



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE  
DIPARTIMENTO DI INGEGNERIA INFORMATICA

---

## Apartament

---

Link Github:  
<https://github.com/Pennelli02/SweProject>

---

*Autori:*

Pennelli Lorenzo Maria

Leuter Lorenzo

*Docente corso:*

Vicario Enrico

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Descrizione del progetto . . . . .	3
1.2	Architettura del progetto e Ambiente di sviluppo . . . . .	3
<b>2</b>	<b>Progettazione</b>	<b>4</b>
2.1	Use Case Diagram . . . . .	4
2.2	Use Case Templates . . . . .	5
2.3	Mockups . . . . .	10
2.4	Class Diagram . . . . .	13
2.5	Dettagli di Progetto . . . . .	15
2.5.1	Singleton . . . . .	15
2.5.2	Mapper . . . . .	16
2.5.3	Builder Telescoping Constructor . . . . .	17
2.5.4	DAO . . . . .	21
2.6	ER Diagram e Relational Model . . . . .	21
2.6.1	Dettagli implementativi del database . . . . .	23
<b>3</b>	<b>Implementazione</b>	<b>24</b>
3.1	Business Logic . . . . .	24
3.1.1	UserController . . . . .	25
3.1.2	ProfileController . . . . .	25
3.1.3	ResearchController . . . . .	25
3.1.4	AdminController . . . . .	25
3.2	Domain Model . . . . .	26
3.2.1	RegisteredUser . . . . .	26
3.2.2	Booking . . . . .	26
3.2.3	Accommodation . . . . .	26
3.2.4	Review . . . . .	26
3.2.5	SearchParametersBuilder e SearchParameters . . . . .	26
3.3	ORM . . . . .	27
3.3.1	DatabaseConnection . . . . .	28
3.3.2	UserDAO . . . . .	28
3.3.3	BookingDAO . . . . .	29
3.3.4	PreferenceDAO . . . . .	30
3.3.5	ReviewDAO . . . . .	30
3.3.6	AccommodationDAO . . . . .	30
3.3.7	DBUtils . . . . .	33
3.4	Interfaccia CLI . . . . .	33
<b>4</b>	<b>Test</b>	<b>34</b>
4.1	Domain Model Test . . . . .	34
4.2	Business Logic Test . . . . .	34
4.3	ORM Test . . . . .	36
4.4	Risultati dei test . . . . .	38

# 1 Introduzione

## 1.1 Descrizione del progetto

L'obiettivo è creare un'applicazione java che permetta di gestire le prenotazioni di alloggi, in particolare verranno trattati B&B, appartamenti e hotel. Gli utenti avranno la possibilità di effettuare ricerche e prenotazioni degli alloggi disponibili, selezionare il numero di persone che soggiorneranno, la data di inizio e di fine del soggiorno, e molti altri filtri che verranno spiegati successivamente. Inoltre, l'utente potrà anche cancellare le prenotazioni, inserire tra i preferiti gli alloggi e lasciare delle recensioni a essi. E inoltre presente un Admin che può cancellare definitivamente gli utenti, gli alloggi o le recensioni, aggiungere nuovi alloggi e modificare gli alloggi già presenti.

## 1.2 Architettura del progetto e Ambiente di sviluppo

L'architettura del progetto è rappresentata dalla figura sottostante:

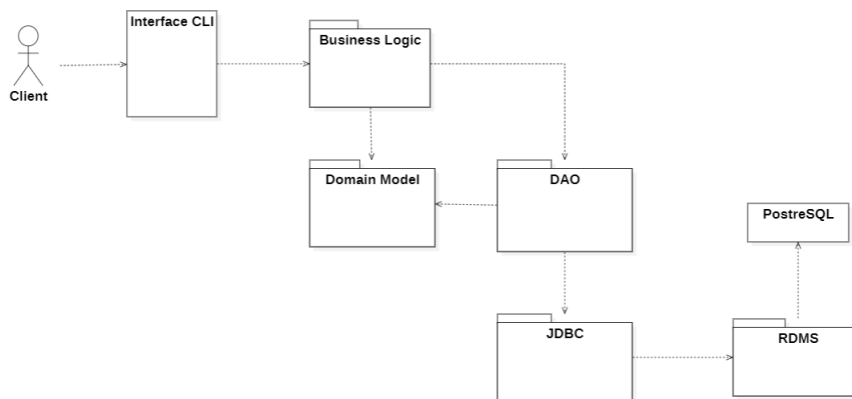


Figura 1: Diagramma dell'architettura del progetto

L'architettura del progetto è articolata in quattro componenti:

- **Business Logic:** contiene le classi che implementano la logica di business nel sistema.
- **Domain Model:** contiene le classi che rappresentano le entità del sistema.
- **ORM:** contiene le classi che permettono di gestire la comunicazione tra l'applicazione e il database andando a separare la logica per l'interazione con i dati dal resto dell'applicazione.
- **Interfaccia CLI:** interfaccia di linea di comando (CLI) dove l'utente può usufruire delle funzioni dell'applicazione e interagire con il sistema.

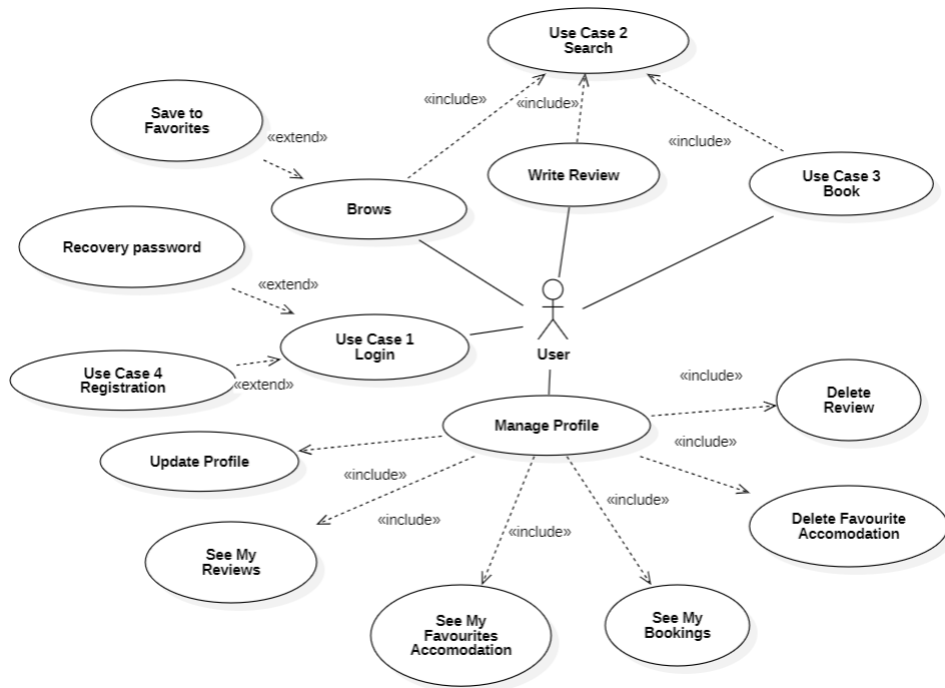
L'applicazione è stata sviluppata nel linguaggio Java e il database è stato implementato con PostgreSQL. La connessione tra il progetto e il database è realizzata tramite JDBC. Sono state utilizzate le seguenti piattaforme e software:

- **IntelliJ IDEA:** IDE per lo sviluppo in Java.
- **StarUML:** software per la creazione di diagrammi UML.
- **Draw.io:** software per la realizzazione di altri diagrammi, tra cui il modello ER.
- **PgAdmin:** software per la gestione del database PostgreSQL.
- **GitHub:** piattaforma contenente il codice sorgente.
- **Lunacy:** software per la realizzazione dei mockup.

## 2 Progettazione

### 2.1 Use Case Diagram

Sono presenti 2 tipi di utenti: lo User e l'Admin. Nel diagramma sottostante vengono rappresentati i casi d'uso per i due tipi di utenti:



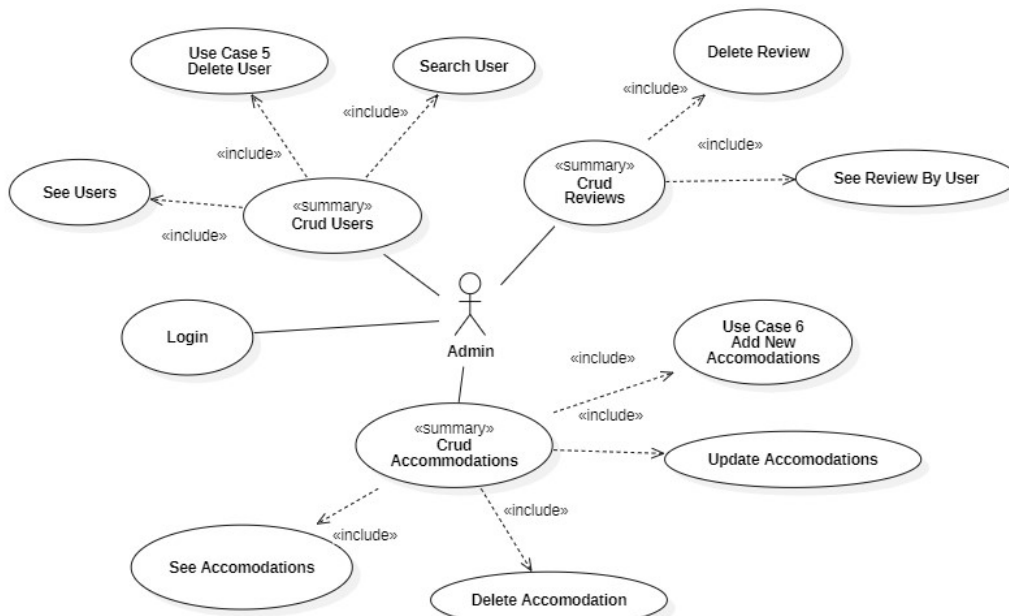


Figura 2: Use Case Diagram dello User e dell'Admin

## 2.2 Use Case Templates

Sono di seguito mostrati alcuni template dei casi d'uso. In alcuni di essi sono presenti riferimenti a test e mockup presenti successivamente (sezione 2.3):

Use Case 1	Login (Tst#1)
Descrizione	L'utente accede al sistema inserendo le sue credenziali.
Livello	Function
Attori	Utente, Admin
Flusso Base	<ol style="list-style-type: none"> <li>1. L'utente inserisce le sue credenziali (email e password) (MK#1 e MK#3).</li> <li>2. L'utente preme il pulsante di Login.</li> <li>3. Il sistema verifica le credenziali.</li> <li>4. Il sistema autentica l'utente.</li> </ol>
Flusso Alternativo	<ol style="list-style-type: none"> <li>3a. Se le credenziali sono errate, il sistema invia un messaggio di errore.</li> <li>3b. Se si verifica un problema all'interno del database durante la ricerca dell'utente, il sistema invia un messaggio di errore.</li> <li>3c. Se l'email è giusta, ma la password sbagliata, il sistema consente un recupero della password o l'immissione di un ulteriore tentativo</li> <li>4a. Se l'accesso viene effettuato dall'utente, sarà indirizzato alla sua pagina personale.</li> </ol>
Post-condizioni	L'utente è autenticato dal sistema e ha accesso alle sue funzionalità.

