



# MobiShare

**Sviluppato da**

Matteo Barbieri

Davide Godino

Vincenzo Russotto

**Progetto di "Progettazione di sistemi software in rete" e  
"Programmazione di applicazioni intelligenti"**

Applicazione cross-platform con sistemi di intelligenza artificiale per il noleggio di  
bici, e-bike e monopattini elettrici.

Dipartimento di Informatica

14 settembre 2025

# Indice

<b>1 Specifica</b>	<b>1</b>
1.1 Requisiti funzionali . . . . .	1
1.2 Utilizzo di A.I. . . . .	6
1.2.1 Riconoscimento facciale . . . . .	6
1.2.2 Formattazione e rating dei feedback . . . . .	8
1.2.3 Pianificazione manutenzioni . . . . .	9
1.2.4 Sistema multi-agente di supporto per utenti, manager e admin .	10
1.3 Requisiti non funzionali . . . . .	13
1.3.1 Architettura a microservizi . . . . .	13
1.3.2 Framework e linguaggi utilizzati . . . . .	14
1.4 Il dominio . . . . .	16
<b>2 Progettazione</b>	<b>18</b>
2.1 Diagramma dei package . . . . .	18
2.1.1 Backend Core . . . . .	19
2.1.2 Backend AI . . . . .	20
2.1.3 Mobile App . . . . .	21
2.2 API REST . . . . .	21
2.2.1 Endpoint Core . . . . .	22
2.2.2 Endpoint AI . . . . .	22
2.3 Topic MQTT . . . . .	22
2.3.1 Valori dei sensori dei mezzi disponibili . . . . .	23

2.3.2	Controllo degli attuatori	24
2.4	Base di dati	25
2.4.1	Database relazionale	25
2.4.2	Database documentale	26
2.5	Deployment	26
2.6	Diagrammi di sequenza	27
A	Diagrammi degli stati	30
Riferimenti		39

# **Capitolo 1**

## **Specific**a

In questo capitolo viene mostrato come abbiamo deciso di modellare le funzionalità del sistema dopo aver effettuato l'analisi della consegna d'esame. Nella prima parte vengono descritte le varie entità coinvolte con il diagramma del dominio, successivamente vengono descritti i requisiti funzionali e le integrazioni con intelligenza artificiale.

### **1.1 Requisiti funzionali**

Per la realizzazione del sistema sono stati individuati i seguenti casi d'uso:

1. Login
2. Registrazione
3. Gestione ricarica credito
4. Verifica disponibilità mezzo
5. Inizio corsa
6. Chiusura corsa
7. Feedback malfunzionamento
8. Assegnamento buono green

9. Addebita credito
10. Ricarica credito
11. Sospensione utente
12. Richiesta riabilitazione
13. Gestione sospensione utente
14. Verifica stato mezzi
15. Destituzione mezzo
16. Inizio manutenzione mezzo
17. Fine manutenzione mezzo
18. Gestione bilanciamento mezzi
19. Consegna mezzo
20. Sgancio mezzo
21. Blocco mezzo
22. Segnalazione stato batteria
23. Modifica stato luce
24. Segnalazione GPS
25. Segnalazione valore anormale
26. Richiesta sgancio blocco
27. Rimuovi gestore
28. Aggiungi gestore
29. Recupero password

30. Cambio password

31. Sposta parcheggio

32. Aggiunta mezzo

In figura 1.1 vengono specificate le varie dipendenze tra i casi d'uso e quali sono gli attori coinvolti.

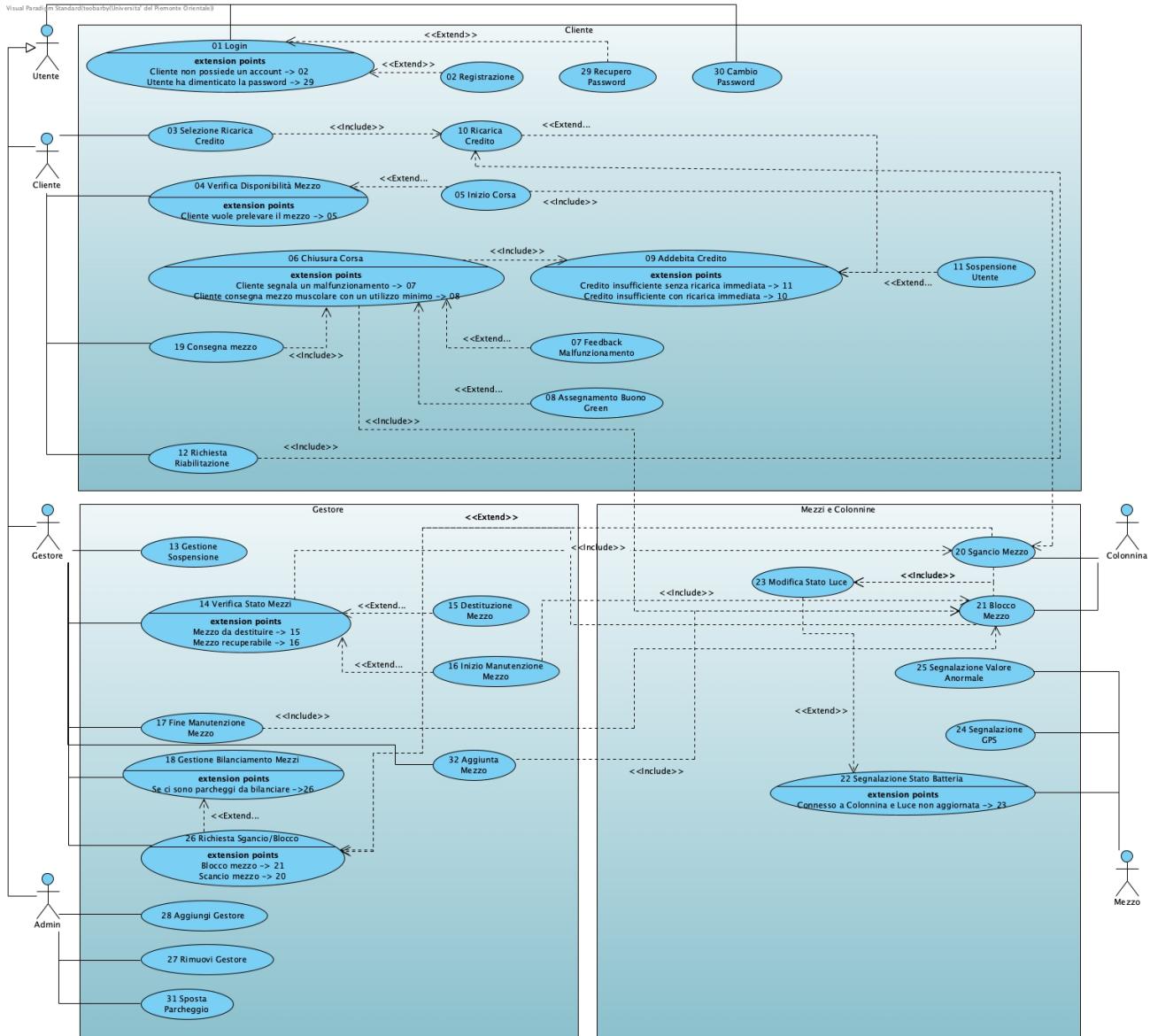


Figura 1.1: Diagramma dei casi d'uso

Ogni caso d'uso è stato approfondito e definito in maniera formale con un diagramma degli stati che potete trovare in [appendice](#).

Data la composizione del nostro gruppo proponiamo di implementare i seguenti componenti accessori (già rappresentati nei casi d'uso):

- un sistema esterno di autenticazione / autorizzazione da utilizzare per permettere l'accesso ai servizi ai soli utenti autorizzati (previa autenticazione).
- la possibilità per il gestore di aggiungere tramite il front-end mezzi al servizio di sharing.
- la possibilità per gli utilizzatori del servizio di fornire un feedback sulla qualità del servizio: questo può avvenire al momento della restituzione di un mezzo.

## 1.2 Utilizzo di A.I.

Per rendere il progetto valido anche per l'esame di "Programmazione di Applicazioni Intelligenti", abbiamo realizzato di tre sistemi di intelligenza artificiale, che si occuperanno di diversi aspetti della piattaforma.

### 1.2.1 Riconoscimento facciale

Si propone un sistema di riconoscimento facciale per l'autenticazione degli utenti, tale sistema è stato realizzato grazie al progetto open-source:

- [face recognition](#)

Il riconoscimento facciale di base si suddivide in 4 step:

- rilevamento del volto
- normalizzazione e allineamento
- codifica dei volti
- identificazione della persona

Di seguito come vengono risolti questi sotto problemi

#### Rilevamento del volto

Il primo passo nella pipeline di riconoscimento facciale è la face detection. Prima di poter distinguere i volti, è necessario localizzarli all'interno dell'immagine.

L'algoritmo utilizzato è l'Histogram of Oriented Gradients (HOG), inventato nel 2005, che funziona attraverso i seguenti passaggi:

1. **Conversione in scala di grigi:** si elimina l'informazione cromatica non necessaria per il rilevamento
2. **Calcolo dei gradienti:** per ogni pixel si determina la direzione in cui l'immagine diventa più scura rispetto ai pixel circostanti

3. **Suddivisione in quadrati  $16 \times 16$ :** l'immagine viene divisa in piccole sezioni e si contano i gradienti per direzione principale
4. **Sostituzione con pattern:** ogni quadrato viene sostituito con le direzioni di gradiente predominanti

Il risultato è una rappresentazione semplificata che cattura la struttura base di un volto, indipendentemente dalla luminosità dell'immagine. Per individuare i volti, si confronta questa rappresentazione HOG con pattern noti estratti da volti di training.

### **Normalizzazione e allineamento**

I volti orientati in direzioni diverse appaiono completamente diversi a un computer. Per risolvere questo problema si utilizza il Face Landmark Estimation, basato sull'algoritmo di Kazemi & Sullivan (2014).

