

**Università Politecnica delle Marche**

Corso di Laurea Triennale in Ingegneria Informatica e dell’Automazione

Progetto per l’esame di Ingegneria del Software

A black and white logo

AI-generated content may be incorrect.

**Un software gestionale per Barber Shop**

**Relatori: Autori:**

Prof. Ursino Domenico Mohamed Jouini

Prof. Traini Davide Gianmaria Sammartino

Ilario Polidori

Anno accademico 2024/2025

Indice

[1 Introduzione 3](#_Toc208588403)

[1.1 Descrizione del progetto 3](#_Toc208588404)

[1.1.1 Panoramica del sistema 3](#_Toc208588405)

[1.1.2 Registrazione cliente 3](#_Toc208588406)

[1.1.3 Prenotazione servizi 3](#_Toc208588407)

[1.1.4 Gestione del personale 3](#_Toc208588408)

[1.1.5 Area amministrativa 4](#_Toc208588409)

[1.2 Glossario dei termini 4](#_Toc208588410)

[2 Analisi dei requisiti 5](#_Toc208588411)

[2.1 Requisiti del sistema 5](#_Toc208588412)

[2.1.1 Requisiti funzionali 6](#_Toc208588413)

[2.1.2 Requisiti non Funzionali 9](#_Toc208588414)

[2.2 Diagrammi dei casi d’uso 11](#_Toc208588415)

[2.2.1 Attori 11](#_Toc208588416)

[2.2.2 Gestione Amministratore 12](#_Toc208588417)

[2.2.3 Gestione Account 21](#_Toc208588418)

[2.2.4 Gestione Cliente 28](#_Toc208588419)

[2.2.5 Gestione Barbiere 38](#_Toc208588420)

[2.2.6 Prenotazione Appuntamento 45](#_Toc208588421)

[2.3 Matrice di mapping 53](#_Toc208588422)

[3 Diagrammi di analisi 54](#_Toc208588423)

[3.1 Diagrammi delle classi di analisi 54](#_Toc208588424)

[3.1.1 Package di analisi 54](#_Toc208588425)

[3.1.2 Package di analisi: Utente 55](#_Toc208588426)

[3.1.3 Package di analisi: Prenotazione 56](#_Toc208588427)

[3.2 Diagrammi di sequenza 57](#_Toc208588428)

[3.2.1 Registra Account 57](#_Toc208588429)

[3.2.2 Login Utente 58](#_Toc208588430)

[3.2.3 Prenota appuntamento 59](#_Toc208588431)

[3.2.4 Pubblica Recensione 60](#_Toc208588432)

[3.2.5 Genera report statistiche 61](#_Toc208588433)

[3.3 Diagrammi di attività 62](#_Toc208588434)

[3.3.1 Registra Account 62](#_Toc208588435)

[3.3.2 Login Account 63](#_Toc208588436)

[3.3.3 Prenota Appuntamento 64](#_Toc208588437)

[3.3.4 Pubblica Recensione 65](#_Toc208588438)

[3.3.5 Genera Report Statistiche 66](#_Toc208588439)

[4 Diagrammi di progettazione 67](#_Toc208588440)

[4.1 Diagrammi delle classi di progettazione 67](#_Toc208588441)

[4.1.1 Package di progettazione: User 68](#_Toc208588442)

[4.1.2 Package di progettazione: Appointment 69](#_Toc208588443)

[4.2 Diagramma dei componenti 70](#_Toc208588444)

[4.3 Diagrammi delle macchine a stati 71](#_Toc208588445)

[5 Implementazione 72](#_Toc208588446)

[5.1 Diagrammi di deployment 72](#_Toc208588447)

[5.2 Tecnologie utilizzate 73](#_Toc208588448)

[5.3 Mockup 74](#_Toc208588449)

[5.4 Descrizione unit tests 75](#_Toc208588450)

[5.4.1 Test profilo amministratore 75](#_Toc208588451)

[5.4.2 Test profilo barbiere 80](#_Toc208588452)

# Introduzione

Il progetto **“Barber Manager”** nasce con l’obiettivo di realizzare un sistema gestionale moderno e intuitivo, pensato per digitalizzare e semplificare le operazioni quotidiane all’interno di un barber shop. Il sistema è progettato per migliorare l’efficienza organizzativa e l’esperienza complessiva di clienti e operatori del settore.

## Descrizione del progetto

Il sistema è strutturato come una piattaforma web accessibile sia da dispositivi desktop che mobili, composta da un'interfaccia utente semplice e reattiva e da un backend robusto per la gestione dei dati. Ogni funzionalità è pensata per ridurre i tempi di gestione manuale, aumentare la trasparenza nelle prenotazioni e migliorare il rapporto con la clientela.

### Panoramica del sistema

L’obiettivo del sistema è digitalizzare le attività principali del salone, quali la gestione delle prenotazioni, dei clienti, dei servizi offerti e del personale impiegato, migliorando al contempo l’esperienza sia per i clienti che per i professionisti del settore.

Il salone è composto da un’area di accoglienza e trattamento, dove operano uno o più barbieri, ciascuno con turni lavorativi definiti.

### Registrazione cliente

l cliente, accedendo all’applicazione, ha la possibilità di registrarsi inserendo i propri dati personali, uno username unico, un indirizzo e-mail valido e una password.

Il cliente registrato potrà accedere a funzionalità avanzate, quali la gestione autonoma delle proprie prenotazioni, la ricezione di promemoria automatici tramite e-mail e la consultazione dello storico degli appuntamenti effettuati. Inoltre, potrà lasciare una review al barbiere associato all’appuntamento, contribuendo così al miglioramento continuo del servizio, e aggiornare in qualsiasi momento i dati del proprio profilo.

### Prenotazione servizi

La pagina di prenotazione consente di scegliere il barbiere preferito, di selezionare i servizi offerti dal barbiere selezionato e indicare data e orario tra i suoi disponibili.

Il sistema, aggiornato in tempo reale, fornisce una panoramica dettagliata dei barbieri e dei loro turni lavorativi. Una volta confermata, la prenotazione viene salvata automaticamente e resa visibile all’interno del profilo personale del cliente.

### Gestione del personale

Ogni barbiere ha a disposizione un’area riservata personale, dalla quale può consultare in tempo reale gli appuntamenti assegnati, al fine di garantire un servizio sempre più personalizzato e di qualità.

### Area amministrativa

L’admin, dotato di credenziali specifiche, potrà accedere a un pannello di controllo completo, dal quale è possibile aggiungere nuovi barbieri o rimuoverli, gestire i turni del personale, consultare l’agenda delle prenotazioni e visualizzare statistiche dettagliate sull’andamento delle attività.

## Glossario dei termini

Di seguito è riportato il glossario relativo al progetto in esame. I termini elencati nella colonna **“Termine”** sono stati selezionati ed estratti dalla sezione **“1 Introduzione”**. Ciascun termine è stato definito con l’obiettivo di ridurre al minimo le ambiguità interpretative, corredandolo dei relativi sinonimi e della specifica tipologia di appartenenza.

|  |  |  |  |
| --- | --- | --- | --- |
| Termine | Descrizione | Tipo | Sinonimi |
| Barber Manager | Nome del sistema gestionale progettato per un barber shop. | Business | Applicazione, App |
| Barber Shop | Salone da barbiere dove vengono erogati servizi legati a capelli e barba. | Business | Salone, Negozio |
| Cliente | Utente finale che usufruisce dei servizi offerti dal barber shop. | Tecnico | Utente |
| Registrazione | Procedura di inserimento dei dati personali per l’accesso all’app. | Tecnico | Iscrizione |
| Servizio | Trattamento offerto dal salone, come taglio capelli, rasatura, ecc. | Business | Trattamento |
| Staff | Personale autorizzato del salone (barbieri e amministratori). | Business | Personale, Operatori |
| Admin | Utente amministratore che gestisce l’intero sistema. | Tecnico | Amministratore |
| Backend | Componente del sistema che gestisce la logica applicativa e l’accesso ai dati. | Tecnico | Server, API |
| Review | Valutazione e commento lasciato da un cliente su un servizio ricevuto. | Business | Recensione |
| Statistiche | Report analitici sull’attività del salone (incassi, trattamenti, clienti). | Business | Analisi, report |
| Promemoria | Notifica automatica inviata per ricordare appuntamenti imminenti. | Tecnico | Avviso |

# Analisi dei requisiti

A seguito di un’approfondita analisi delle esigenze espresse, sono stati individuati e definiti con chiarezza una serie di requisiti fondamentali per la progettazione e lo sviluppo del sistema. Essi sono suddivisi in **requisiti funzionali** e **requisiti non funzionali**.

* **I requisiti funzionali**: si presentano come elenchi di funzionalità o servizi che il sistema deve fornire.
* **I requisiti non funzionali**: rappresentano i vincoli, le proprietà e le caratteristiche relative al sistema.

## Requisiti del sistema

In questa sezione vengono analizzati i requisiti funzionali del sistema. Come illustrato nella **Figura 1**, tali requisiti sono stati organizzati in **package distinti**, ciascuno dei quali rappresenta una specifica area funzionale del sistema.



Figura 1: Package dei requisiti

### Requisiti funzionali

Per garantire una rappresentazione fedele dei flussi operativi reali, tali requisiti sono stati organizzati in cinque aree tematiche principali:

* **Gestione Amministratore**: include le funzionalità riservate al ruolo amministrativo, relative al controllo degli utenti staff, alla gestione delle disponibilità e alla consultazione delle statistiche.
* **Gestione Account**: raccoglie le operazioni che ogni utente può svolgere in autonomia, come la registrazione, l’accesso, la modifica e la cancellazione del proprio account.
* **Gestione Cliente**: comprende le funzionalità dedicate alla consultazione di dati personali, storico appuntamenti e recensioni da parte dei clienti.
* **Gestione Barbiere**: raggruppa le operazioni relative all’inserimento, modifica e cancellazione dei servizi offerti dai vari barbieri del salone.
* **Prenotazione Appuntamento**: include l’intero processo di prenotazione (selezione barbiere, scelta servizi e orari di disponibilità).



Figura 2: Diagramma dei requisiti funzionali

**Gestione Amministratore**

* **RF01 – Aggiungi Barbiere:**

Il sistema dovrà permettere all’amministratore di aggiungere un utente con il ruolo barbiere.

* **RF02 – Rimuovi Barbiere:**

Il sistema dovrà permettere all’amministratore di rimuovere un utente con il ruolo barbiere.

* **RF03 – Visualizza Barbiere:**

Il sistema dovrà permettere all’amministratore di visualizzare gli account dei barbieri.

* **RF04 – Visualizza Disponibilità barbieri:**

Il sistema dovrà permettere all’amministratore di visualizzare le disponibilità dei barbieri.

* **RF05 – Gestisci Disponibilità barbieri:**

Il sistema dovrà permettere all’amministratore di gestire le disponibilità dei barberi.

* **RF06 – Genera Report Statistiche:**

Il sistema dovrà permettere all’amministratore di generare un report di statistiche dell’andamento dell’attività.

**Gestione Account**

* **RF07 – Login Account:**

Il sistema dovrà permettere all’utente di accedere al proprio account.

* **RF08 – Registra Account:**

Il sistema dovrà permettere all’utente di registrare un nuovo account.

* **RF09 – Modifica Account:**

Il sistema dovrà permettere all’utente di modificare i dati relativi al proprio account.

* **RF10 – Elimina Account:**

Il sistema dovrà permettere all’utente di eliminare il proprio account.

* **RF11 – Visualizza Account:**

Il sistema dovrà permettere al cliente di consultare i propri dati anagrafici e le preferenze.

* **RF12 – Recupera Credenziali:**

Il sistema dovrà permettere all’utente di recuperare le proprie credenziali in caso di smarrimento.

**Gestione Cliente**

* **RF13 – Visualizza Storico Appuntamenti:**

Il sistema dovrà permettere al cliente di consultare il proprio storico degli appuntamenti passati.

* **RF14 – Visualizza Appuntamento:**

Il sistema dovrà permettere al cliente di consultare l’appuntamento prenotato.

* **RF15– Cancella Appuntamento:**

Il sistema dovrà permettere al cliente di cancellare un appuntamento prenotato.

* **RF16 – Pubblica Recensione:**

Il sistema dovrà permettere al cliente di pubblicare una recensione per i servizi ricevuti.

* **RF17 – Modifica Recensione:**

Il sistema dovrà permettere al cliente di modificare una recensione precedentemente pubblicata.