Tabella 1: Template che descrive il caso d'uso del login

Use Case 2	Search
Descrizione	L'utente cerca l'alloggio di suo interesse.
Livello	User Goal
Attori	User
Flusso Base	<ol style="list-style-type: none"> <li>1. L'utente inserisce le informazioni per effettuare la sua ricerca (MK#6).</li> <li>2. L'utente preme il pulsante per effettuare la ricerca.</li> <li>3. Il sistema usa le informazioni per ricercare gli alloggi.</li> <li>4. Il sistema restituisce all'utente una lista di alloggi (MK#7).</li> </ol>
Flusso Alternativo	<ol style="list-style-type: none"> <li>3a. Se l'utente non inserisce alcune informazioni necessarie alla ricerca (es: il luogo dove vuole andare, la data di check-in, la data di check-out), il sistema restituisce un messaggio di errore.</li> <li>3b. Se si verifica un problema all'interno del database durante la ricerca dell'alloggio, il sistema invia un messaggio di errore.</li> </ol>
Pre-condizioni	L'utente deve aver fatto il login.
Post-condizioni	L'utente riceverà una lista di alloggi da consultare.

Tabella 2: Template che descrive il caso d'uso dello Search

Use Case 3	Book
Descrizione	L'utente prenota un alloggio.
Livello	User Goal
Attori	User
Flusso Base	<ol style="list-style-type: none"> <li>1. L'utente preme il pulsante per effettuare la prenotazione dell'alloggio (MK#2).</li> <li>2. Il sistema riceve la richiesta di prenotazione e verifica la disponibilità dell'alloggio.</li> <li>3. Il sistema restituisce all'utente un messaggio di conferma.</li> </ol>
Flusso Alternativo	<ol style="list-style-type: none"> <li>2a. Se la disponibilità è zero, il sistema restituisce un messaggio di errore.</li> <li>2b. Se si verifica un problema durante il salvataggio della prenotazione nel database, il sistema invia un messaggio di errore.</li> <li>2c. Se l'utente non aveva inserito certi parametri per la ricerca ( data check-in check-out e numero di stanze e persone), gli verrà chiesto di inserirle per effettuare la prenotazione</li> </ol>
Pre-condizioni	L'utente deve aver fatto il login e deve aver effettuato la ricerca.
Post-condizioni	La prenotazione effettuata verrà aggiunta a quelle già effettuate dell'utente.

Tabella 3: Template che descrive il caso d'uso del Book

Use Case 4	Registration (Tst#2)
Descrizione	L'utente si registra all'interno del sistema.
Livello	User Goal
Attori	User
Flusso Base	<ol style="list-style-type: none"> <li>1. L'utente inserisce i parametri per registrarsi (MK#5).</li> <li>2. L'utente preme il pulsante per effettuare la registrazione.</li> <li>3. Il sistema verifica le informazioni fornite.</li> <li>4. Il sistema crea un nuovo account per l'utente.</li> </ol>
Flusso Alternativo	<ol style="list-style-type: none"> <li>3a. Se l'utente inserisce dati non validi (es: l'email già usata o vuota), il sistema invia un messaggio di errore all'utente.</li> <li>3b. Se si verifica un problema durante il salvataggio dell'utente nel database, il sistema invierà un messaggio di errore.</li> </ol>
Post-condizioni	L'utente è registrato all'interno del sistema e può accedere tramite le sue credenziali.

Tabella 4: Template che descrive il caso d'uso del Registration



Use Case 5	Delete User
Descrizione	L'Admin elimina un utente dal database.
Livello	User Goal
Attori	Admin
Flusso Base	<ol style="list-style-type: none"> <li>1. L'admin inserisce i parametri che caratterizzano l'utente (es: Id o email).</li> <li>2. L'admin preme il pulsante per effettuare l'eliminazione dell'utente.</li> <li>3. Il sistema verifica le informazioni fornite.</li> <li>4. Il sistema elimina l'utente dal database.</li> </ol>
Flusso Alternativo	<ol style="list-style-type: none"> <li>3a. Se l'admin inserisce dati non validi, il sistema invia un messaggio di errore.</li> <li>3b. Se si verifica un problema durante l'eliminazione dell'utente dal database, il sistema invierà un messaggio di errore.</li> </ol>
Pre-condizioni	L'admin deve aver effettuato il login.
Post-condizioni	L'utente è stato eliminato con successo e non può più accedere all'applicazione a meno che non venga effettuata una nuova registrazione.

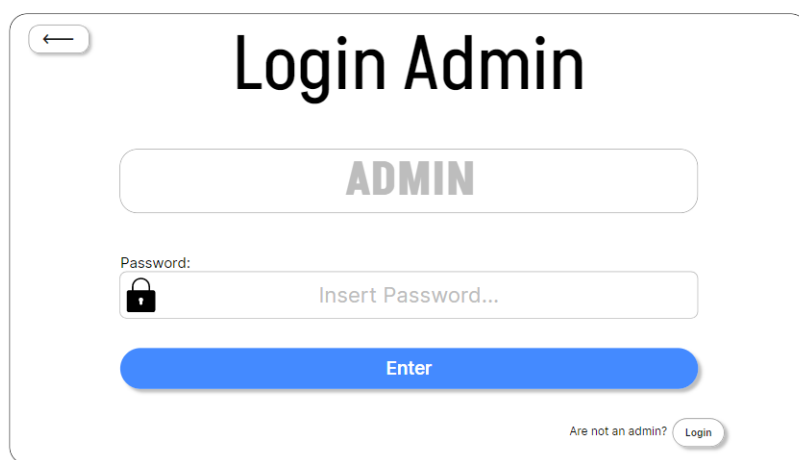
Tabella 5: Template che descrive il caso d'uso del Delete User

Use Case 6	Add Accommodation
Descrizione	L'admin aggiunge un nuovo alloggio al database.
Livello	User Goal
Attori	Admin
Flusso Base	<ol style="list-style-type: none"> <li>1. L'admin inserisce i parametri che caratterizzano l'alloggio.</li> <li>2. L'admin preme il pulsante per effettuare l'aggiunta dell'alloggio.</li> <li>3. Il sistema verifica le informazioni fornite.</li> <li>4. Il sistema registra l'alloggio appena inserito.</li> </ol>
Flusso Alternativo	<ol style="list-style-type: none"> <li>3a. Se si verifica un problema durante la registrazione dell'alloggio nel database, il sistema invia un messaggio di errore.</li> </ol>
Pre-condizioni	L'admin deve aver fatto il login.
Post-condizioni	L'alloggio è stato registrato con successo e sarà disponibile per le successive ricerche degli utenti.

Tabella 6: Template che descrive il caso d'uso dell'Add Accomodation

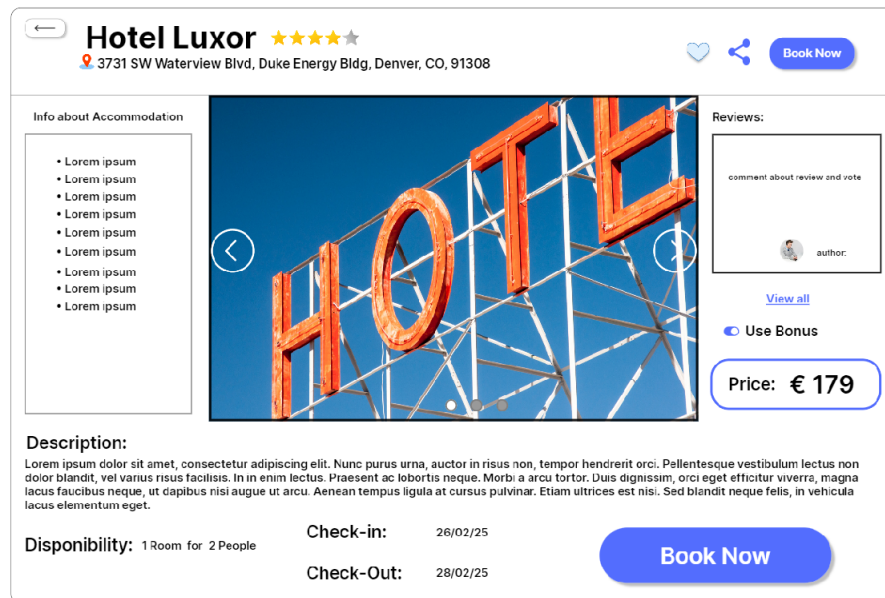
## 2.3 Mockups

Sono riportati alcuni mockup, realizzati con **Lunacy**, relativi ad una possibile interfaccia grafica per l'applicazione:



