



UNIVERSITÀ DI FIRENZE  
DIPARTIMENTO DI INGEGNERIA INFORMATICA

Corso di Laurea Triennale in  
Ingegneria Informatica

Giugno 2025

**AroundTrip: Piattaforma  
Organizzativa per Viaggiatori e Guide**

Professore:

Vicario Enrico

Studenti:

Postiferi Matteo

Impicciatore Francesco

Nicoli Castiglia Edoardo

# Indice

<b>1</b>	<b>Introduzione e Analisi dei Requisiti</b>	<b>1</b>
1.1	Statement . . . . .	1
1.2	Attori Principali . . . . .	1
1.3	Possibili Estensioni e Sviluppi Futuri . . . . .	2
<b>2</b>	<b>Progettazione del Sistema</b>	<b>3</b>
2.1	Diagrammi dei Casi d'Uso . . . . .	3
2.2	Template Dettagliati dei Casi d'Uso . . . . .	5
2.3	Mockup delle Interfacce Utente . . . . .	12
2.4	Database . . . . .	16
2.4.1	Modello E-R . . . . .	16
2.4.2	Modello Relazionale . . . . .	18
<b>3</b>	<b>Struttura del Progetto</b>	<b>19</b>
3.1	Domain Model . . . . .	20
3.2	Business Logic . . . . .	22
3.3	DAO . . . . .	24
<b>4</b>	<b>Implementazione</b>	<b>25</b>
4.1	Project Structure . . . . .	25
4.2	Implementazione della Logica di Business . . . . .	26
4.2.1	Package controller . . . . .	26
4.2.2	Service . . . . .	26
4.3	Implementazione della logica di DAO . . . . .	29
4.4	Implementazione del Database . . . . .	32
<b>5</b>	<b>Strategia di Testing</b>	<b>33</b>
5.1	Unit Test . . . . .	33
5.2	Integration Test . . . . .	34

## Elenco delle figure

1	Use Case Diagram 1 . . . . .	3
2	Use Case Diagram 2 . . . . .	4
3	Use Case Diagram 3 . . . . .	4
4	Use Case Diagram 4 . . . . .	4
5	Use Case Template - Leave a Review . . . . .	5
6	Use Case Template - Book Trip . . . . .	6
7	Use Case Template - View Trips . . . . .	7
8	Use Case Template - Send Applications . . . . .	8
9	Use Case Template - Select Skills . . . . .	9
10	Use Case Template - Track Application Status . . . . .	10
11	Use Case Template - Create Trips . . . . .	11
12	Mockup - Log in . . . . .	12
13	Mockup - Choose Role . . . . .	12
14	Mockup - View Trips . . . . .	13
15	Mockup Add your Skill . . . . .	13
16	Mockup - Trip Details (Guide) . . . . .	14
17	Mockup - Trip Details (Traveler) . . . . .	14
18	Mockup - Leave Review . . . . .	15
19	Mockup - Send your Application . . . . .	15
20	Diagramma E-R . . . . .	16
21	Panoramica generale UML . . . . .	19
22	Domain Model . . . . .	21
23	Business Logic . . . . .	23
24	DAO . . . . .	24
25	Project Structure . . . . .	25
26	Strategy . . . . .	28
27	Strategy - Classe : TravelerFilter . . . . .	28
28	Strategy - Classe : ViewTripService . . . . .	29
29	Metodi in ConcreteTripDAO . . . . .	31
30	Creazione Tabelle su database . . . . .	32
31	Creazione Tabelle su database . . . . .	32
32	Test classe BookingService . . . . .	34
33	Test classe ApplicationDAOIT . . . . .	35
34	Classe ApplicationDAOIT . . . . .	35
35	Test classe ApplicationDAOIT . . . . .	36

# 1 Introduzione e Analisi dei Requisiti

## 1.1 Statement

Il progetto nasce dall'esigenza di proporre una soluzione più efficace rispetto alle piattaforme tradizionali per l'organizzazione di viaggi. In un contesto in cui il turismo si fa sempre più standardizzato e impersonale, abbiamo avvertito la necessità di valorizzare la componente umana e la condivisione diretta di esperienze tra persone.

Questa applicazione web è stata progettata per mettere al centro la collaborazione tra utenti: chiunque può scegliere se candidarsi come Guide, offrendo il proprio tempo, le proprie conoscenze e le proprie passioni per accompagnare altri utenti, oppure partecipare come Traveler, alla ricerca di viaggi autentici guidati da persone che conoscono davvero il territorio.

Le guide possono creare un profilo dettagliato, aggiungere le proprie competenze (*skill*) e candidarsi esclusivamente ai viaggi per cui si sentono preparate, garantendo così un abbinamento mirato tra domanda e offerta. I traveler possono consultare, filtrare e prenotare i viaggi disponibili, affidandosi a guide selezionate e contribuendo a migliorare il servizio attraverso recensioni sulle esperienze vissute.

L'obiettivo principale è quello di offrire uno strumento che renda possibile un'esperienza di viaggio più personalizzata e partecipata, superando la logica delle offerte preconfezionate. In questo modo, la piattaforma diventa uno strumento unico per vivere, organizzare e condividere viaggi in modo collaborativo e coinvolgente.

## 1.2 Attori Principali

I principali attori che interagiscono con la piattaforma sono i seguenti:

- **User:**

L'utente User è colui che si registra per la prima volta sulla piattaforma e accede tramite le proprie credenziali. Una volta effettuato il login, può ogni volta scegliere se entrare nell'interfaccia del viaggiatore (*Traveler*) o della guida (*Guide*), assumendo così uno dei due ruoli operativi principali a seconda delle proprie esigenze o preferenze.

- **Viaggiatore (Traveler):**

Il viaggiatore è l'utente che utilizza la piattaforma per cercare, prenotare e partecipare ai viaggi proposti dall'agenzia e accompagnati dalle guide. Può esplorare le diverse proposte disponibili, scegliere in base alle proprie preferenze, gestire le proprie prenotazioni e lasciare recensioni sulle esperienze vissute e sulle guide che lo hanno accompagnato.

- **Guida (Guide):**

La guida è l'utente che mette a disposizione il proprio tempo, le proprie conoscenze e competenze per accompagnare i viaggiatori nei viaggi creati dall'agenzia. Può completare il proprio profilo con skill specifiche, candidarsi per accompagnare determinati viaggi, essere assegnata dall'agenzia ai viaggi più adatti alle sue capacità e ricevere feedback dai viaggiatori.

- **Agenzia:**

L'agenzia rappresenta l'attore responsabile della creazione, gestione e modifica dei viaggi sulla piattaforma. Si occupa di inserire nuove proposte di viaggio, aggiungere attività per ciascun viaggio, valutare e accettare le candidature delle guide, e garantire che ogni viaggio disponga del numero adeguato di accompagnatori qualificati.

### 1.3 Possibili Estensioni e Sviluppi Futuri

Durante la realizzazione del progetto abbiamo individuato alcune possibili estensioni che potrebbero arricchire ulteriormente la piattaforma nelle sue evoluzioni future:

- **Integrazione di una chat tra utenti:** abbiamo pensato all'implementazione di un sistema di messaggistica interna, in modo che i partecipanti allo stesso viaggio possano comunicare facilmente tra loro e coordinarsi prima, durante e dopo l'esperienza.
- **Metodo di pagamento integrato:** abbiamo considerato l'aggiunta di un sistema di pagamento sicuro direttamente all'interno dell'applicazione, così da semplificare la gestione delle prenotazioni e offrire maggiore tutela ai viaggiatori.

## 2 Progettazione del Sistema

Durante la fase di progettazione del sistema, sono stati utilizzati diversi strumenti:

- **StarUML**: per la creazione dei diagrammi dei casi d'uso e dei diagrammi UML.
- **dbdiagram.io** : per gli schemi entità-relazione.
- **Figma**: per i mockup delle interfacce utente.

### 2.1 Diagrammi dei Casi d'Uso

Data la descrizione dettagliata del sistema fin qui esposta e avendo individuato i tre attori principali — Guide, Traveler e Agency — abbiamo identificato i seguenti Casi d'Uso :

