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

1 Introduzione **2**
- 1.1 Statement . . . 2
- 1.2 Architettura e pratiche utilizzate . . . 2

2 Implementazione **4**
- 2.1 Struttura cartelle . . . 4
- 2.2 Implementazione Database . . . 4
  - 2.2.1 Vincoli di Integrità . . . 5
  - 2.2.2 Dati di Default . . . 5
- 2.3 Domain Model . . . 5
  - 2.3.1 User . . . 6
  - 2.3.2 Vehicle e Location . . . 6
  - 2.3.3 Trip . . . 6
  - 2.3.4 Booking . . . 7
- 2.4 Object-Relational Mapping . . . 7
  - 2.4.1 ConnectionManager . . . 7
  - 2.4.2 UserDAO . . . 8
  - 2.4.3 TripDAO . . . 8
  - 2.4.4 BookingDAO . . . 8
- 2.5 VehicleDAO e LocationDAO . . . 8
- 2.6 Business Logic . . . 8
  - 2.6.1 AuthController . . . 9
  - 2.6.2 TripController . . . 9
  - 2.6.3 BookingController . . . 9
  - 2.6.4 UserController . . . 10
  - 2.6.5 AdminController . . . 10
  - 2.6.6 VehicleController e LocationController . . . 10

3 Testing **11**
- 3.1 Business Logic Test . . . 11
- 3.2 ORM Test . . . 12

4 Note **14**
- 4.1 Utilizzo di AI per lo sviluppo . . . 14

1 Introduzione

Questo elaborato rappresenta la relazione del progetto svolto per l'esame di Ingegneria del Software del corso di Laurea Triennali in Ingegneria Informatica dell'Università degli Studi di Firenze. Il codice sorgente del progetto è disponibile su *GitHub* nel 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 facilitando la mobilità tra sedi universitarie.
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. Questo sistema si ispira al progetto TUSS (The Ultimate Sharing Service) dell'Università di Firenze, servizio che si propone come soluzione al problema di spostamenti tra alcune sedi universitarie, problema affrontato dagli stessi sviluppatori del progetto.

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 nella struttura complessiva del sistema.

- **Domain Model**: definisce le entità principali dell'applicazione.
- **Business Logic**: contiene le classi che implementano le funzionalità principali del sistema, come la gestione delle prenotazioni, il controllo dei permessi, definendo quindi la parte logica del progetto;
- **ORM**: include le classi responsabili del mapping tra oggetti Java e tabelle del database, permettendo la lettura e scrittura dei dati in modo astratto e modulare.

I diagrammi UML, inclusi gli Use Case Diagram e i 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 la 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 LaTeX

2 Implementazione

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

2.1 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

2.2 Implementazione Database

Il database può essere creato tramite lo script `default` che genera tutte le tabelle. Da notare sicuramente è il fatto della 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. In questo modo le operazioni di inserimento diventano più semplici; questo è stato possibile grazie al costrutto `id INTEGER GENERATED ALWAYS AS IDENTITY`, che genera automaticamente un valore intero tranne quando è fornito.
Sono stati aggiunti poi 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, -- TODO: mettere limite a caratteri
           matricolo
4      name VARCHAR(50),
5      surname VARCHAR(50),
6      email VARCHAR(40) UNIQUE NOT NULL,
7      password VARCHAR(100) NOT NULL, -- FIXME: fare hash??
8      role VARCHAR(15) NOT NULL DEFAULT 'STUDENT',
9      license VARCHAR(30) UNIQUE -- driver's license number: FIXME:
           mettere in un altra tabella??
10 );
11
12
13 CREATE TABLE Trip (
14     id INTEGER GENERATED Always As 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,
```

```
    state VARCHAR(20) NOT NULL DEFAULT 'SCHEDULED', -- SCHEDULED, COMPLETED, CANCELLED
    driver INTEGER , --TODO: da decidere come gestire creazione trip
    vehicle INTEGER NOT NULL,
    FOREIGN KEY (origin) REFERENCES Location(id),
    FOREIGN KEY (destination) REFERENCES Location(id),
    FOREIGN KEY (driver) REFERENCES "User"(id) ON DELETE SET NULL,
    FOREIGN KEY (vehicle) REFERENCES Vehicle(id)
);
```

2.2.1 Vincoli di Integrità

Il database implementa vincoli referenziali e di dominio per garantire la consistenza:

- Foreign key con cascading per mantenere l'integrità referenziale
- Check constraints per validare valori numerici positivi
- Unique constraints per email e patenti
- Default values per semplificare le operazioni di inserimento

2.2.2 Dati di Default

Il sistema include uno script `default-unifi` che popola il database con le sedi reali dell'Università di Firenze:

Listing 2: Inserimento dati di default per le location

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

2.3 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 **??**. 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 2.2), 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.

2.3.1 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 precentemente viene usato il seguente ENUM per distinguere tra admin e studente

Listing 3: UserRole ENUM

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

2.3.2 Vehicle e Location

`Vehicle` rappresenta le navette rese disponibili dall'università con attributi `int capacity, VehicleStase 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.

2.3.3 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.

2.3.4 Booking

La classe `Booking` gestisce 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` essendo un `Trip`.

2.4 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.

2.4.1 ConnectionManager

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

Listing 4: Implementazione del ConnectionManager