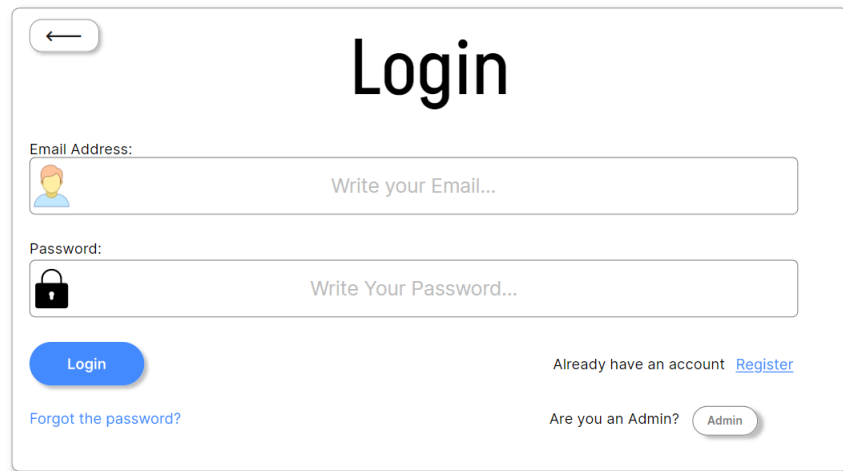
A mockup of an admin login screen. It features a back arrow in the top left corner. The title "Login Admin" is centered at the top. Below it is a large input field with the placeholder text "ADMIN". Underneath is a "Password:" label followed by a lock icon and an input field with the placeholder "Insert Password...". A blue "Enter" button is positioned below the password field. At the bottom right, there is a link "Are not an admin?" followed by a "Login" button.

Figura 3: Mockup per il login effettuato da un admin - MK#1



A mockup of a hotel details page for "Hotel Luxor". The header includes the hotel name with a 5-star rating, the address "3731 SW Waterview Blvd, Duke Energy Bldg, Denver, CO, 91308", a heart icon, a share icon, and a "Book Now" button. The main content area is divided into three sections: "Info about Accommodation" on the left with a list of placeholder text, a large central image of a "HOTEL" sign, and "Reviews" on the right with a placeholder for a review and a "View all" link. Below the image is a "Description" section with placeholder text. At the bottom, there is a "Disponibility" section showing "1 Room for 2 People", "Check-in: 26/02/25", "Check-Out: 28/02/25", and a "Book Now" button. A "Price: € 179" is displayed in a blue box.

Figura 4: Mockup per mostrare in dettaglio un alloggio con la possibilità di prenotarlo - MK#2



← Login

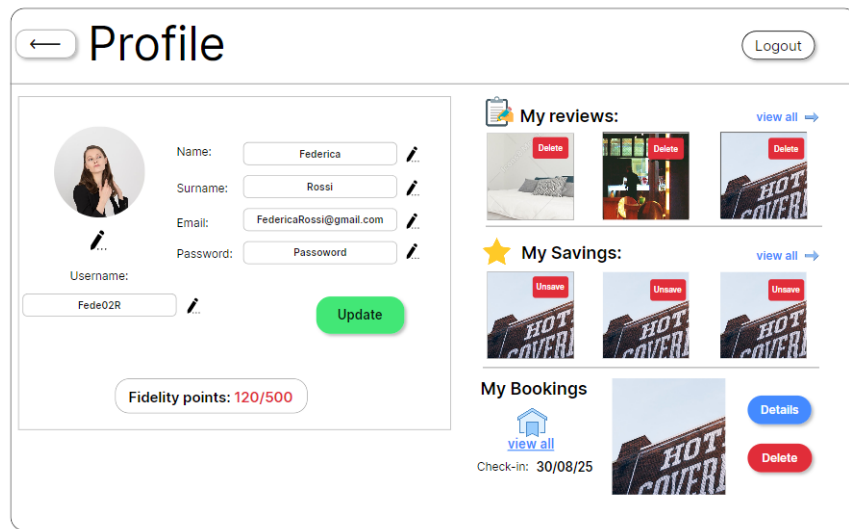
Email Address:

Password:


[Login](#) [Already have an account Register](#)

[Forgot the password?](#) [Are you an Admin? Admin](#)

Figura 5: Mockup per il login effettuato da un utente - MK#3



← Profile Logout



Name:  [✎](#)

Surname:  [✎](#)




Email:  [✎](#)

Password:  [✎](#)




Username:  [✎](#) [Update](#)

Fidelity points: 120/500

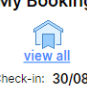

[My reviews:](#) [view all →](#)

 [Delete](#)
 [Delete](#)
 [Delete](#)

[My Savings:](#) [view all →](#)

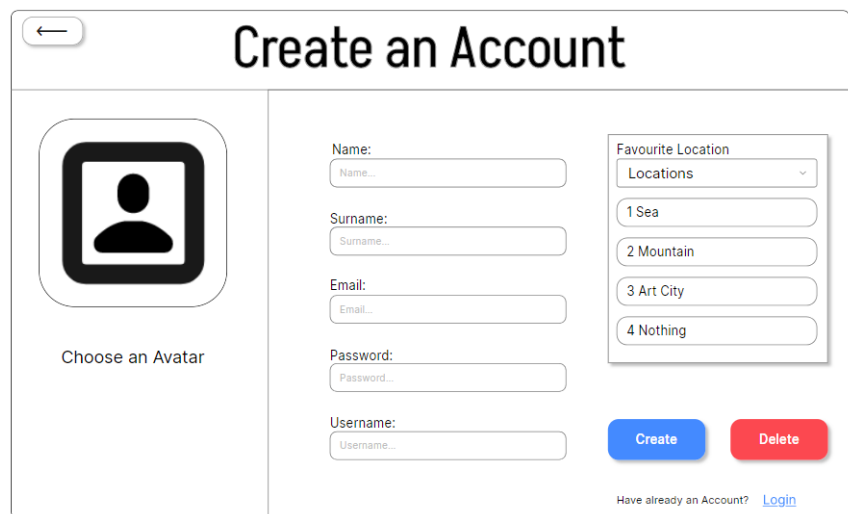
 [Unsave](#)
 [Unsave](#)
 [Unsave](#)

[My Bookings](#)


 [view all](#)
 [Details](#) [Delete](#)

Check-in: 30/08/25

Figura 6: Mockup per mostrare il profilo di un utente - MK#4



← **Create an Account**



Choose an Avatar

**Name:**  
Name...

**Surname:**  
Surname...

**Email:**  
Email...

**Password:**  
Password...

**Username:**  
Username...

**Favourite Location**  
Locations ▾

1 Sea

2 Mountain

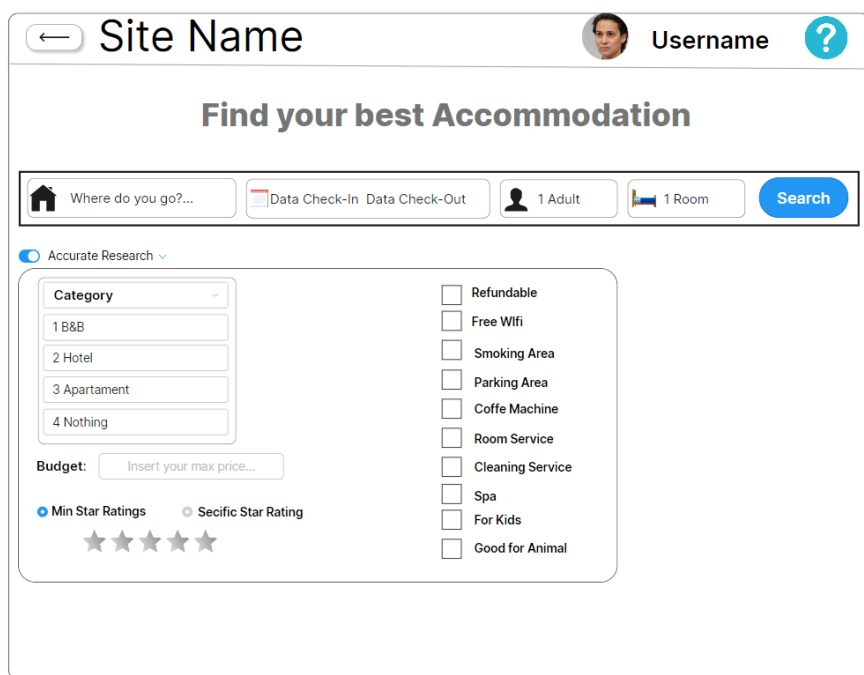
3 Art City



4 Nothing

**Create** **Delete**





Have already an Account? [Login](#)

Figura 7: Mockup per creare un account - MK#5



← **Site Name**  **Username** 

**Find your best Accommodation**

 Where do you go?...
  Data Check-In Data Check-Out
  1 Adult
  1 Room
 **Search**

☒ Accurate Research ▾

**Category** ▾

1 B&B

2 Hotel

3 Apartment

4 Nothing

**Budget:**

☒ Min Star Ratings ☐ Specific Star Rating

★ ★ ★ ★ ★

☐ Refundable

☐ Free Wifi

☐ Smoking Area

☐ Parking Area

☐ Coffe Machine

☐ Room Service

☐ Cleaning Service

☐ Spa

☐ For Kids

☐ Good for Animal

Figura 8: Mockup per effettuare la ricerca attraverso i filtri - MK#6

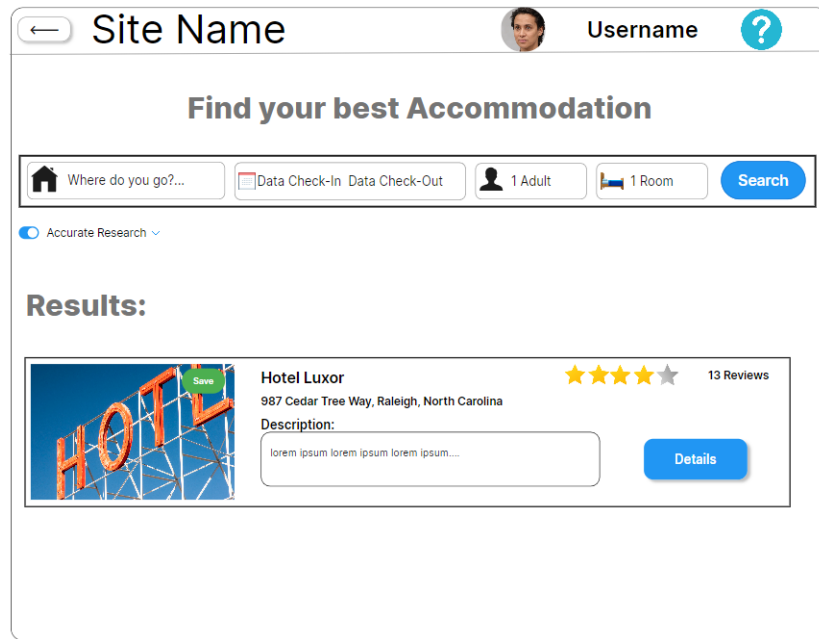


Figura 9: Mockup che mostra i risultati di una ricerca - MK#7

## 2.4 Class Diagram

L'architettura è divisa in 3 package:

- **Domain Model:** consiste nell'insieme di classi che rappresentano i concetti con cui interagisce l'applicazione: **RegisteredUser**, **Review**, **Booking**, **Accommodation**, **SearchParameters** e **SearchParametersBuilder**.