Il processo prevede:

1. **Identificazione di 68 punti di riferimento:** vengono localizzati punti specifici come l'angolo degli occhi, il bordo delle sopracciglia, la punta del naso, ecc.
2. **Trasformazioni affini:** l'immagine viene ruotata, scalata e distorta per centrare occhi e bocca nella stessa posizione relativa
3. **Preservazione delle linee parallele:** si evitano distorsioni 3D che potrebbero alterare significativamente l'immagine

Questo step garantisce che tutti i volti abbiano occhi e bocca in posizioni standardizzate, rendendo molto più accurato il confronto successivo.

### **Codifica dei volti**

Questo è il cuore del sistema di riconoscimento. Invece di confrontare direttamente le immagini (troppo lento per miliardi di foto), si utilizza una Deep Convolutional Neural Network per estrarre 128 misurazioni numeriche per ogni volto.

Il processo di training funziona con:

**1. Triple di immagini:**

- Un volto di una persona nota
- Un altro volto della stessa persona
- Un volto di una persona diversa

**2. Ottimizzazione della rete:** l'algoritmo modifica la rete neurale per rendere più simili le misurazioni delle prime due immagini e più diverse quelle della terza

**3. Generazione di embeddings:** dopo milioni di iterazioni, la rete impara a generare 128 numeri che caratterizzano univocamente ogni volto

Il training richiede circa 24 ore con hardware specializzato, ma esistono reti pre-addestrate come quelle di OpenFace che possono essere utilizzate direttamente.

**Identificazione della persona**

L'ultimo step è il più semplice dal punto di vista algoritmico. Si confrontano le 128 misurazioni del volto sconosciuto con quelle di tutti i volti noti nel database utilizzando un classificatore di machine learning.

Per ulteriori dettagli si rimanda all' articolo [\[2\]](#).

**1.2.2 Formattazione e rating dei feedback**

La piattaforma permette all'utente di lasciare dei feedback relativi allo stato dei veicoli. Tali feedback sono espressi dall'utente in linguaggio naturale e possono contenere informazioni utili per capire lo stato di un parcheggio. Con l'utilizzo di Gemini<sup>[9]</sup>, Large Language Model di casa Google, abbiamo effettuato un rating per capirne la gravità dello stato del veicolo ed una formattazione in lista di punti sintetica per una più rapida analisi da parte del gestore del parcheggio.

Il modello dato un feedback produce un output strutturato di questo tipo:

```
{  
    "id_vehicle": id,  
    "rating": 1...3,  
    "bullet_points": ["Primo difetto", ... , "Ultimo Difetto"]  
}
```

### 1.2.3 Pianificazione manutenzioni

Il potenziale di Gemini è stato sfruttato anche per la generazione di un report sullo stato di un parcheggio, con i veicoli in ordine di priorità da mandare in manutenzione. Come input vengono forniti tutti i feedback precedentemente formattati del parcheggio e viene generato un output strutturato di questo tipo:

```
{  
    "state": 1...3,  
    "summary": sintesi sullo stato del parcheggio,  
    "priorities": [id_mezzo1, ..., id_mezzoN]  
}
```

### 1.2.4 Sistema multi-agente di supporto per utenti, manager e admin

Il backend AI di MobiShare espone, tramite FastAPI, un sistema multi-agente specializzato che supporta tre ruoli distinti: utente finale, manager operativo e amministratore. Gli agenti sono implementati con il framework Agno, utilizzano modelli Gemini di Google per la generazione linguistica e mantengono memoria e sessioni su MongoDB per garantire continuità del contesto. Le loro azioni sono sempre ancorate ai dati e ai processi del backend core (Kotlin/Spring), in modo da evitare risposte speculative e abilitare funzioni operative concrete (ad es. gestione veicoli, manutenzioni, amministrazione utenti), con particolare focus sui parcheggi della piattaforma.

In termini architetturali, la conversazione inizia con una chiamata a uno degli endpoint di chat (`/chat/user`, `/chat/manager`, `/chat/admin`). Il servizio verifica il ruolo delegando al core, istanzia o recupera l'agente associato e lo esegue in asincrono. Ogni agente è istruito a usare i propri strumenti (*tools*) come unica fonte di verità: le guide statiche vengono lette da risorse versionate, mentre i dati e le azioni dinamiche passano attraverso HTTP verso il core. In caso di errore di un tool, l'agente deve dirlo esplicitamente e proporre il passo successivo. Punti chiave dell'architettura:

- Endpoints di chat in `routers/mobi_agent.py` con verifica ruolo (`dependencies.py`) contro il core.
- Agenti Agno per utente, manager e admin, ciascuno con istruzioni, memoria e strumenti dedicati.
- Modello linguistico Gemini configurabile via `GEMINI_MODEL_ID` (default `gemini-2.5-flash-lite`).
- Memoria agentica e sessioni in MongoDB, isolate per ruolo e utente/sessione.
- Toolkits che integrano guide locali e chiamate HTTP verso il core (`BACKEND_CORE`) con risposte tipizzate.

Gli agenti differiscono per scopo, tono e regole di dominio. L'agente *User* assiste l'utente finale nelle attività quotidiane: spiega passo-passo come iniziare una corsa,

gestire il wallet o consultare il profilo, e può recuperare dati personali come storico pagamenti, ricariche, saldo e greenpoints. È bilingue (italiano/inglese), conciso e non inventa informazioni: quando manca un dato, chiede l'identificativo minimo necessario. I suoi strumenti combinano guide statiche (`resources/user_guide/*`) e richieste al core firmate con il token dell'utente. In questo modo, l'esperienza risulta coerente e verificabile, evitando ambiguità.

L'agente *Manager* supporta l'operatività sui parcheggi: aggiungere veicoli, consultare lo stato dei mezzi e le segnalazioni, avviare o chiudere manutenzioni, e ottenere l'elenco delle manutenzioni in corso. Prima di rispondere interroga i servizi del core, mantenendo uno stile sintetico ed esplicitando assunzioni e input richiesti.

L'agente *Admin* estende le capacità del manager al perimetro amministrativo: elencare/creare/eliminare/promuovere manager, consultare e chiudere sospensioni utenti, spostare veicoli tra parcheggi. Le sue istruzioni richiedono risposte concise, dichiarazione degli input mancanti e un'attenzione specifica allo stato “al presente”: una sospensione con data di fine viene considerata nel computo dello stato corrente e sempre inclusa nella reportistica storica quando richiesto. Questa scelta riflette esigenze di controllo e audit tipiche delle funzioni amministrative.

Memoria e flusso applicativo sono progettati per robustezza. La memoria agentica è abilitata e la cronologia viene allegata ai messaggi per dare continuità tra richieste; le collezioni MongoDB sono separate per ruolo (`UserMemory/UserSessions`, `ManagerMemory/ManagerSessions`, `AdminMemory/AdminSessions`). Il flusso standard prevede: autenticazione tramite `Authorization: Bearer <token>`, verifica ruolo sul core, creazione o riuso dell'agente con tool parametrizzati dal token, esecuzione con *Reasoning/Thinking Tools* per strutturare i passaggi interni, ritorno di una risposta testuale pronta per l'interfaccia.

Configurazione essenziale:

- `MONGO_URI` per memoria e sessioni; obbligatoria.
- `BACKEND_CORE` con host:porta del core per tutte le azioni su dati reali (parcheggi, veicoli, utenti).

- GEMINI\_MODEL\_ID e chiave Google AI (GEMINI\_API\_KEY o GOOGLE\_API\_KEY) per il modello.

Sul piano della qualità e della sicurezza, gli agenti sono istruiti a non speculare, a fallire in modo esplicito con messaggi chiari quando un tool non risponde o un payload non è conforme, e a proporre passi successivi pragmatici (ad esempio fornire un identificativo mancante o ripetere un'operazione). L'ancoraggio costante al backend core e l'uso di risorse locali versionate per le guide assicurano che le risposte restino coerenti con lo stato reale del sistema e con le procedure operative sui parcheggi. Il disegno rimane estensibile: è possibile aggiungere nuovi tools per coprire scenari ulteriori, introdurre meccanismi di escalation fra agenti (ad es. da User a Manager) e integrare caching e metriche per migliorare tempi di risposta e osservabilità.

## 1.3 Requisiti non funzionali

La consegna d'esame prevede l'utilizzo del design architetturale a microservizi e l'implementazione di una comunicazione IOT(sensori ed attuatori) utilizzante il protocollo MQTT; ulteriori requisiti come le tecnologie ed i linguaggi utilizzati sono stati scelti da noi avendo come primo obiettivo l'acquisizione di competenze su strumenti non trattati dal corso di Informatica Triennale.

### 1.3.1 Architettura a microservizi

Ogni microservizio è responsabile di una specifica funzionalità del dominio applicativo. Questa suddivisione permette di sviluppare, testare ed effettuare il deploy dei singoli componenti in modo indipendente, facilitando la manutenzione e l'evoluzione del sistema.

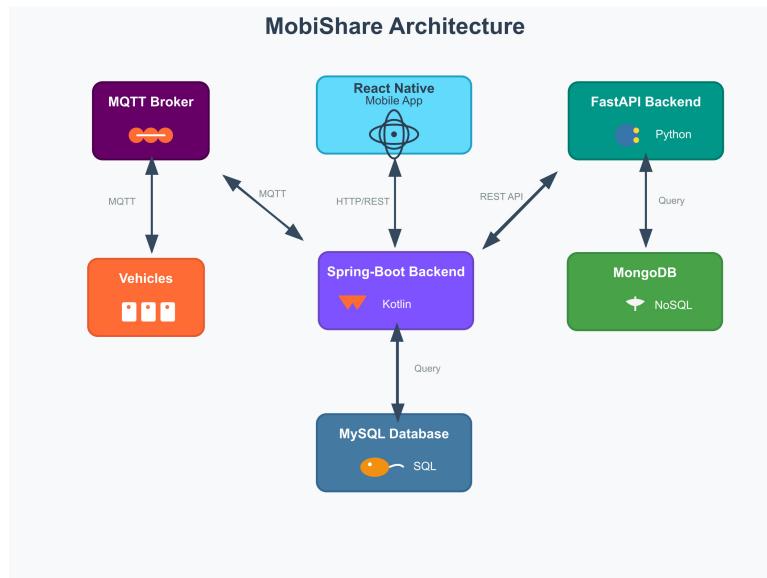


Figura 1.2: Architettura di Mobishare

## MQTT

Per la comunicazione si è scelto di utilizzare RuMQTT[1], un broker scritto in Rust ed opensource.

