CityBus

Applicazioni e Servizi Web

Matteo Bambini - matricola {matteo.bambini@studio.unibo.it}
Michele Nardini - matricola {michele.nardini2@studio.unibo.it}

3 Luglio 2025

Contents

0.1 Introduzione

CityBus è un sistema che ha l'obiettivo di semplificare sia l'utilizzo che la gestione dei trasporti pubblici su gomma.

Per gli utilizzatori dei trasporti offre una piattaforma dalla quale è possibile ottenere informazioni dettagliate su ciascun autobus in circolazione, coma ad esempio orari, ritardi e posizione in tempo reale e di agevolare tutti gli spostamenti tramite una funzione di navigazione che permette di calcolare il percoso migliore da un punto ad un altro all'interno della zona coperta dal servizio.

Per gli enti di trasporto invece offre funzionalità di pianificazione e monitoraggio tramite un sistema di creazione e gestione delle linee e dei percorsi e tramite una dashboard che permette di monitorare ciascun autobus in circolazione.

0.2 Requisiti

0.2.1 Utenti dei mezzi pubblici

Navigazione L'utente deve avere la possibilità di accedere ad un'interfaccia da cui può inserire un punto di partenza e uno di arrivo e viene proposto il percorso più rapido per arrivare a destinazione utilizzando una combinazione di percorsi a piedi e tratte con i mezzi pubblici.

Posizione autobus Deve essere presente un'interfaccia che permette di vedere in tempo reale la posizione di uno specifico autobus e avere una stima del tempo di arrivo alla fermata in cui si trova l'utente.

Tempo di percorrenza Un utente, a bordo del proprio autobus, deve avere la possibilità di osservare la posizione in tempo reale e ottenere una stima del tempo di arrivo alla fermata desiderata.

0.2.2 Enti di trasporto

Monitoraggio mezzi Deve essere disponibile una dashboard che ottiene dati in tempo reale da tutti i mezzi in servizio e fornisce informazioni riguardati posizione e ritardi di ciascun mezzo.

Pianificazione percorsi Deve essere messo a disposizione un sistema che permetta di pianificare nel dettaglio i percorsi dei mezzi.

0.3 Design

0.3.1 Architettura del sistema

Data la complessità del progetto, il sistema è composto da numerosi componenti che interagiscono tra loro; in figura 1 è rappresentato un diagramma che descrive quali sono questi componenti e come comunicano tra loro.

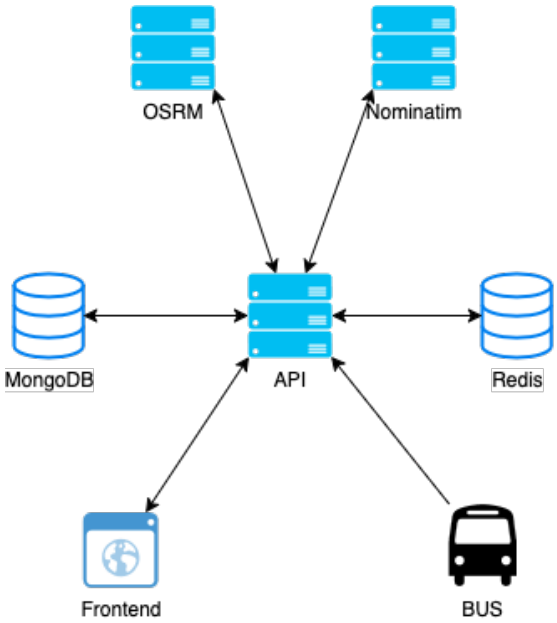

Figure 1: Architettura del sistema

MongoDB e Redis

MongoDB e Redis sono i due database utilizzati per memorizzare i dati del sistema. Redis contiene al suo interno i dati in tempo reale di ciascun autobus che includono posizione, tempo di arrivo alla fermata successiva, i minuti di ritardo e le informazioni sulla fermata successiva. Di seguito è riportato un esempio del formato di questi dati:

```
{
    "position": [12.055590668167374,44.22314450204817],
    "minutesLate": 10,
    "timeToNextStop": 75.7,
    "nextStop": {
        "stopId": "6857d15047831cdd3f8550c4",
        "name": "Galilei"
    }
}
```

MongoDB invece contiene al suo interno tutti gli altri dati del sistema; questi dati includono:

- informazioni sugli utenti
- informazioni su linee e fermate
- grafo del sistema di trasporto pubblico

- informazioni sulle corse
- tracce GPS dei percorsi delle linee

In figura 2 è possibile vedere parte dello schema del database (è stata omessa la collezione contenente le informazioni degli utenti per semplificazione).

Bus Lines La collezione `bus_lines` contiene tutte le informazioni sulle linee; queste informazioni, oltre al nome della linea, includono i dati su ciascuna direzione della linea e la tabella degli orari. Ciascuna direzione è a sua volta caratterizzata da nome, fermate e riferimenti alla traccia GPS del percorso; ciascuna fermata è un oggetto contenente l'id della fermata (che fa riferimento alla collezione `bus_stops`), il nome della fermata (ridondato in modo da ridurre il numero di query da eseguire), il tempo necessario per arrivare alla fermata successiva e il riferimento alla traccia GPS che determina il percorso tra la fermata e quella successiva. Come si può notare, tutte le tracce GPS sono memorizzate in una collezione separata dalle altre; è stata presa questa decisione per mantenere contenute le dimensioni dei documenti delle altre collezioni dato che queste tracce possono essere composte da molti punti e quindi occupare molto spazio.

Bus Stops La collezione `bus_stops` contiene tutte le informazioni riguardanti le fermate; queste informazioni includono nome, posizione della fermata e lista delle direzioni che passano per la fermata. La lista delle direzioni è un array contenente il riferimento all'id della direzione (il campo `directions._id` di `bus_lines`); questo array è utile per ottenere in modo rapido la lista degli autobus che partono da una determinata fermata senza dover scansionare tutta la collezione `bus_lines`,

Bus Rides La collezione `bus_rides` contiene tutte le informazioni riguardanti le corse degli autobus. Ciascun documento rappresenta una singola corsa in un determinato giorno e ad un determinato orario. Le corse sono caratterizzate da: riferimento alla linea e alla direzione (campi `lineId` e `directionId`), orario di partenza previsto, stato della corsa, lista (se è attiva o è terminata) e lista delle fermate. La lista delle fermate è un array di oggetti contenenti id della fermata, nome, l'orario di arrivo previsto e un flag che indica se il bus è già passato da quella fermata oppure no.

Stops Connections La collezione `stops_connections` contiene le informazioni riguardanti le connessioni tra le fermate; ciascun documento si può considerare come un arco del grafo della rete di trasporto in cui ogni fermata è un nodo. Ogni arco è caratterizzato dai riferimenti alle fermate di partenza e arrivo (rispettivamente i campi `from` e `to`) e dalla lista delle linee che coprono il tragitto tra le due fermate; la lista delle linee (campo `lines`) è un array di oggetti contenenti id della linea, id della direzione e tempo di percorrenza.

