



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

---

## BarberShop

Software per la Prenotazione di Appuntamenti dal Barbiere

---

Dipartimento di Ingegneria informatica

Anno accademico 2024/2025

**Autori:** Ferritti Andrea, Sansavini Filippo, Lin Jieke

**Corso principale:**  
Ingegneria del Software

**Docente corso:**  
Enrico Vicario

# Indice

<b>1 Introduzione</b>	<b>4</b>
1.1 Statement . . . . .	4
1.2 Requisiti Funzionali . . . . .	4
1.3 Architettura . . . . .	5
1.3.1 Descrizione dell'architettura . . . . .	5
1.4 Dipendenze del progetto . . . . .	7
1.5 Tecnologie e Strumenti Utilizzati . . . . .	8
<b>2 Progettazione</b>	<b>10</b>
2.1 Use Case Diagram . . . . .	10
2.2 Use Case Template . . . . .	11
2.2.1 Sign up . . . . .	11
2.2.2 Sign in . . . . .	12
2.2.3 View Appointments . . . . .	13
2.2.4 Add Appointment . . . . .	14
2.2.5 Delete Appointment . . . . .	15
2.2.6 View News . . . . .	16
2.2.7 View Profile . . . . .	16
2.2.8 View Services . . . . .	17
2.2.9 Add Service . . . . .	17
2.2.10 Delete Service . . . . .	18
2.2.11 Send Comunication . . . . .	19
2.3 Class Diagram . . . . .	20
2.4 Page Navigation Diagram . . . . .	21
2.5 Interfaccia Grafica . . . . .	22
2.5.1 Registrazione e Accesso . . . . .	22
2.5.2 Visualizzazione appuntamenti . . . . .	25
2.5.3 Prenotazione appuntamenti . . . . .	27
2.5.4 News e Invio delle Comunicazioni . . . . .	30
2.5.5 Gestione Servizi . . . . .	32
2.5.6 Profilo . . . . .	33
2.6 Database . . . . .	34
2.6.1 Entity-Relationship Diagram . . . . .	34
2.6.2 Modello relazionale . . . . .	34
<b>3 Implementazione</b>	<b>36</b>
3.1 Model . . . . .	36
3.2 Helpers . . . . .	37
3.3 PageControllers . . . . .	38
3.4 Services . . . . .	41
3.5 Authentication . . . . .	42
3.6 Security . . . . .	42
3.7 Payment . . . . .	43
3.8 Persistence . . . . .	44
3.8.1 DBConnection . . . . .	44
3.8.2 DAO . . . . .	46
3.9 Database . . . . .	47

<b>4</b>	<b>Test</b>	<b>50</b>
4.1	Test Coverage . . . . .	50
4.2	Unit Tests . . . . .	50
4.3	Integration Tests . . . . .	52
4.4	Functional Tests . . . . .	54
<b>5</b>	<b>Utilizzo di AI</b>	<b>57</b>
5.1	Strumenti AI . . . . .	57
5.2	Esempio di interazione . . . . .	57
5.3	Conclusioni . . . . .	59

## Elenco delle figure

1	Architettura del sistema . . . . .	5
2	Dipendenze esterne . . . . .	7
3	Use Case Diagram . . . . .	10
4	Class Diagram . . . . .	20
5	Page Navigation Diagram - Barber . . . . .	21
6	Page Navigation Diagram - Customer . . . . .	21
7	Registrazione . . . . .	22
8	Registrazione - campi vuoti . . . . .	22
9	Registrazione - codice sbagliato . . . . .	23
10	Accesso . . . . .	23
11	Accesso - credenziali errate . . . . .	24
12	Recupero password . . . . .	24
13	Email non registrata . . . . .	24
14	Appuntamenti - barber . . . . .	25
15	Appuntamenti - customer . . . . .	25
16	Cancellazione appuntamento . . . . .	26
17	Nessun appuntamento prenotato . . . . .	26
18	Selezione data . . . . .	27
19	Selezione slot orario, barbiere e servizio . . . . .	27
20	Servizio non selezionato . . . . .	28
21	Aggiunta servizio extra . . . . .	28
22	Servizio extra aggiunto . . . . .	28
23	Riepilogo appuntamento . . . . .	28
24	Selezione metodo di pagamento . . . . .	29
25	Appuntamento prenotato . . . . .	29
26	News . . . . .	30
27	Nessuna notifica . . . . .	30
28	Invio comunicazioni . . . . .	31
29	Campi comunicazione vuoti . . . . .	31
30	Servizi . . . . .	32
31	Campi servizio vuoti . . . . .	32
32	Cancellazione servizio . . . . .	32
33	Nessun servizio . . . . .	33
34	Informazioni del profilo . . . . .	33
35	Entity-Relationship Diagram . . . . .	34
36	Model . . . . .	36
37	Helpers . . . . .	37

38	PageControllers . . . . .	38
39	Services . . . . .	41
40	Authentication . . . . .	42
41	Security . . . . .	42
42	Payment . . . . .	43
43	Persistence . . . . .	44
44	Copertura del codice . . . . .	50

# 1 Introduzione

## 1.1 Statement

Il progetto ha l'obiettivo di sviluppare un sistema per la gestione degli appuntamenti in un negozio di barbiere. Barbershop è un'applicazione full stack che consente ai clienti di prenotare appuntamenti, scegliendo data, orario, barbiere e servizi desiderati. Durante la prenotazione, è possibile selezionare anche il metodo di pagamento, mentre gli slot orari già occupati vengono automaticamente esclusi dalla selezione.

L'app offre funzionalità per la gestione degli appuntamenti: i clienti possono consultare la lista delle proprie prenotazioni e cancellarle, se necessario. Ogni prenotazione genera una notifica per il barbiere, mentre eventuali cancellazioni avvisano sia il barbiere sia gli altri clienti, segnalando quali slot orari sono tornati disponibili. Tutte le notifiche sono raccolte in una sezione dedicata, garantendo un accesso rapido e organizzato alle comunicazioni importanti.

Dal lato del barbiere, l'app semplifica la gestione del lavoro quotidiano, permettendo di visualizzare l'elenco degli appuntamenti prenotati. Inoltre, i barbieri possono inviare comunicazioni personalizzate ai clienti e aggiungere nuovi servizi con i relativi prezzi.

Infine, l'app consente agli utenti di accedere alle informazioni del proprio profilo e integra un meccanismo per la sicurezza delle password.

## 1.2 Requisiti Funzionali

Le principali funzionalità che il sistema soddisfa sono:

- **Registrazione e accesso:** Gli utenti possono registrarsi come barbieri o clienti, accedere all'applicazione e, se necessario, recuperare la password.
- **Visualizzazione appuntamenti:** I clienti possono visualizzare gli appuntamenti prenotati e cancellarli, mentre i barbieri possono visualizzare gli appuntamenti prenotati dai clienti.
- **Prenotazione appuntamenti:** I clienti possono prenotare appuntamenti selezionando data, orario, barbiere, servizio e metodo di pagamento.
- **Notifiche:** I clienti ricevono notifiche sugli slot orari tornati disponibili e sulle comunicazioni inviate dai barbieri. I barbieri ricevono notifiche quando un cliente prenota o cancella un appuntamento.
- **Visualizzazione profilo:** Gli utenti possono visualizzare le informazioni del proprio profilo e fare logout.

## 1.3 Architettura

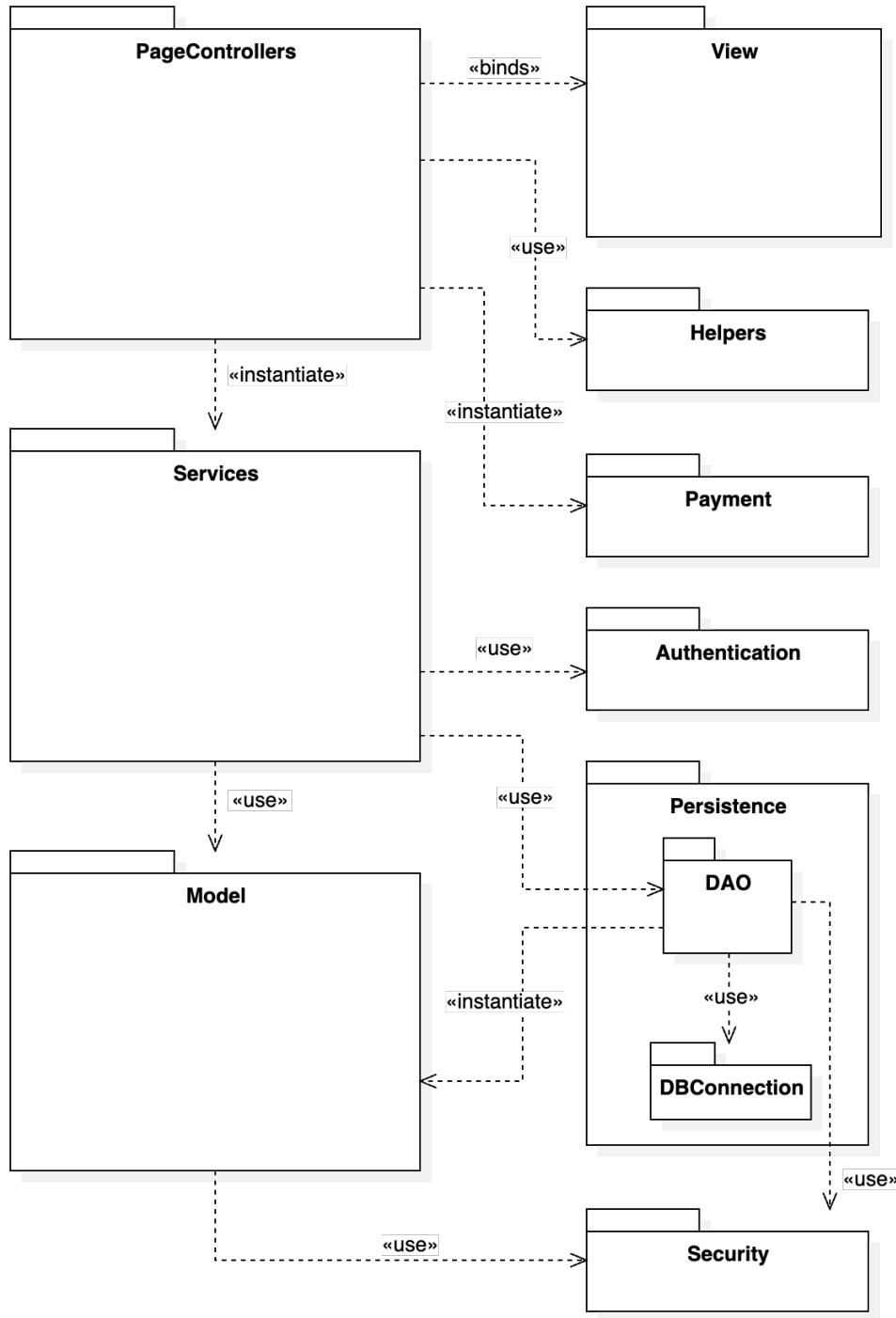
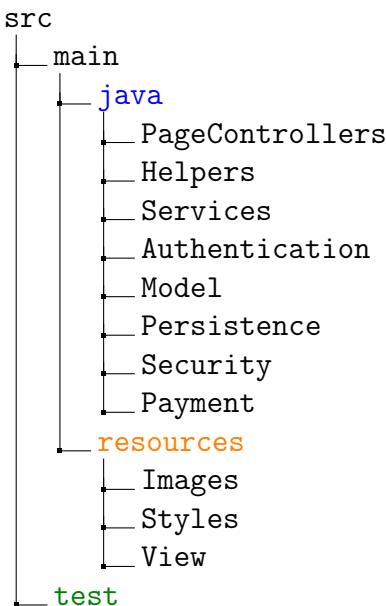


Figura 1: Architettura del sistema

### 1.3.1 Descrizione dell'architettura

Il progetto adotta il pattern **Model–View–Controller (MVC)** per separare gestione dati, interfaccia utente e logica di controllo. La struttura segue la convenzione Maven, con le classi principali organizzate nei package Java e le risorse non Java, come file FXML, CSS e immagini, raccolte nella directory `src/main/resources`. In particolare, le immagini si trovano in `Images`, i file CSS in `Styles` e i file FXML in `View`. I controller, situati nel package

`PageControllers`, carcano tali risorse a runtime tramite `FXMLLoader`, che associa i file FXML ai rispettivi controller Java.



I pacchetti alla base del progetto sono:

1. **View**: Contiene i file FXML i quali definiscono l'interfaccia grafica dell'applicazione. Per una visione più chiara della struttura, si rimanda al Page Navigation Diagram.
2. **PageControllers**: Contiene le classi dei controller, che gestiscono l'interazione tra l'interfaccia utente e la logica di business. Ogni controller è associato a una specifica vista (file FXML) e si occupa di gestire eventi, aggiornare l'interfaccia utente e invocare i metodi appropriati in seguito alle azioni degli utenti.
3. **Helpers**: Contiene classi di supporto per i controller, come `SceneHelper` e `AlertHelper`, che forniscono metodi comuni utilizzati nei controller, ad esempio per il cambio di scena e la creazione di alert di conferma o errore.
4. **Services**: Contiene le classi che implementano la logica di business dell'applicazione. Questo livello si occupa di elaborare i dati ricevuti dal livello di persistenza e di fornire risposte coerenti alle richieste dei controller.
5. **Authentication**: Include la classe `SessionManager`, implementata seguendo il pattern Singleton. Questa classe gestisce la sessione utente, permettendo a ogni scena di accedere alle informazioni relative all'utente attualmente autenticato.
6. **Model**: Contiene le classi che rappresentano le entità principali del sistema, come `User`, `Appointment`, `Service`, ecc. Queste classi sono utilizzate per modellare i dati e le relazioni tra di essi.
7. **Persistence**: Contiene le classi per la gestione della persistenza dei dati, inclusi il pacchetto `DAO` (Data Access Object) e la classe `DBConnection` per la connessione al database.
8. **Security**: Include la classe `HashingPassword`, responsabile della gestione e dell'implementazione delle funzionalità di hashing delle password per garantire la sicurezza dei dati degli utenti.
9. **Payment**: Contiene le classi che gestiscono i metodi di pagamento.

## 1.4 Dipendenze del progetto

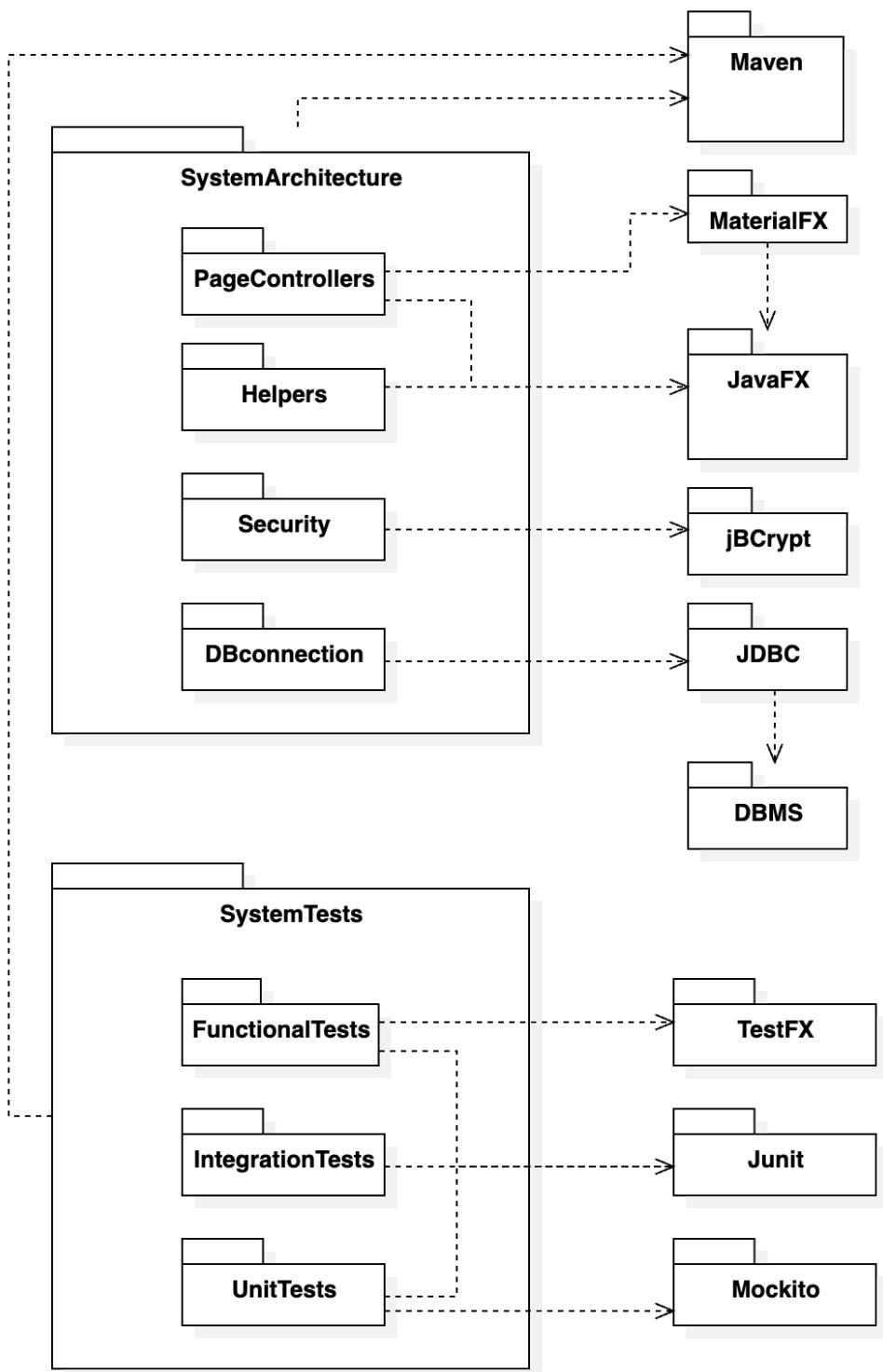


Figura 2: Dipendenze esterne

Il sistema fa ampio uso di dipendenze esterne, fondamentali per semplificare lo sviluppo e potenziare aspetti chiave come l'interfaccia grafica, la sicurezza, la persistenza dei dati e il testing. Per la gestione di queste dipendenze viene utilizzato **Maven**, uno strumento di build per Java che automatizza l'integrazione e l'organizzazione delle librerie e delle risorse del progetto. Dal

punto di vista **architetturale**, il sistema si basa sulle seguenti dipendenze, ciascuna delle quali contribuisce a migliorarne l'efficienza e le funzionalità complessive:

- **MaterialFX**: Un framework avanzato per JavaFX che offre componenti UI moderni e personalizzabili, ispirati al design Material di Google. Questa libreria è stata utilizzata nei controller per migliorare l'aspetto grafico di elementi come pulsanti e campi di testo nella view.
- **JavaFX**: Una potente libreria per la creazione di interfacce grafiche in Java, che consente lo sviluppo di applicazioni desktop con un design moderno e interattivo. I controller hanno sfruttato JavaFX per gestire l'interfaccia utente e le interazioni, caricando dinamicamente i componenti nella view.
- **jBcrypt**: Una libreria dedicata all'hashing delle password, implementata nel pacchetto **Security**. Utilizza l'algoritmo bcrypt per garantire un elevato livello di sicurezza delle credenziali degli utenti.
- **JDBC**: Un'API Java per la connessione e l'interazione con database relazionali. È stata impiegata per gestire la persistenza dei dati, assicurando un accesso rapido e affidabile alle informazioni archiviate.
- **DBMS**: Un sistema di gestione di database relazionali, utilizzato per memorizzare e organizzare i dati dell'applicazione in modo strutturato ed efficiente.

Per il **testing** dell'applicazione sono state utilizzate le seguenti dipendenze:

- **JUnit**: Un framework di testing per Java, utilizzato per scrivere e eseguire test unitari. È stato impiegato per garantire la correttezza del codice e la funzionalità in tutte e tre le tipologie di test: unitari, di integrazione e funzionali.
- **Mockito**: Una libreria per il mocking in Java, utilizzata per creare oggetti finti (mock) durante i test. Ha facilitato il testing delle interazioni tra componenti senza dipendere da implementazioni reali.
- **TestFX**: Un framework di testing per JavaFX, utilizzato per testare l'interfaccia grafica dell'applicazione. Ha permesso di simulare interazioni utente e verificare il comportamento dell'interfaccia in scenari reali.

## 1.5 Tecnologie e Strumenti Utilizzati

L'applicazione è realizzata in **Java**, avvalendosi delle seguenti tecnologie e strumenti:

- **Gestione delle dipendenze**: **Maven** è stato utilizzato per la gestione delle dipendenze e la configurazione del progetto.
- **Database**: Utilizzo di **PostgreSQL** per la gestione dei dati in produzione, con connessione gestita tramite **JDBC**. Per i test, è stato adottato un database **H2** in memoria, garantendo un ambiente di testing rapido e isolato.
- **Interfaccia utente**: Sviluppata con **JavaFX**, per un'esperienza visiva moderna e interattiva. La progettazione delle schermate è stata facilitata dall'uso di **Scene Builder**. Inoltre, il framework **MaterialFX** è stato utilizzato per migliorare l'aspetto grafico di alcuni componenti.