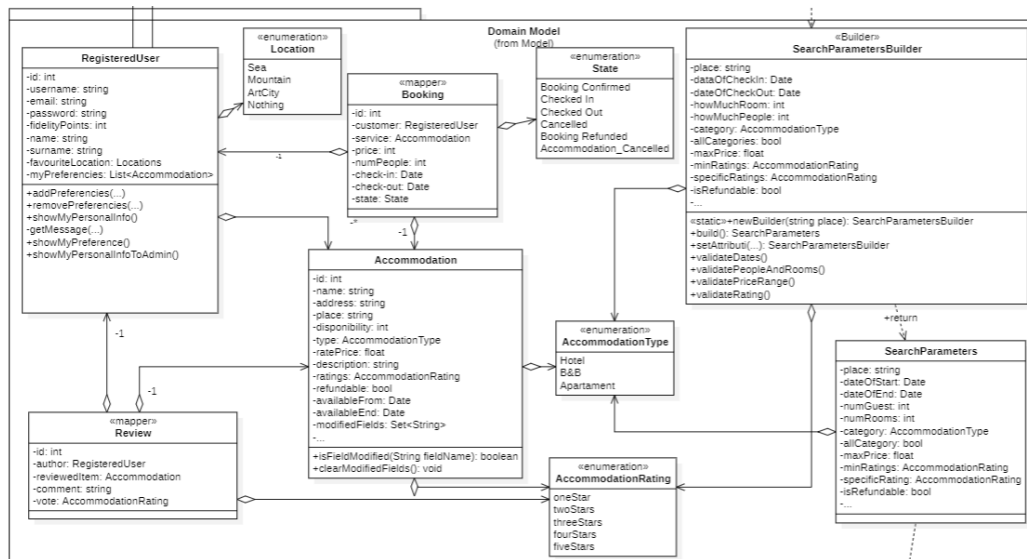


Figura 10: Class Diagram - Domain Model

- **ORM:** è il package che si occupa di gestire la connessione con il database: **UserDAO, BookingDAO, PreferenceDAO, ReviewDAO, AccommodationDAO, DatabaseConnection.**

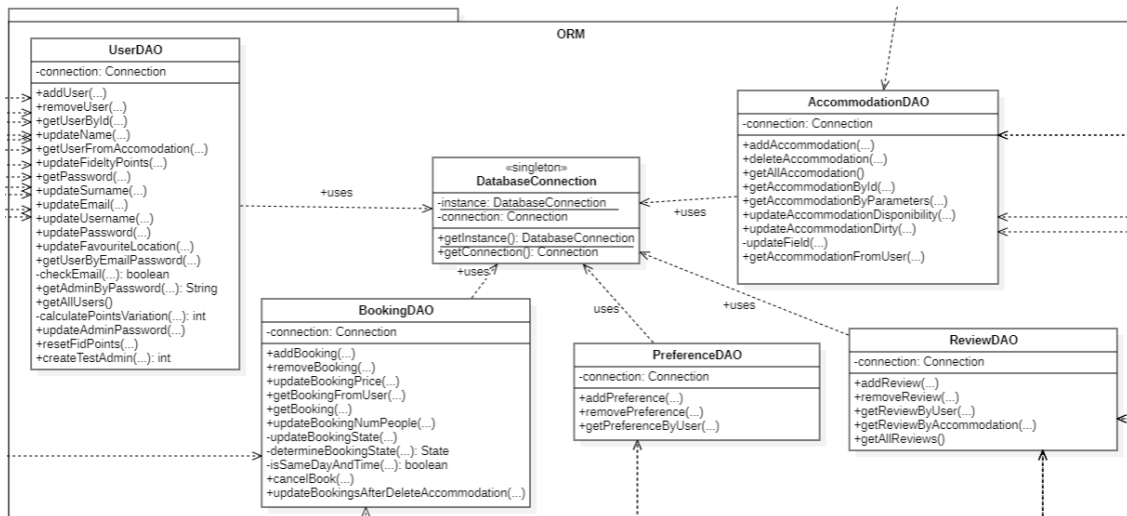


Figura 11: Class Diagram - ORM

- **Business Logic:** è il package che contiene i controller, che sono 4: quello che gestisce l'accesso, la registrazione e un eventuale recupero password (**UserController**), quello che gestisce il profilo dell'utente e una sua eventuale rimozione (**ProfileUserController**), quello che gestisce la logica dell'applicazione (ricerca, prenotazione, recensione) (**ResearchController**) e quello che gestisce le azioni che può effettuare l'Admin (**AdminController**).

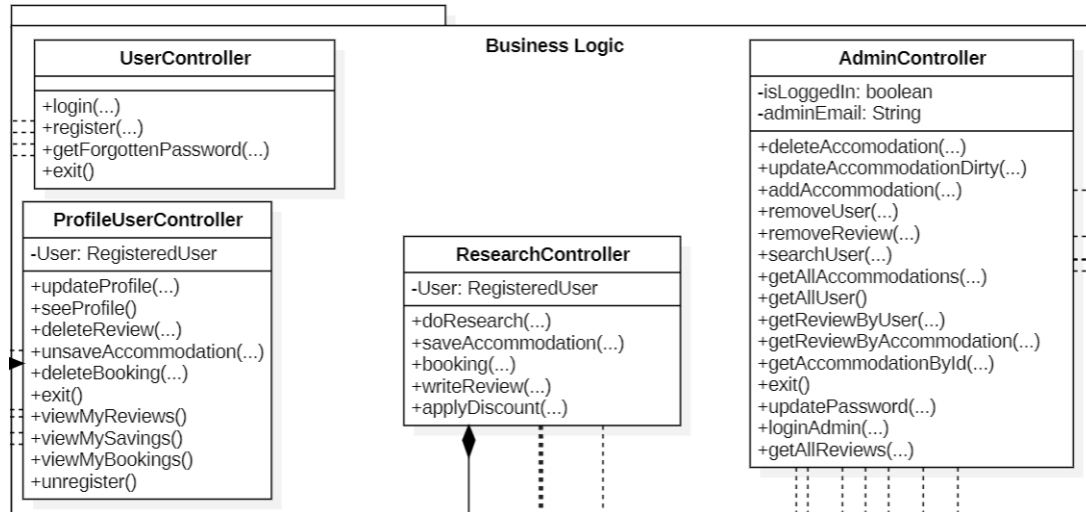


Figura 12: Class Diagram - Business Logic

## 2.5 Dettagli di Progetto

Nell'architettura sono stati usati diversi design pattern:

### 2.5.1 Singleton

Il Singleton è stato utilizzato per garantire che la connessione al database venisse effettuata una singola volta e per evitare conflitti tra connessioni.

```

public class DatabaseConnection {
    // Istanza singleton
    private static DatabaseConnection instance;

    // Connessione al database
    private Connection connection;

    // Costruttore privato per prevenire l'istanziamento diretta
    private DatabaseConnection() {...}

    // Metodo per ottenere l'istanza singleton
    public static synchronized DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }

    // Metodo per ottenere la connessione al database
    public Connection getConnection() { return connection; }
}

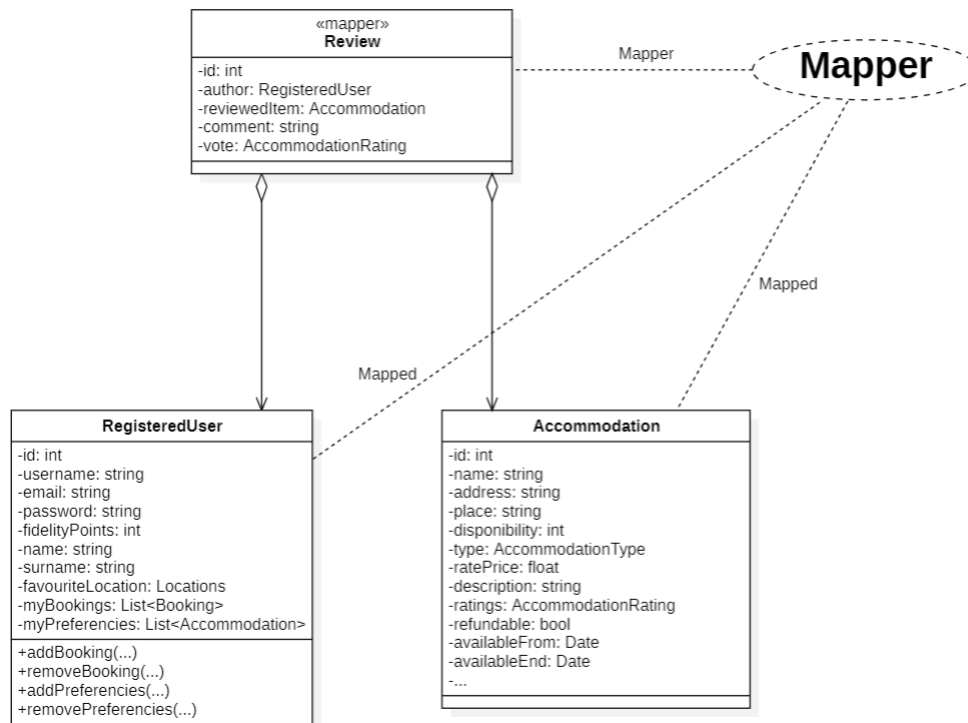
```

Snippet 1: Implementazione Singleton

### 2.5.2 Mapper

Lo scopo del Mapper è quello di creare la relazione tra utenti, recensioni e alloggi, e di creare la relazione tra utenti, prenotazioni e alloggi.





