

CINI Smart City University Challenge 2025

co-located with I-Cities 2025

Mobishare

Supervisori: Giuliana Franceschinis, Davide Cerotti

Team		
Nome e cognome	Tipo di corso di studio	Indirizzo email
<i>Simone Negro</i>	<i>Laurea triennale</i>	<i>20051767@studenti.uniupo.it</i>
<i>Matteo Schintu</i>	<i>Laurea triennale</i>	<i>20051769@studenti.uniupo.it</i>
<i>Cosimo Daniele</i>	<i>Laurea triennale</i>	<i>20051771@studenti.uniupo.it</i>

Indice

1 Definizione del sistema	3
1.1 Acronimi e abbreviazioni	4
1.2 Descrizione del sistema	5
1.2.1 Impieghi nell'ambito delle smart city	5
1.2.2 Scenari applicativi	5
1.3 Casi d'uso	6
1.4 Requisiti Funzionali	10
1.5 Requisiti non Funzionali	12
1.6 Descrizione dell'architettura	13
1.6.1 Architettura	13
1.6.2 Diagramma delle classi	13
1.6.3 Microservizi	14
1.6.4 Containerizzazione dei servizi	16
1.7 Interfacce dei servizi esposti del sistema	18
1.7.1 Funzionalità Utente - Visualizzazione Veicoli	18
1.7.2 Funzionalità Utente - Gestione Noleggi	18
1.7.3 Funzionalità Utente - Gestione Portafoglio	19
1.7.4 Funzionalità Utente - Localizzazione	19
1.8 Funzionalità Staff - Gestione Flotta	20
1.8.1 Funzionalità Staff - Gestione Parcheggi	22
1.9 Funzionalità Amministratore - Gestione Utenti	23
2 Progettazione di dettaglio del sistema	25
2.1 Infrastruttura interna	25
2.2 Infrastruttura esterna	26
2.2.1 Rete WI-FI con Access Point distribuiti	26
2.2.2 Rete cellulare 4G/5G(Proposta di Miglioramento)	27
2.3 Circuito comunicazione GPS	27
2.3.1 ARDUINO UNO REV2 WI-FI	28
2.3.2 Modulo GPS-NEO 6M	28
2.3.3 Schema elettrico	29
2.4 AI Agent	30
2.4.1 Introduzione e Concetto	30
2.4.2 Struttura del Codice Sorgente	30
2.4.3 Architettura e Flusso Decisionale	30
2.4.4 Configurazione e Avvio	31
2.5 Prerequisiti	31
2.6 Flusso di Installazione e Configurazione	32
2.6.1 Clonare il Repository	32
2.6.2 Configurare i Segreti dell'Applicazione	32
2.6.3 Installare i Modelli AI	32
2.6.4 Creare e Aggiornare il Database	32
2.7 Avviare l'Applicazione	33
3 Il sistema realizzato	34
3.1 Interfaccia Utente (Homepage)	34
3.2 Pannello admin	36
4 RESTful API Interfaces	38

1 Definizione del sistema

Il progetto consiste nello sviluppo di un sistema di gestione per un servizio di mobilità sostenibile denominato **Mobishare**, attivo nella città di tutta Italia. Il sistema consente agli utenti registrati di utilizzare mezzi di trasporto in sharing (biciclette muscolari, biciclette elettriche e monopattini elettrici), prelevandoli e restituendoli presso apposite aree di parcheggio.

Gli obiettivi principali del progetto sono:

- Gestione dell'autenticazione degli utenti e del credito associato
- Prenotazione e utilizzo dei mezzi tramite un front-end
- Calcolo del costo delle corse in base alla durata e al tipo di mezzo
- Rilevamento e segnalazione di malfunzionamenti o cariche basse
- Amministrazione del sistema da parte del gestore, che può intervenire per ricariche, riparazioni, gestione degli utenti sospesi e redistribuzione dei mezzi

Una parte integrante del progetto è rappresentata dall'implementazione di un **chatbot intelligente**, che amplia le modalità di interazione con il sistema Mobishare. Il chatbot è stato progettato per:

- Assistere gli utenti nelle operazioni più comuni, come la prenotazione di un mezzo o la segnalazione di un problema
- Interfacciarsi direttamente con il database attraverso un sistema di **tooling** che consente l'esecuzione guidata di query e operazioni
- Automatizzare l'apertura di ticket di assistenza, la verifica del credito disponibile e la richiesta di sblocco account
- Offrire un canale alternativo e conversazionale per accedere alle funzionalità già presenti nella piattaforma, migliorando l'accessibilità e l'esperienza utente

Il sistema è stato progettato secondo un'**architettura a microservizi**, comprendente:

- **Backend REST** per la gestione dei dati (utenti, mezzi, parcheggi, corse, ecc.)
- **Interfaccia utente** per l'accesso alle funzionalità da parte di utenti finali e gestori
- **Modulo IoT** per la simulazione dell'interazione con sensori e attuatori (ad es. spie luminose, stato della batteria)
- **Sistema di suggerimento** per la redistribuzione ottimale dei mezzi in base ai dati storici di utilizzo

Il progetto è stato sviluppato seguendo le seguenti fasi:

- Specifica dei requisiti e dei casi d'uso.
- Progettazione dei microservizi, delle API REST, dei topic MQTT e dei messaggi.
- Integrazione del chatbot all'interno del sistema complessivo.
- Implementazione dei componenti con tecnologia Java (e altre tecnologie se necessarie).
- Testing unitario e di integrazione, sia dei microservizi sia del chatbot.
- Preparazione di una demo finale per illustrare le funzionalità principali e i casi d'uso più rappresentativi.

1.1 Acronimi e abbreviazioni

- **IoT:** Internet of Things (Internet delle Cose)
- **REST:** Representational State Transfer (stile architettonico per sistemi web)
- **API:** Application Programming Interface
- **MQTT:** Message Queuing Telemetry Transport (protocollo di messaggistica che implementa il paradigma publish/subscribe)
- **UML:** Unified Modeling Language (linguaggio di modellazione)
- **UI:** User Interface (Interfaccia Utente)
- **DB:** Database

1.2 Descrizione del sistema

Il sistema **Mobishare** è una piattaforma innovativa per la mobilità sostenibile che mira a rivoluzionare il trasporto urbano nelle città italiane. Gli obiettivi principali includono:

- Ridurre il traffico veicolare e l'inquinamento atmosferico
- Promuovere stili di vita attivi attraverso l'uso di biciclette
- Ottimizzare la distribuzione dei mezzi di trasporto in base alla domanda
- Integrare soluzioni IoT e AI per una gestione intelligente delle risorse

1.2.1 Impieghi nell'ambito delle smart city

Mobishare si inserisce perfettamente nel contesto delle smart city contribuendo a:

- **Mobilità sostenibile:** Offre alternative ecologiche ai veicoli privati
- **Efficienza energetica:** Riduce il consumo di combustibili fossili
- **Inclusione sociale:** Accessibile a diverse fasce della popolazione
- **Gestione dati urbani:** Fornisce insights per la pianificazione cittadina
- **Integrazione IoT:** Monitoraggio in tempo reale dello stato dei veicoli

1.2.2 Scenari applicativi

1. **Scenario pendolarismo:** Un utente prenota un monopattino elettrico per raggiungere la stazione, evitando il traffico mattutino e riducendo le emissioni di CO₂
2. **Scenario turismo sostenibile:** Visitatori utilizzano biciclette per esplorare il centro storico, decongestionando le aree più trafficate
3. **Scenario emergenza parcheggi:** Il sistema ridistribuisce i veicoli in aree sovraffollate grazie all'analisi predittiva
4. **Scenario manutenzione preventiva:** Sensori IoT rilevano anomalie nei veicoli prima che causino guasti

1.3 Casi d'uso

Utente

L'utente è il principale utilizzatore dell'applicazione Mobishare. Può prenotare veicoli, avviare corse, gestire il proprio wallet e interagire con un agente AI per ottenere supporto.

UC1 – Prenotare un veicolo

Attore principale: Utente

Scopo: Selezionare e prenotare un veicolo disponibile

Precondizioni: Utente autenticato, veicolo disponibile, credito sufficiente nel wallet

Flusso principale:

1. L'utente visualizza i veicoli vicini sulla mappa
2. Seleziona un veicolo
3. Conferma la prenotazione

Postcondizione: Il veicolo risulta prenotato

UC2 – Iniziare una corsa

Scopo: Avviare una corsa con un veicolo prenotato

Precondizioni: Veicolo prenotato correttamente

Flusso principale:

1. L'utente sblocca il veicolo
2. Il sistema avvia la corsa e inizia il tracking

Postcondizione: La corsa è attiva e conteggiata

UC3 – Caricare credito sul Wallet

Scopo: Aggiungere fondi al portafoglio utente

Flusso principale:

1. L'utente accede al Wallet
2. Inserisce l'importo e seleziona il metodo di pagamento
3. Conferma la transazione

Postcondizione: Il saldo è aggiornato

UC4 – Visualizzare storico corse

Scopo: Consultare l'elenco delle corse precedenti

Flusso principale:

1. L'utente accede alla sezione storico
2. Il sistema mostra l'elenco delle corse

UC5 – Visualizzare storico Wallet

Scopo: Visualizzare movimenti e transazioni del Wallet

Flusso principale:

1. L'utente apre la sezione Wallet
2. Il sistema mostra le transazioni passate

UC6 – Scrivere un feedback

Scopo: Inviare un commento o valutazione su un servizio

Flusso principale:

1. L'utente seleziona un oggetto (es. corsa)
2. Compila il feedback
3. Invia il commento

UC7 – Aprire un report

Scopo: Segnalare un problema o malfunzionamento

Flusso principale:

1. L'utente compila una segnalazione
2. Il sistema la registra e la inoltra

UC8 – Utilizzare punti accumulati (futura implementazione)

Scopo: Usare punti per sconti o premi

Flusso principale:

1. L'utente apre la sezione punti
2. Seleziona il tipo di conversione o bonus
3. Conferma

UC9 – Interagire con AI Agent

Scopo: Richiedere assistenza o suggerimenti tramite un assistente virtuale

Flusso principale:

1. L'utente apre la chat con l'agente AI
2. Scrive una richiesta (es. informazioni, supporto, posizione veicoli)
3. L'agente risponde o attiva strumenti integrati (tooling)

AI Agent

L'AI Agent è un assistente virtuale integrato nel sistema Mobishare, che interagisce con l'utente fornendo risposte contestuali e automatizzando azioni tramite un tooling system. A seconda del contesto della richiesta dell'utente, può rispondere direttamente o agire per conto dell'utente eseguendo operazioni come prenotazioni o gestione corse.

UC10 – Rispondere a domande sull'applicazione

Scopo: Fornire risposte informative relative all'utilizzo dell'app Mobishare o alle funzionalità offerte

Flusso principale:

1. L'utente invia una domanda o richiesta di informazioni all'AI Agent
2. L'agente analizza il contesto della domanda
3. L'agente fornisce una risposta testuale o suggerimenti rilevanti

UC11 – Eseguire azioni per conto dell’utente

Scopo: Automatizzare azioni dell’utente come prenotare veicoli, iniziare o terminare corse, utilizzando un tooling system

Flusso principale:

1. L’utente richiede un’azione (es. “prenota un veicolo vicino”, “inizia corsa”)
2. L’AI Agent interpreta la richiesta e verifica i permessi
3. L’AI Agent esegue l’azione utilizzando il tooling system integrato
4. L’AI Agent conferma all’utente l’esito dell’operazione

Staff

Lo Staff è l’utente interno al sistema Mobishare incaricato della gestione amministrativa e operativa delle risorse, quali veicoli, aree e parcheggi. Le sue azioni comprendono operazioni di creazione, lettura, aggiornamento e cancellazione (CRUD).

UC12 – Gestire tipologie di veicoli

Scopo: Aggiungere, modificare, visualizzare o eliminare tipologie di veicoli disponibili nel sistema

Flusso principale:

1. Lo Staff accede alla sezione tipologie veicoli
2. Esegue operazioni CRUD sulle tipologie (es. aggiungere nuovo tipo, aggiornare nome, eliminare)

UC13 – Gestire singoli veicoli

Scopo: Amministrare i veicoli specifici, inclusa la loro assegnazione a tipologie e aree

Flusso principale:

1. Lo Staff visualizza l’elenco dei veicoli
2. Aggiunge, modifica o elimina veicoli
3. Assegna i veicoli alle aree o parcheggi appropriati

UC14 – Gestire aree della città

Scopo: Definire e modificare le aree geografiche coperte dal servizio Mobishare

Flusso principale:

1. Lo Staff crea o modifica aree della città nel sistema
2. Visualizza e aggiorna confini o attributi delle aree

UC15 – Gestire aree di parcheggio

Scopo: Amministrare le aree destinate al parcheggio dei veicoli

Flusso principale:

1. Lo Staff aggiunge nuove aree di parcheggio
2. Modifica o elimina aree esistenti

UC16 – Visualizzare operazioni del tecnico

Scopo: Monitorare e visualizzare le attività svolte dal tecnico sul sistema

Flusso principale:

1. Lo Staff accede alla sezione operazioni tecnico
2. Visualizza le attività e i report correlati

Tecnico

Il Tecnico è responsabile della manutenzione e gestione operativa dei veicoli. Le sue attività principali includono la gestione dei ticket di manutenzione e il monitoraggio dello stato dei veicoli.

UC17 – Gestire manutenzione veicoli (futura implementazione)

Scopo: Effettuare la manutenzione dei veicoli, incluso il caricamento e aggiornamento dello stato

Flusso principale:

1. Il Tecnico visualizza i veicoli assegnati o in manutenzione
2. Aggiorna lo stato di carica o condizioni del veicolo
3. Registra interventi eseguiti

UC18 – Gestire ticket di riparazione (futura implementazione)

Scopo: Visualizzare e risolvere i ticket aperti relativi a problemi o guasti dei veicoli

Flusso principale:

1. Il Tecnico accede all'elenco dei ticket aperti
2. Seleziona un ticket da gestire
3. Esegue le operazioni di riparazione o manutenzione
4. Aggiorna lo stato del ticket (risolto, in lavorazione, ecc.)

Admin

L'Admin è l'utente con privilegi completi sul sistema Mobishare. Può eseguire tutte le operazioni di Utente, AI Agent, Staff e Tecnico, oltre a gestire la sicurezza e i permessi degli utenti.

UC19 – Gestire tutte le funzionalità di Utente, AI Agent, Staff e Tecnico

Scopo: Avere accesso completo a tutte le funzionalità offerte dal sistema per garantire la supervisione e la gestione integrale

Flusso principale:

1. L'Admin utilizza le funzionalità di prenotazione, gestione veicoli, manutenzione, AI Agent, ecc.

UC20 – Bloccare account utente

Scopo: Gestire la sicurezza bloccando gli account degli utenti in caso di violazioni o problemi

Flusso principale:

1. L'Admin visualizza l'elenco degli utenti
2. Seleziona un account da bloccare
3. Esegue il blocco dell'account, impedendo ulteriori accessi

Postcondizione: L'utente bloccato non può più accedere al sistema fino a nuovo intervento dell'Admin.

1.4 Requisiti Funzionali

- **FR1 - Registrazione utente:** L'utente può creare un nuovo account personale.
(User, Technician, Staff, Admin)
- **FR2 - Login utente:** L'utente può autenticarsi con email e password.
(User, Technician, Staff, Admin)
- **FR3 - Logout utente:** L'utente può uscire dal proprio account.
(User, Technician, Staff, Admin)
- **FR4 - Recupero password:** L'utente può recuperare la password dimenticata tramite email.
(User, Technician, Staff, Admin)
- **FR5 - Login tramite Google:** L'utente può autenticarsi tramite Google OAuth.
(User, Technician, Staff, Admin)
- **FR6 - Avvio nuova conversazione:** L'utente può iniziare una chat con il chatbot.
(User, Technician, Staff, Admin, AI Agent)
- **FR7 - Invio messaggio chat:** L'utente può inviare messaggi nella chat.
(User, Technician, Staff, Admin, AI Agent)
- **FR8 - Ricezione messaggio chat:** L'utente riceve risposte dal chatbot in tempo reale.
(User, Technician, Staff, Admin, AI Agent)
- **FR9 - Visualizzazione storico chat:** L'utente può vedere le conversazioni precedenti.
(User, Technician, Staff, Admin, AI Agent)
- **FR10 - Visualizzazione veicoli disponibili:** L'utente può vedere i veicoli disponibili sulla mappa.
(User, Technician, Staff, Admin, AI Agent)
- **FR11 - Prenotazione veicolo:** L'utente può prenotare un veicolo libero.
(User, Technician, Staff, Admin, AI Agent)
- **FR12 - Avvio viaggio:** L'utente può iniziare un viaggio con un veicolo prenotato.
(User, Technician, Staff, Admin, AI Agent)
- **FR13 - Termine viaggio:** L'utente può terminare un viaggio in corso.
(User, Technician, Staff, Admin, AI Agent)
- **FR14 - Visualizzazione storico viaggi:** L'utente può vedere lo storico dei viaggi effettuati.
(User, Technician, Staff, Admin, AI Agent)
- **FR15 - Visualizzazione saldo:** L'utente può vedere il proprio saldo disponibile.
(User, Technician, Staff, Admin)
- **FR16 - Ricarica saldo:** L'utente può ricaricare il saldo tramite PayPal.
(User, Technician, Staff, Admin)
- **FR17 - Pagamento viaggio:** Il sistema scala il costo del viaggio dal saldo utente.
(User, Technician, Staff, Admin)
- **FR18 - Visualizzazione mappa:** L'utente può vedere la mappa con città, parcheggi e veicoli.
(User, Technician, Staff, Admin, AI Agent)
- **FR19 - Creazione area parcheggio:** Creazione di una nuova area di parcheggio.
(Technician, Staff, Admin)