- **Testing:**
  - **JUnit** per le tre tipologie di test: unitari, di integrazione e funzionali.
  - **Mockito** per il testing dei componenti mock.
  - **TestFX** per il testing dell’interfaccia grafica.
- **Progettazione e modellazione:**
  - **StarUML** per la creazione degli *Use Case Diagram* e del *Class Diagram*.
  - **app.diagrams** per la progettazione dell’*Entity-Relationship Diagram* e il *Page Navigation Diagram*.
- **Supporto alla scrittura del codice:** Per migliorare l’efficienza e la qualità del codice, sono stati utilizzati modelli di linguaggio avanzati (**LLMs**) come **ChatGPT**, **Claude**, e **Github Copilot** integrato in **IntelliJ IDEA**. Nella sezione 5 viene approfondito l’uso di questi strumenti, evidenziando i vantaggi e le limitazioni riscontrate durante lo sviluppo.

## 2 Progettazione

### 2.1 Use Case Diagram

Gli attori coinvolti nel sistema sono:

- **Customer:** Cliente registrato e autenticato, che può gestire appuntamenti (visualizzare, prenotare, cancellare), consultare comunicazioni ricevute e visualizzare il proprio profilo.
- **Barber:** Barbiere registrato e autenticato, che può gestire appuntamenti dei clienti, inviare e ricevere comunicazioni, aggiungere o rimuovere servizi.

La figura 3 illustra le principali funzionalità del sistema, evidenziando le interazioni tra gli attori coinvolti e il sistema stesso. Questo diagramma rappresenta una panoramica delle operazioni che gli utenti possono eseguire, fornendo una visione completa dei casi d'uso.

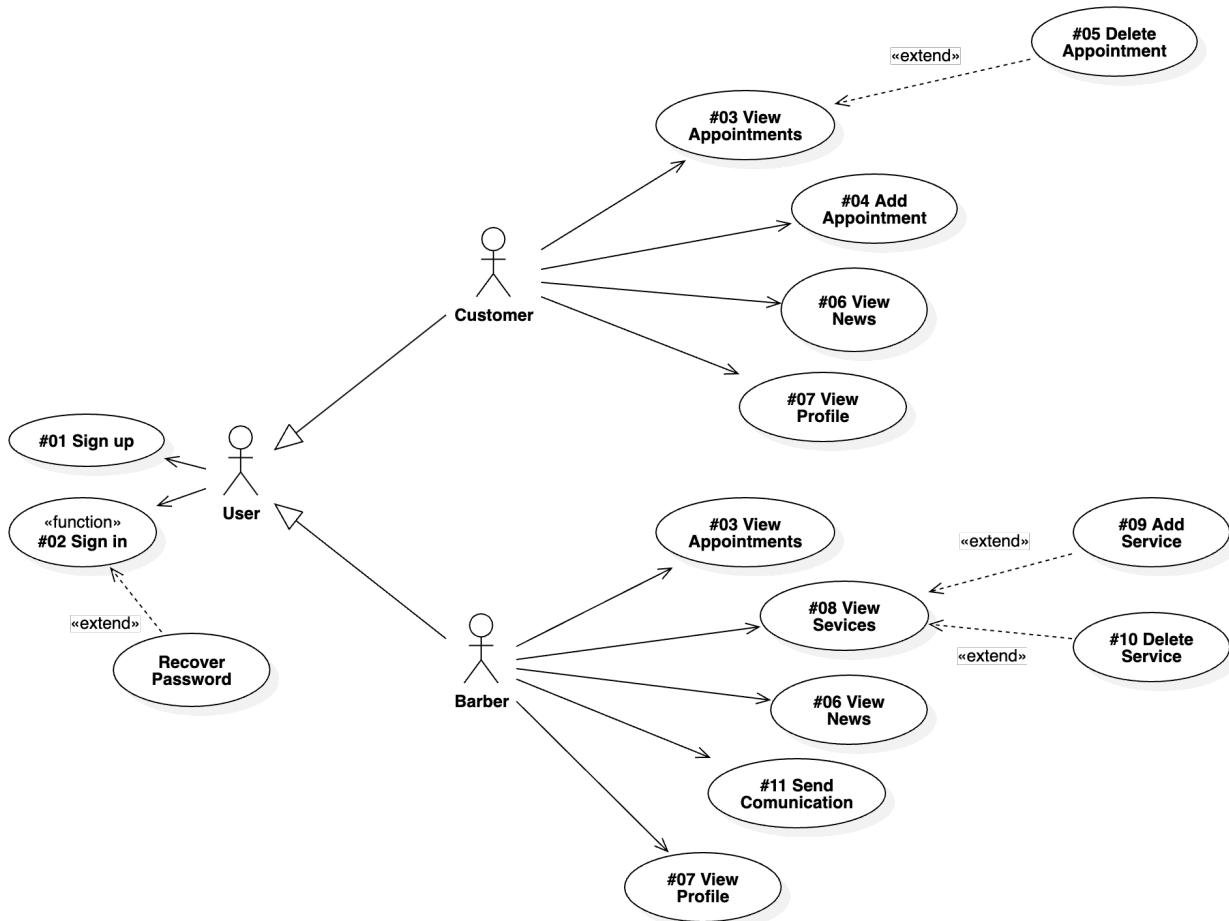


Figura 3: Use Case Diagram

## 2.2 Use Case Template

### 2.2.1 Sign up

<b>UC-01</b>	Sign up
<b>Scope</b>	User Goal
<b>Attori</b>	User
<b>Pre-condizioni</b>	L'utente non è autenticato nel sistema e non ha un account.
<b>Post-condizioni</b>	L'utente è registrato nel sistema e può fare l'accesso
<b>Flusso base</b>	<ol style="list-style-type: none"><li>1. L'utente apre la schermata di registrazione. [7]</li><li>2. L'utente inserisce le informazioni richieste e se è un barbiere inserisce il codice segreto.</li><li>3. L'utente preme il pulsante di registrazione.</li><li>4. Il sistema verifica le informazioni inserite.</li><li>5. Il sistema crea l'account dell'utente.</li><li>6. L'utente viene reindirizzato alla schermata di accesso. [10]</li></ol>
<b>Flusso alternativo</b>	<p>5.1 Se le informazioni inserite non sono valide, il sistema mostra un messaggio di errore. [8, 9]</p>
<b>Test</b>	UT-01-05, UT-01-05.1, IT-01, FT-01, FT-01-05.1

Tabella 1: Use Case UC-01

### 2.2.2 Sign in

<b>UC-02</b>	Sign in
<b>Scope</b>	Function
<b>Attori</b>	User
<b>Pre-condizioni</b>	L'utente ha un account registrato e non è autenticato nel sistema.
<b>Post-condizioni</b>	L'utente è autenticato nel sistema e può usufruire di un'interfaccia personalizzata in base al proprio ruolo.
<b>Flusso base</b>	<ol style="list-style-type: none"> <li>1. L'utente apre la schermata di accesso. [10]</li> <li>2. L'utente inserisce email e password.</li> <li>3. L'utente preme il pulsante di accesso.</li> <li>4. Il sistema verifica le informazioni inserite.</li> <li>5. L'utente viene autenticato e reindirizzato alla schermata degli appuntamenti. [14, 15]</li> </ol>
<b>Flusso alternativo</b>	<ol style="list-style-type: none"> <li>5.1 Se le informazioni inserite non sono valide, il sistema mostra un messaggio di errore. [11]</li> <li>5.2 Se l'utente ha dimenticato la password, può premere il pulsante di recupero password. [12]</li> </ol>
<b>Test</b>	UT-02-05, UT-02-05.1, IT-02, FT-02, FT-02-05.1

Tabella 2: Use Case UC-02

### 2.2.3 View Appointments

<b>UC-03</b>	View Appointments
<b>Scope</b>	User Goal
<b>Attori</b>	Customer, Barber
<b>Pre-condizioni</b>	L'utente è autenticato nel sistema.
<b>Post-condizioni</b>	L'utente vede l'elenco degli appuntamenti prenotati.
<b>Flusso base</b>	<ol style="list-style-type: none"> <li>1. L'utente apre la schermata degli appuntamenti. [14, 15]</li> <li>2. Il sistema mostra l'elenco degli appuntamenti prenotati.</li> <li>3. L'utente può visualizzare i dettagli di ogni appuntamento.</li> </ol>
<b>Flusso alternativo</b>	<p>2.1 Se l'utente non ha appuntamenti prenotati, il sistema mostra un messaggio di avviso. [17]</p>
<b>Test</b>	UT-03-02

Tabella 3: Use Case UC-03

#### 2.2.4 Add Appointment

<b>UC-04</b>	Add Appointment
<b>Scope</b>	User Goal
<b>Attori</b>	Customer
<b>Pre-condizioni</b>	Il customer è autenticato nel sistema.
<b>Post-condizioni</b>	Il customer ha prenotato un appuntamento.
<b>Flusso base</b>	<ol style="list-style-type: none"> <li>1. Il customer apre la schermata di prenotazione. [18]</li> <li>2. Il customer seleziona la data.</li> <li>3. Il customer seleziona il barbiere e il servizio. [19]</li> <li>4. Il sistema chiede se vuole aggiungere un altro servizio. [21]</li> <li>5. Il customer lo aggiunge e preme "yes".</li> <li>6. Il sistema mostra gli slot orari disponibili. [19]</li> <li>7. Il customer seleziona uno slot orario.</li> <li>8. Il sistema mostra un riepilogo dell'appuntamento. [23]</li> <li>9. Il customer conferma e seleziona il metodo di pagamento. [24]</li> <li>10. Il sistema registra l'appuntamento e invia una notifica al barber. [25]</li> </ol>
<b>Flusso alternativo</b>	<ol style="list-style-type: none"> <li>4.1 Se il customer non seleziona il servizio, il sistema mostra un messaggio di errore. [20]</li> <li>5.1 Il customer non aggiunge un altro servizio e preme "no".</li> <li>8.1 Se il customer seleziona no alla conferma, ritorna alla scelta degli slot orari. [19]</li> </ol>
<b>Test</b>	UT-04-6, IT-04, FT-04, FT-04-02, FT-04-04.1

Tabella 4: Use Case UC-04

### 2.2.5 Delete Appointment

<b>UC-05</b>	Delete Appointment
<b>Scope</b>	User Goal
<b>Attori</b>	Customer
<b>Pre-condizioni</b>	Il customer è autenticato nel sistema.
<b>Post-condizioni</b>	Il customer ha cancellato un appuntamento.
<b>Flusso base</b>	<ol style="list-style-type: none"> <li>1. Il customer apre la schermata degli appuntamenti. [15]</li> <li>2. Il customer clicca l'icona del cestino accanto all'appuntamento da cancellare.</li> <li>3. Il sistema chiede conferma per la cancellazione. [16]</li> <li>4. Il customer conferma la cancellazione.</li> <li>5. Il sistema rimuove l'appuntamento e notifica il barber e gli altri customers che lo slot è disponibile.</li> </ol>
<b>Flusso alternativo</b>	<p>4.1 Se il customer non conferma la cancellazione, l'appuntamento rimane nell'elenco.</p>
<b>Test</b>	UT-05-05, IT-05, FT-05,FT-05-04.1

Tabella 5: Use Case UC-05

### 2.2.6 View News

<b>UC-06</b>	View News
<b>Scope</b>	User Goal
<b>Attori</b>	Customer, Barber
<b>Pre-condizioni</b>	L'utente è autenticato nel sistema.
<b>Post-condizioni</b>	L'utente ha visualizzato le news.
<b>Flusso base</b>	<ol style="list-style-type: none"> <li>1. L'utente apre la schermata delle news. [26]</li> <li>2. Il sistema mostra l'elenco delle comunicazioni ricevute a partire dalla più recente.</li> </ol>
<b>Flusso alternativo</b>	<ol style="list-style-type: none"> <li>2.1 Se l'utente non ha ricevuto comunicazioni, il sistema mostra un messaggio di avviso. [27]</li> </ol>
<b>Test</b>	UT-06-02

Tabella 6: Use Case UC-06

### 2.2.7 View Profile

<b>UC-07</b>	View Profile
<b>Scope</b>	User Goal
<b>Attori</b>	Customer, Barber
<b>Pre-condizioni</b>	L'utente è autenticato nel sistema.
<b>Post-condizioni</b>	L'utente ha visualizzato le informazioni del profilo.
<b>Flusso base</b>	<ol style="list-style-type: none"> <li>1. L'utente apre la schermata del profilo. [34]</li> <li>2. Il sistema mostra le informazioni del profilo dell'utente.</li> </ol>
<b>Test</b>	UT-07-02, IT-07

Tabella 7: Use Case UC-07

### 2.2.8 View Services

<b>UC-08</b>	View Services
<b>Scope</b>	User Goal
<b>Attori</b>	Barber
<b>Pre-condizioni</b>	Il barber è autenticato nel sistema.
<b>Post-condizioni</b>	Il barber ha visualizzato i servizi disponibili.
<b>Flusso base</b>	<ol style="list-style-type: none"> <li>1. Il barber apre la schermata di gestione dei servizi. [30]</li> <li>2. Il sistema mostra l'elenco dei servizi disponibili.</li> </ol>
<b>Flusso alternativo</b>	<ol style="list-style-type: none"> <li>2.1 Se il barber non ha servizi disponibili, il sistema mostra un messaggio di avviso. [33]</li> </ol>
<b>Test</b>	IT-08-02

Tabella 8: Use Case UC-08

### 2.2.9 Add Service

<b>UC-09</b>	Add Service
<b>Scope</b>	User Goal
<b>Attori</b>	Barber
<b>Pre-condizioni</b>	Il barber è autenticato nel sistema.
<b>Post-condizioni</b>	Il barber ha aggiunto un nuovo servizio.
<b>Flusso base</b>	<ol style="list-style-type: none"> <li>1. Il barber apre la schermata di gestione dei servizi. [30]</li> <li>2. Il barber inserisce il nome e il prezzo del servizio.</li> <li>3. Il sistema registra il nuovo servizio e lo mostra nell'elenco dei servizi disponibili.</li> </ol>
<b>Flusso alternativo</b>	<ol style="list-style-type: none"> <li>2.1 Se il barber non inserisce il nome e/o il prezzo, il sistema mostra un messaggio di errore. [31]</li> </ol>
<b>Test</b>	UT-09-02.1, UT-09-03, IT-09, FT-09, FT-09-02.1

Tabella 9: Use Case UC-09

### 2.2.10 Delete Service

<b>UC-10</b>	Delete Service
<b>Scope</b>	User Goal
<b>Attori</b>	Barber
<b>Pre-condizioni</b>	Il barber è autenticato nel sistema.
<b>Post-condizioni</b>	Il barber ha eliminato un servizio.
<b>Flusso base</b>	<ol style="list-style-type: none"> <li>1. Il barber apre la schermata di gestione dei servizi. [30]</li> <li>2. Il barber clicca l'icona del cestino accanto al servizio da rimuovere.</li> <li>3. Il sistema chiede conferma per l'eliminazione del servizio. [32]</li> <li>4. Il barber conferma l'eliminazione.</li> <li>5. Il sistema rimuove il servizio dall'elenco.</li> </ol>
<b>Flusso alternativo</b>	<p>4.1 Se il barber non conferma l'eliminazione, il servizio rimane nell'elenco.</p>
<b>Test</b>	UT-10-05, IT-10, FT-10, FT-10-04.1

Tabella 10: Use Case UC-10

### 2.2.11 Send Communication

<b>UC-11</b>	Send Comunication
<b>Scope</b>	User Goal
<b>Attori</b>	Barber
<b>Pre-condizioni</b>	Il barber è autenticato nel sistema.
<b>Post-condizioni</b>	Il barber ha inviato una comunicazione.
<b>Flusso base</b>	<ol style="list-style-type: none"> <li>1. Il barber apre la schermata di invio comunicazioni. [28]</li> <li>2. Il barber inserisce il testo e il titolo della comunicazione.</li> <li>3. Il barber preme il pulsante di invio.</li> <li>4. Il sistema invia la comunicazione a tutti i clienti.</li> </ol>
<b>Flusso alternativo</b>	<p>2.1 Se il barber non inserisce il testo, il sistema mostra un messaggio di errore. [29]</p>
<b>Test</b>	UT-11-02.1, UT-11-04, IT-11, FT-11, FT-11-02.1

