 |
| 3.2 | Design delle Interfacce | 10 |
| 3.2.1 | Storyboard con Mockup Interattivi | 11 |
| 4 | Tecnologie | 14 |
| 5 | Codice | 16 |
| 5.1 | Interazione Componenti Mappa | 16 |
| 5.2 | Gestione Autenticazione Utenti e Ruoli | 19 |
| 5.3 | Esempio di Vuex Store | 21 |
| 5.4 | Modello User MongoDB | 22 |
| 6 | Test | 24 |
| 7 | Deployment | 26 |
| 8 | Conclusioni | 27 |

Capitolo 1

Introduzione

E-go è una applicazione web che permette ai cittadini di interagire con stazioni di ricarica, dislocate in punti strategici della città, per ricaricare i propri veicoli elettrici.

Il progetto nasce da un'esigenza personale, in quanto, nell'ultimo anno passato all'estero ho utilizzato come mezzo di trasporto principale, in combinazione o alternanza alla metropolitana, un monopattino elettrico.

Sebbene questa tipologia di veicoli sia pensata in un contesto di micro-mobilità, ho riscontrato circostanze in cui i 15-20 km di autonomia non sono stati sufficienti a coprire la tratta desiderata, ad esempio, il tragitto casa-lavoro-casa.

Per questo, ho immaginato una realtà alternativa nella quale invece che utilizzare veicoli in sharing, ogni cittadino dotato di un veicolo elettrico possa ricaricarlo in una delle stazioni sparse per la città.

Capitolo 2

Requisiti

Per la definizione delle funzionalità del sistema é stato adottato un approccio **user centered (UCD)**. In particolare gli utenti sono stati virtualizzati tramite la definizione di due **personas** fig. 2.1, fig. 2.2.

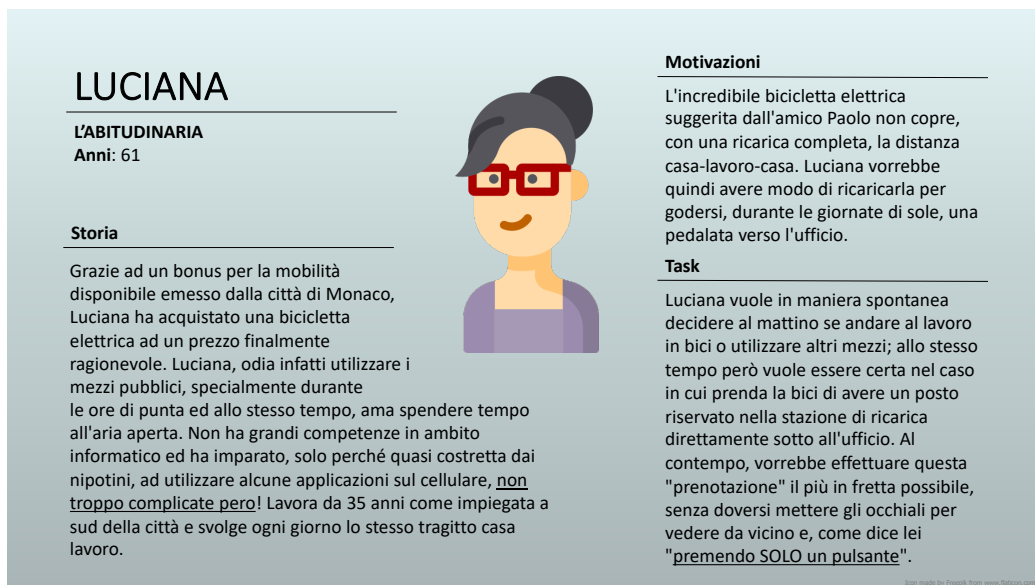


Figura 2.1: Personas 1: Luciana

Le caratteristiche e le funzionalità del sistema sono state definite quindi definite sulla base delle esigenze delle due personas in oggetto.