- **FR20 - Modifica area parcheggio:** Modifica di un'area di parcheggio esistente.
(Technician, Staff, Admin)
- **FR21 - Eliminazione area parcheggio:** Eliminazione di un'area di parcheggio.
(Technician, Staff, Admin)
- **FR22 - Segnalazione problema:** L'utente può segnalare problemi relativi a veicoli o servizio.
(User, Technician, Staff, Admin)
- **FR23 - Visualizzazione segnalazioni:** Visualizzazione di tutte le segnalazioni ricevute.
(Technician, Staff, Admin)
- **FR24 - Gestione segnalazione:** Aggiornamento dello stato di una segnalazione.
(Technician, Staff, Admin)
- **FR25 - Creazione veicolo:** Aggiunta di un nuovo veicolo.
(Technician, Staff, Admin)
- **FR26 - Modifica veicolo:** Modifica dei dati di un veicolo.
(Technician, Staff, Admin)
- **FR27 - Eliminazione veicolo:** Eliminazione di un veicolo.
(Technician, Staff, Admin)
- **FR28 - Creazione tipo veicolo:** Aggiunta di una nuova tipologia di veicolo.
(Staff, Admin)
- **FR29 - Modifica tipo veicolo:** Modifica di una tipologia di veicolo.
(Staff, Admin)
- **FR30 - Eliminazione tipo veicolo:** Eliminazione di una tipologia di veicolo.
(Staff, Admin)
- **FR31 - Creazione città:** Aggiunta di una nuova città.
(Staff, Admin)
- **FR32 - Modifica città:** Modifica di una città esistente.
(Staff, Admin)
- **FR33 - Eliminazione città:** Eliminazione di una città.
(Staff, Admin)
- **FR34 - Risposta automatica AI:** Il sistema fornisce risposte automatiche tramite AI alle domande degli utenti.
(User, Technician, Staff, Admin, AI Agent)
- **FR35 - Esecuzione tool automatici AI:** Il sistema può eseguire azioni automatiche su richiesta dell'AI.
(User, Technician, Staff, Admin, AI Agent)
- **FR36 - Notifiche in tempo reale:** Il sistema invia notifiche e aggiornamenti in tempo reale agli utenti.
(User, Technician, Staff, Admin, AI Agent)
- **FR37 - Gestione utenti:** L'admin può bloccare o sbloccare altri utenti.
(Admin)

1.5 Requisiti non Funzionali

- **NFR1 - Sicurezza:** Il sistema deve proteggere i dati degli utenti tramite autenticazione, autorizzazione, HTTPS e cifratura delle password e delle comunicazioni, inclusi i messaggi MQTT.
- **NFR2 - Affidabilità:** Il sistema deve essere resiliente agli errori e garantire una disponibilità costante dei servizi, anche in caso di disconnessioni temporanee dei dispositivi MQTT.
- **NFR3 - Scalabilità:** L'architettura deve poter gestire un numero crescente di utenti, veicoli (Arduino) e messaggi MQTT, mantenendo performance stabili.
- **NFR4 - Prestazioni:** Le operazioni principali (login, prenotazione, tracking, AI chat) devono essere completate entro 2 secondi nel 95% dei casi.
- **NFR5 - Usabilità:** L'interfaccia sviluppata in Razor Pages e JavaScript deve essere semplice, responsiva e adatta a utenti di ogni livello.
- **NFR6 - Compatibilità:** Il sistema deve essere utilizzabile su browser moderni (Chrome, Firefox, Edge, Safari) e dispositivi mobili.
- **NFR7 - Interoperabilità:** Il sistema deve integrarsi correttamente con:
 - API esterne (es. Google Maps, AI Ollama)
 - Dispositivi Arduino tramite MQTT (per tracking GPS, stato veicolo)
- **NFR8 - Manutenibilità:** Il codice sviluppato in .NET, JavaScript e Razor deve essere strutturato e ben documentato per agevolare modifiche future.
- **NFR9 - Modularità:** Il sistema deve essere suddiviso in moduli indipendenti (chat AI, backend, interfaccia, servizi MQTT).
- **NFR10 - Disponibilità:** Il sistema deve garantire una disponibilità minima del 99% su base mensile, anche tramite infrastruttura gestita con Proxmox.
- **NFR11 - Logging e Monitoraggio:** Tutte le operazioni critiche (accessi, comandi AI, movimenti veicolo) devono essere tracciate tramite un sistema di logging.
- **NFR12 - Privacy:** Il sistema deve rispettare il GDPR, permettendo agli utenti di visualizzare, modificare o cancellare i propri dati.
- **NFR13 - Aggiornabilità:** Il sistema deve permettere l'aggiornamento indipendente di componenti (chatbot AI, API, interfaccia) senza blocchi.
- **NFR14 - Testabilità:** Il progetto deve prevedere test unitari e funzionali per le API, il frontend e la comunicazione MQTT.
- **NFR15 - Documentazione:** Deve essere fornita documentazione tecnica aggiornata per sviluppatori, utenti e amministratori.
- **NFR16 - Backup e Recovery:** I dati devono essere sottoposti a backup regolare e il sistema deve prevedere procedure di ripristino.
- **NFR17 - Efficienza energetica:** L'applicazione web e i dispositivi Arduino devono essere ottimizzati per ridurre il consumo energetico.
- **NFR18 - Multiutenza:** Il sistema deve supportare più utenti e più dispositivi MQTT contemporaneamente, senza conflitti.
- **NFR19 - AI e tempo di risposta:** Il chatbot AI (Ollama) deve rispondere entro 2 secondi nel 90% dei casi, e il servizio deve poter essere aggiornato o sostituito facilmente.

1.6 Descrizione dell'architettura

1.6.1 Architettura

Il sistema è stato progettato secondo un'architettura a microservizi che include:

- Backend REST per la gestione dati
- Interfaccia utente multiplataforma
- Modulo IoT per l'interazione con sensori e attuatori
- Chatbot intelligente con integrazione di tooling avanzato
- Sistema di suggerimento per la redistribuzione ottimale dei mezzi

1.6.2 Diagramma delle classi

AspNetUser viene rappresentato dal AspNetUserClaims per dargli il tipo utente quindi : Utente, Staff, Tecnico e Admin

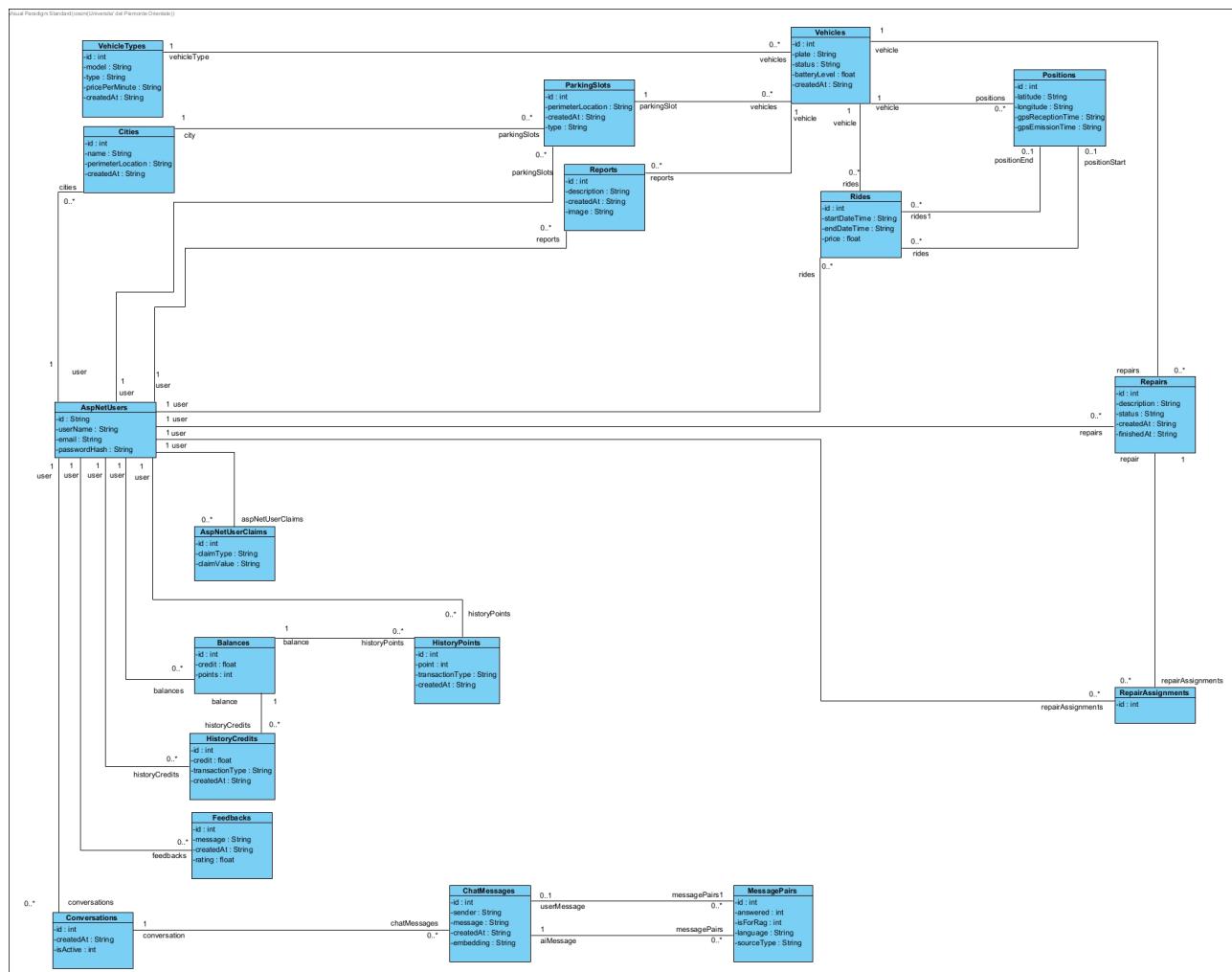


Figura 1: Diagramma delle classi del dominio

1.6.3 Microservizi

L'applicazione è stata sviluppata utilizzando .NET, in Figura 2 si può vedere uno schema dei microservizi mentre in Figura 3 viene mostrata più in dettaglio l'architettura completa.

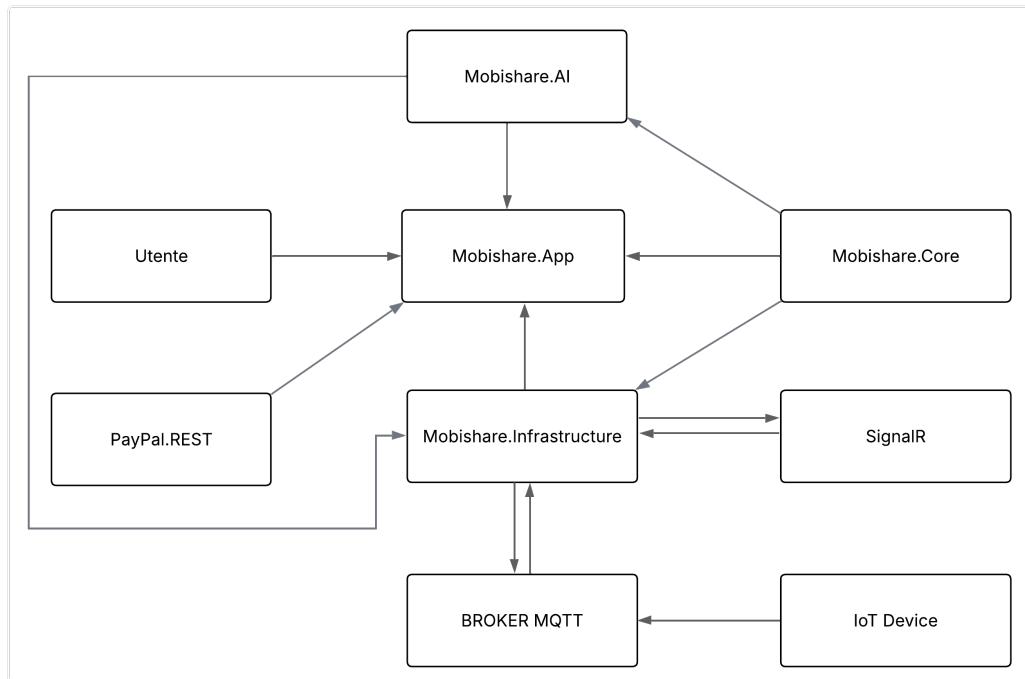


Figura 2: Diagramma dei Microservizi

Mobishare.App (Frontend Web)

Ruolo:

Interfaccia utente web realizzata tramite Razor Pages che gestisce:

- le richieste degli utenti
- visualizzazione mappa
- prenotazione veicoli
- cronologia viaggi
- wallet
- chat(con AI agent).

Dipendenze:

Comunica tramite HTTP/REST e SignalR con **Mobishare.Core** e **Mobishare.Infrastructure**.

Mobishare.Core (Servizi di Dominio e API)

Ruolo:

Contiene:

- la logica di dominio
- i modelli
- i comandi/handler MediatR
- la gestione delle entità (veicoli, utenti, viaggi, parcheggi, ecc.).

Espone API per la gestione di:

- veicoli
- posizioni
- utenti
- viaggi
- città
- parcheggi.

Dipendenze:

- Database (gestito tramite ApplicationDbContext)
- Mobishare.Infrastructure per servizi esterni (es. SignalR, MQTT, AI, ecc.)

Mobishare.Infrastructure (Servizi di Integrazione)

Ruolo:

Gestione servizi esterni e infrastrutturali:

- SignalR: aggiornamenti in tempo reale (posizione veicoli, timer, chat)
- MQTT: ricezione dati da dispositivi IoT/Arduino (posizione veicoli)
- ChatBotAIService: servizi AI/chatbot
- TimerService: gestione timer di prenotazione

Dipendenze:

- Mobishare.Core (per modelli e logica di dominio)
- Servizi esterni (broker MQTT, SignalR Hub, ecc.)

PayPal.REST (Microservizio Pagamenti)

Ruolo:

Gestione dei pagamenti tramite PayPal REST API.

Espone servizi per:

- ricarica del wallet
- pagamenti viaggi.

Dipendenze:

Mobishare.App e Mobishare.Core possono invocare le sue API per effettuare le operazioni di pagamento.

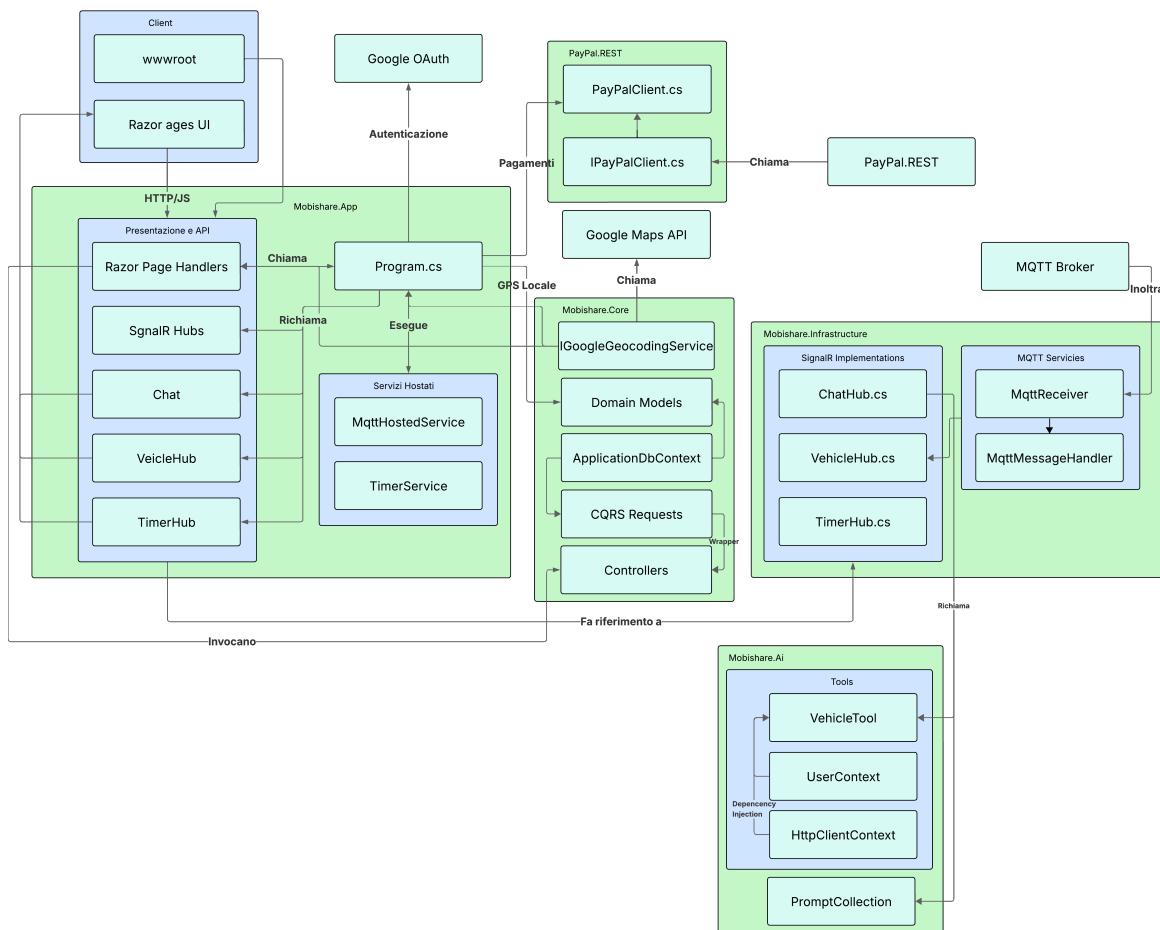


Figura 3: Diagramma Architetturale

1.6.4 Containerizzazione dei servizi

La containerizzazione dei servizi è realizzata su Proxmox (si veda la Sezione 2). Il cluster Proxmox ospita tre container LXC (piattaforma per container integrata di Proxmox), ciascuno dedicato a un servizio specifico del sistema:

.NET

- Ospita il core service dell'applicazione Mobishare, sviluppato in .NET;
- gestisce prenotazione, autenticazione, pagamenti, registrazione utenti e interfacciamento con i database;
- è il punto di accesso principale per le app mobili e per i dispositivi smart.

Assegnazione delle risorse:

- 20-25% CPU;
- 15-20% RAM;
- 10-15% DISCO.

Il servizio .NET è generalmente "leggero" a livello di CPU quando si tratta solo di gestire richieste web/API.
La RAM dipende dalla complessità.

Il disco contiene solo il runtime e log, perciò lo spazio può essere contenuto.

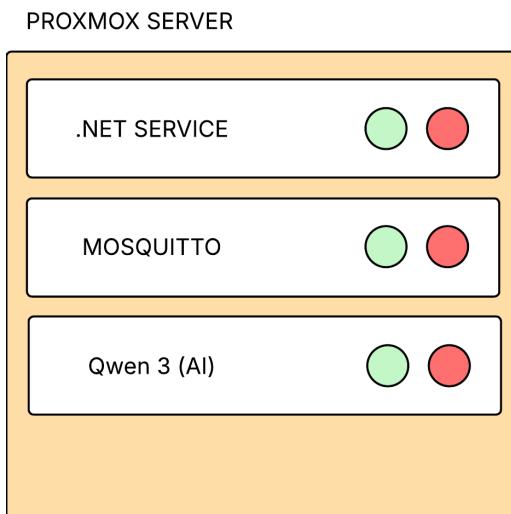


Figura 4: Struttura dei container LXC e allocazione dei servizi

MQTT Mosquitto

- Ospita il broker MQTT per la comunicazione tra dispositivi smart e il sistema;
- utilizzato per scambi rapidi di messaggi (es. aggiornamento posizione, notifiche, segnalazioni);
- leggero ed efficiente, ideale per IoT e comunicazioni asincrone.

Assegnazione risorse:

- 5-10% CPU;
- 5-10% RAM;
- 5% DISCO.