Tabella 11: Use Case UC-11

## 2.3 Class Diagram

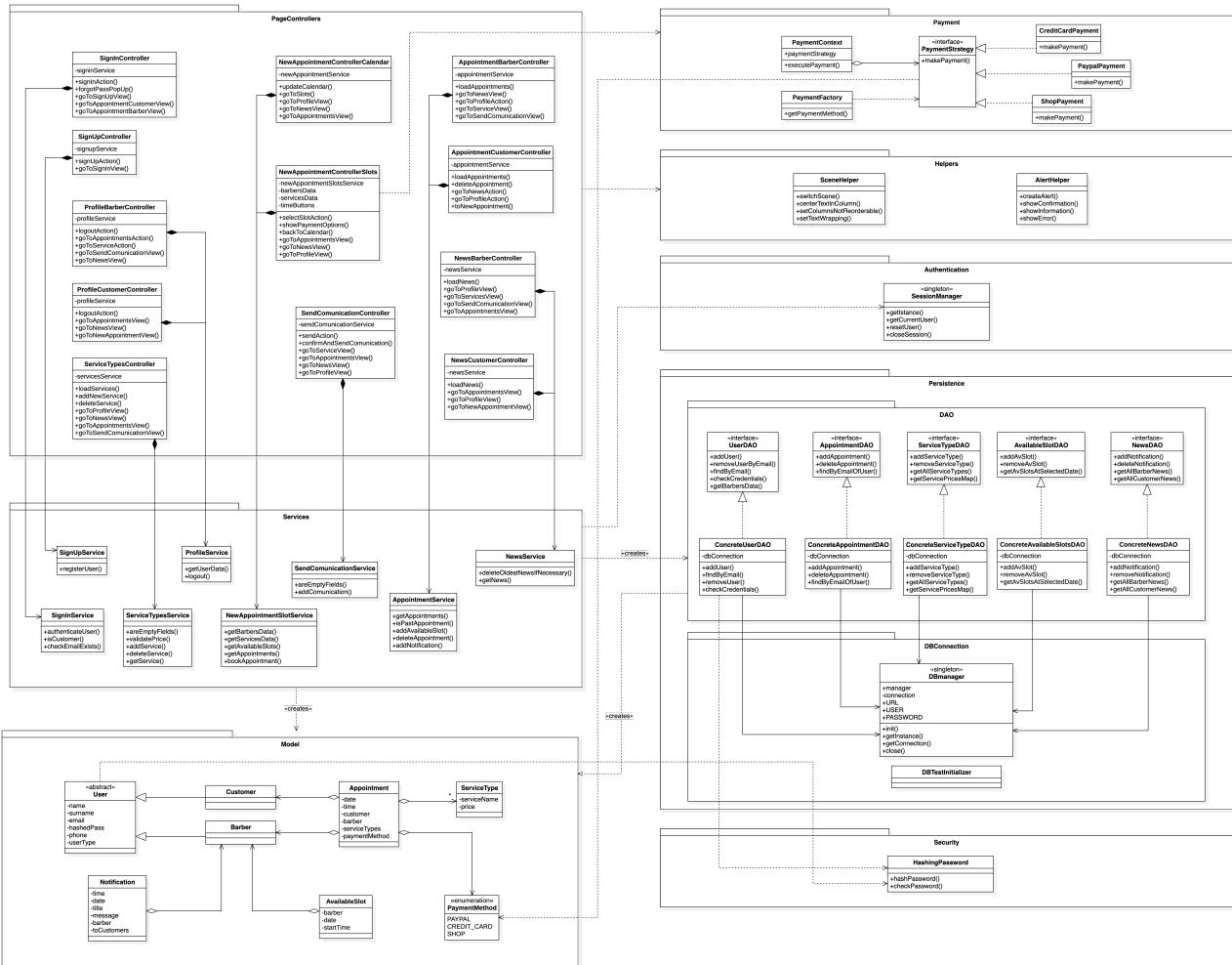


Figura 4: Class Diagram

## 2.4 Page Navigation Diagram

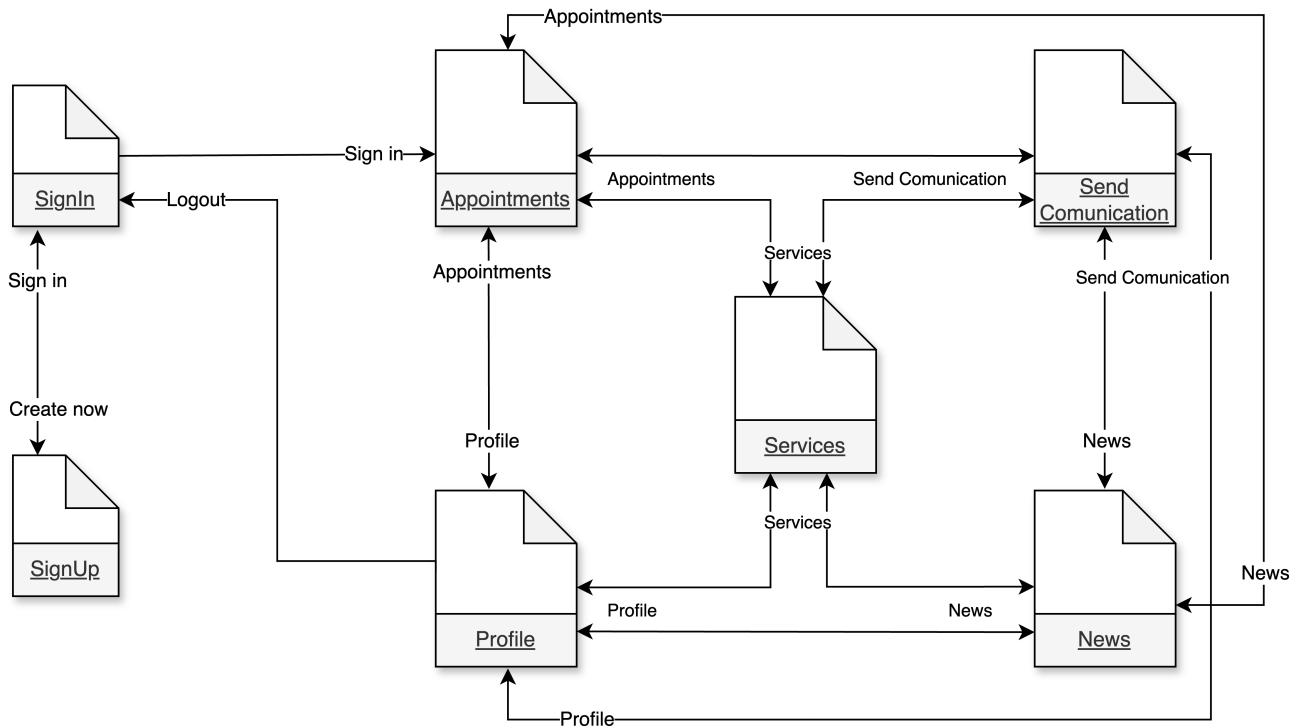


Figura 5: Page Navigation Diagram - Barber

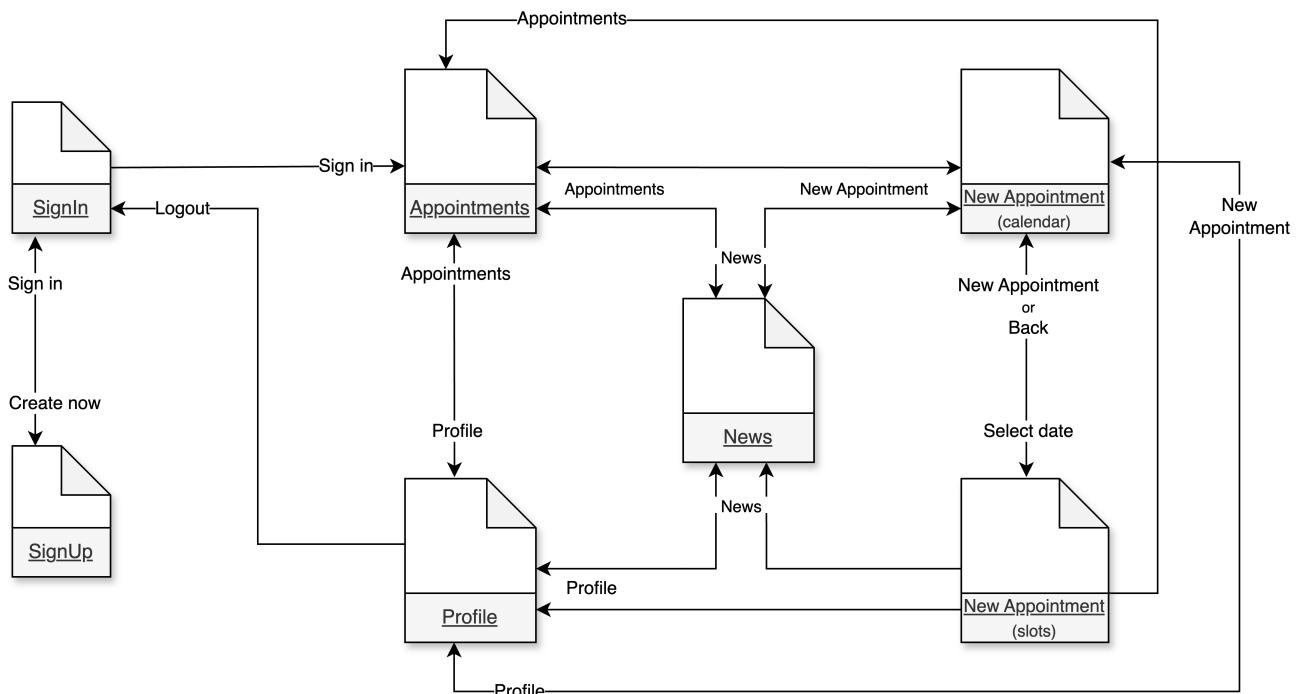


Figura 6: Page Navigation Diagram - Customer

## 2.5 Interfaccia Grafica

### 2.5.1 Registrazione e Accesso

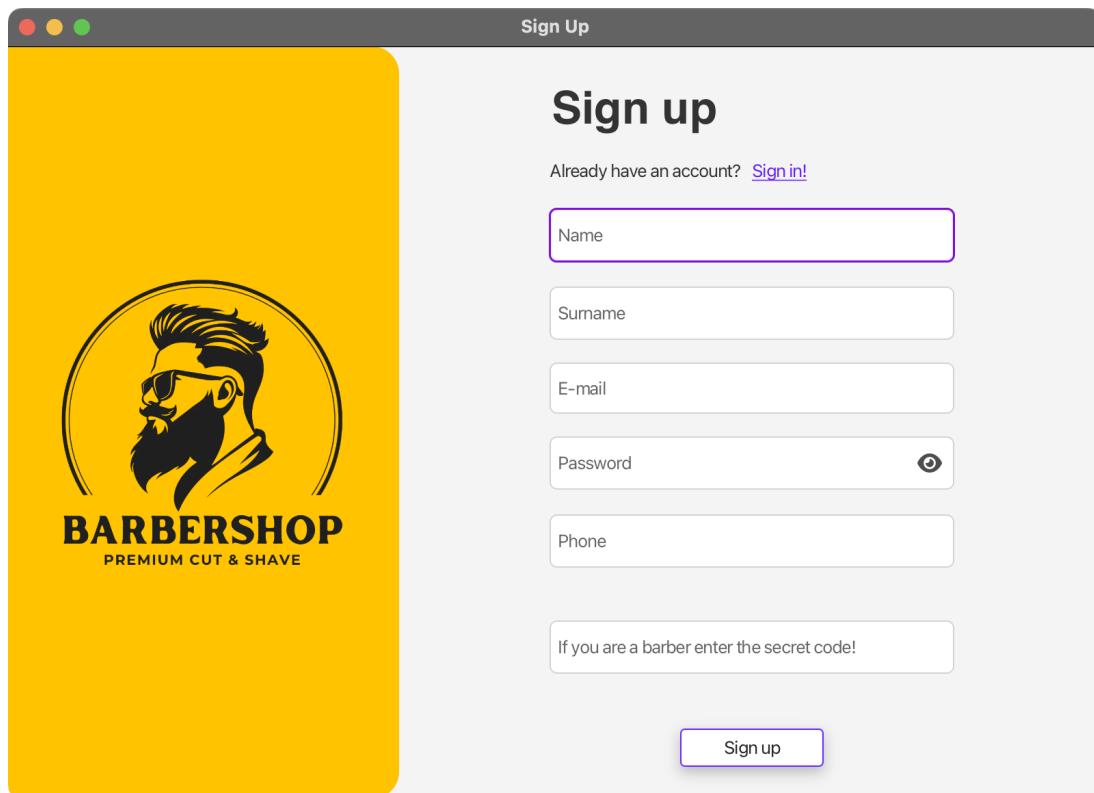


Figura 7: Registrazione

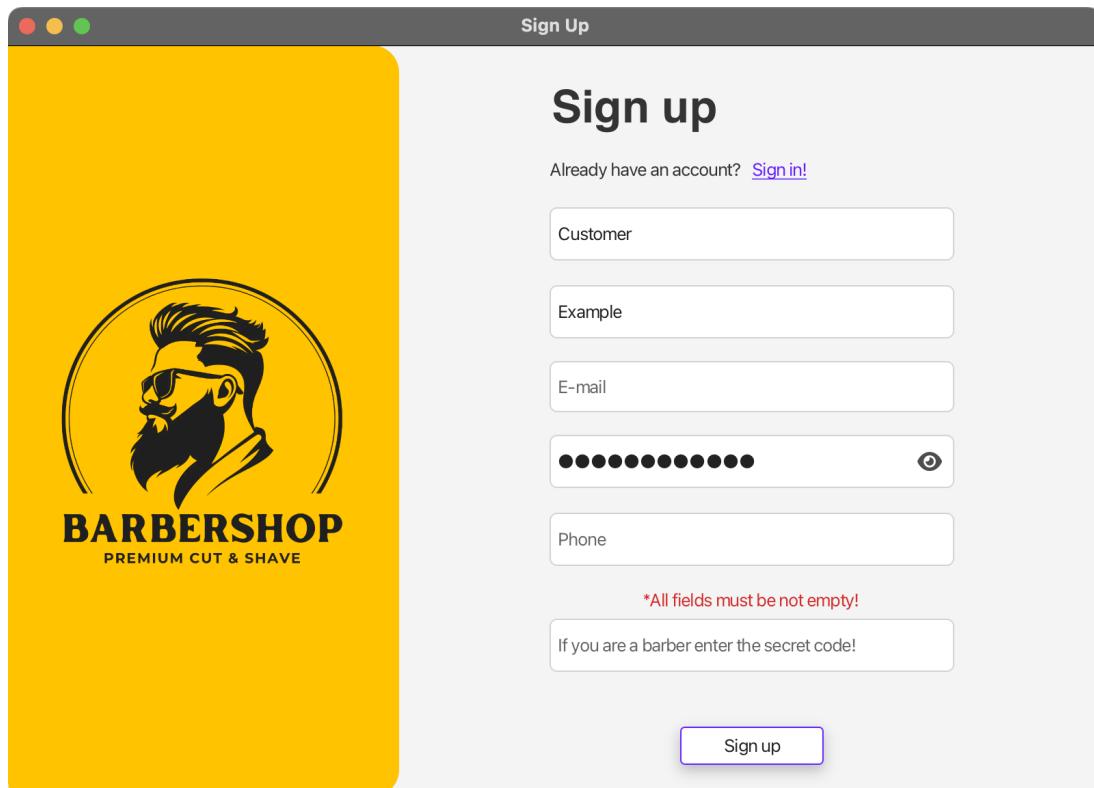


Figura 8: Registrazione - campi vuoti

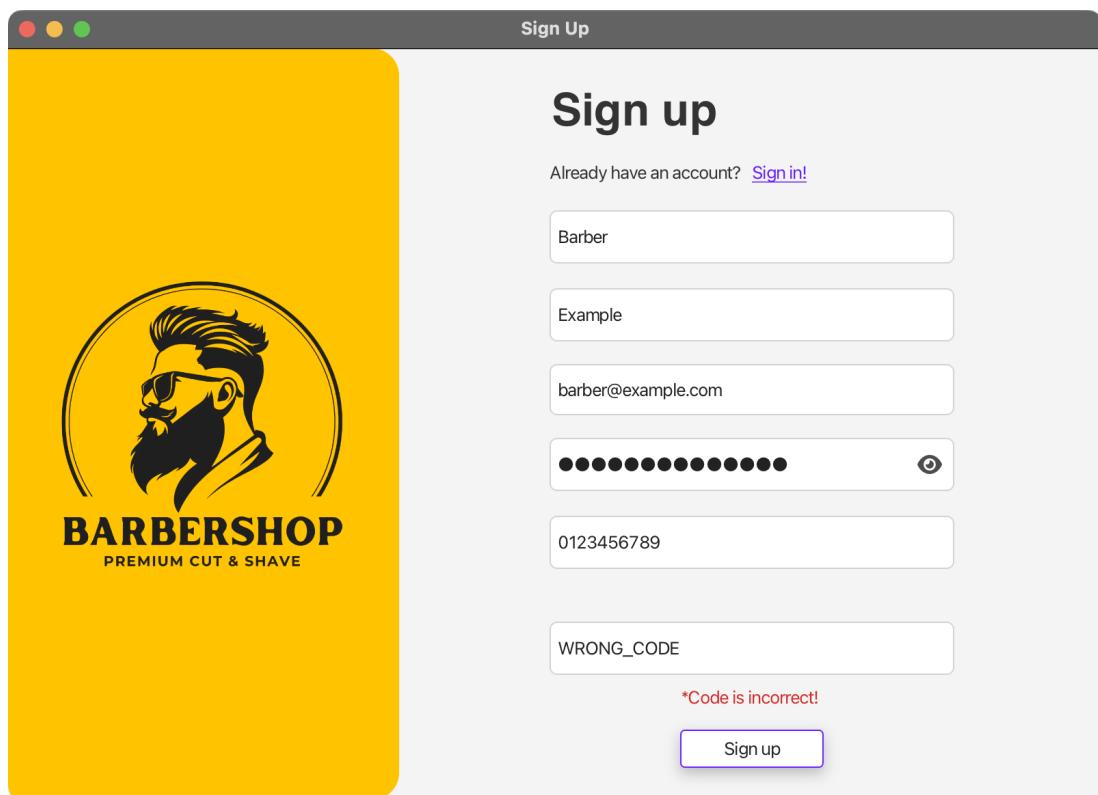


Figura 9: Registrazione - codice sbagliato

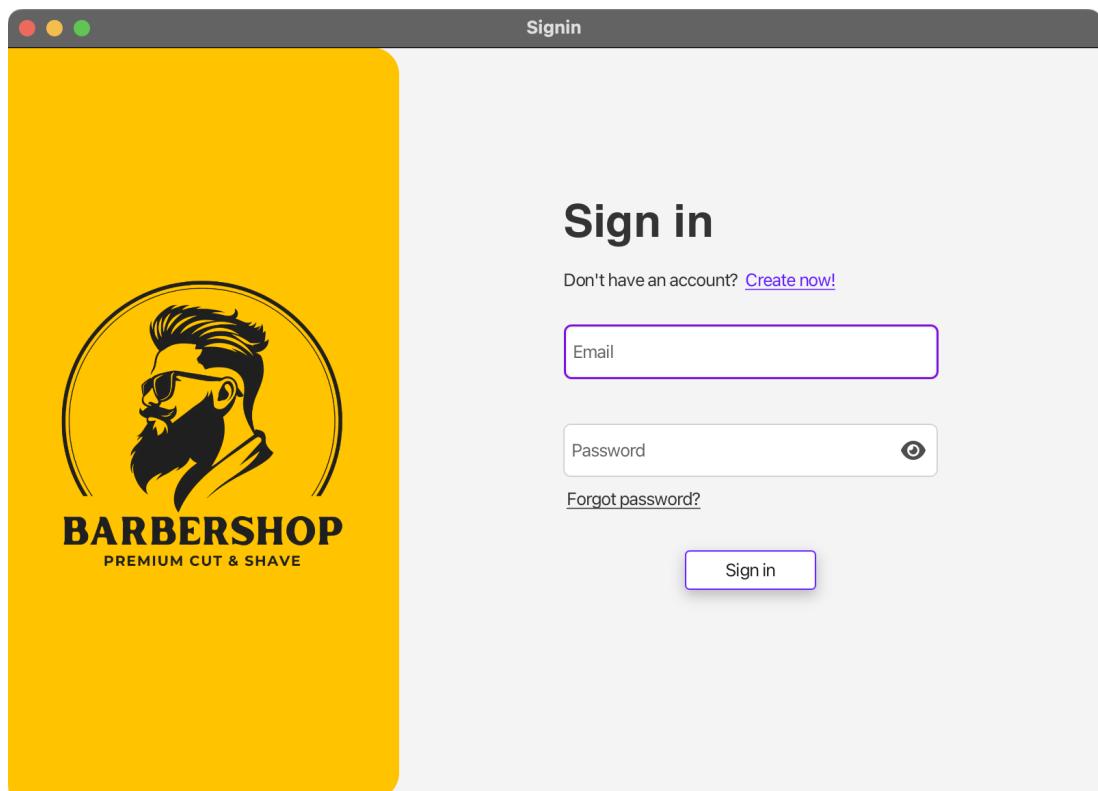


Figura 10: Accesso

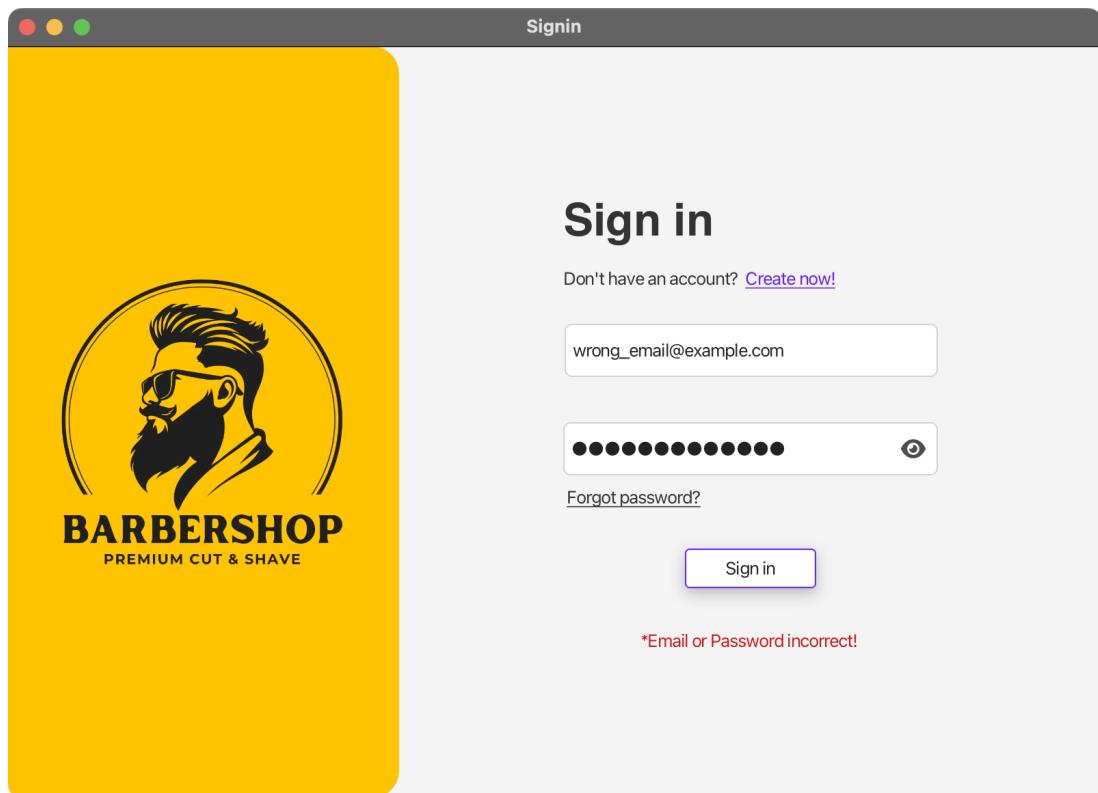


Figura 11: Accesso - credenziali errate

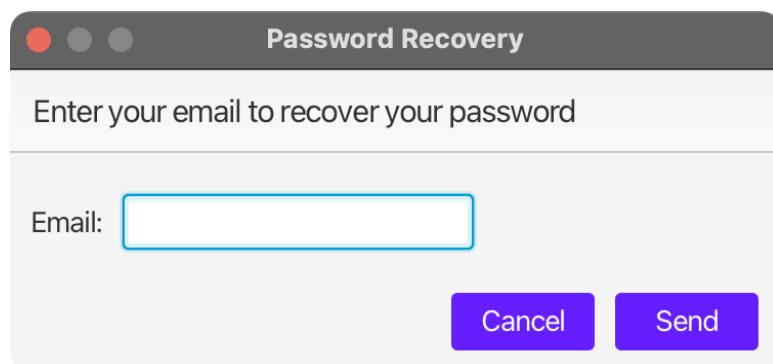


Figura 12: Recupero password

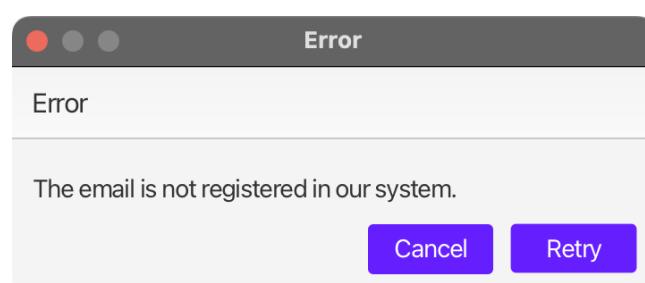


Figura 13: Email non registrata

## 2.5.2 Visualizzazione appuntamenti

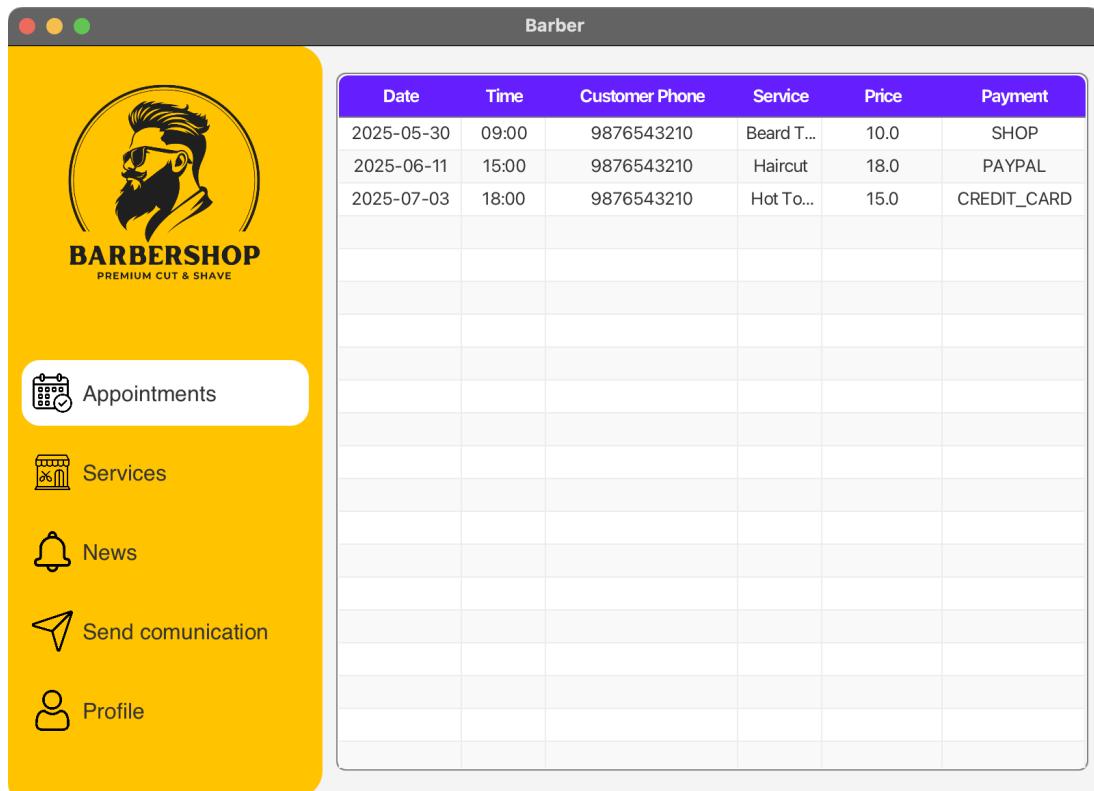


Figura 14: Appuntamenti - barber

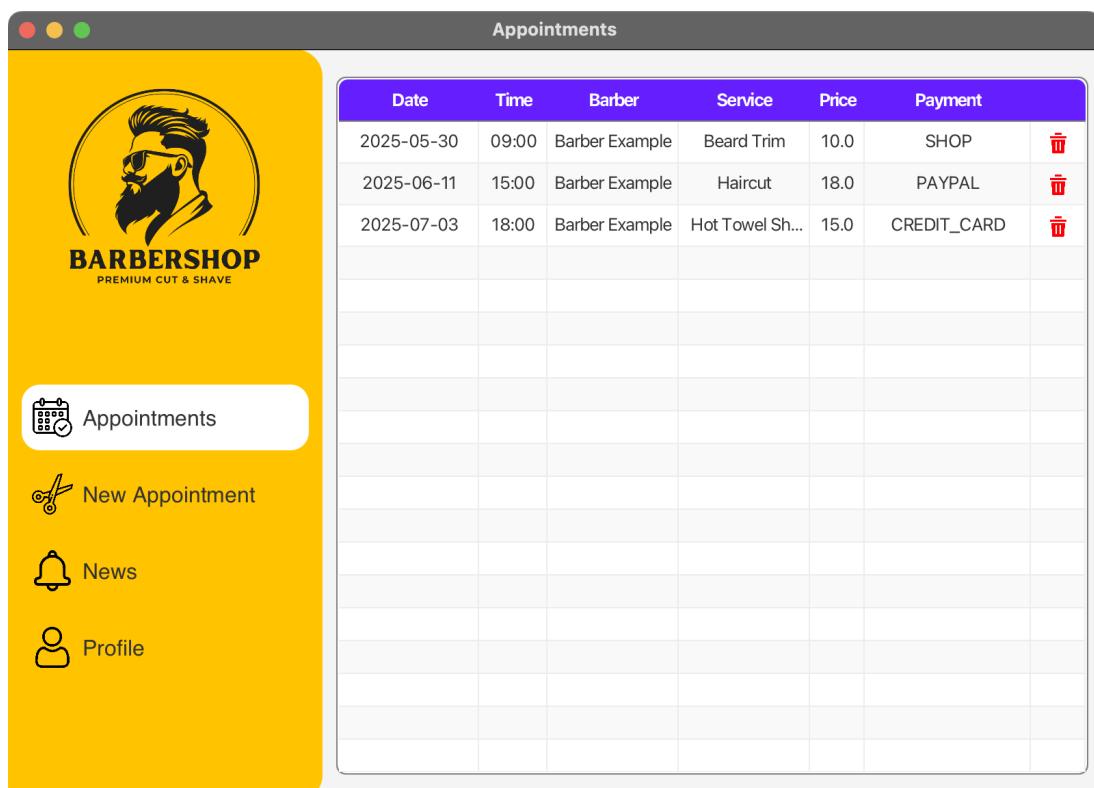


Figura 15: Appuntamenti - customer

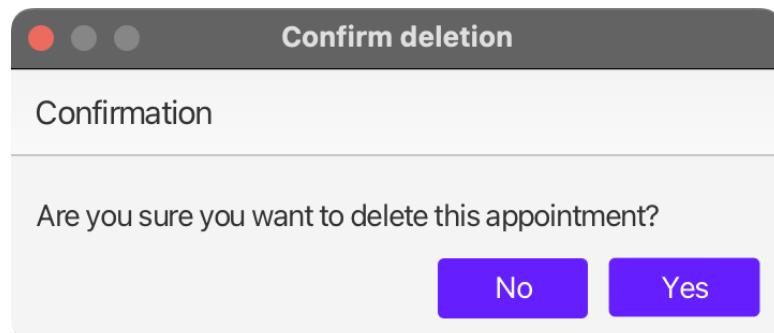


Figura 16: Cancellazione appuntamento

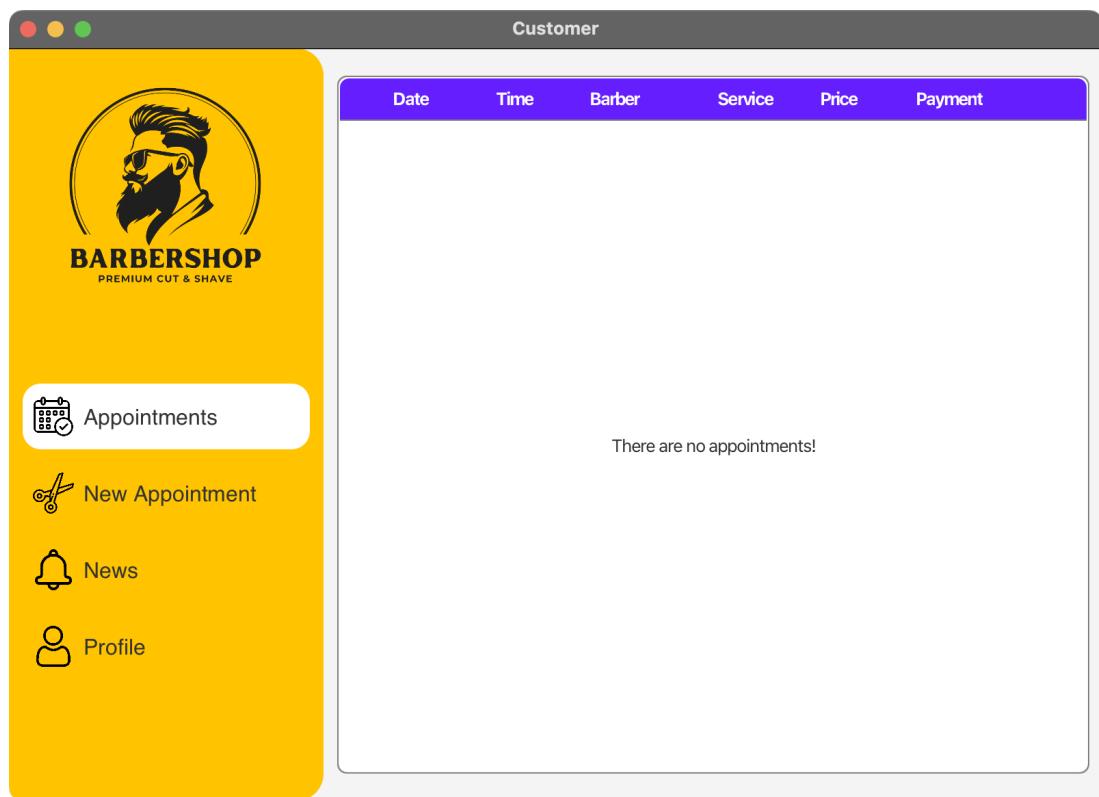