Snippet 2: UML del Mapper

### 2.5.3 Builder Telescoping Constructor

Il Builder Telescoping Constructor viene usato per creare la classe parametri di ricerca che presenta tanti attributi, spesso opzionali, e ne consente una gestione più efficiente. L'unico scopo della classe è quello creare un oggetto di tipo ParametriRicerca e ci riesce grazie ai metodi di setting, che ritornando la classe stessa, consentono di avere una chiara creazione dell'oggetto, attraverso un metodo statico e evitando l'overloading dei costruttori tradizionali (**build**). La classe per definizione presenta tutti gli attributi della classe **SearchParameters** quindi possiede 19 attributi e ognuno di essi ha implementato un setter. Inoltre, per funzionare, creare una classe **SearchParametersBuilder** bisogna fornirgli un valore di tipo String che nel nostro progetto rappresenta il luogo della ricerca (**newBuilder(String place)**). Per evitare che l'utente creasse dei parametri di ricerca invalidi abbiamo inserito nella fase di build dei metodi di validazione che lanciano delle eccezioni quando l'utente sbaglia a inserire un valore. Un esempio può essere la validazione delle date che controlla che l'utente non abbia inserito nella ricerca date passate. Per implementare questa classe

sono state richieste circa 203 linee di codice. Qui sottostante ne riporteremo alcune più significative

```

1 public final class SearchParametersBuilder {
2     private final String place;
3     private LocalDateTime dateOfCheckIn;
4     private LocalDateTime dateOfCheckOut;
5     private Integer howMuchRooms=0;
6     private Integer howMuchPeople=0;
7     private AccommodationType category;
8     private boolean allCategories;
9     private Float maxPrice=0.0f;
10    private AccommodationRating minAccommodationRating;
11    private AccommodationRating specificAccommodationRating;
12    private boolean isRefundable;
13    private boolean haveFreeWifi;
14    private boolean canISmoke;
15    private boolean haveParking;
16    private boolean haveCoffeeMachine;
17    private boolean haveRoomService;
18    private boolean haveCleaningService;
19    private boolean haveSpa;
20    private boolean goodForKids;
21    private boolean canHaveAnimal;
22
23    public SearchParametersBuilder
24        setDateOfCheckIn(LocalDateTime dateOfCheckIn) {
25        this.dateOfCheckIn = dateOfCheckIn;
26        return this;
27    }
28    // ...
29
30    private SearchParametersBuilder(String place) {
31        this.place = place;
32    }
33
34    public static SearchParametersBuilder newBuilder(String
35        place){
36        if(place.trim().isEmpty()){
37            throw new IllegalArgumentException("Place
38                cannot be empty");
39        }
40        return new SearchParametersBuilder(place);
41    }
42
43    public SearchParameters build() {
44
45        // Validazione delle date
46        validateDates();
47
48        // Validazione degli altri parametri
49        validatePeopleAndRooms();

```

```

47         validatePriceRange();
48         validateRating();
49
50         return new SearchParameters(place, dateOfCheckIn,
            dateOfCheckOut, howMuchRooms, howMuchPeople,
            category, allCategories, maxPrice,
            minAccommodationRating,
            specificAccommodationRating, isRefundable,
            haveFreeWifi, canISmoke, haveParking,
            haveCoffeeMachine, haveRoomService,
            haveCleaningService, haveSpa, goodForKids,
            canHaveAnimal);
51     }
52
53     private void validateDates() {
54         LocalDateTime now = LocalDateTime.now();
55
56         if (dateOfCheckIn != null && dateOfCheckOut !=
            null) {
57             // Verifica che le date non siano nel passato
58             if (dateOfCheckIn.isBefore(now)) {
59                 throw new
                    IllegalArgumentException("Check-in date
                        cannot be in the past.");
60             }
61
62             if (dateOfCheckOut.isBefore(now)) {
63                 throw new
                    IllegalArgumentException("Check-out date
                        cannot be in the past.");
64             }
65
66             // Verifica che la data di check-out sia dopo
            // il check-in
67             if (!dateOfCheckOut.isAfter(dateOfCheckIn)) {
68                 throw new
                    IllegalArgumentException("Check-out date
                        must be after check-in date.");
69             }
70
71         } else if (dateOfCheckIn != null || dateOfCheckOut
            != null) {
72             // Se solo una delle due date è specificata
73             throw new IllegalArgumentException("Both
                check-in and check-out dates must be
                provided or both must be null.");
74         }
75     }
76     //...

```

#### 2.5.4 DAO

Il DAO (Data Access Object) è un design pattern che si occupa di separare le classi che si interfacciano al database dall'applicazione. Questo viene fatto per poter meglio implementare il principio di singola responsabilità e aumenta la manutenibilità del codice. Viene implementato nel package ORM.

### 2.6 ER Diagram e Relational Model

Per il database e la sua gestione, è stato realizzato un ER Diagram (Figura 13), e il Relational Model derivato (Figura 14), entrambi realizzati con **Draw.io**. Ci sono state delle scelte progettuali precise, in particolare quella riguardante l'entità **Alloggio**, dove per differenziare i tipi di alloggio è stato usato un attributo al posto di una generalizzazione, dovuto al fatto che nel progetto si faranno uso di informazioni che sono a comune tra i vari alloggi, favorendo accessi più veloci ma a discapito di un notevole spreco di memoria e la presenza di valori nulli. Alla fine sono state definite le seguenti tabelle:

- **User**: rappresenta l'entità **User**.
- **Booking**: rappresenta l'entità **Booking**.
- **Accommodation**: rappresenta l'entità **Accommodation**.
- **Review**: rappresenta la relazione **Review** che avviene tra l'entità **User** e l'entità **Accommodation**.
- **Favourites**: rappresenta la relazione **Like** che avviene tra l'entità **User** e l'entità **Accommodation**.

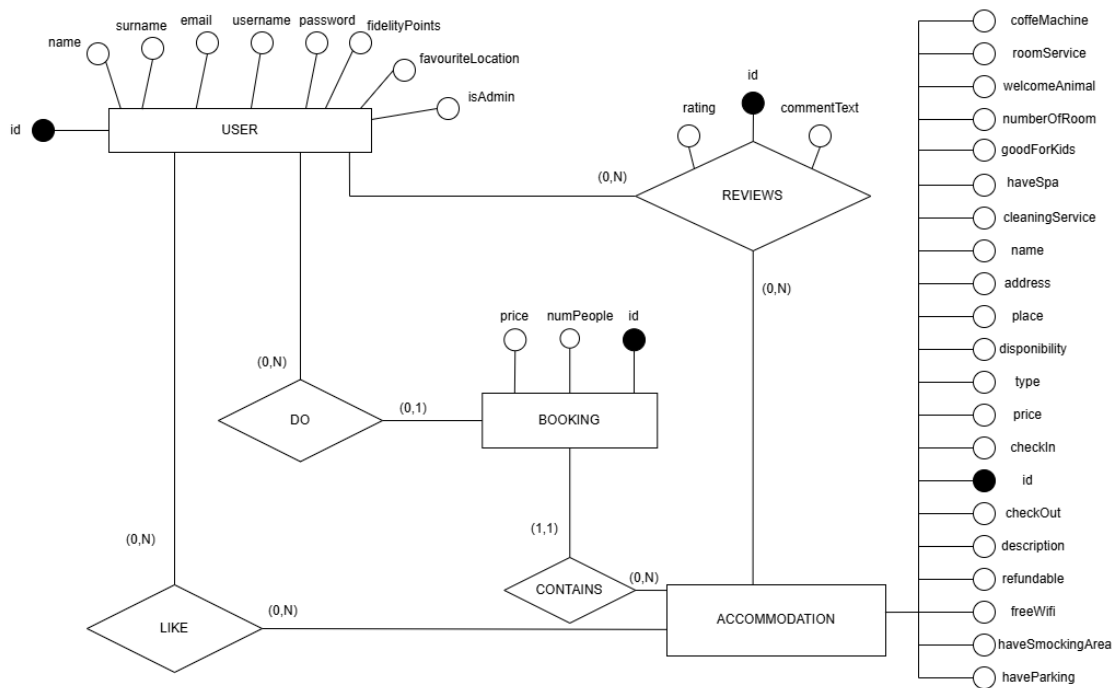


Figura 13: ER Diagram

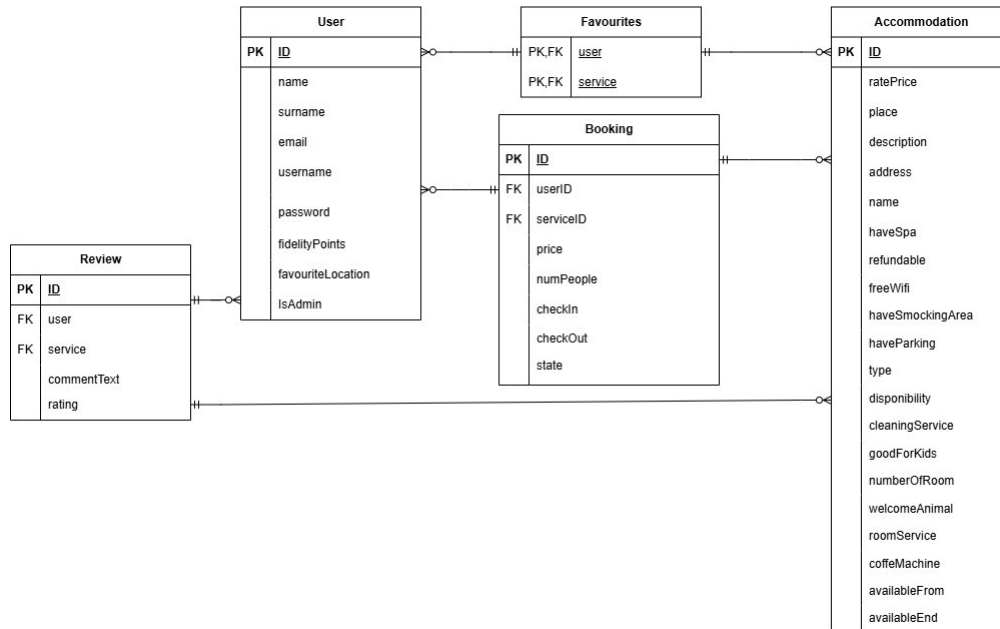


Figura 14: Tables of the database

### 2.6.1 Dettagli implementativi del database

Per gestire l'update del rating dell'alloggio all'interno del database stata optata come soluzione l'utilizzo dei trigger. Il rating dell'alloggio viene aggiornato ogni volta che viene aggiunta o eliminata una sua recensione aggiornandone il valore con la media dei rating delle sue recensioni.