Le risorse assegnate al broker rimangono limitate, in quanto la gestione delle richieste MQTT è leggera.

Modello IA – Qwen3

- Contiene il modello di intelligenza artificiale Qwen3;
- richiede un'ampia dotazione di risorse (CPU, RAM, e opzionalmente GPU).

Assegnazione risorse:

- 65-75% CPU;
- 70-80% RAM;
- 80-85% DISCO.

Le risorse assegnate al modello devono essere elevate, questo perché Qwen3 (ma anche altri modelli IA) richiedono alte prestazioni di calcolo e grandi spazi di memoria per poter operare.

1.7 Interfacce dei servizi esposti del sistema

1.7.1 Funzionalità Utente - Visualizzazione Veicoli

Nome	GetAllVehicles	
Requisiti di riferimento	Visualizzazione mappa veicoli disponibili per noleggio	
Descrizione	Recupera tutti i veicoli disponibili con posizione e stato batteria	
Input	Nome Parametro	Descrizione del parametro
(*) = obbligatorio	Nessun parametro	Endpoint pubblico senza autenticazione
Output	Descrizione dell'output Lista veicoli con ID, tipo, posizione GPS e livello batteria	
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5	

Nome	GetVehicleById	
Requisiti di riferimento	Dettagli specifici veicolo prima del noleggio	
Descrizione	Recupera informazioni dettagliate di un veicolo selezionato	
Input	Nome Parametro	Descrizione del parametro
(*) = obbligatorio	id*	ID univoco del veicolo selezionato sulla mappa
Output	Descrizione dell'output Dettagli completi: tipo, batteria, tariffa, stato disponibilità	
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5	

1.7.2 Funzionalità Utente - Gestione Noleggi

Nome	CreateRide	
Requisiti di riferimento	Avvio sessione di noleggio veicolo	
Descrizione	Inizia un nuovo noleggio bloccando il veicolo per l'utente	
Input	Nome Parametro	Descrizione del parametro
(*) = obbligatorio	UserId*	ID dell'utente autenticato
	VehicleId*	ID del veicolo da sbloccare
	StartTime	Timestamp automatico di inizio noleggio
	PositionStartId	Posizione GPS di partenza
Output	Descrizione dell'output Conferma noleggio attivo con codice di sblocco veicolo	
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5	

Nome	GetAllUserRides				
Requisiti di riferimento	Storico viaggi e fatturazione utente				
Descrizione	Visualizza cronologia completa noleggi con costi sostenuti				
Input (*) = obbligatorio	<table border="1"> <thead> <tr> <th>Nome Parametro</th> <th>Descrizione del parametro</th> </tr> </thead> <tbody> <tr> <td>userId*</td> <td>ID dell'utente autenticato</td> </tr> </tbody> </table>	Nome Parametro	Descrizione del parametro	userId*	ID dell'utente autenticato
Nome Parametro	Descrizione del parametro				
userId*	ID dell'utente autenticato				
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th> </tr> </thead> <tbody> <tr> <td>Lista noleggi con date, durate, percorsi e importi pagati</td> </tr> </tbody> </table>	Descrizione dell'output	Lista noleggi con date, durate, percorsi e importi pagati		
Descrizione dell'output					
Lista noleggi con date, durate, percorsi e importi pagati					
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5				

1.7.3 Funzionalità Utente - Gestione Portafoglio

Nome	GetBalanceByUserId				
Requisiti di riferimento	Visualizzazione saldo e punti fedeltà				
Descrizione	Mostra credito disponibile e punti accumulati dall'utente				
Input (*) = obbligatorio	<table border="1"> <thead> <tr> <th>Nome Parametro</th> <th>Descrizione del parametro</th> </tr> </thead> <tbody> <tr> <td>userId*</td> <td>ID dell'utente autenticato</td> </tr> </tbody> </table>	Nome Parametro	Descrizione del parametro	userId*	ID dell'utente autenticato
Nome Parametro	Descrizione del parametro				
userId*	ID dell'utente autenticato				
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th> </tr> </thead> <tbody> <tr> <td>Saldo corrente in euro e punti fedeltà disponibili</td> </tr> </tbody> </table>	Descrizione dell'output	Saldo corrente in euro e punti fedeltà disponibili		
Descrizione dell'output					
Saldo corrente in euro e punti fedeltà disponibili					
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5				

1.7.4 Funzionalità Utente - Localizzazione

Nome	GetAllParkingSlots				
Requisiti di riferimento	Mappa punti di riconsegna veicoli				
Descrizione	Visualizza tutti i parcheggi disponibili per terminare il noleggio				
Input (*) = obbligatorio	<table border="1"> <thead> <tr> <th>Nome Parametro</th> <th>Descrizione del parametro</th> </tr> </thead> <tbody> <tr> <td>Nessun parametro</td> <td>Informazione pubblica per tutti gli utenti</td> </tr> </tbody> </table>	Nome Parametro	Descrizione del parametro	Nessun parametro	Informazione pubblica per tutti gli utenti
Nome Parametro	Descrizione del parametro				
Nessun parametro	Informazione pubblica per tutti gli utenti				
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th> </tr> </thead> <tbody> <tr> <td>Lista parcheggi con coordinate GPS e posti disponibili</td> </tr> </tbody> </table>	Descrizione dell'output	Lista parcheggi con coordinate GPS e posti disponibili		
Descrizione dell'output					
Lista parcheggi con coordinate GPS e posti disponibili					
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5				

1.8 Funzionalità Staff - Gestione Flotta

Nome	CreateVehicle									
Requisiti di riferimento	Aggiunta veicolo alla flotta aziendale									
Descrizione	Registra nuovo veicolo nel sistema con assegnazione parcheggio									
Input	<table border="1"> <thead> <tr> <th>Nome Parametro</th> <th>Descrizione del parametro</th> </tr> </thead> <tbody> <tr> <td>Plate*</td> <td>Targa veicolo di esattamente 5 caratteri</td> </tr> <tr> <td>VehicleTypeId*</td> <td>ID del tipo di veicolo selezionato</td> </tr> <tr> <td>ParkingSlotId*</td> <td>ID del parcheggio di assegnazione iniziale</td> </tr> </tbody> </table>		Nome Parametro	Descrizione del parametro	Plate*	Targa veicolo di esattamente 5 caratteri	VehicleTypeId*	ID del tipo di veicolo selezionato	ParkingSlotId*	ID del parcheggio di assegnazione iniziale
Nome Parametro	Descrizione del parametro									
Plate*	Targa veicolo di esattamente 5 caratteri									
VehicleTypeId*	ID del tipo di veicolo selezionato									
ParkingSlotId*	ID del parcheggio di assegnazione iniziale									
(*) = obbligatorio										
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th> </tr> </thead> <tbody> <tr> <td>Conferma registrazione veicolo con stato disponibile</td> </tr> </tbody> </table>		Descrizione dell'output	Conferma registrazione veicolo con stato disponibile						
Descrizione dell'output										
Conferma registrazione veicolo con stato disponibile										
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5									

Nome	CreateVehicleType											
Requisiti di riferimento	Definizione nuove categorie di veicoli											
Descrizione	Crea nuovo tipo di veicolo con caratteristiche e tariffazione											
Input	<table border="1"> <thead> <tr> <th>Nome Parametro</th> <th>Descrizione del parametro</th> </tr> </thead> <tbody> <tr> <td>VehicleTypeName*</td> <td>Nome identificativo del tipo veicolo</td> </tr> <tr> <td>VehicleDescription*</td> <td>Descrizione caratteristiche e capacità</td> </tr> <tr> <td>Price*</td> <td>Tariffa di noleggio per unità di tempo</td> </tr> <tr> <td>IsAvailable*</td> <td>Stato di disponibilità per il noleggio</td> </tr> </tbody> </table>		Nome Parametro	Descrizione del parametro	VehicleTypeName*	Nome identificativo del tipo veicolo	VehicleDescription*	Descrizione caratteristiche e capacità	Price*	Tariffa di noleggio per unità di tempo	IsAvailable*	Stato di disponibilità per il noleggio
Nome Parametro	Descrizione del parametro											
VehicleTypeName*	Nome identificativo del tipo veicolo											
VehicleDescription*	Descrizione caratteristiche e capacità											
Price*	Tariffa di noleggio per unità di tempo											
IsAvailable*	Stato di disponibilità per il noleggio											
(*) = obbligatorio												
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th> </tr> </thead> <tbody> <tr> <td>Conferma creazione tipo veicolo con ID generato</td> </tr> </tbody> </table>		Descrizione dell'output	Conferma creazione tipo veicolo con ID generato								
Descrizione dell'output												
Conferma creazione tipo veicolo con ID generato												
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5											

Nome	UpdateVehicleType												
Requisiti di riferimento	Modifica caratteristiche tipo veicolo esistente												
Descrizione	Aggiorna informazioni e tariffazione di un tipo veicolo												
Input (*) = obbligatorio	<table border="1"> <thead> <tr> <th>Nome Parametro</th><th>Descrizione del parametro</th></tr> </thead> <tbody> <tr> <td>Id*</td><td>Identificativo univoco del tipo veicolo</td></tr> <tr> <td>VehicleTypeName*</td><td>Nome aggiornato del tipo veicolo</td></tr> <tr> <td>VehicleDescription*</td><td>Descrizione aggiornata</td></tr> <tr> <td>Price*</td><td>Nuova tariffa di noleggio</td></tr> <tr> <td>IsAvailable*</td><td>Stato di disponibilità aggiornato</td></tr> </tbody> </table>	Nome Parametro	Descrizione del parametro	Id*	Identificativo univoco del tipo veicolo	VehicleTypeName*	Nome aggiornato del tipo veicolo	VehicleDescription*	Descrizione aggiornata	Price*	Nuova tariffa di noleggio	IsAvailable*	Stato di disponibilità aggiornato
Nome Parametro	Descrizione del parametro												
Id*	Identificativo univoco del tipo veicolo												
VehicleTypeName*	Nome aggiornato del tipo veicolo												
VehicleDescription*	Descrizione aggiornata												
Price*	Nuova tariffa di noleggio												
IsAvailable*	Stato di disponibilità aggiornato												
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th></tr> </thead> <tbody> <tr> <td>Conferma aggiornamento tipo veicolo</td></tr> </tbody> </table>	Descrizione dell'output	Conferma aggiornamento tipo veicolo										
Descrizione dell'output													
Conferma aggiornamento tipo veicolo													
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5												
Nome	DeleteVehicleType												
Requisiti di riferimento	Rimozione tipo veicolo dal sistema												
Descrizione	Elimina definitivamente un tipo veicolo dalla configurazione												
Input (*) = obbligatorio	<table border="1"> <thead> <tr> <th>Nome Parametro</th><th>Descrizione del parametro</th></tr> </thead> <tbody> <tr> <td>Id*</td><td>Identificativo univoco del tipo veicolo da eliminare</td></tr> </tbody> </table>	Nome Parametro	Descrizione del parametro	Id*	Identificativo univoco del tipo veicolo da eliminare								
Nome Parametro	Descrizione del parametro												
Id*	Identificativo univoco del tipo veicolo da eliminare												
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th></tr> </thead> <tbody> <tr> <td>Conferma eliminazione tipo veicolo</td></tr> </tbody> </table>	Descrizione dell'output	Conferma eliminazione tipo veicolo										
Descrizione dell'output													
Conferma eliminazione tipo veicolo													
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5												

1.8.1 Funzionalità Staff - Gestione Parcheggi

Nome	CreateParkingSlot						
Requisiti di riferimento	Espansione rete punti di parcheggio						
Descrizione	Aggiunge nuovo slot di parcheggio con validazione geografica						
Input	<table border="1"> <thead> <tr> <th>Nome Parametro</th> <th>Descrizione del parametro</th> </tr> </thead> <tbody> <tr> <td>ParkingArea*</td> <td>Poligono GeoJSON che definisce l'area di parcheggio</td> </tr> <tr> <td>Type*</td> <td>Tipologia slot (Standard, Premium, Riservato)</td> </tr> </tbody> </table> <p>(*) = obbligatorio</p>	Nome Parametro	Descrizione del parametro	ParkingArea*	Poligono GeoJSON che definisce l'area di parcheggio	Type*	Tipologia slot (Standard, Premium, Riservato)
Nome Parametro	Descrizione del parametro						
ParkingArea*	Poligono GeoJSON che definisce l'area di parcheggio						
Type*	Tipologia slot (Standard, Premium, Riservato)						

Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th></tr> </thead> <tbody> <tr> <td>Conferma creazione slot con validazione area città</td></tr> </tbody> </table>	Descrizione dell'output	Conferma creazione slot con validazione area città
Descrizione dell'output			
Conferma creazione slot con validazione area città			
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5		

Nome	UpdateParkingSlot								
Requisiti di riferimento	Modifica configurazione slot esistente								
Descrizione	Aggiorna area geografica e tipologia slot di parcheggio								
Input	<table border="1"> <thead> <tr> <th>Nome Parametro</th> <th>Descrizione del parametro</th> </tr> </thead> <tbody> <tr> <td>Id*</td> <td>Identificativo univoco dello slot di parcheggio</td> </tr> <tr> <td>ParkingArea*</td> <td>Poligono GeoJSON aggiornato</td> </tr> <tr> <td>Type*</td> <td>Tipologia slot aggiornata</td> </tr> </tbody> </table> <p>(*) = obbligatorio</p>	Nome Parametro	Descrizione del parametro	Id*	Identificativo univoco dello slot di parcheggio	ParkingArea*	Poligono GeoJSON aggiornato	Type*	Tipologia slot aggiornata
Nome Parametro	Descrizione del parametro								
Id*	Identificativo univoco dello slot di parcheggio								
ParkingArea*	Poligono GeoJSON aggiornato								
Type*	Tipologia slot aggiornata								
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th> </tr> </thead> <tbody> <tr> <td>Conferma aggiornamento slot con validazione</td> </tr> </tbody> </table>	Descrizione dell'output	Conferma aggiornamento slot con validazione						
Descrizione dell'output									
Conferma aggiornamento slot con validazione									
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5								

Nome	DeleteParkingSlot				
Requisiti di riferimento	Rimozione slot parcheggio dal sistema				
Descrizione	Elimina definitivamente uno slot di parcheggio				
Input	<table border="1"> <thead> <tr> <th>Nome Parametro</th> <th>Descrizione del parametro</th> </tr> </thead> <tbody> <tr> <td>Id*</td> <td>Identificativo univoco dello slot da eliminare</td> </tr> </tbody> </table> <p>(*) = obbligatorio</p>	Nome Parametro	Descrizione del parametro	Id*	Identificativo univoco dello slot da eliminare
Nome Parametro	Descrizione del parametro				
Id*	Identificativo univoco dello slot da eliminare				
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th> </tr> </thead> <tbody> <tr> <td>Conferma eliminazione slot di parcheggio</td> </tr> </tbody> </table>	Descrizione dell'output	Conferma eliminazione slot di parcheggio		
Descrizione dell'output					
Conferma eliminazione slot di parcheggio					
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5				

1.9 Funzionalità Amministratore - Gestione Utenti

Nome	FilterUsers							
Requisiti di riferimento	Ricerca e filtro utenti del sistema							
Descrizione	Filtrà utenti per email e ruolo per gestione amministrativa							
Input	<table border="1"> <thead> <tr> <th>Nome Parametro</th> <th>Descrizione del parametro</th> </tr> </thead> <tbody> <tr> <td>Email</td> <td>Filtro parziale per indirizzo email</td> </tr> <tr> <td>Role</td> <td>Filtro per ruolo (User, Technician, Staff, Admin)</td> </tr> </tbody> </table>		Nome Parametro	Descrizione del parametro	Email	Filtro parziale per indirizzo email	Role	Filtro per ruolo (User, Technician, Staff, Admin)
Nome Parametro	Descrizione del parametro							
Email	Filtro parziale per indirizzo email							
Role	Filtro per ruolo (User, Technician, Staff, Admin)							
(*) = obbligatorio								
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th> </tr> </thead> <tbody> <tr> <td>Lista utenti filtrati con email, ruolo e stato</td> </tr> </tbody> </table>		Descrizione dell'output	Lista utenti filtrati con email, ruolo e stato				
Descrizione dell'output								
Lista utenti filtrati con email, ruolo e stato								
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5							
Nome	SuspendUser							
Requisiti di riferimento	Sospensione temporanea accesso utente							
Descrizione	Blocca accesso utente al sistema fino a revoca amministratore							
Input	<table border="1"> <thead> <tr> <th>Nome Parametro</th> <th>Descrizione del parametro</th> </tr> </thead> <tbody> <tr> <td>Id*</td> <td>Identificativo univoco dell'utente da sospendere</td> </tr> </tbody> </table>		Nome Parametro	Descrizione del parametro	Id*	Identificativo univoco dell'utente da sospendere		
Nome Parametro	Descrizione del parametro							
Id*	Identificativo univoco dell'utente da sospendere							
(*) = obbligatorio								
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th> </tr> </thead> <tbody> <tr> <td>Conferma sospensione con blocco accesso</td> </tr> </tbody> </table>		Descrizione dell'output	Conferma sospensione con blocco accesso				
Descrizione dell'output								
Conferma sospensione con blocco accesso								
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5							
Nome	EditRole							
Requisiti di riferimento	Modifica privilegi e autorizzazioni utente							
Descrizione	Cambia ruolo utente con aggiornamento permessi sistema							
Input	<table border="1"> <thead> <tr> <th>Nome Parametro</th> <th>Descrizione del parametro</th> </tr> </thead> <tbody> <tr> <td>Id*</td> <td>Identificativo univoco dell'utente</td> </tr> <tr> <td>NewRole*</td> <td>Nuovo ruolo (User, Technician, Staff, Admin)</td> </tr> </tbody> </table>		Nome Parametro	Descrizione del parametro	Id*	Identificativo univoco dell'utente	NewRole*	Nuovo ruolo (User, Technician, Staff, Admin)
Nome Parametro	Descrizione del parametro							
Id*	Identificativo univoco dell'utente							
NewRole*	Nuovo ruolo (User, Technician, Staff, Admin)							
(*) = obbligatorio								
Output	<table border="1"> <thead> <tr> <th>Descrizione dell'output</th> </tr> </thead> <tbody> <tr> <td>Conferma aggiornamento ruolo e permessi</td> </tr> </tbody> </table>		Descrizione dell'output	Conferma aggiornamento ruolo e permessi				
Descrizione dell'output								
Conferma aggiornamento ruolo e permessi								
Diagrammi di sequenza	vedere diagramma di sequenza in seguito Figura 5							