Routes come accennato in precedenza la collezione `routes` contiene le tracce GPS dei percorsi delle linee. Ciascun documento è caratterizzato da: id della direzione, tipo di traccia e la traccia vera e propria (il campo `path`); il tipo indica se la traccia è parziale (ovvero il percorso tra due fermate) oppure completa (ovvero la traccia di tutta la linea). Il campo `path` è un array di oggetti ciascuno dei quali indica uno step della traccia; questi step sono caratterizzati da tempo di percorrenza e un array di geometrie spaziali in formato GeoJSON (campo `path.geometry`).

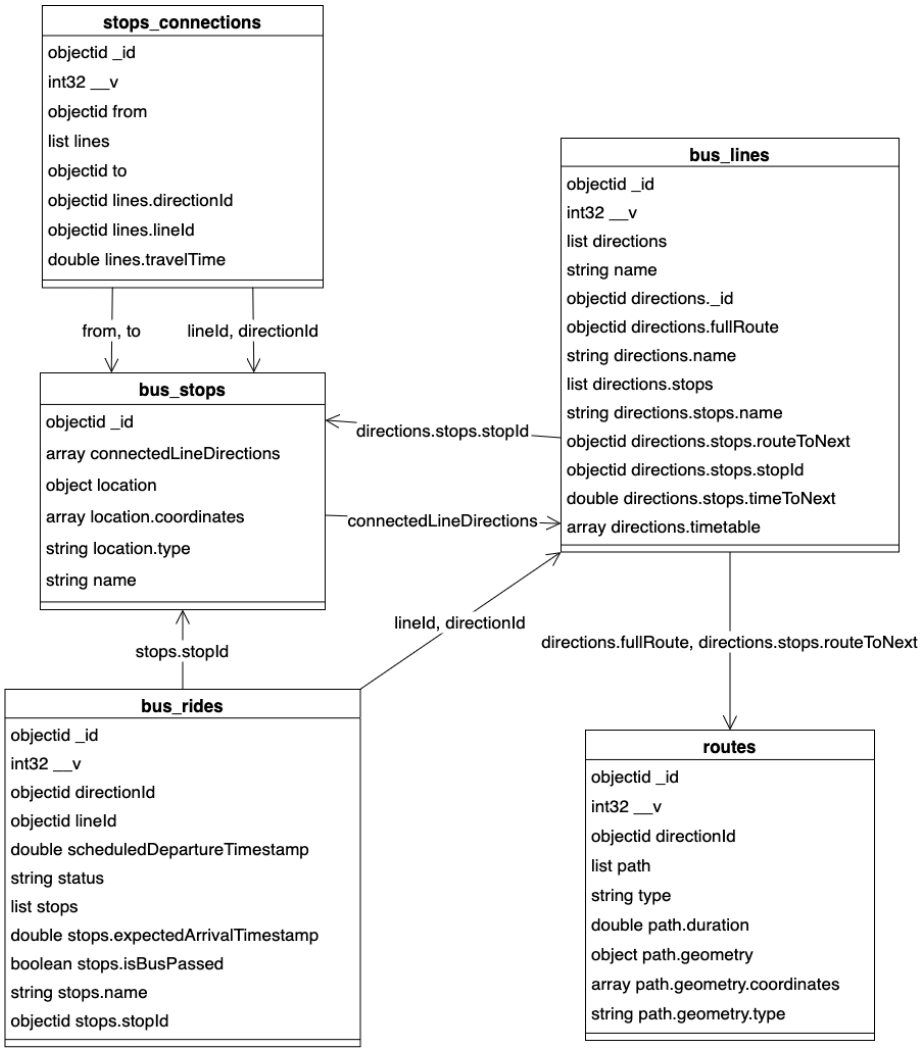

Figure 2: Schema del database

API

Le API sono il componente che gestisce la logica del sistema e fa da ponte tra i client e le basi di dati e servizi esterni. Sono state progettate e sviluppate seguendo il paradigma REST in modo da aderire a un modello standard e contemporaneamente avere un'architettura semplice e flessibile e tutti gli endpoint sono stati documentati utilizzando lo standard OpenAPI. All'interno del repository è possibile trovare sia il file con le specifiche (`api/api-schema.yaml`) che un file HTML contenente la documentazione generata in automatico (`api/docs/index.html`); è possibile inoltre visualizzare la documentazione a questo link.

OSRM e Nominatim

OSRM e Nominatim sono due servizi esterni che forniscono delle funzionalità necessarie al sistema tramite API. OSRM è un routing engine, ovvero un sistema che permette di calcolare i percorsi tra due punti; si basa sui dati di OpenStreetMap come il resto del sistema sviluppato. Nominatim invece è un servizio di geocoding, ovvero permette di convertire il nome di un luogo o un indirizzo in coordinate.

Come si può notare dallo schema in figura 1 questi servizi non vengono contattati direttamente dai client ma dalle API; il sistema è stato progettato in questo modo per rendere trasparente ai client i servizi esterni che vengono utilizzati e non creare quindi una dipendenza diretta, rendendoli intercambiabili senza la necessità di effettuare modifiche al di fuori delle API.

0.3.2 Architettura interfacce utente

Questo capitolo descrive il percorso di definizione del design delle interfacce utente, approfondendo i principi adottati, le scelte stilistiche e i mockup realizzati come base per l'implementazione.

Verranno presentati i prototipi sviluppati durante le fasi di analisi e progettazione, che hanno permesso di validare in anticipo la struttura dell'applicazione e l'organizzazione dei contenuti. Il risultato finale non si discosta molto dai mockup iniziali, ma sono state apportate modifiche laddove fosse necessario tenendo in considerazione i feedback utenti. I mockup sono stati creati sia per la versione mobile che la versione desktop dell'applicazione.

Autenticazione

La pagina di login in figura 3 e 4 rappresenta il primo punto di contatto tra l'utente e l'applicazione Citybus: un'interfaccia chiara e accessibile in questa fase è fondamentale per garantire un'esperienza positiva fin dall'inizio. In questa sezione vengono presentati i mockup realizzati per la schermata di login, sia nella versione mobile che in quella desktop.

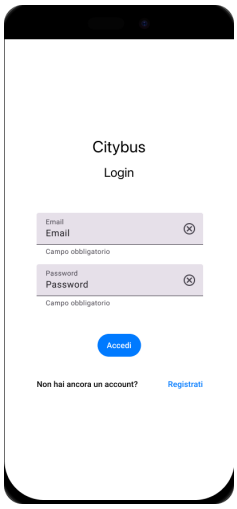

Figure 3: Mockup Login - Versione Mobile

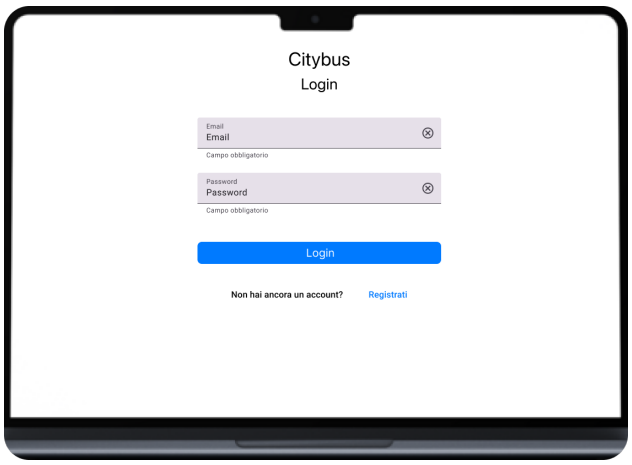

Figure 4: Mockup Login - Versione Desktop

Le due varianti di design si basano su un layout minimalista che evidenzia i campi di input (email e password), i messaggi di validazione, il pulsante di accesso e il link per la registrazione.