* **RF18 – Elimina Recensione:**

Il sistema dovrà permettere al cliente di eliminare una recensione pubblicata.

* **RF19 – Visualizza Recensione:**

Il sistema dovrà permettere al cliente di consultare una recensione precedentemente pubblicata.

* **RF20 – Visualizza Recensioni:**

Il sistema dovrà permettere al cliente di visualizzare tutte le recensioni pubblicate nel sistema.

* **RF21 – Notifica Promemoria Appuntamento:**

Il sistema dovrà inviare una notifica automatica al cliente come promemoria per l’appuntamento.

**Gestione Barbiere**

* **RF22 – Aggiungi Servizi Barbiere:**

Il sistema dovrà permettere al barbiere di aggiungere un nuovi servizi al proprio account.

* **RF23 – Modifica Servizio Barbiere:**

Il sistema dovrà permettere al barbiere di modificare i dettagli di un proprio servizio esistente.

* **RF24 – Elimina Servizio Barbiere:**

Il sistema dovrà permettere al barbiere di rimuovere un servizio dal proprio catalogo.

* **RF25 – Visualizza Servizi Barbiere:**

Il sistema dovrà permettere al barbiere di consultare il proprio catalogo dei servizi offerti.

**Prenotazione Appuntamento**

* **RF26– Prenota Appuntamento:**

Il sistema dovrà permettere al cliente di prenotare un appuntamento.

* **RF27 – Visualizza Elenco Barbiere:**

Il sistema dovrà permettere al cliente di visualizzare una lista dei barbieri presenti nel sistema.

* **RF28 – Visualizza Disponibilità Barbiere:**

Il sistema dovrà permettere al cliente di visualizzare gli orari di disponibilità del barbiere selezionato.

* **RF29 – Seleziona Barbiere:**

Il sistema dovrà permettere al cliente di selezionare il barbiere da cui usufruire dei servizi scelti.

* **RF30 – Seleziona Servizi:**

Il sistema dovrà permettere al cliente di selezionare i servizi che desidera ricevere.

* **RF31 – Seleziona Data:**

Il sistema dovrà permettere al cliente di selezionare la data in cui si presenterà al salone.

### Requisiti non Funzionali

Per una migliore leggibilità e organizzazione, tali requisiti sono stati suddivisi in tre aree tematiche principali:

* **Gestione Implementazione**: raccoglie i vincoli tecnici relativi alla scelta delle tecnologie e dell’architettura software utilizzata.
* **Gestione Sicurezza**: comprende i requisiti relativi alla protezione dei dati, all'autenticazione e alla gestione dei permessi.
* **Gestione Manutenzione**: include gli aspetti legati alla documentazione, alla testabilità e alla facilità di distribuzione del sistema.

****

Figura 3: Diagramma dei requisiti non funzionali

**Gestione Implementazione**

* **RNF01 – Utilizzo di Python:**

Il sistema dovrà essere sviluppato utilizzando il linguaggio Python per la logica applicativa.

* **RNF02 – Interfaccia Grafica:**

Il sistema dovrà includere un’interfaccia grafica user-friendly accessibile da desktop o browser.

* **RNF03 – Utilizzo di Database Relazionale:**

Il sistema dovrà utilizzare un database relazionale (PostgreSQL) per la gestione dei dati.

**Gestione Sicurezza**

* **RNF04 – Gestione Ruoli e Permessi:**

Il sistema dovrà distinguere chiaramente tra i ruoli utente, staff e amministratore, assegnando permessi appropriati a ciascuno.

* **RNF05 – Protezione dei Dati Utente:**

Il sistema dovrà garantire la sicurezza dei dati personali mediante l’utilizzo di credenziali.

* **RNF06 – Autenticazione Sicura:**

Il sistema dovrà implementare meccanismi di autenticazione sicura.

**Gestione Manutenzione**

* **RNF07 – Documentazione del Codice:**

Il codice del sistema dovrà essere adeguatamente documentato per favorire la manutenzione e la comprensione da parte di altri sviluppatori.

* **RNF08 – Test Automatizzati:**

Il sistema dovrà includere test automatici per le funzionalità principali, al fine di garantire stabilità e affidabilità.

* **RNF09 – Utilizzo di Docker:**

Il sistema dovrà essere eseguibile tramite container Docker, al fine di garantire una configurazione coerente tra ambienti di sviluppo, test e produzione, semplificando l’installazione e la distribuzione.

## Diagrammi dei casi d’uso

L’analisi dei requisiti ha permesso di individuare i casi d’uso del sistema, ovvero gli scenari tipici che si verificano durante l’interazione degli attori con il sistema gestionale.

Per una maggiore chiarezza e modularità, i diagrammi dei casi d’uso sono stati suddivisi in aree funzionali, corrispondenti ai diversi ambiti di interazione tra attori e sistema. Questo approccio permette di isolare e analizzare in modo indipendente le funzionalità specifiche associate a ciascun tipo di attore.

### Attori

Nella seguente immagine è rappresentato il diagramma degli attori identificati. Gli attori principali del sistema sono:

* **Utente**: attore generico che rappresenta ogni persona registrata nel sistema.
* **Cliente**: deriva dall’utente, rappresenta l’utente che usufruisce dei servizi del barber shop.
* **Barbiere**: deriva dall’utente, rappresenta i barbieri che erogano i servizi.
* **Amministratore**: deriva dall’utente con privilegi elevati per la gestione completa del sistema.
* **Tempo**: attore esterno, per azioni automatiche (es. promemoria)

Questa distinzione riflette la separazione dei ruoli e dei privilegi tra gli utenti comuni e l’amministratore, che ha accesso esclusivo a funzionalità di configurazione e controllo.



Figura 4: Diagramma degli attori

### Gestione Amministratore

Di seguito è riportata una panoramica del diagramma dei casi d’uso relativi alle funzionalità amministrative del sistema.

Il diagramma descrive le principali interazioni tra l’attore **“Amministratore”** e i casi d’uso che gli consentono di gestire il personale, monitorare la disponibilità e accedere a statistiche sul funzionamento del barber shop.



Figura 5: Diagramma della gestione dell’amministratore

#### Caso d’uso: AggiungiBarbiere

|  |
| --- |
| Caso d’uso: AggiungiBarbiere |
| ID: 1 |
| Descrizione: Questo caso d’uso permette all’amministratore di aggiungere un nuovo utente con il ruolo barbiere. |
| Attori primari: Amministratore. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’amministratore è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’amministratore seleziona l’opzione “Aggiungi barbiere”. 2. While le informazioni del nuovo barbiere non sono valide.    1. Il sistema richiede l’inserimento dei dati del nuovo barbiere.    2. Il sistema verifica la validità dei dati inseriti. 3. Il sistema genera una password per il nuovo barbiere. 4. Il sistema crea un nuovo account per il nuovo barbiere. 5. Il sistema invia la password al nuovo barbiere tramite la sua e-mail. |
| Postcondizioni: Il barbiere è stato registrato nel database del sistema. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: RimuoviBarbiere

|  |
| --- |
| Caso d’uso: RimuoviBarbiere |
| ID: 2 |
| Descrizione: Questo caso d’uso permette all’amministratore di rimuovere un account di un barbiere dal sistema. |
| Attori primari: Amministratore. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’amministratore è autenticato nel sistema. 2. Il barbiere da rimuovere è presente nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’amministratore seleziona l’opzione “Rimuovi barbiere”. 2. L’amministratore seleziona il barbiere da rimuovere. 3. Il sistema elimina l’account selezionato. |
| Postcondizioni: Il barbiere è stato rimosso dal database del sistema. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaBarbiere

|  |
| --- |
| Caso d’uso: VisualizzaBarbiere |
| ID: 3 |
| Descrizione: Questo caso d’uso permette all’amministratore di visualizzare i dati di un barbiere. |
| Attori primari: Amministratore. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’amministratore è autenticato nel sistema. 2. È stato selezionato un barbiere presente nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’amministratore seleziona l’opzione “Visualizza barbiere”. 2. Il sistema mostra le informazioni relative al barbiere selezionato. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaElencoBarbieri

|  |
| --- |
| Caso d’uso: VisualizzaElencoBarbieri |
| ID: 4 |
| Descrizione: Questo caso d’uso permette all’amministratore di visualizzare l’elenco dei barbieri presenti nel sistema. |
| Attori primari: Amministratore. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’amministratore è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’amministratore seleziona l’opzione “Visualizza elenco barbieri”. 2. If almeno un barbiere è presente nel sistema.    1. For ogni barbiere nel sistema.       1. Include (VisualizzaBarbiere). 3. Else    1. Il sistema avvisa che non ha nessun barbiere registrato nel database. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaDisponibilitaBarbieri

|  |
| --- |
| Caso d’uso: VisualizzaDisponibilitaBarbieri |
| ID: 5 |
| Descrizione: Questo caso d’uso permette all’amministratore di visualizzare le disponibilità settimanali di ogni barbiere. |
| Attori primari: Amministratore. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’amministratore è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’amministratore seleziona l’opzione “Visualizza disponibilità barbieri”. 2. If almeno un barbiere è presente nel sistema.    1. For ogni barbiere nel sistema.       1. Include (VisualizzaDisponibilitaBarbiere). 3. Else    1. Il sistema avvisa che non ha nessun barbiere registrato nel database. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaDisponibilitaBarbiere

|  |
| --- |
| Caso d’uso: VisualizzaDisponibilitaBarbiere |
| ID: 6 |
| Descrizione: Questo caso d’uso permette all’amministratore di visualizzare le disponibilità settimanali di un barbiere. |
| Attori primari: Amministratore. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’amministratore è autenticato nel sistema. 2. È stato selezionato un barbiere presente nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’amministratore seleziona l’opzione “Visualizza disponibilità barbiere”. 2. Il sistema mostra gli orari di disponibilità relative al barbiere selezionato. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: GestisciDisponibilitaBarbieri

|  |
| --- |
| Caso d’uso: GestisciDisponibilitaBarbieri |
| ID: 7 |
| Descrizione: Questo caso d’uso permette all’amministratore di modificare le disponibilità di ciascun barbiere. |
| Attori primari: Amministratore. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’amministratore è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’amministratore seleziona l’opzione “Gestisci disponibilità barbieri”. 2. Il sistema mostra la lista dei barbieri. 3. L’amministratore seleziona un barbiere 4. Il sistema mostra il calendario delle disponibilità del barbiere selezionato. 5. L’Amministratore modifica le disponibilità. 6. Il sistema salva le nuove disponibilità. |
| Postcondizioni: Le disponibilità del barbiere sono state aggiornate. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: GeneraReportStatistiche

|  |
| --- |
| Caso d’uso: GeneraReportStatistiche |
| ID: 8 |
| Descrizione: Questo caso d’uso permette all’amministratore di modificare le disponibilità di ciascun barbiere. |
| Attori primari: Amministratore. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’amministratore è autenticato nel sistema. 2. Il sistema ha dati disponibili da analizzare. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’amministratore seleziona l’opzione “Genera report statistiche”. 2. Il sistema elabora i dati statistici. 3. L’amministratore seleziona un barbiere 4. Il sistema mostra le statistiche relative all’andamento dell’attività. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

### Gestione Account

Di seguito è riportata una panoramica del diagramma dei casi d’uso relativi alla gestione degli account all’interno del sistema.

Il diagramma evidenzia le interazioni tra l’attore **“Utente”** e le principali operazioni disponibili, tra cui la registrazione, l’accesso, il recupero delle credenziali e la gestione del proprio account.

Queste funzionalità rappresentano il fulcro dei processi di autenticazione e gestione dell’identità, delineando i flussi essenziali che regolano l’accesso e la personalizzazione dell’esperienza all’interno della piattaforma.