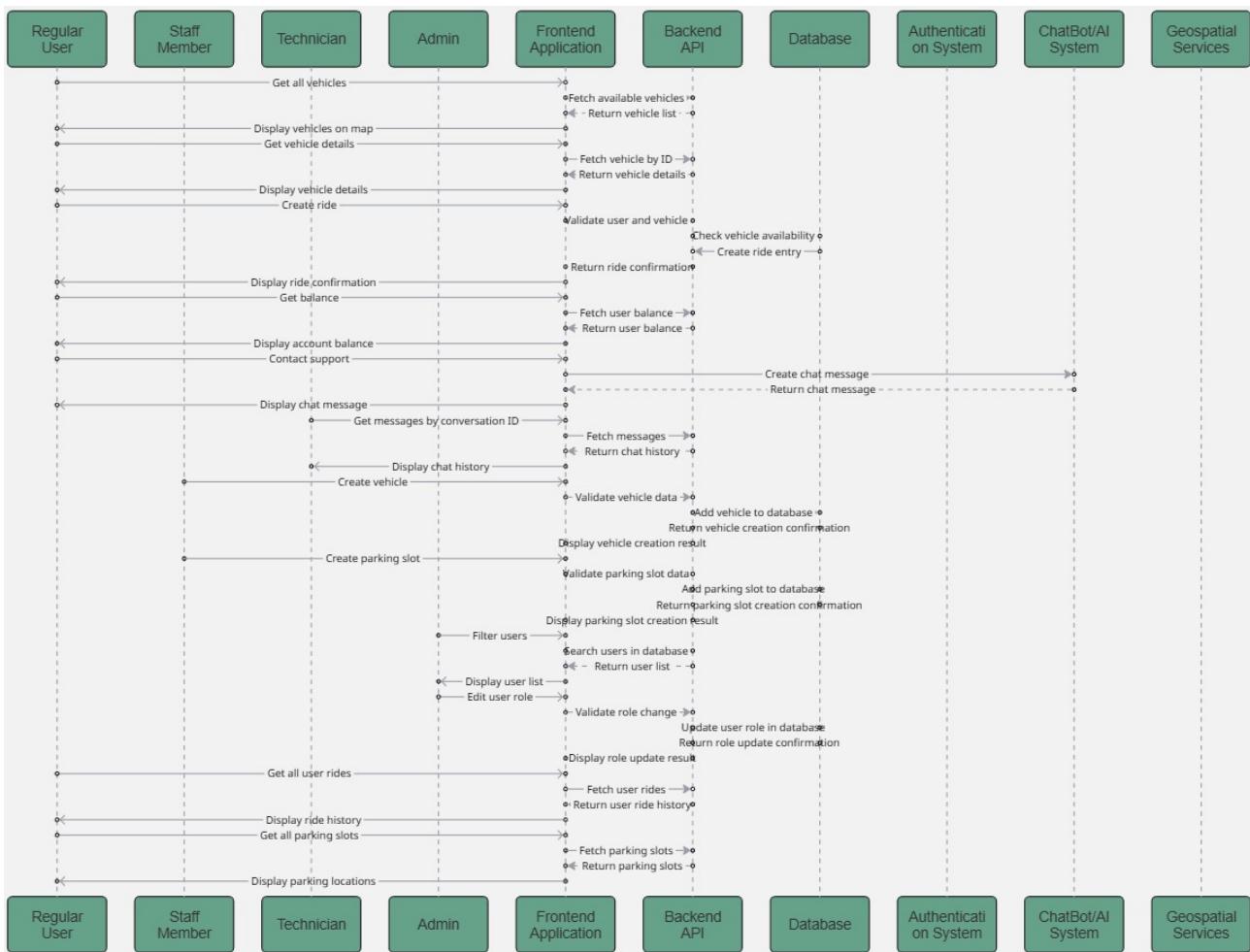


Figura 5: Diagramma di sequenza

2 Progettazione di dettaglio del sistema

Il sistema di Mobishare si basa su un'architettura di rete ibrida che consente la comunicazione tra veicoli smart (biciclette e scooter) con un server centrale. La rete è progettata per garantire affidabilità, scalabilità e accessibilità in scenari urbani ed extraurbani. L'infrastruttura è composta da una rete interna centralizzata, che ospita l'elaborazione dei dati e i servizi centrali dell'applicazione, e da una rete esterna distribuita, utilizzata dai dispositivi mobili (i veicoli) per inviare e ricevere informazioni in real time.

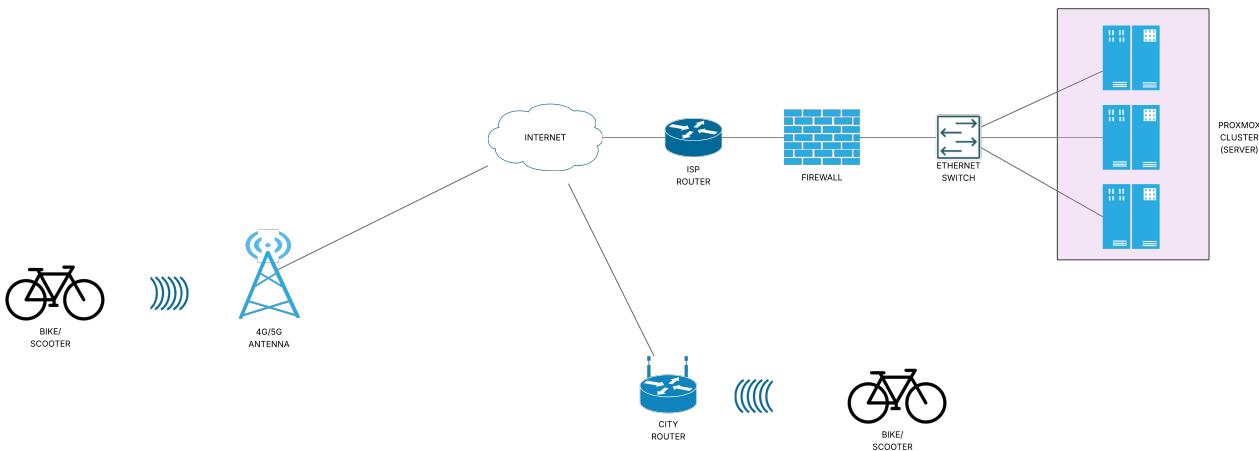


Figura 6: Architettura generale del sistema Mobishare

2.1 Infrastruttura interna

L'infrastruttura interna è basata su un cluster realizzato con Proxmox che funge da server centrale per l'intero sistema. Le macchine componenti il cluster sono collegate a uno switch di rete che, a sua volta, si interfaccia con un router fornito dall'ISP, il quale gestisce l'accesso alla rete globale.

- il cluster è configurato come un'entità unica e accede alla rete esterna attraverso un unico indirizzo IP statico fornito dall'ISP;
- tutte le richieste dai veicoli e dagli utenti passano attraverso questo punto centrale, il quale gestisce:
 - routing;
 - autenticazione;
 - elaborazione dei dati;
 - comunicazioni tra servizi.

Tra il router e lo switch è interposto un firewall, configurato per filtrare e controllare il traffico in ingresso e in uscita; questo ha il compito di proteggere l'infrastruttura interna da accessi non autorizzati, attacchi esterni e altre possibili minacce.

- il firewall applica regole per permettere solo il traffico necessario verso i container (es. porte HTTP/HTTPS, MQTT) e bloccare tutte le connessioni non esplicitamente autorizzate;
- Viene così implementato un modello di sicurezza a perimetro, che isola il cluster e protegge i servizi da eventuali attacchi di rete. Questa architettura garantisce un controllo centralizzato e una gestione semplificata delle operazioni di rete e sicurezza.

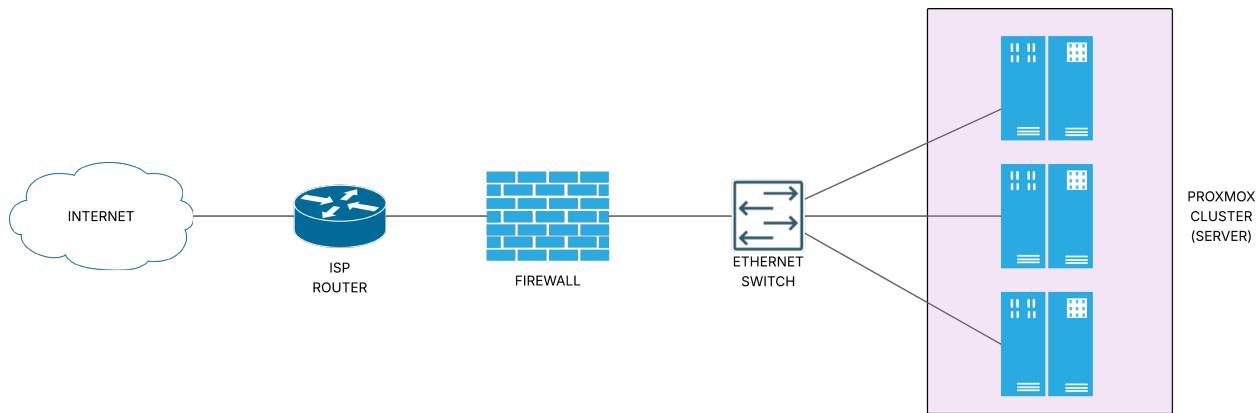


Figura 7: Struttura delle macchine virtuali nel cluster Proxmox

2.2 Infrastruttura esterna

L'infrastruttura esterna è progettata per permettere ai veicoli smart (biciclette e scooter) di comunicare con il server centrale in due modalità alternative:

- rete WI-FI con Access Point distribuiti;
- rete cellulare 4G/5G (proposta di miglioramento).

2.2.1 Rete WI-FI con Access Point distribuiti

In questa configurazione, i veicoli si connettono alla rete tramite una serie di Access Point (Wi-Fi) installati strategicamente sul territorio (stazioni di ricarica, centri urbani, parcheggi, ecc...). I dispositivi del veicolo si connettono automaticamente all'AP più vicino per inviare dati al server centrale (es. posizione GPS, stato del veicolo).

- Vantaggi: Basso costo di connessione, alta velocità di trasmissione, controllo dell'infrastruttura.
- Svantaggi: Copertura limitata, richiede un'infrastruttura fisica ben distribuita.

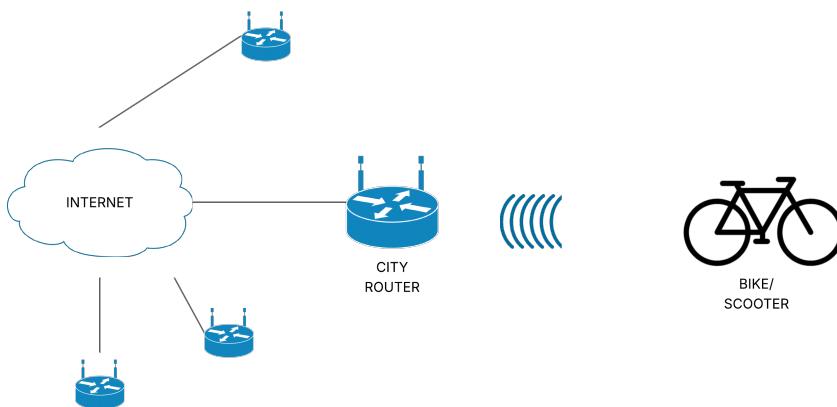


Figura 8: Configurazione della connessione Wi-Fi con Access Point distribuiti

2.2.2 Rete cellulare 4G/5G(Proposta di Miglioramento)

Come evoluzione dell'infrastruttura, si propone l'adozione di moduli cellulari 4G/5G da integrare nei veicoli, sfruttando le reti mobili già esistenti. Con questo miglioramento ogni veicolo è autonomo nella connessione e può comunicare con il server in qualsiasi momento, ovunque si trovi (a patto che sia disponibile la rete mobile).

- Pro: Ampia copertura, indipendenza dagli AP, maggiore mobilità.
- Contro: Costo maggiore per i moduli e la connettività, dipendenza da operatori telefonici.

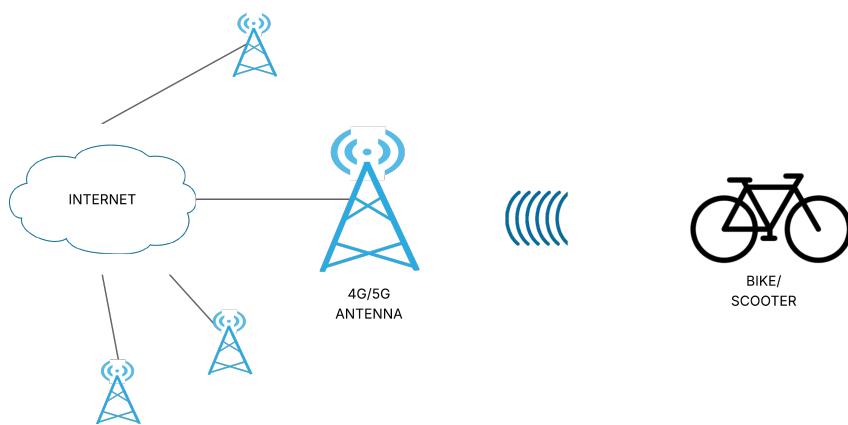


Figura 9: Configurazione della connessione cellulare 4G/5G

2.3 Circuito comunicazione GPS

Per la localizzazione dei veicoli, Mobishare utilizza un circuito elettronico composto da:

- ARDUINO UNO REV2 WI-FI;
- modulo GPS-NEO 6M.

2.3.1 ARDUINO UNO REV2 WI-FI

Una variante della board Arduino UNO, con integrato un chip ESP32 che consente la connessione Wi-Fi nativa. Questo elimina l'utilizzo di moduli esterni per la connettività wireless.

Motivazione della scelta:

- ESP32 integrato = Wi-Fi più stabile e programmabile;
- economica e facile da integrare nei veicoli;
- ampio supporto nella IoT community.

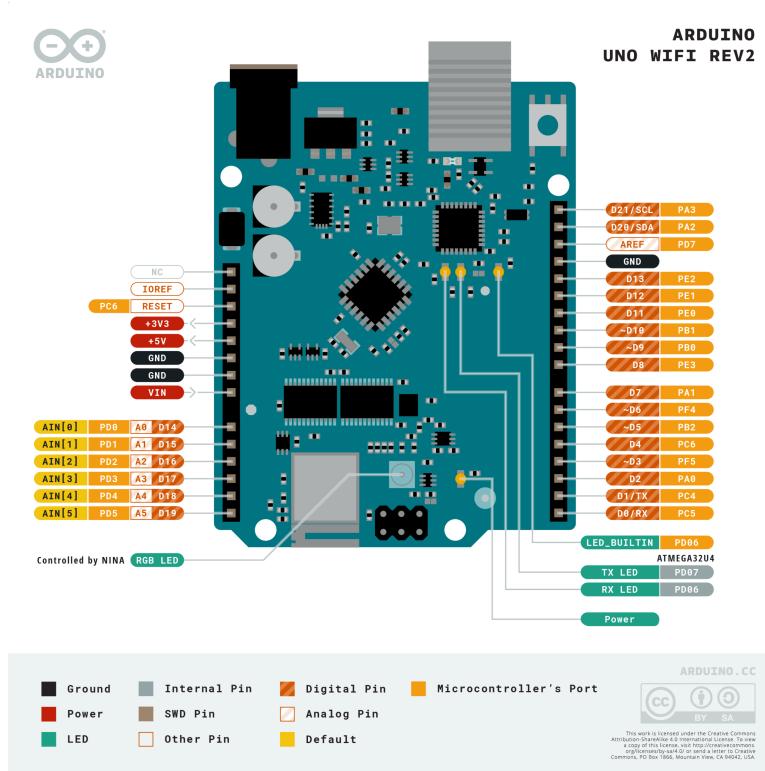


Figura 10: Schema generale della scheda Arduino UNO REV2 Wi-Fi

2.3.2 Modulo GPS-NEO 6M

Modulo GPS economico e preciso, compatibile con Arduino. Fornisce in tempo reale:

- coordinate geografiche;
- velocità e altitudine;
- timestamp sincronizzati con i satelliti.

Il modulo comunica con i seguenti sistemi satellitari di posizionamento:

- GPS (Stati Uniti d'America);
- QZSS (Giappone).

2.3.3 Schema elettrico

Per abilitare la localizzazione dei veicoli, il sistema di Mobishare utilizza un circuito elettronico semplice basato su una scheda Arduino con modulo Wi-Fi ESP32 e un ricevitore GPS. Di seguito le illustrazioni dei due possibili schemi di collegamento del modulo GPS.

3.3V	VCC
GND	GND
8	TX
~9(PWM)	RX

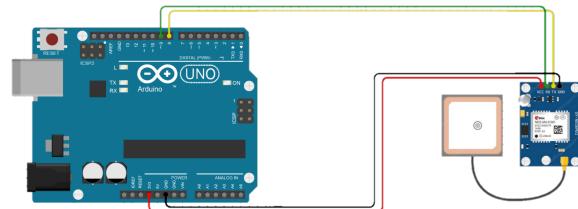


Figura 11: Tabella dei collegamenti e schema corrispondente

Nota: Il modulo GPS può anche essere alimentato a 3.3V, ma l'alimentazione a 5V garantisce maggiore stabilità in ambienti urbani con interferenze.

5V	VCC
GND	GND
8	TX
~9(PWM)	RX

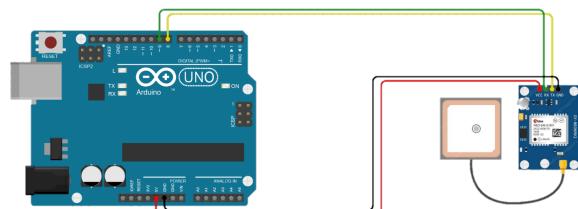


Figura 12: Tabella dei collegamenti e schema corrispondente

Il microcontrollore si occupa di raccogliere i dati GPS e inviarli periodicamente al server centrale tramite Wi-Fi, usando un protocollo leggero (es. MQTT).

2.4 AI Agent

2.4.1 Introduzione e Concetto

Una volta autenticato, l'utente viene reindirizzato alla pagina principale dell'applicazione, dove può interagire con un assistente virtuale (AI Agent) tramite l'apposita icona della chat.

Per garantire la massima pertinenza e accuratezza, ogni sessione di chat è concepita per essere indipendente. Questo significa che la conversazione viene periodicamente reimpostata, permettendo all'agente di fornire risposte più precise e specifiche, basate unicamente sul contesto della discussione corrente. Questo approccio, oltre a migliorare le performance, garantisce un'esperienza utente più chiara e mirata.

2.4.2 Struttura del Codice Sorgente

La logica dell'intelligenza artificiale è interamente incapsulata nel progetto `Mobishare.Ai`, che agisce come il "cervello" dell'assistente virtuale. Gli altri progetti della soluzione collaborano per integrare questo servizio centrale:

- `Mobishare.App` (Livello UI): Fornisce l'interfaccia utente della chat.
- `Mobishare.Infrastructure` (Livello Comunicazione): Gestisce la connessione in tempo reale tramite SignalR (ChatHub).
- `Mobishare.Core` (Livello Logico/API): Orchestra il flusso e invoca i servizi in `Mobishare.Ai`.