La pagina di registrazione costituisce il secondo step fondamentale nel flusso di autenticazione dell'applicazione Citybus, offrendo agli utenti un'interfaccia chiara per la creazione del proprio account. La disposizione dei campi (nome, cognome, email, password, conferma password) è pensata per guidare l'utente in un processo lineare e veloce, riducendo il rischio di errori e migliorando l'esperienza complessiva come si può vedere in Fig.5 e Fig.6.

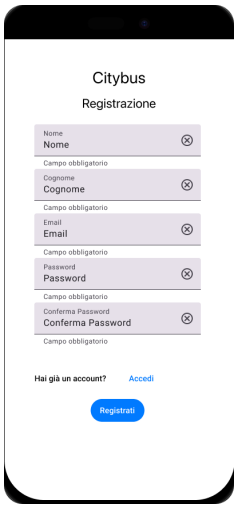

Figure 5: Mockup Registrazione - Versione Mobile

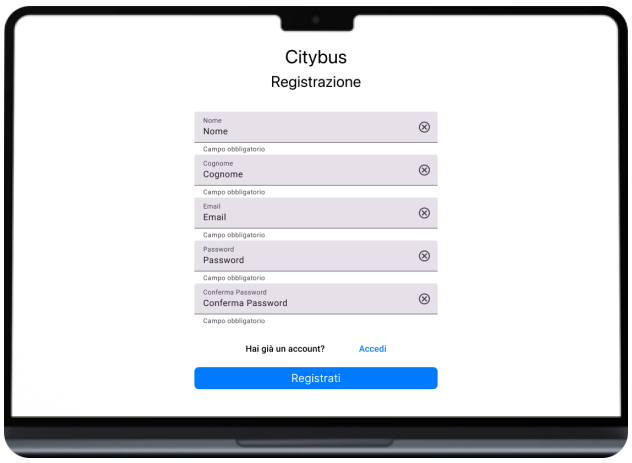

Figure 6: Mockup Registrazione - Versione Desktop

Navigazione & Partenze

Una volta effettuata l'autenticazione si verrà portati alla pagina principale, ovvero alla pagina di navigazione, che si può vedere come il cuore dell'applicazione Citybus come mostrato in fig.7, fig.8 e fig. 9. La pagina permette all'utente di cercare e visualizzare il percorso più veloce per raggiungere la destinazione scelta utilizzando i mezzi pubblici. Il sistema consente di specificare il punto di partenza, la destinazione(queste ultime due inseribili tramite mappa o direttamente dai campi di input) e l'orario desiderato di partenza, per restituire in tempo reale la sequenza ottimale di fermate e la mappa interattiva con il percorso del bus.

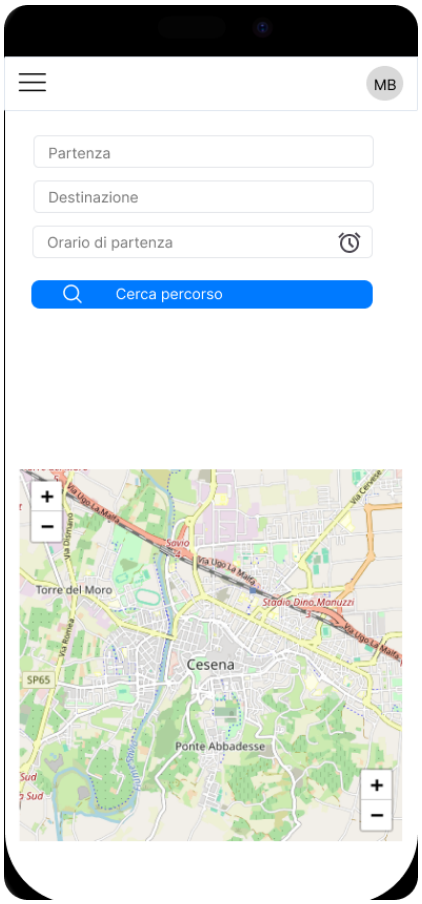

Figure 7: Navigazione Mobile - Ricerca percorso

Figure 8: Navigazione Mobile - Percorso trovato

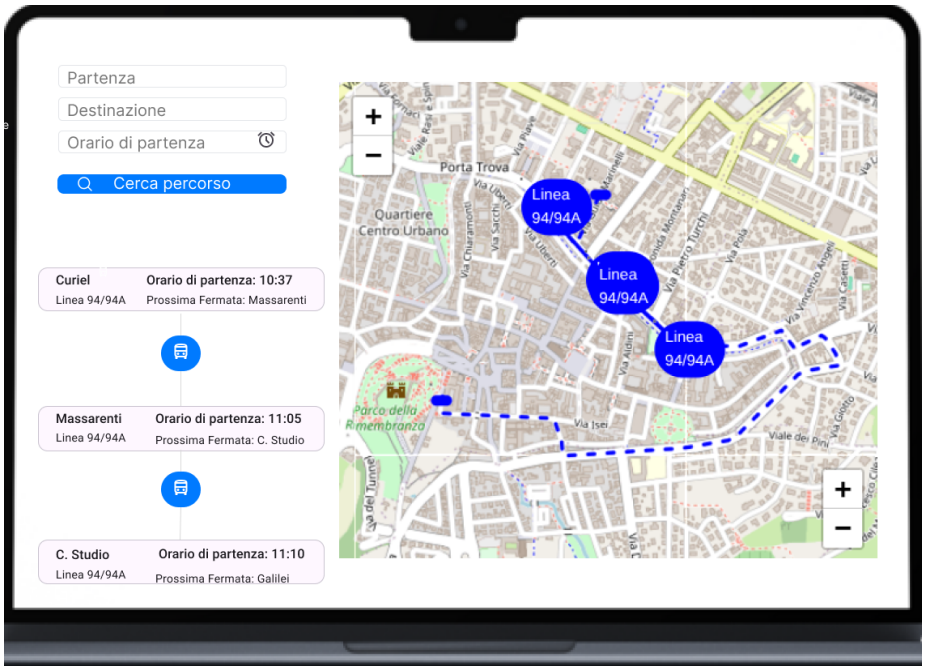

Figure 9: Navigazione - Versione Desktop

La pagina delle partenze in fig.12 completa il flusso principale dell'applicazione Citybus, offrendo agli utenti la possibilità di cercare le corse in arrivo presso una fermata specifica, a partire da un orario indicato. L'interfaccia è progettata per fornire un elenco chiaro delle corse previste, evidenziando lo stato di ciascuna di esse(in orario o in ritardo) e consentendo il monitoraggio dettagliato del percorso scelto. Dalla vista mobile si prevede di interrompere il monitoraggio della linea grazie ad un apposito pulsante che permetta di tornare alla vista precedente come si può vedere in fig.10 e fig.11.