### 1.3.2 Framework e linguaggi utilizzati

Per la realizzazione del sistema abbiamo optato per l'utilizzo di tre linguaggi di programmazione: **Kotlin**, **Python** e **TypeScript**; utilizzati rispettivamente per lo sviluppo delle principali funzioni degli utenti/gestori, le funzionalità di intelligenza artificiale e lo sviluppo di una mobile app per Android.

#### SpringBoot Backend & MySQL

La scelta di sviluppare il core delle funzionalità in **Kotlin**[5] è motivata dalla familiarità nei linguaggi JVM based e il voler sperimentare un nuovo linguaggio di programmazione. Il framework **SpringBoot**[8] ci ha permesso di integrare con facilità l'interazione con il database **MySQL**[6] ed i livelli di autenticazione ed autorizzazione per l'uso degli endpoint.

## FastApi Backend & MongoDB

Per le funzionalità di AI abbiamo optato per l'utilizzo di **Python**[7] data la popolarità per i progetti di intelligenza artificiale. Per mettere a disposizione le funzionalità abbiamo realizzato un' interfaccia REST grazie a **FastAPI**[10] uno dei maggiori framework per lo sviluppo di API; il funzionamento del framework si sposa con il meccanismo di type hint del linguaggio, permettendo di rinforzare i tipi dei dati che vengono utilizzati. I dati utilizzati da questo backend vengono salvati in un database non relazionale a documenti, **MongoDB**[4]; questo tipo di database si presta molto bene visto il formato a documenti simil JSON, il formato principale in REST.

## React Native Mobile App

Il lato client della piattaforma è una mobile app cross-platform realizzata con React Native, per poter distribuire scrivendone un codice solo sia per IOS che per Android. Non avendo disponibilità di un dispositivo IOS, ci siamo concentrati sullo sviluppo Android; nonostante ciò quanto sviluppato è facilmente estendibile.

## 1.4 Il dominio

Le entità principali individuate sono:

- Cliente
- Gestore
- Parcheggio
- Mezzo
- Sensori

Queste cinque entità sono gli oggetti di interesse e la loro interazione, illustrata nel diagramma delle classi in figura 1.3 con maggiore dettaglio, permette di realizzare le specifiche della consegna.

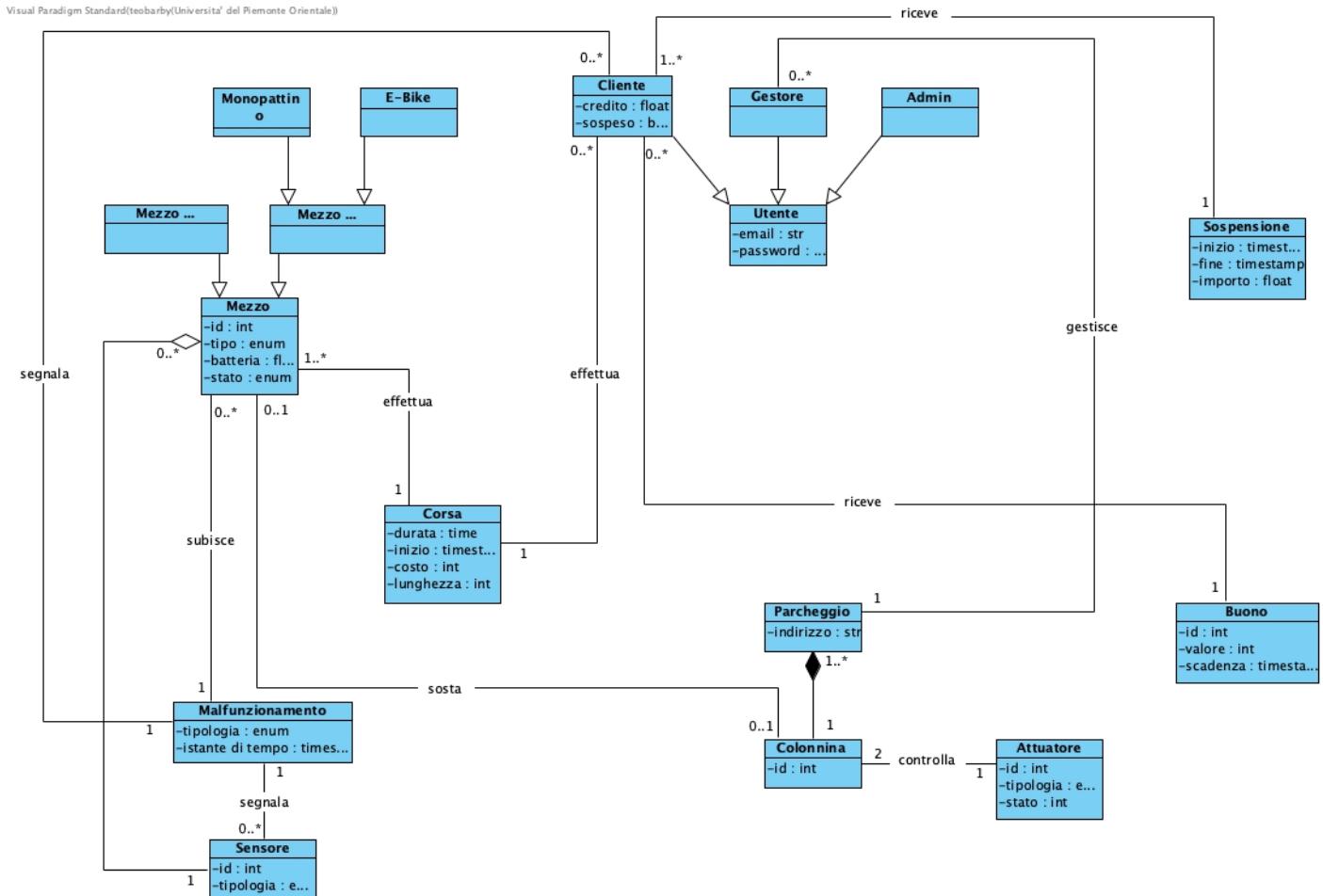


Figura 1.3: Diagramma delle classi del dominio

# **Capitolo 2**

## **Progettazione**

In questo capitolo viene presentata la progettazione del sistema di MobiShare, illustrando i diagrammi dei package dei singoli componenti, la descrizione degli endpoint dei backend, i topic MQTT, l'organizzazione delle basi di dati, il deployment ed infine due esempi di interazioni complesse tra i vari componenti.

### **2.1 Diagramma dei package**

Il diagramma dei package illustra la struttura modulare del sistema e le dipendenze tra i diversi componenti. Ogni package rappresenta un microservizio con le sue responsabilità specifiche.