2.4.3 Architettura e Flusso Decisionale

L'interazione tra l'utente e l'agente segue un processo strutturato che unisce il flusso tecnico con la logica decisionale del modello.

1. **Richiesta Utente:** Il messaggio dell'utente viene inviato dal frontend al backend tramite una connessione SignalR.
2. **Costruzione del Prompt:** Il backend costruisce un prompt dettagliato che include contesto di sistema, cronologia, il messaggio dell'utente e una lista di *tools* disponibili. I *tools* sono funzioni C# sicure che l'agente può eseguire (es. query specializzate, prevenzione di operazioni critiche, etc.).
3. **Elaborazione e Meccanismo Decisionale LLM:** Il prompt viene inviato al modello su Ollama. A differenza di un LLM tradizionale, l'agente analizza l'intento, cerca corrispondenze semantiche con i *tools* e determina l'azione migliore:
 - **Eseguire un tool:** Se la richiesta corrisponde a una funzione, il modello restituisce una richiesta di 'function calling'. Il backend esegue il tool, e il risultato viene inviato di nuovo al modello per l'elaborazione finale.
 - **Rispondere direttamente:** Se la domanda è pertinente ma non richiede un tool, il modello fornisce una risposta basata sulle informazioni di contesto.
 - **Richiedere chiarimenti:** Per richieste ambigue, il modello chiede maggiori dettagli all'utente.
 - **Gestire richieste non pertinenti:** Se la domanda è fuori tema, il modello comunica all'utente l'impossibilità di rispondere.
4. **Funzionalità di Ragionamento Avanzato:** Per compiti complessi come l'assegnazione di un ticket di manutenzione, l'agente può attivare un sotto-processo di ragionamento (pattern "LLM-as-a-Tool"), dove un prompt specializzato viene usato per analizzare dati complessi (es. carico di lavoro e competenze dei tecnici) e prendere una decisione ottimale.
5. **Risposta Finale:** Il modello genera la risposta definitiva, che viene trasmessa all'utente tramite SignalR.

Questo approccio garantisce:

- Precisione:** Solo operazioni autorizzate e pertinenti vengono eseguite.
- Sicurezza:** Nessuna azione rischiosa o al di fuori del contesto consentito.
- Usabilità:** Feedback chiaro all'utente in ogni scenario.

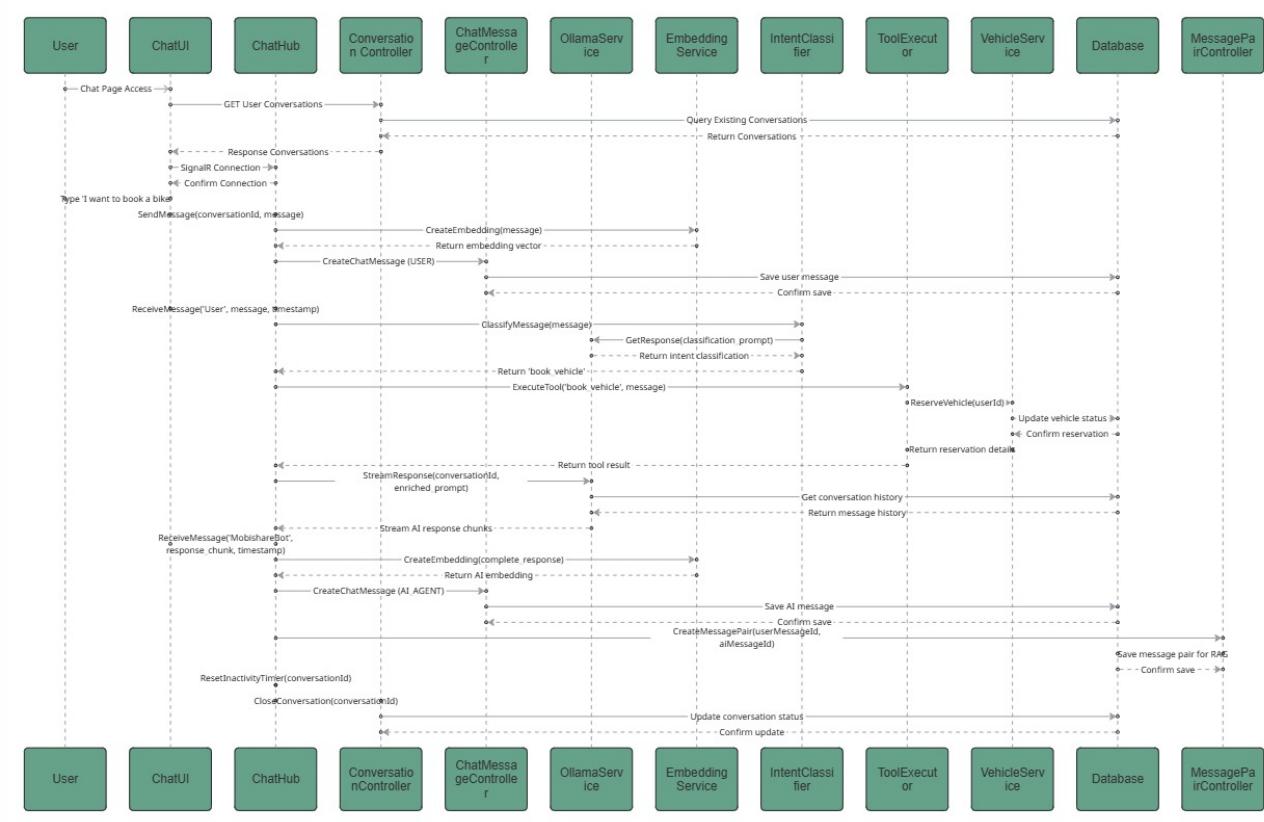


Figura 13: Diagramma di sequenza dell'interazione con l'AI Agent.

2.4.4 Configurazione e Avvio

Questa guida descrive i passaggi necessari per configurare ed eseguire il progetto MobiShare in un ambiente di sviluppo locale.

2.5 Prerequisiti

Prima di iniziare, assicurarsi di avere installato tutto il software necessario:

- .NET SDK:** Versione 8.0.
- IDE/Text-editor:** Visual Studio o Visual Studio Code.
- Git:** Per clonare il repository.
- Ollama:** L'applicazione Ollama deve essere installata e in esecuzione in background.
- Database:** Un'istanza di SQL Server.

2.6 Flusso di Installazione e Configurazione

2.6.1 Clonare il Repository

Aprire un terminale e clonare il repository sulla macchina locale, poi entrare nella cartella del progetto.

```
git clone https://github.com/your-username/mobishare.git  
cd tuo-repo
```

2.6.2 Configurare i Segreti dell'Applicazione

Il progetto richiede chiavi API e una stringa di connessione al database. Nella cartella del progetto Mobishare.App, trovare il file appsettings.json che seguirà la seguente struttura:

```
{  
    "ConnectionStrings": {  
        "DefaultConnection": "Data Source=../Mobishare.db"  
    },  
    "Logging": {  
        "LogLevel": {  
            "Default": "Information",  
            "Microsoft.AspNetCore": "Warning"  
        }  
    },  
    "AllowedHosts": "*",  
    "Ollama": {  
        "Llms": {  
            "DefaultUrlApiClient": "http://localhost:11434",  
            "Qwen3": {  
                "UrlApiClient": "http://localhost:11434",  
                "ModelName": "qwen3:latest"  
            },  
            "DeepSeek": {  
                "UrlApiClient": "http://localhost:11434",  
                "ModelName": "deepseek-r1"  
            }  
        },  
        "Embedding": {  
            "UrlApiClient": "http://localhost:11434",  
            "ModelName": "nomic-embed-text"  
        }  
    }  
}
```

2.6.3 Installare i Modelli AI

Assicurarsi che Ollama sia in esecuzione, poi aprire un terminale e scaricare i modelli linguistici richiesti.

```
ollama pull qwen3:latest  
ollama pull nomic-embed-text
```

2.6.4 Creare e Aggiornare il Database

Questo comando creerà il database e applicherà lo schema. Eseguire il comando dalla cartella del progetto che contiene il DbContext.

```
dotnet ef database update
```

2.7 Avviare l'Applicazione

Una volta completati i passaggi precedenti, si è pronti per avviare il progetto.

1. **Tramite Visual Studio:** Aprire la soluzione (.sln) e premere F5.
2. **Tramite la riga di comando:** Aprire un terminale nella cartella principale della soluzione e lanciare:

```
dotnet run --project Mobishare.App/Mobishare.App.csproj
```

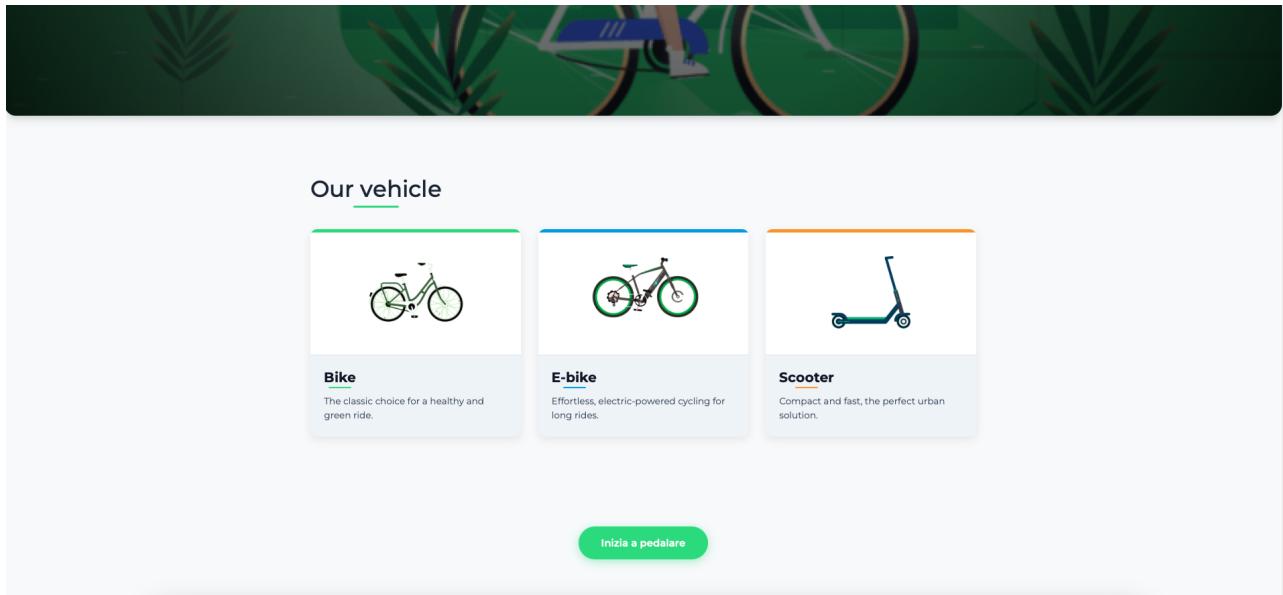
L'applicazione sarà disponibile all'URL mostrato nel terminale (di solito `https://localhost:XXXX`).

3 Il sistema realizzato

3.1 Interfaccia Utente (Homepage)

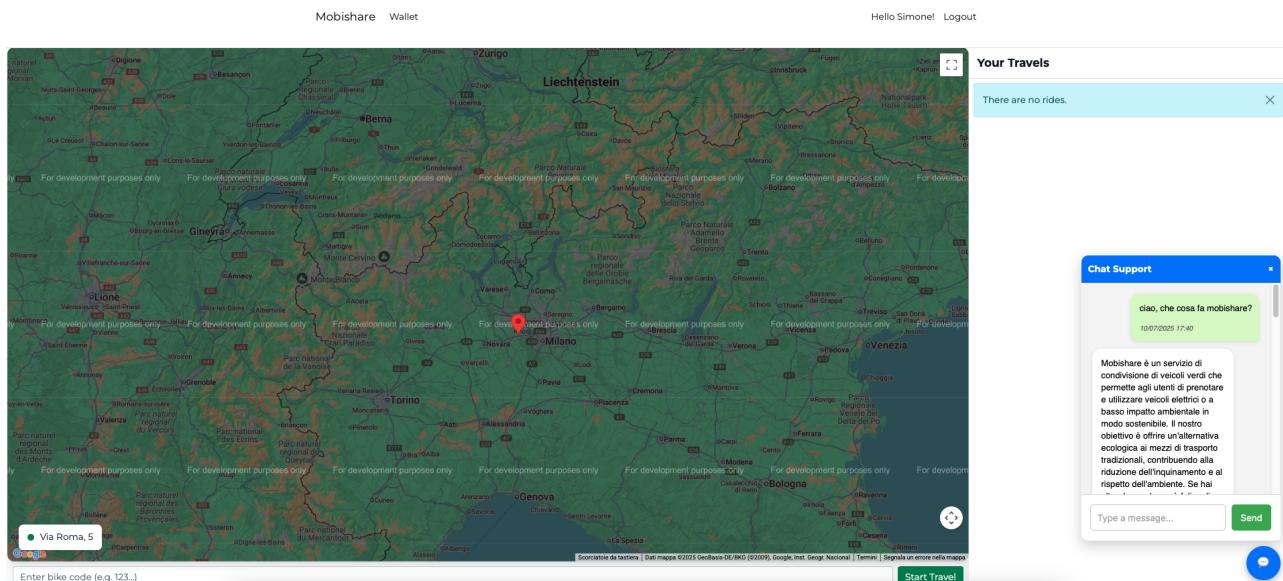
La HomePage presenta:

- dashboard intuitiva con servizi di mobilità in primo piano;



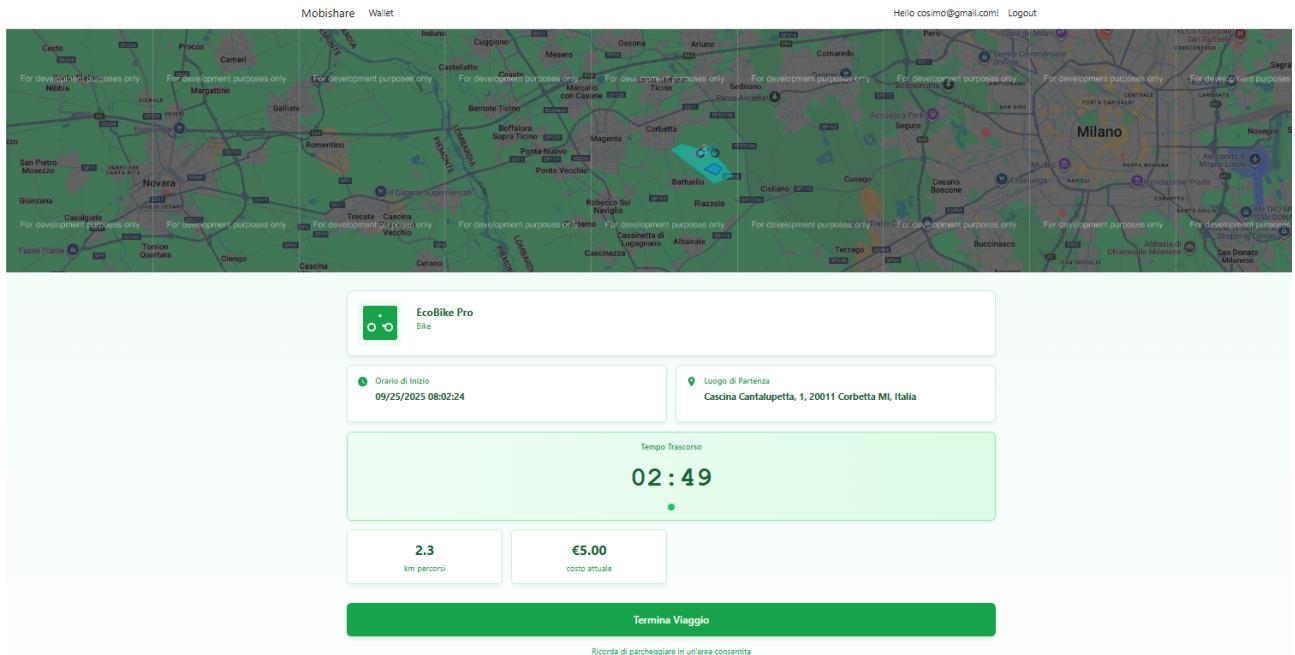
The screenshot shows a dark-themed homepage with a green banner at the top featuring a bicycle. Below it, a section titled "Our vehicle" displays three options: "Bike" (classic choice for a healthy and green ride), "E-bike" (effortless, electric-powered cycling for long rides), and "Scooter" (compact and fast, the perfect urban solution). A green button labeled "Inizia a pedalare" is located at the bottom of this section.

- mappa interattiva per visualizzazione veicoli/disponibilità in tempo reale;
- prenotazione veicoli (monopattini/bici elettriche);
- cronologia viaggi;
- parlare con l'agente ia;

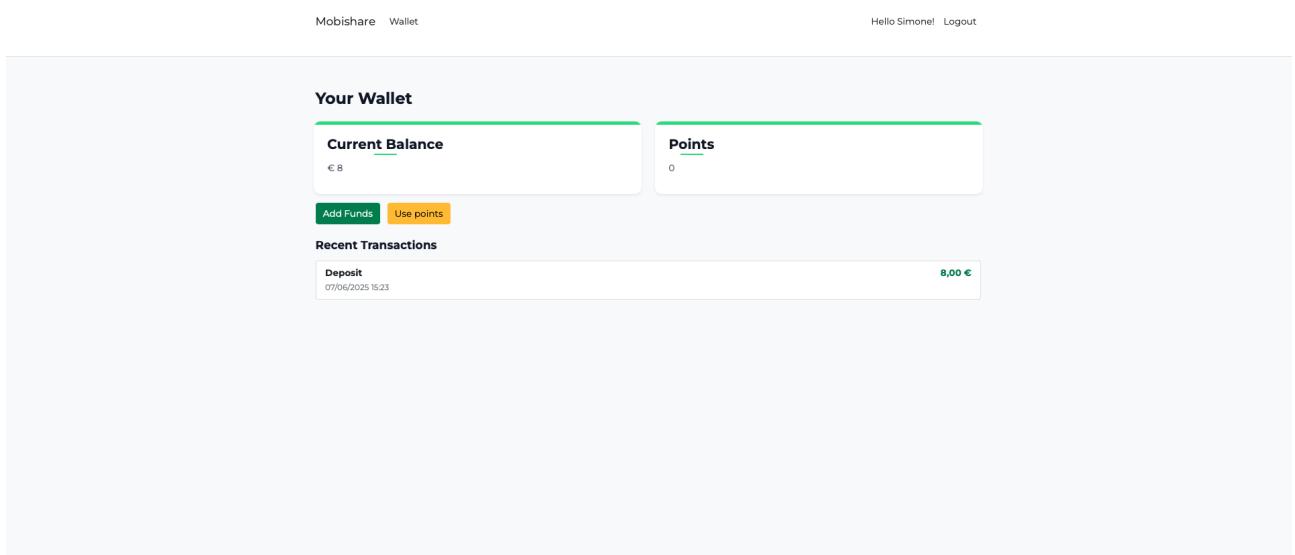


The screenshot shows a mobile application interface. At the top, there are navigation links for "Mobishare" and "Wallet", and a user greeting "Hello Simone! Logout". On the right, there is a "Your Travels" section which currently says "There are no rides." Below the map, there is a "Chat Support" window with a message from the bot: "ciao, che cosa fa mobishare?" and a timestamp "10/07/2025 17:40". The main area features a detailed map of Italy and surrounding regions, with a red marker indicating a location. A search bar at the bottom left says "Enter bike code (e.g. ROMA_5)" and a "Start Travel" button at the bottom right.