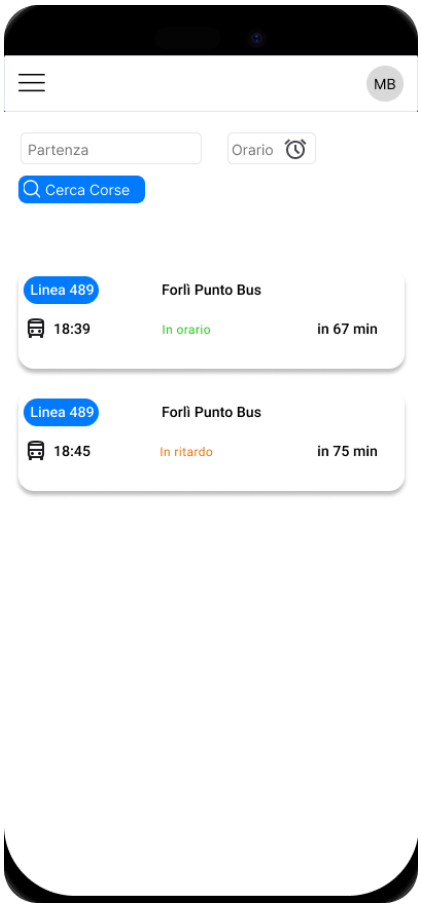

Figure 10: Partenze - Mobile: Ricerca corse per fermata e orario

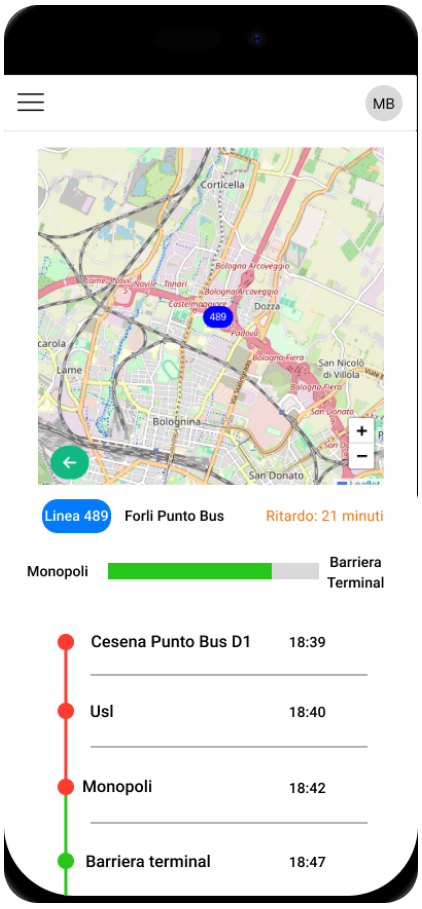

Figure 11: Partenze - Mobile: Monitoraggio corsa selezionata

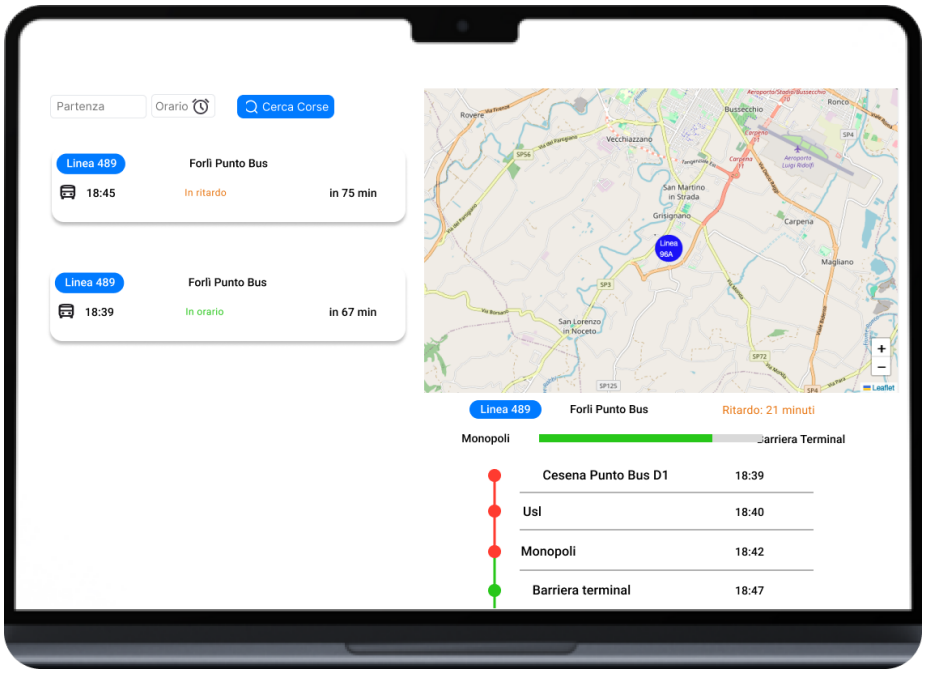

Figure 12: Partenze - Versione Desktop: ricerca e monitoraggio corse

Monitoraggio linee

La pagina mostrata in fig.13 e fig.14 è destinata agli amministratori dell'applicazione Citybus e fornisce una panoramica in tempo reale dello stato di tutte le corse attive sul territorio. Attraverso l'interfaccia, l'amministratore può monitorare ogni linea in servizio, filtrare le corse in ritardo e monitorare e seguire sulla mappa una corsa specifica una volta selezionata. Questa interfaccia è progettata per supportare gli operatori nella gestione del servizio in tempo reale, garantendo un controllo completo sulla regolarità delle corse e facilitando interventi rapidi in caso di criticità.

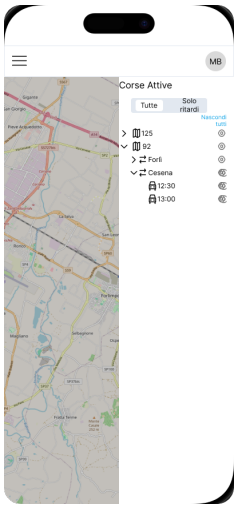

Figure 13: Monitoraggio linee - Desktop

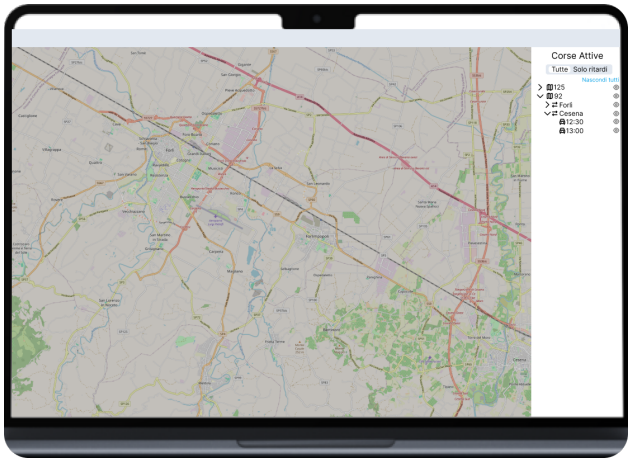

Figure 14: Monitoraggio linee - Mobile

Creazione linee

Il processo di creazione di una nuova linea è stato suddiviso in 3 step in modo da rendere il tutto più intuitivo e ordinato. Il primo step permette di inserire le informazioni di base necessarie per la configurazione del servizio. Gli utenti amministratori possono definire:

- il nome della linea
- le diverse direzioni associate alla linea, aggiungendo o rimuovendo voci in modo dinamico.

Le figure 16 e 15 mostrano questo primo step.

