



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE  
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

## Carsharing Management Between University Campuses

*Autori*

Tommaso Bellucci  
Edoardo Rustichini  
Tiziano Talini

*N° Matricola*

7072924  
7076614  
7079207

*Corso Principale*

Ingegneria del Software

*Docente corso*

Enrico Vicario

*Anno Accademico*

2024/2025

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Statement . . . . .	3
1.2	Architettura e pratiche utilizzate . . . . .	3
<b>2</b>	<b>Analisi dei requisiti</b>	<b>5</b>
2.1	Use Case Diagrams . . . . .	5
2.2	Use Case Template . . . . .	7
2.3	Page Navigation Diagram . . . . .	9
2.4	Mockup . . . . .	10
<b>3</b>	<b>Class Diagram</b>	<b>12</b>
3.1	Domain Model . . . . .	12
3.2	ORM . . . . .	14
3.3	Business Logic . . . . .	15
3.4	Diagramma ER e schema relazionale . . . . .	16
<b>4</b>	<b>Implementazione</b>	<b>18</b>
4.1	Struttura cartelle . . . . .	18
4.2	Implementazione Database . . . . .	18
4.2.1	Dati di Default . . . . .	19
4.3	Domain Model . . . . .	19
4.3.1	User . . . . .	20
4.3.2	Vehicle e Location . . . . .	20
4.3.3	Trip . . . . .	20
4.3.4	Booking . . . . .	21
4.4	Object-Relational Mapping . . . . .	21
4.4.1	ConnectionManager . . . . .	21
4.4.2	UserDAO . . . . .	22
4.4.3	TripDAO . . . . .	23
4.4.4	BookingDAO . . . . .	23
4.4.5	VehicleDAO e LocationDAO . . . . .	23
4.5	Business Logic . . . . .	23
4.5.1	AuthController . . . . .	23
4.5.2	TripController . . . . .	23
4.5.3	BookingController . . . . .	24
4.5.4	UserController . . . . .	24
4.5.5	AdminController . . . . .	24
4.5.6	VehicleController e LocationController . . . . .	25
<b>5</b>	<b>Testing</b>	<b>26</b>
5.1	Test riguardanti casi d'uso . . . . .	26
5.2	Elenco unit test implementati . . . . .	27

<b>6</b>	<b>Note</b>	<b>30</b>
6.1	Utilizzo di AI per lo sviluppo . . . . .	30

# 1 Introduzione

Questo elaborato rappresenta la relazione del progetto svolto per l'esame di Ingegneria del Software del corso di Laurea Triennale in Ingegneria Informatica dell'Università degli Studi di Firenze. Il codice sorgente del progetto è disponibile su *GitHub* al seguente indirizzo: SWE-Project.

## 1.1 Statement

Questo progetto si propone di sviluppare un'applicazione per la gestione e l'utilizzo di servizi di carsharing universitario: l'università mette a disposizione delle navette che possono essere usate dagli studenti per muoversi tra sedi universitarie in modo efficiente e sostenibile.

L'applicazione sarà usata dagli studenti per la prenotazione di viaggi in qualità di passeggeri o guidatori, e dagli amministratori per la gestione delle risorse. Il sistema deve permettere la registrazione e autenticazione di studenti o admin all'applicativo, e la possibilità di prenotare e visualizzare i viaggi. Il progetto trae ispirazione da TUSS (The Ultimate Sharing Service) dell'Università di Firenze, un servizio sviluppato per offrire una soluzione concreta alle difficoltà di spostamento tra le diverse sedi.

## 1.2 Architettura e pratiche utilizzate

Il software è stato sviluppato in Java, con il supporto di un database PostgreSQL per la memorizzazione e la gestione persistente dei dati. Per la connessione tra l'applicazione e il database è stata utilizzata la libreria JDBC (Java DataBase Connectivity).

Per garantire una chiara separazione delle responsabilità l'architettura del progetto è stata suddivisa in tre macro-componenti principali: Domain Model, Business Logic e ORM. Ciascun package ha un ruolo ben definito:

- **Domain Model:** definisce le entità principali del dominio dell'applicazione
- **ORM:** include le classi responsabili del mapping tra oggetti Java e tabelle del database, permettendo la lettura e scrittura dei dati in modo astratto
- **Business Logic:** comprende le classi che gestiscono le regole di funzionamento dell'applicazione, coordinando le operazioni tra il dominio e la persistenza dei dati, come la prenotazione dei viaggi e la verifica dei permessi. I diagrammi degli Use Case e quelli di Class Diagram sono stati realizzati seguendo lo standard UML (Unified Modeling Language).  
Di seguito l'elenco degli strumenti e delle piattaforme utilizzati durante lo sviluppo:

- **IntelliJ IDEA:** IDE utilizzato per lo sviluppo in Java
- **PgAdmin:** interfaccia grafica per la gestione di database PostgreSQL

- **GitHub**: piattaforma per il versionamento e condivisione del codice sorgente
- **draw.io**: utilizzato per la realizzazione del navigation diagram e dei diagrammi UML
- **Figma**: software per la creazione di mock-up
- **Overleaf.com**: piattaforma per la stesura collaborativa del report in  $\text{\LaTeX}$

## 2 Analisi dei requisiti

Come descritto nell'introduzione, abbiamo progettato un applicativo per la gestione e organizzazione dell'utilizzo di navette fornite dall'università per dare la possibilità a studenti di fare spostamenti tra quelle sedi universitarie che non sono dotate di un trasporto pubblico adatto. Abbiamo individuato come attori principali i seguenti:

- **Studente:** utente a cui è rivolto principalmente l'applicativo;
- **Admin:** utente autorizzato alla gestione delle risorse e degli utenti

Gli studenti avranno la possibilità di creare un profilo, nel quale dovranno inserire i propri dati, e successivamente potranno aggiungersi a viaggi già esistenti o crearne nuovi, indicando data, orario, sede di partenza e sede di arrivo; devono poter modificare o cancellare le proprie prenotazioni, e le proprie credenziali.

L'admin invece deve avere totale controllo delle risorse disponibili, potendo aggiungere o eliminare navette in caso di guasti o di nuove disponibilità, e cancellare o modificare i viaggi. Deve avere anche la possibilità di gestire i profili utenti, revocare la possibilità di guida o rimuovere profili.

### 2.1 Use Case Diagrams

I seguenti sono gli Use Case diagrams trovati, contenendo i casi d'uso più importanti riguardanti gli attori *Student* (Vedi fig 1), e *Admin* (vedi fig.2):