Figura 17: Nessun appuntamento prenotato

### 2.5.3 Prenotazione appuntamenti

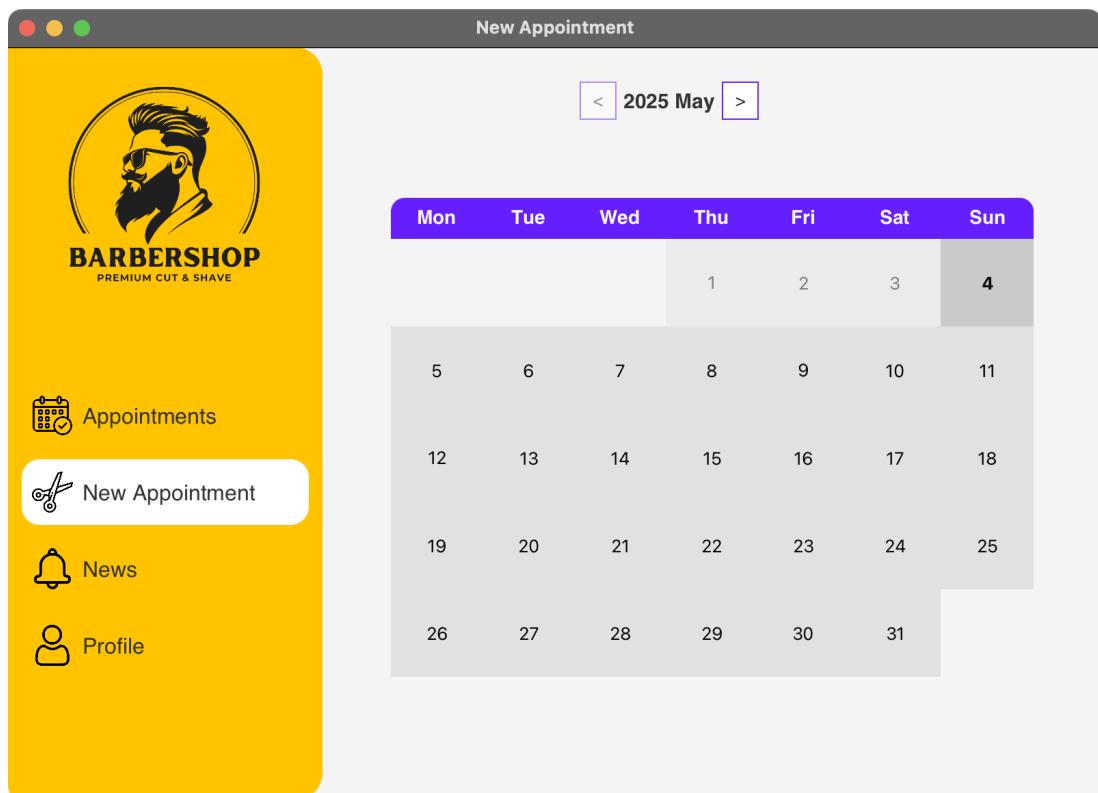


Figura 18: Selezione data

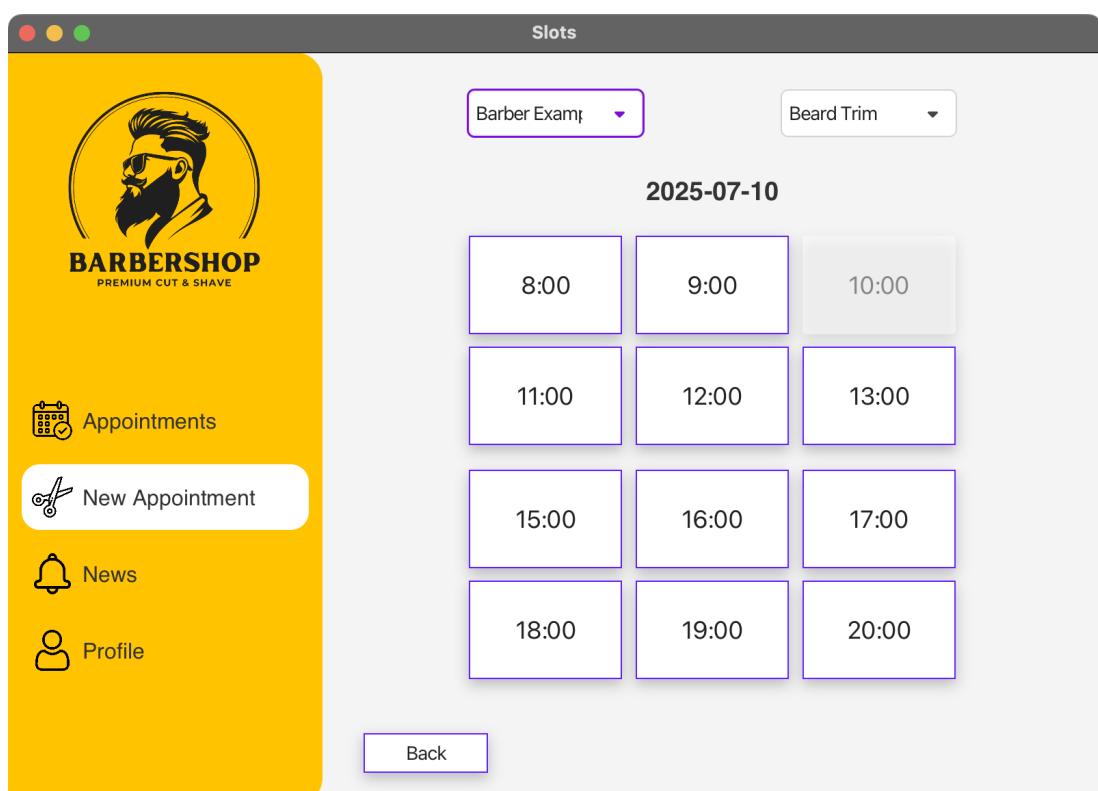


Figura 19: Selezione slot orario, barbiere e servizio

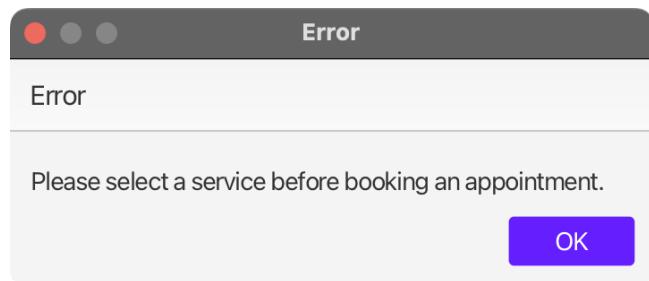


Figura 20: Servizio non selezionato

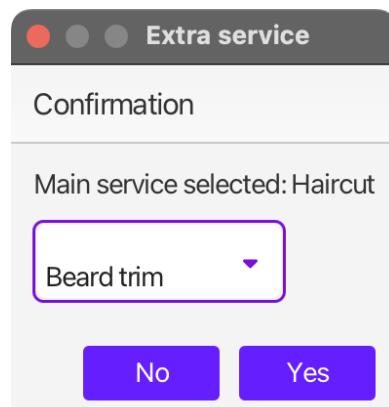


Figura 21: Aggiunta servizio extra

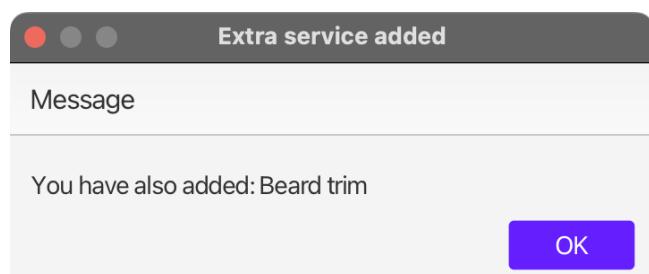


Figura 22: Servizio extra aggiunto

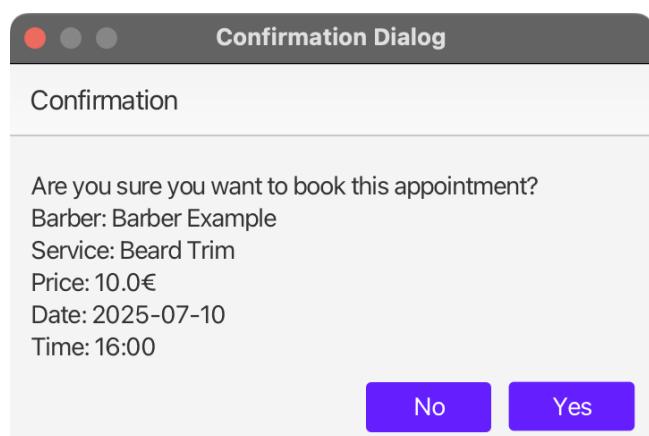


Figura 23: Riepilogo appuntamento

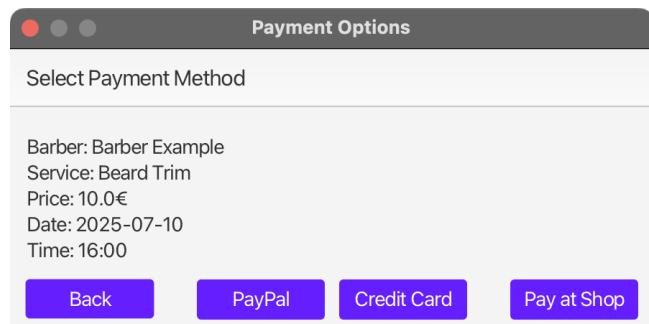


Figura 24: Selezione metodo di pagamento

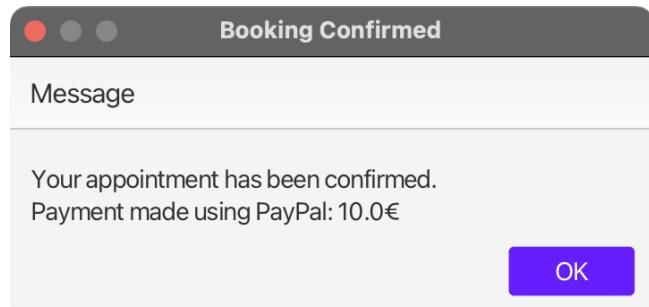


Figura 25: Appuntamento prenotato

## 2.5.4 News e Invio delle Comunicazioni

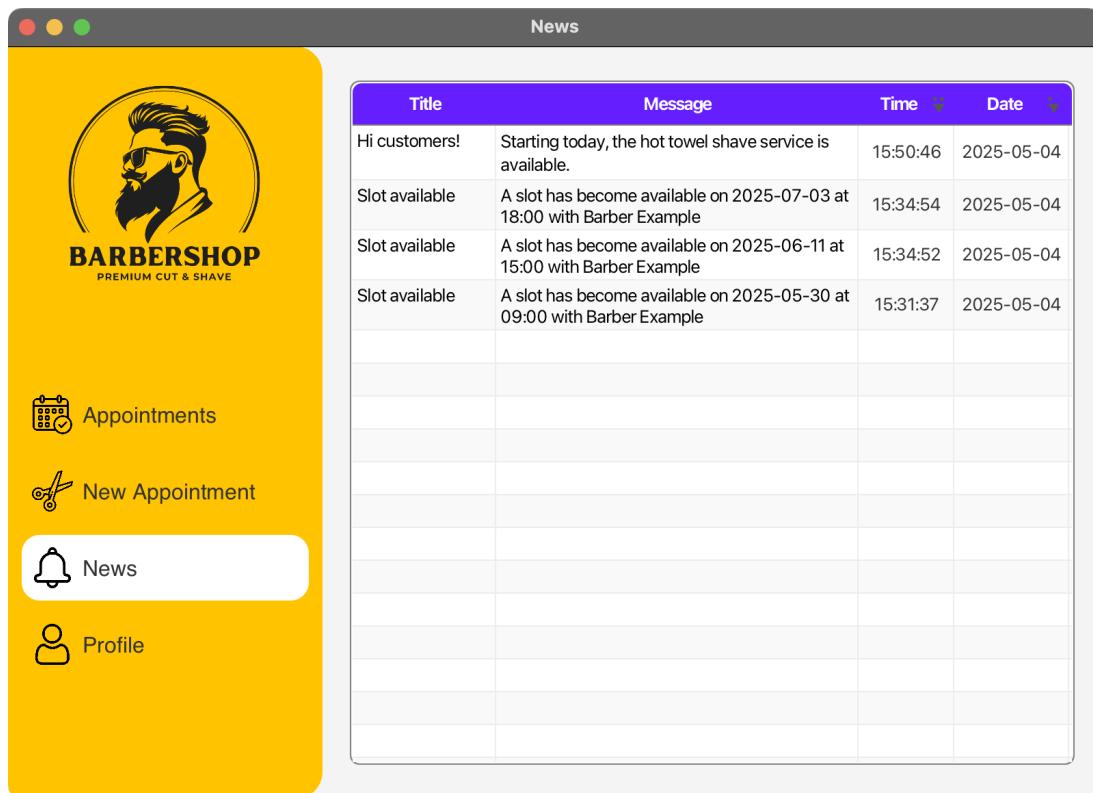


Figura 26: News

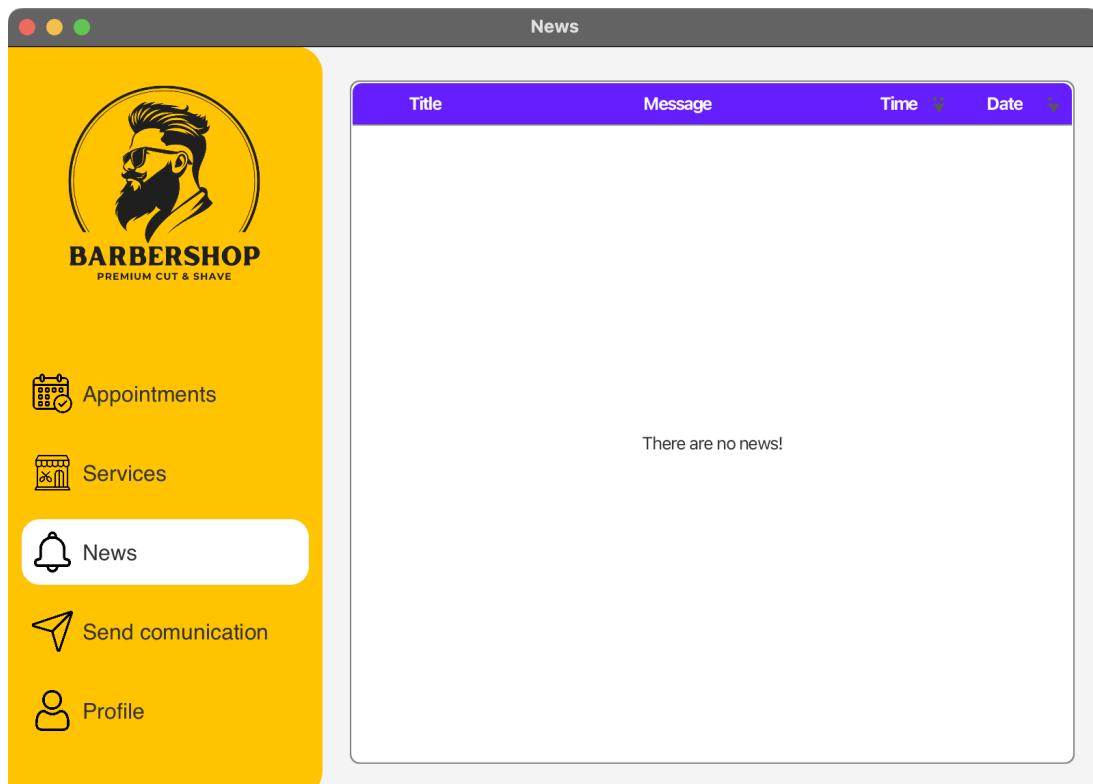


Figura 27: Nessuna notifica

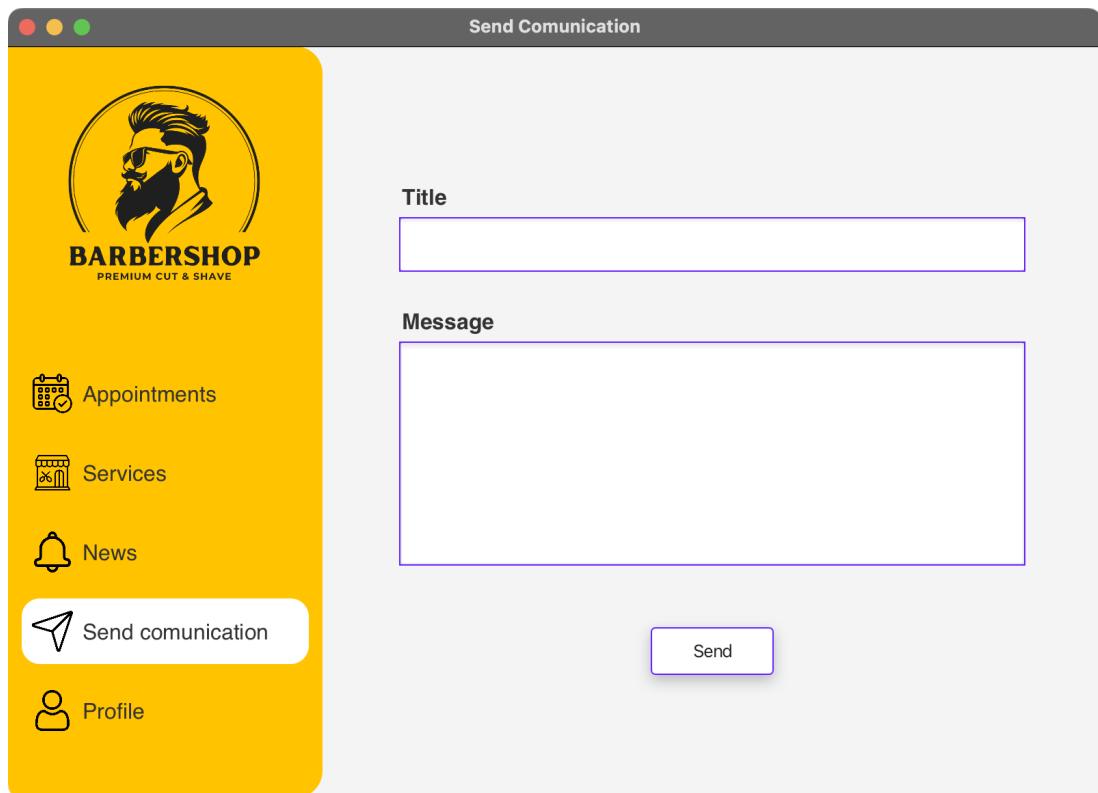


Figura 28: Invio comunicazioni

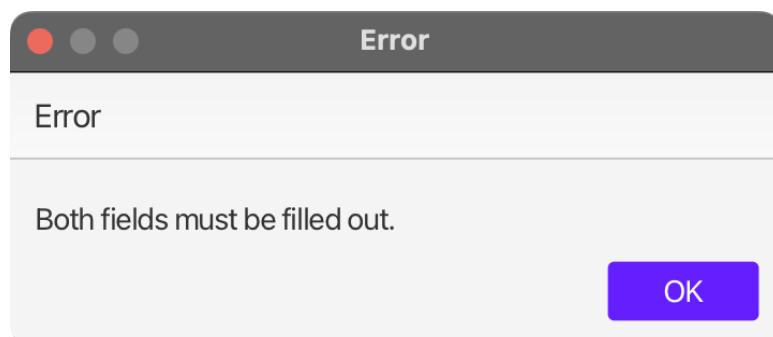


Figura 29: Campi comunicazione vuoti

### 2.5.5 Gestione Servizi

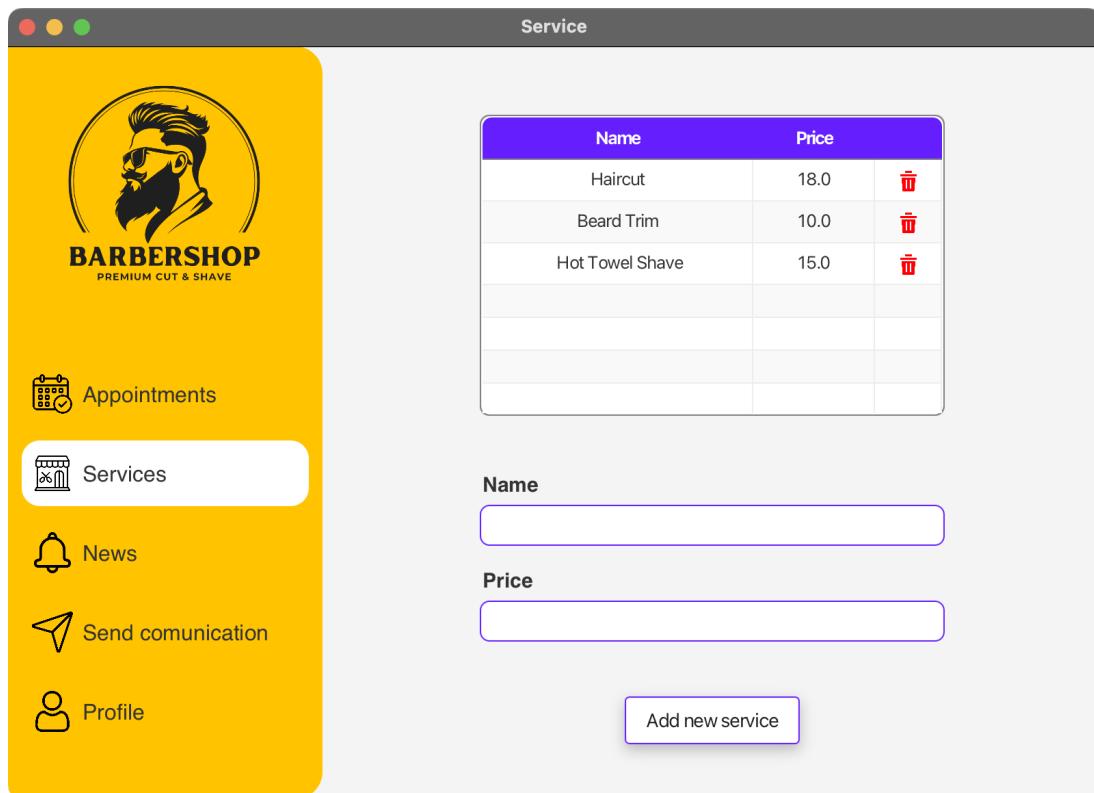


Figura 30: Servizi

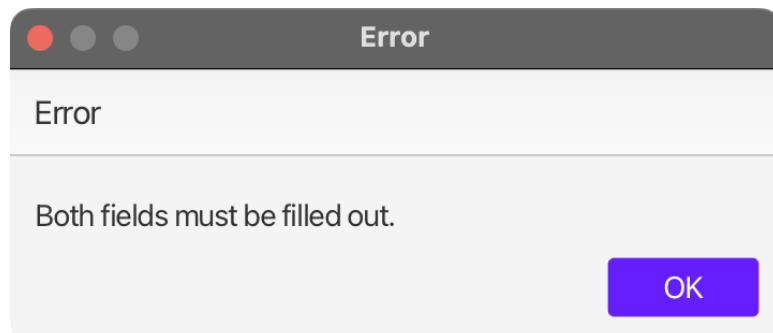