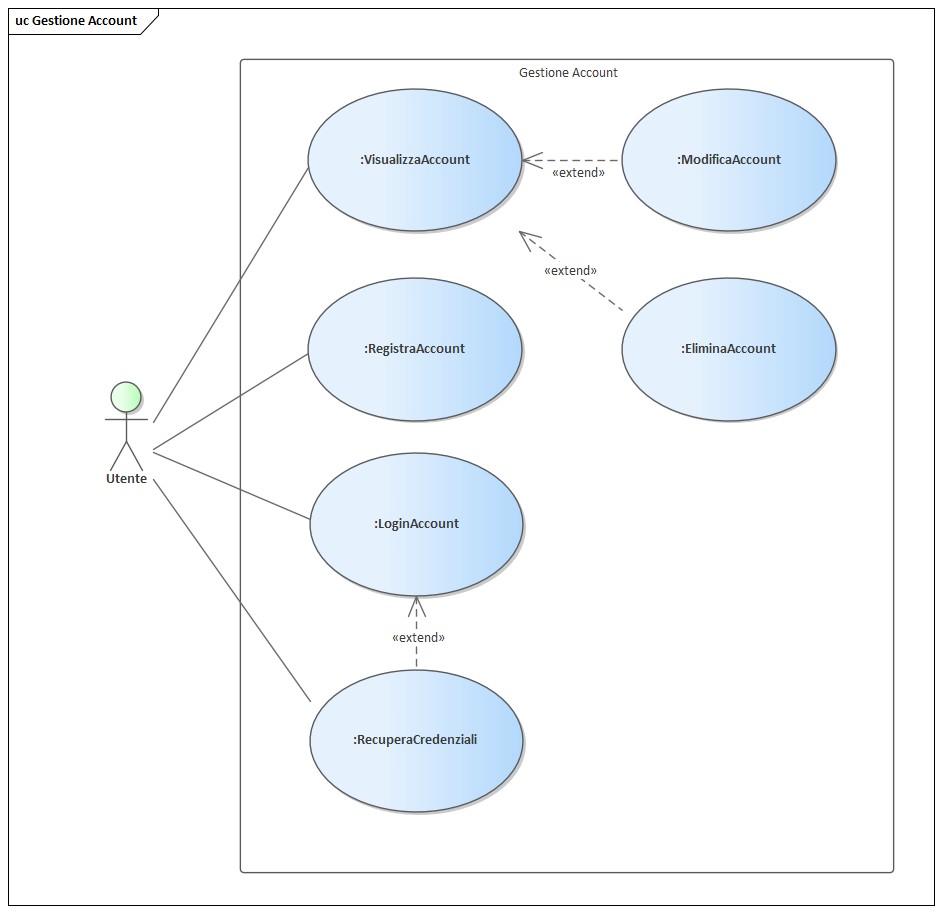


Figura 6: Diagramma della gestione degli account

#### Caso d’uso: LoginAccount

|  |
| --- |
| Caso d’uso: LoginAccount |
| ID: 9 |
| Descrizione: Questo caso d’uso permette all’utente di accedere al proprio account. |
| Attori primari: Utente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’utente deve essere già registrato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Login”. 2. While le credenziali inserite non sono valide.    1. Il sistema richiede l’inserimento di e-mail e password.    2. Il sistema verifica la validità delle credenziali inserite. 3. Il sistema autentica l’utente |
| Postcondizioni:   1. L’utente ha effettuato l’accesso al proprio account. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: RegistraAccount

|  |
| --- |
| Caso d’uso: RegistraAccount |
| ID: 10 |
| Descrizione: Questo caso d’uso permette all’utente di registrare un nuovo account sul sistema. |
| Attori primari: Utente. |
| Attori secondari: Nessuno. |
| Precondizioni: Nessuna. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Registra nuovo account”. 2. While le credenziali inserite non sono valide.    1. Il sistema richiede l’inserimento delle informazioni dell’utente: nome, cognome, e-mail e password.    2. Il sistema verifica la validità delle credenziali inserite. 3. Il sistema registra un nuovo account per l’utente nel sistema. |
| Postcondizioni:   1. Un nuovo account è stato registrato nel database per l’utente. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaAccount

|  |
| --- |
| Caso d’uso: VisualizzaAccount |
| ID: 11 |
| Descrizione: Questo caso d’uso permette all’utente di visualizzare i dati relativi al proprio account. |
| Attori primari: Utente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’utente deve essere già registrato nel sistema. 2. L’utente è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Visualizza account”. 2. Il sistema mostra le informazioni relative all’account dell’utente. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: ModificaAccount

|  |
| --- |
| Caso d’uso: ModificaAccount |
| ID: 12 |
| Descrizione: Questo caso d’uso permette all’utente di modificare i dati relativi al proprio account. |
| Attori primari: Utente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’utente deve essere già registrato nel sistema. 2. L’utente è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Modifica account”. 2. L’utente seleziona le informazioni da modificare del proprio account. 3. Il sistema salva le nuove informazioni inserite. |
| Postcondizioni:   1. Le informazioni dell’account dell’utente sono state aggiornate. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: EliminaAccount

|  |
| --- |
| Caso d’uso: EliminaAccount |
| ID: 13 |
| Descrizione: Questo caso d’uso permette all’utente di eliminare il proprio account. |
| Attori primari: Utente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’utente deve essere già registrato nel sistema. 2. L’utente è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Elimina account”. 2. L’utente seleziona le informazioni da modificare del proprio account. 3. Il sistema elimina l’account dell’utente. |
| Postcondizioni:   1. L’account dell’utente è stato rimosso dal database del sistema. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: RecuperaCredenziali

|  |
| --- |
| Caso d’uso: RecuperaCredenziali |
| ID: 14 |
| Descrizione: Questo caso d’uso permette all’utente di recuperare le proprie credenziali di accesso in caso di smarrimento. |
| Attori primari: Utente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. L’utente deve essere già registrato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Recupera credenziali”. 2. While l’e-mail inserita non è valida.    1. Il sistema all’utente di inserire l’e-mail associata al proprio account.    2. Il sistema verifica la validità dell’e-mail inserita. 3. Il sistema invia le credenziali dell’utente all’e-mail inserita. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

### Gestione Cliente

Di seguito è riportato il diagramma dei casi d’uso relativi alla gestione delle attività del Cliente all’interno del sistema.

Il diagramma rappresenta le interazioni tra l’attore principale **“Utente”** e le principali funzionalità disponibili, come la consultazione di recensioni, appuntamenti e dello storico delle prenotazioni. Il Cliente ha inoltre la possibilità di gestire autonomamente le proprie recensioni e di cancellare appuntamenti. Alcuni casi d’uso sono strutturati come inclusioni o estensioni per rendere più fluida e coerente l’esperienza.  
È inoltre previsto un sistema di promemoria automatico, in cui interviene l’attore secondario **“Tempo”**, incaricato di attivare le notifiche per gli appuntamenti imminenti.



Figura 7: Diagramma della gestione dei clienti

#### Caso d’uso: VisualizzaAppuntamento

|  |
| --- |
| Caso d’uso: VisualizzaAppuntamento |
| ID: 15 |
| Descrizione: Questo caso d’uso permette all’cliente consultare il un proprio appuntamento prenotato in passato. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. Il cliente ha già prenotato almeno un appuntamento. 4. È stato selezionato un appuntamento presente nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Visualizza appuntamento”. 2. Il sistema mostra le informazioni relative all’appuntamento selezionato del cliente. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaStoricoAppuntamenti

|  |
| --- |
| Caso d’uso: VisualizzaStoricoAppuntamenti |
| ID: 16 |
| Descrizione: Questo caso d’uso permette all’cliente consultare il proprio storico degli appuntamenti prenotati in passato. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Visualizza storico appuntamenti”. 2. If almeno un appuntamento è stato prenotato dal cliente in passato.    1. For ogni appuntamento nell’account del cliente.       1. Include (VisualizzaAppuntamento). 3. Else    1. Il sistema avvisa che il cliente non ha mai prenotato un appuntamento. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: CancellaAppuntamento

|  |
| --- |
| Caso d’uso: CancellaAppuntamento |
| ID: 17 |
| Descrizione: Questo caso d’uso permette all’cliente di cancellare l’appuntamento prenotato in corso. |
| Attori primari: Cliente. |
| Attori secondari: Tempo. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. Il cliente ha già prenotato l’appuntamento. 4. L’appuntamento è ancora in corso. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Cancella appuntamento”. 2. Include (VisualizzaAppuntamento) 3. Il cliente conferma di voler cancellare l’appuntamento. 4. Il sistema cancella la prenotazione dell’appuntamento del cliente. |
| Postcondizioni: L’appuntamento è stato cancellato dal sistema. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: NotificaPromemoriaAppuntamento

|  |
| --- |
| Caso d’uso: NotificaPromemoriaAppuntamento |
| ID: 18 |
| Descrizione: Questo caso d’uso consente al sistema di inviare automaticamente un promemoria via e-mail al cliente poco prima dell'orario previsto per l’appuntamento. |
| Attori primari: Cliente. |
| Attori secondari: Tempo, Sistema. |
| Precondizioni:   1. Il cliente ha un appuntamento registrato nel sistema. |
| Sequenza eventi principale:   1. Il sistema controlla regolarmente gli appuntamenti pianificati. 2. Quando un appuntamento si avvicina, il sistema genera automaticamente un promemoria. 3. Il sistema invia il promemoria via e-mail al cliente. |
| Postcondizioni: Il cliente ha ricevuto un promemoria via e-mail prima dell’appuntamento. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: PubblicaRecensione

|  |
| --- |
| Caso d’uso: PubblicaRecensione |
| ID: 19 |
| Descrizione: Questo caso d’uso permette al cliente di pubblicare una recensione riguardo ai servizi ottenuti. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. Il cliente ha già prenotato almeno un appuntamento per il barbiere per cui vuole recensire i servizi. 4. Il cliente non ha già pubblicato una recensione al barbiere. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Pubblica Recensione”. 2. Il sistema richiede al cliente di inserire le seguenti informazioni della recensione: titolo, descrizione e voto. 3. Il cliente inserisce le informazioni della recensione richieste. 4. Il sistema aggiunge la recensione al profilo del barbiere. |
| Postcondizioni: Il cliente ha pubblicato una recensione al barbiere. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaRecensione

|  |
| --- |
| Caso d’uso: VisualizzaRecensione |
| ID: 20 |
| Descrizione: Questo caso d’uso permette al cliente di visualizzare una recensione di un barbiere. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. È stata selezionata una recensione presente nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Visualizza recensione”. 2. Il sistema mostra la recensione selezionata. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaRecensioniBarbiere

|  |
| --- |
| Caso d’uso: VisualizzaRecensioniBarbiere |
| ID: 21 |
| Descrizione:Questo caso d’uso permette al cliente di visualizzare tutte le recensioni di un barbiere presenti nel sistema. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. È stato selezionato un barbiere presente nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Visualizza recensioni barbiere”. 2. If almeno una recensione è presente nel profilo del barbiere.    1. For ogni recensione del barbiere.       1. Include (VisualizzaRecensione). 3. Else    1. Il sistema avvisa che il barbiere selezionato non ha ancora nessuna recensione. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: ModificaRecensione

|  |
| --- |
| Caso d’uso: ModificaRecensione |
| ID: 22 |
| Descrizione: : Questo caso d’uso permette al cliente di modificare una propria recensione precedentemente pubblicata. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. Il cliente ha pubblicato una recensione al barbiere. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Modifica Recensione”. 2. Include (VisualizzaRecensione) 3. Il sistema richiede al cliente di inserire le seguenti informazioni della recensione: titolo, descrizione e voto. 4. Il cliente inserisce le informazioni della recensione richieste. 5. Il sistema modifica la recensione al profilo del barbiere. |
| Postcondizioni: I dettagli della recensione sono stati aggiornati. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: EliminaRecensione

|  |
| --- |
| Caso d’uso: EliminaRecensione |
| ID: 23 |
| Descrizione: : Questo caso d’uso permette al cliente di eliminare una propria recensione precedentemente pubblicata. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. Il cliente ha pubblicato una recensione al barbiere. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Elimina Recensione”. 2. Include (VisualizzaRecensione) 3. Il sistema rimuove la recensione al profilo del barbiere. |
| Postcondizioni: La recensione del barbiere è stata eliminata dal database del sistema. |
| Sequenza degli eventi alternativa: Nessuna. |

### Gestione Barbiere

Di seguito è riportato il diagramma dei casi d’uso delle funzionalità generali offerte dal sistema in relazione alla gestione dei barbieri.

Il diagramma evidenzia le interazioni tra l’attore principale **“Barbiere”** e le funzionalità che riguardano la gestione dei servizi offerti e la consultazione delle recensioni ricevute dai clienti. Alcuni casi d’uso sono organizzati secondo relazioni di inclusione, al fine di strutturare in modo chiaro e modulare le operazioni disponibili per l’utente.

Il diagramma si concentra sull’operatività quotidiana del Barbiere, mettendo in risalto gli strumenti a sua disposizione per mantenere aggiornate le informazioni sui servizi e monitorare il livello di soddisfazione della clientela.