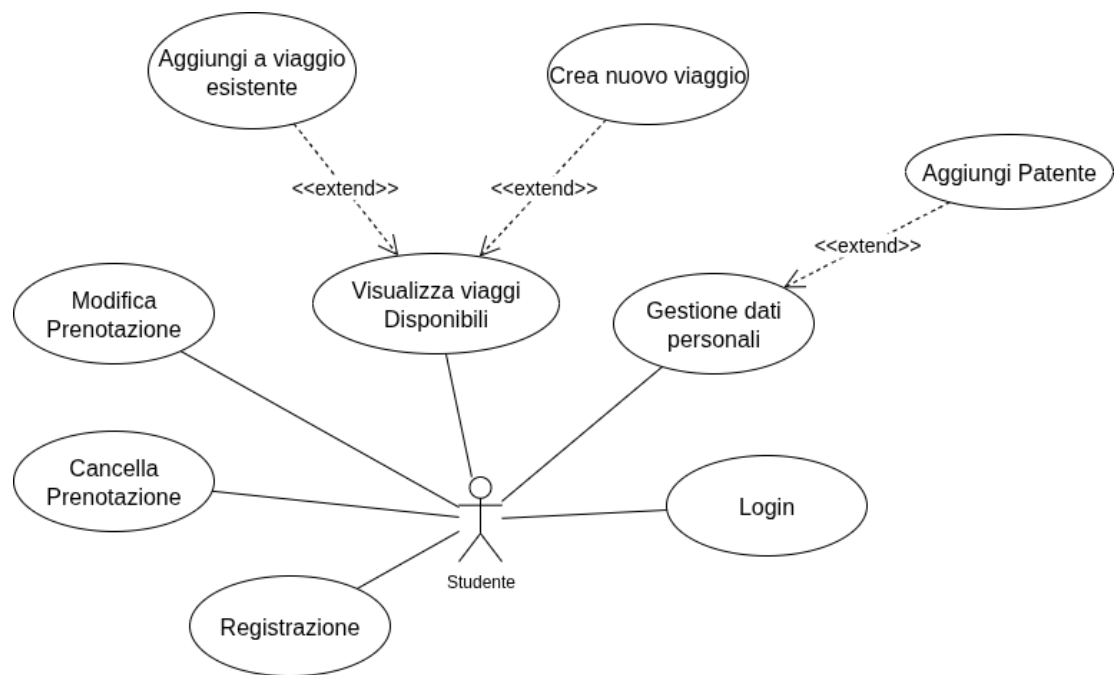


Figura 1: Use Case diagram per Studente

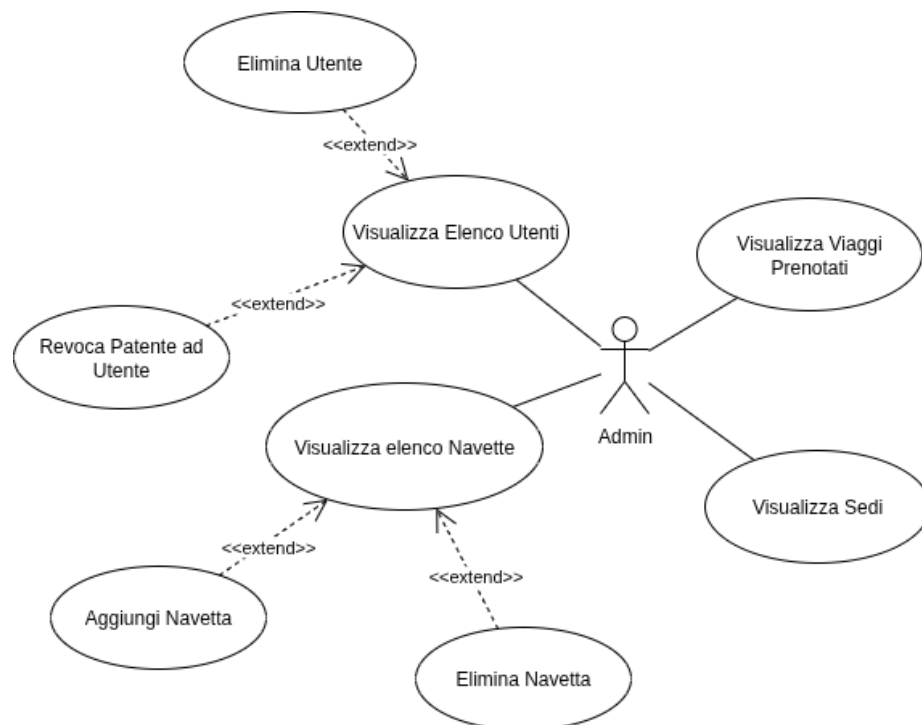


Figura 2: Use Case diagram per Admin

## 2.2 Use Case Template

Di seguito sono riportati gli use case template per i principali casi d'uso del progetto; ognuno di questi descrive in modo strutturato alcune funzionalità del sistema.

Use Case #1	Studente - Registrazione
<b>Brief Description</b>	Lo studente crea un nuovo account sull'applicazione.
<b>Level</b>	User Goal
<b>Actors</b>	Studente
<b>Pre-conditions</b>	<ul style="list-style-type: none"> <li>- Studente non è ancora registrato</li> <li>- Si trova nella schermata di login/sign up</li> </ul>
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. Utente seleziona "Registrati"</li> <li>2. Sistema mostra il form di registrazione (vedi mockup fig. 5)</li> <li>3. L'utente inserisce i dati richiesti</li> <li>4. Conferma la registrazione</li> <li>5. Viene fatto il login automatico</li> </ol>
<b>Alternative Flows</b>	<ol style="list-style-type: none"> <li>4a. Dati non validi → sistema mostra errore</li> </ol>
<b>Post-conditions</b>	L'account dello studente è creato e salvato nel database

Use Case #2	Visualizza viaggi disponibili
<b>Brief Description</b>	Lo studente cerca viaggi già programmati per una specifica tratta e fascia oraria.
<b>Level</b>	User Goal
<b>Actors</b>	Studente
<b>Pre-conditions</b>	Lo studente ha effettuato l'accesso all'applicazione e si trova nella schermata principale.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. Dalla pagina principale, lo studente sceglie l'opzione per visualizzare i viaggi disponibili (vedi mockup fig.7)</li> <li>2. Lo studente inserisce la sede di partenza e arrivo, il giorno e la fascia oraria</li> <li>3. Il sistema mostra i viaggi compatibili</li> </ol>
<b>Alternative Flows</b>	<ol style="list-style-type: none"> <li>3a. Nessun viaggio compatibile con i criteri di ricerca → sistema mostra opzioni per creare nuovo viaggio (vedi mockup fig. 6)</li> </ol>
<b>Post-conditions</b>	L'utente visualizza la lista dei viaggi disponibili per i criteri scelti



<b>Use Case #3</b>	<b>Aggiungiti a Viaggio Esistente</b>
<b>Brief Description</b>	Lo studente si unisce a un viaggio esistente come passeggero.
<b>Level</b>	User Goal
<b>Actors</b>	Studente
<b>Pre-conditions</b>	Lo studente sta visualizzando la lista dei viaggi disponibili (vedi mockup 7).
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. Lo studente seleziona un viaggio</li> <li>2. Premendo "Aggiungiti", il sistema verifica disponibilità</li> <li>3. Il sistema associa la prenotazione e decrementa posti disponibili</li> <li>4. Sistema mostra la conferma della prenotazione</li> </ol>
<b>Alternative Flows</b>	<b>2a.</b> Viaggio pieno → sistema mostra errore
<b>Post-conditions</b>	Lo studente è stato aggiunto come passeggero al viaggio scelto

<b>Use Case #4</b>	<b>Crea Nuovo Viaggio</b>
<b>Brief Description</b>	Lo studente crea un nuovo viaggio proponendosi come guidatore.
<b>Level</b>	User Goal
<b>Actors</b>	Studente (Guidatore)
<b>Pre-conditions</b>	<ul style="list-style-type: none"> <li>- UC-2 non ha prodotto risultati</li> <li>- Studente è registrato come Guidatore</li> <li>- Ci sono dei veicoli disponibili</li> </ul>
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. Dopo una ricerca negativa, sistema mostra "Crea Nuovo Viaggio"</li> <li>2. Studente seleziona l'opzione e conferma la creazione</li> <li>3. Sistema crea nuovo Trip con studente come Driver</li> <li>4. Sistema mostra conferma creazione</li> </ol>
<b>Alternative Flows</b>	<b>3a.</b> Viaggio identico creato da altri → sistema annulla e propone di unirsi
<b>Post-conditions</b>	Un nuovo viaggio è stato creato con lo studente come guidatore