Figura 31: Campi servizio vuoti

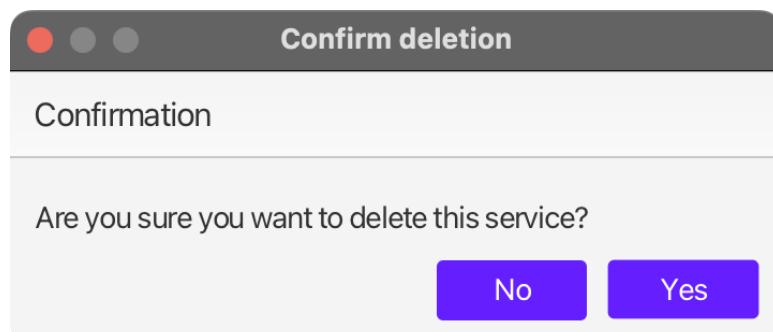


Figura 32: Cancellazione servizio

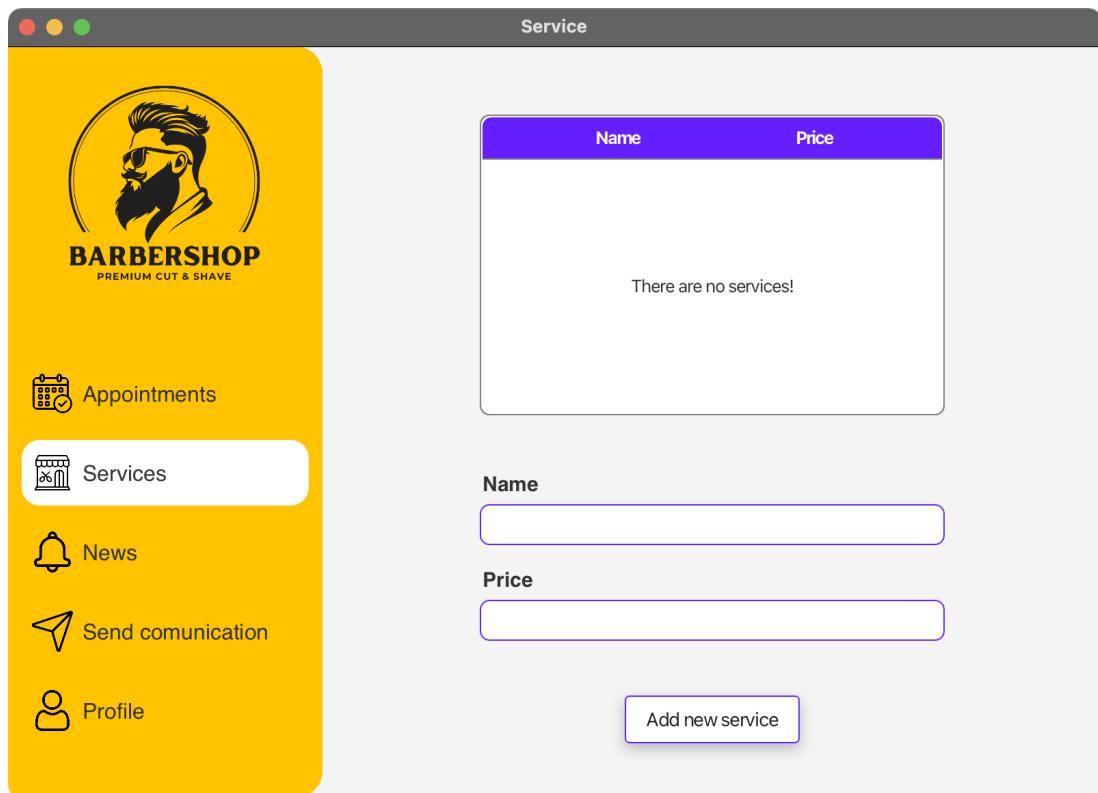


Figura 33: Nessun servizio

## 2.5.6 Profilo

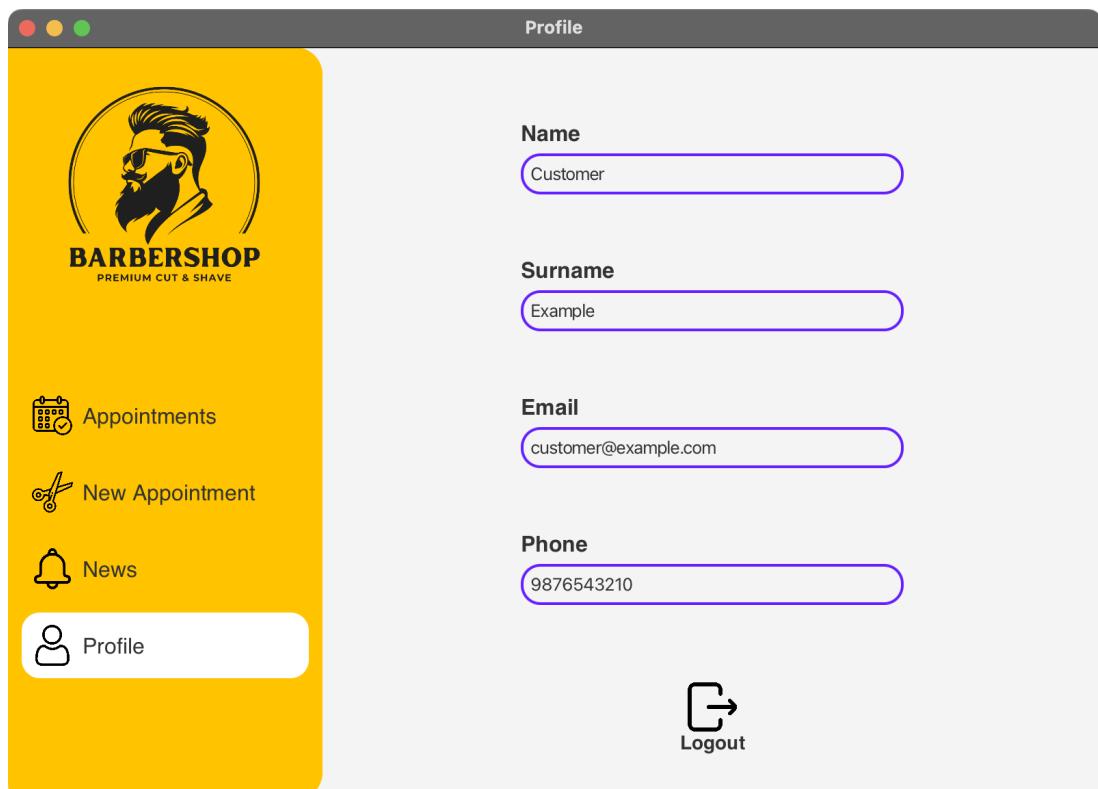


Figura 34: Informazioni del profilo

## 2.6 Database

### 2.6.1 Entity-Relationship Diagram

Il diagramma **Entity-Relationship** (ER) rappresenta in modo formale le entità principali del sistema, i loro attributi e le relazioni che le collegano. Il diagramma ER in figura 35 descrive la struttura concettuale di un'applicazione per la gestione di un Barber Shop, evidenziando l'organizzazione degli appuntamenti tra clienti e barbieri, gli slot orari, i servizi offerti e l'invio di comunicazioni (news).

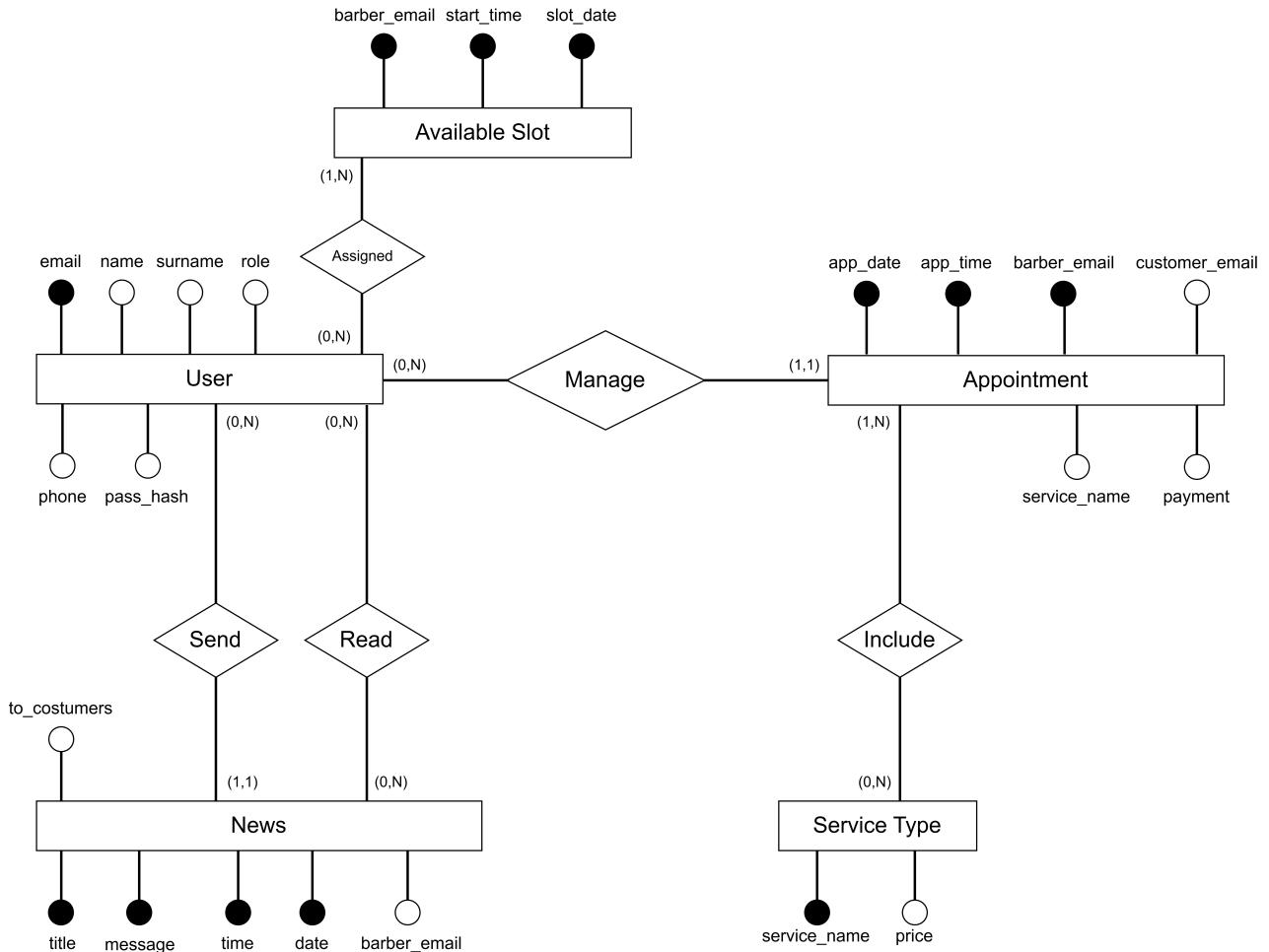


Figura 35: Entity-Relationship Diagram

### 2.6.2 Modello relazionale

La tabella **User** rappresenta tutti gli utenti del sistema, distinguendo tra barbieri e clienti tramite l'attributo **role**. L'attributo **email** è stato scelto come chiave primaria, poiché garantisce l'unicità di ogni utente registrato. L'introduzione di un ulteriore attributo **id** è stata ritenuta superflua, in quanto ridondante e priva di reale necessità.