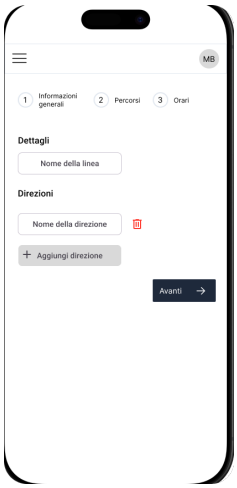

Figure 15: Creazione Nuova Linea - Mobile: Inserimento informazioni generali

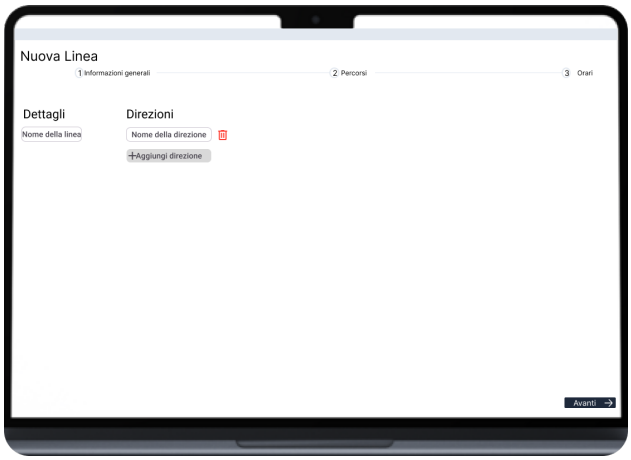

Figure 16: Creazione Nuova Linea - Desktop: Inserimento informazioni generali

Nel secondo step della procedura guidata mostrata in fig.17 e fig.18 per la creazione di una nuova linea, l'utente definisce il percorso dettagliato per ogni direzione configurata in precedenza. In questa schermata è possibile:

- inserire le fermate attraverso un elenco ordinato, scegliendo fermate già esistenti o aggiungendole direttamente dalla mappa
- visualizzare i tempi di percorrenza stimati tra le fermate, aggiornati dinamicamente
- generare automaticamente il percorso sulla mappa tramite il pulsante "Genera percorso", che calcola il tragitto più efficiente in base all'ordine delle fermate

La mappa interattiva mostra immediatamente la traiettoria generata, permettendo un riscontro visivo istantaneo sul percorso configurato. La suddivisione in tab per ogni direzione (es. Direzione 1, Direzione 2, Direzione 3) rende l'esperienza semplice e ordinata anche per linee complesse con più direzioni.

Figure 17: Creazione Nuova Linea - Mobile: Inserimento fermate e generazione percorso

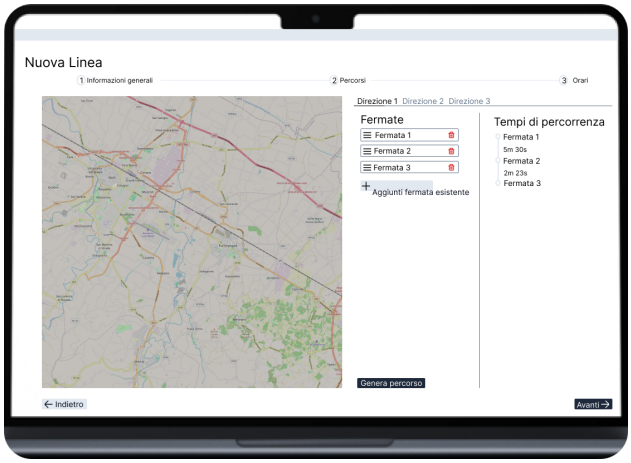

Figure 18: Creazione Nuova Linea - Desktop: Inserimento fermate e mappa percorso

Come si può vedere nelle fig.19 e 20, il terzo e ultimo step della procedura guidata per la creazione di una nuova linea consente all'amministratore di configurare gli orari di partenza per ciascuna direzione definita. Inserendo un orario di partenza, il sistema calcola automaticamente gli orari stimati di arrivo per tutte le fermate della direzione, sulla base dei tempi di percorrenza impostati nello step precedente.

Figure 19: Creazione Nuova Linea - Mobile: Inserimento orari di partenza e arrivo alle fermate

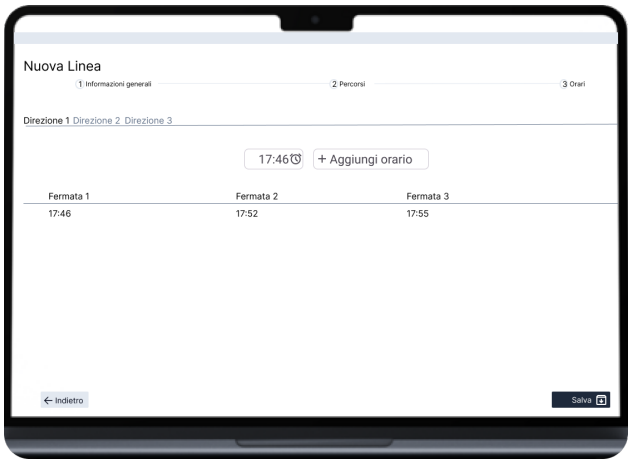

Figure 20: Creazione Nuova Linea - Desktop: inserimento orari e visualizzazione arrivi

Gestione linee

La pagina di gestione delle linee rappresenta il centro di controllo per gli amministratori, permettendo la supervisione completa di tutte le linee configurate nell'applicazione. Attraverso una tabella chiara e ordinata, è possibile:

- visualizzare in un colpo d'occhio tutte le linee con le relative direzioni, partenze e arrivi.
- selezionare una o più linee per effettuare azioni di modifica o eliminazione.
- Avviare la procedura di creazione di una nuova linea tramite il pulsante "Nuova Linea".

Questa schermata, disponibile nelle versioni mobile(fig.21 e desktop(fig.22, è progettata per semplificare al massimo la gestione della rete di trasporto

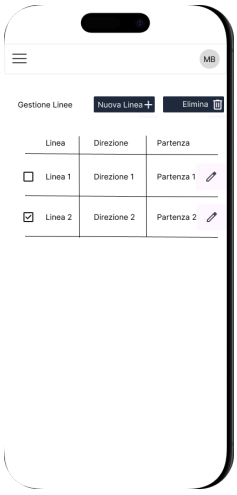

Figure 21: Gestione Linee - Mobile: tabella linee con azioni di modifica ed eliminazione

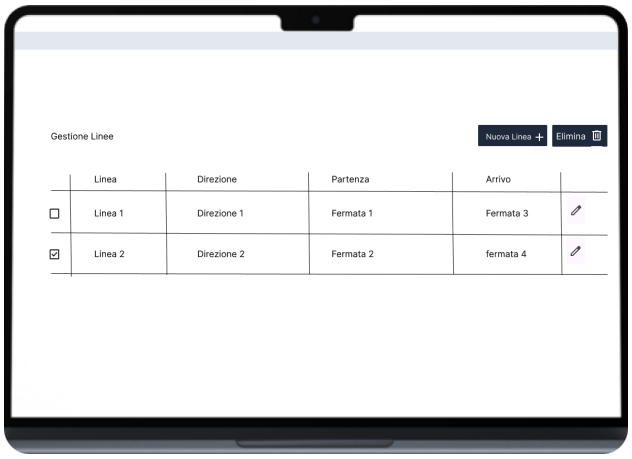

Figure 22: Gestione Linee - Desktop: panoramica linee e funzionalità di amministrazione

0.4 Tecnologie

0.4.1 Stack e librerie

Alla base dell'applicativo c'è lo stack MEVN (MongoDB, Express.js, Vue.js, Node.js); a questo sono state aggiunte sia librerie che forniscono componenti di UI sia librerie che implementano funzionalità essenziali per l'applicazione.