<b>Use Case #5</b>	<b>Admin - Elimina Utente</b>
<b>Brief Description</b>	L'amministratore elimina un account utente.
<b>Level</b>	User Goal
<b>Actors</b>	Admin
<b>Pre-conditions</b>	L'utente è registrato nel sistema.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. Admin accede alla gestione utenti</li> <li>2. Cerca un utente per nome, matricola o email</li> <li>3. Seleziona l'utente da eliminare</li> <li>4. Conferma l'eliminazione</li> <li>5. L'utente viene eliminato dal sistema</li> </ol>
<b>Alternative Flows</b>	<ol style="list-style-type: none"> <li>3a. L'utente ha prenotazioni future o è assegnato come guidatore in un viaggio → sistema richiede conferma aggiuntiva</li> </ol>
<b>Post-conditions</b>	L'utente è rimosso dal sistema, incluse eventuali prenotazioni

<b>Use Case #6</b>	<b>Admin - Aggiungi Navetta</b>
<b>Brief Description</b>	L'admin registra una nuova navetta nel sistema.
<b>Level</b>	Function
<b>Actors</b>	Admin
<b>Pre-conditions</b>	L'admin è autenticato e c'è un nuovo mezzo.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. Admin accede alla sezione "gestione navette"</li> <li>2. Sistema mostra il form per inserire i dati del veicolo</li> <li>3. Admin inserisce i dati della nuova navetta (posti, targa, ecc.)</li> <li>4. Conferma l'inserimento</li> <li>5. Viene aggiornata la lista dei veicoli</li> </ol>
<b>Alternative Flows</b>	<ol style="list-style-type: none"> <li>2a. Dati non validi → sistema mostra errore</li> <li>3a. La navetta è già presente → impedisce duplicazione</li> </ol>
<b>Post-conditions</b>	La navetta è disponibile per l'assegnazione ai gruppi

## 2.3 Page Navigation Diagram

Nel seguente diagramma sono riportate le possibili azioni che un utente può compiere, mostrando i modi con il quale si può navigare tra le pagine.

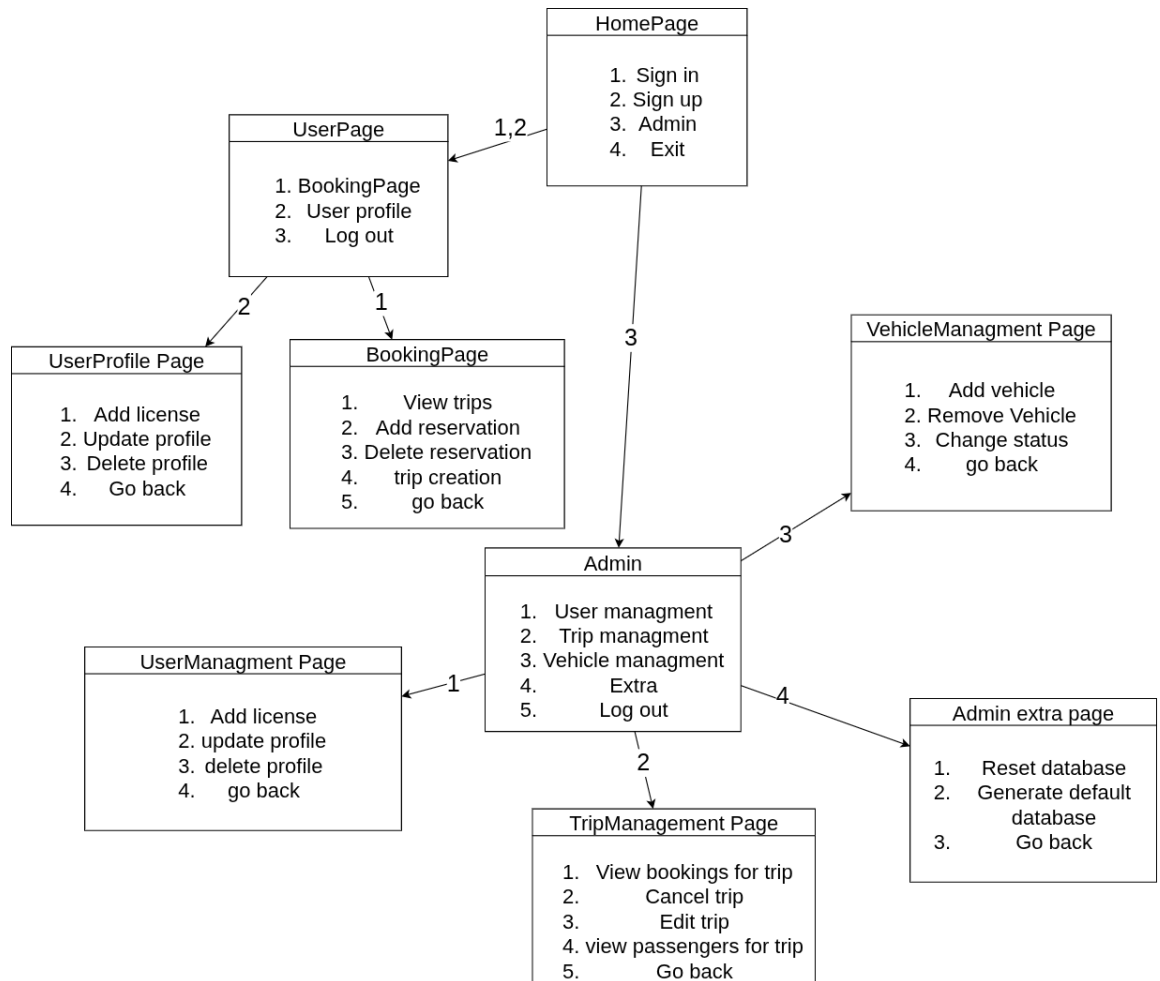
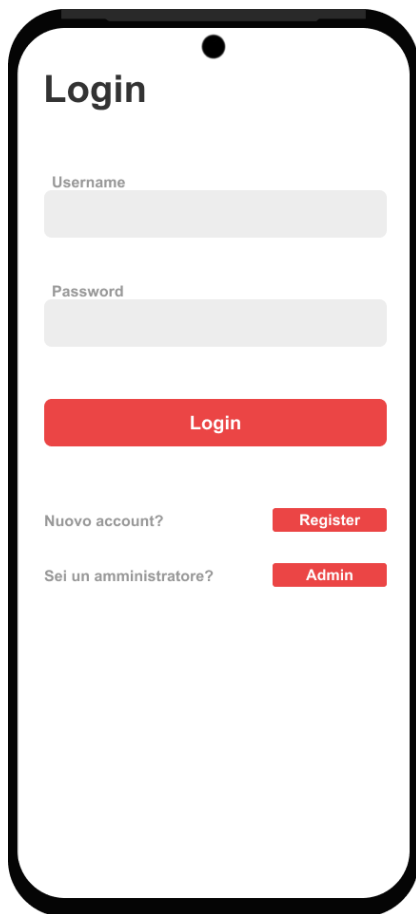


Figura 3: Page Navigation Diagram

## 2.4 Mockup

Di seguito presentiamo alcune rappresentazioni grafiche di una possibile implementazione dell'interfaccia utente del sistema. Le immagini sono state realizzate con Figma, uno strumento dedicato alla progettazione di interfacce grafiche. La scelta di proporre mockup per dispositivi mobili è motivata dall'ipotesi che la maggior parte degli utenti accederà al servizio tramite smartphone.



A mobile app login screen with a white background and rounded corners. At the top, the word "Login" is displayed in a bold, black font. Below it are two light gray input fields for "Username" and "Password". A red button with the text "Login" is positioned below the password field. At the bottom, there are two links: "Nuovo account?" followed by a red "Register" button, and "Sei un amministratore?" followed by a red "Admin" button.