La tabella **Available\_Slot** rappresenta le disponibilità orarie dei barbieri, mentre l'entità **Appointment** modella gli appuntamenti fissati tra clienti e barbieri. Per entrambe le entità, la chiave primaria è costituita dalla combinazione degli attributi **date**, **time** e **barber\_email**. Questa scelta consente di garantire l'unicità di ciascuna occorrenza, poiché gli slot orari sono condivisi tra tutti i barbieri, rendendo insufficienti i soli attributi **date** e **time**.

La tabella **Appointment\_Services** rappresenta la relazione tra gli appuntamenti e i servizi associati. La chiave primaria è costituita dalla combinazione degli attributi **app\_date**,

`app_time`, `barber_email` e `service_name`. Questa scelta consente di garantire l'unicità di ciascuna occorrenza, poiché un barbiere può offrire più servizi nello stesso appuntamento.

La tabella `Service_Type` definisce le tipologie di servizi offerti dal Barber Shop. L'attributo `service_name`, vincolato a essere `unique` e `not null`, è stato scelto come chiave primaria.

Infine, la tabella `News` gestisce le comunicazioni, sia quelle inviate manualmente dai barbieri ai clienti, sia quelle generate automaticamente dal sistema. Esempi di notifiche automatiche includono quelle inviate quando un cliente prenota o annulla un appuntamento, notificando il barbiere, oppure quando uno slot diventa disponibile a seguito della cancellazione di una prenotazione. La chiave primaria è costituita dalla combinazione degli attributi `title`, `message`, `date` e `time`, poiché i soli `title` e `message` non sarebbero sufficienti a garantire l'unicità.

**Users** (email, name, surname, pass\_hash, phone, role)

**Service\_Types** (service\_name, price)

**Appointments** (app\_date, app\_time, barber\_email, customer\_email, payment)

FK (`barber_email`) REF (**Users**)

FK (`customer_email`) REF (**Users**)

**Appointments\_Services** (app\_date, app\_time, barber\_email, service\_name)

FK (`barber_email`) REF (**Users**)

FK (`service_name`) REF (**Service\_Types**)

FK (`app_date`, `app_time`, `barber_email`) REF (**Appointments**)

**Available\_Slots** (barber\_email, slot\_date, start\_time)

FK (`barber_email`) REF (**Users**)

**News** (title, message, time, date, barber\_email, to\_costumers)

FK (`barber_email`) REF (**Users**)

### 3 Implementazione

#### 3.1 Model

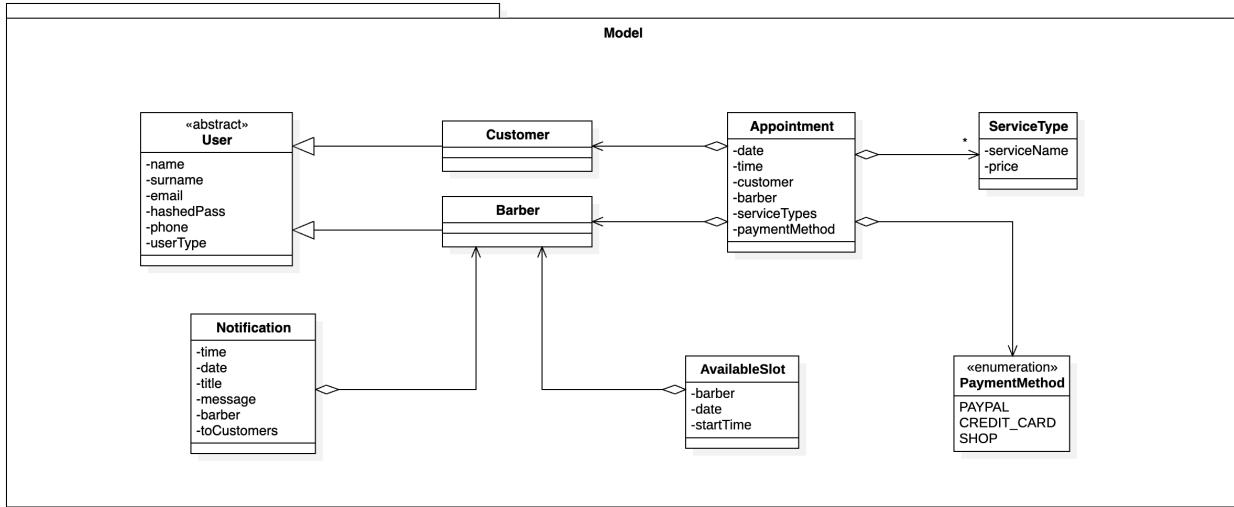


Figura 36: Model

Il *Domain Model* illustrato in figura 36 descrive la struttura e le relazioni tra le principali entità del sistema. La classe **User** è progettata come una superclasse astratta, condividendo attributi e metodi comuni con le sue sottoclassi **Customer** e **Barber**. Questa scelta garantisce una progettazione modulare e riutilizzabile, evitando l'instanziazione diretta della classe **User**. La classe **Appointment** rappresenta gli appuntamenti prenotati dai clienti e stabilisce una relazione di aggregazione con le classi **Customer** e **Barber**. Inoltre, include un riferimento alla classe **ServiceType**, che definisce i servizi offerti dai barbieri. Gli attributi della classe **Appointment** completano i dettagli fondamentali dell'appuntamento, come data, orario e metodo di pagamento, fornendo una rappresentazione completa e coerente delle informazioni necessarie. La classe **AvailableSlot** rappresenta gli slot orari disponibili per la prenotazione degli appuntamenti e include un riferimento alla classe **Barber**, poiché più barbieri possono condividere gli stessi slot orari. Questo approccio consente una gestione flessibile e scalabile degli orari disponibili. La classe **Notification** rappresenta le comunicazioni inviate dai barbieri ai clienti, nonché le notifiche relative a prenotazioni cancellate o slot liberati. La classe include un riferimento alla classe **Barber** e un attributo booleano **toCustomers**, che specifica se la notifica deve essere inviata ai clienti. Questo design garantisce una gestione chiara e organizzata delle notifiche all'interno del sistema. Infine, l'enumerazione **PaymentMethod** definisce i metodi di pagamento disponibili per gli appuntamenti, come PayPal, carta di credito o pagamento in negozio. Questa enumerazione consente una gestione semplice e coerente dei metodi di pagamento, facilitando l'integrazione con eventuali sistemi di pagamento esterni.

## 3.2 Helpers

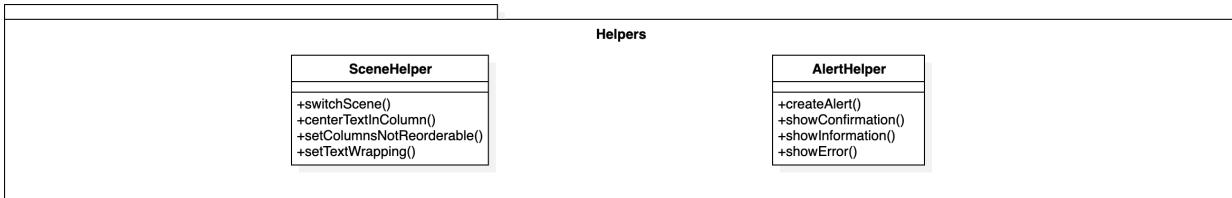


Figura 37: Helpers

La figura 37 mostra il pacchetto *Helpers* contenente le classi di supporto implementate nel progetto, progettate per semplificare e ottimizzare le operazioni comuni dei controller. Tra queste, la classe **SceneHelper** centralizza la logica di navigazione attraverso il metodo statico `switchScene()`, migliorando la manutenibilità del codice e riducendo la duplicazione.

Un'altra classe di rilievo è **AlertHelper**, che fornisce metodi statici per creare alert personalizzati, come quelli di errore, informativi e di conferma. Questo approccio garantisce un utilizzo uniforme degli alert in tutto il progetto, rendendo il codice più leggibile e organizzato.

Di seguito è riportato il metodo `switchScene()`, che consente di cambiare scena in modo efficiente. Il metodo accetta un nodo esistente, un percorso FXML e un titolo come parametri. Utilizza `FXMLLoader` per caricare il file FXML specificato, crea una nuova scena con il contenuto caricato e la imposta nella finestra corrente:

```
1  public static void switchScene(Node node, String fxmlPath, String title) {
2      try {
3          FXMLLoader loader = new FXMLLoader(SceneHelper.class.getResource(fxmlPath));
4          Parent root = loader.load();
5
6          Stage stage = (Stage) node.getScene().getWindow();
7          stage.setScene(new Scene(root));
8          stage.setTitle(title);
9          stage.show();
10     } catch (IOException e) {
11         e.printStackTrace();
12     }
13 }
```

Un metodo importante presente nella classe **AlertHelper** è `createAlert()`, che consente di creare alert personalizzati in base al tipo specificato (ad esempio, errore, informativo o di conferma). Il metodo accetta tre parametri: il tipo di alert (`AlertType`), il titolo e il messaggio. Utilizza la libreria JavaFX per generare l'alert e applica uno stile CSS personalizzato, garantendo un aspetto uniforme e coerente con il design dell'applicazione:

```
1  private static Alert createAlert(AlertType type, String title, String message) {
2      Alert alert = new Alert(type);
3      alert.setTitle(title);
4      alert.setContentText(message);
5      alert.getDialogPane().getStylesheets().add(ALERT_CSS);
6      alert.getDialogPane().getStyleClass().add(CUSTOM_ALERT_CLASS);
7      return alert;
8  }
```

### 3.3 PageControllers

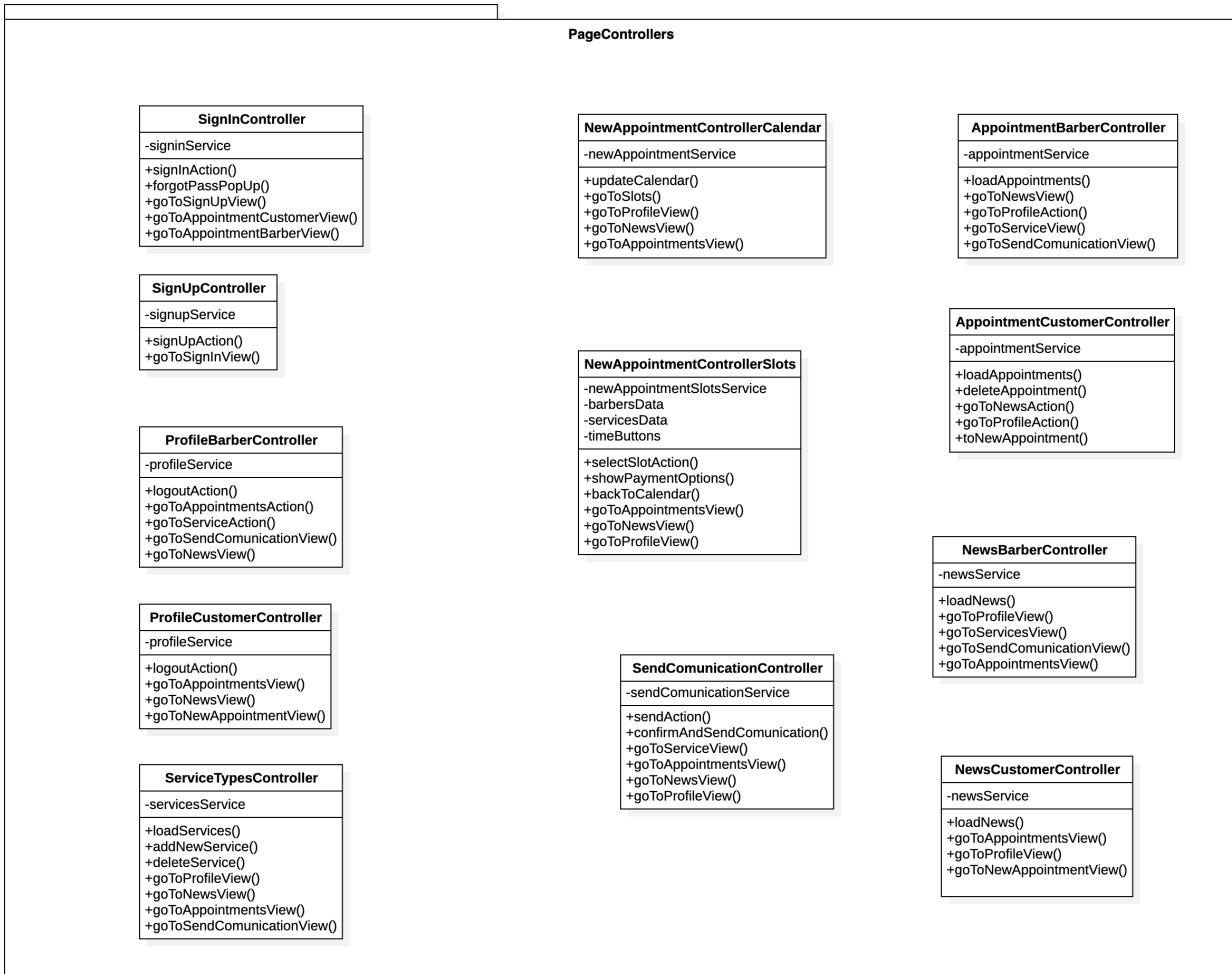


Figura 38: PageControllers

I controller gestiscono l’interazione tra l’interfaccia utente e la logica di business, occupandosi di specifiche schermate dell’applicazione per garantire una navigazione fluida e coerente. La figura 38 mostra il pacchetto *PageControllers*, che contiene le classi controller implementate, evidenziandone gli attributi e i metodi principali.

#### Ruolo dei Controller

I controller fungono da intermediari tra le interazioni dell’utente e la logica di business, implementata nei servizi del pacchetto *Services*. Le loro principali responsabilità includono:

- Gestione degli eventi dell’interfaccia utente, come i click sui pulsanti.
- Aggiornamento della vista in base alle azioni dell’utente.
- Recupero dei dati dai servizi e passaggio all’interfaccia utente per la visualizzazione.

In ogni controller viene dichiarata e inizializzata tra gli attributi una **reference variable** al **service** corrispondente il quale fornisce le funzionalità necessarie, inoltre ognuno di essi include metodi per il cambio di scena, che sfruttano il metodo statico **switchScene()** della classe **SceneHelper** (illustrata nella figura 37). Questo approccio centralizzato riduce la duplicazione del codice e migliora la manutenibilità complessiva del progetto.

## Suddivisione dei Controller

I controller sono stati progettati seguendo una suddivisione basata sul ruolo dell’utente, distinguendo tra quelli destinati ai barbieri e quelli riservati ai clienti. Questa scelta architetturale consente di offrire interfacce personalizzate e di ottimizzare la gestione delle funzionalità specifiche per ciascun profilo. Nonostante alcune operazioni risultino simili tra i vari controller, la separazione è stata adottata con l’obiettivo di migliorare l’esperienza utente e garantire una maggiore coerenza nell’interazione con il sistema.

Gli unici controller condivisi tra i due tipi di utenti sono:

- **SignUpController**: Responsabile della registrazione.
- **SignInController**: Dedicato alla gestione dell’accesso.

## Controller Specifici

- **Profilo Utente**:

- **ProfileCustomerController** e **ProfileBarberController** gestiscono le schermate relative alle informazioni dell’utente autenticato.

- **News e Comunicazioni**:

- **NewsBarberController** e **NewsCustomerController** si occupano della visualizzazione delle comunicazioni ricevute.
  - **SendCommunicationController** gestisce l’interfaccia dedicata all’invio delle comunicazioni ai clienti.

- **Appuntamenti**:

- **AppointmentBarberController** e **AppointmentCustomerController** gestiscono la visualizzazione e la gestione degli appuntamenti prenotati. Le informazioni mostrate nelle tabelle sono personalizzate in base al tipo di utente.
  - Nel caso del controller dedicato ai clienti, un alert di conferma viene mostrato prima di eliminare un appuntamento. Solo dopo la conferma, il controller delega l’operazione di eliminazione al servizio specifico.

- **Servizi**:

- **ServiceTypesController** gestisce la visualizzazione e la gestione dei servizi offerti dai barbieri.

- **Prenotazione Appuntamenti**:

- **NewAppointmentControllerCalendar**: Gestisce la schermata per la selezione della data, consentendo agli utenti di scegliere il giorno desiderato per l’appuntamento.
  - **NewAppointmentControllerSlot**: Si occupa della selezione dello slot orario, del barbiere e del servizio. Mostra un riepilogo dettagliato dell’appuntamento, richiede la conferma dell’utente e permette la scelta del metodo di pagamento.

## Esempio di Gestione degli Eventi

Il frammento di codice seguente mostra come il NewAppointmentControllerSlot gestisce la sequenza di eventi successiva alla selezione dello slot orario:

```
1  public void selectSlotAction(javafx.event.ActionEvent actionEvent) {
2      MFXButton button = (MFXButton) actionEvent.getSource();
3      String timeString = button.getText();
4
5      if (timeString.length() == 4) {
6          timeString = "0" + timeString;
7      }
8
9      LocalTime time = LocalTime.parse(timeString);
10     LocalDate date = LocalDate.parse(dateLabel.getText());
11
12     List<Appointment> userAppointments = newAppointmentSlotsService.getAppointments()
13     ;
14     for (Appointment appointment : userAppointments) {
15         if (newAppointmentSlotsService.isSameDateTime(appointment, date, time)) {
16             AlertHelper.showError("Error", "You already have an appointment on this
date and time");
17             return;
18         }
19     }
20
21     String selectedBarber = barberComboBox.getSelectionModel().getSelectedItem();
22     String selectedService = serviceComboBox.getSelectionModel().getSelectedItem();
23
24     if (selectedService == null || selectedService.isEmpty()) {
25         AlertHelper.showError("Error", "Please select a service before booking an
appointment.");
26         return;
27     }
28
29     String confirmationMessage = "Barber: " + selectedBarber + "\n" +
30         "Service: " + selectedService + "\n" +
31         "Price: " + servicesData.get(selectedService) + "$" + "\n" +
32         "Date: " + date + "\n" +
33         "Time: " + time + "\n";
34
35     boolean confirmed = AlertHelper.showConfirmation("Confirmation Dialog",
36         "Are you sure you want to book this appointment?\n" + confirmationMessage
37     );
38     if (confirmed) {
39         showPaymentOptions(selectedBarber, selectedService, date, time);
40     }
41 }
```

### 3.4 Services

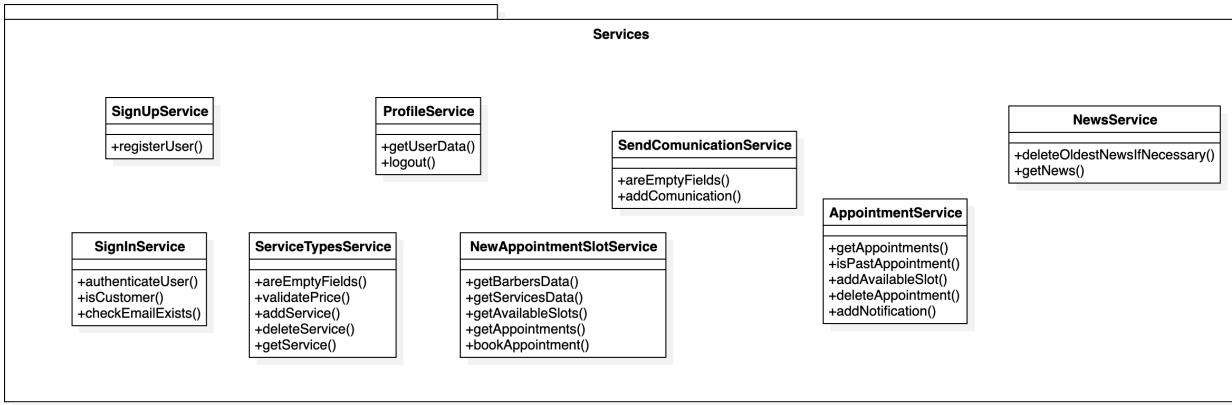


Figura 39: Services

La figura 39 illustra il pacchetto *Services*, che contiene le classi responsabili della logica di business del sistema. Queste classi, denominate servizi, gestiscono le operazioni principali, come la registrazione e l'autenticazione degli utenti, la gestione degli appuntamenti e dei servizi, e l'invio delle comunicazioni.