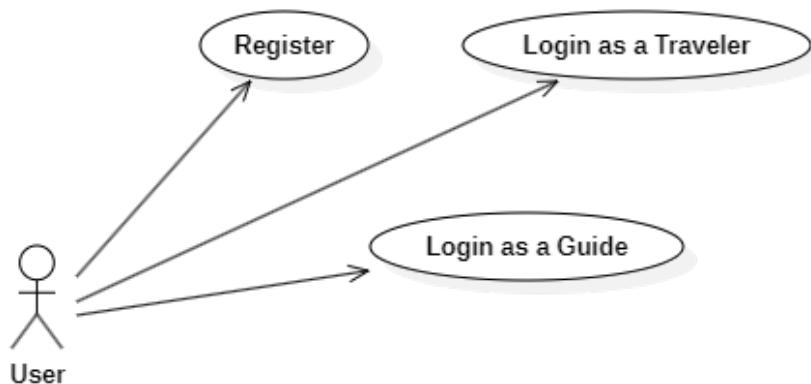


Figura 1: Use Case Diagram 1

## Progettazione del Sistema

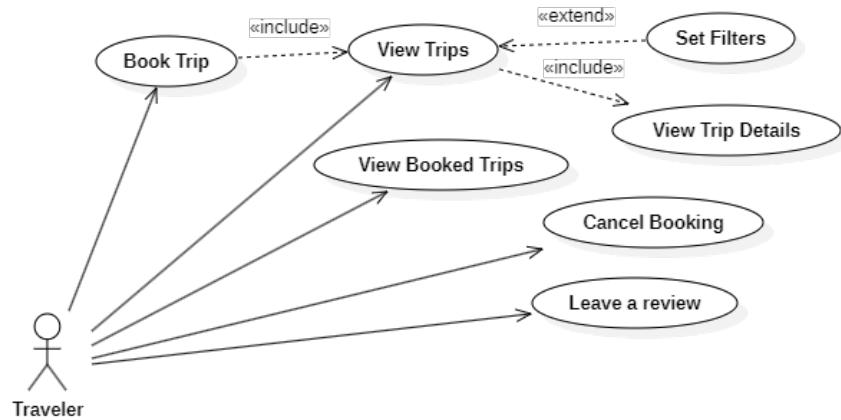


Figura 2: Use Case Diagram 2

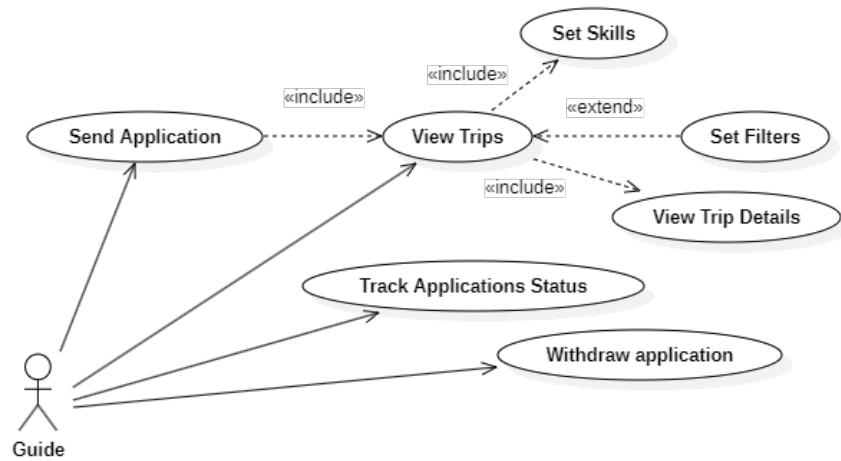


Figura 3: Use Case Diagram 3

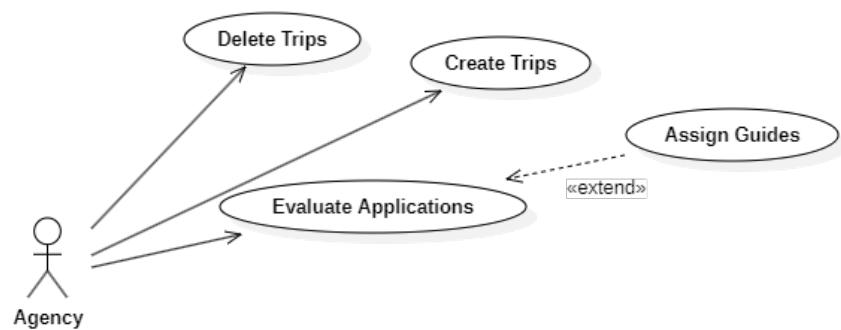


Figura 4: Use Case Diagram 4

## 2.2 Template Dettagliati dei Casi d'Uso

Sono stati selezionati e descritti sette casi d'uso che rappresentano i flussi operativi più significativi all'interno della piattaforma. La scelta di concentrarsi su questi specifici scenari nasce dall'esigenza di definire in modo chiaro e mirato le interazioni fondamentali tra utenti e sistema, così da guidare efficacemente le fasi successive di progettazione e implementazione.

In particolare, sono stati elaborati due use case dedicati alle funzionalità del viaggiatore, tre relativi alle operazioni che può svolgere la guida, uno che riguarda un'interazione comune a entrambi e uno specifico per l'agenzia. Questa suddivisione riflette la struttura a ruoli del sistema e consente di coprire in modo coerente i comportamenti attesi per ciascun attore.

UC-1	Leave a Review
Descrizione	Il viaggiatore può lasciare una recensione sia sul viaggio nel suo complesso che sulla guida. Una volta inserita, il sistema aggiorna automaticamente la valutazione media del viaggio e/o della guida.
Livello	Utente
Attori principali	Viaggiatore
Precondizioni	Il viaggiatore deve aver partecipato al viaggio e può scrivere la recensione solo dopo che il viaggio è terminato.
Flusso principale	<ol style="list-style-type: none"> <li>1. Al termine del viaggio, il sistema offre al viaggiatore la possibilità di lasciare una recensione.</li> <li>2. Il viaggiatore seleziona una valutazione numerica da 1 a 5.</li> <li>3. Facoltativamente, può inserire un commento testuale.</li> <li>4. Il viaggiatore invia la recensione e il sistema la registra.</li> <li>5. Il sistema aggiorna la valutazione media del viaggio e/o della guida.</li> </ol>
Flusso alternativo	1A Se l'utente ignora la richiesta nessuna recensione viene registrata. 3A Se l'utente non inserisce alcun commento il sistema salva solo il punteggio numerico.
Postcondizioni	La recensione è memorizzata e associata al viaggio e alla guida.

Figura 5: Use Case Template - Leave a Review

## Progettazione del Sistema

---

UC-2	Book Trip
Descrizione	Il viaggiatore può prenotare un viaggio disponibile. La prenotazione viene poi salvata nel sistema e collegata al profilo del viaggiatore.
Livello	Utente
Attori principali	Viaggiatore
Precondizioni	Il viaggiatore deve aver effettuato l'accesso e visualizzato i dettagli del viaggio. Il viaggio deve essere ancora prenotabile.
Flusso principale	<ol style="list-style-type: none"> <li>1. Il viaggiatore seleziona un viaggio tra quelli disponibili.</li> <li>2. Il viaggiatore clicca su "View Details" per avere una panoramica completa del viaggio.</li> <li>3. Il viaggiatore clicca su "Book trip"</li>   <li>4. Il sistema chiede all'utente di selezionare il numero di posti da prenotare (fino al massimo disponibile).</li> <li>5. Il viaggiatore seleziona il numero di posti.</li> <li>6. Il sistema registra la prenotazione e la collega al profilo dell'utente.</li> </ol>
Flusso alternativo	5A Se non ci sono sufficienti posti disponibili per soddisfare la richiesta del viaggiatore, il sistema mostra un messaggio d'errore e chiede di selezionare un numero valido.
Postcondizioni	La prenotazione è salvata e visibile nella sezione "View My Bookings".

Figura 6: Use Case Template - Book Trip

UC-3	View Trips
Descrizione	L'utente autenticato può consultare l'elenco dei viaggi disponibili nel sistema e, optionalmente, filtrarli.
Livello	Utente
Attori principali	Guida, Viaggiatore
Precondizioni	Deve esistere almeno un viaggio disponibile. Se l'utente è una guida deve aver inserito le proprie skill.
Flusso principale	<ol style="list-style-type: none"> <li>1. Il sistema mostra l'elenco completo dei viaggi disponibili.</li> <li>2. L'utente può applicare filtri in base alle proprie preferenze (per destinazione, data, durata, ecc.).</li> <li>3. Il sistema aggiorna i risultati in base ai filtri selezionati.</li> <li>4. L'utente seleziona un viaggio per visualizzarne i dettagli.</li> </ol>
Flusso alternativo	<p>2A Se l'utente non imposta alcun filtro il sistema mostra tutti i viaggi disponibili</p> <p>3A Se i filtri impostati non restituiscono risultati il sistema mostra un messaggio del tipo "Nessun viaggio trovato con i criteri selezionati".</p>
Postcondizioni	L'utente dispone dell'elenco dei viaggi disponibili, filtrato in base ai criteri selezionati, ed è pronto a visualizzare i dettagli del viaggio prescelto.

Figura 7: Use Case Template - View Trips

## Progettazione del Sistema

---

UC-4	Send Applications
Descrizione	La guida può candidarsi per uno dei viaggi disponibili.
Livello	Utente
Attori principali	Guida
Precondizioni	La guida deve essere autenticata, deve avere selezionato le proprie competenze e non può essersi già candidata in precedenza.
Flusso principale	<ol style="list-style-type: none"> <li>1. La guida seleziona il viaggio di interesse e accede ai dettagli.</li> <li>2. Clicca sul pulsante "Send Application".</li> <li>3. La guida compila il form per l'application</li> <li>4. Il sistema registra la richiesta di candidatura.</li> </ol>
Flusso alternativo	<p>2A Se la guida ha già inviato una candidatura per questo viaggio il sistema mostra un messaggio e impedisce l'invio.</p> <p>3A Se il form non è stato compilato correttamente, la guida riceve un messaggio di errore</p>
Postcondizioni	La candidatura è registrata nel sistema con stato iniziale PENDING, in attesa di essere valutata dall'agenzia.

Figura 8: Use Case Template - Send Applications

UC-5	Select Skills
Descrizione	La guida seleziona le proprie competenze da un elenco predefinito, che verranno usate per filtrare i viaggi.
Livello	Utente
Attori principali	Guida
Precondizioni	L'utente ha eseguito l'accesso come guida.
Flusso principale	<ol style="list-style-type: none"> <li>1. Il sistema mostra un elenco di competenze disponibili.</li> <li>2. La guida seleziona una o più competenze.</li> <li>3. Il sistema aggiorna il profilo della guida</li> </ol>
Flusso alternativo	2A Se la guida non seleziona alcuna competenza, il sistema chiede di selezionarne almeno una prima di procedere.
Postcondizioni	Il sistema può ora confrontare le competenze della guida con i requisiti richiesti dai viaggi.

Figura 9: Use Case Template - Select Skills

## Progettazione del Sistema

---

UC-6	Track Applications Status
Descrizione	La guida può visualizzare le candidature precedentemente inviate, controllandone lo stato attuale.
Livello	Utente
Attori principali	Guida
Precondizioni	La guida è autenticata e ha inviato almeno una candidatura.
Flusso principale	<ol style="list-style-type: none"><li>1. La guida accede alla sezione "Le mie candidature".</li><li>2. Il sistema mostra un elenco dei viaggi per cui la guida si è candidata.</li><li>3. La guida può cliccare su un viaggio per vedere i dettagli della candidatura, tra cui lo stato.</li></ol>
Flusso alternativo	2A Se non è presente alcuna candidatura il sistema mostra il messaggio: "Non hai ancora inviato alcuna candidatura."
Postcondizioni	La guida può decidere se ritirare una candidatura o attendere una risposta.