Login

Username

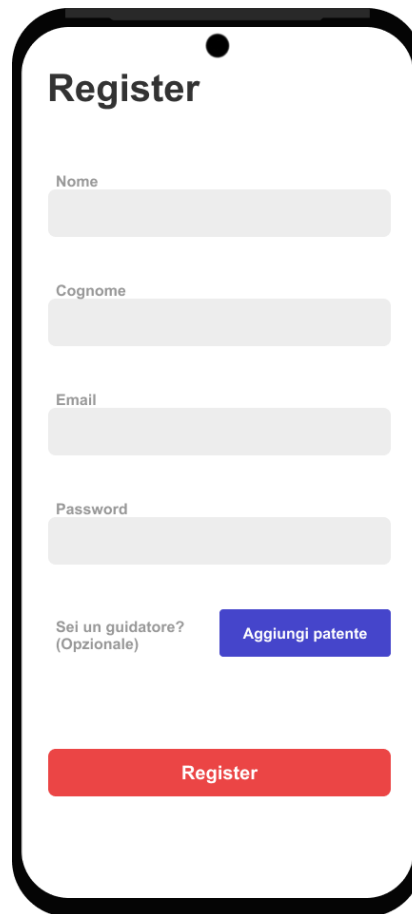
Password

Login

Nuovo account? Register

Sei un amministratore? Admin

Figura 4: Schermata di login



A mobile app registration screen with a white background and rounded corners. At the top, the word "Register" is displayed in a bold, black font. Below it are four light gray input fields for "Nome", "Cognome", "Email", and "Password". A blue button with the text "Aggiungi patente" is located below the "Email" field. At the bottom, there is a red button with the text "Register".

Register

Nome

Cognome

Email

Password

Aggiungi patente

Register

Figura 5: Schermata di registrazione



Figura 6: Schermata di creazione tragitto

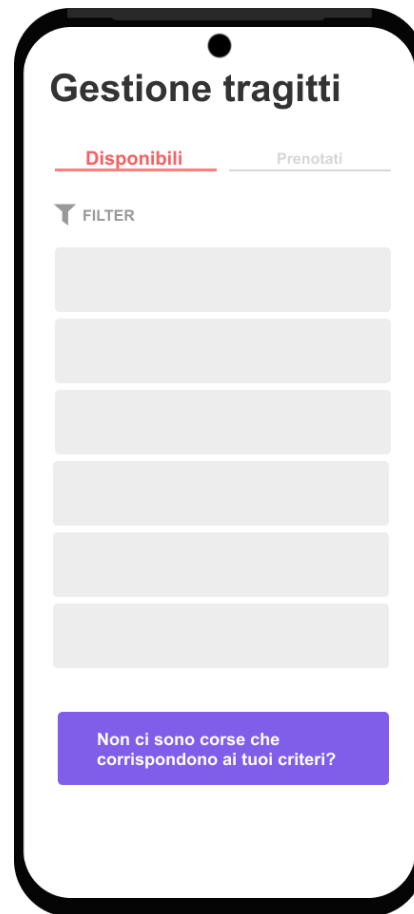


Figura 7: Schermata di visualizzazione dei tragitti

### 3 Class Diagram

Vista la progettazione a tre livelli, di seguito sono riportati i diagrammi delle classi per ciascuno dei modelli. Nella figura 8 è possibile osservare le dipendenze tra i vari package del sistema.

#### 3.1 Domain Model

Il Domain Model ha il compito di definire le entità concettuali fondamentali del dominio dell'applicazione; le sue classi rappresentano i concetti del "mondo reale", infatti le istanze saranno utilizzate negli altri layer dell'applicazione per

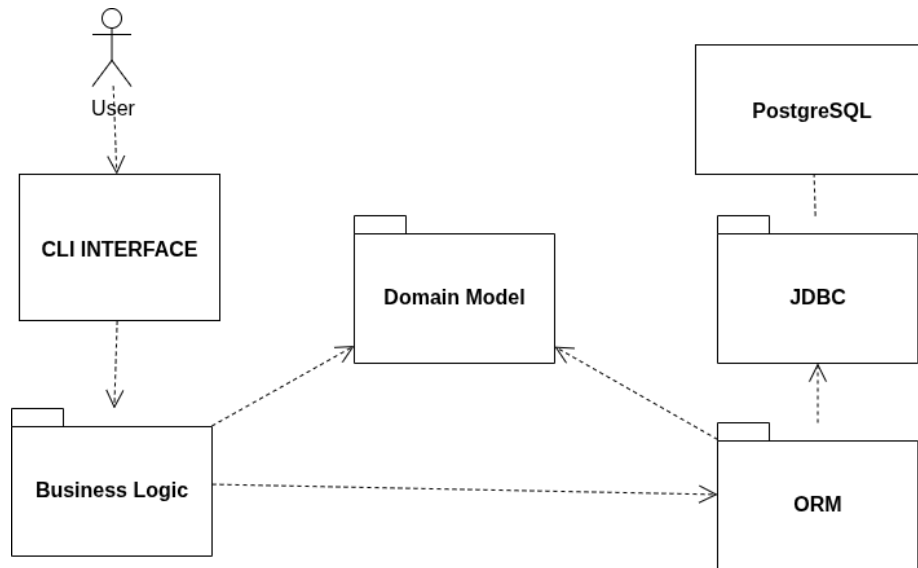


Figura 8: Package Dependency Diagram

eseguire elaborazioni sui dati senza richiedere un accesso diretto al database; solitamente sono implementate come *POJO* (*Plain Old Java Object*), non implementando nessun tipo di metodi oltre a getters e setters. Le entità principali che abbiamo individuato sono le seguenti:

- **User:** è la classe che rappresenta gli utenti; abbiamo deciso di distinguere studenti ed admin tramite l'uso di un attributo `UserRole` dato che in questo contesto condividono tutti gli altri attributi, ma nello strato di business avranno diversi privilegi operativi
- **Trip:** rappresenta un viaggio organizzato incapsulando tutte le informazioni sul viaggio tra cui: origine, destinazione, data, orario, guidatore assegnato e veicolo
- **Booking:** modella le prenotazioni fatte dagli studenti per i viaggi; in questa abbiamo deciso di non mettere data e orario perché sarebbe stata una duplicazione essendo già presenti nell'attributo di tipo `Trip`
- **Location:** modella le sedi universitarie tra cui gli studenti possono spostarsi
- **Vehicle:** rappresenta le navette che verranno usate nei viaggi

Nel seguente diagramma di classe vengono visualizzate le relazioni di composizione tra le classi.