Questa logica stata implementata mediante due trigger SQL associati alla tabella **Reviews**: uno attivato dopo l'inserimento di una nuova recensione (**AFTER INSERT**) e l'altro dopo la sua eliminazione (**AFTER DELETE**). Entrambi i trigger invocano la medesima funzione **update\_accommodation\_rating()**, definita in linguaggio **PL/pgSQL**. La funzione si occupa di calcolare la media di tutti i valori di rating presenti nella tabella **Reviews** per lo specifico alloggio coinvolto. Il risultato, convertito in numero intero mediante la funzione **ROUND**, viene poi utilizzato per aggiornare il campo **rating** della corrispondente voce nella tabella **Accommodation**.

Questo garantisce che ogni variazione delle recensioni si rifletta immediatamente sul rating complessivo visibile all'utente. L'utilizzo di trigger rappresenta una scelta efficace per garantire l'integrità e l'affidabilità del dato, rendendo il sistema più robusto e automatizzato.

```

CREATE OR REPLACE FUNCTION update_accommodation_rating()
RETURNS TRIGGER AS $$
DECLARE
    avg_rating INTEGER;
BEGIN
    SELECT ROUND(AVG(rating))::INTEGER
    INTO avg_rating
    FROM Reviews
    WHERE accommodationId = NEW.accommodationId;

    UPDATE Accommodation
    SET rating = avg_rating
    WHERE id = NEW.accommodationId;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger dopo l'inserimento di una recensione
CREATE TRIGGER trg_update_rating_after_insert
AFTER INSERT ON Reviews
FOR EACH ROW
EXECUTE FUNCTION update_accommodation_rating();

-- Trigger dopo la cancellazione di una recensione
CREATE TRIGGER trg_update_rating_after_delete
AFTER DELETE ON Reviews
FOR EACH ROW
EXECUTE FUNCTION update_accommodation_rating();

```

Figura 15: trigger implementati per l'aggiornamento del rating per gli alloggi.

## 3 Implementazione

### 3.1 Business Logic

È il package che contiene i controller e che espone all'esterno le funzionalità dell'applicazione. È responsabilità dei controller gestire le dipendenze tra gli oggetti creati dai DAO. I controller accedono alle funzionalità del DAO grazie ai loro metodi dove vengono istanziati le classi DAO necessarie. La loro relazione si può rappresentare come la dipendenza d'uso. Qui sotto mostriamo un esempio di come comunica un controller con il DAO:



```

1 public class ResearchController {
2     private RegisteredUser user;
3     //...
4     //metodo per effettuare la ricerca
5     public ArrayList<Accommodation>
6         doResearch(SearchParameters SP) {
7         //istanziamento del DAO necessario
8         AccommodationDAO accommodationDAO = new
9             AccommodationDAO();
10        return
11            accommodationDAO.getAccommodationByParameter(SP);
12    }
13    //...

```

Snippet 4: Frammento di codice del ResearchController implementazione della ricerca degli alloggi

### 3.1.1 UserController

È la classe che si occupa di implementare le funzionalità di un utente generico per l'accesso all'applicazione. Infatti presenta i metodi di login, di registrazione e un eventuale recupera password.

### 3.1.2 ProfileController

È la classe che si occupa di gestire le informazioni del profilo utente e dei servizi di cui ha usufruito (**deleteReview()**, **deleteBooking()**, **unsavedAccommodation()**, **viewMySavings()**, **viewMyReviews()**, **viewMyBookings()**).

### 3.1.3 ResearchController

È la classe che si occupa di fornire i metodi che implementano la logica dell'applicazione. Infatti, permette la ricerca in base a dei parametri (**doResearch()**) e sugli alloggi ricercati permette, a un utente registrato, di effettuare una prenotazione (**booking()**), salvarlo tra i preferiti (**saveAccommodation()**) e scrivere una recensione su quell'alloggio (**writeReview()**). Per funzionare, oltre ai collegamenti ai relativi DAO, questa classe usa il **SearchParametersBuilder** che si trova nel **Domain Model**.

### 3.1.4 AdminController

È la classe che si occupa di implementare le operazioni dell'admin, il quale può leggere, modificare, cancellare e aggiungere alloggi, mentre può solo leggere e cancellare utenti e recensioni.

## 3.2 Domain Model

È il package che rappresenta il modello dei dati e implementa le classi che raffigurano le entità del sistema.

### 3.2.1 RegisteredUser

Contiene le informazioni relative all'utente registrato. Gli attributi della classe sono: id (utilizzato come identificativo), username, email (unica all'interno dell'applicazione), password, nome, cognome, punti fedeltà (che si aggiornano ad ogni acquisto e raggiunta una certa soglia, permette di avere degli sconti), località preferita che indica un genere di esperienza che preferisce, la lista delle prenotazioni effettuate e la lista dei suoi alloggi preferiti.

### 3.2.2 Booking

È la classe che rappresenta l'entità prenotazione, con tutte le informazioni relative ad essa. Possiede un id identificativo, l'acquirente, il servizio, per quante persone è la prenotazione, il prezzo, il check-in, il check-out e lo stato della prenotazione.

### 3.2.3 Accommodation

È la classe che rappresenta l'entità alloggio e tiene conto dei suoi attributi. Molti dei suoi attributi non sono obbligatori, ma dipendono dai servizi che offre l'alloggio. Ha un id (univoco), un nome, un indirizzo, un luogo, quante persone possono alloggiarci, tipo di alloggio (B&B, appartamento e hotel), il prezzo, il periodo di disponibilità, la descrizione di cosa offre, un insieme di stringhe che permette di sapere cosa è stato modificato e diversi parametri aggiuntivi che non sono obbligatori (visualizzabili nelle figure precedenti) .

### 3.2.4 Review

È la classe che rappresenta l'entità recensione. I campi sono id, autore, alloggio recensito, commento e il voto. Consente di stabilire una correlazione tra l'utente e l'alloggio in maniera discreta, senza che tale connessione sia direttamente percepita dagli interessati.

### 3.2.5 SearchParametersBuilder e SearchParameters

Queste classi implementano il design pattern Builder Telescopic Constructor per la creazione dei parametri di ricerca. Il **SearchParametersBuilder** consente una creazione più facile da gestire e da estendere dei parametri di ricerca. Infatti il suo unico scopo è di creare la classe **SearchParameters**. Quest'ultima classe possiede solo gli attributi che poi serviranno alla ricerca all'interno del database. Gli attributi sono per la maggior parte uguali alla classe **Accommodation**. Qui di seguito mostreremo un esempio del suo utilizzo:

```

try {
    SearchParameters sp = SearchParametersBuilder.newBuilder((String) filter[0])
        .setDateOfCheckIn((LocalDateTime) filter[1])
        .setDateOfCheckOut((LocalDateTime) filter[2])
        .setHowMuchRooms((int) filter[3])
        .setHowMuchPeople((int) filter[4])
        .setCategory((AccommodationType) filter[5])
        .setAllCategories((boolean) filter[6])
        .setMaxPrice((float) filter[7])
        .setMinRatingStars((AccommodationRating) filter[8])
        .setSpecificRatingStars((AccommodationRating) filter[9])
        .setRefundable((boolean) filter[10])
        .setHaveFreeWifi((boolean) filter[11])
        .setCanISmoke((boolean) filter[12])
        .setHaveParking((boolean) filter[13])
        .setHaveCoffeeMachine((boolean) filter[14])
        .setHaveRoomService((boolean) filter[15])
        .setHaveCleaningService((boolean) filter[16])
        .setHaveSpa((boolean) filter[17])
        .setGoodForKids((boolean) filter[18])
        .setCanHaveAnimal((boolean) filter[19])
        .build();

    mySearchParameters = sp;
    return rc.doResearch(sp);
} catch (RuntimeException e) {
    System.err.println(e.getMessage());
}

```

Figura 16: Esempio di utilizzo del **SearchParametersBuilder** nel creare **SearchParameters**

### 3.3 ORM

È il package che implementa il design pattern DAO descritto nella sezione 2.5.4. Le classi in questo package permettono alle classi presenti nella **Business Logic** di accedere ai vari dati di loro interesse nel database. Ogni classe possiede un attributo di tipo **Connection**, tipo del **JDBC** che rappresenta la connessione al database ottenuta dalla classe **DatabaseConnection**.

```

1 public class AccommodationDAO {
2     private Connection connection;
3     public AccommodationDAO() {
4         try {
5             this.connection =
6                 DatabaseConnection.getInstance().getConnection();
7         } catch (Exception e) {
8             System.err.println(e.getMessage());
9         }
10    }

```

```
10 //...
```

Snippet 5: Esempio di come una classe DAO ottiene la connessione al db

Per la complessità dell'applicazione stessa le classi DAO devono comunicare tra loro e lo fanno passando dai controller della **Business Logic**. Esistono alcuni metodi che per comunicare non passano dal controller, ma vengono richiamati direttamente dalla classe DAO stessa per una questione di comodità. Un esempio il metodo **getUserByEmailPassword(...)** del **UserDAO**.