Figura 10: Use Case Template - Track Application Status

UC-7	Create Trips
Descrizione	L'agenzia può creare un nuovo viaggio specificando informazioni come destinazione, durata, descrizione, attività pianificate e competenze richieste per le guide.
Livello	Sistema
Attori principali	Agenzia
Precondizioni	Null
Flusso principale	<ol style="list-style-type: none"><li>1. L'agenzia clicca su "Create Trip".</li><li>2. L'agenzia inserisce i dati riguardanti il viaggio</li></ol>
Flusso alternativo	
Postcondizioni	Il viaggio è stato correttamente registrato nel sistema ed è ora visibile a viaggiatori e guide.

Figura 11: Use Case Template - Create Trips

## 2.3 Mockup delle Interfacce Utente

Di seguito si riportano i mockups creati con il fine di esplicitare nella maniera più immediata e comprensibile possibile i casi d'uso precedentemente riportati.

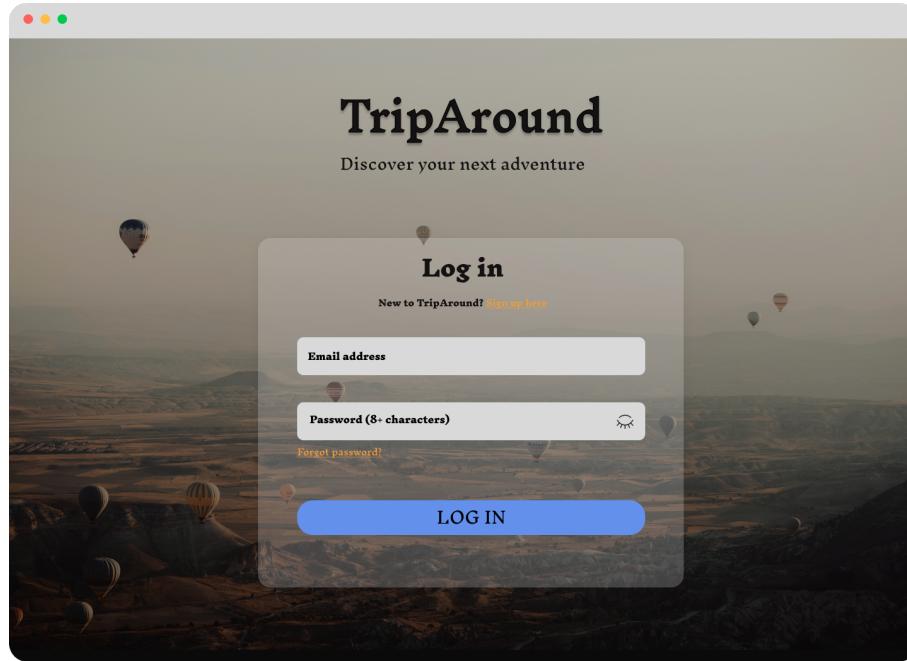


Figura 12: Mockup - Log in

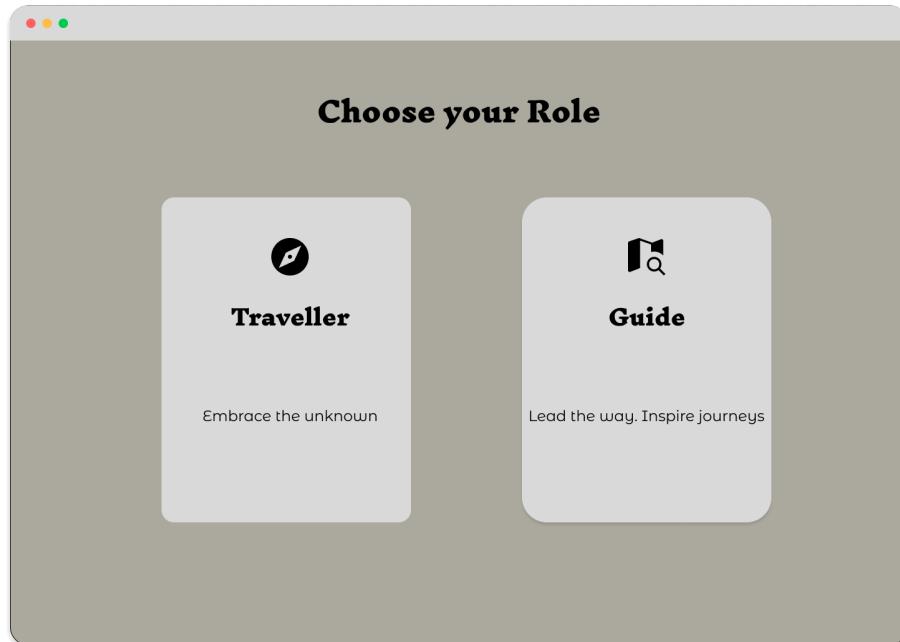


Figura 13: Mockup - Choose Role

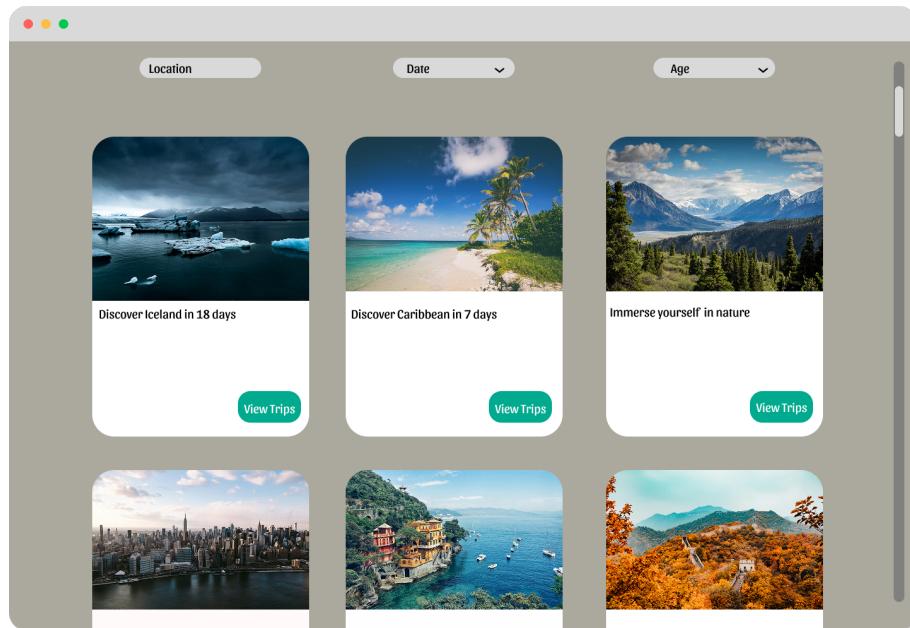


Figura 14: Mockup - View Trips

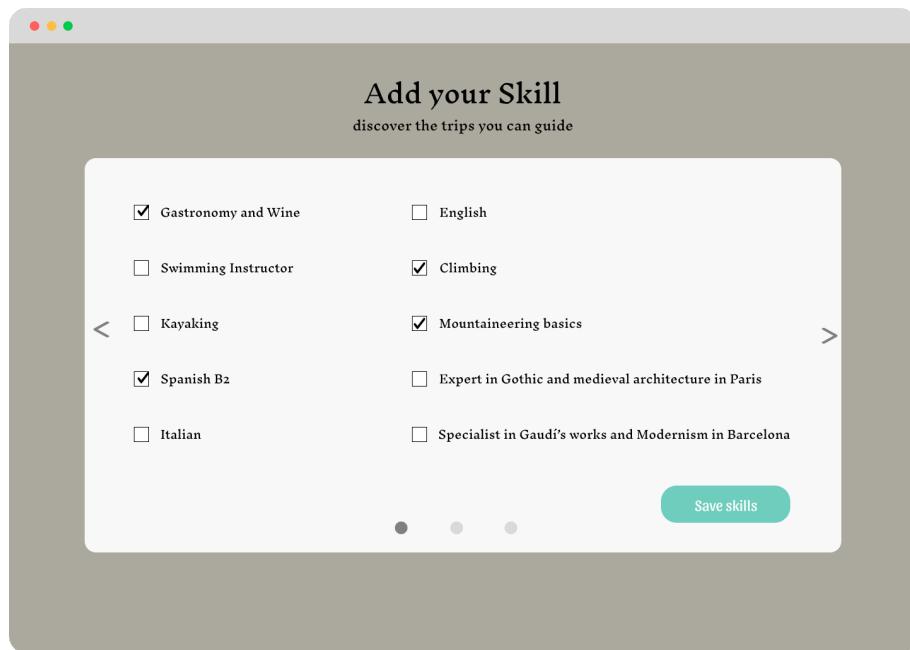


Figura 15: Mockup Add your Skill

## Progettazione del Sistema

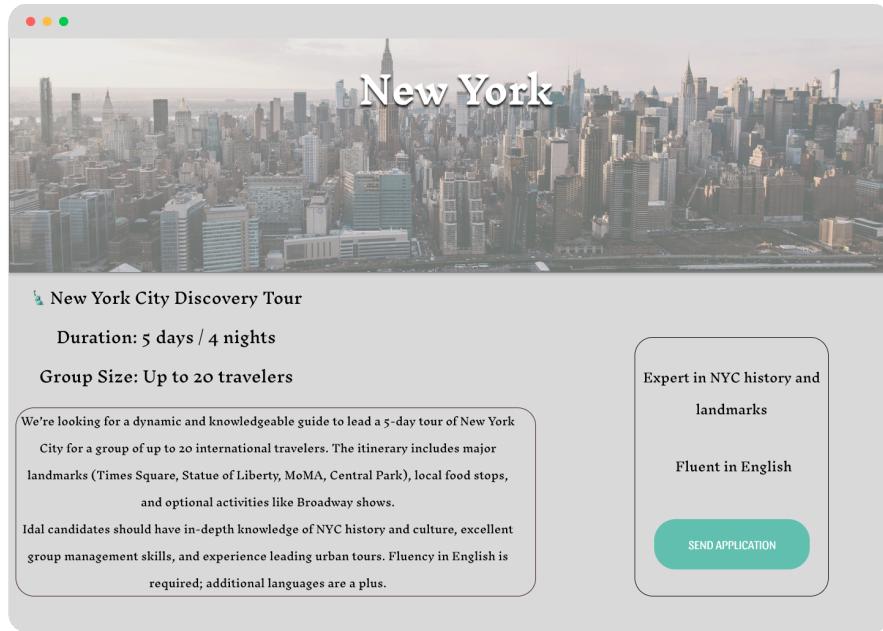


Figura 16: Mockup - Trip Details (Guide)