## 2.1.1 Backend Core

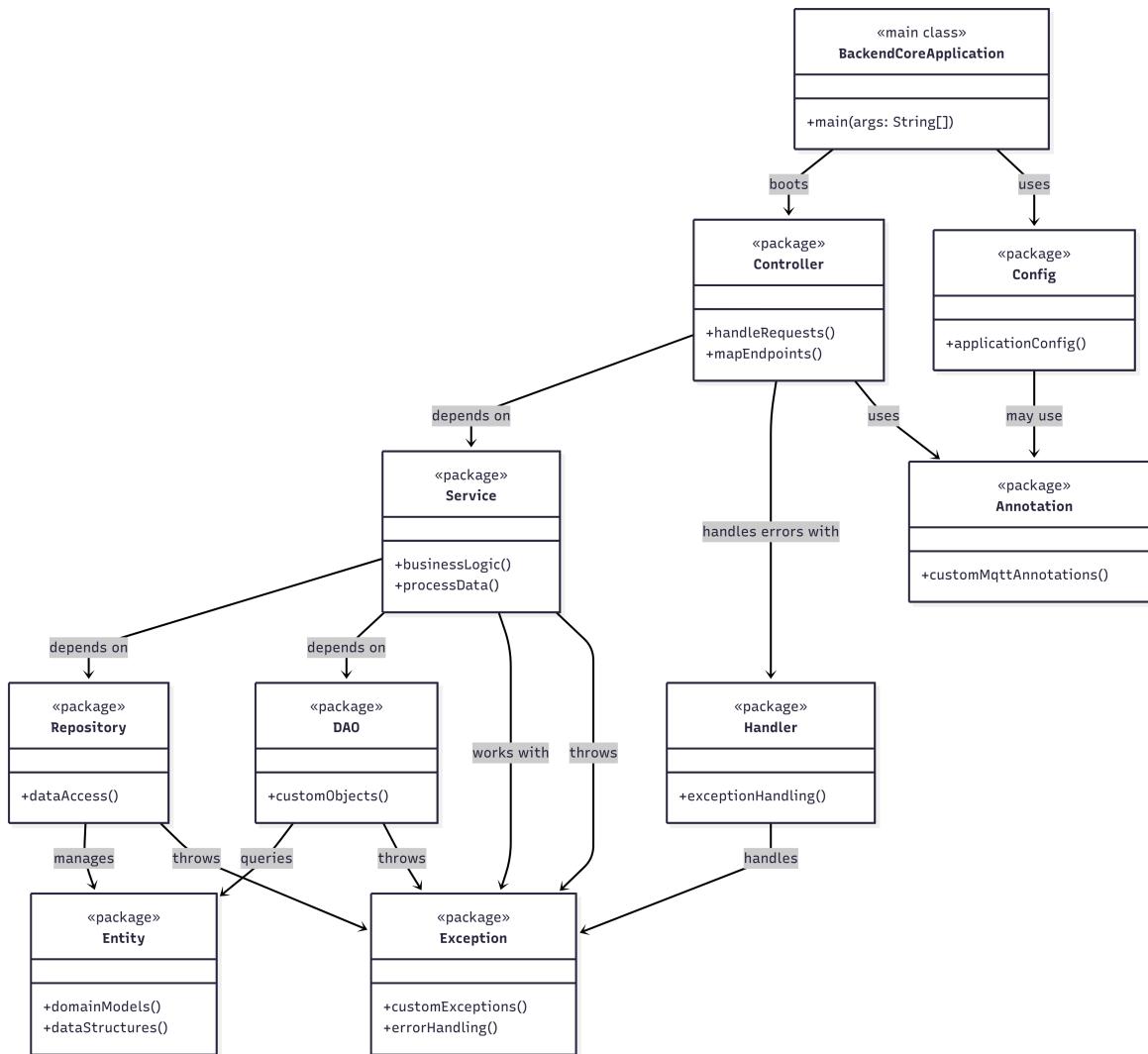


Figura 2.1: Diagramma dei package del Backend Core

## 2.1.2 Backend AI

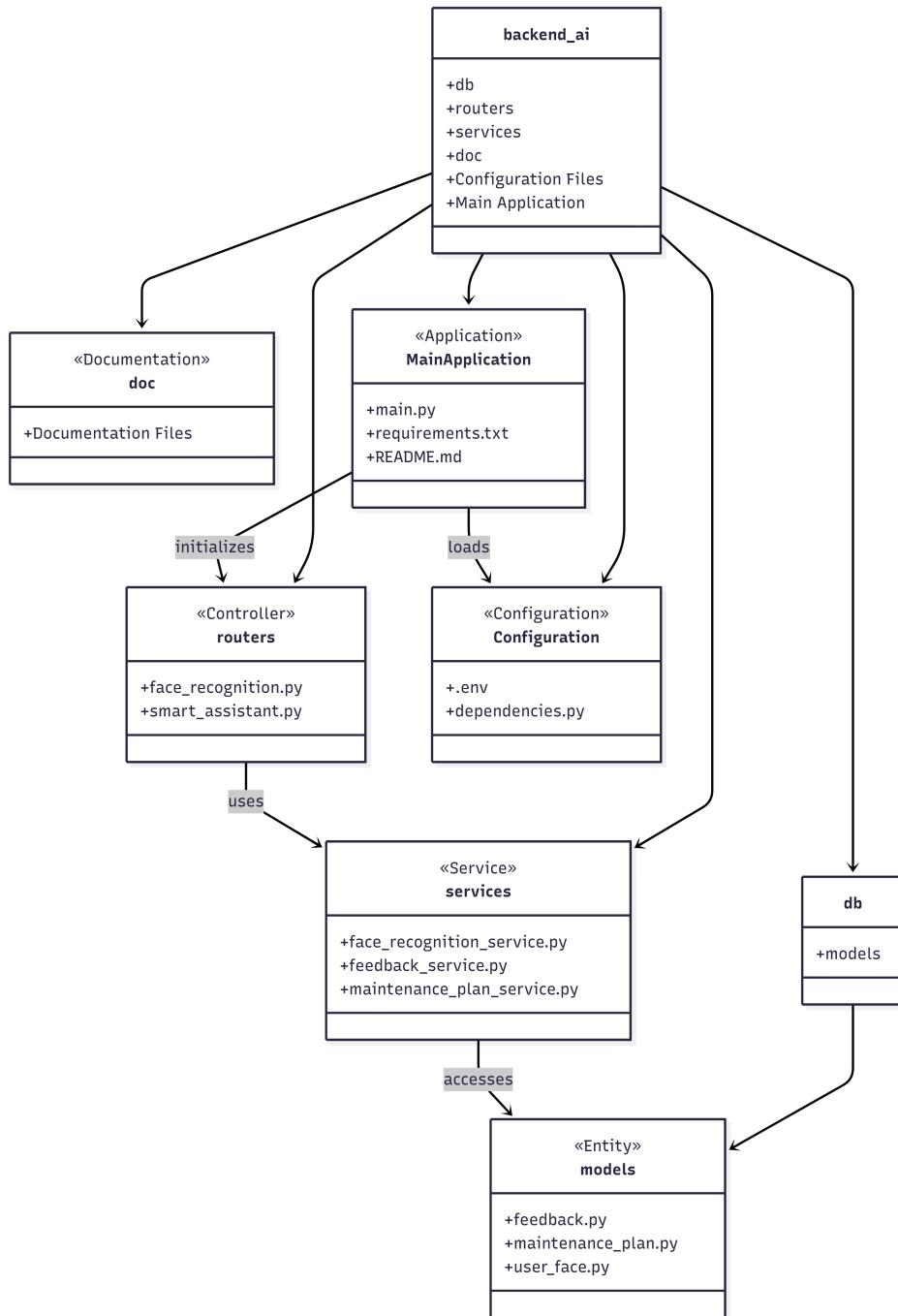


Figura 2.2: Diagramma dei package del Backend AI

### 2.1.3 Mobile App

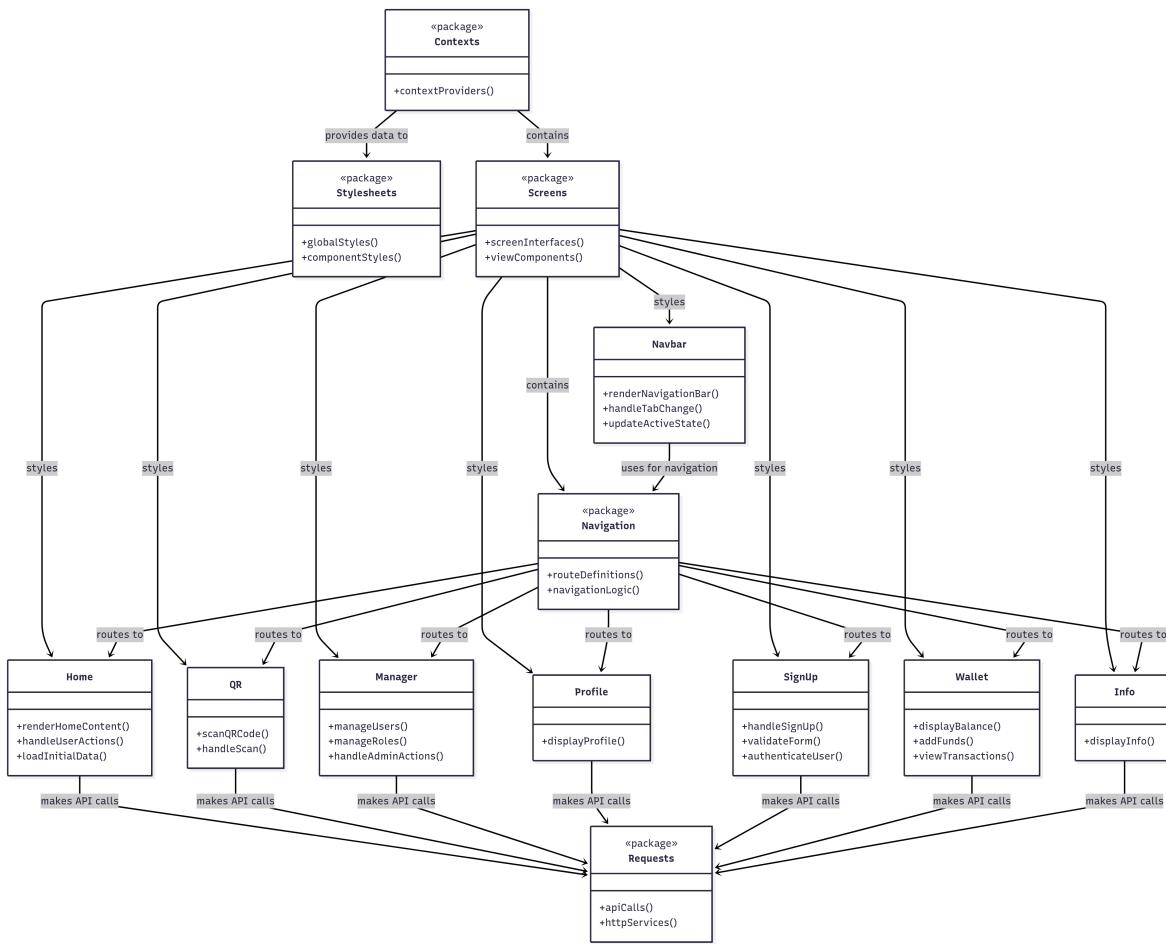


Figura 2.3: Diagramma dei package della Mobile App

## 2.2 API REST

Le API REST rappresentano il principale meccanismo di comunicazione asincrona tra i microservizi e con i client esterni. Ogni servizio espone endpoint specifici seguendo i principi REST e utilizzando i metodi HTTP appropriati.

Tutte le API implementano autenticazione Bearer Token per garantire la sicurezza delle operazioni e utilizzano codici di stato HTTP standard per comunicare l'esito delle richieste.

La documentazione dettagliata di ogni endpoint la si può trovare nella repository del progetto in formato **.yaml**

### 2.2.1 Endpoint Core

Gli endpoint messi a disposizione sono stati raccolti nelle seguenti categorie:

- suspension
- manager
- vehicle
- race
- park
- customer
- auth

### 2.2.2 Endpoint AI

Gli endpoint messi a disposizione sono stati raccolti nelle seguenti categorie:

- face recognition
- smart assistant

## 2.3 Topic MQTT

Di seguito i topic ed il formato dei messaggi per permettere la comunicazione tra i sensori ed il controllo degli attuatori.

### 2.3.1 Valori dei sensori dei mezzi disponibili

I mezzi sono dotati di sensori per la batteria, pressione delle gomme e geolocalizzazione; la scelta di avere tre topic diversi nonostante lo stesso formato di messaggio è stata presa per poter differenziare tra le eventuali logiche di gestione dei dati in arrivo.