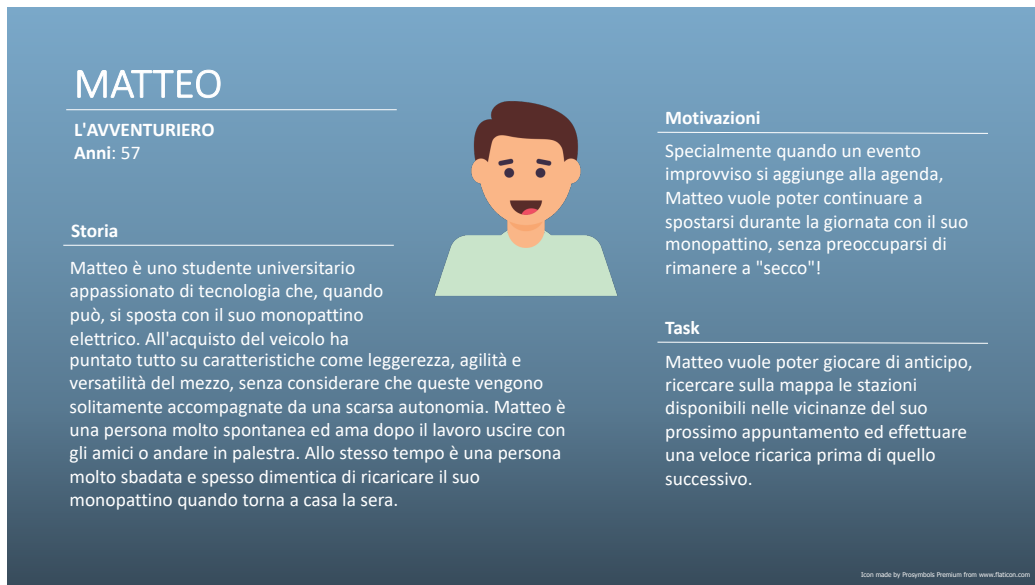


Figura 2.2: Personas 2: Matteo

2.1 Requisiti Funzionali

- **Registrazione e Login:** il sistema deve permettere agli utenti di registrarsi e successivamente di effettuare il login.
- **Registrazione di un veicolo:** prima di procedere all'utilizzo del sistema l'utente deve obbligatoriamente registrare almeno un veicolo ed averlo selezionato come veicolo in uso.
- **Cambio del veicolo in uso:** con l'idea di avere tassi di ricarica diversi in base alla tipologia di dispositivo (bicicletta o monopattino) l'utente deve poter cambiare il veicolo attualmente in uso.
- **Geolocalizzazione:** il sistema deve consentire all'utente di determinare in modo semplice e veloce qual è la stazione di ricarica più vicina a lui, per questo deve poter visualizzare sulla mappa la propria posizione.
- **Ricerca di una luogo:** allo stesso modo l'utente deve poter capire dove si troverà la torretta più vicina ad un luogo che vuole visitare.

- **Visualizzazione dello stato delle stazioni sulla mappa:** l'utente deve poter visualizzare lo stato delle stazioni sulla mappa con la relativa disponibilità di torrette libere.
- **Prenotazione di una stazione:** l'utente deve poter prenotare una stazione (la torretta gli verrà assegnata in maniera casuale in base alla disponibilità) per la durata di 30 min. Allo scadere del tempo, la prenotazione viene annullata e la torretta resa disponibile ad altri utenti.
- **Eliminazione di una prenotazione:** l'utente deve poter eliminare una prenotazione quando vuole prima dello scadere del tempo e liberare così la torretta per altri utenti.
- **Connessione diretta ad una stazione:** l'utente deve poter connettersi ad una stazione direttamente, senza previa prenotazione.
- **Connessione alla torretta:** una volta selezionata la stazione, all'utente vengono dati 30 secondi per avvicinarsi alla torretta assegnata, inserire il proprio veicolo nella rastrelliera ed infine appoggiare lo smartphone sul lettore NFC così da chiudere il lucchetto ed attivare la ricarica.
- **Disconnessione dalla torretta:** l'utente deve potersi disconnettere dalla torretta di ricarica
- **Selezione delle stazioni preferite:** L'utente deve poter selezionare le proprie stazioni preferite.
- **Visualizzazione delle sole stazioni preferite:** l'utente deve poter scegliere di visualizzare solo le stazioni da lui preferite.
- **Visualizzazione dello stato della ricarica:** in qualsiasi momento l'utente può visualizzare lo stato della batteria del dispositivo in carica.

- **Visualizzazione dello storico delle ricariche:** l'utente deve poter visualizzare la lista di ricariche effettuate con le relative informazioni: costo, durata, stazione, dispositivo e quantità di batteria ricaricata.

2.2 Requisiti non Funzionali

- **Mobile first:** l'applicazione è pensata per essere utilizzata da smartphone, perciò ogni dettaglio deve essere finalizzato a massimizzare l'esperienza utente su dispositivo mobile.
- **Semplicità di utilizzo:** l'applicazione è pensata per svolgere un solo compito: permettere all'utente di caricare un proprio veicolo, e per questo deve essere intuitiva e minimale.

Capitolo 3

Design

3.1 Design Architettuale

La Single Page Application è stata realizzata tramite **stack MEVN** ed è stata **dockerizzata** per favorire portabilità e manutenibilità.