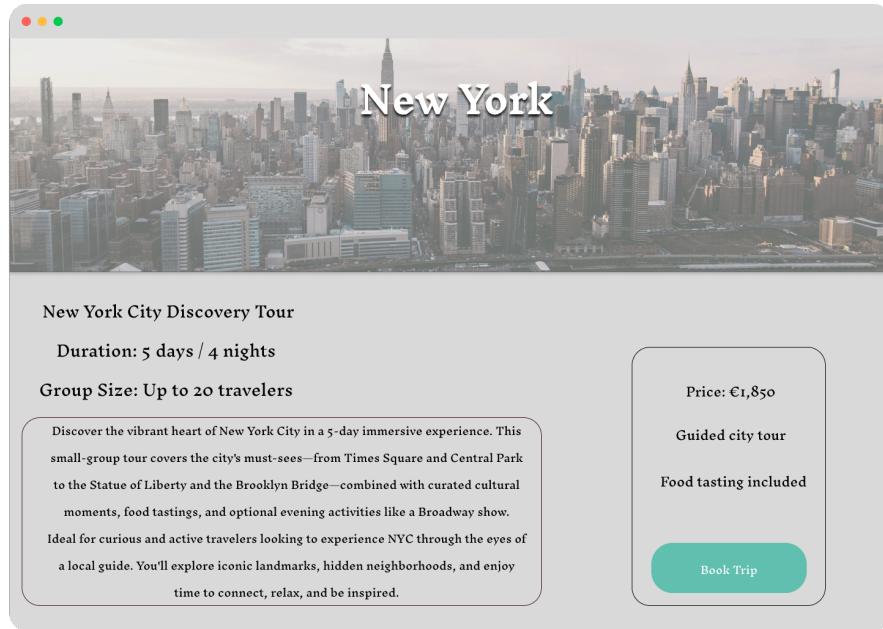


Figura 17: Mockup - Trip Details (Traveler)



Figura 18: Mockup - Leave Review

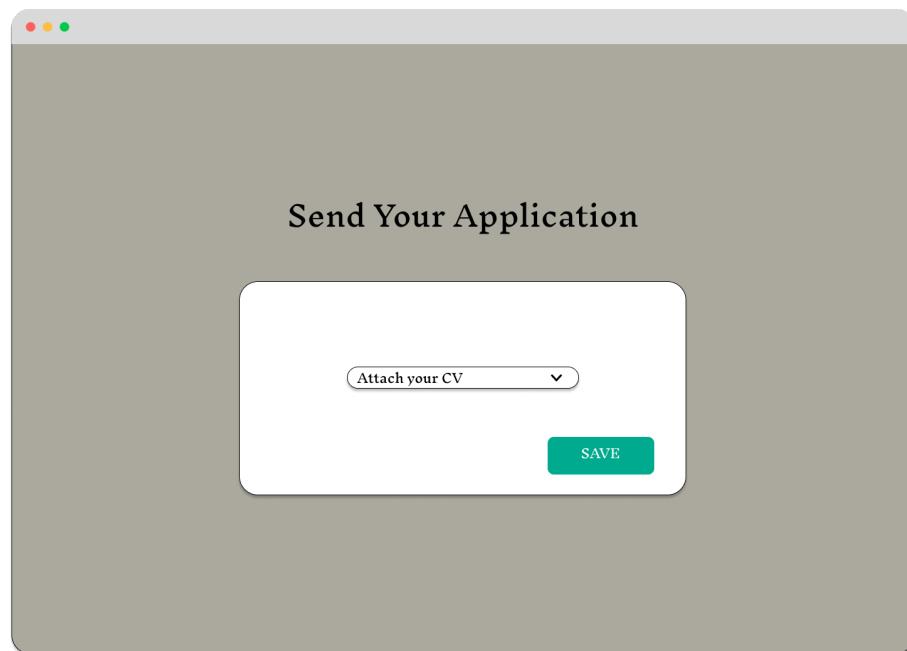


Figura 19: Mockup - Send your Application

## 2.4 Database

Per la gestione della persistenza dei dati è stato utilizzato il DBMS PostgreSQL e per l'accesso al database abbiamo scelto di utilizzare JDBC

### 2.4.1 Modello E-R

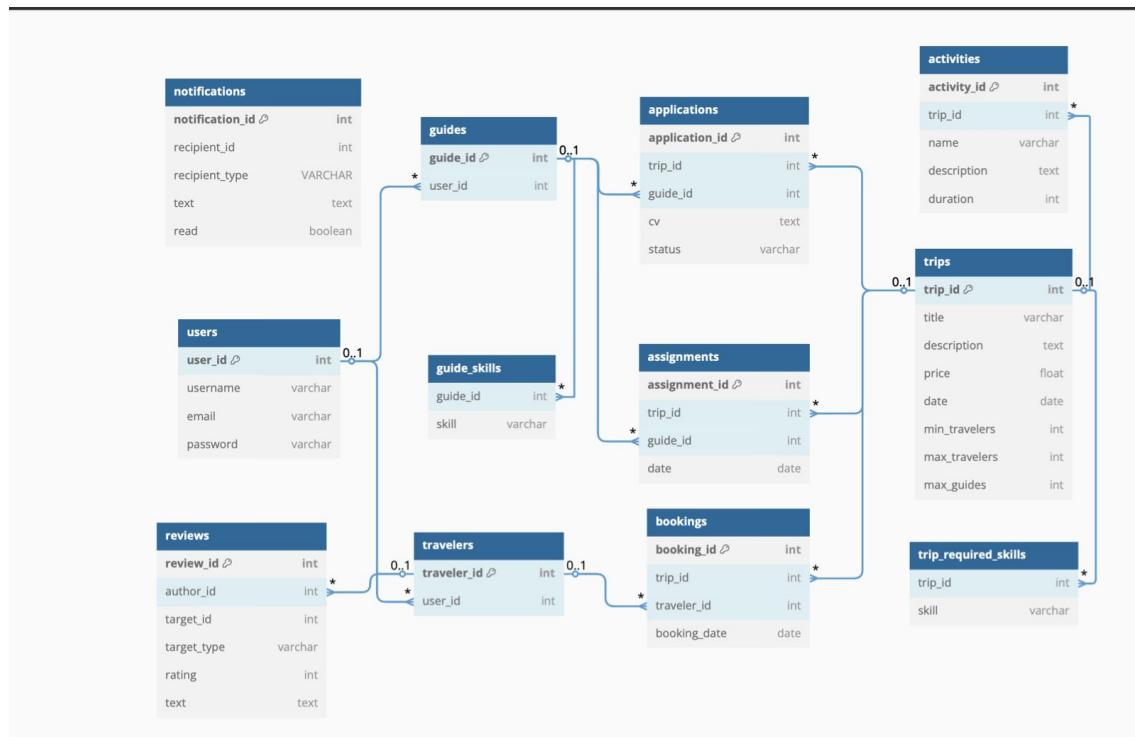


Figura 20: Diagramma E-R

Sono presenti le seguenti relazioni :

- Le relazioni uno a uno tra **User** e **Guide** e tra **User** e **Traveler** indicano la possibilità per ciascun utente di avere un profilo guida, un profilo viaggiatore o entrambi.
- La relazione uno a molti tra **Guide** e **Skill** indica la possibilità per una guida di possedere una o più Skill.

- **Assignment:** rappresenta la reificazione della relazione tra **Guide** e **Trip**. Essa consente di memorizzare non solo il collegamento tra guida e viaggio, ma anche informazioni aggiuntive relative all'assegnazione, come la data. Nel modello, ciò si traduce in due relazioni uno-a-molti: ciascuna guida può essere associata a più assignment e ciascun viaggio può avere più guide assegnate tramite assignment distinti.
- **Booking :** reificazione della relazione molti-a-molti tra **Traveler** e **Trip**. In questo modo la connessione diretta viene sostituita da due relazioni uno-a-molti: ogni viaggiatore può avere più prenotazioni e ogni viaggio può essere prenotato da più viaggiatori . Ciascuna prenotazione, a sua volta, fa riferimento in modo univoco a un solo viaggiatore e a un solo viaggio.
- **Application:** entità risultante dalla reificazione della relazione molti-a-molti tra **Guide** e **Trip**. Ogni guida può inviare più candidature, ogni viaggio può ricevere candidature da più guide.
- La relazione uno a molti tra **Trip** e **Activity**, indica la possibilità per un viaggio di avere al suo interno una o più attività.
- La relazione uno a molti tra **Trip** e **Trip\_required\_Skill** indica la possibilità per un viaggio di avere una o più skill richieste.
- La relazione uno a molti tra **Traveler** e **Review**, indica che ogni viaggiatore può lasciare una o più recensioni.