Topic	Publisher	Subscriber
vehicle/battery	mezzo	SpringBoot Backend
vehicle/pressure	mezzo	SpringBoot Backend
vehicle/position	mezzo	SpringBoot Backend

Il client Spring Boot backend si iscrive per semplicità al topic **vehicle/+**, utilizzando la wildcard **+**. Il formato dei messaggi scambiati è il seguente:

```
{
    "sensorId": int,
    "value": float,
    "time": "2024-07-14 h:m:s"
}
```

### 2.3.2 Controllo degli attuatori

Per ogni dock presente nella piattaforma, vengono creati i seguenti topic dove **parkId** e **dockId** fanno da placeholder ai valori effettivi degli id dei parcheggi e delle dock.

Topic	Publisher	Subscriber
parks/ <b>parkId</b> /docks/ <b>dockId</b> /in	dock	SpringBoot Backend
parks/ <b>parkId</b> /docks/ <b>dockId</b> /out	SpringBoot Backend	dock

Il messaggio scambiato nel primo topic, è semplicemente la comunicazione dell'id del mezzo:

```
{
    "vehicleId": int,
}
```

Nel secondo topic invece, il backend imposta l'apertura delle ganasce e l'accensione delle luci:

```
{
    "actuator": tipo,
    "operation": operazione
}
```

## 2.4 Base di dati

Per le due basi di dati utilizzati vengono forniti rispettivamente il modello ER e l'insieme delle collezioni con il loro schema.

### 2.4.1 Database relazionale

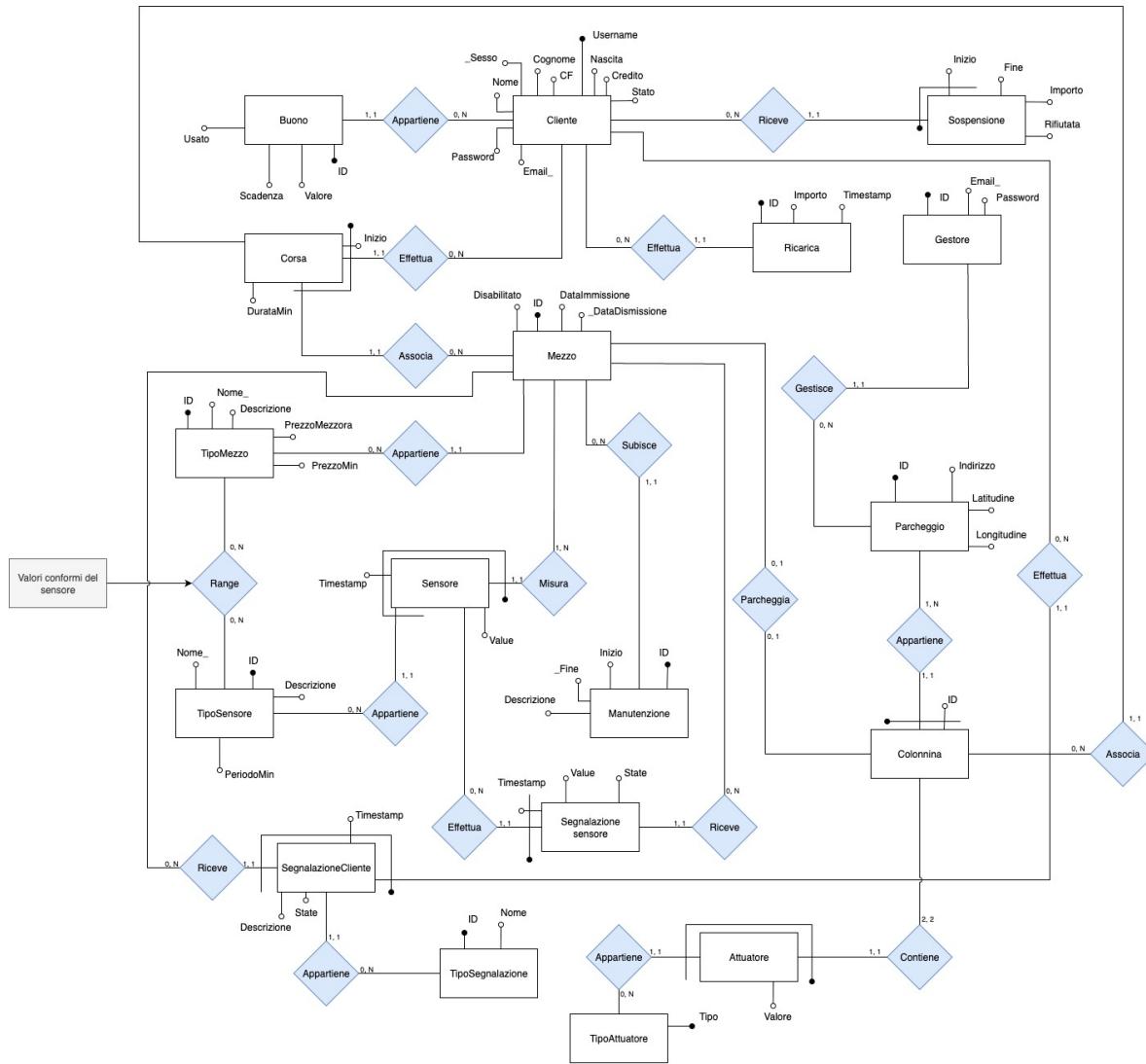


Figura 2.4: Schema ER del database di MobiShare

## 2.4.2 Database documentale

All'interno del database documentale sono presenti due collezioni:

- Face
- Feedback

La prima è utilizzata per salvare il mapping tra l'utente e l'encoding del suo volto ed ha il seguente schema:

```
{
    "username": username,
    "encoding": list[float]
}
```

La seconda contiene i feedback formattati dei veicoli:

```
{
    "id_vehicle": id,
    "rating": 1...3,
    "bullet_points": list[string]
}
```

## 2.5 Deployment

Per permettere il test dell'applicazione su una singola macchina, è stato creato un ulteriore branch con la containerizzazione delle parti coinvolte utilizzando **Docker**[\[3\]](#). Nella versione containerizzata abbiamo dovuto effettuare delle modifiche alle versioni dei software utilizzati poichè non tutte le immagini presenti erano compatibili; Per questo motivo vi sono due branch. In sede di discussione del progetto, il sistema in funzione verrà mostrato su più dispositivi in una stessa rete locale.

## 2.6 Diagrammi di sequenza

Per mostrare l'interazione tra tutti i componenti di seguito illustriamo i diagrammi di sequenza di due delle operazioni più complesse:

- l'inizio di una nuova corsa
- l'invio di un feedback relativo al mezzo

La complessità di queste operazioni è data da i componenti coinvolti, nel primo caso il sotto-sistema IOT e nel secondo il backend AI:

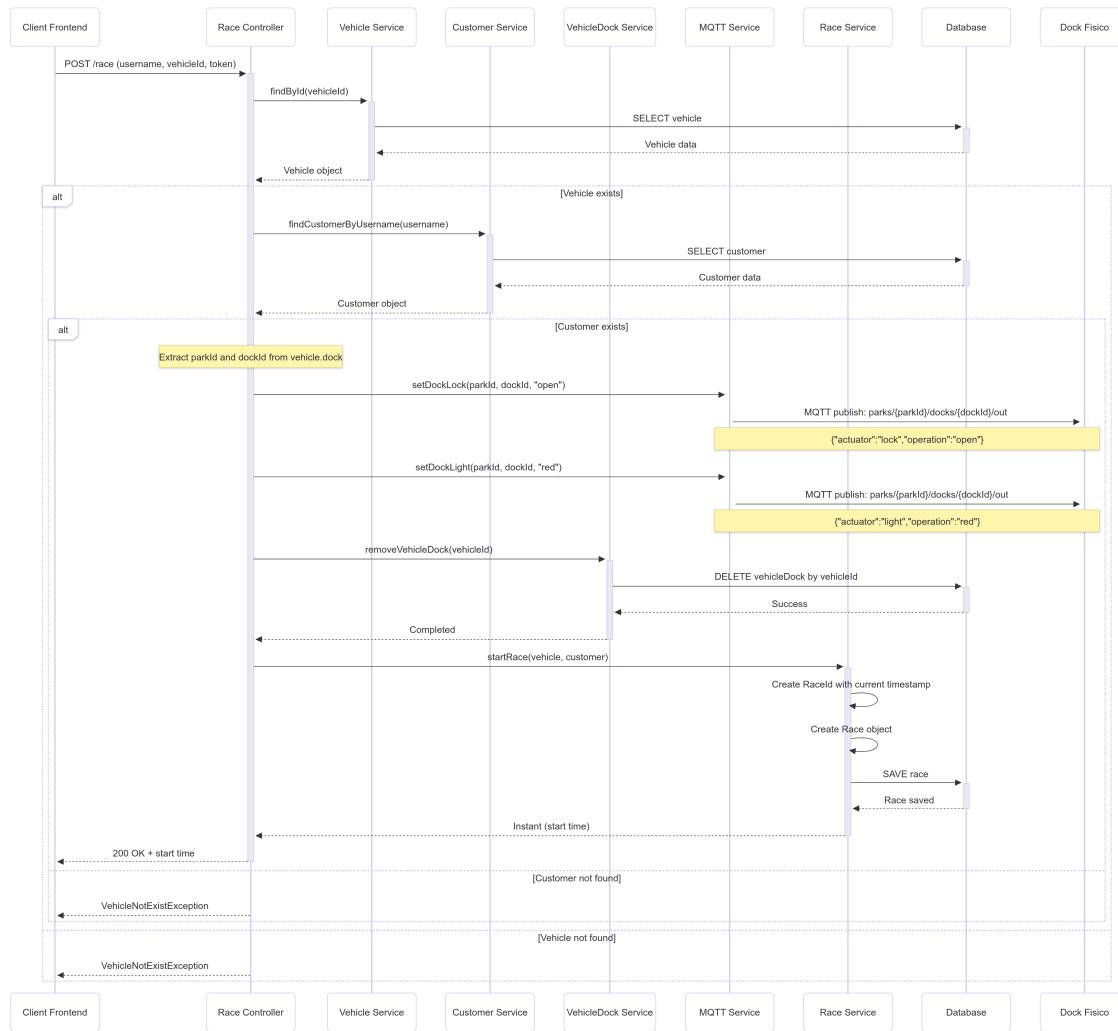


Figura 2.5: Inizio di una corsa

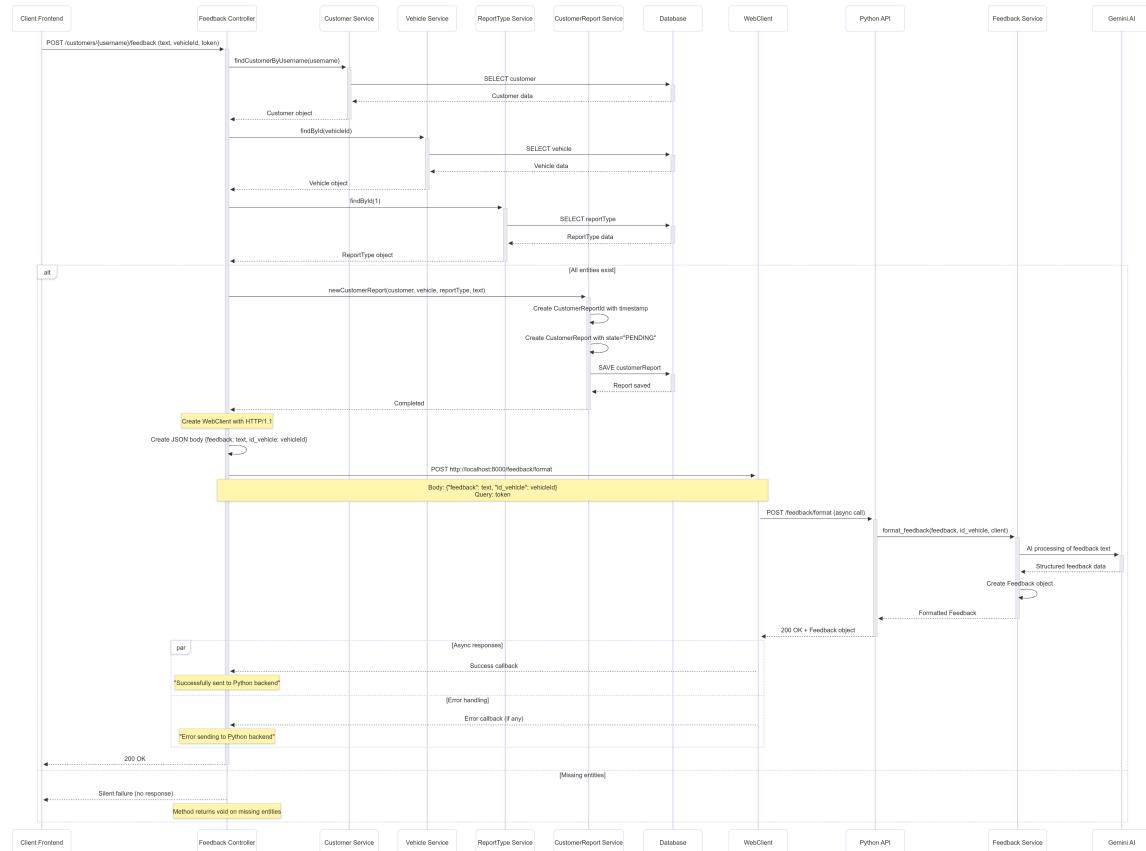


Figura 2.6: Invio di un feedback

# Appendice A

## Diagrammi degli stati

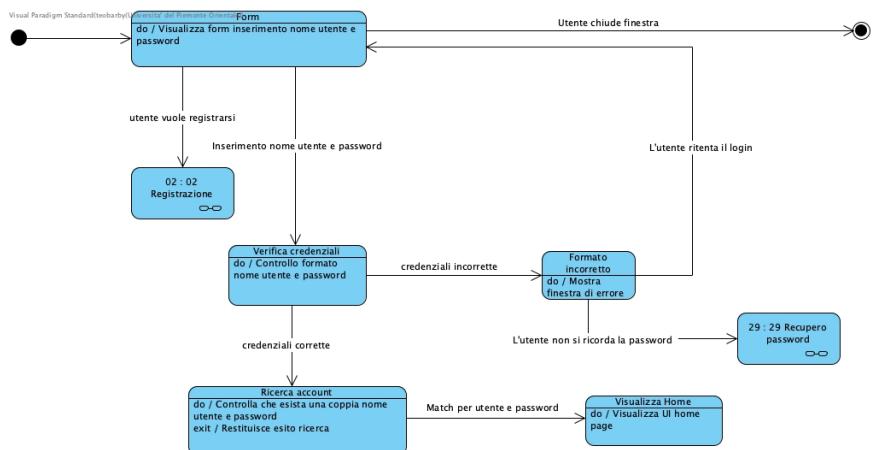


Figura A.1: 01 Login

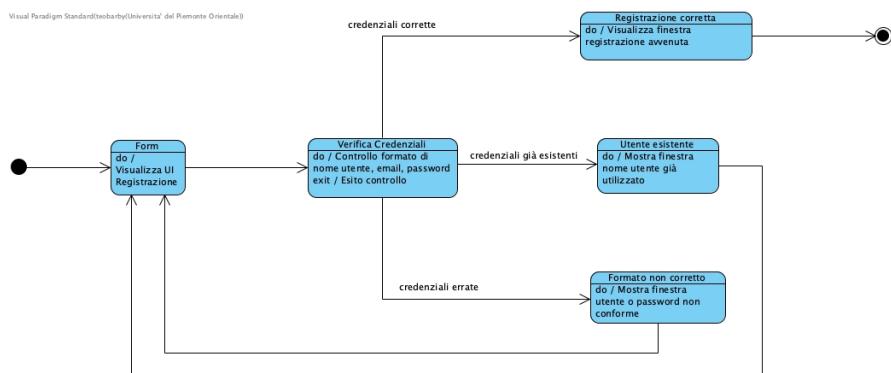


Figura A.2: 02 Registrazione

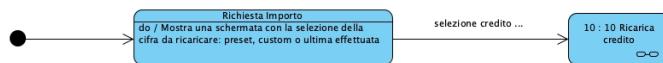
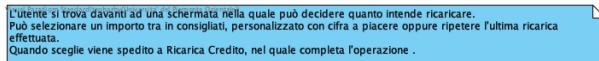