I servizi sono progettati per essere modulari e riutilizzabili, garantendo una chiara separazione tra la logica di business e l'interfaccia utente. Questa architettura facilita l'integrazione con i controller, semplifica la manutenzione del codice e ne migliora l'estendibilità. Ogni servizio è dedicato a una specifica area funzionale del sistema, assicurando una gestione organizzata e coerente delle operazioni.

La classe **SessionManager**, situata nel pacchetto *Authentication* (figura 40), centralizza la gestione della sessione utente, consentendo ai servizi di accedere facilmente alle informazioni dell'utente autenticato. In ogni servizio è presente una **reference variable** a **SessionManager**, che viene ottenuta tramite il metodo statico `getInstance()` secondo il pattern Singleton. Questo garantisce l'esistenza di un'unica istanza condivisa in tutta l'applicazione. I servizi interagiscono con il database tramite specifici DAO (Data Access Object) definiti nel pacchetto *Persistence*. In ogni servizio, vengono dichiarate le **reference variables** alle interfacce DAO, che vengono inizializzate nel costruttore con le rispettive implementazioni concrete. Questa struttura assicura un accesso ai dati persistenti che sia organizzato, sicuro e conforme ai principi di separazione delle responsabilità.

A differenza dei controller, i servizi, ad eccezione di quelli progettati esclusivamente per un ruolo, sono condivisi tra entrambe le tipologie di controller. Questa scelta architettonica consente di evitare la duplicazione del codice, garantendo una gestione centralizzata delle funzionalità comuni. Le classi dei servizi implementate sono le seguenti:

- **SignInService** e **SignUpService**: Gestiscono rispettivamente l'autenticazione e la registrazione degli utenti.
- **AppointmentService**: Si occupa della gestione degli appuntamenti esistenti.
- **NewAppointmentService**: Gestisce la prenotazione di nuovi appuntamenti.
- **SendCommunicationService**: Implementa la logica per l'invio delle comunicazioni da parte dei barbieri.
- **ServiceTypesService**: Gestisce i servizi offerti dai barbieri, come l'aggiunta e la rimozione.

- **ProfileService**: Recupera le informazioni del profilo utente.
- **NewsService**: Gestisce il recupero e la visualizzazione delle comunicazioni ricevute.

### 3.5 Authentication

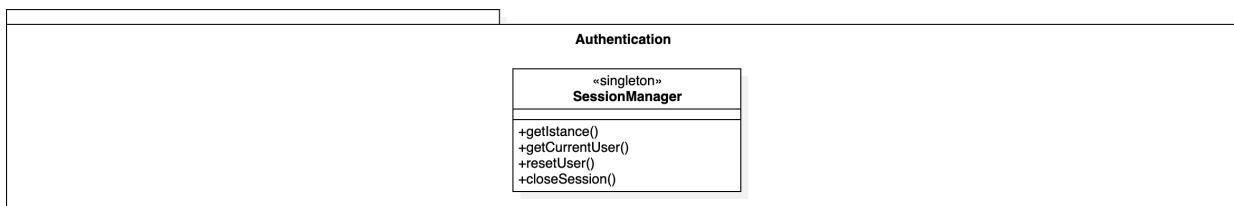


Figura 40: Authentication

Il pacchetto *Authentication*, mostrato in figura 40, include la classe **SessionManager**, che si occupa della gestione della sessione utente nell'applicazione Barbershop. Questa classe mantiene le informazioni relative all'utente attualmente autenticato e implementa il **Singleton Pattern**, garantendo l'esistenza di una sola istanza accessibile globalmente. Il costruttore della classe è privato, mentre l'accesso all'istanza avviene tramite il metodo statico `getInstance()`.

### 3.6 Security

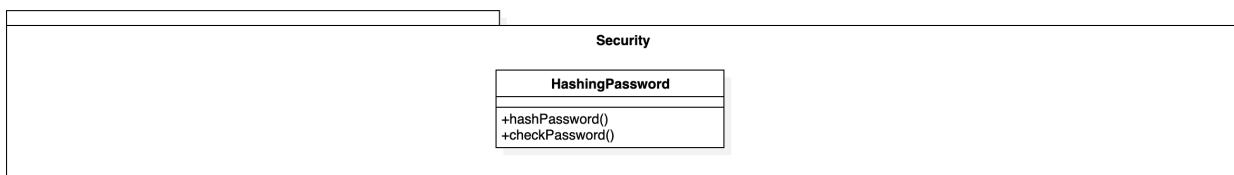


Figura 41: Security

Il pacchetto *Security* in figura 41 include la classe **HashingPassword**, progettata per garantire la protezione delle credenziali utente mediante l'algoritmo **BCrypt**, implementato tramite la libreria `jBCrypt`. Le principali funzionalità offerte comprendono l'hashing delle password, che prevede innanzitutto la verifica che la password in ingresso non sia `null`. Successivamente, viene generato un *salt* univoco e calcolato l'hash mediante `BCrypt`. L'output è una stringa che incorpora sia l'hash sia il *salt*.

Un'altra funzionalità importante è il controllo della password: anche in questo caso si verifica preliminarmente che la password fornita non sia `null`, quindi si estrae il *salt* dall'hash e si ricalcola l'hash in modo sicuro, prevenendo attacchi basati sul *timing*. Il metodo restituisce `true` se le due password corrispondono, `false` altrimenti.

La scelta di implementare questa classe è stata fatta per garantire la sicurezza delle credenziali dei clienti e dei barbieri, proteggendo le informazioni sensibili che altrimenti sarebbero state esposte in chiaro nel database.

### 3.7 Payment

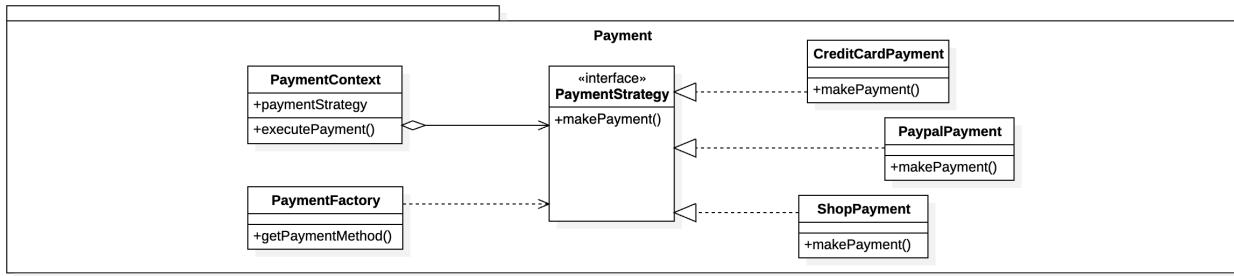


Figura 42: Payment

Il pacchetto `Payment`, illustrato in figura 42, è dedicato alla gestione dei metodi di pagamento e implementa il pattern Strategy per incapsulare le diverse modalità di pagamento (PayPal, carta di credito, pagamento in negozio).

L'interfaccia `PaymentStrategy` definisce il metodo `makePayment`, che viene implementato dalle classi concrete `PaypalPayment`, `CreditCardPayment` e `ShopPayment`. Ciascuna di queste classi è responsabile della logica specifica per il formato dell'importo e il canale di pagamento scelto.

La classe `PaymentFactory` utilizza l'enumerazione `PaymentMethod` del Domain Model per creare istanze delle strategie di pagamento. Il metodo statico `getPaymentMethod` restituisce l'istanza appropriata di `PaymentStrategy`, isolando il client dai dettagli delle classi concrete.

Infine, la classe `PaymentContext` mantiene un riferimento a una strategia di pagamento e delega a essa l'esecuzione del pagamento tramite il metodo `executePayment(amount)`. Questo approccio garantisce che il contesto rimanga indipendente dal tipo di metodo di pagamento utilizzato.

## 3.8 Persistence

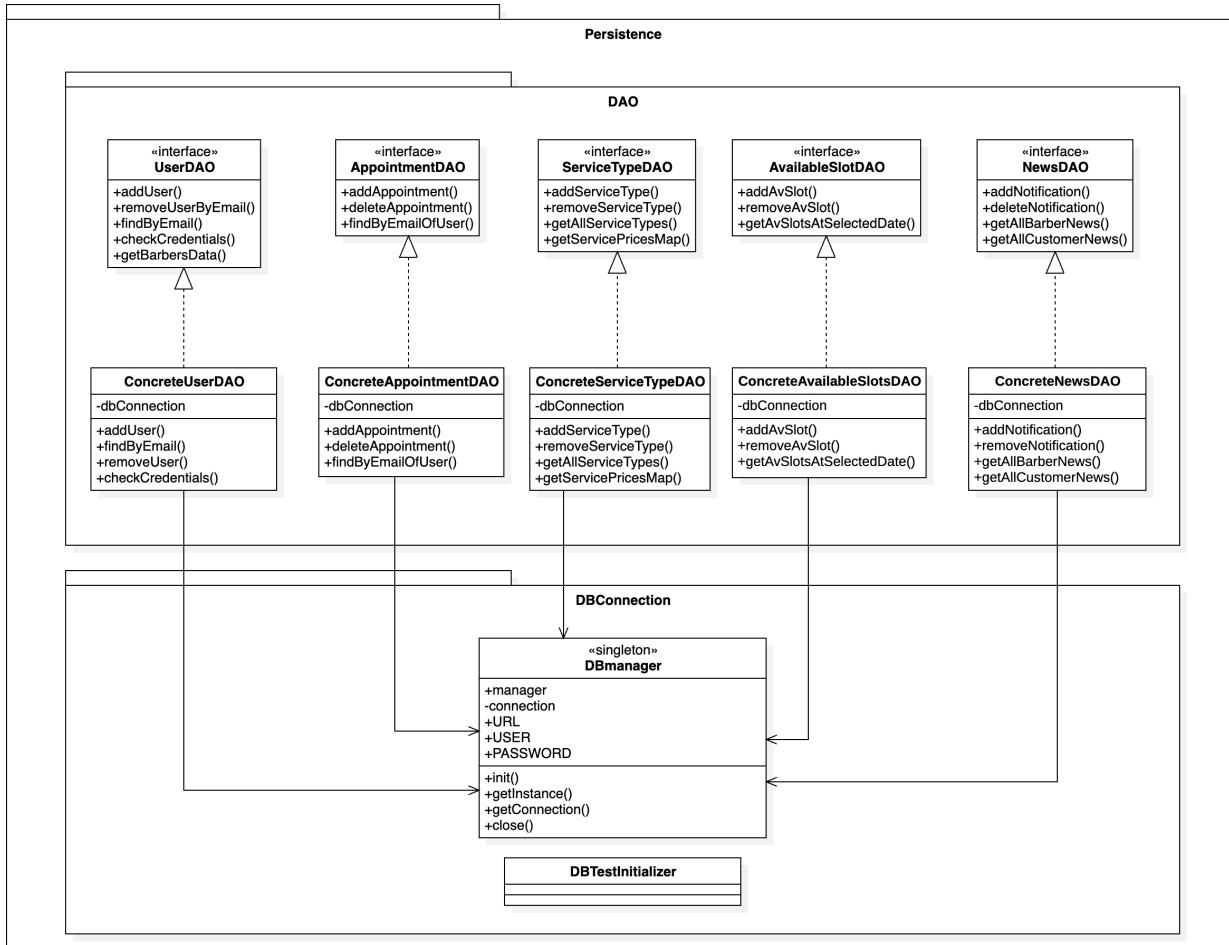


Figura 43: Persistence

Il pacchetto *Persistence* in figura 43 si occupa della gestione della persistenza dei dati, garantendo un accesso strutturato e sicuro alle informazioni archiviate nel database.

### 3.8.1 DBConnection

Il package **DBConnection** contiene la classe **DBManager**, responsabile della gestione della connessione al database. La classe implementa il pattern **Singleton** per garantire che durante l'intero ciclo di vita dell'applicazione esista una sola istanza di **DBManager**. Questo viene realizzato tramite una variabile statica **manager** che contiene l'istanza della classe, un costruttore privato che impedisce la creazione di istanze esterne e il metodo **getInstance** che restituisce l'istanza esistente o la crea se ancora assente. Questo approccio assicura una gestione centralizzata e controllata della connessione al database, evitando istanze multiple e possibili conflitti. Dal codice si nota anche la presenza del parametro **isTest** per distinguere la connessione al database di test. La classe **DBManager** è così implementata:

```

1  public class DBManager {
2      public static DBManager manager = null;
3      private Connection connection = null;
4
5      // Parametri di connessione per la produzione

```

```

6   public static final String PROD_URL = "jdbc:postgresql://localhost:5432/BarberShop_DB";
7   ";
8   public static final String PROD_USER = "SWEuser";
9   public static final String PROD_PASSWORD = "swepass";
10  // Parametri di connessione per il database H2 (usato nei test)
11  public static final String TEST_URL = "jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1";
12  public static final String TEST_USER = "sa";
13  public static final String TEST_PASSWORD = "";
14
15  private DBManager(boolean isTest) {
16      try {
17          // Scegli la connessione in base al flag isTest
18          String url = isTest ? TEST_URL : PROD_URL;
19          String user = isTest ? TEST_USER : PROD_USER;
20          String password = isTest ? TEST_PASSWORD : PROD_PASSWORD;
21
22          this.connection = DriverManager.getConnection(url, user, password);
23          System.out.println("Connection established to " + (isTest ? "H2 Test DB" : "PostgreSQL DB"));
24      } catch (SQLException e) {
25          e.printStackTrace();
26          System.out.println("Connection failed");
27      }
28  }
29
30  private static DBManager init(boolean isTest) {
31      if (manager == null) {
32          manager = new DBManager(isTest);
33      }
34      return manager;
35  }
36
37  public static DBManager getInstance(boolean isTest) {
38      return init(isTest);
39  }
40
41  public Connection getConnection() {
42      return connection;
43  }
44
45  public void close() {
46      try {
47          if (manager != null && manager.connection != null && !manager.connection.
48              isClosed()) {
49              manager.connection.close();
50              System.out.println("Connection closed");
51              manager = null;
52          }
53      } catch (SQLException e) {
54          e.printStackTrace();
55      }
56  }

```

### 3.8.2 DAO

Il package DAO contiene le interfacce e le relative implementazioni concrete. La struttura delle classi adotta il pattern **DAO** (Data Access Object), che isola la logica di accesso ai dati dal resto dell'applicazione. Questo approccio consente di definire in un'interfaccia i metodi di persistenza e di realizzarne una o più implementazioni concrete.

Ad esempio, l'interfaccia `AppointmentDAO` dichiara i metodi per aggiungere, eliminare e ricercare appuntamenti, mentre `ConcreteAppointmentDAO` fornisce l'implementazione concreta utilizzando JDBC per interagire con il database. Ogni operazione sfrutta la connessione fornita da `DBManager`. La separazione tra interfaccia e implementazione rende il codice più modulare, manutenibile e facilmente estendibile.

Segue il codice relativo alla ricerca degli appuntamenti di un barbiere, in cui è stata eseguita una query che unisce le tabelle `Appointments`, `Users` e `Service_Types` attraverso dei join: .

```
1  public List<Appointment> findByEmailOfBarber(String email) {
2      List<Appointment> appointments = new ArrayList<>();
3
4      try {
5          Connection connection = dbManager.getConnection();
6
7          PreparedStatement stmt = connection.prepareStatement(
8              "SELECT a.app_date, a.app_time, a.payment, " +
9                  "c.name AS customer_name, c.surname AS customer_surname, " +
10                 "c.email AS customer_email, c.pass_hash AS customer_pass, c.
11                 phone AS customer_phone, " +
12                 "b.name AS barber_name, b.surname AS barber_surname, " +
13                 "b.email AS barber_email, b.pass_hash AS barber_pass, b.phone
14                 AS barber_phone " +
15
16                 "FROM Appointments a " +
17                 "JOIN Users b ON a.barber_email = b.email " +
18                 "JOIN Users c ON a.customer_email = c.email " +
19                 "WHERE b.email = ? " +
20                 "ORDER BY a.app_date, a.app_time"
21         );
22
23
24         stmt.setString(1, email);
25         ResultSet rs = stmt.executeQuery();
26
27
28         while (rs.next()) {
29             LocalDate appDate = rs.getDate("app_date").toLocalDate();
30             LocalTime appTime = rs.getTime("app_time").toLocalTime();
31             String barberEmail = rs.getString("barber_email");
32
33             Customer customer = new Customer(
34                 rs.getString("customer_name"),
35                 rs.getString("customer_surname"),
36                 rs.getString("customer_email"),
37                 rs.getString("customer_pass"),
38                 rs.getString("customer_phone")
39             );
40
41             Barber barber = new Barber(
42                 rs.getString("barber_name"),
43                 rs.getString("barber_surname"),
44                 rs.getString("barber_email"),
45                 rs.getString("barber_pass"),
46                 rs.getString("barber_phone")
47             );
48
49             Appointment appointment = new Appointment(
50                 appDate,
51                 appTime,
52                 payment,
53                 customer,
54                 barber
55             );
56
57             appointments.add(appointment);
58         }
59     }
60
61     return appointments;
62 }
```

```

38         rs.getString("barber_surname"),
39         rs.getString("barber_email"),
40         rs.getString("barber_pass"),
41         rs.getString("barber_phone")
42     );
43
44     PaymentMethod paymentMethod = PaymentMethod.valueOf(rs.getString("payment
45   "));
46
47     List<ServiceType> serviceTypes = getServicesForAppointment(connection,
48     appDate, appTime, barberEmail);
49
50     Appointment appointment = new Appointment(
51       appDate,
52       appTime,
53       customer,
54       barber,
55       serviceTypes,
56       paymentMethod
57     );
58
59     appointments.add(appointment);
60   }
61   rs.close();
62   stmt.close();
63 } catch (SQLException e) {
64   e.printStackTrace();
65 }
66 return appointments;
}

```

### 3.9 Database

Di seguito vengono mostrati la creazione di una **tabella** del database (Available Slots), la creazione di una **funzione** associata ad essa e il **trigger** che la chiama.

```

1 CREATE TABLE Available_Slots (
2   barber_email VARCHAR(100) NOT NULL,
3   slot_date DATE NOT NULL,
4   start_time TIME(0) NOT NULL,
5   PRIMARY KEY (barber_email, slot_date, start_time),
6   FOREIGN KEY (barber_email) REFERENCES Users(email) ON DELETE CASCADE
7 );

```

Questa tabella memorizza gli slot orari disponibili per ogni barbiere. Ogni record rappresenta una fascia oraria in cui un determinato barbiere è disponibile in una data specifica. La tabella è collegata alla tabella Users attraverso una chiave esterna sul campo barber\_email, assicurandosi che solo utenti registrati con ruolo "BARBER" possano avere fasce orarie assegnate. Grazie alla clausola ON DELETE CASCADE, eventuali eliminazioni di utenti comportano automaticamente la rimozione delle relative disponibilità, assicurando coerenza e pulizia del database.

```

1 CREATE OR REPLACE FUNCTION create_slots_for_new_barber_function()

```

```

2 RETURNS TRIGGER AS $$ 
3 DECLARE
4     start_date DATE;
5     end_date DATE;
6 BEGIN
7
8     IF NEW.role = 'BARBER' THEN
9
10        start_date := CURRENT_DATE;
11
12        end_date := start_date + INTERVAL '1 year';
13
14
15        WHILE start_date < end_date LOOP
16
17            IF NOT EXISTS (
18                SELECT 1
19                FROM Available_Slots
20                WHERE barber_email = NEW.email
21                AND slot_date = start_date
22                AND start_time IN ('08:00:00', '09:00:00', '10:00:00', '11:00:00',
23                '12:00:00', '13:00:00',
24                                '15:00:00', '16:00:00', '17:00:00', '18:00:00',
25                '19:00:00', '20:00:00')
26            ) THEN
27
28                INSERT INTO Available_Slots (barber_email, slot_date, start_time)
29                VALUES (NEW.email, start_date, '08:00:00'),
30                        (NEW.email, start_date, '09:00:00'),
31                        (NEW.email, start_date, '10:00:00'),
32                        (NEW.email, start_date, '11:00:00'),
33                        (NEW.email, start_date, '12:00:00'),
34                        (NEW.email, start_date, '13:00:00');
35
36                INSERT INTO Available_Slots (barber_email, slot_date, start_time)
37                VALUES (NEW.email, start_date, '15:00:00'),
38                        (NEW.email, start_date, '16:00:00'),
39                        (NEW.email, start_date, '17:00:00'),
40                        (NEW.email, start_date, '18:00:00'),
41                        (NEW.email, start_date, '19:00:00'),
42                        (NEW.email, start_date, '20:00:00');
43            END IF;
44
45            start_date := start_date + INTERVAL '1 day';
46        END LOOP;
47    END IF;
48
49    RETURN NEW;
50 END;
51 $$ LANGUAGE plpgsql;

```

Questa funzione trigger viene eseguita automaticamente ogni volta che viene inserito un nuovo utente nella tabella Users. Se il nuovo utente ha il ruolo BARBER, la funzione genera automaticamente le disponibilità orarie (Available\_slots) per il barbiere per un anno a partire dalla data corrente. Per ogni giorno, vengono inseriti dodici slot orari predefiniti (dalle 08:00 alle 13:00 e dalle 15:00 alle 20:00). Prima di inserire gli slot, la funzione verifica che non esistano già per evitare duplicati.

```
1 CREATE TRIGGER create_slots_for_new_barber
2 AFTER INSERT ON Users
3 FOR EACH ROW
4 EXECUTE FUNCTION create_slots_for_new_barber_function();
```

Questo trigger è associato alla tabella Users e si attiva dopo l'inserimento di ogni nuovo record. Il trigger esegue la funzione `create_slots_for_new_barber_function` quando viene aggiunto un nuovo utente, che genera automaticamente le disponibilità orarie per l'utente se il suo ruolo è BARBER.