Qui sotto mostriamo un esempio di comunicazione passando dai controller:

```
1 // cancella una prenotazione ma non la rimuove e attiva
  tutte le funzioni del caso
2 public void cancelABooking(Booking booking) {
3     BookingDAO bookingDAO=new BookingDAO();
4     bookingDAO.cancelBook(booking);
5     AccommodationDAO accommodationDAO=new
        AccommodationDAO();
6     UserDAO userDAO=new UserDAO();
7     userDAO.updateFidPoints(user, -booking.getPrice());
8     accommodationDAO.updateAccommodationDisponibility(
9         booking.getAccommodation().getId(),
10        booking.getAccommodation().getDisponibility()+1
11        );
12 }
```

Snippet 6: Codice di cancellazione di una prenotazione nel ProfileUserController in cui si usa BookingDAO, UserDAO, AccommodationDAO

### 3.3.1 DatabaseConnection

È la classe che si occupa di gestire la connessione al database per le altre classi DAO tramite il metodo **getConnection()**. Classe implementata usando il design pattern *Singleton* per evitare conflitti tra connessioni.

### 3.3.2 UserDAO

È la classe che si occupa della gestione dei dati degli utenti. Questa classe contiene molti metodi, offrendo la possibilità di aggiungere nuovi utenti e di rimuovere quelli già presenti nel database (rispettivamente **addUser()** e **removeUser()**), la possibilità di recuperare un utente tramite l'id identificativo (**getUserById()**). Infine la classe presenta i metodi che permettono di aggiornare i dati personali di un utente e di eliminarlo.

### 3.3.3 BookingDAO

È la classe che si occupa della gestione dei dati che riguardano le prenotazioni effettuate dagli utenti. La classe mette a disposizione metodi che permettono di aggiungere o rimuovere delle prenotazioni (rispettivamente **addBooking()** e **removeBooking()**), di visualizzare le prenotazioni fatte da uno specifico utente (**getBookingFromUser()**) e di modificare lo stato della prenotazione.

```
1 public Booking addBooking(RegisteredUser user,
2     Accommodation accommodation, LocalDateTime datein,
3     LocalDateTime dateout, int nPeople, int price) {
4     if(accommodation.getDisponibility()==0){
5         throw new RuntimeException("This accommodation is
6             not disponibile");
7     }
8     PreparedStatement preparedStatement=null;
9     try {
10         String query="insert into booking
11             (userid,accommodationid,checkin,checkout,price,
12             numpeople, state) values(?,?,?,?,?,?,?) RETURNING
13             id";
14
15         preparedStatement=connection.prepareStatement(query);
16         preparedStatement.setInt(1, user.getId());
17         preparedStatement.setInt(2, accommodation.getId());
18         preparedStatement.setTimestamp(3,
19             java.sql.Timestamp.valueOf(datein));
20         preparedStatement.setTimestamp(4,
21             java.sql.Timestamp.valueOf(dateout));
22         preparedStatement.setInt(5, price);
23         preparedStatement.setInt(6, nPeople);
24         preparedStatement.setString(7,
25             State.Booking_Confirmed.name());
26         ResultSet rs = preparedStatement.executeQuery();
27         if(rs.next()) {
28             return new Booking( rs.getInt(1), user,
29                 accommodation, price, nPeople, datein,
30                 dateout, State.Booking_Confirmed);
31         }
32     } catch (SQLException e) {
33         DBUtils.printSQLException(e);
34     }finally{
35         DBUtils.closeQuietly(preparedStatement);
36     }
37     return null;
38 }
```

Snippet 7: Query che inserisce un record di booking nel database e che ne restituisce un'istanza nel codice

### 3.3.4 PreferenceDAO

È la classe che si occupa della gestione dei dati che riguardano le liste di alloggi preferiti dagli utenti, i quali posso essere visionati senza dover fare una nuova ricerca. Questa classe contiene i metodi che permettono di aggiungere un nuovo alloggio tra i preferiti (**addPreference()**), di rimuovere un alloggio tra i preferiti (**removePreference()**) e di visualizzare gli alloggi preferiti di uno specifico utente (**getPreferenceByUser()**).

### 3.3.5 ReviewDAO

È la classe che si occupa della gestione delle recensioni scritte sugli alloggi da parte degli utenti. La classe contiene i metodi che permettono di aggiungere nuove recensioni (**addReview()**), di rimuovere le recensioni dall'applicazione (**removeReview()**), e di visualizzare le recensioni scritte da uno specifico utente (**getReviewByUser()**) o visualizzare le tutte le recensioni scritte su uno specifico alloggio (**getReviewByAccommodation()**).

### 3.3.6 AccommodationDAO

È la classe che si occupa della gestione dei dati degli alloggi. Questa classe presenta molti metodi, permettendo di aggiungere nuovi alloggi (**addAccommodation()**), di rimuovere gli alloggi già presenti (**deleteAccommodation()**), di visualizzare tutti gli alloggi (**getAllAccommodation()**), di visualizzare uno nello specifico tramite il suo identificativo (**getAccommodationById()**), questo metodo è molto utile per la gestione degli alloggi da parte dell'admin) e di visualizzare gli alloggi che vengono ricercati tramite l'uso dei filtri (**getAccommodationByParameters()**). Infine sono presenti i 2 metodi che permettono di aggiornare i dati di un alloggio attraverso l'uso di un insieme di stringhe per evitare accessi inutili al database (**updateAccommodationDirty()**, **updateField()**). Qui di seguito verrà riportato un frammento di codice di **getAccommodationByParameters()** data la sua importanza. Purtroppo per una questione di sintesi non verrà riportato tutto, essendo lungo 185 righe di codice, ma saranno presenti solo le righe più importanti che fanno capire la logica e la complessità di tale metodo.

```
1 public ArrayList<Accommodation>
   getAccommodationByParameter(SearchParameters
   searchParameters) {
2     ArrayList<Accommodation> accommodations = new
       ArrayList<>();
3     StringBuilder queryBuilder = new StringBuilder(
4         "SELECT *
5         FROM accommodation a
6         WHERE disponibility > 0"
7     );
8
9     // Lista parametri per PreparedStatement
```

```

10 List<Object> parameters = new ArrayList<>();
11
12 // Aggiunta condizioni in base ai parametri non nulli
13 if (searchParameters.getPlace() != null &&
14     !searchParameters.getPlace().isEmpty()) {
15     queryBuilder.append(" AND a.place ILIKE ?");
16     parameters.add(searchParameters.getPlace());
17 }
18
19 if (searchParameters.getDateOfCheckIn() != null &&
20     searchParameters.getDateOfCheckOut() != null) {
21     queryBuilder.append(
22         " AND a.availablefrom <= ? " + // L'alloggio
23         "    disponibile gia' alla data di check-in
24         " AND a.availableend >= ? " // L'alloggio
25         "    disponibile almeno fino alla data di check-out
26     );
27     parameters.add(java.sql.Timestamp.valueOf(
28         searchParameters.getDateOfCheckIn()));
29     parameters.add(java.sql.Timestamp.valueOf(
30         searchParameters.getDateOfCheckOut()));
31 }
32
33 if (searchParameters.getHowMuchRooms() > 0) {
34     queryBuilder.append(" AND a.numberofroom >= ?");
35     parameters.add(searchParameters.getHowMuchRooms());
36 }
37
38 if (searchParameters.getHowMuchPeople() > 0) {
39     queryBuilder.append(" AND a.maxpeople >= ?");
40     parameters.add(searchParameters.getHowMuchPeople());
41 }
42
43 if (!searchParameters.isAllCategories() &&
44     searchParameters.getCategory() != null) {
45     queryBuilder.append(" AND a.type = ?");
46     parameters.add(searchParameters.getCategory().
47         toString());
48 }
49
50 if (searchParameters.getMaxPrice() > 0) {
51     queryBuilder.append(" AND a.rateprice <= ?");
52     parameters.add(searchParameters.getMaxPrice());
53 }
54
55 if (searchParameters.getMinAccommodationRating() !=
56     null) {
57     queryBuilder.append(" AND a.rating >= ?");
58     parameters.add(searchParameters.getMinAccommodation-
59         Rating().getNumericValue());
60 }

```

```

53     } else if
        (searchParameters.getSpecificAccommodationRating()
         != null) {
54         queryBuilder.append(" AND a.rating = ?");
55         parameters.add(searchParameters.getSpecific-
56             AccommodationRating().getNumericValue());
57     }
58
59     // Aggiunta condizioni per i servizi (solo se true)
60     if (searchParameters.isRefundable()) {
61         queryBuilder.append(" AND a.refundable = TRUE");
62     }
63
64     if (searchParameters.isHaveFreeWifi()) {
65         queryBuilder.append(" AND a.freewifi = TRUE");
66     }
67
68     //... altre condizioni booleane
69
70     // Aggiunto ordinamento per rating (decrescente) e
        prezzo (crescente)
71     queryBuilder.append(" ORDER BY a.rating DESC,
        a.ratePrice ASC");
72
73     // Esecuzione query
74     PreparedStatement ps = null;
75
76     try {
77         ps = connection.prepareStatement(queryBuilder.
78             toString());
79
80         // Imposta tutti i parametri
81         for (int i = 0; i < parameters.size(); i++) {
82             Object param = parameters.get(i);
83             if (param instanceof String) {
84                 ps.setString(i + 1, (String) param);
85             } else if (param instanceof Integer) {
86                 ps.setInt(i + 1, (Integer) param);
87             } else if (param instanceof Float) {
88                 ps.setFloat(i + 1, (Float) param);
89             } else if (param instanceof java.sql.Timestamp)
                {
90                 ps.setTimestamp(i + 1, (java.sql.Timestamp)
                    param);
91             }
92         }
93
94         ResultSet resultSet = ps.executeQuery();
95         //... qui ci sara' la creazione delle istanze alloggio

```



#### Snippet 8: Query dinamica in base alle scelte dell'utente per la ricerca degli alloggi

All'interno del metodo **getAccommodationByParameter()**, viene utilizzato un oggetto **StringBuilder** per la costruzione dinamica della query SQL. **StringBuilder** è una classe di Java progettata per gestire concatenazioni di stringhe in maniera efficiente, evitando la creazione continua di nuove istanze di stringa. In questo contesto, è fondamentale perché permette di aggiungere condizioni alla query solo se i corrispondenti parametri dell'utente sono effettivamente presenti. Ci consente di ottenere una query personalizzata e ottimizzata, migliorando al contempo le prestazioni del sistema e la manutenibilità del codice.