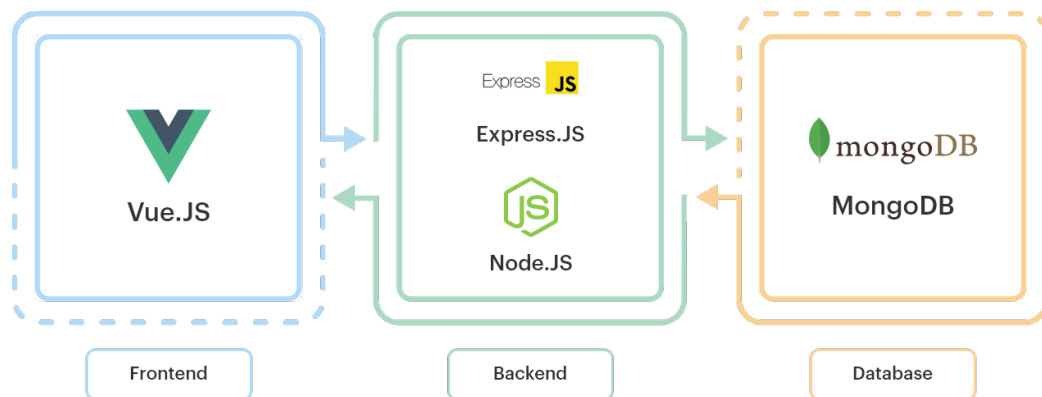


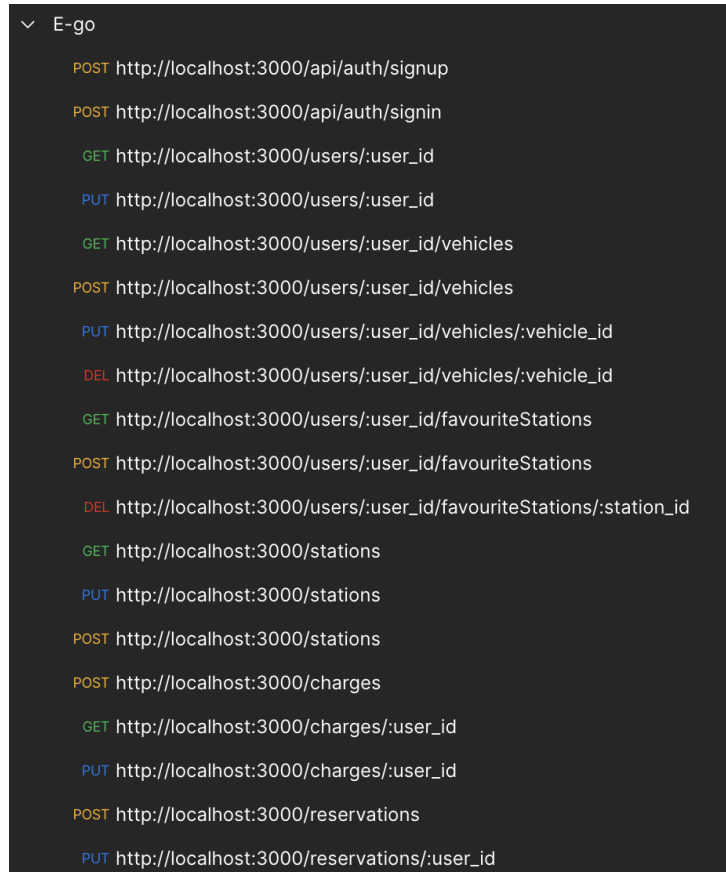
Figura 3.1: MEVN stack

La logica dell'applicazione è gestita lato server da 5 controllers distinti:

- **Authorization Controller**
- **Users Controller**
- **Stations Controller**

- Reservations Controller
- Charges Controller

Ciascuno di questi componenti espone una **REST API** che permette la comunicazione con il server.



```

  ▾ E-go
    POST http://localhost:3000/api/auth/signup
    POST http://localhost:3000/api/auth/signin
    GET http://localhost:3000/users/:user_id
    PUT http://localhost:3000/users/:user_id
    GET http://localhost:3000/users/:user_id/vehicles
    POST http://localhost:3000/users/:user_id/vehicles
    PUT http://localhost:3000/users/:user_id/vehicles/:vehicle_id
    DEL http://localhost:3000/users/:user_id/vehicles/:vehicle_id
    GET http://localhost:3000/users/:user_id/favouriteStations
    POST http://localhost:3000/users/:user_id/favouriteStations
    DEL http://localhost:3000/users/:user_id/favouriteStations/:station_id
    GET http://localhost:3000/stations
    PUT http://localhost:3000/stations
    POST http://localhost:3000/stations
    POST http://localhost:3000/charges
    GET http://localhost:3000/charges/:user_id
    PUT http://localhost:3000/charges/:user_id
    POST http://localhost:3000/reservations
    PUT http://localhost:3000/reservations/:user_id

```

Figura 3.2: User API

Inoltre per un aggiornamento real time di tutti gli utenti connessi al servizio, è stato sfruttato il meccanismo delle socket tramite **Socket.io**.

Un esempio di interazione fra i vari componenti è mostrato dal diagramma di sequenza in fig. 3.3 che mostra come vengono realizzate la prenotazione di una stazione e la connessione ad una torretta.