## 2.4.2 Modello Relazionale

Listing 1: Modello relazionale

```
USER (PK (user_id), username, password, email)
GUIDE (PK (guide_id), FK (user_id ref id on USER))
TRAVELER (PK (traveler_id), FK (user_id ref id on USER))
GUIDE_SKILL (skill, FK (guide_id ref id on GUIDE))
REVIEW (PK (review_id), FK (author_id ref id on TRAVELER), target_id,
        target_type, rating, text)
BOOKING (PK (booking_id), FK (trip_id ref id on TRIP), FK (traveler_id ref id
        on TRAVELER), booking_date)
ASSIGNMENT (PK (assignment_id), FK (trip_id ref id on TRIP), FK (guide_id ref
        id on GUIDE), date)
APPLICATION (PK (application_id), FK (trip_id ref id on TRIP), FK (guide_id
        ref id on GUIDE), cv, status)
ACTIVITY (PK (activity_id), FK (trip_id ref id on TRIP), name, description,
        duration)
TRIP (PK (trip_id), title, description, price, date, min_travelers,
       max_travelers, max_guides)
TRIP_REQUIRED_SKILL ( skill, FK (trip_id ref id on TRIP))
NOTIFICATION (PK (notification_id), recipient_id ref, recipient_type, text,
               read)
```

### 3 Struttura del Progetto

L'implementazione del progetto è stata strutturata secondo una suddivisione in package, in modo da favorire la modularità e la manutenibilità del codice. L'architettura individuata prevede i seguenti package principali:

- **Domain Model:** contiene tutte le classi che rappresentano i concetti fondamentali e le entità del dominio applicativo, come utenti, viaggi, guide, candidature e attività. Questo package funge da base per la modellazione dei dati e l'interazione tra gli oggetti principali del sistema.
- **Business Logic:** racchiude i componenti responsabili della logica applicativa e della gestione dei processi principali del sistema. All'interno di questo package sono presenti sia i *controller*, che si occupano della gestione delle richieste provenienti dall'esterno, sia i *service*, che implementano le regole di business e coordinano le operazioni tra i diversi livelli dell'applicazione.
- **DAO** questo package comprende le classi deputate all'accesso e alla gestione dei dati persistenti. Include i *DAO* (Data Access Object), responsabili delle operazioni di lettura e scrittura sul database, e i componenti per la gestione delle connessioni e delle transazioni con il sistema di archiviazione dati.

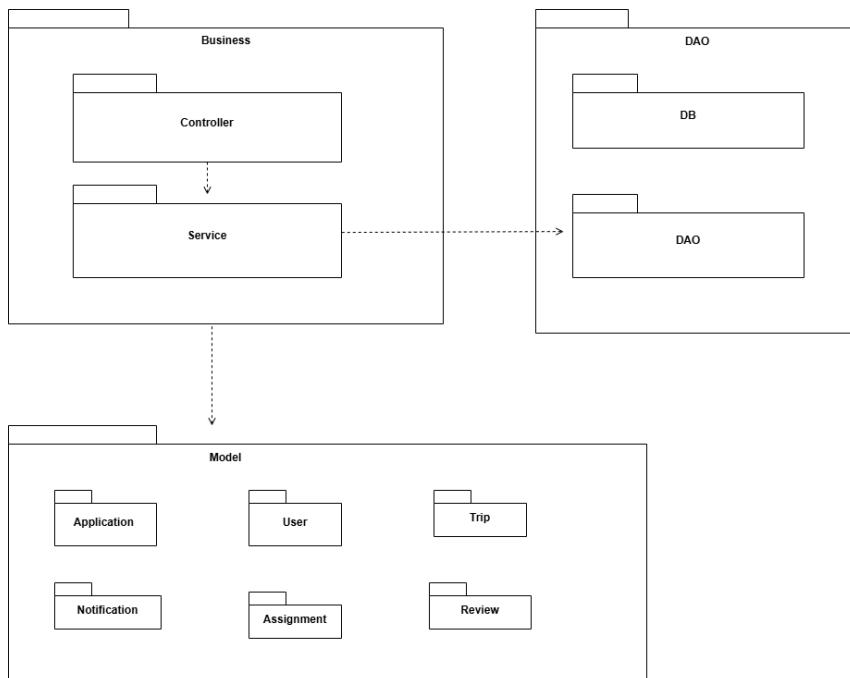


Figura 21: Panoramica generale UML

### 3.1 Domain Model

Il domain model della piattaforma è stato progettato per offrire una rappresentazione fedele dei principali attori e delle relazioni che caratterizzano il sistema. La classe centrale è **User**, che raccoglie tutte le informazioni di base di ogni utente e permette di assumere, di volta in volta, sia il ruolo di guida che quello di viaggiatore tramite i profili associati.

Ogni guida può dichiarare le proprie competenze attraverso la classe **Skill** e arricchire il profilo in base alle esperienze accumulate. Al centro dell'offerta c'è la classe **Trip**, che descrive ogni viaggio organizzato dall'agenzia con dettagli su titolo, descrizione, prezzo, data, attività pianificate e skill richieste alle guide. La gestione delle relazioni tra attori e viaggi si fonda sull'uso di mapper come **Application**, **Assignment**, **Booking** e **Review**. Queste classi fungono da "ponti" che collegano utenti, viaggi, candidature e recensioni, rendendo esplicativi e tracciabili tutti i legami molti-a-molti presenti nel sistema. Ogni mapper è affiancato da un registro dedicato (come **ApplicationRegister** o **BookingRegister**) che funge da contenitore e a facilitare la gestione delle collezioni, garantendo ordine.

L'introduzione di queste strutture nasce dall'esigenza di mantenere separate le responsabilità e di favorire una crescita futura del sistema: mapper e registri rendono infatti semplice estendere le funzionalità o introdurre nuove regole senza appesantire la logica delle singole classi principali. Completano il quadro il sistema di notifiche (**Notification** e *Notifiable*), fondamentale per mantenere gli utenti aggiornati sugli eventi rilevanti, e la gestione delle recensioni, che contribuisce a migliorare la qualità dell'esperienza per tutta la community.

## Dettagli di Implementazione

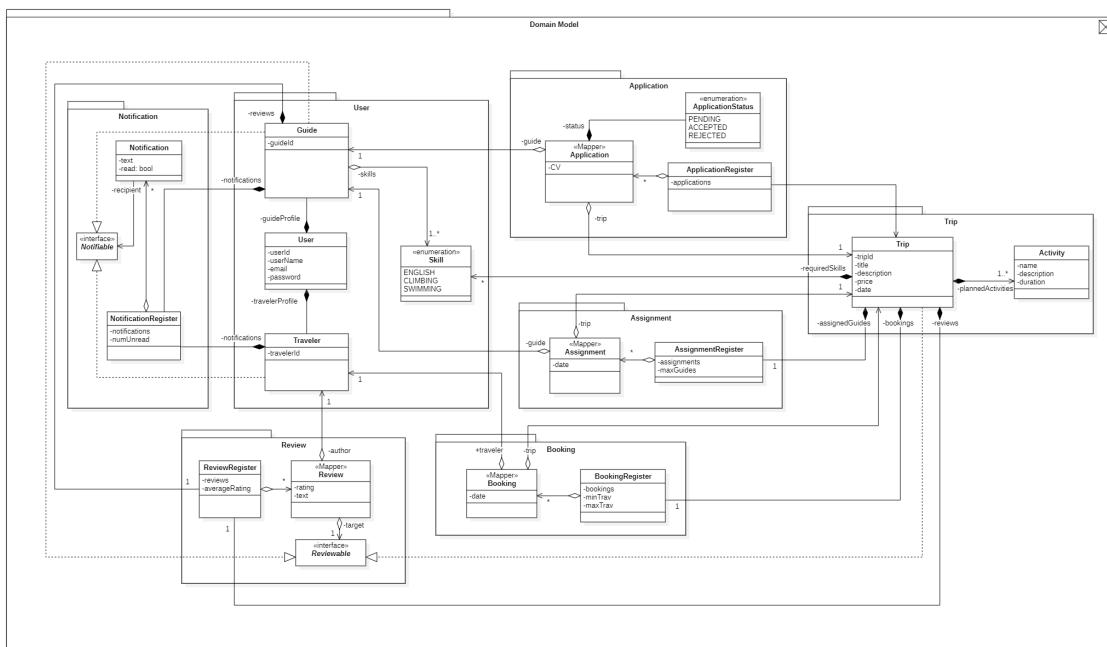


Figura 22: Domain Model

### 3.2 Business Logic

La logica applicativa della piattaforma è organizzata in due macro-componenti: controllers e services. I controller (`UserController`, `GuideController`, `TravelerController` e `AgencyController`) gestiscono l'interazione con i principali attori del sistema, appoggiandosi ai servizi necessari per ciascuna operazione. I servizi encapsulano la logica operativa di ciascuna area funzionale; tra questi spicca `ViewTripsService`, che gestisce la visualizzazione e il filtraggio dinamico dei viaggi tramite il pattern `Strategy` (con `TripFilterStrategy`), permettendo di adattare i criteri di selezione alle esigenze di ogni ruolo.

Completano il quadro altri servizi dedicati alla gestione di viaggi, prenotazioni, recensioni, candidature, assegnazioni e notifiche. Questa organizzazione, basata su una chiara separazione tra il coordinamento dei controller e l'implementazione della logica nei servizi, assicura un'architettura robusta e facilmente estendibile. La scelta di adottare il design pattern `Strategy` per la visualizzazione dei viaggi risponde all'esigenza di mantenere flessibilità e ordine anche in presenza di logiche operative differenti per i vari ruoli della piattaforma.