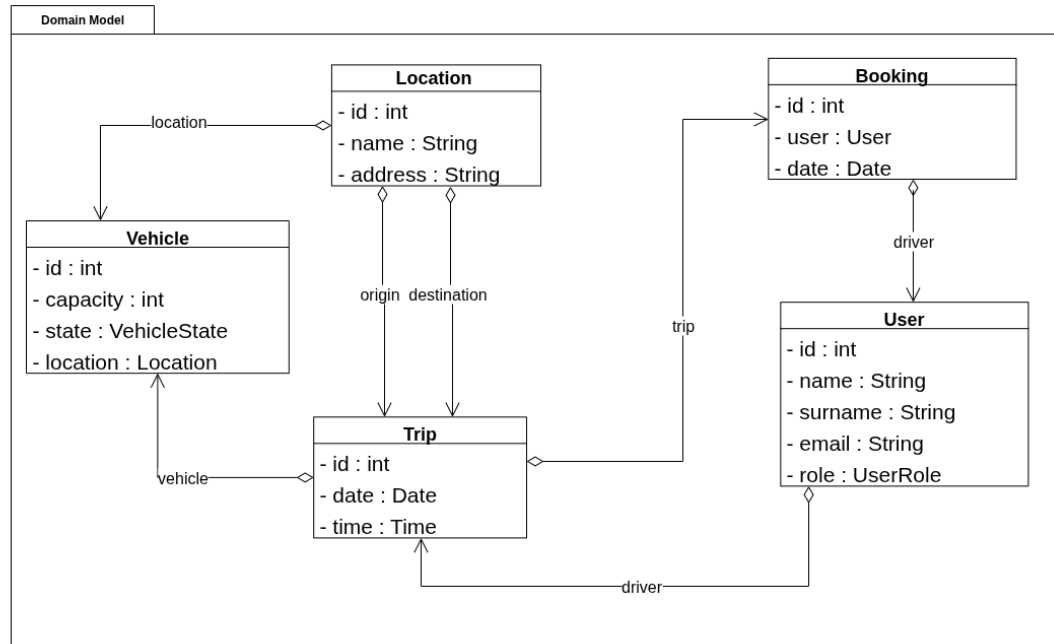


Figura 9: Domain Model class diagram

### 3.2 ORM

Questo livello ha il compito di gestire l'interazione con il database scelto; viene implementato il design pattern DAO (*Data Access Object*) fornendo un livello di astrazione utile per separare la logica di accesso ai dati nel database dalle operazioni di controllo e le entità del dominio applicativo. Questo approccio consente di riutilizzare la stessa logica di business su diversi database, modificando unicamente l'implementazione dei DAO e riducendo così l'accoppiamento tra i livelli.

Nel livello ORM è presente un DAO per ciascuna entità del Domain Model, ognuno dei quali implementa le operazioni CRUD. Inoltre, una classe di utilità denominata `ConnectionManager` centralizza la gestione della connessione al database, semplificando eventuali modifiche di configurazione senza impattare gli altri componenti. Di seguito viene riportato il diagramma delle classi

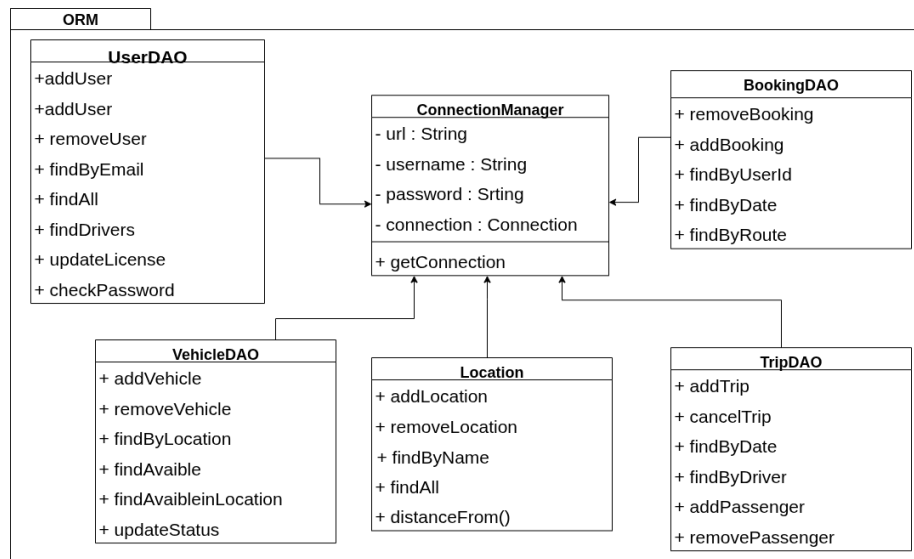


Figura 10: ORM class diagram

### 3.3 Business Logic

La Business Logic rappresenta la parte logica/operativa dell'applicazione; in questa vengono definite e implementate le regole di business e le operazioni di controllo, coordinando tutte le possibili operazioni.

Questo strato fa quindi uso dei DAO per l'accesso ai dati persistenti e delle classi del Domain Model per l'elaborazione delle informazioni dei calcoli. Ad ogni entità del Domain Model corrisponde un controller che ha il compito di gestire le operazioni principali che possono essere eseguite su quella entità; in più è presente una classe chiamata AuthController che si occupa della gestione dell'autenticazione e del controllo dei permessi degli utenti.



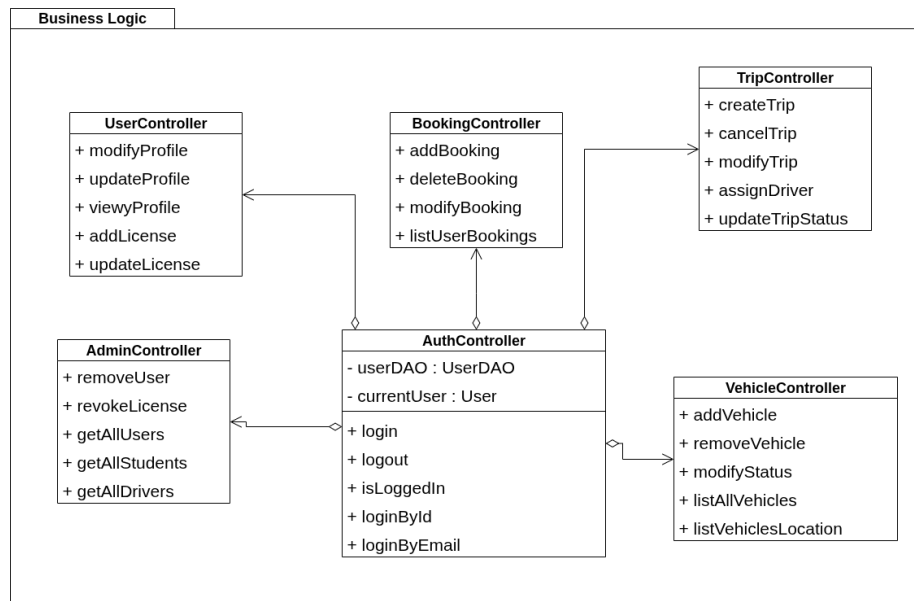


Figura 11: Business Logic Diagram

### 3.4 Diagramma ER e schema relazionale

Per la progettazione del database abbiamo realizzato un diagramma ER (vedi fig. 12), lo abbiamo trasformato in una rappresentazione più comprensibile (vedi fig. 13), e successivamente abbiamo costruito lo schema relazionale (vedi fig. 14).