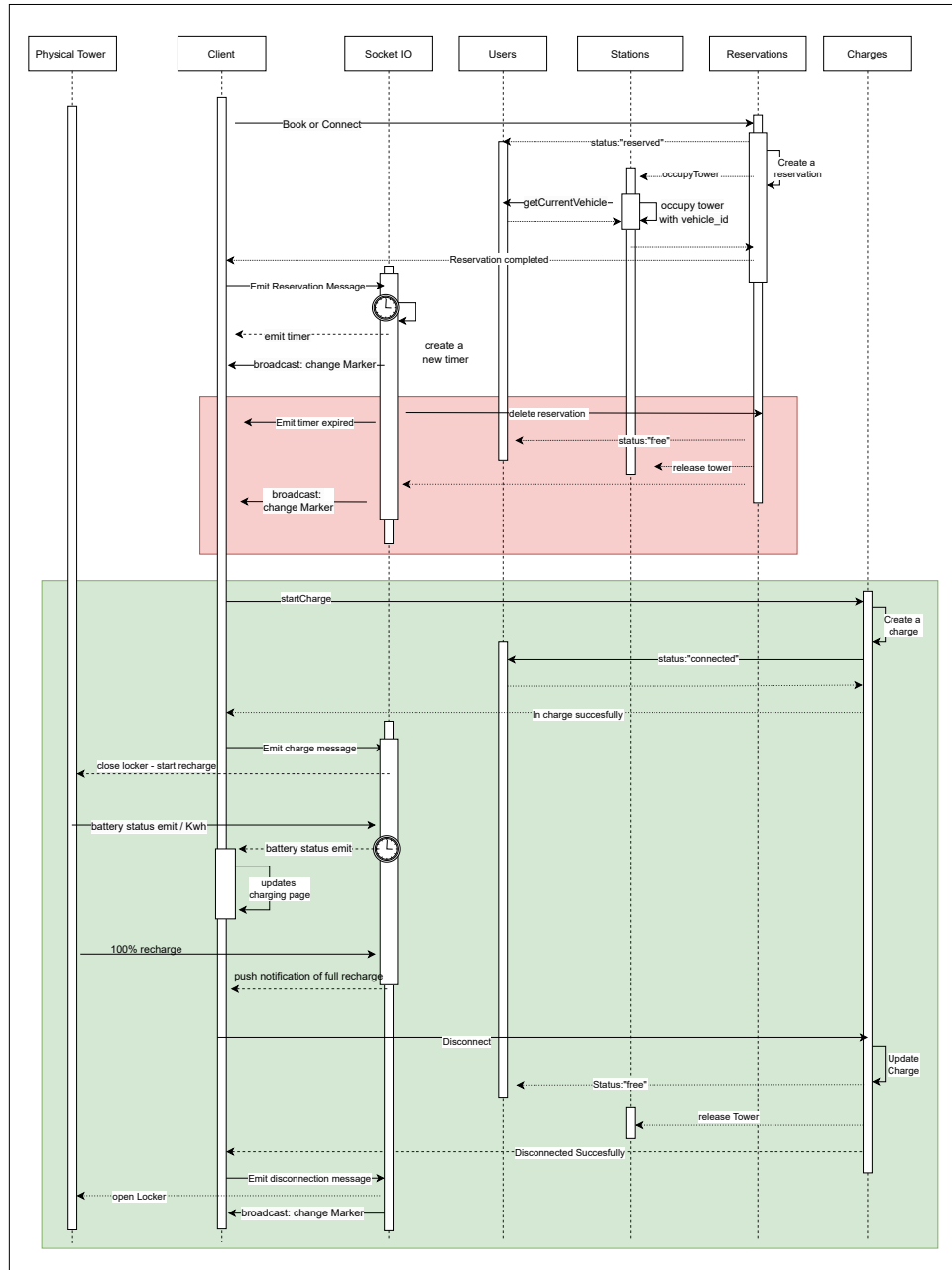


Figura 3.3: Sequence diagram for reservations and startCharge

3.2 Design delle Interfacce

Il design delle interfacce, è stato fortemente caratterizzato dalle esigenze degli utenti, nel caso specifico, le due *personas* esposte nel capitolo precedente. Inoltre, lo studio delle interazioni con i componenti è stato svolto seguendo i principi **KISS**, **Less Is More**, e le **Euristiche di Nielsen**. Questo ha portato ad avere ben chiare le idee su come sviluppare l'applicazione e ad effettuare cambiamenti minimi durante lo sviluppo.

Per comunicare l'idea dietro al progetto (e-go), ossia puntare sempre più ad una mobilità sostenibile, è stato scelto di utilizzare colori tenui e tendenti al verde.

Inoltre per massimizzare l'esperienza utente, minimizzare gli errori e un possibile rifiuto da parte di alcuni utenti, si è deciso di optare per **Google Maps** e **Google Map Autocomplete** come componenti principali in quanto si è ritenuto essere i più familiari a tutte le categorie di pubblico.

Per la progettazione dell'interfaccia utente è stata utilizzata la tecnica delle **storyboard con mockup interattivi** tramite l'utilizzo di Figma. Inoltre per rispondere al requisito non funzionale "Mobile First", l'applicazione è stata sviluppata come mobile first con **responsive design**.