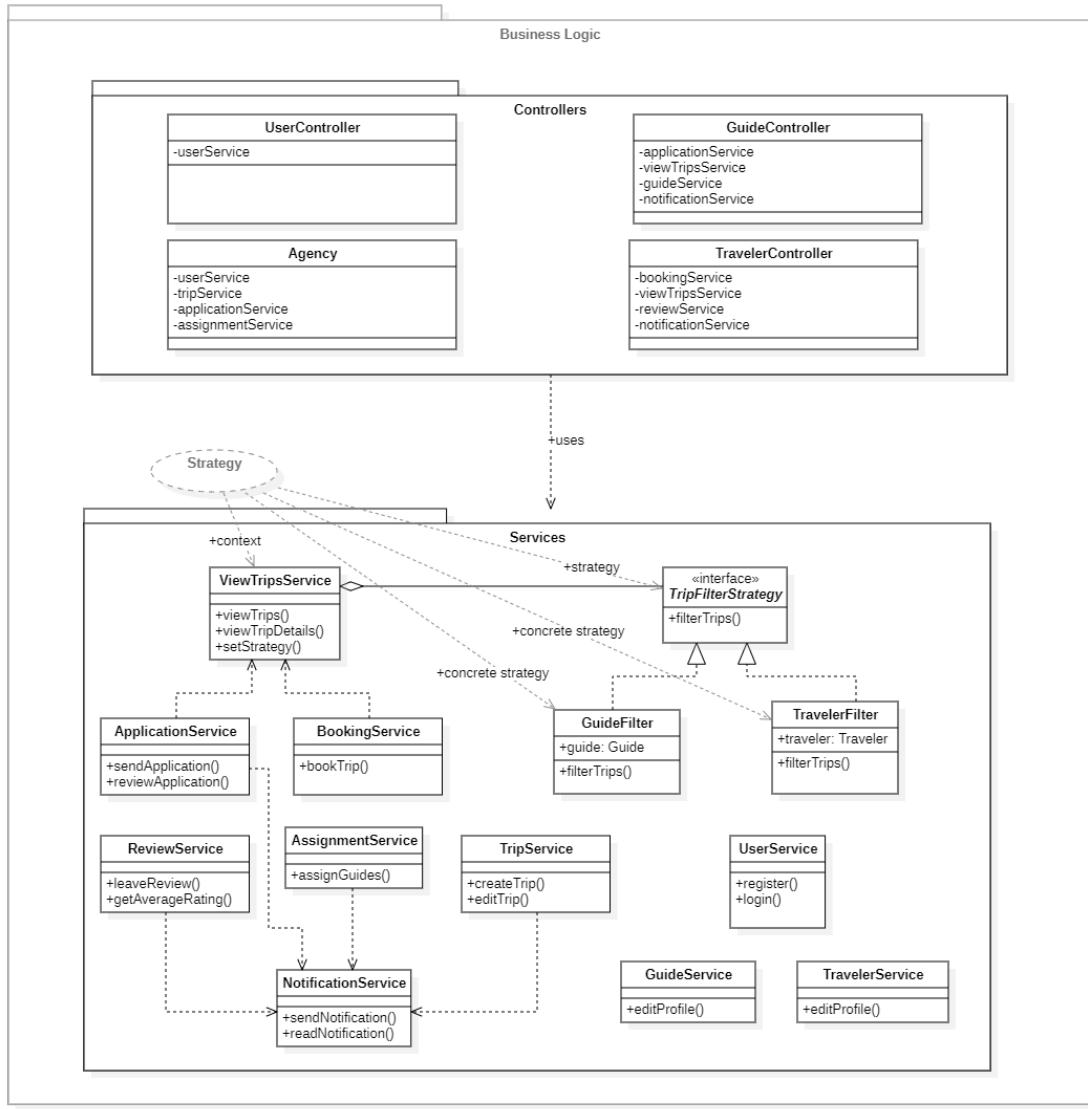


Figura 23: Business Logic

### 3.3 DAO

Il package DAO (Data Access Object) rappresenta il livello di accesso ai dati dell'applicazione. Questo modulo è responsabile dell'interazione diretta con il database e incapsula tutte le operazioni di lettura e scrittura sui dati . Il principale obiettivo è separare la logica di business dalla logica di persistenza.

All'interno del package sono state definite una serie di interfacce DAO per ciascuna entità del dominio, UserDao, TripDAO, BookingDAO, ReviewDAO, GuideDAO.

A ciascuna interfaccia è associata una relativa classe concreta, che ne implementa i metodi. Questa separazione consente di isolare la definizione del comportamento (interfaccia) dalla sua implementazione concreta, facilitando l'eventuale sostituzione o test delle classi.

Tutte le classi concrete condividono una dipendenza verso il componente DBManager , responsabile della gestione della connessione al database. Il DBManager è stato progettato seguendo il design pattern *Singleton*, assicurando che esista una sola istanza condivisa in tutto il sistema. Offre i metodi init(), getInstance() e disconnect() per inizializzare, ottenere e chiudere la connessione al database.

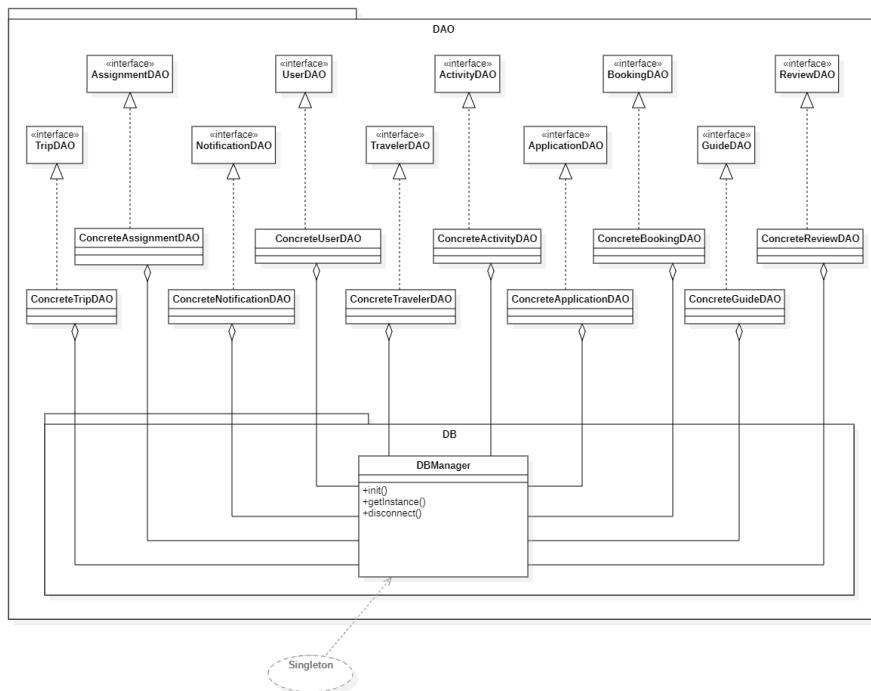


Figura 24: DAO

## 4 Implementazione

Di seguito vengono descritte le principali classi dei package business e dao, in quanto racchiudono la logica applicativa e di accesso ai dati del sistema. Il package model non viene trattato poiché contiene solo la definizione delle entità senza particolari aspetti implementativi di rilievo. Inoltre, per l'implementazione del codice abbiamo fatto uso di Large Language Model come **ChatGPT** e **Claude**.

### 4.1 Project Structure

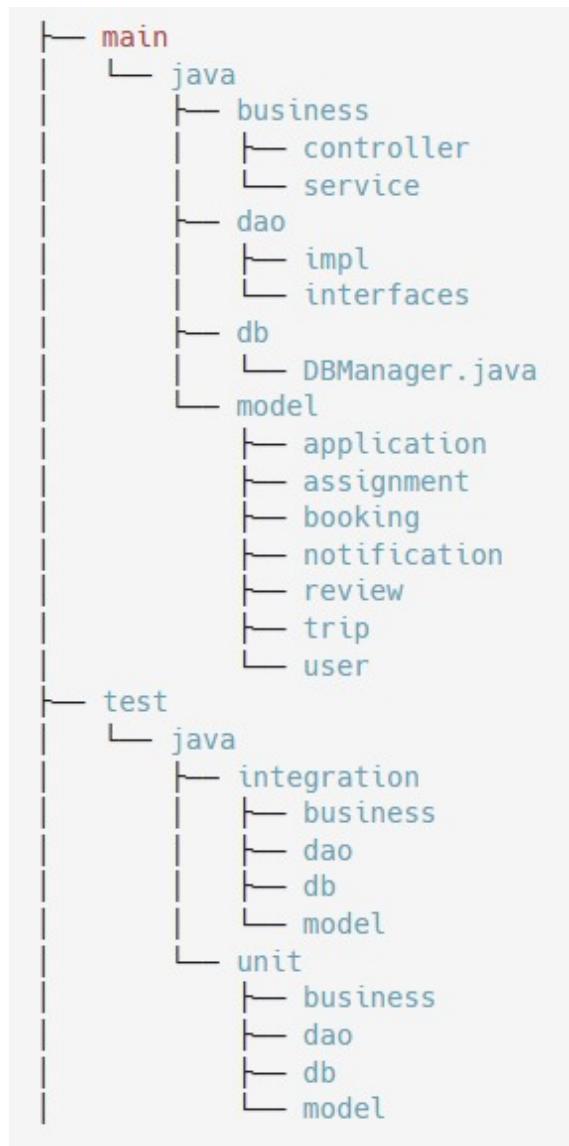


Figura 25: Project Structure

## 4.2 Implementazione della Logica di Business

### 4.2.1 Package controller

**UserController** Gestisce registrazione, login/logout e creazione dinamica dei profili guida e viaggiatore. Verifica la presenza dei profili ed effettua operazioni sullo user attualmente autenticato.

**GuideController** Offre alla guida strumenti per visualizzare i viaggi compatibili, candidarsi, gestire le proprie skill, consultare le notifiche, e tenere traccia dei viaggi assegnati.