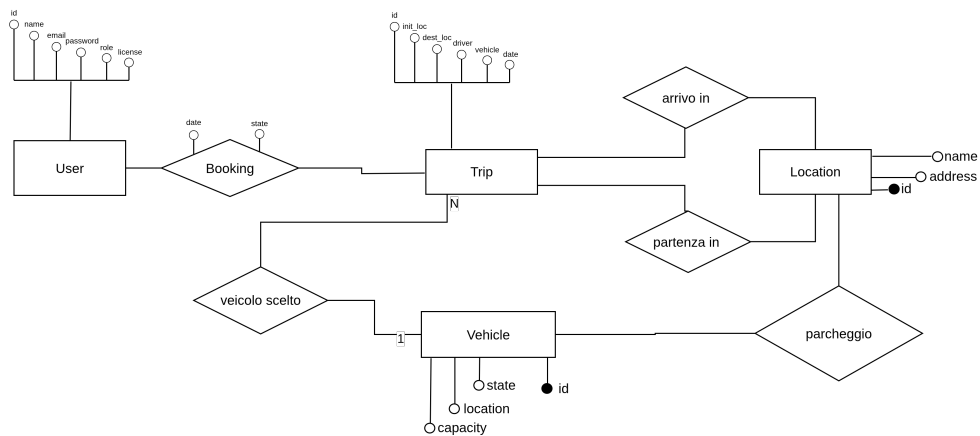


Figura 12: ER diagram

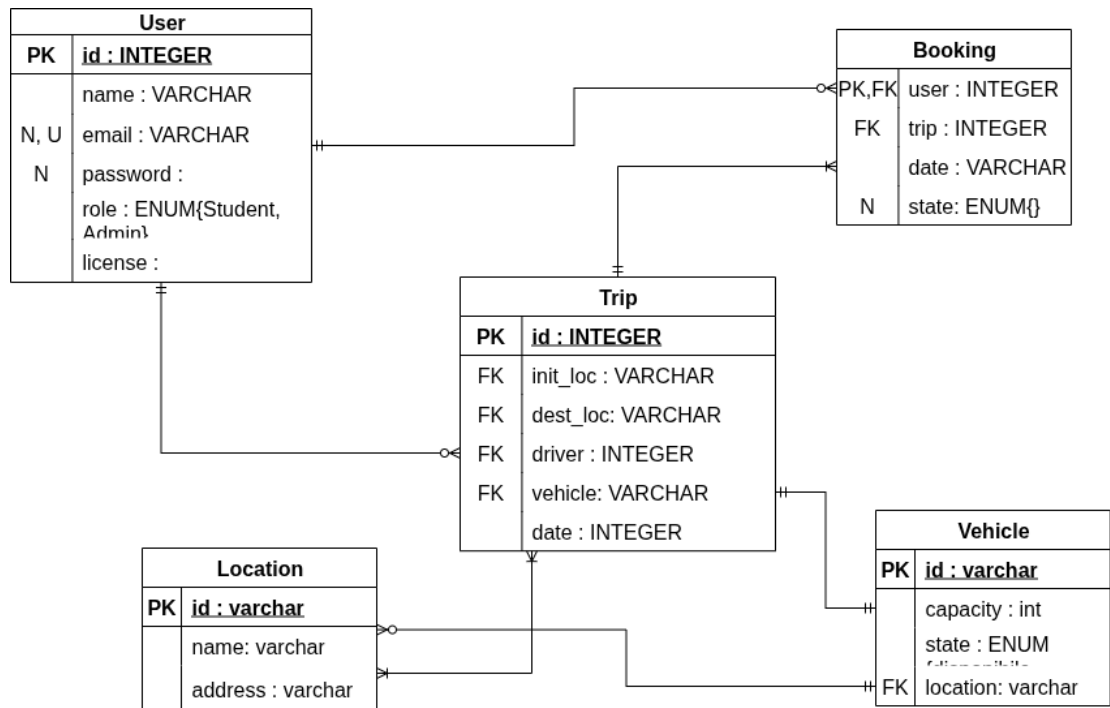


Figura 13: ER diagram (rappresentazione semplificata)

```

User(id, name, email, password, role)
Vehicle(id, capacity, state, location)
Trip(id, init_loc, dest_loc, driver, vehicle, date)
Location(id, name, address)
Booking(id, user, date, trip, state)

```

Figura 14: Schema relazionale

Come si può vedere dai diagrammi, il modello ER è stato tradotto in tabelle relazionali, mantenendo le stesse entità e relazioni.

## 4 Implementazione

L'implementazione del codice sorgente mantiene la struttura a tre livelli.

### 4.1 Struttura cartelle

La struttura delle directory del progetto è stata organizzata in modo da separare i vari componenti del sistema; si trovano le seguenti:

- docs: contiene la documentazione del progetto
- src/main: contiene il codice sorgente del progetto
- src/test : contiene i test del progetto
- src/sql: contiene gli script .sql

### 4.2 Implementazione Database

Di seguito viene mostrata l'implementazione del database PostgreSQL, che segue la struttura descritta in sez 3.4. Da notare è la scelta di come vengono generate i valori per i campi id di ogni classe: abbiamo pensato infatti che gli utenti avranno sicuramente una matricola, e quindi dovrà essere lo stesso utente ad inserirla, ma per tutte le altre classi quel campo viene generato automaticamente nel caso in cui non venga fornito, grazie al costrutto `id INTEGER GENERATED ALWAYS AS IDENTITY`.

Sono stati aggiunti alcuni vincoli per evitare anche a livello di database l'inserimento di valori non validi.

Listing 1: Esempio creazione tabelle User e Trip

```
1  -- Create tables
2  CREATE TABLE "User" (
3      id INTEGER primary key,
4      name VARCHAR(50),
5      surname VARCHAR(50),
6      email VARCHAR(40) UNIQUE NOT NULL,
7      password VARCHAR(100) NOT NULL,
8      role VARCHAR(15) NOT NULL DEFAULT 'STUDENT',
9      license VARCHAR(30) UNIQUE -- driver's license number
10 );
11
12
13 CREATE TABLE Trip (
14     id INTEGER GENERATED BY DEFAULT IDENTITY primary key NOT NULL,
15     origin INTEGER NOT NULL,
16     destination INTEGER NOT NULL,
17     date DATE NOT NULL,
18     time TIME NOT NULL,
19     state VARCHAR(20) NOT NULL DEFAULT 'SCHEDULED', -- SCHEDULED,
20     driver INTEGER,
21     vehicle INTEGER NOT NULL,
```

```

22 FOREIGN KEY (origin) REFERENCES Location(id),
23 FOREIGN KEY (destination) REFERENCES Location(id),
24 FOREIGN KEY (driver) REFERENCES "User"(id) ON DELETE SET NULL,
25 FOREIGN KEY (vehicle) REFERENCES Vehicle(id)
26 );

```

#### 4.2.1 Dati di Default

Il sistema include uno script `default-unifi` che popola il database con le sedi reali dell'Università di Firenze, alcuni veicoli, studenti e trip, ed un admin di default.

Listing 2: Esempio inserimento dati di default per le location

```

1 INSERT INTO Location (name, address) VALUES
2 ('Rettorato - UNIFI', 'Piazza San Marco, 4, 50121 Firenze FI'),
3 ('Facoltà di Ingegneria', 'Via Santa Marta, 3, 50139 Firenze FI'),
4 ('Centro Didattico Morgagni', 'Viale Morgagni, 65, 50134 Firenze
   FI'),

```

Listing 3: Esempio inserimento dati di default per le location

```

1 INSERT INTO Location (name, address) VALUES
2 ('Rettorato - UNIFI', 'Piazza San Marco, 4, 50121 Firenze FI'),
3 ('Facoltà di Ingegneria', 'Via Santa Marta, 3, 50139 Firenze FI'),
4 ('Centro Didattico Morgagni', 'Viale Morgagni, 65, 50134 Firenze
   FI'),

```

### 4.3 Domain Model

Nel package `main.DomainModel` sono implementate le classi che rappresentano le entità del dominio applicativo del sistema di car sharing universitario e l'implementazione segue la descrizione fornita in sez 3.1. Ogni classe, implementa tutti i metodi getter e setter per ogni attributo. In tutte le classi è presente un attributo `int id` che mantiene un intero che identifica quell'oggetto.

Da notare come, data l'implementazione del database (vedi sez 4.2), in tutte le classi tranne `User` sono implementati almeno due costruttori: uno per oggetti nuovi non ancora presenti nel database (e quindi senza `id` perché verrà generato automaticamente), e uno invece standard.

Listing 4: Esempio di costruttori della classe `Vehicle`

```

public Vehicle(int id, int capacity, VehicleState state) {
    this.id = id;
    this.capacity = capacity;
    this.state = state;
    this.location = null; // Default location is null
}
// constructor for new vehicles (id will be set by the database)
public Vehicle(int capacity, VehicleState state, Location location)
{
    this.id = 0;
}

```

```
        this.capacity = capacity;
        this.state = state;
        this.location = location;
    }
```

#### 4.3.1 User

La classe **User** rappresenta gli utenti del sistema, sia studenti che amministratori. I campi principali includono:

- **name, surname**: dati anagrafici
- **email, password**: credenziali di accesso
- **license**: patente di guida (opzionale per studenti)
- **role**: ruolo dell'utente (ADMIN o STUDENT)

Come descritto precedentemente viene usato il seguente ENUM per distinguere tra admin e studente

Listing 5: UserRole ENUM

```
public enum UserRole {
    ADMIN, STUDENT
}
```

#### 4.3.2 Vehicle e Location

**Vehicle** rappresenta le navette rese disponibili dall'università con attributi **int capacity**, **VehicleState state**, **Location location**. **Location** modella le sedi universitarie con nome, indirizzo e numero di parcheggi per le navette che ci sono. Ogni veicolo è associato alla **Location** nella quale si trova grazie all'attributo **location**, e lo stato indica se è funzionante, non funzionante o in riparazione.