PrimeVue

PrimeVue è una libreria che fornisce componenti UI avanzati che permettono di realizzare interfacce con funzionalità avanzate.

Socket.io

Socket.io è una libreria che implementa una comunicazione in tempo reale tra server e client utilizzando diverse tecnologie, tra le quali c'è anche websocket. È stata utilizzata per i seguenti scopi:

- trasmettere la posizione degli autobus in tempo reale
- ottenere la posizione degli autobus in tempo reale per la dashboard di monitoraggio
- ottenere la posizione degli autobus in tempo reale per la navigazione

Leaflet e Vue Leaflet

Leaflet è una libreria che permette di integrare le mappe di OpenStreetMap all'interno di pagine web utilizzando codice JavaScript. Vue Leaflet è un wrapper che mette a disposizione le funzioni di Leaflet tramite componenti ed API Vue.js.

CSA.js

La libreria csa (Connection Scan Algorithm) è una libreria progettata per calcolare percorsi ottimali nei sistemi di trasporto pubblico, come autobus, treni o reti multimodali. Si basa sull'algoritmo Connection Scan, un algoritmo efficiente per la ricerca di percorsi minimi in orari di trasporto pubblico rappresentati come insiemi di "connessioni" (tratte tra due fermate con orari di partenza e arrivo).

0.4.2 Software di terze parti

Data l'elevata complessità del progetto, è stato necessario appoggiarsi a numerose tecnologie e progetti esterni. Di seguito verranno esplicitati tutti con una breve descrizione dell'utilizzo che ne abbiamo fatto.

OpenStreetMap

OpenStreetMap è un progetto che fornisce in modo gratuito mappe di tutto il mondo; è stato utilizzato come sorgente dati per la visualizzazione di tutte le mappe dell'applicativo.

Nominatim

Nominatim è il servizio che permette di cercare luoghi geografici tramite il loro nome (geocoding). Anche questo è un servizio open source e utilizza i dati di OpenStreetMap. È stato utilizzato sia all'interno del sistema di pianificazione dei percorsi che nell'interfaccia di navigazione per agevolare l'utente, permettendogli di cercare luoghi tramite il loro nome e non dover per forza selezionare punti dalla mappa o inserire manualmente le coordinate.

Project OSRM

Project OSRM è un routing engine open source che si appoggia ai dati di OpenStreetMap. La sua funzionalità principale è quella di calcolare il percorso più veloce da un punto A a un punto B. Viene utilizzato dall'interfaccia di pianificazione dei percorsi per generare in automatico il percorso migliore per ciascun autobus che passi per tutte le fermate definite dall'utente.

Redis

Redis è un database in-memory, questo significa che conserva tutti i dati che vengono inseriti al suo interno esclusivamente nella memoria RAM; questa sua

caratteristica gli permette di avere un accesso ai dati estremamente veloce. Redis implementa inoltre il paradigma publish/subscribe (in maniera simile al protocollo MQTT).

Grazie a queste sue caratteristiche è stato utilizzato per eseguire 2 diversi compiti.

- **memorizzazione posizione degli autobus**: La funzione di database in-memory è stata utilizzata per memorizza la posizione in tempo reale degli autobus. Dato che questa viene aggiornata con una frequenza molto elevata memorizzarla all'interno di MongoDB sarebbe stato sicuramente meno veloce ed efficiente e, dato che la dimensione del dato è estremamente ridotta, si presta ed essere memorizzata all'interno di Redis.
- **broker per i dati in tempo reale**: È stata sfruttata la funzione publish/subscribe per inviare i dati che gli autobus trasmettono in tempo reale ai client.

Docker

Docker è stato utilizzato sia in fase di sviluppo che per il deployment.

Tramite l'estensione `devcontainer` in fase di sviluppo è stato possibile creare un ambiente controllato e unificato che ha agevolato il lavoro, permettendo di avere in ciascuna macchina esattamente lo stesso ambiente e stack software.

Durante il deployment invece è stato estremamente utile sia per semplificare l'installazione di tutti i componenti e software di terze parti necessari, che per realizzare un'infrastruttura in cui è possibile scalare ogni singolo componente in base al carico e alle risorse utilizzate.

0.5 Codice

0.5.1 Struttura del progetto

Il progetto CityBus è suddiviso in due macro-componenti principali:

- Backend: Implementa la logica di business, l'accesso ai dati, la gestione delle linee, fermate, corse e il calcolo dei percorsi.
- Frontend: Applicazione web (SPA, Vue.js) per la consultazione, gestione e simulazione del servizio.

0.5.2 Backend

Gestione delle corse e simulazione

La gestione delle corse (`BusRide`) rappresenta uno degli aspetti centrali. Questo modulo consente di creare, aggiornare, monitorare e simulare le corse degli autobus in tempo reale, offrendo sia funzionalità amministrative che strumenti per la visualizzazione dinamica lato utente.

La gestione delle corse è affidata a due componenti principali: `BusRideManager` e `RideAgent`. Questi orchestrano la simulazione delle corse, il loro avanzamento e la comunicazione in tempo reale con i client tramite WebSocket e Redis. Il `BusRideManager` mantiene una lista di agenti (`RideAgent`) attivi e, tramite un ciclo periodico, verifica quali corse devono essere avviate o terminate. Per ogni corsa attiva, viene creato un nuovo agente che ne gestisce la simulazione.

```
// Estratto da BusRideManager.js
class BusRideManager {
    constructor() { this.agents = []; }
    async init(io) {
        this.io = io;
        this.startLoop();
    }
    async startLoop() {
        setInterval(async () => {
            // Per ogni corsa programmata
            const lines = await BusLine.find().populate('directions
                ');
            lines.map(line =>
                line.directions.map(direction =>
                    direction.timetable.map(async time => {
                        const ride = await BusRide.findOne({ /* ...
                             */ });
                        const agent = this.agents.find(ag => ag.
                            ride._id.toString() == ride._id.
                            toString());
                        if (!agent && ride) {
                            const newAgent = new RideAgent(ride,
                                this.io);
                            newAgent.start();
                            this.agents.push(newAgent);
                        }
                    })
                )
            );
        }, 10000);
    }
}
```

RideAgent e simulazione

Ogni `RideAgent` simula l'avanzamento della corsa, calcola la posizione del bus e comunica gli aggiornamenti tramite WebSocket. Se la corsa termina, l'agente la marca come conclusa e si ferma.

```
// Estratto da RideAgent.js
class RideAgent {
    constructor(ride, io) { /* ... */ }
    async start() {
        this.interval = setInterval(async () => {
            const position = await getBusPosition(this.ride);
            if (!position) {
                this.ride.stops.forEach(stop => stop.isBusPassed =
                    true);
```

```
                await this.ride.save();
                this.stop();
                return;
            }
            this.socket.emit("put", JSON.stringify(position));
        }, 1000);
    }
    stop() {
        clearInterval(this.interval);
        if (this.socket) this.socket.disconnect();
    }
}
```