**TravelerController** Permette al viaggiatore di esplorare e prenotare viaggi, cancellare prenotazioni, lasciare recensioni su guide e viaggi, e visualizzare notifiche. Utilizza filtri personalizzati per l'esperienza utente.

**Agency** Gestisce la creazione, modifica e cancellazione dei viaggi, accetta o rifiuta candidature, assegna le guide ai viaggi e consente la consultazione delle candidature per ciascun viaggio.

### 4.2.2 Service

**ApplicationService** Gestisce tutte le operazioni legate alle candidature delle guide per i viaggi. Si occupa dell'invio (`sendApplication()`), aggiornamento stato (`updateApplicationStatus()`), ritiro, caricamento e conteggio delle candidature, oltre alla selezione automatica delle migliori guide (`selectBestGuidesForTrip()`). Integra notifiche automatiche e supporta l'accettazione/rifiuto delle candidature.

**BookingService** Gestisce le prenotazioni dei viaggiatori, occupandosi di creare e cancellare prenotazioni (`bookTrip()`, `cancelBooking()`), caricare i dati dal database, inviare notifiche agli utenti e verificare lo stato dei viaggi (posti disponibili, minimo partecipanti). Fornisce anche utility per recuperare tutte le prenotazioni di un viaggio o di un viaggiatore.

**GuideService** si occupa della gestione delle guide turistiche, offrendo operazioni di creazione, aggiornamento, cancellazione (`addGuide()`, `updateGuide()`, `deleteGuide()`) e ricerca. Permette inoltre di recuperare tutti i viaggi assegnati a una guida e di filtrare quelli futuri o passati (`getAssignedTrips()`, `getUpcomingAssignedTrips()`, `getPastAssignedTrips()`).

**NotificationService** Gestisce l'invio, la lettura e la cancellazione delle notifiche tra utenti. Offre metodi per inviare notifiche (`sendNotification()`), marcare come lette singole notifiche o tutte quelle di un destinatario (`markAsRead()`, `markAllAsRead()`), e per caricare o ottenere le notifiche associate a guide o viaggiatori. Esegue il lazy loading delle notifiche solo quando necessario.

**ReviewService** Gestisce la creazione, il salvataggio, l'aggiornamento e la cancellazione delle recensioni. Permette di aggiungere recensioni (`addReview()`, `createAndAddReview()`), ottenerle per autore o destinatario (`getReviewsByTarget()`, `getReviewsByAuthor()`), calcolare la media delle valutazioni (`getAverageRating()`) e notificare il destinatario in caso di nuova recensione.

**TravelerService** Si occupa delle operazioni legate ai viaggiatori. Consente di aggiungere, modificare, eliminare o recuperare un `Traveler` tramite i metodi `addTraveler()`, `updateTraveler()`, `deleteTraveler()`, `getTravelerById()`. Inoltre, offre funzionalità avanzate come `getBookings()` per ottenere tutte le prenotazioni, `getUpcomingTrips()` e `getPastTrips()` per distinguere viaggi futuri e passati, `hasMetGuide()` e `hasCompletedTrip()` per verificare interazioni avvenute, e `canCancelBooking()` per controllare se una prenotazione può essere annullata.

**TripService** Permette il caricamento completo delle sue relazioni (`Activity`, `Booking`, `Assignment`, `Application`). Include funzioni per aggiornare un viaggio (`updateTrip()`) o eliminarlo in modo sicuro (`deleteTrip()`), notificando guide e viaggiatori coinvolti tramite il `NotificationService`. Permette inoltre il caricamento dettagliato di un viaggio con tutte le sue informazioni (`getCompleteTripById()`).

**Strategy** Per adattare la visualizzazione dei viaggi alle esigenze di guide e viaggiatori, è stato adottato lo *Strategy Pattern*, implementato nella classe `ViewTripsService`. Questo servizio gestisce la logica di filtro attraverso

## Dettagli di Implementazione

---

so l'interfaccia `TripFilterStrategy`, il cui comportamento viene assegnato dinamicamente tramite il metodo `setStrategy()`.

Nel caso delle guide, la strategia `GuideFilter` consente di selezionare i viaggi compatibili con le `skills` possedute dalla guida e all'interno di un determinato intervallo temporale. Per i viaggiatori, la strategia `TravelerFilter` filtra invece in base alla disponibilità di posti, al prezzo massimo e al periodo selezionato.

Il metodo `viewTrips()` di `ViewTripsService` recupera l'elenco completo dei viaggi tramite `tripDAO.findAll()` e applica il metodo `filterTrips()` della strategia attiva. In questo modo, grazie alla separazione tra contesto (`ViewTripsService`) e comportamento variabile (`GuideFilter`, `TravelerFilter`), si ottiene un comportamento diverso a seconda dell'utente che ne fa uso.

```
1 package business.service;
2
3 import model.trip.Trip;
4 import java.util.List;
5
6 public interface TripFilterStrategy {
7     List<Trip> filterTrips(List<Trip> allTrips);
8 }
```

Figura 26: Strategy

```
@Override
public List<Trip> filterTrips(List<Trip> allTrips) {
    List<Trip> result = new ArrayList<>();
    for (Trip trip : allTrips) {
        BookingRegister bookingRegister = trip.getBookingRegister();
        boolean hasFreeSpots = bookingRegister.getBookings().size() < bookingRegister.getMaxTrav();
        boolean dateOK = (minDate == null || !trip.getDate().isBefore(minDate)) &&
                        (maxDate == null || !trip.getDate().isAfter(maxDate));
        boolean priceOK = (maxPrice == null || trip.getPrice() <= maxPrice);

        if (hasFreeSpots && dateOK && priceOK) {
            result.add(trip);
        }
    }
    return result;
}
```

Figura 27: Strategy - Classe : `TravelerFilter`

```

public List<Trip> viewTrips() {
    List<Trip> allTrips = tripDAO.findAll(); // Usa findAll() invece di getAll()
    if (strategy != null) {
        return strategy.filterTrips(allTrips);
    }
    return allTrips;
}

```

Figura 28: Strategy - Classe : ViewTripService

### 4.3 Implementazione della logica di DAO

**ConcreteActivityDAO** Implementa l’interfaccia ActivityDAO per gestire la persistenza delle attività (Activity) associate a un viaggio. Fornisce operazioni CRUD complete: save(), update(), findById(), findByTripId(), delete(). Include inoltre metodi aggiuntivi come addToTrip() e removeFromTrip() per associare o rimuovere attività da uno specifico viaggio, supportando operazioni dirette legate al contesto applicativo.

**ConcreteApplicationDAO** ConcreteApplicationDAO Gestisce tutte le operazioni sulle candidature (Application), inclusa la creazione (save()), aggiornamento dello stato (updateStatus()), eliminazione (delete()), e ricerca per ID, guida, viaggio o stato. Fornisce metodi per verificare candidature duplicate (hasGuideAppliedForTrip()), contare candidature (countApplicationsByTripId()), e caricare le candidature nei viaggi (loadApplicationsForTrip()). Supporta anche l’iniezione di GuideDAO e TripDAO per popolare oggetti guida/viaggio nelle candidature.

**ConcreteAssignmentDAO** Gestisce l’assegnazione delle guide ai viaggi. Offre metodi per salvare (save()), rimuovere (delete()) e recuperare assegnazioni (findByTripId(), findByGuideId()), oltre al metodo assignBestGuide() che seleziona automaticamente la guida per un viaggio. Supporta inoltre l’inizializzazione del registro delle assegnazioni tramite loadAssignmentRegister()

**ConcreteBookingDAO** Gestisce le prenotazioni effettuate dai viaggiatori. Permette di salvare (save()), modificare (updateBooking()), eliminare (delete()), e recuperare prenotazioni singole (getById()), per viaggiatore (getByTraveler()), per viaggio (getByTripId()) o tutte (getAll()). Il metodo loadBookingsForTrip() associa le prenotazioni al BookingRegister del viaggio corrispondente, popolando anche i riferimenti a Traveler e Trip se disponibili. Utilizza createBookingFromResultSet() per costruire oggetti Booking a partire dal database.

**ConcreteGuideDAO** Gestisce l'accesso ai dati delle guide. Consente di salvare (save()), aggiornare (update()), e cancellare (delete()) le guide, includendo la gestione delle loro skill tramite saveGuideSkills() e updateGuideSkills(). Fornisce metodi per recuperare guide (findById(), findByUserId(), getAll()), e i viaggi a cui sono assegnate (getAssignedTrips()). Ricostruisce oggetti Guide completi con createGuideFromResultSet(), integrando anche ReviewRegister e NotificationRegister tramite loadReviewRegister() e loadNotificationRegister().

**ConcreteNotificationDAO** Gestisce la persistenza e il recupero delle notifiche per guide e viaggiatori. Consente di salvare (save()), aggiornare (update()), e cancellare (delete()) notifiche, oltre a segnarle come lette (markAsRead(), markAllAsRead()). Offre metodi di query filtrati per utente (getByUserId()), per destinatario specifico (getByRecipientId()), per guida o viaggiatore (getByGuide(), getByTraveler()), e per notifiche non lette (getUnreadByRecipient()). Supporta il caricamento nel NotificationRegister dell'utente (loadNotificationsForRecipient()) e ricostruisce notifiche complete con destinatari tramite createNotificationFromResultSet().

**ConcreteReviewDAO** Responsabile della gestione delle recensioni scritte dai viaggiatori. Supporta inserimento, aggiornamento, rimozione e query filtrate per autore (getByAuthor()) o target (getByTarget()). Utilizza TravelerDAO, GuideDAO e TripDAO per costruire oggetti completi in createReviewFromResultSet().