Figura 8: Diagramma della gestione dei barbieri

#### Caso d’uso: AggiungiServizioBarbiere

|  |
| --- |
| Caso d’uso: AggiungiServizioBarbiere |
| ID: 24 |
| Descrizione: : Questo caso d’uso permette al barbiere di aggiungere al proprio catalogo di servizi offerti un nuovo servizio. |
| Attori primari: Barbiere. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il barbiere deve essere già registrato nel sistema. 2. Il barbiere è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Aggiungi servizio”. 2. Il sistema richiede al barbiere di inserire le seguenti informazioni del servizio: nome e prezzo. 3. Il barbiere inserisce le informazioni del servizio richieste. 4. Il sistema aggiunge il servizio al barbiere. |
| Postcondizioni: Il nuovo servizio è stato aggiunto al catalogo dei servizi del barbiere. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: ModificaServizioBarbiere

|  |
| --- |
| Caso d’uso: ModificaServizioBarbiere |
| ID: 25 |
| Descrizione: Questo caso d’uso permette al barbiere di modificare un servizio presente nel proprio catalogo di servizi offerti. |
| Attori primari: Barbiere. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il barbiere deve essere già registrato nel sistema. 2. Il barbiere è autenticato nel sistema. 3. Il barbiere deve avere almeno un servizio offerto. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Modifica Recensione”. 2. Include (VisualizzaServizioBarbiere) 3. Il sistema richiede al barbiere di inserire le seguenti informazioni del servizio: nome e prezzo. 4. Il barbiere inserisce le informazioni del servizio richieste. 5. Il sistema modifica il servizio del barbiere. |
| Postcondizioni: I dettagli del servizio offerto sono stati aggiornati. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: EliminaServizioBarbiere

|  |
| --- |
| Caso d’uso: EliminaServizioBarbiere |
| ID: 26 |
| Descrizione: Questo caso d’uso permette al barbiere di eliminare un servizio presente nel proprio catalogo di servizi offerti. |
| Attori primari: Barbiere. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il barbiere deve essere già registrato nel sistema. 2. Il barbiere è autenticato nel sistema. 3. Il barbiere deve avere almeno un servizio offerto. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Elimina servizio”. 2. Include (VisualizzaServizioBarbiere) 3. Il sistema rimuove il servizio dal profilo del barbiere. |
| Postcondizioni: Il servizio è stato rimosso dal catalogo dei servizi del barbiere. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaServizioBarbiere

|  |
| --- |
| Caso d’uso: VisualizzaRecensione |
| ID: 27 |
| Descrizione: Questo caso d’uso consente al barbiere di visualizzare uno dei servizi che offre. |
| Attori primari: Barbiere. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il barbiere deve essere già registrato nel sistema. 2. Il barbiere è autenticato nel sistema. 3. È stato selezionato un servizio offerto dal barbiere. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Visualizza servizio”. 2. Il sistema mostra la recensione selezionata. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaServiziOfferti

|  |
| --- |
| Caso d’uso: VisualizzaServiziOfferti |
| ID: 28 |
| Descrizione:Questo caso d’uso permette al barbiere di visualizzare tutti i servizi offerti nel proprio catalogo. |
| Attori primari: Barbiere. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il barbiere deve essere già registrato nel sistema. 2. Il barbiere è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Visualizza servizi offerti”. 2. If almeno un servizio è presente nel catalogo del barbiere.    1. For ogni servizio del barbiere.       1. Include (VisualizzaServizioBarbiere) 3. Else    1. Il sistema avvisa che il barbiere non ha ancora nessun servizio offerto. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaRecensioniRicevute

|  |
| --- |
| Caso d’uso: VisualizzaRecensioniRicevute |
| ID: 29 |
| Descrizione:Questo caso d’uso permette al barbiere di visualizzare le proprie recensioni ricevute dai clienti. |
| Attori primari: Barbiere. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il barbiere deve essere già registrato nel sistema. 2. Il barbiere è autenticato nel sistema. 3. Il barbiere ha ricevuto almeno una recensione. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Visualizza recensioni ricevute”. 2. If almeno una recensione è presente nel profilo del barbiere.    1. For ogni recensione del barbiere.       1. Il sistema mostra le informazioni della recensione corrente. 3. Else    1. Il sistema avvisa che il barbiere non ha ancora nessuna recensione. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

### Prenotazione Appuntamento

Di seguito è riportato il diagramma dei casi d’uso relativi alla prenotazione dell’appuntamento all’interno del sistema.

Il diagramma descrive le interazioni tra l’attore principale **“Cliente”** e le diverse fasi del processo di prenotazione. Il Cliente inizia con la funzione di prenotazione appuntamento, che include la visualizzazione dell’elenco dei barbieri, in cui può selezionare il professionista desiderato.

Successivamente, ha la possibilità di visualizzare i servizi offerti dal barbiere scelto e selezionare quelli di interesse. Infine, il Cliente visualizza la disponibilità del barbiere e seleziona la data per l’appuntamento.

Questo flusso di interazioni è strutturato in modo da garantire un processo di prenotazione chiaro e intuitivo per l’utente.



Figura 9: Diagramma delle prenotazione degli appuntamenti

#### Caso d’uso: PrenotaAppuntamento

|  |
| --- |
| Caso d’uso: PrenotaAppuntamento |
| ID: 30 |
| Descrizione:Questo caso d’uso permette al cliente di prenotare un appuntamento sul sistema. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Prenota appuntamento”. 2. Il sistema richiede al cliente di selezionare il barbiere da cui usufruire dei servizi. 3. Include (VisualizzaElencoBarbieri) 4. Il cliente seleziona il barbiere preferito. 5. Il sistema richiede al cliente di selezionare i servizi che desidera ottenere. 6. Include (VisualizzaServiziBarbiere) 7. Il cliente seleziona i servizi che desidera. 8. Il sistema richiede al cliente di selezionare la data in cui fissare l’appuntamento. 9. Include (VisualizzaDisponibilitaBarbiere) 10. Il cliente seleziona la data che preferisce. 11. Il sistema crea un nuovo appuntamento al cliente. |
| Postcondizioni: Un nuovo appuntamento è stato aggiunto nel sistema. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaElencoBarbieri

|  |
| --- |
| Caso d’uso: VisualizzaElencoBarbieri |
| ID: 32 |
| Descrizione: Questo caso d’uso permette al cliente di visualizzare l’elenco dei barbieri presenti nel sistema. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando il cliente seleziona l’opzione “Visualizza elenco barbieri”. 2. If almeno un barbiere è presente nel sistema.    1. For ogni barbiere nel sistema.       1. Il sistema mostra le informazioni del barbiere corrente. 3. Else    1. Il sistema avvisa che non ha nessun barbiere registrato nel database. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaServiziBarbiere

|  |
| --- |
| Caso d’uso: VisualizzaServiziBarbiere |
| ID: 33 |
| Descrizione:Questo caso d’uso permette al cliente di visualizzare tutti i servizi offerti nel proprio catalogo di un barbiere. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. È stato selezionato un barbiere presente nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando l’utente seleziona l’opzione “Visualizza servizi barbiere”. 2. If almeno un servizio è presente nel catalogo del barbiere.    1. For ogni servizio del barbiere.       1. Il sistema mostra le informazioni del servizio corrente. 3. Else    1. Il sistema avvisa che il barbiere non ha ancora nessun servizio offerto. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: VisualizzaDisponibilitaBarbiere

|  |
| --- |
| Caso d’uso: VisualizzaDisponibilitaBarbiere |
| ID: 34 |
| Descrizione: Questo caso d’uso permette al cliente di visualizzare le disponibilità settimanali di un barbiere. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. È stato selezionato un barbiere presente nel sistema. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando il cliente seleziona l’opzione “Visualizza disponibilità barbiere”. 2. Il sistema mostra gli orari di disponibilità relative al barbiere selezionato. |
| Postcondizioni: Nessuna. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: SelezionaBarbiere

|  |
| --- |
| Caso d’uso: SelezionaBarbiere |
| ID: 35 |
| Descrizione: Questo caso d’uso permette al cliente di selezionare un barbiere da cui usufruire dei servizi per l’appuntamento. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. È in corso la prenotazione di un appuntamento. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando il cliente seleziona l’opzione “Seleziona barbiere”. 2. Il cliente seleziona il barbiere preferito. 3. Il sistema aggiunge il barbiere alla prenotazione dell’appuntamento |
| Postcondizioni: Il barbiere selezionato è stato aggiunto all’appuntamento. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: SelezionaServizi

|  |
| --- |
| Caso d’uso: SelezionaServizi |
| ID: 36 |
| Descrizione: Questo caso d’uso permette al cliente di selezionare un i servizi di cui usufruire per l’appuntamento. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. È in corso la prenotazione di un appuntamento. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando il cliente seleziona l’opzione “Seleziona servizi”. 2. Il cliente seleziona i servizi desiderati. 3. Il sistema aggiunge i servizi alla prenotazione dell’appuntamento |
| Postcondizioni: I servizi selezionati sono stati aggiunti all’appuntamento. |
| Sequenza degli eventi alternativa: Nessuna. |

#### Caso d’uso: SelezionaData

|  |
| --- |
| Caso d’uso: SelezionaData |
| ID: 37 |
| Descrizione: Questo caso d’uso permette al cliente di selezionare la data in cui fissare l’appuntamento. |
| Attori primari: Cliente. |
| Attori secondari: Nessuno. |
| Precondizioni:   1. Il cliente deve essere già registrato nel sistema. 2. Il cliente è autenticato nel sistema. 3. È in corso la prenotazione di un appuntamento. |
| Sequenza eventi principale:   1. Il caso d’uso inizia quando il cliente seleziona l’opzione “Seleziona data”. 2. Il cliente seleziona la data preferita. 3. Il sistema aggiunge la data alla prenotazione dell’appuntamento |
| Postcondizioni: La data selezionata è stata aggiunta all’appuntamento. |
| Sequenza degli eventi alternativa: Nessuna. |

## Matrice di mapping

È importante mantenere una tracciabilità tra i casi d’uso individuati e i requisiti funzionali, poiché questo consente di verificare in modo sistematico se la progettazione del sistema procede coerentemente con gli obiettivi prefissati.

Attraverso questa corrispondenza è possibile accertarsi che ogni funzionalità prevista venga effettivamente considerata nella fase di analisi e sviluppo, garantendo così che il sistema finale rispecchi fedelmente ciò che si intende realizzare.

Inoltre, questo approccio permette di individuare tempestivamente eventuali lacune progettuali o deviazioni rispetto alle aspettative iniziali, migliorando il controllo sull’intero processo di sviluppo.



Figura 10: Matrice di mapping dei requisiti e casi d'uso

# Diagrammi di analisi

Dopo l’identificazione dei casi d’uso, sono stati sviluppati i diagrammi di analisi con l’obiettivo di rappresentare in modo strutturato e comprensibile le principali funzionalità del sistema.

Questi diagrammi consentono di visualizzare le entità fondamentali, le loro responsabilità e le relazioni che le legano, offrendo una visione d’insieme dell’architettura logica del software.

## Diagrammi delle classi di analisi

I diagrammi delle classi di analisi descrivono le entità principali del sistema e le loro relazioni logiche. Ogni package di analisi viene espanso in dettaglio per mostrare gli attributi e le operazioni più rilevanti delle classi, insieme alle generalizzazioni, associazioni e cardinalità, riflettendo quanto emerso durante l’analisi dei requisiti e dei casi d’uso.

### Package di analisi

I diagrammi delle classi di analisi rappresentano le principali entità del sistema e le loro relazioni logiche, fornendo una visione strutturata dell’architettura concettuale del software. Ogni package è stato approfondito per mostrare gli attributi e le operazioni più rilevanti delle classi coinvolte, insieme alle generalizzazioni, associazioni e cardinalità.

Questa rappresentazione riflette fedelmente i requisiti funzionali e i casi d’uso individuati durante la fase di analisi. La suddivisione è stata organizzata in base alle principali aree funzionali del sistema:

* **Utente**: contiene la gerarchia degli attori del sistema e le funzionalità ad essi associate.
* **Prenotazione**: descrive le entità coinvolte nella gestione della prenotazione degli appuntamenti.