Aggiornamento tramite WebSocket e Redis

Gli aggiornamenti di posizione vengono inviati tramite WebSocket ai client. Il controller (`socketsController.js`) riceve i dati, aggiorna lo stato della corsa e li propaga tramite Redis per garantire scalabilità e sincronizzazione tra più istanze.

```
// Estratto da socketsController.js
socket.on('put', async (position) => {
    const rideData = await calculateRealTimeRideData({rideId,
        position: JSON.parse(position)});
    await rideDataProvider.setRide(rideId, rideData); // Aggiorna
        Redis
});
rideDataEvent.onMessage((rideData) => {
    socket.emit('ride_update', rideData); // Inoltra ai client
});
```

Navigazione

La funzionalità di navigazione è affidata al modulo `pathFinder`, che implementa la logica per il calcolo dei percorsi ottimali tra due punti della rete di trasporto pubblico. Il cuore di questo componente è la funzione `getNavigationPath`, che sfrutta l'algoritmo Connection Scan (CSA) per trovare il percorso più efficiente combinando tratte a piedi e tratte in autobus.

Per ogni richiesta di percorso, il sistema:

- Identifica le fermate più vicine ai punti di partenza e arrivo tramite query geospaziali.
- Costruisce una lista di "connessioni" che rappresentano sia i tratti a piedi (dall'utente alla fermata e viceversa) sia le tratte in autobus, includendo orari di partenza e arrivo.
- Per ogni connessione bus, calcola gli orari effettivi in base alla tabella oraria della linea e della direzione.

```
// Estratto da pathFinder.js
conns.push({
    '@id': connectionId,
    travelMode: 'bus',
    departureStop: departureStopId,
    departureTime: departureTime,
    arrivalStop: arrivalStopId,
    arrivalTime: arrivalTime
});
// ...aggiunta delle connessioni bus e a piedi...
```

Le connessioni vengono ordinate per orario di partenza e passate a un planner CSA, che elabora il percorso ottimale tramite uno stream. Il risultato è una sequenza di tratte (bus e piedi) che minimizza il tempo totale di viaggio.

```
// Esecuzione del planner CSA
const planner = new csa.BasicCSA({
    departureStop: departureStop,
    arrivalStop: arrivalStop,
    departureTime: departureTime
});
const connectionsReadStream = Readable.from(conns);
connectionsReadStream.pipe(planner);

planner.on("result", function (result) {
    resolve(result);
});
```

Il percorso calcolato viene restituito come lista di tratte, ciascuna con informazioni su tipo di spostamento, orari, e fermate. Questa logica è utilizzata dal controller delle rotte per fornire agli utenti itinerari dettagliati e ottimizzati.

0.5.3 Frontend

Autenticazione e Registrazione: gestione dei Bearer Token

L'autenticazione e la registrazione nel frontend sono progettate per garantire sicurezza e praticità, facendo uso di Bearer Token (JWT) per la gestione delle sessioni utente. L'intero flusso si basa su chiamate HTTP sicure e sulla conservazione dei token in locale, con rinnovo automatico quando necessario.

Il login è gestito tramite `AuthenticationService`. Al successo, il backend restituisce un JWT e un token di rinnovo (`renewToken`). Questi vengono salvati nello store locale e utilizzati per autenticare tutte le richieste successive tramite l'header `Authorization: Bearer <jwt>`.

```
// src/service/AuthenticationService.js
export const AuthenticationService = {
    async login(email, password) {
        const res = await requests.post(`/auth/session`, { data: {
            email, password} })
        if (res.status == 201) {
            return {jwt: res.data.jwt, renewToken: res.data.
                renewToken}
        }
        throw 'Email o password non validi'
```

```
    }
}
```

```
// src/stores/authentication.js
export const useAuthenticationStore = defineStore('authentication', {
    state: () => ({
        jwt: localStorage.getItem('jwt'),
        renewToken: localStorage.getItem('renewToken')
    }),
    actions: {
        setTokens(jwt, renewToken) {
            this.jwt = jwt
            localStorage.setItem('jwt', jwt)
            this.renewToken = renewToken
            localStorage.setItem('renewToken', renewToken)
        },
        deleteTokens() {
            localStorage.removeItem('jwt')
            localStorage.removeItem('renewToken')
        }
    }
});
```

Prima di ogni richiesta autenticata, il frontend verifica la validità del JWT. Se il token è scaduto, viene automaticamente rinnovato utilizzando il `renewToken`. Tutte le richieste protette includono l'header `Authorization` con il Bearer Token.

```
// src/lib/requests.js
async function getAuthorizationHeader() {
    const authStore = useAuthenticationStore()
    const decoded = jwtDecode(authStore.jwt)
    if (decoded.exp < Math.floor(Date.now() / 1000) ) {
        await renewAuthenticationToken(authStore);
    }
    return `Bearer ${authStore.jwt}`
}

async function request(method, endpoint, data, authenticated) {
    const headers = {}
    if (authenticated) {
        headers['Authorization'] = await getAuthorizationHeader()
    }
    return await instance.request({
        url: endpoint,
        method: method,
        data: data,
        headers: headers
    })
}
```

0.6 Test

Le funzionalità del sistema sono state testate approfonditamente dal team su vari browser moderni (Google Chrome, Microsoft Edge, Mozilla Firefox, Safari), con l'obiettivo di verificarne il corretto funzionamento, la portabilità e la coerenza visuale in contesti differenti. In particolare, è stata posta attenzione all'interazione utente nelle schermate di ricerca partenze, monitoraggio corse e creazione/modifica delle linee, che rappresentano le aree a maggior complessità interattiva.

Anche le API REST sviluppate lato server sono state verificate tramite strumenti di test manuali e automatizzati, garantendone l'accurata restituzione dei dati relativi a fermate, linee e orari in tempo reale.

Per una valutazione qualitativa dell'esperienza utente, il sistema è stato sottoposto al test di usabilità secondo le 10 euristiche di Nielsen. Da questa analisi sono emerse le seguenti considerazioni:

- **Controllo e libertà:** sono state introdotte funzionalità per permettere all'utente di tornare sui propri passi (es. pulsanti "Indietro" tra gli step di creazione linea) e di correggere eventuali errori nei campi obbligatori, con validazioni visive attive. È stato inoltre previsto un meccanismo di guida progressiva step-by-step.

- **Aiuto utente:** nelle schermate più complesse (come la creazione della linea o la pagina di monitoraggio delle linee) sono stati aggiunti messaggi esplicativi e tooltip per spiegare le funzionalità. In caso di errore, vengono mostrate label o messaggi mirati (es. campo obbligatorio non compilato o formato errato).

- **Prevenzione errori:** il sistema impedisce all'utente di procedere con l'inserimento incompleto dei dati. Le azioni sono guidate da input validati, messaggi contestuali e feedback visivi costanti.

- **Riconoscimento più che ricordo:** l'utente non è costretto a ricordare nomi o riferimenti. Ad esempio, grazie all'autocompletamento per le fermate e all'uso di label intuitive nei percorsi, l'interazione si basa su selezione e riconoscimento immediato.