3.2.1 Storyboard con Mockup Interattivi

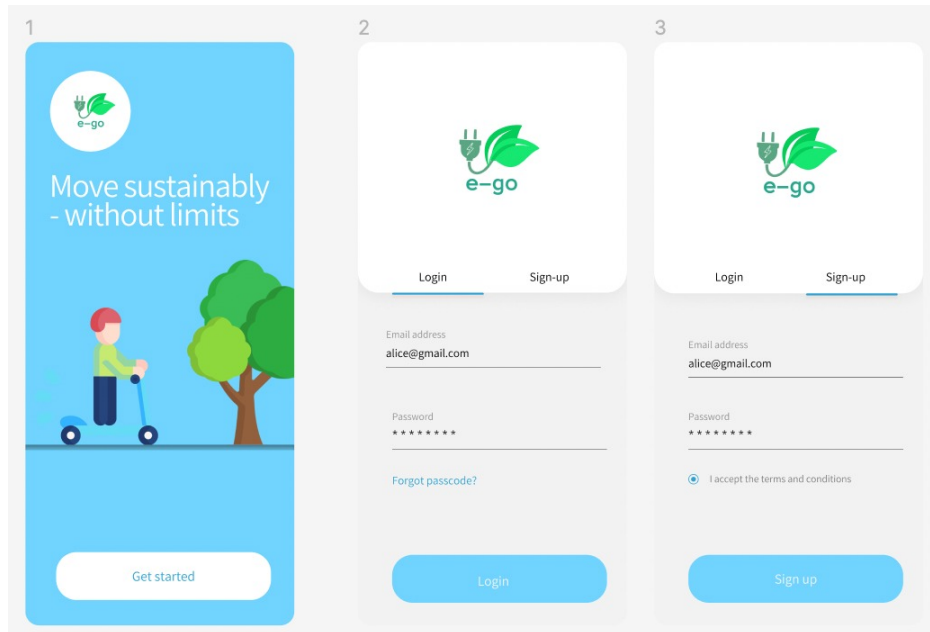


Figura 3.4: Mockup - Login page

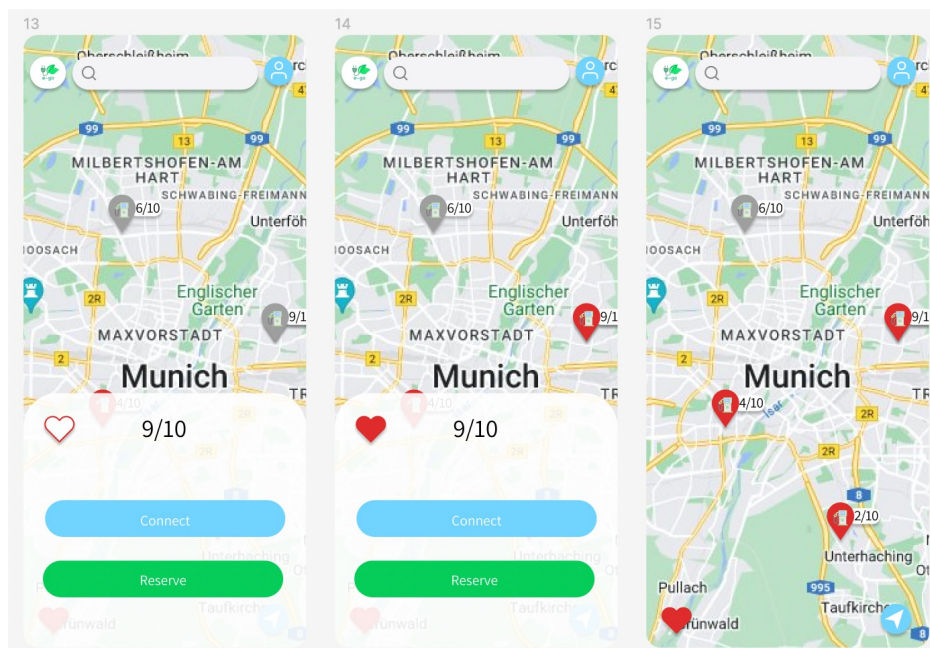


Figura 3.5: Mockup - Home page - favourites

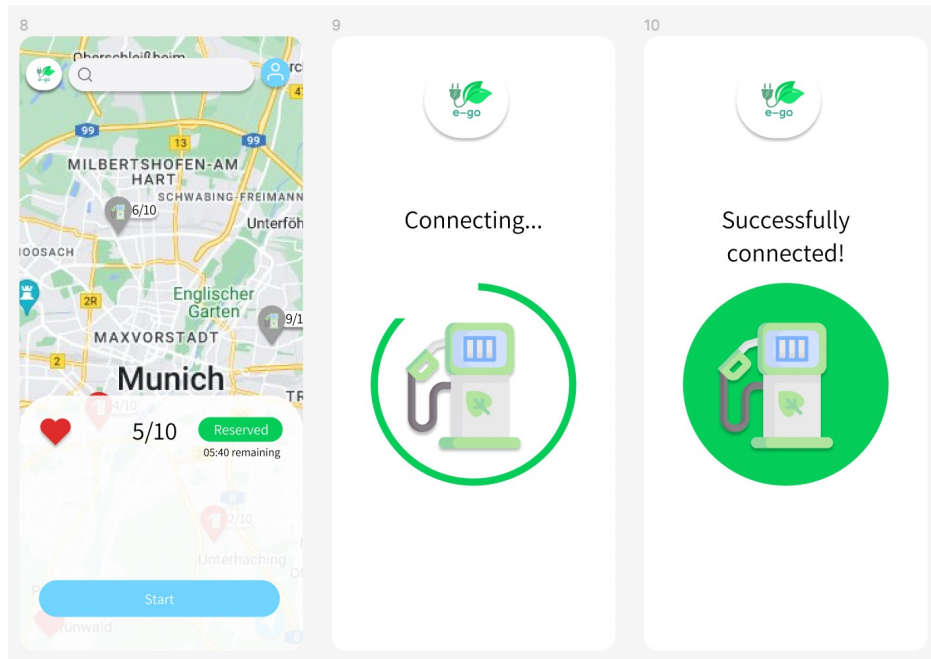


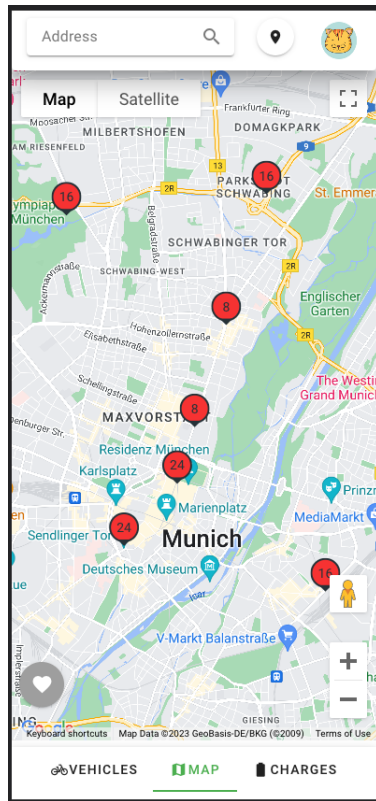
Figura 3.6: Mockup - charge page

Il mockup in fig. 3.5 mostra la risposta a due delle esigenze di Luciana. La prima riguarda la possibilità di selezionare le stazioni preferite, e conseguentemente di mostrare solo queste ultime sulla mappa. Questo riduce il disturbo visivo e il tempo di risposta dell'utente nel completare un task. La seconda riguarda l'aggiunta del bottone "reserve" che permette a Luciana di assicurarsi un posto nella stazione sotto l'ufficio ancora prima di uscire di casa, rimuovendo lo stress di non trovare una stazione libera.

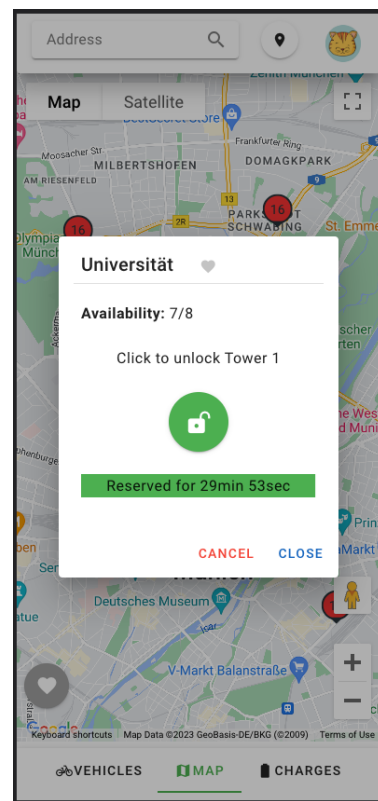
La barra di ricerca posta in alto, risponde invece ai bisogni di Matteo, che vuole poter cercare un luogo di interesse e vedere le stazioni al momento libere nelle vicinanze di quella posizione.