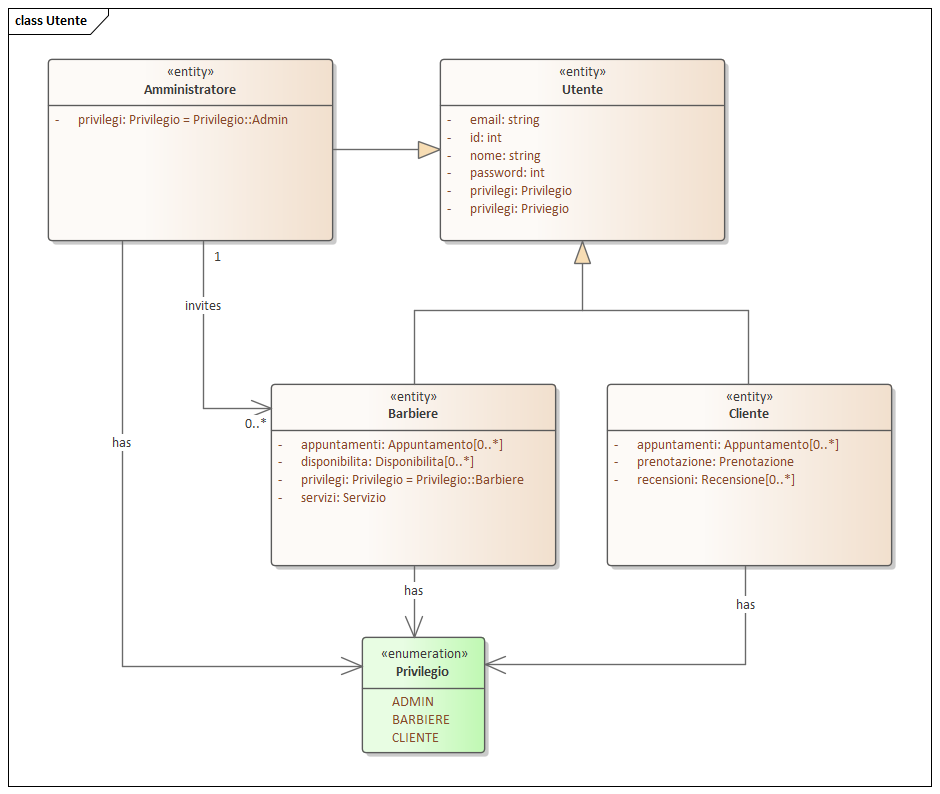
A diagram of a computer

AI-generated content may be incorrect.

### Package di analisi: Utente

Questo diagramma mostra la struttura della gerarchia degli utenti del sistema: l’entità generale Utente da cui derivano Cliente, Barbiere e Amministratore, ognuno con responsabilità e privilegi differenti.

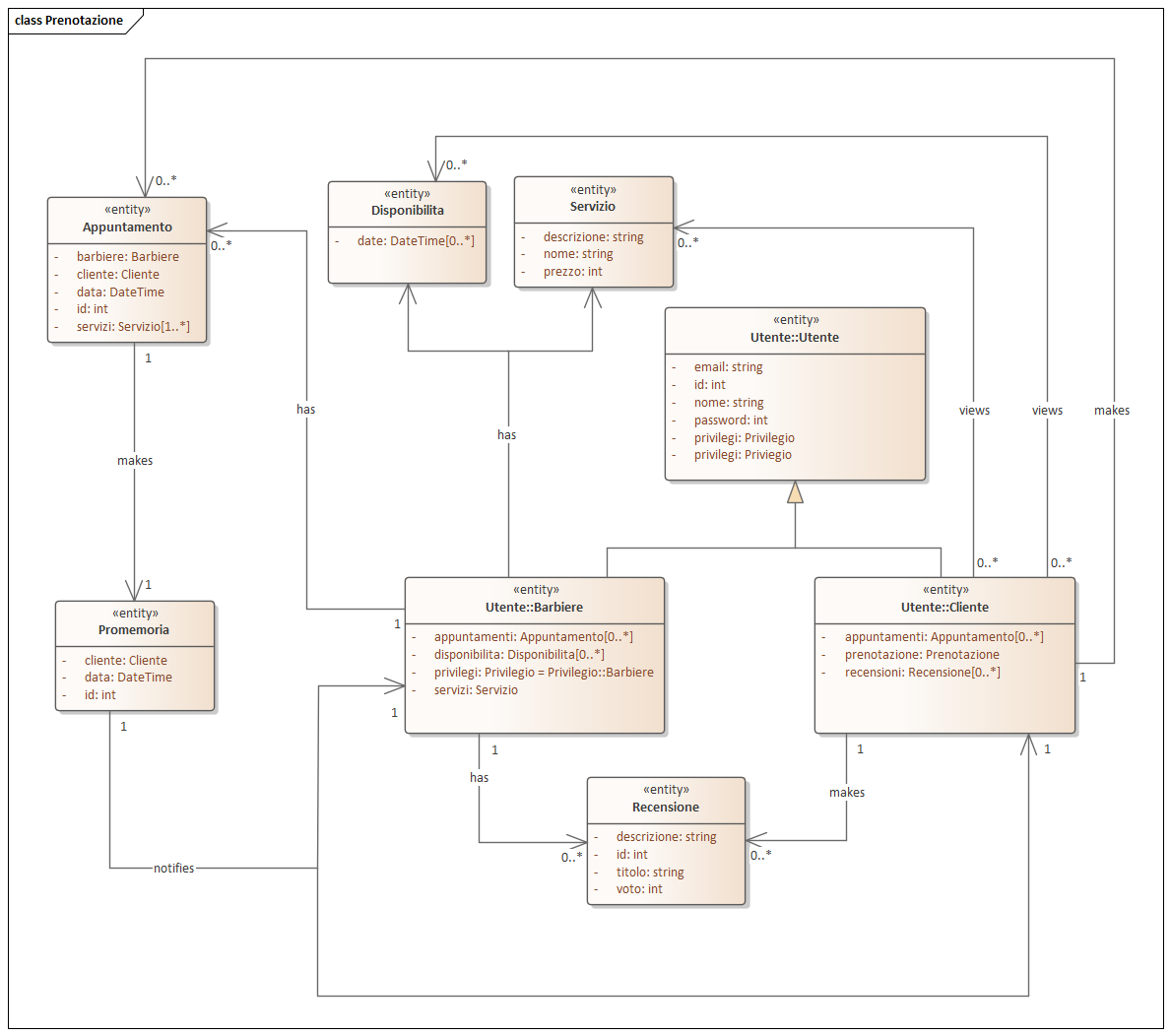
Tale organizzazione facilita la gestione dei permessi e delle funzionalità accessibili in base al ruolo.



### Package di analisi: Prenotazione

In questo diagramma sono rappresentate le entità coinvolte nel processo di prenotazione: Appuntamento, Disponibilità, Servizio e Promemoria.

Le relazioni descrivono il flusso completo della prenotazione da parte del cliente, compresa l’interazione con i barbieri e l’invio automatico dei promemoria.



## Diagrammi di sequenza

I diagrammi di sequenza rappresentano una sequenza ben definita di messaggi e operazioni necessari per eseguire una determinata funzione o scenario.

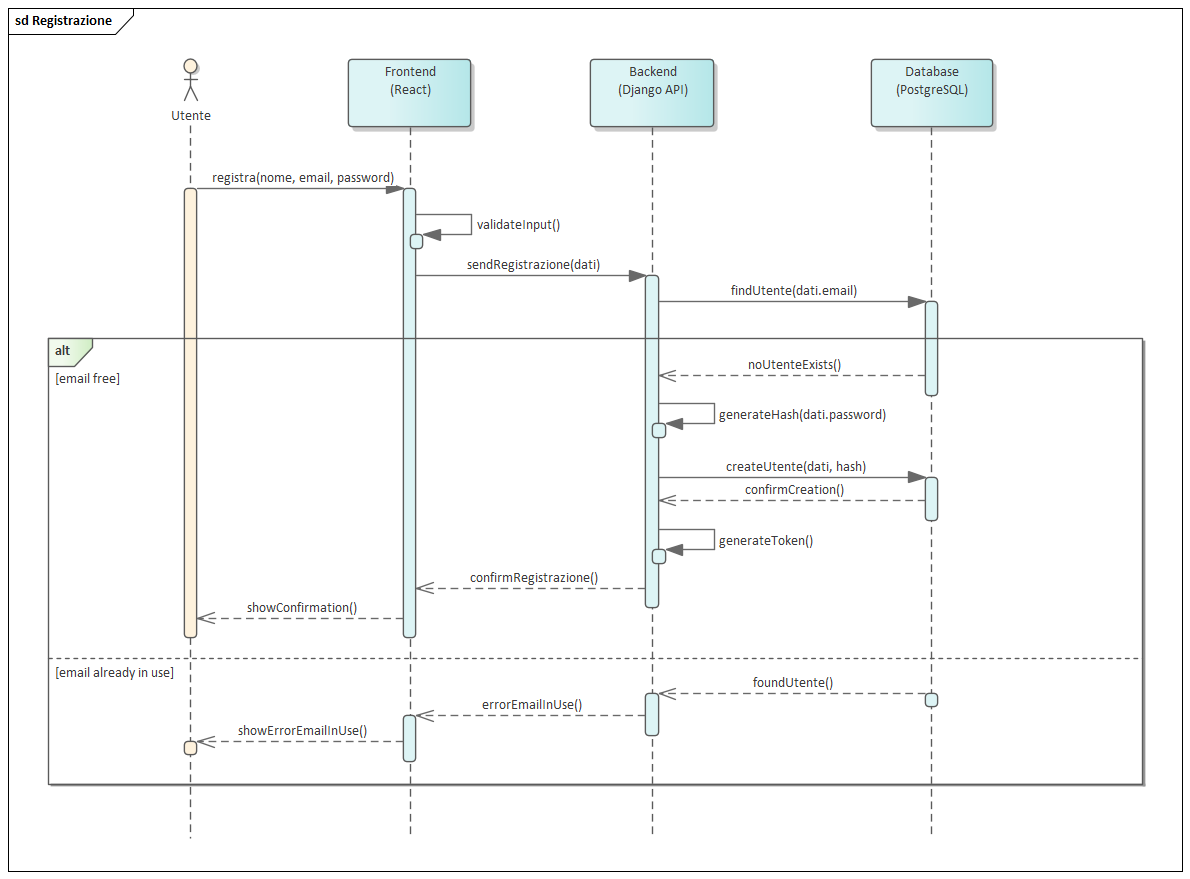
Ogni interazione è ordinata temporalmente e si sviluppa a partire dall'attivazione da parte di un attore esterno, il quale innesca una serie di comunicazioni tra i vari componenti del sistema.

Grazie a questa rappresentazione chiara e strutturata del flusso di eventi, i diagrammi di sequenza facilitano la comprensione del comportamento dinamico del sistema e supportano una progettazione e un’implementazione più efficiente e coerente del software.

### Registra Account

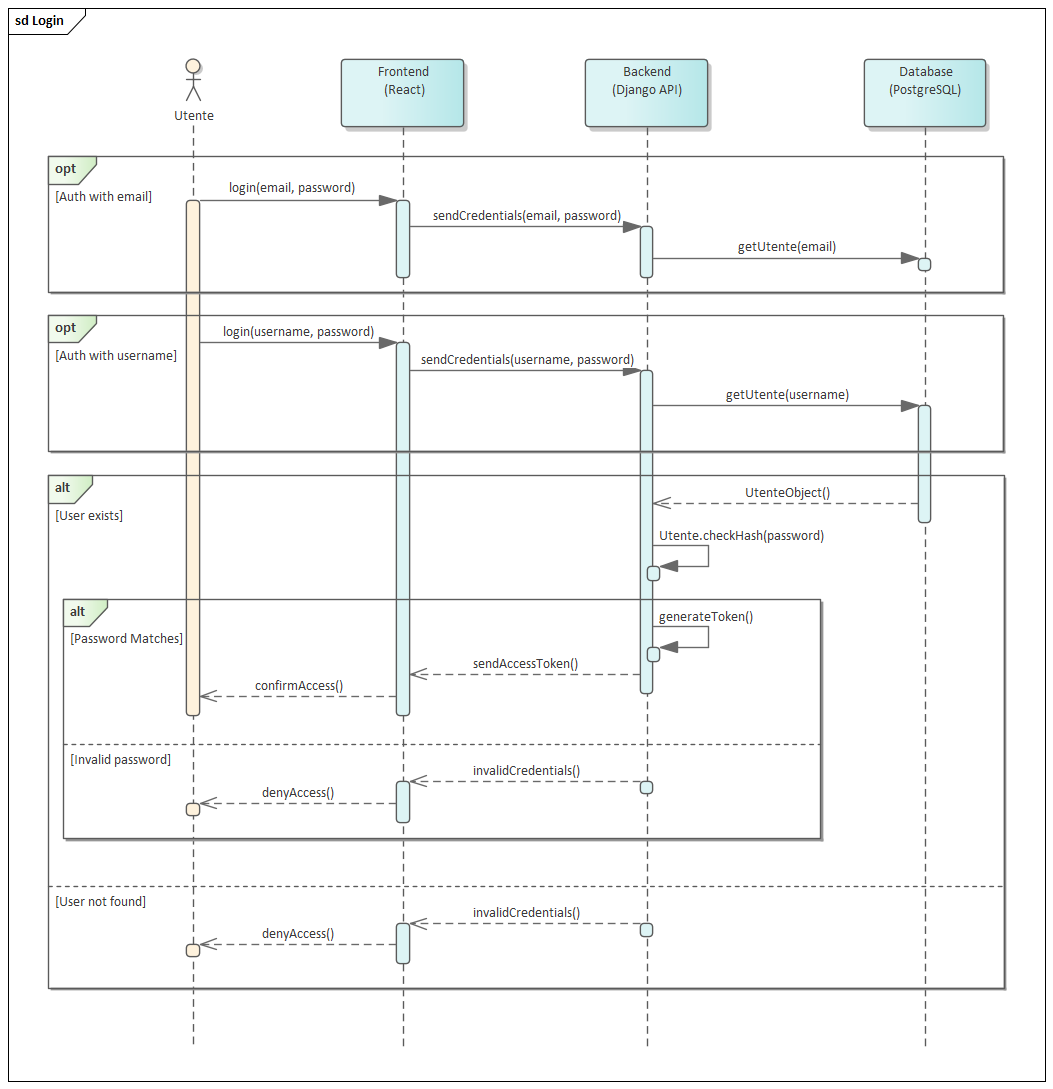
Questo diagramma descrive l’interazione tra un utente e il sistema per effettuare la registrazione. L’utente inserisce i dati richiesti, che vengono inviati al sistema, il quale valida i dati e verifica che non esistano già account con la stessa e-mail.