Figura A.3: 03 Gestione ricarica credito

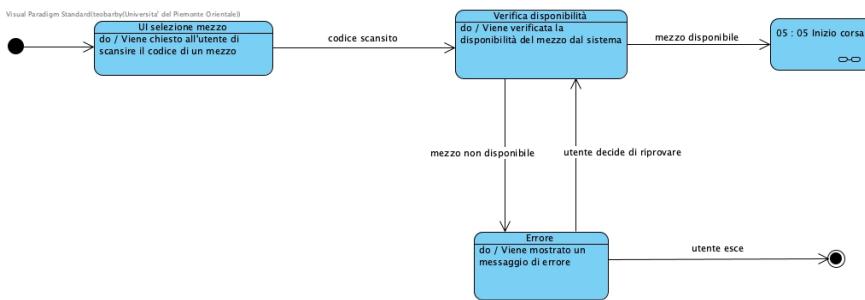


Figura A.4: 04 Verifica disponibilità mezzo

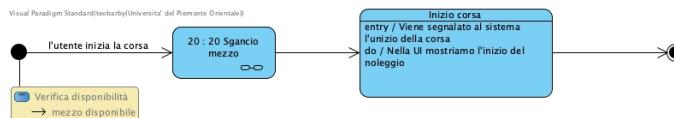


Figura A.5: 05 Inizio corsa

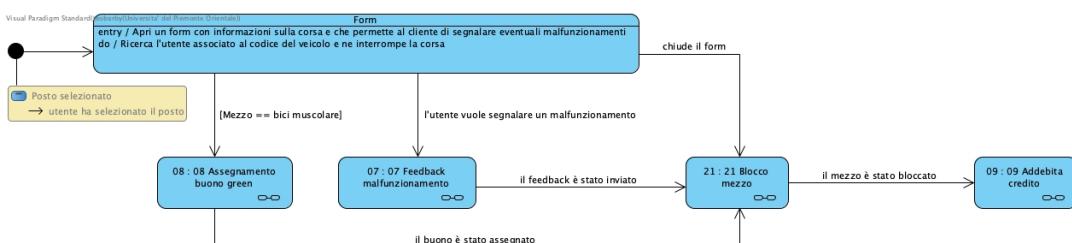


Figura A.6: 06 Chiusura corsa

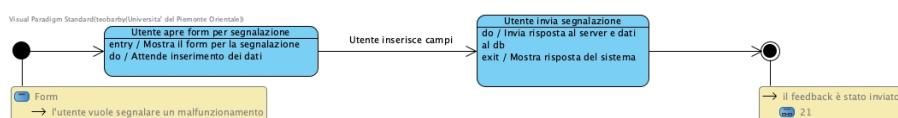


Figura A.7: 07 Feedback malfunzionamento

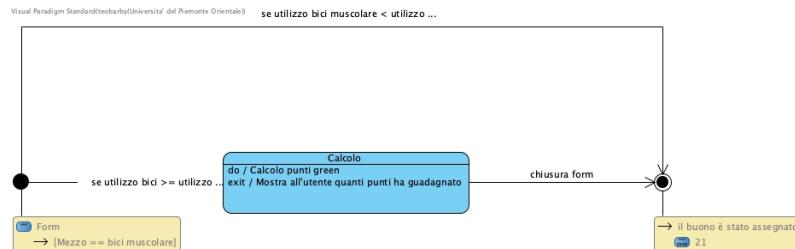


Figura A.8: 08 Assegnamento buono green

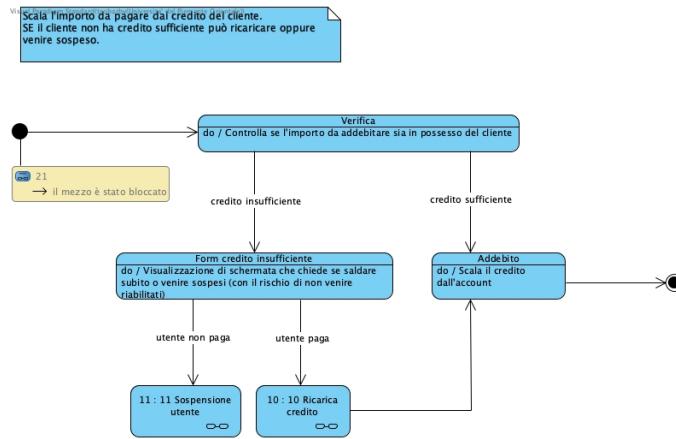


Figura A.9: 09 Addebita credito

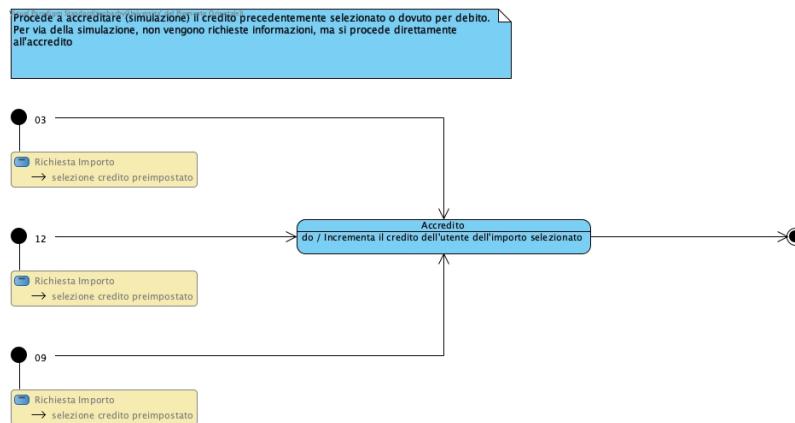


Figura A.10: 10 Ricarica credito

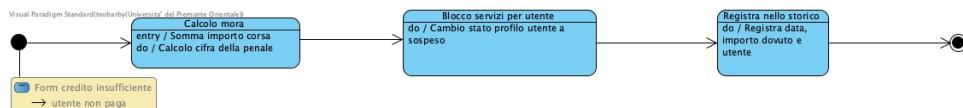


Figura A.11: 11 Sospensione utente

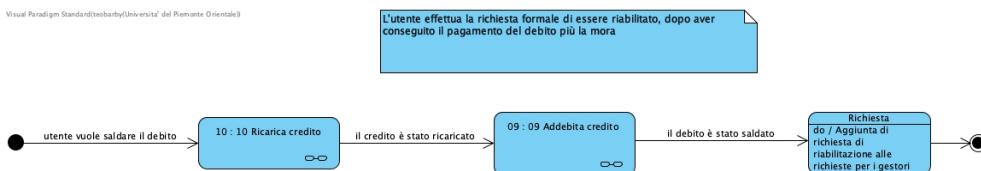


Figura A.12: 12 Richiesta riabilitazione

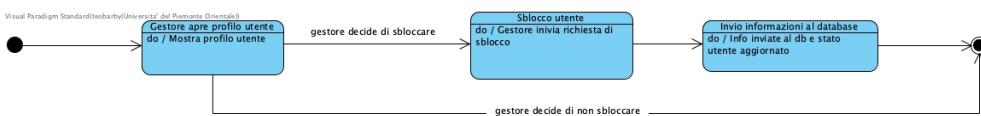


Figura A.13: 13 Gestione sospensione utente

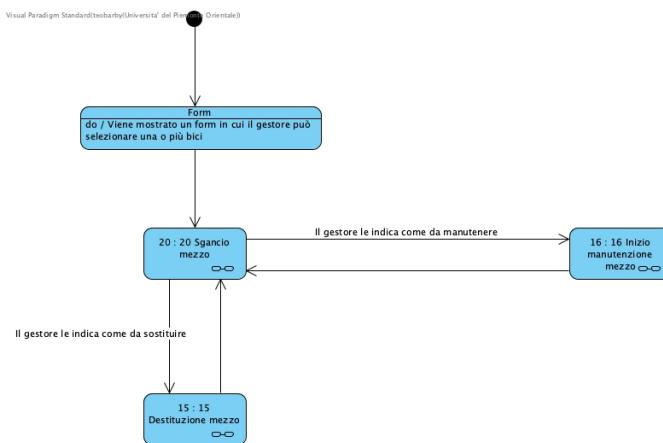


Figura A.14: 14 Verifica stato mezzi

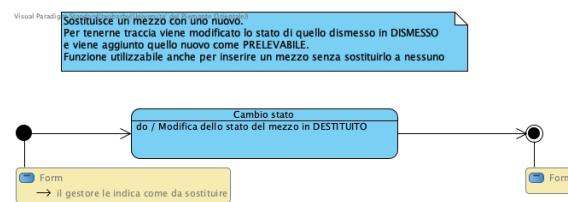


Figura A.15: 15 Destituzione mezzo

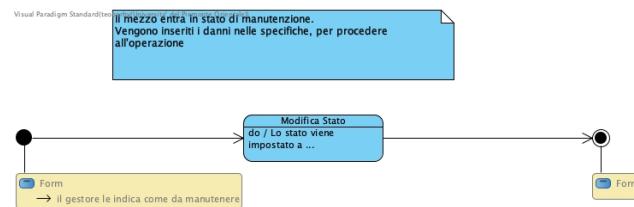


Figura A.16: 16 Inizio manutenzione mezzo

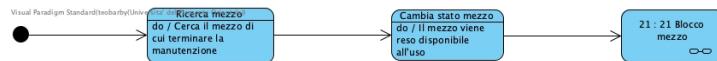


Figura A.17: 17 Fine manutenzione mezzo

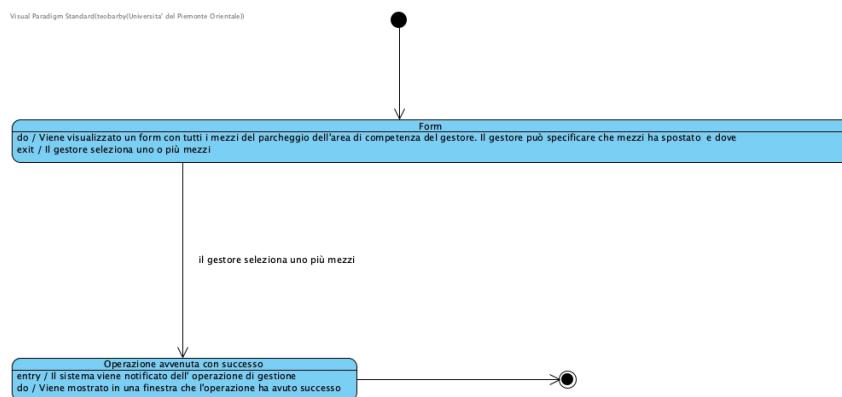


Figura A.18: 18 Gestione bilanciamento mezzi

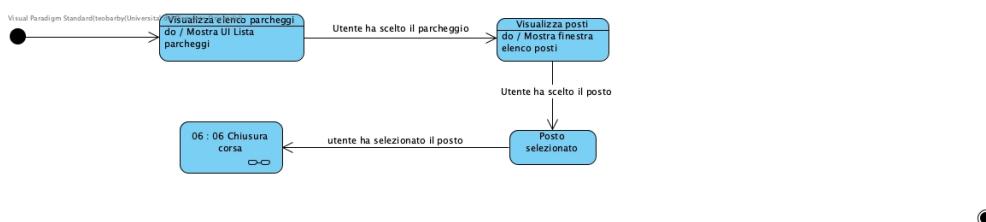


Figura A.19: 19 Consegna mezzo

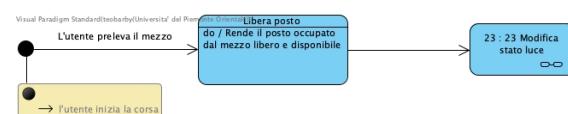


Figura A.20: 20 Sgancio mezzo

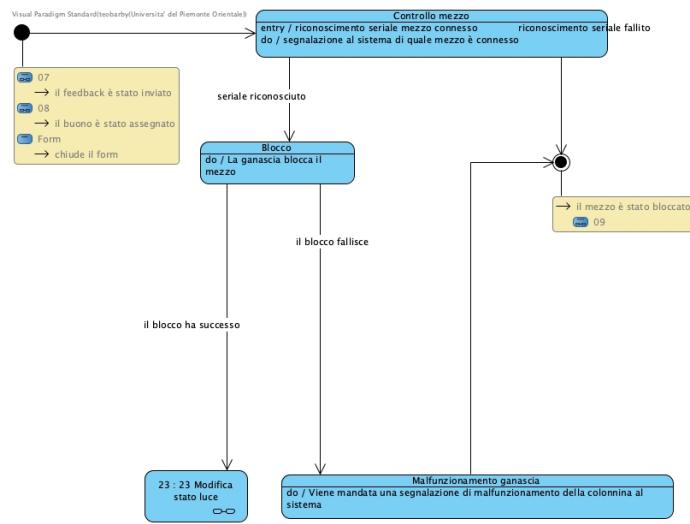


Figura A.21: 21 Blocco mezzo



Figura A.22: 22 Segnalazione stato batteria

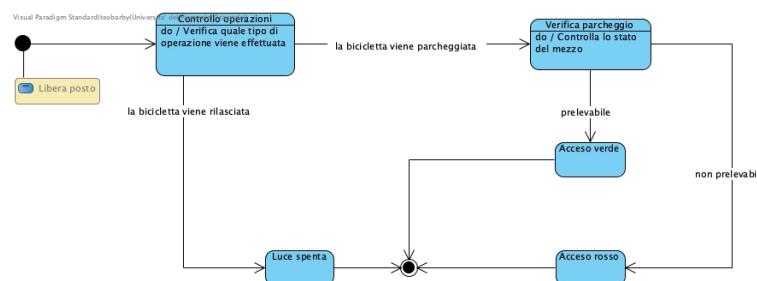


Figura A.23: 23 Modifica stato luce

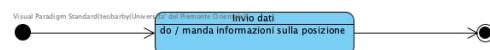


Figura A.24: 24 Segnalazione GPS