Il mockup in fig. 3.6 mostra invece l'interazione con la torretta di ricarica, la "station card" contenente le informazioni sulla stazione, lo stato di quest'ultima e il pulsante "connect".

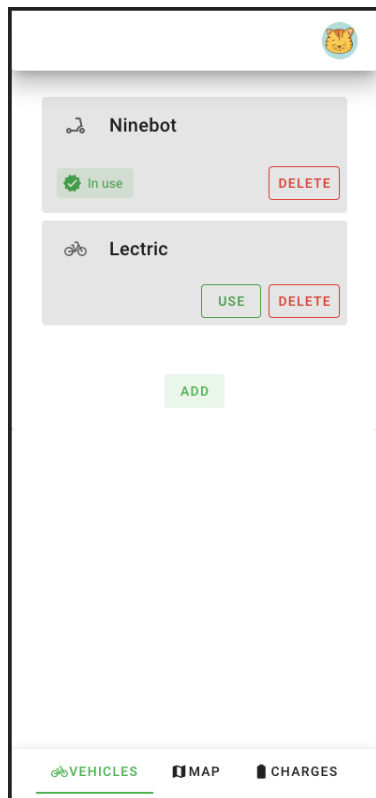
Di seguito sono invece riportati alcuni screenshot della applicazione per mostrare le funzionalità non riportate nei mockup.



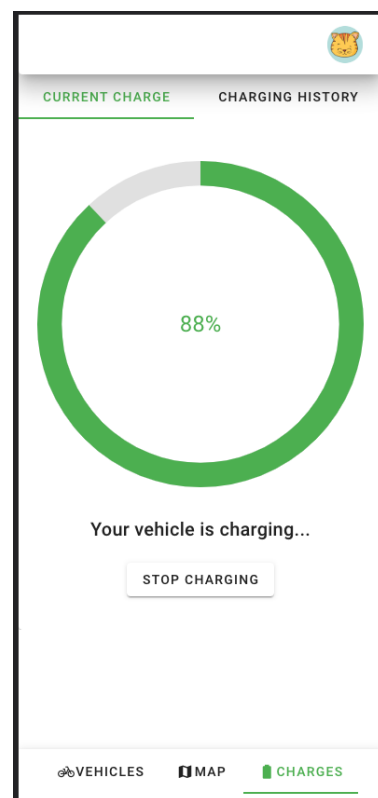
(a) App - Map page



(b) App - Station card



(c) App - Vehicles



(d) App - Charges

Capitolo 4

Tecnologie

Le tecnologie utilizzate per lo svolgimento del progetto sono molteplici, di seguito sono riportate quelle ritenute più importanti.

Considerando lo stack **MEVN** adottato sono stati utilizzati:

- **Mongo DB** per quanto riguarda la persistenza dello stato del sistema. In particolare sono state realizzate collezioni per stazioni, utenti, prenotazioni, e ricariche.
- **Express.js** come framework Node.js per implementare il server web
- **Vue 3** come framework lato client per la definizione delle interfacce in maniera modulare.
- **Node.js** come motore javascript.

L'utilizzo di Vue 3 ha portato al ricorso dell'utilizzo di librerie come:

- **Vue-Router** come router ufficiale di Vue.js per l'implementazione di una Single Page Application
- **Vuex** per la gestione dello stato globale mantenuto anche lato client
- **Vuetify** per la personalizzazione dei componenti Vue.
- **Vue-socket.io** che è un wrapper di socket.io-client che favorisce l'integrazione con Vue

- **Vee-validate** per la validazione dei dati all'interno del form di registrazione

Per quanto riguarda la mappa e la ricerca dei luoghi di interesse è stato fatto uso delle **Google Maps API** con le seguenti librerie: **Markers**, **Places** e **Autocomplete**. Sebbene esistano progetti open source che dovrebbero integrare Google Maps direttamente come componente Vue(es. Vue 3 Google maps) ho riscontrato molte difficoltà nell'utilizzo di questi componenti in maniera flessibile, perciò ho deciso di adattare i componenti originali a Vue. Altre tecnologie utilizzate sono state:

- **Mongoose** libreria per la modellazione di oggetti MongoDB in Node.js, semplifica la creazione di query fornendo una sintassi più snella e la possibilità di eseguirle in modo asincrono, inoltre, permette di validare dati in modo automatico
- **Axios** client HTTP per Node.js basato su Promise, dunque su programmazione asincrona. Permette di realizzare richieste a servizi REST in modo semplice e veloce e fornisce la possibilità di gestire le risposte e gli errori in modo puntuale
- **Socket.io** libreria Node.js per la gestione di notifiche push real-time, basata su Websocket e dunque una comunicazione bidirezionale tra client e server invece che su un'architettura a polling
- **Bcrypt** libreria che implementa l'omonimo algoritmo di hashing considerato lo stato dell'arte per la cifratura delle password. Viene utilizzato all'interno dall'authentication controller per la registrazione e autenticazione degli utenti.

Capitolo 5

Codice

Riporto di seguito alcune dettagli implementativi che ritengo importanti o peculiari.

5.1 Interazione Componenti Mappa

Nell'adattare l'API tradizionale di Google Maps alla struttura dei componenti imposta da Vue, ho dovuto risolvere alcuni problemi aggiuntivi, come ad esempio la condivisione di coordinate fra diversi componenti. In particolare è stato necessario permettere all'Autocomplete Component di passare alla mappa le nuove coordinate e forzarne l'aggiornamento.