**Location** contiene anche un attributo **int availableParkingSpots** che indica quanti posti liberi ci sono per parcheggiare le navette in quella sede, e poi attributi **name** e **address**, ed un intero **int capacity** che indica quanti veicoli in totale possono essere parcheggiati in quella sede.

#### 4.3.3 Trip

La classe **Trip** modella i viaggi creati dagli utenti con patente. Contiene informazioni su:

- Origine e destinazione (oggetti **Location**)
- Data e ora di partenza
- Conducente e veicolo assegnati

- Stato del viaggio (SCHEDULED, ONGOING, COMPLETED, CANCELED)

Per la data e l'ora abbiamo scelto rispettivamente `java.sql.Date` e `java.sql.Time` che ci ha permesso poi nei DAO di non aver problemi. La classe `Trip` rappresenta la classe più importante dopo `User` perché lega tutti i vari concetti insieme e rappresenta il centro del sistema di car sharing.

#### 4.3.4 Booking

La classe `Booking` rappresenta le prenotazioni degli studenti per i viaggi disponibili; ad ogni oggetto booking viene associato:

- `User user`: utente a cui è associata la prenotazione
- `Trip trip`: viaggio a cui è associata la prenotazione
- `BookingState state`: lo stato della prenotazione

è stato deciso di non mettere data e orario del viaggio a cui è associata la prenotazione perché sono informazioni che si possono estrarre dall'attributo `trip`.

### 4.4 Object-Relational Mapping

Nel package `main.ORM` sono implementate le classi DAO che gestiscono la persistenza dei dati e l'interazione con il database PostgreSQL.

#### 4.4.1 ConnectionManager

La classe `ConnectionManager` ha il compito di gestire la connessione al database per tutte le altre classi DAO attraverso il metodo statico `getConnection`. Questa inoltre contiene i parametri di accesso al database.

Listing 6: Implementazione del ConnectionManager

```
public static ConnectionManager getInstance() {
    if (instance == null) {
        instance = new ConnectionManager();
    }
    return instance;
}
public static Connection getConnection() throws SQLException {
    if (connection == null)
        try {
            connection = DriverManager.getConnection(url, username,
                password);
        } catch (SQLException e) {
            System.err.println("Error: " + e.getMessage());
        }
    return connection;
}
```

Come si nota dalla definizione, questa classe implementa il pattern *Singleton*, garantendo che esista una sola istanza condivisa di **ConnectionManager** in tutta l'applicazione. In questo modo tutti i DAO utilizzano la stessa connessione al database, semplificando la gestione delle risorse e riducendo il rischio di apertura simultanea di troppe connessioni. Il costruttore privato impedisce la creazione di istanze multiple, mentre il metodo statico `getInstance()` fornisce l'accesso globale all'unica istanza disponibile.

**DAO** Per creare le connessioni nei DAO abbiamo deciso di usare il costrutto di java *try-with-resources* perché garantisce la chiusura sicura dello statment anche in caso di eccezione, senza necessità di dover scrivere un blocco `finally` dove si chiude.

Listing 7: Esempio di uso di try-with-resources per la connessione nel metodo `insertUser` di `UserDAO`

```
String insertSQL = "INSERT INTO \"User\" (id, name, surname, email,
    password, role, license) VALUES (?, ?, ?, ?,?, ?, ?)";
try (PreparedStatement preparedStatement = connection.
    preparedStatement(insertSQL))
{
    // ... insert ...
}
```

#### 4.4.2 UserDAO

La classe `UserDAO` si occupa della gestione della persistenza per l'entità `User`, implementando tutte le operazioni CRUD (Create, Read, Update, Delete). Oltre ai metodi di base, sono presenti metodi specifici per la gestione della patente di guida degli utenti, come l'aggiunta o la rimozione della patente.

Per supportare le funzionalità amministrative, `UserDAO` offre metodi per il recupero filtrato degli utenti, ad esempio per ottenere tutti gli studenti, tutti gli amministratori, oppure solo gli studenti che possiedono o meno una patente. Per evitare duplicazione di codice, è stato adottato un approccio che prevede un metodo privato che esegue una query parametrica e popola una lista di oggetti `User` in base ai criteri specificati. Questo pattern è stato riutilizzato anche negli altri DAO, garantendo maggiore riusabilità e manutenibilità del codice.

Listing 8: Metodo per ottenere tutti gli `User` che soddisfano alcuni criteri

```
private void getUsersFromQuery(String selectSQL, List<User>
    userList) { //...
}
// esempio
public List<User> getAllStudents() throws SQLException {
    String selectSQL = "SELECT * FROM \"User\" WHERE role = '
        STUDENT'";
    List<User> users = new ArrayList<>();
    getUsersFromQuery(selectSQL, users);
    return users;
}
```

#### 4.4.3 TripDAO

La classe TripDAO implementa tutte le operazioni CRUD per l'entità Trip, consentendo la creazione, modifica, cancellazione e recupero dei viaggi dal database. Oltre alle operazioni di base offre metodi per la ricerca e il filtraggio dei viaggi, come la possibilità di ottenere tutti i viaggi programmati in una certa data, quelli associati a un determinato utente (come conducente o passeggero), o i viaggi disponibili in base allo stato e alla disponibilità di posti.

#### 4.4.4 BookingDAO

Implementa la gestione delle prenotazioni con supporto per nel database, offrendo anche una funzionalità per recuperare tutte le prenotazioni associate a un determinato utente o viaggio, oppure per il calcolo dei posti disponibili nei viaggi.

#### 4.4.5 VehicleDAO e LocationDAO

**VehicleDAO** Oltre alle operazioni standard, questa classe offre metodi per recuperare tutti i veicoli disponibili in una determinata sede e verificare la disponibilità di veicoli per un determinato viaggio. Inoltre offre metodi per aggiornare la posizione del veicolo quando viene spostato tra sedi diverse.

**LocationDAO** Questa classe permette di gestire le sedi universitarie, consentendo l'inserimento di nuove location, la modifica delle informazioni (nome, indirizzo, numero di parcheggi disponibili) e la rimozione di sedi non più utilizzate.

### 4.5 Business Logic

Nel package `main.BusinessLogic` sono implementati i controller che gestiscono la logica di business del sistema per ogni servizio.

#### 4.5.1 AuthController

Questa classe definisce le regole e i metodi per la gestione dell'autenticazione degli utenti, in particolare il metodo `login`. Mantiene un attributo `currentUser` che rappresenta l'utente attualmente autenticato nell'applicazione.

Grazie a questo durante la sessione è possibile verificare i permessi dell'utente e il suo ruolo (admin o studente) tramite i metodi `isAdmin` e `isLogged`.

#### 4.5.2 TripController

Ha il compito di gestione dei viaggi proponendo metodi come `createTrip`, `modifyTrip`, `isFull`. Il metodo `createTrip` implementa una serie di controlli di validazione prima di creare il viaggio, verificando autenticazione, disponibilità del veicolo e possesso della patente.