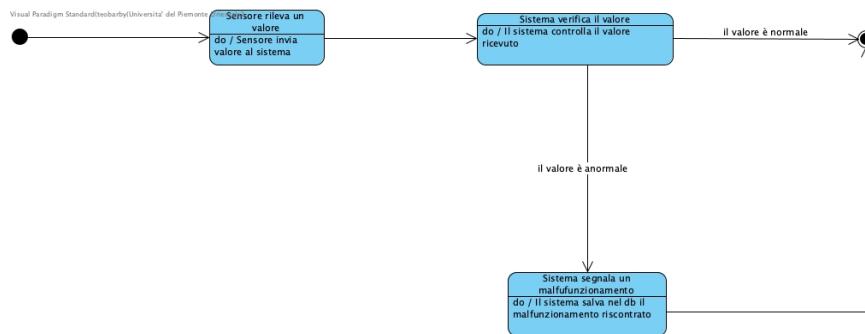


Figura A.25: 25 Segnalazione valore anormale

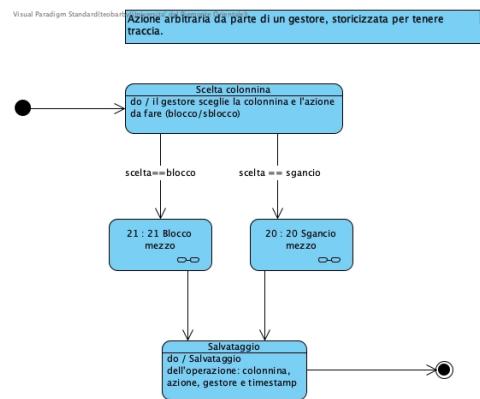


Figura A.26: 26 Richiesta sgancio blocco

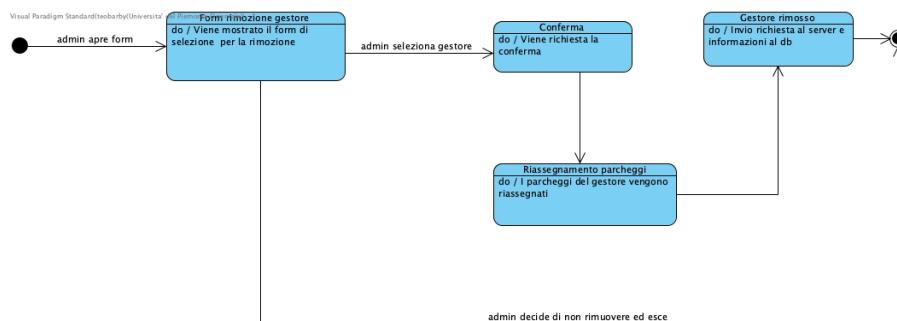


Figura A.27: 27 Rimuovi gestore

## APPENDICE A. DIAGRAMMI DEGLI STATI

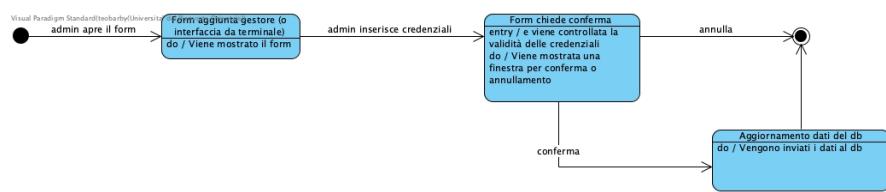


Figura A.28: 28 Aggiungi gestore

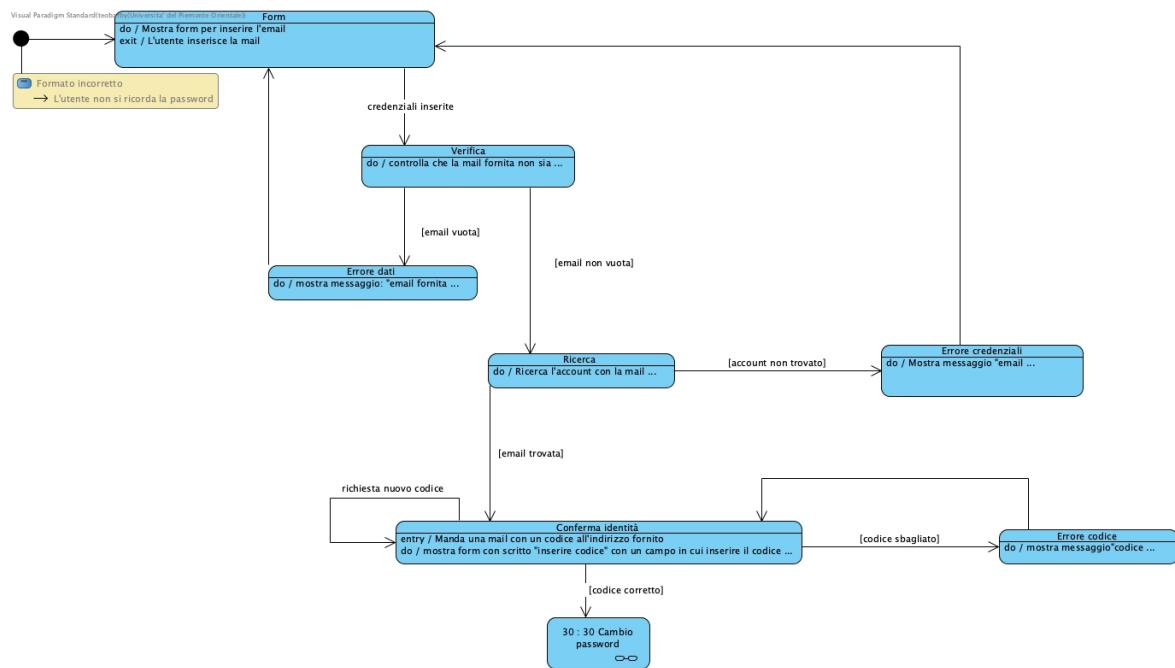


Figura A.29: 29 Recupero password

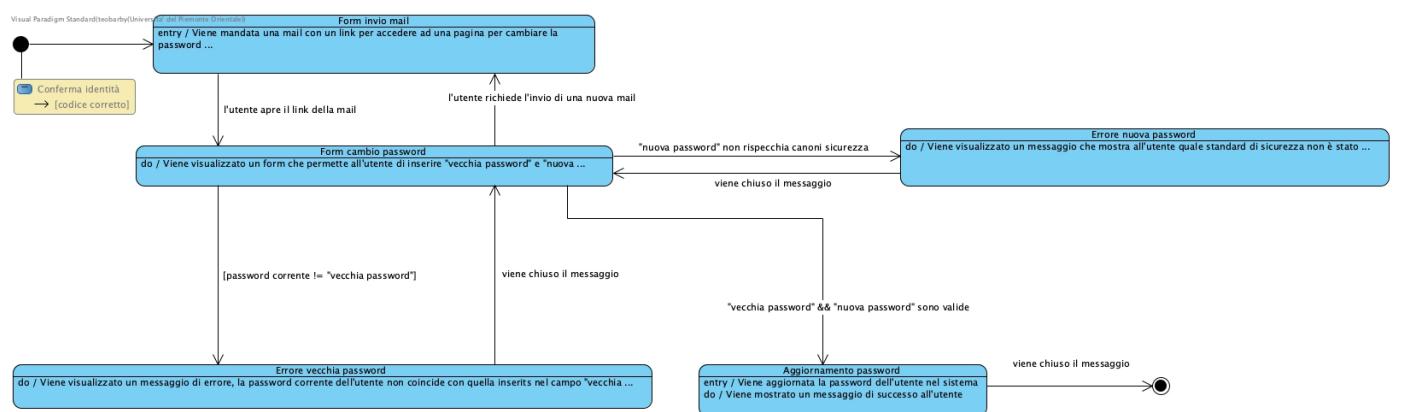


Figura A.30: 30 Cambio password

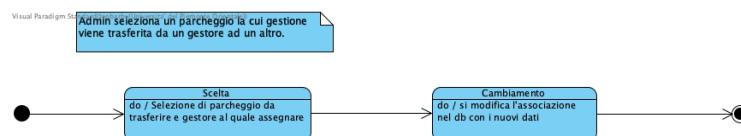


Figura A.31: 31 Sposta parcheggio



Figura A.32: 32 Aggiunta mezzo

# Riferimenti

- [1] bytebeamio. *The mqtt ecosystem in Rust*. URL: <https://github.com/bytebeamio/rumqtt>.
- [2] Adam Geitgey. *Modern Face Recognition with Deep Learning*. 2016. URL: <https://medium.com/@ageitgey/machine-learning-is-fun-part-4-modern-face-recognition-with-deep-learning-c3cffc121d78>.
- [3] Docker Inc. *Docker containers*. URL: <https://www.docker.com/>.
- [4] MongoDB Inc. *MongoDB document based database*. URL: <https://www.mongodb.com/>.
- [5] JetBrains. *Kotlin Programming Language*. URL: <https://kotlinlang.org/>.
- [6] Oracle. *MySQL relational database*. URL: <https://www.mysql.com/>.
- [7] Guido Von Rossum. *Python programming language*. URL: <https://www.python.org/>.
- [8] VMware Tanzu. *Spring Boot framework*. URL: <https://spring.io/projects/spring-boot>.
- [9] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2025. arXiv: [2312.11805 \[cs.CL\]](https://arxiv.org/abs/2312.11805). URL: <https://arxiv.org/abs/2312.11805>.
- [10] tiangolo. *FastAPI web framework*. URL: <https://fastapi.tiangolo.com/>.