#### 3.3.7 DBUtils

Classe che si occupa di gestire le eccezioni lanciate dai db. Il suo scopo è di fornire una spiegazione chiara dell'eccezione lanciata e di chiudere in modo sicuro, evitando memory leak, i prepared statement (oggetto che rappresenta un precompilato SQL statement). Tale classe presenta solo 2 metodi statici:

- **printSQLException()** che stampa tutti i dettagli dell'eccezione SQL, comprese le eventuali eccezioni concatenate
- **closeQuietly()** che chiude in sicurezza un oggetto AutoCloseable.

Non è stata inserita nel Class Diagram perché il suo scopo è utile solo al fine di debug.

### 3.4 Interfaccia CLI

Per l'applicazione è stata realizzata un'interfaccia a linea di comando, implementata nel file **Main.java**. L'utente, inserendo i vari comandi indicati dal sistema, naviga all'interno dell'applicazione, resa più user friendly grazie all'uso di colori per evidenziare le parole importanti o mancanti. La gestione delle scelte effettuate dall'utente è ottenuta attraverso i costrutti *do-while* e *switch-case*, mentre le varie funzionalità sono implementate in metodi specifici delle classi contenute nella **Business Logic**.

```

public static void userMenu(RegisteredUser registeredUser) throws SQLException, ClassNotFoundException {
    Scanner scanner = new Scanner(System.in);
    ArrayList<Accommodation> accommodations;
    ProfileUserController puc= new ProfileUserController(registeredUser);
    int choice;
    do{
        System.out.println("MENU USER: " +
            "\n1. Manage Profile" +
            "\n2. Research: do an apartment search " +
            "\n3. Log out ");

        choice = scanner.nextInt();

        switch(choice) {
            case 1:{
                profileMenu(registeredUser,puc);
                break;
            }
        }
    }
}

```

Figura 17: implementazione del menu dell'utente.

## 4 Test

Sono stati realizzati i test per verificare il corretto funzionamento del sistema, focalizzandosi principalmente sulle funzionalità della **Business Logic** e del **Dao**. Sono stati realizzati utilizzando la libreria **JUnit**.

### 4.1 Domain Model Test

Sono stati implementati i test più rilevanti per il corretto funzionamento della logica di sistema e dei design pattern. I test implementati sono:

- **AccommodationTest.**
- **SearchParametersBuilderTest.**

### 4.2 Business Logic Test

Per i controller sono stati effettuati dei test su tutte le loro funzioni (test funzionali), prestando attenzione che seguano le direttive dello use case. Ne vengono riportati alcuni.

- Il test sul metodo di login di un utente che vuole dimostrare che funzioni sia nel caso di flusso normale che alternativo.

```

@Test
void login() throws SQLException, ClassNotFoundException {
    //testiamo il login di una persona registrata nel database
    user=userController.register(testEmail, testPassword, testUsername, testName, testSurname, testLocation);
    RegisteredUser loginUser=userController.login(testEmail, testPassword);
    assertNotNull(loginUser);
    assertEquals(testUsername,loginUser.getUsername());
    assertEquals(testPassword,loginUser.getPassword());
    assertEquals(testEmail,loginUser.getEmail());
    assertEquals(testName,loginUser.getName());
    assertEquals(testSurname,loginUser.getSurname());
    assertEquals(testLocation, loginUser.getFavouriteLocations());

    //testiamo il login nel caso di una persona non registrata
    String testEmail2="test2@gmail.com";
    String testPassword2="Test1234!";
    loginUser=userController.login(testEmail2, testPassword2);
    assertNull(loginUser);

    //testiamo il login nel caso uno metta la password errata, ma l'email giusta

    loginUser=userController.login(testEmail, testPassword2);
    //possibilità di recupero password
    assertEquals( expected: -1, loginUser.getId());
    assertEquals(testEmail, loginUser.getEmail());
}

```

Figura 18: test che verifica il login all'interno dell'applicazione -Tst#1.

- Il test sulla registrazione da parte di un utente all'applicazione, tenendo conto che funzioni sia nel flusso normale che non.

```

@Test
void register() throws SQLException, ClassNotFoundException {
    //testiamo il caso che vada tutto bene
    user=userController.register(testEmail, testPassword, testUsername, testName, testSurname, testLocation);
    assertNotNull(user);
    assertEquals(testEmail, user.getEmail());
    assertEquals(testPassword, user.getPassword());
    assertEquals(testUsername, user.getUsername());
    assertEquals(testName, user.getName());
    assertEquals(testSurname, user.getSurname());
    assertEquals(testLocation, user.getFavouriteLocations());

    //testiamo il caso che un utente provi a registrarsi con un email già usata
    RegisteredUser user2=userController.register(testEmail, password: "test", username: "test", testName, testSurname, testLocation);
    assertNull(user2); // non viene registrato

    //testiamo il caso che inserisca degli spazi al posto dell'email
    user2=userController.register( email: " ", password: "test2", username: "test", testName, testSurname, testLocation);
    assertNull(user2); // non viene registrato
}

```

Figura 19: test che verifica la registrazione all'interno dell'applicazione  
-Tst#2.

Qui di seguito vengono mostrati tutti i risultati dei test dei metodi della Business Logic:

✓ AdminControllerTest	1 sec 191 ms
✓ loginAdmin()	745 ms
✓ deleteAccommodation()	69 ms
✓ removeReview()	31 ms
✓ addAccommodation()	14 ms
✓ updateAccommodation()	31 ms
✓ searchUser()	23 ms
✓ getAllReview()	98 ms
✓ getReviewByUser()	37 ms
✓ getAllUser()	44 ms
✓ getReviewByAccommodation()	23 ms
✓ changePassword()	9 ms
✓ removeUser()	41 ms
✓ getAccommodationById()	8 ms
✓ getAllAccommodation()	18 ms

Figura 20: i test per AdminController.

✓ ProfileUserControllerTest	72 ms
✓ updateProfile()	10 ms
✓ unSaveAccommodation()	10 ms
✓ removeReview()	10 ms
✓ getReviewsByUser()	7 ms
✓ unRegister()	6 ms
✓ viewMyBookings()	4 ms
✓ removeBooking()	13 ms
✓ cancelABooking()	12 ms

Figura 21: i test per ProfileUserController.

✓ ResearchControllerTest	120 ms
✓ doResearch()	45 ms
✓ booking()	26 ms
✓ applyDiscount()	14 ms
✓ saveAccommodation()	12 ms
✓ writeReview()	12 ms
✓ getReviews()	11 ms

Figura 21: i test per ResearchController.

### 4.3 ORM Test

Per ogni classe del DAO sono stati effettuati dei test strutturali in modo da assicurarsi che i metodi si relazionino correttamente con il database. Questo controllo viene effettuato attraverso la cattura delle eccezioni in caso di errore. Ne vengono riportati alcuni.

- Il test che consente di ottenere le recensioni scritte da un determinato utente, identificato tramite il suo id. Per dimostrarne il funzionamento, viene inserita una recensione di prova, che viene poi visualizzata a schermo.

```
@Test  * lorenzo pennelli *
void getReviewByUser() {
    ArrayList<Review> reviews=assertDoesNotThrow()->reviewDAO.getReviewByUser(registeredUser));
    assertNotNull(reviews);
    assertTrue(reviews.isEmpty());

    // aggiungiamo delle recensioni per test
    reviewDAO.addReview(registeredUser, accommodationDAO.getAccommodationByID(accommodationId),
        content: "test test test", AccommodationRating.FourStar);
    reviews=assertDoesNotThrow()->reviewDAO.getReviewByUser(registeredUser));
    assertNotNull(reviews);
    assertFalse(reviews.isEmpty());
}
```

Figura 22: Il test verifica la corretta acquisizione delle recensioni tramite l'id dell'utente.

- Il test che consente di salvare un alloggio negli alloggi preferiti.

```
@Test  * lorenzo pennelli
void save() throws SQLException, ClassNotFoundException {
    assertDoesNotThrow(() -> preferenceDAO.save(registeredUser.getId(),accommodationId));

    registeredUser =userController.login(testEmail, testPassword);
    assertFalse(registeredUser.getMyPreferences().isEmpty());
}
```

Figura 23: Il test di salvataggio degli alloggi preferiti.

## 4.4 Risultati dei test



The screenshot displays a list of test results in a table format. The table has two columns: the first column lists the test names, each preceded by a green checkmark and a right-pointing arrow, and the second column shows the execution time for each test in milliseconds. The total execution time for all tests is shown at the top right as 1 sec 424 ms.

✓ test	1 sec 424 ms
> ✓ DatabaseConnectionTest	567 ms
> ✓ AdminControllerTest	293 ms
> ✓ ReviewDAOTest	72 ms
> ✓ ResearchControllerTest	95 ms
> ✓ PreferenceDAOTest	30 ms
> ✓ AccommodationTest	
> ✓ UserDAOTest	148 ms
> ✓ ProfileUserControllerTest	69 ms
> ✓ UserControllerTest	16 ms
> ✓ SearchParametersBuilderTest	7 ms
> ✓ AccommodationDAOTest	72 ms
> ✓ BookingDAOTest	55 ms

Figura 24: risultati dei test.