- visualizzazione corsa;



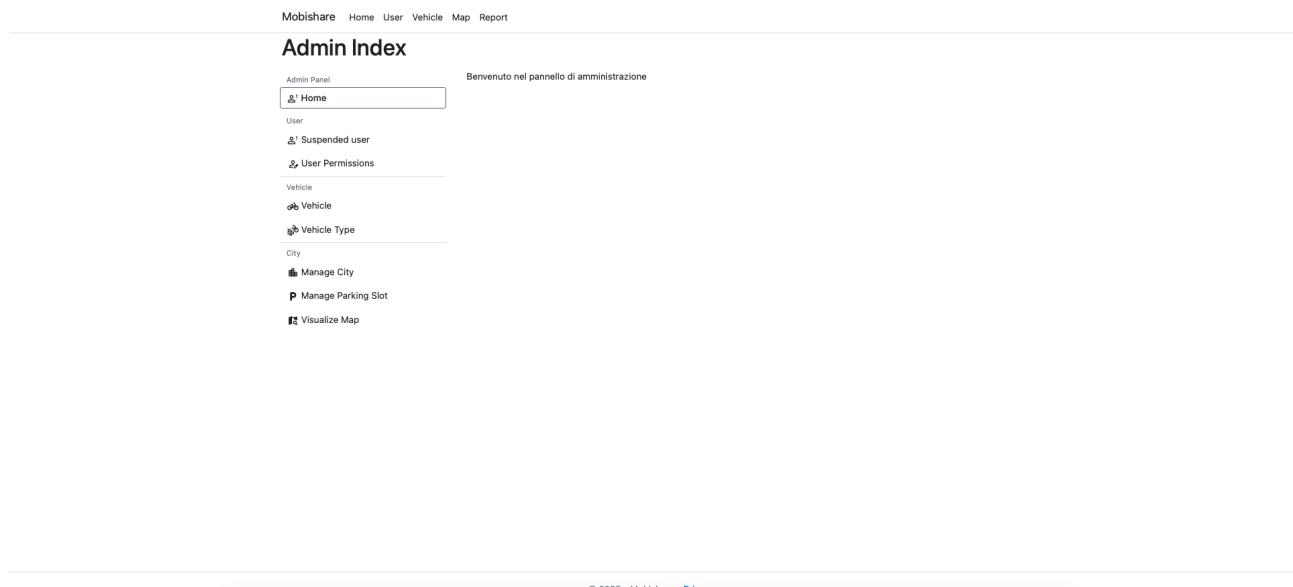
- ricarica saldo;
- monitoraggio transizioni;
- credito residuo.



3.2 Pannello admin

La pagina di admin permette di:

- visualizzare le operazioni disponibili;



Admin Index

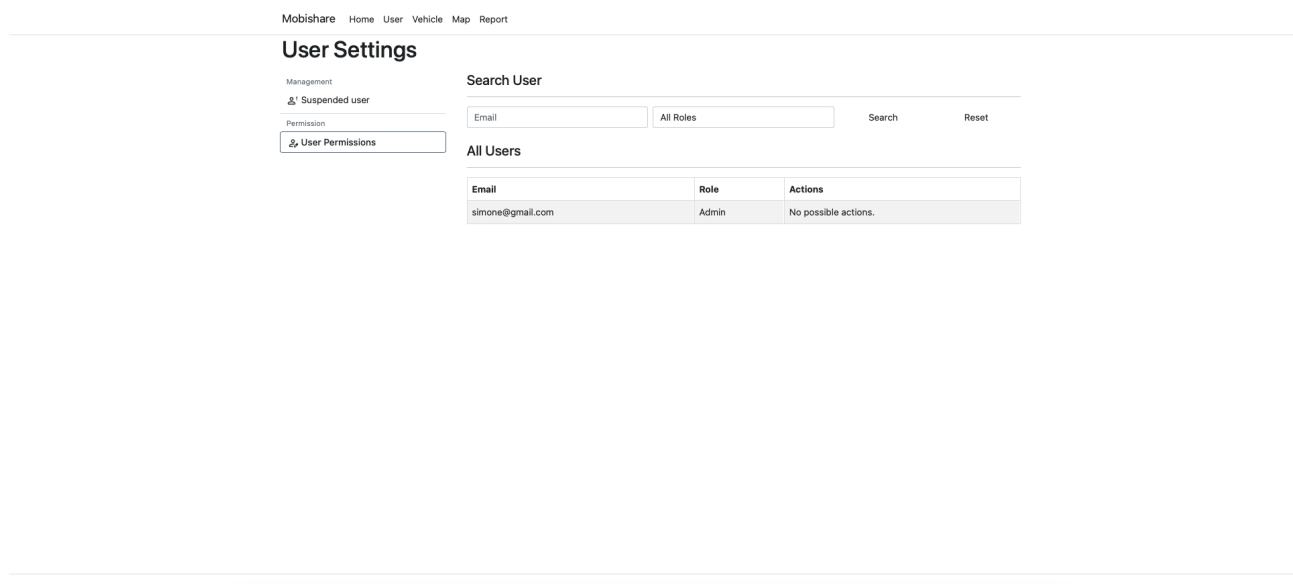
Benvenuto nel pannello di amministrazione

Admin Panel

- Home**
- Suspended user
- User Permissions
- Vehicle
- Vehicle Type
- City
- Manage City
- Manage Parking Slot
- Visualize Map

© 2025 - Mobishare - [Privacy](#)

- gestire i ruoli degli utenti;
- possibilità di bloccare utenti (ad esempio, un utente che non paga o non rispetta le policy);



User Settings

Management

- Suspended user
- User Permissions**

Search User

Email	All Roles	Search	Reset

All Users

Email	Role	Actions
simone@gmail.com	Admin	No possible actions.

© 2025 - Mobishare - [Privacy](#)

- creare delle zone geofencing (zone operative/vietate);

Mobishare Home User Vehicle Map Report

Map Management

Management

- [Manage City](#)
- [Manage Parking Slot](#)
- [Map Visualization](#)
- [Visualize Map](#)

Add New City



Clear Polygon (i)

Mapped area*

City Name*

Add New City

Existing Cities

City Name	Actions
Milano	

- specifiche tecniche (modello, targa, ...);
- tariffazione personalizzata.

Mobishare Home User Vehicle Map Report

Vehicle Management

Management

- [Vehicle](#)
- [Vehicle Type](#)

Add New Vehicle Type

Vehicle Model*

Vehicle Type*

Select Vehicle Type

Price Per Minute (€)*

Add New Vehicle Type

Existing Vehicle Types

No vehicle types available.

4 RESTful API Interfaces

Di seguito sono elencate le API presenti all'interno del sistema.

API Spec

2025-09-24

Mobishare API

Overview

API documentation for Mobishare.

Version

v1

POST /api/Balance

Create a new balance.

Creates and stores a new balance.

Response 201:

Message created successfully.

Content: [text/plain](#) | [CreateBalance](#)

```
{  
    id: integer;  
    credit: number;  
    points: integer;  
    user: IdentityUser;  
    userId: string;  
}
```

Content: [application/json](#) | [CreateBalance](#)

```
{  
    id: integer;  
    credit: number;  
    points: integer;  
    user: IdentityUser;  
    userId: string;  
}
```

Content: [text/json](#) | [CreateBalance](#)

```
{  
    id: integer;  
    credit: number;  
    points: integer;  
    user: IdentityUser;  
    userId: string;  
}
```

Response 400:

Invalid request payload.

PUT /api/Balance

Update an existing balance.

Updates an existing balance.

Request Body:

Balance update payload.

Content: application/json | [UpdateBalance](#)

```
{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: text/json | [UpdateBalance](#)

```
{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: application/*+json | [UpdateBalance](#)

```
{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
  userId: string;
}
```

Response 200:

Balance updated successfully.

Content: text/plain | [UpdateBalance](#)

```
{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: application/json | [UpdateBalance](#)

```
{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: text/json | [UpdateBalance](#)

```
{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
  userId: string;
}
```

```
}
```

Response 400:

Invalid request payload.

GET /api/Balance/{userId}

Get balance by user ID.

Retrieves the balance for a specific user.

Request Parameters:

`userId: string;` // The ID of the user whose balance is to be retrieved.

Response 200:

Balance retrieved successfully.

Content: `text/plain` | [CreateBalance](#)

```
{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: `application/json` | [CreateBalance](#)

```
{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: `text/json` | [CreateBalance](#)

```
{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
  userId: string;
}
```

Response 404:

Balance not found for the specified user ID.

POST /api/ChatMessage

Create a new chat message.

Creates and stores a new message in the specified conversation.

Request Body:

Message creation payload.

Content: application/json | [CreateChatMessage](#)

```
{  
    id: integer;  
    conversation: Conversation;  
    conversationId: integer;  
    sender: string;  
    message: string;  
    embedding: string;  
    createdAt: string;  
}
```

Content: text/json | [CreateChatMessage](#)

```
{  
    id: integer;  
    conversation: Conversation;  
    conversationId: integer;  
    sender: string;  
    message: string;  
    embedding: string;  
    createdAt: string;  
}
```

Content: application/*+json | [CreateChatMessage](#)

```
{  
    id: integer;  
    conversation: Conversation;  
    conversationId: integer;  
    sender: string;  
    message: string;  
    embedding: string;  
    createdAt: string;  
}
```

Response 201:

Message created successfully.

Content: text/plain | [CreateChatMessage](#)

```
{  
    id: integer;  
    conversation: Conversation;  
    conversationId: integer;  
    sender: string;  
    message: string;  
    embedding: string;  
    createdAt: string;  
}
```

Content: application/json | [CreateChatMessage](#)

```
{  
    id: integer;  
    conversation: Conversation;  
    conversationId: integer;  
    sender: string;  
    message: string;  
    embedding: string;  
    createdAt: string;  
}
```

Content: text/json | [CreateChatMessage](#)

```
{
  id: integer;
  conversation: Conversation;
  conversationId: integer;
  sender: string;
  message: string;
  embedding: string;
  createdAt: string;
}
```

Response 400:

Invalid request payload.

GET /api/ChatMessage/{conversationId}

Get Messages by Conversation ID.

Returns a list of chat messages for a specific conversation.

Request Parameters:

conversationId: integer; // Conversation ID.

Response 200:

List of chat messages.

Content: text/plain | [Array<GetMessagesByConversationId>](#)

```
{
  conversationId: integer;
}
```

Content: application/json | [Array<GetMessagesByConversationId>](#)

```
{
  conversationId: integer;
}
```

Content: text/json | [Array<GetMessagesByConversationId>](#)

```
{
  conversationId: integer;
}
```

Response 404:

No messages found for this conversation.

POST /api/City

Create a new city.

Creates and stores a new message in the specified conversation.

Request Body:

Content: application/json | [CreateCity](#)

```
{  
    id: integer;  
    name: string;  
    perimeterLocation: string;  
    createdAt: string;  
    user: IdentityUser;  
    userId: string;  
}  
Content: text/json | CreateCity  
{  
    id: integer;  
    name: string;  
    perimeterLocation: string;  
    createdAt: string;  
    user: IdentityUser;  
    userId: string;  
}  
Content: application/*+json | CreateCity  
{  
    id: integer;  
    name: string;  
    perimeterLocation: string;  
    createdAt: string;  
    user: IdentityUser;  
    userId: string;  
}
```

Response 201:

Message created successfully.

Content: text/plain | [CreateCity](#)

```
{  
    id: integer;  
    name: string;  
    perimeterLocation: string;  
    createdAt: string;  
    user: IdentityUser;  
    userId: string;  
}
```

Content: application/json | [CreateCity](#)

```
{  
    id: integer;  
    name: string;  
    perimeterLocation: string;  
    createdAt: string;  
    user: IdentityUser;  
    userId: string;  
}
```

Content: text/json | [CreateCity](#)

```
{  
    id: integer;  
    name: string;  
    perimeterLocation: string;  
    createdAt: string;  
    user: IdentityUser;
```

```

    userId: string;
}

```

Response 400:

Invalid request payload.

Response 500:

Internal server error.

PUT /api/City

Update an existing city.

Updates the details of an existing city.

Request Body:

City update request.

Content: application/json | [UpdateCity](#)

```
{
  id: integer;
  name: string;
  perimeterLocation: string;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}
```

Content: text/json | [UpdateCity](#)

```
{
  id: integer;
  name: string;
  perimeterLocation: string;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}
```

Content: application/*+json | [UpdateCity](#)

```
{
  id: integer;
  name: string;
  perimeterLocation: string;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}
```

Response 200:

City updated successfully.

Content: text/plain | [UpdateCity](#)

```
{
  id: integer;
  name: string;
```

```

perimeterLocation: string;
createdAt: string;
user: IdentityUser;
userId: string;
}

Content: application/json | UpdateCity

{
  id: integer;
  name: string;
  perimeterLocation: string;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}

Content: text/json | UpdateCity

{
  id: integer;
  name: string;
  perimeterLocation: string;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}

```

Response 400:

Invalid request payload.

Response 404:

City not found.

Response 500:

Internal server error.

DELETE /api/City/{id}

Delete a city.

Deletes a city by its ID.

Request Parameters:

`id: integer;` // The ID of the city to delete.

Response 204:

City deleted successfully.

Response 404:

City not found.

Response 500:

Internal server error.

GET /api/City/AllCities

Get all cities.

Retrieves a list of all cities.

Response 200:

List of cities retrieved successfully.

Content: [text/plain](#) | [Array<City>](#)

```
{
  id: integer;
  name: string;
  perimeterLocation: string;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}
```

Content: [application/json](#) | [Array<City>](#)

```
{
  id: integer;
  name: string;
  perimeterLocation: string;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}
```

Content: [text/json](#) | [Array<City>](#)

```
{
  id: integer;
  name: string;
  perimeterLocation: string;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}
```

Response 500:

Internal server error.

POST /api/Conversation

Create a new conversation.

Creates and stores a new conversation.

Request Body:

Conversation creation payload.

Content: [application/json](#) | [CreateConversation](#)

```
{
  id: integer;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}
```

```

        isActive: boolean;
    }

Content: text/json | CreateConversation

{
    id: integer;
    createdAt: string;
    user: IdentityUser;
    userId: string;
    isActive: boolean;
}

Content: application/*+json | CreateConversation

{
    id: integer;
    createdAt: string;
    user: IdentityUser;
    userId: string;
    isActive: boolean;
}

```

Response 201:

Conversation created successfully.

Content: text/plain | [CreateConversation](#)

```
{
    id: integer;
    createdAt: string;
    user: IdentityUser;
    userId: string;
    isActive: boolean;
}
```

Content: application/json | [CreateConversation](#)

```
{
    id: integer;
    createdAt: string;
    user: IdentityUser;
    userId: string;
    isActive: boolean;
}
```

Content: text/json | [CreateConversation](#)

```
{
    id: integer;
    createdAt: string;
    user: IdentityUser;
    userId: string;
    isActive: boolean;
}
```

Response 400:

Invalid request payload.

GET /api/Conversation/Close/{conversationId}

Close a conversation.

Closes a conversation by its ID.

Request Parameters:

conversationId: **integer**; // The ID of the conversation to close.

Response 200:

OK.

GET /api/Conversation/AllConversations

Get all conversations.

Retrieves all conversations.

Response 200:

List of conversations.

Content: **text/plain** | [Array< GetAllConversations >](#)

Content: **application/json** | [Array< GetAllConversations >](#)

Content: **text/json** | [Array< GetAllConversations >](#)

Response 404:

No conversations found.

GET /api/Conversation/AllConversationsByUserId/{userId}

Get all conversations by user ID.

Retrieves all conversations for a specific user.

Request Parameters:

userId: **string**; // User ID.

Response 200:

List of conversations.

Content: **text/plain** | [Array< GetAllConversationsByUserId >](#)

```
{  
  userId: string;  
}
```

Content: **application/json** | [Array< GetAllConversationsByUserId >](#)

```
{  
  userId: string;  
}
```

Content: **text/json** | [Array< GetAllConversationsByUserId >](#)

```
{  
  userId: string;  
}
```

Response 404:

No conversations found for the user.

GET /api/Conversation/{conversationId}

Get Conversation by Conversation ID.

Retrieves a conversation by its ID.

Request Parameters:

conversationId: **integer**; // Conversation ID.

Response 200:

List of chat messages.

Content: **text/plain** | [Array<GetMessagesByConversationId>](#)

```
{
    conversationId: integer;
}
```

Content: **application/json** | [Array<GetMessagesByConversationId>](#)

```
{
    conversationId: integer;
}
```

Content: **text/json** | [Array<GetMessagesByConversationId>](#)

```
{
    conversationId: integer;
}
```

Response 404:

No messages found for this conversation.

POST /api/HistoryCredit

Create History Credit.

Creates a new history credit record.

Request Body:

Content: **application/json** | [CreateHistoryCredit](#)

```
{
    id: integer;
    credit: number;
    user: IdentityUser;
    userId: string;
    balance: Balance;
    balanceId: integer;
    transactionType: string;
    createdAt: string;
}
```

Content: **text/json** | [CreateHistoryCredit](#)

```
{
  id: integer;
  credit: number;
  user: IdentityUser;
  userId: string;
  balance: Balance;
  balanceId: integer;
  transactionType: string;
  createdAt: string;
}

Content: application/*+json | CreateHistoryCredit

{
  id: integer;
  credit: number;
  user: IdentityUser;
  userId: string;
  balance: Balance;
  balanceId: integer;
  transactionType: string;
  createdAt: string;
}
```

Response 200:

OK.

GET /api/HistoryCredit/AllHistoryCredits/{userId}

Get All History Credits.

Retrieves all history credits for a specific user.

Request Parameters:

userId: string; // User ID.

Response 200:

OK.

POST /api/HistoryPoint

Create a new history point.

Creates and stores a new history point.