Se la validazione ha successo, il sistema crea un nuovo account utente e restituisce una conferma della registrazione accompagnata da un token di accesso. Altrimenti viene mostrato un messaggio di errore.



### Login Utente

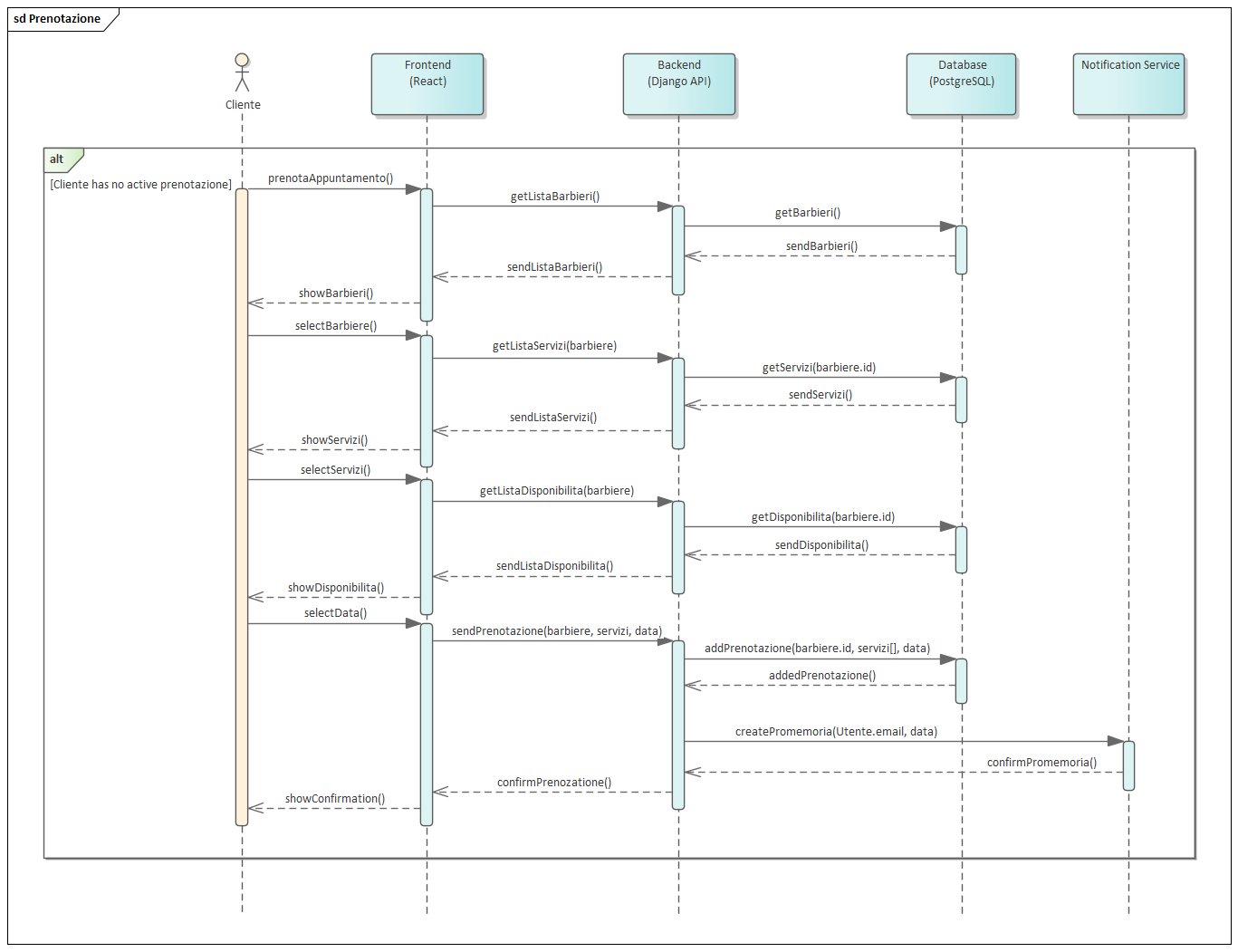
Questo diagramma descrive l’interazione tra un utente e il sistema per effettuare il login. L’utente inserisce le proprie credenziali, che vengono validate dal sistema. Se le credenziali sono corrette, viene restituito un token di autenticazione o una sessione utente attiva, altrimenti viene mostrato un messaggio di errore.



### Prenota appuntamento

Questo diagramma illustra il flusso attraverso cui un cliente può prenotare un appuntamento. Dopo aver consultato le disponibilità dei barbieri e selezionato quello preferito, il cliente sceglie i servizi desiderati tra quelli offerti. Successivamente, ha la possibilità di selezionare la data e l’orario dell’appuntamento.

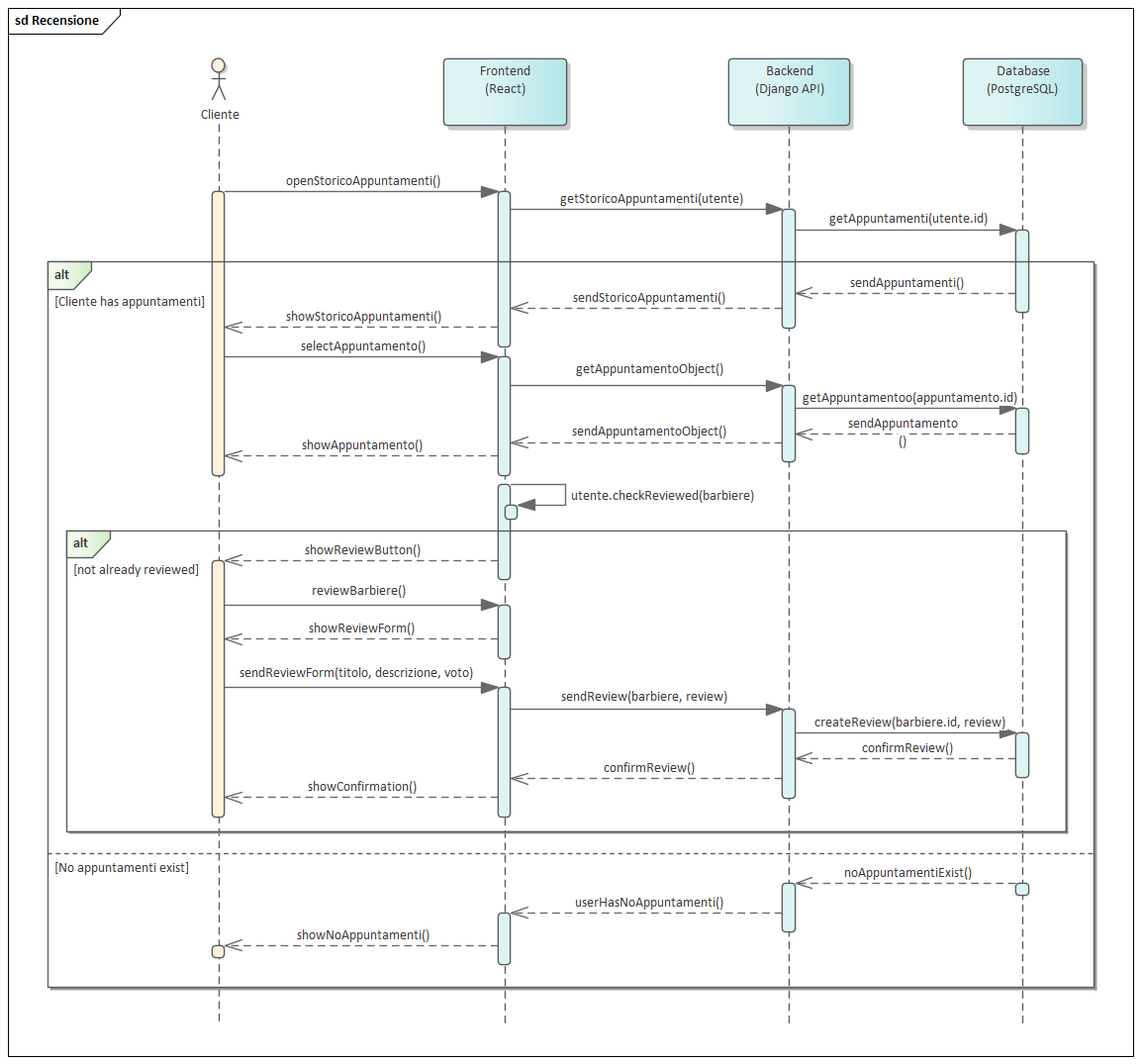
Il sistema registra quindi la prenotazione e invia automaticamente un promemoria poco prima dell’orario previsto, a conferma dell’appuntamento.



### Pubblica Recensione

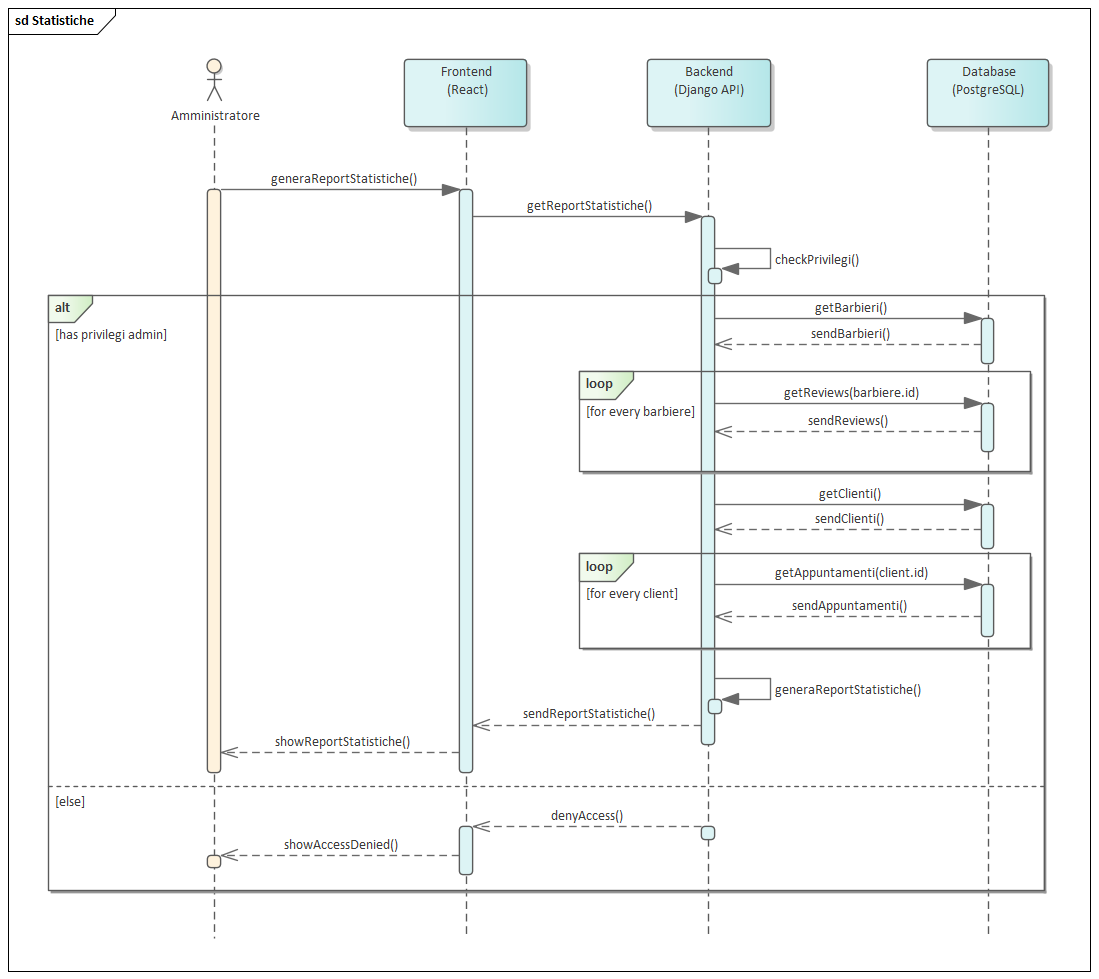
Questo diagramma descrive l’interazione tra un cliente e il sistema per lasciare una recensione. Il cliente accede allo storico degli appuntamenti completati e seleziona quello che desidera recensire.