```
public class ConnectionManager {
private static final String url = "jdbc:postgresql://localhost
    :5432/SWE?currentSchema=public";
private static final String username = "postgres";
private static final String password = "password";
private ConnectionManager() {} // Utility class

public static Connection getConnection() throws SQLException {
    return DriverManager.getConnection(url, username, password);
}
}
```

Come si nota dalla definizione, questa classe è una *Utility Class* con costruttore privato per impedire l'istanzazione. Un'altra possibile scelta era quella di implementare un *Singleton* in questa classe, ma abbiamo optato per la prima opzione perché in questo modo ogni DAO può avere la propria connessione, e non ci deve essere un'unica connessione per tutti i DAO che vengono istanziati, rendendo la gestione molto più semplice.

DAO Importante notare che data questa scelta ogni DAO dovrà creare una propria connessione ogni volta che vuole eseguire una query sul database, e abbiamo deciso di usare il costrutto di java *try-with-resources* perché garantisce la chiusura sicura della connessione al database anche in caso di eccezione, rendendo il codice più robusto.

Listing 5: 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 (Connection connection = ConnectionManager.getConnection();
    PreparedStatement preparedStatement = connection.
        prepareStatement(insertSQL))
{
    // ... insert ...
}
```

2.4.2 UserDAO

Implementa le operazioni CRUD per l'entità User, con metodi specializzati per la gestione delle patenti. Il DAO implementa anche metodi per il recupero filtrato di utenti (tutti gli studenti, tutti gli admin, studenti con/senza patente) per supportare le funzionalità amministrative, tramite un metodo privato nel quale si può specificare i parametri da cambiare; questa tecnica è stata usata anche negli altri DAO per evitare duplicazione di codice.

Listing 6: Metodo per ottenere tutti gli User che soddisfano alcuni criteri

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

2.4.3 TripDAO

Gestisce la persistenza dei viaggi con supporto per query complesse:

2.4.4 BookingDAO

Implementa la gestione delle prenotazioni con supporto per aggregazioni: Il metodo supporta il calcolo dei posti disponibili nei viaggi, fondamentale per la logica di business del sistema.

2.5 VehicleDAO e LocationDAO

2.6 Business Logic

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

2.6.1 AuthController

è l'unica classe in cui sono definite le regole per la gestione dell'autenticazione degli utenti `login` e la verifica con `isLoggedIn()`; ha un attributo `User currentUser` che rappresenta l'user che si è autenticato nell'applicazione. Questo controller è stato iniettato in altri controller tramite l'uso del pattern di *Dependency Injection* principalmente usato da altri controller per controllare i permessi dell'utente per fare operazioni sulle risorse.

2.6.2 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.

2.6.3 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 rimangano salvate nel database ma viene modificato lo stato; sarà poi l'admin a decidere se rimuoverle completamente con `removeBooking`

Listing 7: 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();
        }
    }
```

2.6.4 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`).

2.6.5 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.

2.6.6 VehicleController e LocationController

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

3 Testing

Per verificare il corretto funzionamento del progetto, sono stati realizzati dei test funzionali sia sul livello di Business Logic sia sul livello di ORM, con l'obiettivo di verificare il corretto funzionamento dei metodi principali e delle operazioni di persistenza dei dati. 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. Per alcune classi sono stati mostrati solo i test più rappresentativi, mentre i test aggiuntivi, inclusi casi limite e casi di errore, sono descritti nella legenda degli snippet.

3.1 Business Logic Test

Sono stati implementati i seguenti test nelle seguenti classi:

- `UserControllerTest`: `testRegister()`, `testDeleteThisProfile()`, `testDeleteProfile()`, `testLogin()`, `testLoginFail()`.
- `TripControllerTest`: `createTrip()`, `findById()`, `modifyTrip()`, `cancelTrip()`, `getAvailableSeatsAndIsFull()`, `createTripWithPastDate()`, `cancelTripNonExistent()`.
- `BookingControllerTest`: `bookAndCancel()` (da implementare con DAO mock).

I test della classe `UserControllerTest` verificano il corretto funzionamento dei metodi di registrazione, login e cancellazione del profilo utente, comprese le situazioni di errore come credenziali non valide. I metodi `testDeleteThisProfile()` e `testDeleteProfile()` garantiscono che l'utente venga rimosso correttamente dal database e che la sessione di autenticazione venga terminata.
I test della classe `TripControllerTest` coprono la creazione, modifica, ricerca e cancellazione dei viaggi, includendo casi limite come la creazione di viaggi con date passate o la cancellazione di viaggi inesistenti. Inoltre, vengono verificati i metodi di gestione dei posti disponibili e lo stato di pieno dei veicoli.
I test della classe `BookingControllerTest` verificano la creazione e cancellazione delle prenotazioni, assicurando la coerenza tra Business Logic e database.

Listing 8: Registrazione di un utente in UserControllerTest