Request Body:

Content: application/json | [CreateHistoryPoint](#)

```
{
  id: integer;
  point: integer;
  user: IdentityUser;
  userId: string;
  balance: Balance;
  balanceId: integer;
  transactionType: string;
```

```

    createdAt: string;
}

Content: text/json | CreateHistoryPoint

{
  id: integer;
  point: integer;
  user: IdentityUser;
  userId: string;
  balance: Balance;
  balanceId: integer;
  transactionType: string;
  createdAt: string;
}

Content: application/*+json | CreateHistoryPoint

{
  id: integer;
  point: integer;
  user: IdentityUser;
  userId: string;
  balance: Balance;
  balanceId: integer;
  transactionType: string;
  createdAt: string;
}

```

Response 201:

Message created successfully.

Content: text/plain | [CreateHistoryPoint](#)

```
{
  id: integer;
  point: integer;
  user: IdentityUser;
  userId: string;
  balance: Balance;
  balanceId: integer;
  transactionType: string;
  createdAt: string;
}
```

Content: application/json | [CreateHistoryPoint](#)

```
{
  id: integer;
  point: integer;
  user: IdentityUser;
  userId: string;
  balance: Balance;
  balanceId: integer;
  transactionType: string;
  createdAt: string;
}
```

Content: text/json | [CreateHistoryPoint](#)

```
{
  id: integer;
  point: integer;
  user: IdentityUser;
```

```

userId: string;
balance: Balance;
balanceId: integer;
transactionType: string;
createdAt: string;
}

```

Response 400:

Invalid request payload.

POST /api/MessagePair

Create a Message Pair.

Creates a new message pair in the chat system.

Request Body:

Content: application/json | [CreateMessagePair](#)

```
{
  id: integer;
  userMessage: ChatMessage;
  userMessageId: integer;
  aiMessage: ChatMessage;
  aiMessageId: integer;
  isForRag: boolean;
  sourceType: string;
  answered: boolean;
  language: string;
}
```

Content: text/json | [CreateMessagePair](#)

```
{
  id: integer;
  userMessage: ChatMessage;
  userMessageId: integer;
  aiMessage: ChatMessage;
  aiMessageId: integer;
  isForRag: boolean;
  sourceType: string;
  answered: boolean;
  language: string;
}
```

Content: application/*+json | [CreateMessagePair](#)

```
{
  id: integer;
  userMessage: ChatMessage;
  userMessageId: integer;
  aiMessage: ChatMessage;
  aiMessageId: integer;
  isForRag: boolean;
  sourceType: string;
  answered: boolean;
  language: string;
}
```

Response 201:

Message pair created successfully.

Content: text/plain | [CreateMessagePair](#)

```
{
  id: integer;
  userMessage: ChatMessage;
  userMessageId: integer;
  aiMessage: ChatMessage;
  aiMessageId: integer;
  isForRag: boolean;
  sourceType: string;
  answered: boolean;
  language: string;
}
```

Content: application/json | [CreateMessagePair](#)

```
{
  id: integer;
  userMessage: ChatMessage;
  userMessageId: integer;
  aiMessage: ChatMessage;
  aiMessageId: integer;
  isForRag: boolean;
  sourceType: string;
  answered: boolean;
  language: string;
}
```

Content: text/json | [CreateMessagePair](#)

```
{
  id: integer;
  userMessage: ChatMessage;
  userMessageId: integer;
  aiMessage: ChatMessage;
  aiMessageId: integer;
  isForRag: boolean;
  sourceType: string;
  answered: boolean;
  language: string;
}
```

Response 400:

Invalid request data.

Response 500:

Internal server error.

POST /api/ParkingSlot

Create a Parking Slot.

Creates a new parking slot in the system.

Request Body:

Content: application/json | [CreateParkingSlot](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: text/json | [CreateParkingSlot](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: application/*+json | [CreateParkingSlot](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Response 201:

Parking slot created successfully.

Content: text/plain | [CreateParkingSlot](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: application/json | [CreateParkingSlot](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
}
```

```

cityId: integer;
user: IdentityUser;
userId: string;
}

Content: text/json | CreateParkingSlot

{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}

```

Response 400:

Invalid request data.

Response 500:

Internal server error.

PUT /api/ParkingSlot

Update a Parking Slot.

Updates an existing parking slot in the system.

Request Body:

Content: application/json | [UpdateParkingSlot](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: text/json | [UpdateParkingSlot](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: application/*+json | [UpdateParkingSlot](#)

```
{
}
```

```

id: integer;
perimeterLocation: string;
createdAt: string;
type: string;
city: City;
cityId: integer;
user: IdentityUser;
userId: string;
}

```

Response 200:

Parking slot updated successfully.

Content: [text/plain](#) | [UpdateParkingSlot](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: [application/json](#) | [UpdateParkingSlot](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: [text/json](#) | [UpdateParkingSlot](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Response 400:

Invalid request data.

Response 404:

Parking slot not found.

Response 500:

Internal server error.

DELETE /api/ParkingSlot/{id}

Delete a Parking Slot.

Deletes a parking slot from the system.

Request Parameters:

`id: integer;` // The ID of the parking slot to delete.

Response 204:

Parking slot deleted successfully.

Response 404:

Parking slot not found.

Response 500:

Internal server error.

GET /api/ParkingSlot/AllParkingSlots

Get all Parking Slots.

Retrieves all parking slots in the system.

Response 200:

List of parking slots retrieved successfully.

Content: `text/plain | Array<ParkingSlot>`

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: `application/json | Array<ParkingSlot>`

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: `text/json | Array<ParkingSlot>`

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Response 500:

Internal server error.

GET /api/ParkingSlot/AllAvailableParkingSlots

Get all Available Parking Slots.

Retrieves all Available parking slots in the system.

Response 200:

List of parking slots retrieved successfully.

Content: text/plain | [Array<ParkingSlot>](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: application/json | [Array<ParkingSlot>](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Content: text/json | [Array<ParkingSlot>](#)

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

```
}
```

Response 500:

Internal server error.

POST /api/Position

Create a new position for a vehicle.

Creates a new position for a vehicle based on the provided details.

Request Body:

Content: application/json | [CreatePosition](#)

```
{
  id: integer;
  latitude: number;
  longitude: number;
  gpsReceptionTime: string;
  gpsEmissionTime: string;
  vehicleId: integer;
}
```

Content: text/json | [CreatePosition](#)

```
{
  id: integer;
  latitude: number;
  longitude: number;
  gpsReceptionTime: string;
  gpsEmissionTime: string;
  vehicleId: integer;
}
```

Content: application/*+json | [CreatePosition](#)

```
{
  id: integer;
  latitude: number;
  longitude: number;
  gpsReceptionTime: string;
  gpsEmissionTime: string;
  vehicleId: integer;
}
```

Response 201:

Position created successfully.

Content: text/plain | [Position](#)

```
{
  id: integer;
  latitude: number;
  longitude: number;
  gpsReceptionTime: string;
  gpsEmissionTime: string;
  vehicleId: integer;
}
```

Content: application/json | [Position](#)

```
{
  id: integer;
  latitude: number;
  longitude: number;
  gpsReceptionTime: string;
  gpsEmissionTime: string;
  vehicleId: integer;
}
Content: text/json | Position
{
  id: integer;
  latitude: number;
  longitude: number;
  gpsReceptionTime: string;
  gpsEmissionTime: string;
  vehicleId: integer;
}
```

Response 400:

Invalid request data.

Response 500:

Internal server error.

GET /api/Position/{vehicleId}

Get the current position of a vehicle.

Retrieves the current position of a vehicle based on its ID.

Request Parameters:

VehicleId: integer; // The unique identifier of the vehicle.

Response 200:

Position retrieved successfully.

Content: text/plain | [Position](#)

```
{
  id: integer;
  latitude: number;
  longitude: number;
  gpsReceptionTime: string;
  gpsEmissionTime: string;
  vehicleId: integer;
}
```

Content: application/json | [Position](#)

```
{
  id: integer;
  latitude: number;
  longitude: number;
  gpsReceptionTime: string;
  gpsEmissionTime: string;
  vehicleId: integer;
```

```

}

Content: text/json | Position

{
  id: integer;
  latitude: number;
  longitude: number;
  gpsReceptionTime: string;
  gpsEmissionTime: string;
  vehicleId: integer;
}

```

Response 404:

Position not found.

Response 500:

Internal server error.

POST /api/Repair

Create a new repair.

Creates a new repair.

Request Body:

Content: application/json | [CreateRepair](#)

```

{
  id: integer;
  description: string;
  createdAt: string;
  report: Report;
  reportId: integer;
}

```

Content: text/json | [CreateRepair](#)

```

{
  id: integer;
  description: string;
  createdAt: string;
  report: Report;
  reportId: integer;
}

```

Content: application/*+json | [CreateRepair](#)

```

{
  id: integer;
  description: string;
  createdAt: string;
  report: Report;
  reportId: integer;
}

```

Response 201:

Repair created successfully.

Content: text/plain | [Repair](#)

```
{
  id: integer;
  description: string;
  createdAt: string;
  report: Report;
  reportId: integer;
}
```

Content: application/json | [Repair](#)

```
{
  id: integer;
  description: string;
  createdAt: string;
  report: Report;
  reportId: integer;
}
```

Content: text/json | [Repair](#)

```
{
  id: integer;
  description: string;
  createdAt: string;
  report: Report;
  reportId: integer;
}
```

Response 400:

Invalid request data.

Response 500:

Internal server error.

POST /api/Report

Create a new report.

This endpoint allows you to create a new report.

Request Body:

Content: application/json | [CreateReport](#)

```
{
  id: integer;
  description: string;
  createdAt: string;
  image: string;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
  status: string;
  repairs: Array<Repair>;
}
```

Content: text/json | [CreateReport](#)

```
{
  id: integer;
  description: string;
  createdAt: string;
  image: string;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
  status: string;
  repairs: Array<Repair>;
}
Content: application/*+json | CreateReport
{
  id: integer;
  description: string;
  createdAt: string;
  image: string;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
  status: string;
  repairs: Array<Repair>;
}
```

Response 201:

Report created successfully.

```
Content: text/plain | CreateReport
{
  id: integer;
  description: string;
  createdAt: string;
  image: string;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
  status: string;
  repairs: Array<Repair>;
}
Content: application/json | CreateReport
{
  id: integer;
  description: string;
  createdAt: string;
  image: string;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
  status: string;
  repairs: Array<Repair>;
}
Content: text/json | CreateReport
```

```
{
  id: integer;
  description: string;
  createdAt: string;
  image: string;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
  status: string;
  repairs: Array<Repair>;
}
```

Response 400:

Invalid request payload.

GET /api/Report/AllReports/{userId}

Get all reports for a technician.

This endpoint retrieves all reports for a specific user.

Request Parameters:

`userId: string; // The ID of the user whose reports are to be retrieved.`

Response 200:

Reports retrieved successfully.

Content: `text/plain | Array<Report>`

```
{
  id: integer;
  description: string;
  createdAt: string;
  image: string;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
  status: string;
  repairs: Array<Repair>;
}
```

Content: `application/json | Array<Report>`

```
{
  id: integer;
  description: string;
  createdAt: string;
  image: string;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
  status: string;
  repairs: Array<Repair>;
}
```

Content: `text/json | Array<Report>`

```
{
  id: integer;
  description: string;
  createdAt: string;
  image: string;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
  status: string;
  repairs: Array<Repair>;
}
```

Response 404:

User not found or no reports found.

PUT /api/Report/Assign

Change report status.

This endpoint change report status to Assigned.

Request Body:

Content: application/json | [UpdateReport](#)

```
{
  id: integer;
  status: string;
}
```

Content: text/json | [UpdateReport](#)

```
{
  id: integer;
  status: string;
}
```

Content: application/*+json | [UpdateReport](#)

```
{
  id: integer;
  status: string;
}
```

Response 200:

Reports retrieved successfully.

Content: text/plain | [UpdateReport](#)

```
{
  id: integer;
  status: string;
}
```

Content: application/json | [UpdateReport](#)

```
{
  id: integer;
  status: string;
}
```

Content: text/json | [UpdateReport](#)

```
{  
    id: integer;  
    status: string;  
}
```

Response 400:

Invalid request payload.

Response 404:

User not found or no reports found.

PUT /api/Report/Close

Change report status.

This endpoint change report status to Closed.

Request Body:

Content: application/json | [UpdateReport](#)

```
{  
    id: integer;  
    status: string;  
}
```

Content: text/json | [UpdateReport](#)

```
{  
    id: integer;  
    status: string;  
}
```

Content: application/*+json | [UpdateReport](#)

```
{  
    id: integer;  
    status: string;  
}
```

Response 200:

Reports retrieved successfully.

Content: text/plain | [UpdateReport](#)

```
{  
    id: integer;  
    status: string;  
}
```

Content: application/json | [UpdateReport](#)

```
{  
    id: integer;  
    status: string;  
}
```

Content: text/json | [UpdateReport](#)

```
{
```

```

    id: integer;
    status: string;
}

```

Response 400:

Invalid request payload.

Response 404:

User not found or no reports found.

POST /api/ReportAssignment

Create a new report assignment.

This endpoint allows you to create a new report assignment.

Request Body:

Content: application/json | [CreateReportAssignment](#)

```
{
    id: integer;
    user: IdentityUser;
    userId: string;
    report: Report;
    reportId: integer;
}
```

Content: text/json | [CreateReportAssignment](#)

```
{
    id: integer;
    user: IdentityUser;
    userId: string;
    report: Report;
    reportId: integer;
}
```

Content: application/*+json | [CreateReportAssignment](#)

```
{
    id: integer;
    user: IdentityUser;
    userId: string;
    report: Report;
    reportId: integer;
}
```

Response 201:

Report assignment created successfully.

Content: text/plain | [CreateReportAssignment](#)

```
{
    id: integer;
    user: IdentityUser;
    userId: string;
    report: Report;
    reportId: integer;
}
```

Content: application/json | [CreateReportAssignment](#)

```
{
  id: integer;
  user: IdentityUser;
  userId: string;
  report: Report;
  reportId: integer;
}
```

Content: text/json | [CreateReportAssignment](#)

```
{
  id: integer;
  user: IdentityUser;
  userId: string;
  report: Report;
  reportId: integer;
}
```

Response 400:

Invalid request payload.

POST /api/Ride

Create a new ride.

This endpoint allows you to create a new ride for a vehicle.

Request Body:

Content: application/json | [CreateRide](#)

```
{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}
```

Content: text/json | [CreateRide](#)

```
{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
```

```

user: IdentityUser;
userId: string;
vehicle: Vehicle;
vehicleId: integer;
}

Content: application/*+json | CreateRide

{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}

```

Response 201:

Ride created successfully.

Content: text/plain | [CreateRide](#)

```
{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}
```

Content: application/json | [CreateRide](#)

```
{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}
```

Content: text/json | [CreateRide](#)

```
{
  id: integer;
  name: string;
  startTime: string;
  endTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}
```

Response 400:

Invalid request payload.

Response 404:

Vehicle not found.

PUT /api/Ride

Update an existing ride.

This endpoint allows you to update an existing ride for a vehicle.

Request Body:

Content: application/json | [UpdateRide](#)

```
{
  id: integer;
  startTime: string;
  endTime: string;
  price: number;
  positionStartId: integer;
  positionEndId: integer;
  userId: string;
  vehicleId: integer;
  tripName: string;
}
```

Content: text/json | [UpdateRide](#)

```
{
  id: integer;
  startTime: string;
  endTime: string;
  price: number;
  positionStartId: integer;
  positionEndId: integer;
  userId: string;
  vehicleId: integer;
  tripName: string;
}
```

Content: application/*+json | [UpdateRide](#)

```
{
  id: integer;
  startTime: string;
  endTime: string;
  price: number;
  positionStartId: integer;
  positionEndId: integer;
  userId: string;
  vehicleId: integer;
  tripName: string;
}
```

Response 200:

Ride updated successfully.

Content: text/plain | [UpdateRide](#)

```
{
  id: integer;
  startTime: string;
  endTime: string;
  price: number;
  positionStartId: integer;
  positionEndId: integer;
  userId: string;
  vehicleId: integer;
  tripName: string;
}
```

Content: application/json | [UpdateRide](#)

```
{
  id: integer;
  startTime: string;
  endTime: string;
  price: number;
  positionStartId: integer;
  positionEndId: integer;
  userId: string;
  vehicleId: integer;
  tripName: string;
}
```

Content: text/json | [UpdateRide](#)

```
{
  id: integer;
  startTime: string;
  endTime: string;
  price: number;
  positionStartId: integer;
  positionEndId: integer;
  userId: string;
  vehicleId: integer;
  tripName: string;
}
```

Response 400:

Invalid request payload.

Response 404:

Ride not found.

GET /api/Ride/AllRides

Get all rides.

This endpoint retrieves all rides for a vehicle.

Response 200:

Rides retrieved successfully.

Content: [text/plain](#) | [Array<Ride>](#)

```
{  
    id: integer;  
    name: string;  
    startDateTime: string;  
    endDateTime: string;  
    price: number;  
    positionStart: Position;  
    positionStartId: integer;  
    positionEnd: Position;  
    positionEndId: integer;  
    user: IdentityUser;  
    userId: string;  
    vehicle: Vehicle;  
    vehicleId: integer;  
}
```

Content: [application/json](#) | [Array<Ride>](#)

```
{  
    id: integer;  
    name: string;  
    startDateTime: string;  
    endDateTime: string;  
    price: number;  
    positionStart: Position;  
    positionStartId: integer;  
    positionEnd: Position;  
    positionEndId: integer;  
    user: IdentityUser;  
    userId: string;  
    vehicle: Vehicle;  
    vehicleId: integer;  
}
```

Content: [text/json](#) | [Array<Ride>](#)

```
{  
    id: integer;  
    name: string;  
    startDateTime: string;  
    endDateTime: string;  
    price: number;  
    positionStart: Position;  
    positionStartId: integer;  
    positionEnd: Position;  
    positionEndId: integer;
```

```

user: IdentityUser;
userId: string;
vehicle: Vehicle;
vehicleId: integer;
}

```

Response 404:

No rides found.

GET /api/Ride/AllUserRides/{userId}

Get all rides for a user.

This endpoint retrieves all rides for a specific user.

Request Parameters:

userId: string; // The ID of the user whose rides are to be retrieved.

Response 200:

User rides retrieved successfully.

Content: [text/plain](#) | [Array<Ride>](#)

```
{
  id: integer;
  name: string;
  startTime: string;
  endTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}
```

Content: [application/json](#) | [Array<Ride>](#)

```
{
  id: integer;
  name: string;
  startTime: string;
  endTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}
```

Content: [text/json](#) | [Array<Ride>](#)

```
{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}
```

Response 404:

User not found or no rides found.

GET /api/Ride/User/{userId}

Get the last ride for a specific user.

This endpoint retrieves the last ride for a specific user.

Request Parameters:

userId: string; // The ID of the user whose ride is to be retrieved.

Response 200:

Ride retrieved successfully.

Content: text/plain | [Ride](#)

```
{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}
```

Content: application/json | [Ride](#)