Il sistema visualizza il modulo di recensione, che il cliente compila indicando la propria valutazione e un eventuale commento. Una volta inviato il modulo, il sistema salva la recensione e la pubblica, associandola automaticamente al barbiere che ha effettuato il servizio in quell’appuntamento.



### Genera report statistiche

Questo diagramma rappresenta l’interazione tra l’amministratore e il sistema per generare un report statistico. Includendo il numero di clienti e barbieri registrati, appuntamenti effettuati, e recensioni pubblicate. Il sistema elabora le informazioni, producendo un report consultabile e scaricabile.



## Diagrammi di attività

I diagrammi di attività mostrano il flusso operativo di un processo. Diversamente dai diagrammi di sequenza, si focalizzano sulle azioni, sulle decisioni e sui possibili percorsi alternativi di un caso d’uso.

### Registra Account

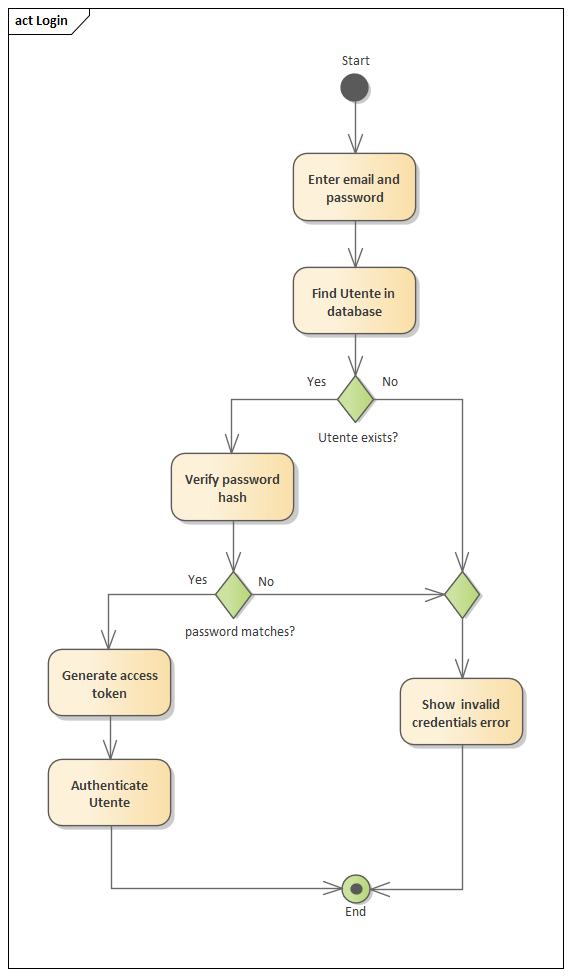
Questo diagramma descrive le attività per registrare un nuovo account: inserimento dati, verifica delle informazioni, controllo e-mail, creazione account, invio conferma o segnalazione di errore.

A diagram of a company

AI-generated content may be incorrect.

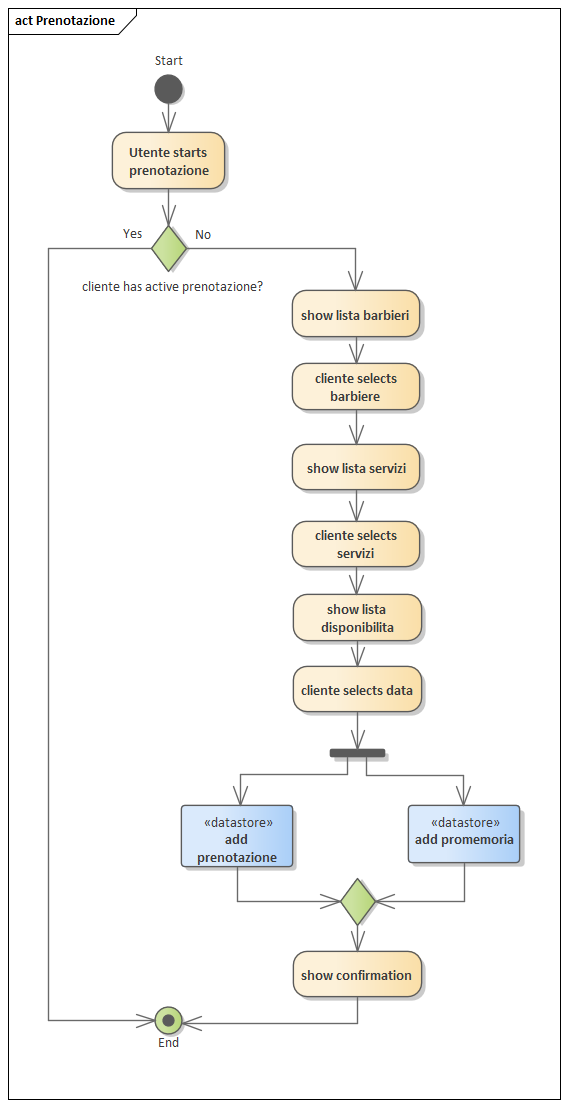
### Login Account

Il diagramma descrive il processo attraverso cui un utente accede al sistema. Comprende l’inserimento delle credenziali, la loro validazione da parte del sistema e la generazione di un token di autenticazione oppure, in caso di errore, la notifica di accesso non riuscito.



### Prenota Appuntamento

Il diagramma illustra le attività coinvolte nella prenotazione di un appuntamento. Il cliente seleziona il barbiere, i servizi desiderati e l’orario disponibile. Il sistema registra la prenotazione e programma l’invio di un promemoria automatico.



### Pubblica Recensione

Questo diagramma descrive le attività svolte dal cliente per pubblicare una recensione. Dopo aver selezionato un appuntamento completato, l’utente compila il modulo con valutazione e commento. Il sistema salva e pubblica la recensione associandola al barbiere.

A diagram of a flowchart

AI-generated content may be incorrect.

### Genera Report Statistiche

Il diagramma evidenzia le attività necessarie all’amministratore per generare un report statistico. Dopo la richiesta, il sistema raccoglie i dati su utenti, appuntamenti e recensioni, li elabora e fornisce il report pronto per la consultazione o il download.

A diagram of a work flow

AI-generated content may be incorrect.

# Diagrammi di progettazione

La fase di progettazione ha l’obiettivo di trasformare i concetti individuati durante l’analisi in un'architettura software concreta e implementabile. In questa fase sono state prese decisioni su come strutturare il sistema in termini di classi, componenti e comportamenti, tenendo conto di aspetti come modularità, manutenibilità e riusabilità del codice.

I diagrammi realizzati hanno lo scopo di guidare lo sviluppo, riducendo le ambiguità e facilitando il lavoro dei programmatori.

## Diagrammi delle classi di progettazione

Il diagramma delle classi di progettazione rappresenta l’evoluzione delle classi di analisi verso un’implementazione concreta. Le relazioni tra le sono state raffinate per tenere conto delle scelte architetturali e tecnologiche.

A diagram of a application

AI-generated content may be incorrect.

### Package di progettazione: User

Questo diagramma mostra la progettazione delle classi che rappresentano gli utenti e i loro ruoli nel sistema. La classe base **User**, derivata da **AbstractUser**, viene specializzata in **Admin**, **Barber** e **Client**, ciascuno con attributi e metodi specifici. La presenza dell’enumerazione **Roles** permette l’assegnazione dei profili e la gestione centralizzata delle funzionalità legate ai diversi tipi di utenza.

A screenshot of a computer flowchart

AI-generated content may be incorrect.

### Package di progettazione: Appointment

Questo diagramma mostra la progettazione delle classi per la gestione delle prenotazioni, con **Appointment** come entità centrale collegata a servizi, disponibilità e **Review** per le recensioni.

La classe **AppointmentService** è stata introdotta per garantire che i dettagli dei servizi associati agli appuntamenti restino invariati anche se il barbiere li modifica o elimina in seguito, assicurando la coerenza dei dati storici.

A diagram of a computer program

AI-generated content may be incorrect.

## Diagramma dei componenti

Il diagramma dei componenti rappresenta l’organizzazione logica del sistema in moduli software (componenti) e le loro dipendenze.

Sono stati identificati i principali componenti funzionali, come: **Frontend** (interfaccia utente), **Backend** (logica applicativa), **Database** (persistenza) e **Sistema di notifica** (per notifiche e-mail).

Il diagramma aiuta a visualizzare come il sistema è suddiviso in blocchi coesi e indipendenti, che comunicano attraverso interfacce ben definite.

<DIAGRAMMA DEI COMPONENTI QUI>

## Diagrammi delle macchine a stati

Per descrivere il comportamento dinamico di alcune classi chiave, sono stati modellati i diagrammi delle macchine a stati. Questi diagrammi mostrano come lo stato interno di un oggetto cambia in risposta agli eventi, aiutando a comprendere meglio il ciclo di vita di entità complesse.

<DIAGRAMMI PER 3 CLASSI DA SCEGLIERE>

# Implementazione

Questa sezione descrive l’architettura generale del sistema, le tecnologie principali utilizzate e le scelte progettuali adottate nella realizzazione della soluzione, corredando il tutto con diagrammi tecnici, mockup e riferimenti metodologici.

## Diagrammi di deployment

In questa sezione sono illustrati i diagrammi di deployment che rappresentano la disposizione delle componenti su cui il sistema viene eseguito. Tali diagrammi permettono una chiara visualizzazione dei processi distribuiti, delle interazioni tra server e client, e delle integrazioni con servizi esterni.

<INSERIRE DIAGRAMMA DI DEPLOYMENT>

## Tecnologie utilizzate

Qui vengono presentate tutte le principali tecnologie, librerie e framework impiegati per lo sviluppo del sistema, sia per quanto riguarda il back-end che il front-end. La scelta delle tecnologie è stata orientata a garantire affidabilità, scalabilità e semplicità nella manutenzione.

<TECNOLOGIE UTILIZZATE ELENCO>

## Mockup

Questa sezione contiene i mockup grafici realizzati nella fase progettuale. Essi forniscono una rappresentazione visiva delle principali interfacce utente e aiutano a comprendere il flusso delle operazioni all’interno dell’applicazione.

<INSERIRE MOCKUP QUI>

## Descrizione unit test

La seguente sezione descrive la suite di unit test realizzati con Django **TestCase**/**APITestCase**, progettata per garantire l’affidabilità e la correttezza delle principali funzionalità applicative e dei casi d’uso del sistema. I test sono organizzati per macrocategoria utente e coprono sia i casi principali di interazione che le situazioni di errore e controllo degli accessi, assicurando una copertura completa e la possibilità di individuare tempestivamente anomalie in fase di sviluppo.

### Test profilo amministratore

La classe **AdminProfileTest** raccoglie tutti gli unit test relativi alle funzionalità del profilo dell’utente amministratore.