```
@Test
void testRegister() throws SQLException {
    userController.register(1, "John", "Doe", "john.doe@example.com
        ", "password123", "B12345", User.UserRole.STUDENT);
    User registeredUser = userDAO.findById(1);
    assertNotNull(registeredUser);
    assertEquals("John", registeredUser.getName());
    authController.loginById(1, "password123");
    assertTrue(authController.isLoggedIn());
}
```

Listing 9: Creazione di un viaggio in TripControllerTest

```
@Test
void createTrip() {
    Location origin = locationDAO.addLocation(new Location("Origin"
        , "Addr1", 5));
    Location destination = locationDAO.addLocation(new Location("
        Destination", "Addr2", 5));
    Vehicle vehicle = vehicleDAO.insertVehicle(new Vehicle(4,
        Vehicle.VehicleState.WORKING, origin));
    Trip trip = tripController.createTrip(vehicle.getId(), origin,
        destination, Date.valueOf("2025-09-07"), Time.valueOf("
        10:00:00"));
    assertNotNull(trip);
    assertEquals("Origin", trip.getOrigin().getName());
    assertEquals("Destination", trip.getDestination().getName());
}
```

3.2 ORM Test

Sono stati implementati i seguenti test nelle seguenti classi:

- `UserDAOTest`: `insertUser()`, `updateUserEmail()`, `findUserByID()`, `addLicense()`, `removeLicense()`, `removeUserByID()`.
- `TripDAOTest`: `insertTrip()`, `findById()`, `updateTripDate()`.
- `LocationDAOTest`: `addLocation()`, `findByName()`, `updateCapacity()`, `updateCapacityNegative()`, `removeLocation()`.
- `VehicleDAOTest`: `insertVehicle()`, `findInLocation()`, `findAvailableInLocation()`, `updateStatus()`, `removeVehicle()`.

I test delle classi DAO verificano la corretta persistenza dei dati nel database, coprendo tutte le operazioni CRUD. Sono stati aggiunti anche test per casi limite, come la modifica della capacità di una location con valori negativi, o la ricerca di veicoli disponibili filtrando quelli fuori servizio. I metodi `findById()` e `updateTripDate()` della classe `TripDAOTest` garantiscono la corretta gestione degli aggiornamenti e delle ricerche nel database.

Listing 10: Inserimento di un utente in UserDAOTest

```
@Test
void insertUser() throws SQLException {
    userDAO.insertStudent(12345, "Mario", "Rossi", "
        mariorossi@prova.com", "password123");
    User user = userDAO.findById(12345);
    assertEquals("Mario", user.getName());
    assertEquals("Rossi", user.getSurname());
}
```

Listing 11: Aggiunta di una location in LocationDAOTest

```
@Test
```

```
void addLocation() {
    Location location = locationDAO.addLocation(new Location("Test
        Location", "123 Test St", 10));
    Location retrieved = locationDAO.findById(location.getId());
    assertNotNull(retrieved);
    assertEquals("Test Location", retrieved.getName());
}
```

Listing 12: Inserimento di un veicolo in VehicleDAOTest

```
@Test
void insertVehicle() {
    Vehicle v = new Vehicle(20, Vehicle.VehicleState.WORKING,
        location);
    Vehicle found = vehicleDAO.insertVehicle(v);
    assertNotNull(found);
    assertEquals(20, found.getCapacity());
    assertEquals(Vehicle.VehicleState.WORKING, found.getState());
}
```

La combinazione dei test di Business Logic e dei test ORM assicura che i metodi principali del sistema funzionino correttamente sia a livello di logica applicativa sia a livello di gestione dei dati, garantendo l'integrità e la consistenza del database.

4 Note

4.1 Utilizzo di AI per lo sviluppo

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

- GitHub Copilot si è rilevato 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