### 4.5.3 BookingController

Questa classe ha lo scopo di permettere agli utenti di creare prenotazioni (`createBooking`), modificarle (`modifyBooking`) o cancellarle con `cancelBooking`. Per la cancellazione abbiamo deciso che inizialmente le prenotazioni rimangono salvate nel database modificando però modificato lo stato; sarà poi l'admin a decidere se rimuoverle completamente con `removeBooking`

Listing 9: Cancellazione e rimozione di una Prenotazione

```
public boolean cancelBooking(int bookingId) {
    try {
        Booking booking = bookingDAO.findBookingByID(bookingId);
        ;

        //... operazioni di controllo

        booking.setState(Booking.BookingState.CANCELED); //
            cambia solo lo stato
        bookingDAO.updateBooking(booking);
        return true;
    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
}

public void removeBooking(int bookingId) {
    try {
        // ... check if admin ...
        bookingDAO.removeBooking(bookingId); // viene rimosso
            dal database
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

### 4.5.4 UserController

Tramite questa classe l'utente ha la possibilità di registrarsi (`register`), gestire, modificare o visualizzare il proprio profilo (`deleteThisProfile`, `viewProfile`), e fare operazioni riguardanti la patente (`addLicense`, `removeLicense`, `hasLicense`).

### 4.5.5 AdminController

Fornisce funzionalità amministrative per la gestione di utenti e sistema, tra cui `removeUser` per la rimozione del profilo di un utente, e `revokeLicense` per la revoca del permesso di guidare ad uno studente. Ogni metodo amministrativo implementa controlli di autorizzazione per garantire che solo gli admin possano eseguire operazioni privilegiate.

#### 4.5.6 VehicleController e LocationController

Queste due classi implementano la logica per le operazioni di gestione dei veicoli e delle sedi.

**Application Manager** Questa classe ha il compito di inizializzare e collegare tutti i controller tra loro, in modo che possano collaborare per eseguire le operazioni richieste dagli utenti.

## 5 Testing

Per verificare il corretto funzionamento del progetto abbiamo suddiviso i test due categorie principali:

- Test ORM: Verificano la corretta persistenza dei dati nel database attraverso le classi DAO.
- Test di Business Logic: Verificano il corretto funzionamento dei controller che gestiscono la logica di business del sistema.

Abbiamo deciso di non fare test su Domain Model in quanto contiene classi composte esclusivamente attributi e metodi getter-setter, senza logica complessa da testare, e non presentano dipendenze esterne.

I test sono stati sviluppati con JUnit 5, e ciascuna classe di test si occupa di ripulire i dati inseriti nel database al termine dell'esecuzione, garantendo così l'indipendenza tra i test.

Negli unit test è stato usato il pattern di *Dependency Injection* per iniettare le dipendenze necessarie alle classi da testare, facilitando così l'isolamento delle unità di codice.

Listing 10: Esempio di Dependency Injection in UserControllerTest

```
@BeforeEach
void setUp() {
    userDao = new UserDao();
    authController = new AuthController(userDao);
    userController = new UserController(userDao, authController);
    // ... creazione di un oggetto User per i test ...
}
@Test
void testRegister() throws SQLException {
    userController.register(1, "Prova", "Register", "prova.
        register@example.com", "password123", "B12345");
    User registeredUser = userDao.findById(1); // recupero dell'
        utente registrato dal database
    assertNotNull(registeredUser);
    assertEquals("Prova", registeredUser.getName());
    authController.loginById(1, "password123");
    assertTrue(authController.isLoggedIn());
}
```

### 5.1 Test riguardanti casi d'uso

Per ogni caso d'uso descritto nella sezione 2.2 sono stati implementati test specifici per verificare il corretto funzionamento delle funzionalità principali del sistema: di sotto viene riportato il mapping tra i test fatti e la descrizione dei casi d'uso.

- testRegister → UC-1

- `testViewAvailableTrips` → UC-2
- `testJoinTrip` → UC-3
- `testCreateNewTrip` → UC-4
- `testDeleteUser` → UC-5
- `testAddVehicle` → UC-6

## 5.2 Elenco unit test implementati

**Test ORM** Di seguito sono elencati i principali test implementati per verificare il corretto funzionamento delle classi DAO:

- `UserDAOTest`:
  - `testInsertUser()`
  - `testUpdateUserEmail()`
  - `testFindUserByID()`
  - `testAddLicense()`
  - `testRemoveLicense()`
  - `testRemoveUserByID()`.
- `TripDAOTest`:
  - `testInsertTrip()`
  - `testFindById()`
  - `testUpdateTripDate()`.
- `LocationDAOTest`:
  - `testAddLocation()`
  - `testFindByName()`
  - `testUpdateCapacity()`
  - `testUpdateCapacityNegative()`
  - `testRemoveLocation()`
- `VehicleDAOTest`:
  - `testInsertVehicle()`
  - `testFindInLocation()`
  - `testFindAvailableInLocation()`
  - `testUpdateStatus()`
  - `testRemoveVehicle()`

**Business Logic Test** Per rendere indipendenti i test di Business Logic dai DAO, è stato usato il mocking tramite Mockito, che simulano il comportamento di funzionalità delle classi reali, restituendo dati predefiniti.

Listing 11: Esempio uso di mocking in BookingControllerTest.

```
@Mock
private BookingDAO bookingDAO;
@Mock
private TripDAO tripDAO;
@Mock
private AuthController authController;
@Mock
private TripController tripController;
@InjectMocks
private BookingController bookingController;

private Trip testTrip;
void setup() {
    // .. Setup di un qualche oggetto trip per i test ..
    testTrip = //...
}

void testBookTripFull() throws SQLException {
    when(tripController.isFull(testTrip)).thenReturn(true); //
        simula che viaggio sia pieno

    Booking newBooking = bookingController.createBooking(
        testTrip.getId());

    verify(bookingDAO, never()).insertBooking(any()); //
        verifica che insertBooking non venga mai chiamato
    assertNull(newBooking); //verifica che il risultato sia
        null
}
```

Come si può vedere nel listato 11, l'annotazione `@Mock` viene utilizzata per creare oggetti mock delle classi DAO e del controller di autenticazione, creando automaticamente degli oggetti mock per quei tipi, mentre `@InjectMocks` viene usata per iniettare questi mock nel controller che si vuole testare, in questo caso `BookingController`.

Di seguito sono elencati i principali test implementati per verificare il corretto funzionamento dei controller:

- UserControllerTest:
  - registerStudent()
  - testDeleteThisProfile()
  - testDeleteProfile()
  - testLogin()
  - testLoginFail().

- TripControllerTest:
  - testCreateTrip()
  - testFindById()
  - testModifyTrip()
  - testCancelTrip()
- BookingControllerTest
  - testBookAndCancel()
  - testBookFullTrip()
- LocationControllerTest
  - testAddLocation()
  - testAddLocationNotAdmin()
  - testFindById()
  - testUpdateCapacity()
- VehicleControllerTest
  - testAddVehicle()
  - testAddVehicleNotAdmin()
  - testFindById()
  - testModifyStatus()
  - testListAvailableVehiclesForLocation()

## 6 Note

### 6.1 Utilizzo di AI per lo sviluppo

Durante lo sviluppo dell'applicativo abbiamo utilizzati LLMs come Claude e GitHub Copilot per accelerare delle fasi di scrittura del codice:

- GitHub Copilot si è rivelato molto utile in casi di scrittura di codice semplice e ripetitivo, come la scrittura di alcune varianti dei metodi CRUD nei DAO
- Claude è stato utilizzato principalmente per operazioni di refactoring e valutazione/esplorazione di possibili alternative progettuali, fornendo spunti utili

Importante sottolineare che l'uso di questi strumenti e il codice da loro generato è sempre stato sottoposto ad un'attenta revisione da parte degli sviluppatori di questo progetto; è stato osservato che in alcuni scenari più complessi (come i metodi per TripController dove ci sono molte cose che devono accadere) e dove serviva una visione completa del progetto, veniva generato eccessivamente semplificato e spesso concettualmente sbagliato