```
{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position:
```

```

positionStartId: integer;
positionEnd: Position;
positionEndId: integer;
user: IdentityUser;
userId: string;
vehicle: Vehicle;
vehicleId: integer;
}

Content: text/json | Ride

{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}

```

Response 404:

Ride not found.

GET /api/Ride/{ridId}

Get a ride by its ID.

This endpoint retrieves a ride by its unique identifier.

Request Parameters:

ridId: integer; // The ID of the ride to retrieve.

Response 200:

Ride retrieved successfully.

Content: text/plain | [Ride](#)

```

{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}

```

```
}
```

Content: application/json | [Ride](#)

```
{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}
```

Content: text/json | [Ride](#)

```
{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}
```

Response 404:

Ride not found.

GET /api/TechnicianApis/GetTechniciansReports

Get technician reports.

Retrieves all technician reports.

Response 200:

OK.

POST /api/UserContext/SetLocation

Request Body:

Content: application/json | [LocationDto](#)

```
{
```

```

    lat: number;
    lon: number;
}

Content: text/json | LocationDto

{
    lat: number;
    lon: number;
}
}

Content: application/*+json | LocationDto

{
    lat: number;
    lon: number;
}

```

Response 200:

OK.

POST /api/Vehicle

Create a new vehicle.

This endpoint allows you to create a new vehicle.

Request Body:

Content: application/json | [CreateVehicle](#)

```
{
    id: integer;
    plate: string;
    status: string;
    batteryLevel: number;
    createdAt: string;
    parkingSlot: ParkingSlot;
    parkingSlotId: integer;
    vehicleType: VehicleType;
    vehicleTypeId: integer;
    positions: Array<Position>;
}
```

Content: text/json | [CreateVehicle](#)

```
{
    id: integer;
    plate: string;
    status: string;
    batteryLevel: number;
    createdAt: string;
    parkingSlot: ParkingSlot;
    parkingSlotId: integer;
    vehicleType: VehicleType;
    vehicleTypeId: integer;
    positions: Array<Position>;
}
```

Content: application/*+json | [CreateVehicle](#)

```
{
```

```

id: integer;
plate: string;
status: string;
batteryLevel: number;
createdAt: string;
parkingSlot: ParkingSlot;
parkingSlotId: integer;
vehicleType: VehicleType;
vehicleTypeId: integer;
positions: Array<Position>;
}

```

Response 201:

Vehicle created successfully.

Content: text/plain | [CreateVehicle](#)

```
{
id: integer;
plate: string;
status: string;
batteryLevel: number;
createdAt: string;
parkingSlot: ParkingSlot;
parkingSlotId: integer;
vehicleType: VehicleType;
vehicleTypeId: integer;
positions: Array<Position>;
}

```

Content: application/json | [CreateVehicle](#)

```
{
id: integer;
plate: string;
status: string;
batteryLevel: number;
createdAt: string;
parkingSlot: ParkingSlot;
parkingSlotId: integer;
vehicleType: VehicleType;
vehicleTypeId: integer;
positions: Array<Position>;
}

```

Content: text/json | [CreateVehicle](#)

```
{
id: integer;
plate: string;
status: string;
batteryLevel: number;
createdAt: string;
parkingSlot: ParkingSlot;
parkingSlotId: integer;
vehicleType: VehicleType;
vehicleTypeId: integer;
positions: Array<Position>;
}

```

Response 400:

Invalid request payload.

PUT /api/Vehicle

Update an existing vehicle.

This endpoint allows you to update an existing vehicle.

Request Body:

Content: application/json | [UpdateVehicle](#)

```
{
  id: integer;
  plate: string;
  status: string;
  batteryLevel: number;
  createdAt: string;
  parkingSlot: ParkingSlot;
  parkingSlotId: integer;
  vehicleType: VehicleType;
  vehicleTypeId: integer;
  positions: Array<Position>;
}
```

Content: text/json | [UpdateVehicle](#)

```
{
  id: integer;
  plate: string;
  status: string;
  batteryLevel: number;
  createdAt: string;
  parkingSlot: ParkingSlot;
  parkingSlotId: integer;
  vehicleType: VehicleType;
  vehicleTypeId: integer;
  positions: Array<Position>;
}
```

Content: application/*+json | [UpdateVehicle](#)

```
{
  id: integer;
  plate: string;
  status: string;
  batteryLevel: number;
  createdAt: string;
  parkingSlot: ParkingSlot;
  parkingSlotId: integer;
  vehicleType: VehicleType;
  vehicleTypeId: integer;
  positions: Array<Position>;
}
```

Response 200:

Vehicle updated successfully.

Content: text/plain | [UpdateVehicle](#)

```
{
```

```

id: integer;
plate: string;
status: string;
batteryLevel: number;
createdAt: string;
parkingSlot: ParkingSlot;
parkingSlotId: integer;
vehicleType: VehicleType;
vehicleTypeId: integer;
positions: Array<Position>;
}

```

Content: application/json | [UpdateVehicle](#)

```
{
id: integer;
plate: string;
status: string;
batteryLevel: number;
createdAt: string;
parkingSlot: ParkingSlot;
parkingSlotId: integer;
vehicleType: VehicleType;
vehicleTypeId: integer;
positions: Array<Position>;
}

```

Content: text/json | [UpdateVehicle](#)

```
{
id: integer;
plate: string;
status: string;
batteryLevel: number;
createdAt: string;
parkingSlot: ParkingSlot;
parkingSlotId: integer;
vehicleType: VehicleType;
vehicleTypeId: integer;
positions: Array<Position>;
}

```

Response 400:

Invalid request payload.

Response 404:

Vehicle not found.

DELETE /api/Vehicle/{id}

Delete a vehicle.

This endpoint allows you to delete a vehicle by its ID.

Request Parameters:

id: **integer**; // The ID of the vehicle to delete.

Response 200:

Vehicle deleted successfully.

Response 404:

Vehicle not found.

Response 500:

Internal server error.

GET /api/Vehicle/{id}

Get vehicle by ID.

This endpoint retrieves a vehicle by its ID.

Request Parameters:

`id: integer;` // The ID of the vehicle to retrieve.

Response 200:

Vehicle retrieved successfully.

Content: `text/plain` | [Vehicle](#)

```
{  
    id: integer;  
    plate: string;  
    status: string;  
    batteryLevel: number;  
    createdAt: string;  
    parkingSlot: ParkingSlot;  
    parkingSlotId: integer;  
    vehicleType: VehicleType;  
    vehicleTypeld: integer;  
    positions: Array<Position>;  
}
```

Content: `application/json` | [Vehicle](#)

```
{  
    id: integer;  
    plate: string;  
    status: string;  
    batteryLevel: number;  
    createdAt: string;  
    parkingSlot: ParkingSlot;  
    parkingSlotId: integer;  
    vehicleType: VehicleType;  
    vehicleTypeld: integer;  
    positions: Array<Position>;  
}
```

Content: `text/json` | [Vehicle](#)

```
{  
    id: integer;  
    plate: string;  
    status: string;  
    batteryLevel: number;  
    createdAt: string;
```

```

parkingSlot: ParkingSlot;
parkingSlotId: integer;
vehicleType: VehicleType;
vehicleTypeId: integer;
positions: Array<Position>;
}

```

Response 404:

Vehicle not found.

GET /api/Vehicle/AllVehicles

Get all vehicles.

This endpoint retrieves all vehicles.

Response 200:

Vehicles retrieved successfully.

Content: [text/plain](#) | [Array<Vehicle>](#)

```
{
  id: integer;
  plate: string;
  status: string;
  batteryLevel: number;
  createdAt: string;
  parkingSlot: ParkingSlot;
  parkingSlotId: integer;
  vehicleType: VehicleType;
  vehicleTypeId: integer;
  positions: Array<Position>;
}
```

Content: [application/json](#) | [Array<Vehicle>](#)

```
{
  id: integer;
  plate: string;
  status: string;
  batteryLevel: number;
  createdAt: string;
  parkingSlot: ParkingSlot;
  parkingSlotId: integer;
  vehicleType: VehicleType;
  vehicleTypeId: integer;
  positions: Array<Position>;
}
```

Content: [text/json](#) | [Array<Vehicle>](#)

```
{
  id: integer;
  plate: string;
  status: string;
  batteryLevel: number;
  createdAt: string;
  parkingSlot: ParkingSlot;
  parkingSlotId: integer;
  vehicleType: VehicleType;
```

```
        vehicleTypeId: integer;
        positions: Array<Position>;
    }
```

GET /api/Vehicle/GetAvailableVehicles

Get all available vehicles.

This endpoint retrieves all available vehicles.

Response 200:

Available vehicles retrieved successfully.

Content: text/plain | [Array<Vehicle>](#)

```
{
    id: integer;
    plate: string;
    status: string;
    batteryLevel: number;
    createdAt: string;
    parkingSlot: ParkingSlot;
    parkingSlotId: integer;
    vehicleType: VehicleType;
    vehicleTypeId: integer;
    positions: Array<Position>;
}
```

Content: application/json | [Array<Vehicle>](#)

```
{
    id: integer;
    plate: string;
    status: string;
    batteryLevel: number;
    createdAt: string;
    parkingSlot: ParkingSlot;
    parkingSlotId: integer;
    vehicleType: VehicleType;
    vehicleTypeId: integer;
    positions: Array<Position>;
}
```

Content: text/json | [Array<Vehicle>](#)

```
{
    id: integer;
    plate: string;
    status: string;
    batteryLevel: number;
    createdAt: string;
    parkingSlot: ParkingSlot;
    parkingSlotId: integer;
    vehicleType: VehicleType;
    vehicleTypeId: integer;
    positions: Array<Position>;
}
```

POST /api/VehicleType

Create a new vehicle type.

This endpoint allows you to create a new vehicle type.

Request Body:

Content: application/json | [CreateVehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

Content: text/json | [CreateVehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

Content: application/*+json | [CreateVehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

Response 200:

Vehicle type created successfully.

Content: text/plain | [VehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

Content: application/json | [VehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

Content: text/json | [VehicleType](#)

```
{
  id: integer;
```

```

model: string;
type: string;
pricePerMinute: number;
createdAt: string;
}

```

Response 400:

Invalid request data.

Response 500:

Internal server error.

PUT /api/VehicleType

Update an existing vehicle type.

This endpoint allows you to update an existing vehicle type.

Request Body:

Content: application/json | [UpdateVehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

Content: text/json | [UpdateVehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

Content: application/*+json | [UpdateVehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

Response 200:

Vehicle type updated successfully.

Content: text/plain | [VehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
```

```

    createdAt: string;
}

Content: application/json | VehicleType

{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}

Content: text/json | VehicleType

{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}

```

Response 400:

Invalid request data.

Response 404:

Vehicle type not found.

Response 500:

Internal server error.

DELETE /api/VehicleType/{id}

Delete a vehicle type.

This endpoint allows you to delete a vehicle type by its ID.

Request Parameters:

`id: integer;` // The ID of the vehicle type to delete.

Response 200:

Vehicle type deleted successfully.

Response 404:

Vehicle type not found.

Response 500:

Internal server error.

GET /api/VehicleType/{id}

Get a vehicle type by its ID.

Retrieves a vehicle type by its unique identifier.

Request Parameters:

`id: integer;` // The ID of the vehicle type to retrieve.

Response 200:

Vehicle type retrieved successfully.

Content: `text/plain` | [VehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

Content: `application/json` | [VehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

Content: `text/json` | [VehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

Response 404:

Vehicle type not found.

GET /api/VehicleType/Vehicle/{id}

Get a vehicle type by vehicle ID.

Retrieves a vehicle type by vehicle Id.

Request Parameters:

`id: integer;` // The ID of the vehicle to retrieve the vehicleType.

Response 200:

Vehicle type retrieved successfully.

Content: `text/plain` | [VehicleType](#)

```
{
  id: integer;
  model: string;
  type: string;
```

```

    pricePerMinute: number;
    createdAt: string;
}
Content: application/json | VehicleType

{
    id: integer;
    model: string;
    type: string;
    pricePerMinute: number;
    createdAt: string;
}
Content: text/json | VehicleType

{
    id: integer;
    model: string;
    type: string;
    pricePerMinute: number;
    createdAt: string;
}

```

Response 404:

Vehicle type not found.

GET /api/VehicleType/AllVehicleTypes

Get all vehicle types.

Retrieves a list of all vehicle types available in the system.

Response 200:

Vehicle types retrieved successfully.

Content: text/plain | [Array<VehicleType>](#)

```
{
    id: integer;
    model: string;
    type: string;
    pricePerMinute: number;
    createdAt: string;
}
```

Content: application/json | [Array<VehicleType>](#)

```
{
    id: integer;
    model: string;
    type: string;
    pricePerMinute: number;
    createdAt: string;
}
```

Content: text/json | [Array<VehicleType>](#)

```
{
    id: integer;
    model: string;
    type: string;
```

```
    pricePerMinute: number;  
    createdAt: string;  
}
```

Response 404:

No vehicle types found.

Response 500:

Internal server error.

Schemas

Balance

```
{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
  userId: string;
}
```

ChatMessage

```
{
  id: integer;
  conversation: Conversation;
  conversationId: integer;
  sender: string;
  message: string;
  embedding: string;
  createdAt: string;
}
```

City

```
{
  id: integer;
  name: string;
  perimeterLocation: string;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}
```

Conversation

```
{
  id: integer;
  createdAt: string;
  user: IdentityUser;
  userId: string;
  isActive: boolean;
}
```

CreateBalance

```
{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
```

```

    userId: string;
}

```

CreateChatMessage

```

{
  id: integer;
  conversation: Conversation;
  conversationId: integer;
  sender: string;
  message: string;
  embedding: string;
  createdAt: string;
}

```

CreateCity

```

{
  id: integer;
  name: string;
  perimeterLocation: string;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}

```

CreateConversation

```

{
  id: integer;
  createdAt: string;
  user: IdentityUser;
  userId: string;
  isActive: boolean;
}

```

CreateHistoryCredit

```

{
  id: integer;
  credit: number;
  user: IdentityUser;
  userId: string;
  balance: Balance;
  balanceId: integer;
  transactionType: string;
  createdAt: string;
}

```

CreateHistoryPoint

```
{
  id: integer;
  point: integer;
  user: IdentityUser;
  userId: string;
  balance: Balance;
  balanceId: integer;
  transactionType: string;
  createdAt: string;
}
```

CreateMessagePair

```
{
  id: integer;
  userMessage: ChatMessage;
  userMessageId: integer;
  aiMessage: ChatMessage;
  aiMessageId: integer;
  isForRag: boolean;
  sourceType: string;
  answered: boolean;
  language: string;
}
```

CreateParkingSlot

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

CreatePosition

```
{
  id: integer;
  latitude: number;
  longitude: number;
  gpsReceptionTime: string;
  gpsEmissionTime: string;
  vehicleId: integer;
}
```

CreateRepair

```
{
  id: integer;
```

```

description: string;
createdAt: string;
report: Report;
reportId: integer;
}

```

CreateReport

```

{
  id: integer;
  description: string;
  createdAt: string;
  image: string;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
  status: string;
  repairs: Array<Repair>;
}

```

CreateReportAssignment

```

{
  id: integer;
  user: IdentityUser;
  userId: string;
  report: Report;
  reportId: integer;
}

```

CreateRide

```

{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}

```

CreateVehicle

```
{
  id: integer;
}
```

```

plate: string;
status: string;
batteryLevel: number;
createdAt: string;
parkingSlot: ParkingSlot;
parkingSlotId: integer;
vehicleType: VehicleType;
vehicleTypeId: integer;
positions: Array<Position>;
}

```

CreateVehicleType

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}

```

GetAllConversations

GetAllConversationsByUserId

```
{
  userId: string;
}

```

GetMessagesByConversationId

```
{
  conversationId: integer;
}

```

IdentityUser

```
{
  id: string;
  userName: string;
  normalizedUserName: string;
  email: string;
  normalizedEmail: string;
  emailConfirmed: boolean;
  passwordHash: string;
  securityStamp: string;
  concurrencyStamp: string;
  phoneNumber: string;
  phoneNumberConfirmed: boolean;
  twoFactorEnabled: boolean;
}

```

```

lockoutEnd: string;
lockoutEnabled: boolean;
accessFailedCount: integer;
}

```

LocationDto

```
{
  lat: number;
  lon: number;
}
```

ParkingSlot

```
{
  id: integer;
  perimeterLocation: string;
  createdAt: string;
  type: string;
  city: City;
  cityId: integer;
  user: IdentityUser;
  userId: string;
}
```

Position

```
{
  id: integer;
  latitude: number;
  longitude: number;
  gpsReceptionTime: string;
  gpsEmissionTime: string;
  vehicleId: integer;
}
```

Repair

```
{
  id: integer;
  description: string;
  createdAt: string;
  report: Report;
  reportId: integer;
}
```

Report

```
{
  id: integer;
  description: string;
```

```

createdAt: string;
image: string;
user: IdentityUser;
userId: string;
vehicle: Vehicle;
vehicleId: integer;
status: string;
repairs: Array<Repair>;
}

```

Ride

```

{
  id: integer;
  name: string;
  startDateTime: string;
  endDateTime: string;
  price: number;
  positionStart: Position;
  positionStartId: integer;
  positionEnd: Position;
  positionEndId: integer;
  user: IdentityUser;
  userId: string;
  vehicle: Vehicle;
  vehicleId: integer;
}

```

UpdateBalance

```

{
  id: integer;
  credit: number;
  points: integer;
  user: IdentityUser;
  userId: string;
}

```

UpdateCity

```

{
  id: integer;
  name: string;
  perimeterLocation: string;
  createdAt: string;
  user: IdentityUser;
  userId: string;
}

```

UpdateParkingSlot

```
{
  id: integer;
}
```

```

perimeterLocation: string;
createdAt: string;
type: string;
city: City;
cityId: integer;
user: IdentityUser;
userId: string;
}

```

UpdateReport

```
{
  id: integer;
  status: string;
}
```

UpdateRide

```
{
  id: integer;
  startTime: string;
  endTime: string;
  price: number;
  positionStartId: integer;
  positionEndId: integer;
  userId: string;
  vehicleId: integer;
  tripName: string;
}
```

UpdateVehicle

```
{
  id: integer;
  plate: string;
  status: string;
  batteryLevel: number;
  createdAt: string;
  parkingSlot: ParkingSlot;
  parkingSlotId: integer;
  vehicleType: Vehicle Type;
  vehicleTypeId: integer;
  positions: Array<Position>;
}
```

UpdateVehicleType

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```

```
}
```

Vehicle

```
{
  id: integer;
  plate: string;
  status: string;
  batteryLevel: number;
  createdAt: string;
  parkingSlot: ParkingSlot;
  parkingSlotId: integer;
  vehicleType: VehicleType;
  vehicleTypeId: integer;
  positions: Array<Position>;
}
```

VehicleType

```
{
  id: integer;
  model: string;
  type: string;
  pricePerMinute: number;
  createdAt: string;
}
```