- **Visibilità dello stato del sistema:** l'interfaccia comunica in modo chiaro lo stato corrente dell'interazione, come il passo del processo attualmente attivo o la posizione del bus nella timeline. In alcune pagine, i dati si aggiornano dinamicamente per riflettere lo stato in tempo reale.

- **Corrispondenza tra sistema e mondo reale:** l'interfaccia adotta terminologie e concetti familiari all'utente (es. "fermata", "linea", "orario"), facilitando la comprensione senza richiedere formazione. Sono state aggiunte anche icone visive per rafforzare il messaggio semantico.

- **Coerenza e standard:** l'intero sistema adotta uno stile visivo coerente, con componenti UI standardizzati, colori uniformi e pattern ripetuti. Le

azioni comuni (modifica, salvataggio, eliminazione) sono riconoscibili in tutte le schermate.

- **Estetica e progettazione minimalista:** ogni schermata offre solo gli elementi essenziali per completare l'azione richiesta, evitando sovraccarico cognitivo. Lo spazio è utilizzato in modo bilanciato sia su desktop che mobile, garantendo chiarezza e ordine.
- **Flessibilità ed efficienza:** sebbene il sistema sia pensato per essere semplice e intuitivo, sono stati previsti comportamenti rapidi come l'aggiunta di fermate con un click e il riutilizzo di dati esistenti per semplificare operazioni ripetitive.
- **Aiuto e documentazione**: è stato integrato un sistema minimo di supporto utente tramite messaggi contestuali, tooltip e piccole guide inline nei punti critici del flusso. Vista la semplicità d'uso, si è ritenuto non necessario includere una guida esterna completa, privilegiando un design autoesplicativo.

Una volta completato lo sviluppo delle funzionalità e l'implementazione dell'interfaccia, il sistema è stato sottoposto all'attenzione delle due principali tipologie di utenti coinvolte: l'utente generico (passeggero) e l'amministratore del sistema (gestore delle linee). Questa fase ha permesso di valutare la correttezza dell'utilizzo del sistema da entrambe le prospettive, verificando che i flussi previsti risultassero comprensibili, intuitivi ed efficienti.

Nella fase finale, tutti i componenti del team si sono immedesimati nei target user individuati durante l'analisi, simulando casi d'uso realistici. Sono stati testati scenari tipici (es. consultazione partenze, simulazione corsa, modifica di una linea) e situazioni limite (es. ricerca incompleta, input errati, utilizzo su schermi mobili). Questa attività ha confermato la robustezza, chiarezza e coerenza dell'interfaccia, oltre alla capacità del sistema di guidare l'utente nella risoluzione autonoma di eventuali errori.

0.7 Deployment

Per il deployment dell'applicativo si è scelto di utilizzare Docker in modo da facilitare l'installazione e al contempo avere un'elevata modularità e semplicità di gestione dell'infrastruttura. Come si può vedere in figura 23 deployment è composto da 4 container: 2 contengono i database (MongoDB e Redis) e gli altri due contengono rispettivamente le API e l'interfaccia web. Si è scelto di separare API e frontend principalmente per una questione di scalabilità; questa separazione permette infatti di aumentare il numero di repliche di uno dei due componenti nel caso si rendesse necessario per questioni di prestazioni e di gestione del carico di lavoro, senza dover aumentare anche il numero di repliche dell'altro componente e quindi senza sprecare risorse.

Si è anche scelto di utilizzare il container contenente il frontend come entrypoint di tutto l'applicativo, senza quindi dover esporre direttamente nessuna

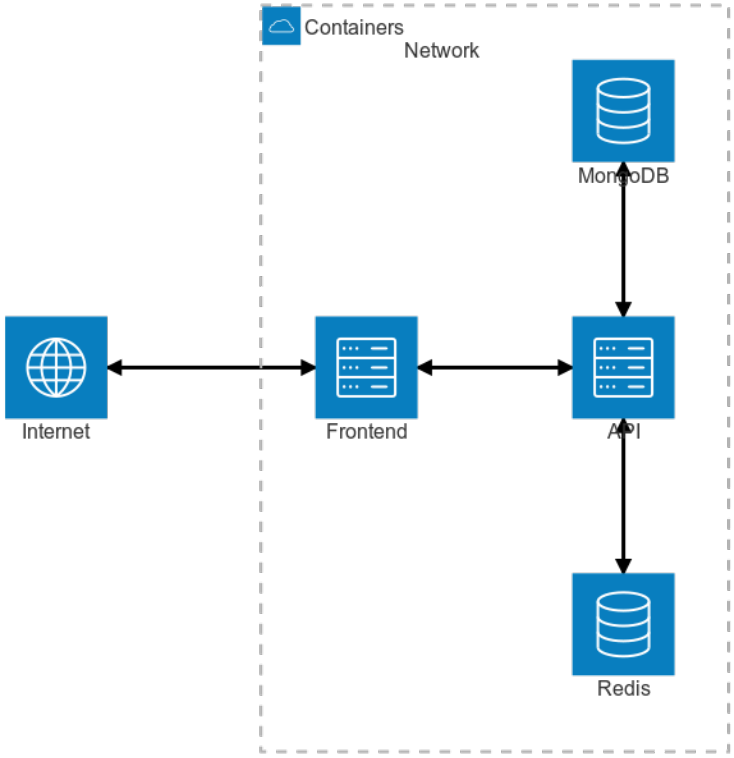

Figure 23: Schema deployment

porta dagli altri container; all'interno del container del frontend è infatti presente il web server NGINX che offre la possibilità di configurare un reverse proxy che permette di raggiungere altri web server (in questo caso quello delle API) senza che vengano esposti pubblicamente e mantenendo un unico punto di ingresso.

Per favorire maggiormente la scalabilità e il bilanciamento del carico si potrebbe fare un'ulteriore modifica a questa infrastruttura: si potrebbe utilizzare come entrypoint un'istanza dedicata di NGINX configurata in modo da comportarsi come load balancer facendo reverse proxy sia sul web server che espone le API sia su quello che espone l'interfaccia.

0.7.1 Istruzioni per il deployment

Eseguire il deployment dell'applicativo è un'operazione estremamente semplice: è sufficiente aprire un terminale, spostarsi nella root directory del repository ed

eseguire il comando `docker compose up -d`. Il compose file è strutturato in modo da creare automaticamente le immagini e avviare tutti i servizi.

È inoltre possibile caricare dei dati di esempio eseguendo il comando

```
1 docker compose exec mongo mongorestore -d citybus 'mongodb://root:
    password@mongo:27017/citybus?authSource=admin&directConnection=
    true' /dump
```

Una volta caricati i dati, sarà possibile eseguire il login con un utente di amministrazione appositamente creato; l'email per l'autenticazione è `admin@citybus.com` e la password è `admin`.

0.8 Conclusioni

Il progetto CityBus è stato senza dubbio molto impegnativo, ma anche estremamente stimolante. Il sistema sviluppato è complesso e ricco di funzionalità, integrate tra loro in modo efficace, e pensato per rispondere sia alle esigenze degli utenti target. La complessità non si ferma solamente ai componenti sviluppati dal team ma si estende anche alla difficoltà di integrare numerosi software e progetti esterni all'interno di un unico sistema.

Il risultato finale ci soddisfa pienamente: siamo riusciti a realizzare un'applicazione completa, ben strutturata e funzionante, che rispecchia fedelmente quanto ci eravamo prefissati.