Per fare questo ho fatto ricorso a meccanismi di Vue come le **props**, le **emits** ed i **watcher**.

1. Essendo i componenti fratelli, ho definito le coordinate come dato del componente Home e le ho passate come **props** ai componenti figli

```
1 <template>
2   <v-app-bar>
3     <AutocompleteComponent@newLocation="updateLocation":geoCords="this.
4       geoCords" />
5   </v-app-bar>
6   <v-main>
7     <Map :coords="this.coords" />
8   </v-main>
9 </template>
10 export default {
11   name: "Home",
12   data() {
13     return {
14       coords: {},
15       geoCords: {},
16       watcher: null,
17     }
18   },
19   mounted(){
20     this.watcher = navigator.geolocation.watchPosition(
21       position => {
22         this.geoCords.lat = position.coords.latitude
23         this.geoCords.lng = position.coords.longitude
24       }
25     )
26   },
27   methods:{
28     updateLocation(newLocation) {
29       this.coords.lat = newLocation.lat;
30       this.coords.lng = newLocation.lng;
31     }
32   }
}
```

Listing 5.1: Coordinates management in Home Component

2. Successivamente ho registrato un metodo di **emit** sull'Autocomplete component per andare a modificare in maniera safe le coordinate su Home.

```
1 export default {
2   emits: ['newLocation'],
3   props: ['geoCords'],
4   name: "UserLocation.vue",
5   data() {
6     return{
7       address: null,
8       searchedPos: {},
9     }
10  },
11  mounted() {
12    const autocomplete = new google.maps.places.Autocomplete(document.
13      getElementById("autocomplete"), {
14        bounds: new google.maps.LatLngBounds(
15          new google.maps.LatLng(this.geoCords)
16        )
17      });
18    autocomplete.addListener("place_changed", ()=>{
19      ...
20      this.$emit('newLocation', this.searchedPos)
21    })
22  },
23  methods:{
24    onLocate(){
25      ...
26      this.$emit('newLocation', this.geoCords)
27      ...
28    }
29  }
30 }
```

Listing 5.2: Coordinates management in Autocomplete Component

3. Infine ho dovuto registrare un **deepwatcher** nel component Map che rimane in ascolto di cambiamenti sulle coordinate e va a richiamare il metodo `setMapCenter(newPos)`.

```
1 export default {
2   name: "Map",
3   components: {StationCard},
4   props: {
5     coords: {
6       type: Object,
7       default() {
8         return {lat: 48.15143929407981, lng: 11.580534476878478}
9       },
10    },
11  },
12  watch: {
13    coords: {
14      handler(newPos, oldPos) {
15        this.setMapCenter(newPos)
16        ...
17        locationMarker = new google.maps.marker.AdvancedMarkerView({
18          position: new google.maps.LatLng(this.coords),
19          map: map,
20          content: geopos.element
21        })
22      },
23      deep: true
24    }
25  },
```

Listing 5.3: Coordinates management in Map Component

5.2 Gestione Autenticazione Utenti e Ruoli

Come esposto in precedenza il login e l'autenticazione degli utenti sono stati gestiti con la libreria **Bcrypt**. Nel listato è possibile notare come ad ogni utente venga assegnato un token in base al suo ruolo (User/Admin/Moderator) nel sistema.

Questa funzionalità pone le basi per una futura espansione dell'applicazione nella quale sia presente anche un'interfaccia per la creazione/aggiornamento delle stazioni. Tale funzionalità è stata testata ed utilizzata tramite Postman.

```
1 exports.signin = (req, res) => {
2   User.findOne({username: req.body.username})
3     .populate("roles", "-__v").exec((err, user) => {
4     if (err) {
5       res.status(500).send({ message: err });
6       return;
7     }
8     if (!user) {
9       return res.status(404).send({ message: "User Not found." });
10    }
11    var passwordIsValid = bcrypt.compareSync(
12      req.body.password,
13      user.password
14    );
15    if (!passwordIsValid) {
16      return res.status(401).send({
17        accessToken: null,
18        message: "Invalid Password!"
19      });
20    }
21    var token = jwt.sign({ id: user.id }, config.secret, {expiresIn:
22      86400}); //24h
23    var authorities = [];
24    for (let i = 0; i < user.roles.length; i++) {
25      authorities.push("ROLE_" + user.roles[i].name.toUpperCase());
26    }
27    res.status(200).send({
28      user:user,
29      roles: authorities,
30      accessToken: token
31    });
32  });
33 }
```

Listing 5.4: Login Function

5.3 Esempio di Vuex Store

Nel listato è riportato un esempio di Vuex store per il mantenimento in locale dello stato dell'utente (free/reserved/connected).

```
1 const user = JSON.parse(localStorage.getItem('user'));
2 const initialState = user
3   ? { status: user.status , station: user.occupiedStationId }
4   : { status: "FREE", station: null };
5
6 export const userState = {
7   namespaced: true,
8   state: initialState,
9   actions: {
10    goToReservedStatus({commit}, station_id){
11      commit('toReservedStatus', station_id)
12    },
13    goToConnectedStatus({commit}, station_id){
14      commit('toConnectedStatus', station_id)
15    },
16    goToFreeStatus({commit}){
17      commit('toFreeStatus')
18    },
19  },
20  mutations: {
21    toReservedStatus(state, station_id){
22      state.status = "RESERVED";
23      state.station = station_id
24    },
25    toConnectedStatus(state, station_id){
26      state.status = "CONNECTED";
27      state.station = station_id
28    },
29    toFreeStatus(state){
30      state.status = "FREE";
31    },
32  }
};
```