## 4 Test

### 4.1 Test Coverage

Sono state implementate tre tipologie di test: **Unit Tests**, **Integration Tests** e **Functional Tests**. In totale sono stati creati **163** test, di cui **78** unitari, **71** funzionali e **14** di integrazione. La copertura, calcolata attraverso Jacoco, è del **96%**, ed è così distribuita:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxtx	Missed Lines	Missed Methods	Missed Classes
Persistence.DAO		92%		71%	17	58	53	370
PageControllers		97%		77%	38	206	24	682
Helpers		94%		72%	7	32	7	79
Persistence.DBConnection		89%		81%	4	21	6	39
Payment		88%		87%	2	15	2	20
Services		99%		88%	7	71	2	165
Security		88%		100%	1	5	1	8
Model		100%		n/a	0	36	0	80
Authentication		100%		100%	0	8	0	14
Total	228 of 5,814	96%	66 of 316	79%	76	452	95	1,457
							12	293
								0
								49

Figura 44: Copertura del codice

Le sezioni seguenti elencano le **classi** di test implementate per ciascuna tipologia, evidenziando i **metodi** più significativi. Ogni metodo è associato a uno **use case template**, con il riferimento specificato accanto al nome della classe a cui appartiene. Per alcuni metodi, è incluso un frammento di codice che ne illustra il funzionamento.

### 4.2 Unit Tests

I test unitari sono stati implementati per verificare il comportamento corretto e isolato delle classi di servizio. È stato utilizzato il framework JUnit per la scrittura e l'esecuzione dei test. Per simulare il comportamento dei DAO e delle classi di supporto, come **SessionManager**, è stata utilizzata la libreria Mockito. I mock sono stati creati e iniettati nei costruttori dei servizi tramite il meccanismo di Dependency Injection, consentendo di testare le funzionalità dei servizi in modo indipendente dai componenti esterni.

#### 1. SignUpServiceTest [UC-01]:

- UT-01-05: registerCustomerSuccess
- UT-01-05.1: registerFailEmptyFields

#### 2. SignInServiceTest [UC-02]:

```
1      @BeforeEach
2      void setUp() {
3          userDAO = mock(UserDAO.class);
4          sessionManager = mock(SessionManager.class);
5          signInService = new SignInService(userDAO, sessionManager);
6      }
7
```

- UT-02-05: authenticateValidCredentials

```
1      @Test
2      void authenticateValidCredentials() {
3          Customer customer = new Customer("Mario", "Rossi", EMAIL, PASSWORD,
4          "1234567890");
```

```

4
5     when(userDAO.checkCredentials(EMAIL, PASSWORD)).thenReturn(true);
6     when(userDAO.findByEmail(EMAIL)).thenReturn(customer);
7
8     assertTrue(signInService.authenticateUser(EMAIL, PASSWORD));
9     verify(userDAO, times(1)).checkCredentials(EMAIL, PASSWORD);
10    verify(userDAO, times(1)).findByEmail(EMAIL);
11    verify(sessionManager).setCurrentUser(customer);
12 }
13

```

- **UT-02-05.1:** authenticateInvalidCredentials

```

1 @Test
2 void authenticateInvalidCredentials() {
3     when(userDAO.checkCredentials(EMAIL, PASSWORD)).thenReturn(false);
4
5     assertFalse(signInService.authenticateUser(EMAIL, PASSWORD));
6     verify(userDAO, times(1)).checkCredentials(EMAIL, PASSWORD);
7     verify(userDAO, never()).findByEmail(anyString());
8     verify(sessionManager, never()).setCurrentUser(any());
9 }
10

```

3. **AppointmentServiceTest** [UC-03, UC-05]:

- **UT-03-02:** getAppointmentsBarber
- **UT-05-05:** deleteAppointmentSuccess

4. **NewAppointmentSlotsServiceTest** [UC-04]:

- **UT-04-06:** getAvailableSlots

5. **NewsServiceTest** [UC-06]:

- **UT-06-02:** getNewsCustomer

6. **ProfileServiceTest** [UC-07]:

- **UT-07-02:** getUserDataWithLoggedUser

7. **ServiceTypesServiceTest** [UC-09, UC-10]:

- **UT-09-02.1:** emptyFieldsShouldBeTrue
- **UT-09-03:** addService
- **UT-10-05:** deleteService

8. **SendCommunicationServiceTest** [UC-11]:

- **UT-11-02.1:** emptyFieldsShouldBeTrue
- **UT-11-04:** addCommunicationSuccess

9. **SessionManagerTest**

10. **DBManagerTest**

## 11. DBTestInitializerTest

## 12. PaymentTest

## 13. HashingPasswordTest

### 4.3 Integration Tests

I test di integrazione sono stati sviluppati per verificare l'interazione tra i servizi e i DAO. Per la loro implementazione è stato utilizzato il framework JUnit. Inoltre, è stato scelto di utilizzare il database reale PostgreSQL per verificare che le operazioni di persistenza funzionino correttamente.

#### 1. UserTest [UC-01, UC-02]:

- **IT-01:** testSignUpBarber
- **IT-02:** testSignInBarber

#### 2. AppointmentAndSlotsTest [UC-04, UC-05]:

- **IT-04:** testBookAppointmentAndSlotRemoval

```
1  @Test
2  public void testBookAppointmentAndSlotRemoval() {
3      List<AvailableSlot> availableSlots = newAppointmentSlotsService.
4          getAvailableSlots(barber.getEmail(), testDate);
5      boolean slotExists = availableSlots.stream().anyMatch(s -> s.
6          getStartTime().equals(testTime));
7      assertTrue(slotExists);
8
9
10     boolean booked = newAppointmentSlotsService.bookAppointment(
11         barber.getName() + " " + barber.getSurname(),
12         servicesName,
13         testDate,
14         testTime,
15         PaymentMethod.CREDIT_CARD
16     );
17     assertTrue(booked);
18
19
20     List<AvailableSlot> slotsAfter = newAppointmentSlotsService.
21         getAvailableSlots(barber.getEmail(), testDate);
22     boolean slotRemoved = slotsAfter.stream().anyMatch(s -> s.getStartTime()
23         .equals(testTime));
24     assertFalse(slotRemoved);
25
26
27     List<Appointment> appointments = appointmentService.getAppointments();
28     boolean appointmentExists = appointments.stream()
29         .anyMatch(a -> a.getDate().equals(testDate) && a.getTime().
30             equals(testTime));
31     assertTrue(appointmentExists);
32
33
34     List<Notification> barberNotifications = newsDAO.getAllBarberNews(
35         barber.getEmail());
36     boolean notificationExists = barberNotifications.stream()
37         .anyMatch(n -> n.getTitle().equals("New Appointment"))
```

```

28             && n.getMessage().contains("Luigi Verdi has booked an
29             appointment with you"));
30         assertTrue(notificationExists);
31     }

```

- **IT-05:** testDeleteAppointmentAndSlotReintroducing

```

1  @Test
2  public void testDeleteAppointmentAndSlotReintroducing() {
3      List<Appointment> appointments = appointmentService.getAppointments();
4      Appointment toDelete = appointments.stream()
5          .filter(a -> a.getDate().equals(testDate) && a.getTime().equals
6          (testTime))
7          .findFirst()
8          .orElse(null);
9
10     assertNotNull(toDelete);
11
12     boolean deleted = appointmentService.deleteAppointment(toDelete);
13     assertTrue(deleted);
14
15     AppointmentService.addNotification(toDelete);
16
17     AppointmentService.addAvailableSlot(toDelete);
18
19     List<AvailableSlot> slots = newAppointmentSlotsService.
20     getAvailableSlots(barber.getEmail(), testDate);
21     boolean slotExists = slots.stream().anyMatch(s -> s.getStartTime().
22     equals(testTime));
23     assertTrue(slotExists);
24
25     List<Notification> barberNotifications = newsDAO.getAllBarberNews(
26     barber.getEmail());
27     boolean notificationExists = barberNotifications.stream()
28         .anyMatch(n -> n.getTitle().equals("Slot available")
29             && n.getMessage().contains("A slot has become available
30             on " + testDate + " at " + testTime + " with " + barber.getName()));
31     assertTrue(notificationExists);
32 }

```

### 3. ProfileTest [UC-07]:

- **IT-07:** testGetUserData

### 4. ServiceTypesTest [UC-08, UC-09, UC-10]:

- **IT-08-02:** testGetServices
- **IT-09:** testBarberCreateService
- **IT-10:** testBarberDeleteService

### 5. CommunicationTest [UC-11]:

- **IT-11:** testAddMultipleAndDeleteComunications

## 4.4 Functional Tests

I test funzionali sono stati sviluppati per verificare che i controller rispettino i requisiti funzionali definiti. Per la loro implementazione è stato utilizzato il framework TestFX, che consente di simulare l’interazione dell’utente con l’interfaccia grafica e JUnit per la scrittura e l’esecuzione dei test. Per garantire un ambiente di test isolato e rapido, è stato utilizzato il database H2 in memoria.

### 1. SignUpControllerTest [UC-01]:

- **FT-01:** testSuccessfulSignupNavigatesToSignin
- **FT-01-05.1:** testInvalidSecretCode

### 2. SignInControllerTest [UC-02]:

- **FT-02:** testSuccessfulSigninAsBarber
- **FT-02-05.1:** testSigninFailureWrongCredentials

### 3. AppointmentCustomerControllerTest [UC-05]:

- **FT-05:** testDeleteAppointmentSuccess
- **FT-05-04.1:** testDeleteAppointmentNotConfirmed

### 4. AppointmentBarberControllerTest

### 5. NewAppointmentControllerCalendarTest [UC-4]:

- **FT-04-02:** testClickOnDayCellNavigatesToSlotsView

### 6. NewAppointmentControllerSlotsTest [UC-04]:

- **FT-04:** testBookAppointmentWithOneServiceSuccess

```
1  @Test
2  public void testBookAppointmentWithOneServiceSuccess() throws Exception {
3      try (Statement stmt = DBManager.getInstance(true).getConnection().
4          createStatement();
5          ResultSet rs = stmt.executeQuery("SELECT COUNT(*) AS total FROM
6          Appointments WHERE customer_email = 'm.rossi@example.com')) {
7
8          rs.next();
9          int total = rs.getInt("total");
10         assertEquals(1, total);
11     }
12
13     WaitForAsyncUtils.waitForFxEvents();
14     Thread.sleep(500);
15
16     Node caretB = lookup("#barberComboBox").lookup(".caret").query();
17     clickOn(caretB);
18     WaitForAsyncUtils.waitForFxEvents();
19     Thread.sleep(500);
20     clickOn("Luca Verdi");
21
22     Node caretS = lookup("#serviceComboBox").lookup(".caret").query();
```

```

21     clickOn(caretS);
22     WaitForAsyncUtils.waitForFxEvents();
23     Thread.sleep(500);
24     clickOn("Taglio Capelli");
25
26     Thread.sleep(500);
27     clickOn("No");
28
29     Button tenAMButton = lookup("#tenAMButton").queryAs(Button.class);
30     tenAMButton.setDisable(false);
31     tenAMButton.setOpacity(1);
32     WaitForAsyncUtils.waitForFxEvents();
33     Thread.sleep(500);
34     moveTo("#tenAMButton").clickOn("#tenAMButton");
35
36     clickOn("Yes");
37     clickOn("PayPal");
38     clickOn("OK");
39
40     WaitForAsyncUtils.waitForFxEvents();
41     Thread.sleep(500);
42
43     try (Statement stmt = DBManager.getInstance(true).getConnection().
44         createStatement()) {
45
46         ResultSet rs = stmt.executeQuery("SELECT COUNT(*) AS total FROM
47             Available_Slots WHERE barber_email = 'l.verdi@example.com'");
48
49         rs.next();
50         int total = rs.getInt("total");
51         assertEquals(0, total);
52     }
53     try (Statement stmt = DBManager.getInstance(true).getConnection().
54         createStatement()) {
55
56         ResultSet rs = stmt.executeQuery("SELECT COUNT(*) AS total FROM
57             Appointments WHERE customer_email = 'm.rossi@example.com'");
58
59         rs.next();
60         int total = rs.getInt("total");
61         assertEquals(2, total);
62     }
63     try (Statement stmt = DBManager.getInstance(true).getConnection().
64         createStatement()) {
65
66         ResultSet rs = stmt.executeQuery("SELECT COUNT(*) AS total FROM
67             News ");
68
69         rs.next();
70         int total = rs.getInt("total");
71         assertEquals(2, total);
72     }
73 }
```

- FT-04-04.1: testBookAppointmentFailsWithoutServiceSelection

7. **ServiceTypesControllerTest** [UC-09, UC-10]:

- **FT-09:** testAddServiceSuccess
- **FT-09-02.1:** testAddServiceEmptyFields
- **FT-10:** testDeleteAnyService
- **FT-10-04.1:** testDeleteServiceNotConfirmed

8. **SendCommunicationControllerTest** [UC-11]:

- **FT-11:** testSendCommunicationSuccess
- **FT-11-02.1:** testSendCommunicationEmptyFields

9. **NewsCustomerControllerTest**

10. **NewsBarberControllerTest**

11. **ProfileCustomerControllerTest**

12. **ProfileBarberControllerTest**

## 5 Utilizzo di AI

### 5.1 Strumenti AI

Per supportare la progettazione e lo sviluppo dell'applicazione, sono stati utilizzati diversi strumenti basati su modelli linguistici avanzati, tra cui i modelli di grandi dimensioni (**LLMs**) ChatGPT-4o e Claude 3.7 Sonnet, oltre a GitHub Copilot, un assistente alla programmazione integrato in IntelliJ e alimentato da un LLM.

Questi strumenti si sono rivelati utili per la risoluzione di errori nel codice e la generazione di porzioni di codice ripetitive, come la creazione dinamica dei componenti dell'interfaccia utente e delle query SQL.

In particolare, **ChatGPT-4o** ha offerto un valido supporto nella scrittura di query SQL complesse, **Claude 3.7 Sonnet** si è dimostrato efficace nell'interpretazione di diagrammi UML e nella generazione di codice Java a partire da essi, mentre **GitHub Copilot** ha agevolato la scrittura del codice fornendo suggerimenti in tempo reale e correggendo errori.

È importante sottolineare che, nonostante l'utilizzo di questi strumenti, è stata comunque necessaria una revisione manuale accurata. Il codice generato ha richiesto modifiche e perfezionamenti, oltre a più iterazioni tramite prompt per ottenere i risultati desiderati.

### 5.2 Esempio di interazione

Un esempio significativo è rappresentato dalla query per la ricerca degli appuntamenti associati a un barbiere, mostrata in precedenza (`findByEmailOfBarber`). Questa query è stata generata da ChatGPT e ha richiesto diversi prompt per ottenere un risultato ottimale:

**Prompt** Sto scrivendo il codice per un progetto JAVA che consiste in un'app per la gestione di appuntamenti in un negozio di barbieri. Sto utilizzando il pattern DAO, ogni classe DAO effettua la connessione al database tramite la classe DBManager (Singleton). Genera il metodo `findByEmailOfBarber` per la classe `ConcreteAppointmentDAO` che implementa una query SQL la quale fa un filtraggio con l'email del barber e genera una tabella dove ogni tupla è un appuntamento del barber, ci sono le informazioni riguardo il ServiceType scelto e le informazioni del customer coinvolto. Il metodo in questione deve ritornare la lista degli appuntamenti. Di seguito ti lascio le classi (presenti nel domain model) coinvolte e il codice SQL di creazione delle tabelle che costituiscono il database:

\codice classi e tabelle...

**ChatGPT** Ecco un'implementazione di `findByEmailOfBarber` in `ConcreteAppointmentDAO`. La logica è:

- Eseguire una query che recupera tutte le tuple di appuntamento e i relativi servizi in un'unica join.
- Iterare sul ResultSet, raggruppando per data/ora per costruire la lista di ServiceType per ciascun Appointment.
- Restituire la lista di Appointment.

\codice generato...

**Prompt** La query deve essere interna al metodo per essere coerente con le precedenti implementazioni dei DAO. Non creare una variabile stringa ma mettila direttamente come input del metodo `prepareStatement`.

**ChatGPT** Ecco la versione aggiornata di `findByEmailOfBarber` con la query inserita direttamente dentro la chiamata a `prepareStatement`:

\codice generato...

Cosa è cambiato:

- La stringa SQL è passata direttamente a `conn.prepareStatement(...)`, senza usare costanti esterne.
- Tutto il resto della logica di raggruppamento e popolamento degli oggetti rimane invariata.

In questo modo hai la query “interna” al metodo, esattamente nel punto in cui viene preparata

**Prompt** Preferirei che utilizzassi come struttura dati una `ArrayList` invece che una `LinkedHashMap`.

**ChatGPT** Ecco la versione di `findByEmailOfBarber` che utilizza una semplice `List<Appointment>` al posto della `LinkedHashMap`. In questo caso, per ogni riga del `ResultSet` cerchiamo in lista un appuntamento con la stessa data e ora; se non lo troviamo lo creiamo, altrimenti ne recuperiamo il riferimento e aggiungiamo il `ServiceType`.

\codice generato...

Punti chiave:

- Usiamo una `List<Appointment>` e, per ogni riga del `ResultSet`, cerchiamo in lista un appuntamento con la stessa data e ora.
- Se non lo troviamo, lo creiamo (e inizializziamo la lista di `ServiceType`) e lo aggiungiamo alla lista; altrimenti riutilizziamo quello esistente.
- Alla fine restituiamo direttamente la `List<Appointment>`.

In questo modo non abbiamo più bisogno di una mappa intermedia, e manteniamo comunque l’ordine cronologico dato dall’`ORDER BY` della query.

**Prompt** Nota bene che gli oggetti appuntamento che crei nel metodo devono ricevere nel costruttore la lista di ServiceTypes che caratterizzano il singolo appuntamento, tali servizi devono essere prelevati dal database attraverso un metodo esterno già esistente: getServicesForAppointment. Inoltre la tabella che crea la query non c'è bisogno che includa, tra i campi delle tuple, le informazioni riguardo il ServiceType dato che un singolo appuntamento può avere più servizi.

**ChatGPT** Perfetto! Allora aggiorniamo il metodo findByEmailOfBarber seguendo scrupolosamente i tuoi vincoli:

Vincoli richiesti:

- Niente LinkedHashMap: usiamo una List<Appointment>.
- Nessuna JOIN sui servizi nella query: la query estraе solo dati da Appointments, Users (customer e barber).
- Recupero dei servizi: avviene esternamente, tramite il metodo già esistente getServicesForAppointment(Connection, LocalDate, LocalTime, String), dopo la lettura delle info principali dell'appuntamento.

\codice generato...

Sono state necessarie **quattro iterazioni** per ottenere il risultato finale. Durante il processo, sono stati forniti vincoli specifici e richieste dettagliate per affinare il metodo. Complessivamente, ChatGPT si è dimostrato flessibile, adattandosi rapidamente e apportando le modifiche richieste in modo efficace. Nonostante la complessità della query SQL, ChatGPT ha compreso immediatamente il contesto, ricordando nei messaggi successivi l'implementazione delle classi, i parametri dei costruttori e la struttura delle tabelle del database, incluse le chiavi primarie.

### 5.3 Conclusioni

L'integrazione di strumenti basati su AI ha notevolmente accelerato lo sviluppo del progetto, consentendo di generare porzioni di codice in modo rapido ed efficace. Tuttavia, abbiamo appreso che, nonostante l'utilità di questi strumenti, è fondamentale mantenere un **approccio critico** e procedere con una revisione manuale del codice generato, specialmente in progetti di questa grandezza. Ottenere il risultato desiderato spesso richiede **più iterazioni** e richieste specifiche, guidando il modello passo dopo passo. Gli strumenti AI si sono rivelati preziosi alleati, ma devono essere utilizzati come supporto, con una **supervisione attenta** in ogni fase del processo di implementazione.