**ConcreteTravelerDAO** Si occupa della persistenza dei viaggiatori nel sistema. Permette operazioni CRUD (save(), update(), delete()), oltre al caricamento tramite ID (findById()) o ID utente (findByUserId()). Utilizza UserDao

per ricostruire l'oggetto User associato, e NotificationDAO per popolare il NotificationRegister. Il metodo createTravelerFromResultSet() costruisce l'oggetto Traveler completo a partire dal risultato SQL.

**ConcreteTripDAO** Questa classe gestisce il salvataggio, l'aggiornamento e il recupero dei viaggi dal database, occupandosi anche delle relazioni con attività, skill richieste, prenotazioni, assegnazioni e candidature.

Adotta una logica di **lazy loading**: il metodo findById(int) carica solo i dati essenziali del viaggio, mentre il trip completo viene caricato solo tramite findByIdFull(int), che invoca anche loadTripRelations(Trip). Questo approccio permette di ottimizzare le prestazioni ed evitare overhead non necessari.

```

@Override
public Trip findById(int id) {
    String sql = "SELECT * FROM trip WHERE trip_id = ?";
    try (Connection conn = dbManager.getConnection();
        PreparedStatement stmt = conn.prepareStatement(sql)) {
        stmt.setInt(1, id);
        ResultSet rs = stmt.executeQuery();
        if (rs.next()) {
            return createBasicTripFromResultSet(rs);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return null;
}

@Override
public Trip findByIdFull(int id) {
    Trip trip = findById(id);
    if (trip != null) {
        loadTripRelations(trip);
    }
    return trip;
}

```

Figura 29: Metodi in ConcreteTripDAO

**ConcreteUserDAO** Gestisce tutte le operazioni CRUD sugli utenti, inclusa la creazione e il caricamento dei profili associati (Guide e Traveler) tramite i rispettivi DAO. I metodi principali sono save(), findById(), find-

ByEmail(), update(), delete(). L'associazione con i profili avviene tramite loadUserProfiles().

### 4.4 Implementazione del Database

Come già accennato nella sezione precedente, il database è stato progettato per rispecchiare fedelmente i requisiti dell'applicazione. Sono state implementate le principali tabelle come users, guides, travelers, trips, bookings e applications, secondo lo schema presentato in Figura 20. I file SQL necessari alla creazione e alla gestione del database sono contenuti nella cartella src/main/resources/db.properties.

```
1 CREATE TABLE users (
2     user_id SERIAL PRIMARY KEY,
3     username VARCHAR(50) NOT NULL,
4     email VARCHAR(100) UNIQUE NOT NULL,
5     password VARCHAR(255) NOT NULL
6 );
7
8 CREATE TABLE guides (
9     guide_id SERIAL PRIMARY KEY,
10    user_id INT NOT NULL UNIQUE,
11    FOREIGN KEY (user_id) REFERENCES users(user_id)
12 );
13
14 CREATE TABLE travelers (
15    traveler_id SERIAL PRIMARY KEY,
16    user_id INT NOT NULL UNIQUE,
17    FOREIGN KEY (user_id) REFERENCES users(user_id)
18 );
```

Figura 30: Creazione Tabelle su database

```
CREATE TABLE bookings (
    booking_id SERIAL PRIMARY KEY,
    trip_id INT NOT NULL,
    traveler_id INT NOT NULL,
    booking_date DATE NOT NULL,
    FOREIGN KEY (trip_id) REFERENCES trips(trip_id),
    FOREIGN KEY (traveler_id) REFERENCES travelers(traveler_id)
);

CREATE TABLE applications (
    application_id SERIAL PRIMARY KEY,
    trip_id INT NOT NULL,
    guide_id INT NOT NULL,
    cv TEXT,
    status VARCHAR(20),
    FOREIGN KEY (trip_id) REFERENCES trips(trip_id),
    FOREIGN KEY (guide_id) REFERENCES guides(guide_id)
);
```

Figura 31: Creazione Tabelle su database

## 5 Strategia di Testing

La strategia di testing adottata si concentra principalmente su unit test, che garantiscono un controllo puntuale e isolato delle funzionalità delle singole classi attraverso l'utilizzo di JUnit e Mockito. Grazie ai mock, è stato possibile simulare le dipendenze esterne e valutare il comportamento dei metodi in modo indipendente dal resto del sistema.

Abbiamo affiancato a questi anche alcuni integration test, limitati ai flussi e alle interazioni considerate più critiche, come la comunicazione tra servizi e l'accesso al database.

La selezione delle classi da sottoporre a test approfonditi è stata guidata dalla rilevanza della logica di business e dalla centralità nella gestione dei dati. Al contrario, sono stati esclusi dal testing dettagliato i componenti che svolgono una funzione di rappresentazione dei dati, come le classi del package model, poiché privi di logica.

### 5.1 Unit Test

Gli unit test sono stati scritti per tutte le classi del package service, poiché la loro logica rappresenta un elemento fondamentale per il corretto funzionamento del sistema. Inoltre, abbiamo testato anche alcune classi del package dao la cui logica risultava particolarmente complessa, sebbene queste siano poi ulteriormente coperte dagli integration test. In totale, sono stati sviluppati 109 unit test.

```
@BeforeEach
void setUp() {
    bookingService = new BookingService(bookingDAO, tripDAO, travelerDAO, notificationService);
}

@Test
void testBookTrip_Success() {
    // Arrange
    when(trip.getBookingRegister()).thenReturn(bookingRegister);
    when(bookingRegister.getAvailableSpots()).thenReturn(1);
    when(bookingRegister.hasBooking(traveler)).thenReturn(false);
    when(trip.getTitle()).thenReturn("Test Trip");

    // Act
    boolean result = bookingService.bookTrip(traveler, trip);

    // Assert
    assertTrue(result);
    verify(bookingDAO).save(any(Booking.class));
    verify(bookingRegister).addBooking(any(Booking.class));
    verify(notificationService).sendNotification(eq(traveler), contains("confermata"));
}
```

Figura 32: Test classe BookingService

## 5.2 Integration Test

Per quanto riguarda gli Integration Test, abbiamo scelto di concentrarci sui flussi più rilevanti del sistema, verificando l'interazione tra i diversi componenti e il corretto funzionamento dell'accesso ai dati. In totale, sono stati sviluppati 41 integration test, focalizzati sulle principali classi DAO concrete (ConcreteApplicationDAOIT, ConcreteBookingDAOIT, ConcreteTripDAOIT) e sui controller più significativi (AgencyController e GuideController). Questa scelta ci ha permesso di validare il comportamento complessivo delle operazioni fondamentali, simulando scenari d'uso realistici e garantendo la robustezza dell'intero sistema.

```

@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class ConcreteApplicationDAOIT {

    private static ConcreteApplicationDAO dao;
    private static final int TEST_GUIDE_ID = 999;
    private static final int TEST_TRIP_ID = 999;
    private static final int TEST_USER_ID = TEST_GUIDE_ID;

    private static Application testApplication;

    @BeforeAll
    static void setupAll() {
        dao = new ConcreteApplicationDAO();
    }

    try (Connection conn = DBManager.getInstance().getConnection()) {
        System.out.println("↳ Inizio setup test...");

        // 0) Cleanup preventivo per evitare conflitti
        cleanupTestData(conn);

        // 1) Creo l'utente
        try (PreparedStatement ps = conn.prepareStatement(
                "INSERT INTO users (user_id, username, email, password) VALUES (?, 'testuser999', 'test999@test.com', 'testpass')"))
        {
            ps.setInt(1, TEST_USER_ID);
            ps.executeUpdate();
            System.out.println("✓ Utente test creato");
        }
    }
}

```

Figura 33: Test classe ApplicationDAOIT

```

@BeforeAll
static void setupAll() {
    try (Connection conn = DBManager.getInstance().getConnection()) {
        System.out.println("↳ Inizio setup test...");

        dao = new ConcreteApplicationDAO();
        System.out.println("✓ Setup test completato");
    }
}

@AfterAll
static void cleanupAll() {
    try (Connection conn = DBManager.getInstance().getConnection()) {
        System.out.println("↳ Inizio cleanup finale...");

        cleanupTestData(conn);
        System.out.println("✓ Cleanup finale completato");

    } catch (SQLException e) {
        System.err.println("✗ Errore durante cleanupAll: " + e.getMessage());
        throw new RuntimeException("Errore nel cleanup finale", e);
    }
}

```

Figura 34: Classe ApplicationDAOIT

## Strategia di Testing

---

```
@Test
@Order(1)
void testCreateApplication() {
    System.out.println("📝 Test 1: Creazione application");

    // Creo una nuova Application
    testApplication = new Application(
        0,                                     // 0 → nuovo record
        "Test CV Content - Integration Test",
        TEST_GUIDE_ID,
        TEST_TRIP_ID,
        ApplicationStatus.PENDING
    );

    // Verifico che inizialmente l'ID sia 0
    assertEquals(0, testApplication.getApplicationId(), "ID dovrebbe essere 0 prima del save");

    // Salvo l'application
    dao.save(testApplication);

    // Verifico che l'ID sia stato generato automaticamente
    assertTrue(testApplication.getApplicationId() > 0, "ID dovrebbe essere generato automaticamente dopo il save");

    // Verifico che la ricerca funzioni
    Application result = dao.findByGuideAndTrip(TEST_GUIDE_ID, TEST_TRIP_ID);
    assertNotNull(result, "L'application creata dovrebbe essere trovata");
    assertEquals("Test CV Content - Integration Test", result.getCV());
    assertEquals(ApplicationStatus.PENDING, result.getStatus());
    assertEquals(TEST_GUIDE_ID, result.getGuideId());
    assertEquals(TEST_TRIP_ID, result.getTripId());

    System.out.println("✅ Application creata con ID: " + testApplication.getApplicationId());
}
```

Figura 35: Test classe ApplicationDAOIT

In allegato il repository del nostro progetto: <https://github.com/Piccia03/SWEViaggi>