Listing 5.5: Vuex status store

5.4 Modello User MongoDB

La scelta di modellare la persistenza dell'applicazione tramite MongoDB ha richiesto di definire le entità tramite modello documentale. Si riporta di seguito come esempio, il Model della collezione User, definito tramite la libreria **Mongoose**

```
1 const User = mongoose.model(  
2   "User",  
3   new mongoose.Schema({  
4     username: {  
5       type: String,  
6       required: true  
7     },  
8     email: {  
9       type: String,  
10      required: true  
11    },  
12    password: {  
13      type: String,  
14      required: true  
15    },  
16    profilePicture: String,  
17    status: {  
18      type: String,  
19      enum: ["FREE", "RESERVED", "CONNECTED"],  
20      default: "FREE"  
21    },  
22    occupiedStationId: String,  
23    showOnlyFavourites: {  
24      type: Boolean,  
25      default: false  
26    },  
27    vehicles : [{  
28      name: String,  
29      vehicleType: String,  
30      img: String,  
31      batteryLevel: Number,
```

```
32         isCharging: Boolean,  
33         isCurrent: Boolean,  
34     }],  
35     favouriteStations : [  
36         {  
37             type: mongoose.Schema.Types.ObjectId,  
38             ref: "Station"  
39         }  
40     ],  
41     roles: [  
42         {  
43             type: mongoose.Schema.Types.ObjectId,  
44             ref: "Role"  
45         }  
46     ]  
47 })  
48 );
```

Listing 5.6: User Mongoose Model

Capitolo 6

Test

Durante tutta la fase di sviluppo e nella traduzione dei mockup in codice sono state utilizzate le **euristiche di Nielsen** per testare in maniera graduale e progressiva l'usabilità dell'interfaccia utente. Ad esempio l'introduzione della barra di caricamento per mostrare il tempo residuo di una prenotazione e il cerchio di bufferizzazione, per mostrare la quantità di batteria ricaricata, contribuiscono a fornire all'utente una costante **visibilità dello stato del sistema**. Oppure, per offrire un **design ed estetica minimalista** ed aumentare la **flessibilità e l'efficienza d'uso**, l'intera applicazione è stata scomposta su sole tre tab *vehicles*, *map* e *charges*, fra le quali l'utente casuale e/o intermittente, può spostarsi tramite i bottoni posti nella barra inferiore dell'applicazione, mentre l'utente più esperto può utilizzare uno swipe. Infine ad ogni azione sull'interfaccia è stato associato un messaggio su snackbar che **aiuti l'utente ad identificare un errore**. Ad esempio, un utente appena iscritto può voler tentare di prenotare una stazione senza aver registrato un dispositivo, in tal caso viene mostrato un messaggio su snackbar in rosso che gli suggerisce di registrare un nuovo dispositivo ed assegnarlo come dispositivo in uso.

Inoltre durante l'implementazione sono state effettuate sessioni di valutazione cooperativa (**Think Aloud Protocol**) durante le quali veniva chiesto agli utenti di svolgere determinati task definendo in maniera chiara e ad alta voce

le proprie idee ed intenzioni mentre interagivano con l'applicazione. Uno dei feedback ricevuti e successivamente implementato è stato il passaggio diretto dalla mappa alla pagina di ricarica alla pressione del pulsante "connect", in modo da dare consistenza al flusso di task del sistema.

Per quanto riguarda la parte server, le api sono state testate tramite Postman, simulando alcuni casi limite. Inoltre è stata realizzata una classe di test per l'API USER tramite l'utilizzo delle librerie **Chai** e **Mocha**. Tale classe di test non è esaustiva, ma rappresenta un tentativo di realizzazione di test più sofisticati e completi che potranno essere completati in caso di espansione dell'applicazione.

Capitolo 7

Deployment

Sebbene l'applicazione sia composta di soli 3 servizi: frontend, server e database, si è deciso di dockerizzare il sistema tramite **Docker** per favorirne il deployment.

In fase di deployment vengono inoltre rigenerati tutti i `node_modules` per i due progetti node relativi al frontend ed al backend, e viene effettuato un restore del db **E-GO-db** all'interno del container contenente mongodb.

Questo permette di dispiegare e rendere direttamente utilizzabile l'intera applicazione tramite il comando:

```
docker compose up
```

E-go sarà raggiungibile su: `http://localhost:8080/`

Una volta raggiunta la pagina di Login, è possibile registrarsi, o accedere con l'utente pre registrato:

```
username: dummy  
password: dummy
```

Capitolo 8

Conclusioni

Il progetto è stato un'importante occasione per consolidare le nozioni di applicazioni e sviluppo web che ritengo fondamentali in un mondo così in rapida crescita ed in cui, avere la possibilità di dare vita ad una propria idea, a costo zero, rappresenta un vantaggio notevole rispetto a molte altre categorie di studenti/lavori.

Inoltre e-go rappresenta il primo progetto di tutta la mia carriera universitaria svolto completamente da solo e soprattutto da studente lavoratore full time.

Se da un lato ciò ha rappresentato una notevole sfida in particolare per quanto riguarda i limiti sui tempi di realizzazione, dall'altro mi ha dato la possibilità di riscoprire la passione per lo sviluppo e di studiare ed approfondire tutti gli aspetti dietro alla creazione dell'applicazione.

Paragonando questo progetto a quello svolto in triennale durante il corso di Tecnologie Web, ho apprezzato molto la scomposizione aggiuntiva che è possibile ottenere tramite JS framework come Vue che, se da un lato "rompono" il paradigma MVC dall'altro, permettono di definire logica e view in maniera molto più strutturata.

Concludo dicendo che mi ritengo estremamente soddisfatto del lavoro svolto, del risultato ottenuto e delle capacità acquisite che sicuramente spenderò in futuri progetti personali e lavorativi.