In particolare, vengono verificati i seguenti aspetti:

* la possibilità per l’admin di visualizzare il proprio profilo e i dati statistici aggregati,
* la gestione delle anagrafiche di barbieri e clienti (visualizzazione, invito tramite email, eliminazione),
* la creazione e modificazione delle disponibilità giornaliere dei barbieri,
* la visualizzazione di tutti gli appuntamenti del sistema.

I test comprendono anche i controlli di permesso, assicurando che solo l’utente amministratore possa accedere alle endpoint riservate e gestire le risorse critiche. La copertura di questi casi contribuisce a individuare tempestivamente eventuali malfunzionamenti logici o violazioni di sicurezza, mantenendo stabile la parte più sensibile dell’applicativo.

class AdminProfileTest(APITestCase):

    """

    Tests for all admin features, profile getter, barber and availabilities management, statistics generation.

    """

    def test\_get\_admin\_profile\_success(self):

        """

        Admin can fetch their profile, and sees all statistics on their dashboard.

        """

        service\_1 = Service.objects.create(

            barber=self.barber,

            name="Haircut",

            price=Decimal("5.50")

        )

        service\_2 = Service.objects.create(

            barber=self.barber,

            name="Buzz",

            price=Decimal("10.00")

        )

        # Create cancelled appointment

        appointment\_1 = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber,

            date=datetime.date.today(),

            slot=datetime.time(12, 0),

            status=AppointmentStatus.CANCELLED.value,

        )

        \_add\_services(appointment\_1, [service\_1, service\_2])

        # Create appointment same day as cancelled one

        appointment\_2 = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber,

            date=datetime.date.today(),

            slot=datetime.time(14, 0),

            status=AppointmentStatus.COMPLETED.value,

        )

        \_add\_services(appointment\_2, [service\_1, service\_2])

        # Create completed appointment on another day

        appointment\_3 = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber,

            date=datetime.date.today() + datetime.timedelta(days=1),

            slot=datetime.time(14, 0),

            status=AppointmentStatus.COMPLETED.value,

        )

        \_add\_services(appointment\_3, [service\_2])

        # Create ongoing appointment on another day

        appointment\_4 = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber,

            date=datetime.date.today() + datetime.timedelta(days=2),

            slot=datetime.time(15, 0),

            status=AppointmentStatus.ONGOING.value,

        )

        \_add\_services(appointment\_4, [service\_2])

        # Create review for completed appointment

        Review.objects.create(

            client=self.client\_user,

            barber=self.barber,

            rating=5,

            comment="top!"

        )

        # Create another client

        client\_1 = Client.objects.create\_user(

            username="testerclient",

            password="sugomadic",

            email="clients@xxx.com",

            name="Bomber",

            surname="Romber",

            is\_active=True,

        )

        # Create another barber

        barber\_1 = Barber.objects.create\_user(

            username="testerbarber",

            password="ligmabals",

            email="barbers@xxx.com",

            name="Dudong",

            surname="Sorcer",

            is\_active=True,

        )

        # Create appointment by another client

        appointment\_5 = Appointment.objects.create(

            client=client\_1,

            barber=barber\_1,

            date=datetime.date.today(),

            slot=datetime.time(14, 0),

            status=AppointmentStatus.COMPLETED.value,

        )

        \_add\_services(appointment\_5, [service\_2])

        # Create review by another client for this barber

        Review.objects.create(

            client=client\_1,

            barber=self.barber,  # Review is for self.barber

            rating=2,

            comment="trash..."

        )

        self.login\_as\_admin()

        response = self.client.get(self.manage\_profile\_url)

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        profile = response.data["profile"]

        self.assertEqual(profile, self.admin.to\_dict())

        self.assertEqual(profile['role'], Roles.ADMIN.value)

    def test\_get\_admin\_profile\_requires\_admin(self):

        """

        Only admins can access profile admin endpoint (403/401 otherwise).

        """

        # Not authenticated at all

        response = self.client.get(self.manage\_profile\_url)

        self.assertEqual(response.status\_code, status.HTTP\_401\_UNAUTHORIZED)

        self.login\_as\_client()

        response = self.client.get(self.manage\_profile\_url)

        self.assertEqual(response.status\_code, status.HTTP\_403\_FORBIDDEN)

    def test\_get\_barbers\_list\_success(self):

        """

        Return a list of all barbers to authenticated admin

        """

        self.login\_as\_admin()

        response = self.client.get(self.get\_all\_barbers)

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        barbers = response.data["barbers"]

        self.assertIn(self.barber.to\_dict(), barbers)

        self.assertIn(self.barber\_inactive.to\_dict(), barbers)

    def test\_get\_clients\_list\_success(self):

        """

        Return a list of all clients to authenticated admin

        """

        # Create another client

        client\_1 = Client.objects.create\_user(

            username="testerclient",

            password="sugomadic",

            email="clients@xxx.com",

            name="Bomber",

            surname="Romber",

            is\_active=True,

        )

        self.login\_as\_admin()

        response = self.client.get(self.get\_all\_clients)

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        clients = response.data["clients"]

        self.assertIn(self.client\_user.to\_dict(), clients)

        self.assertIn(client\_1.to\_dict(), clients)

    def test\_invite\_barber\_success(self):

        """

        Admin can invite a barber via email: creates inactive user, sends invite.

        """

        self.login\_as\_admin()

        email = "newbarb@nowhere.com"

        response = self.client.post(self.invite\_url, {"email": email}, format="json")

        self.assertEqual(response.status\_code, status.HTTP\_201\_CREATED)

        self.assertIn("Barber invited successfully", response.data["detail"])

        # Barber should now exist, inactive

        newbarber = Barber.objects.get(email=email)

        self.assertFalse(newbarber.is\_active)

        # Email send side effect: should have been called (if using real mail system)

        self.assertGreaterEqual(len(mail.outbox), 1)

        self.assertIn(email, mail.outbox[-1].to)

    def test\_invite\_barber\_existing\_email(self):

        """

        Fails if trying to invite using an email already registered.

        """

        self.login\_as\_admin()

        resp = self.client.post(self.invite\_url, {"email": self.barber.email}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("already taken", str(resp.data["detail"]))

    def test\_invite\_barber\_requires\_admin(self):

        """

        Only admin can invite barbers.

        """

        resp = self.client.post(self.invite\_url, {"email": "who@foo.com"}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_401\_UNAUTHORIZED)

        self.login\_as\_client()

        resp = self.client.post(self.invite\_url, {"email": "some@b.com"}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_403\_FORBIDDEN)

    def test\_delete\_barber\_success(self):

        """

        Admin can delete an active barber by id.

        """

        self.login\_as\_admin()

        url = reverse("delete\_barber", kwargs={"barber\_id": self.barber.id})

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_204\_NO\_CONTENT)

        self.assertFalse(Barber.objects.filter(pk=self.barber.pk).exists())

    def test\_delete\_barber\_not\_found(self):

        """

        Returns error if barber does not exist.

        """

        self.login\_as\_admin()

        url = reverse("delete\_barber", kwargs={"barber\_id": 99991})

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("does not exist", str(resp.data["id"][0]))

    def test\_delete\_barber\_requires\_admin(self):

        """

        Only admins can delete barbers.

        """

        url = reverse("delete\_barber", kwargs={"barber\_id": self.barber.id})

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_401\_UNAUTHORIZED)

        self.login\_as\_client()

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_403\_FORBIDDEN)

    def test\_create\_availability\_success(self):

        """

        Admin can create a barber's availability for a single date via new bulk API schema.

        """

        self.login\_as\_admin()

        url = reverse("create\_barber\_availability", kwargs={"barber\_id": self.barber.id})

        today = datetime.date.today()

        data = {

            "start\_date": str(today),

            "start\_time": "10:00",

            "end\_time": "12:00",

            "slot\_interval": 60,   # so slots: ["10:00", "11:00"], same as old test

        }

        resp = self.client.post(url, data, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_201\_CREATED)

        self.assertIn("created successfully", resp.data["detail"])

        self.assertTrue(Availability.objects.filter(barber=self.barber, date=today).exists())

        avail = Availability.objects.get(barber=self.barber, date=today)

        self.assertEqual(sorted(avail.slots), ["10:00", "11:00"])

    def test\_create\_availability\_upserts(self):

        """

        Creating for same barber/date should update the availability (not fail).

        """

        today = datetime.date.today()

        Availability.objects.create(barber=self.barber, date=today, slots=["09:00"])

        self.login\_as\_admin()

        url = reverse("create\_barber\_availability", kwargs={"barber\_id": self.barber.id})

        data = {

            "start\_date": str(today),

            "start\_time": "10:00",

            "end\_time": "12:00",

            "slot\_interval": 60,   # slots: ["10:00", "11:00"]

        }

        resp = self.client.post(url, data, format="json")

        # Now should be 201, not 400; upsert/overwrite OK

        self.assertEqual(resp.status\_code, status.HTTP\_201\_CREATED)

        self.assertIn("created successfully", resp.data["detail"])

        avail = Availability.objects.get(barber=self.barber, date=today)

        self.assertEqual(sorted(avail.slots), ["10:00", "11:00"])

    def test\_create\_availability\_requires\_admin(self):

        """

        Only admins may create availabilities for barbers.

        """

        url = reverse("create\_barber\_availability", kwargs={"barber\_id": self.barber.id})

        today = datetime.date.today()

        data = {"date": str(today), "slots": ["13:00"]}

        resp = self.client.post(url, data, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_401\_UNAUTHORIZED)

        self.login\_as\_client()

        resp = self.client.post(url, data, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_403\_FORBIDDEN)

    def test\_update\_availability\_success(self):

        """

        Admin can patch the slots of an availability by id, using start\_time/end\_time [slot\_interval optional].

        """

        availability = Availability.objects.create(

            barber=self.barber,

            date=datetime.date.today(),

            slots=["09:30"]

        )

        self.login\_as\_admin()

        url = reverse("manage\_barber\_availability", kwargs={"barber\_id": self.barber.id, "availability\_id": availability.id})

        # Want slots at 10:30 and 11:30? Use start=10:30, end=12:00, interval=60

        response = self.client.patch(

            url,

            {

                "start\_time": "10:30",

                "end\_time": "12:30",

                "slot\_interval": 60  # So slots generated will be 10:30 and 11:30

            },

            format="json"

        )

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        availability.refresh\_from\_db()

        self.assertEqual(availability.date, datetime.date.today())

        self.assertEqual(sorted(availability.slots), sorted(["10:30", "11:30"]))

    def test\_update\_availability\_default\_interval(self):

        """

        If slot\_interval is omitted, default (30) is used.

        """

        availability = Availability.objects.create(

            barber=self.barber,

            date=datetime.date.today(),

            slots=["08:00"]

        )

        self.login\_as\_admin()

        url = reverse("manage\_barber\_availability", kwargs={"barber\_id": self.barber.id, "availability\_id": availability.id})

        response = self.client.patch(

            url,

            {

                "start\_time": "09:00",

                "end\_time": "10:00"

                # slot\_interval omitted, so 30min slots expected: 09:00, 09:30

            },

            format="json"

        )

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        availability.refresh\_from\_db()

        self.assertEqual(availability.slots, ["09:00", "09:30"])

    def test\_update\_availability\_not\_found(self):

        """

        Returns error if availability does not exist for given barber+id.

        """

        self.login\_as\_admin()

        url = reverse("manage\_barber\_availability", kwargs={"barber\_id": self.barber.id, "availability\_id": 99994})

        response = self.client.patch(

            url,

            {

                "start\_time": "10:00",

                "end\_time": "12:00"

            },

            format="json"

        )

        # This will now allow the serializer to reach the "does not exist" error for that (barber, id)

        self.assertEqual(response.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        error\_str = str(response.data)

        # Could also be in a nested key, depending on serializer, so check all keys

        self.assertIn("does not exist", error\_str)

    def test\_update\_availability\_requires\_start\_and\_end\_time(self):

        """

        start\_time and end\_time are required for PATCH.

        """

        availability = Availability.objects.create(

            barber=self.barber,

            date=datetime.date.today(),

            slots=["09:00"]

        )

        self.login\_as\_admin()

        url = reverse("manage\_barber\_availability", kwargs={"barber\_id": self.barber.id, "availability\_id": availability.id})

        response = self.client.patch(url, {}, format="json")

        self.assertEqual(response.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        # Since response.data might be a string (JSON dumped), parse if needed

        if isinstance(response.data, str):

            import ast

            resp\_dict = ast.literal\_eval(response.data)

        else:

            resp\_dict = response.data

        self.assertIn("start\_time", resp\_dict)

        self.assertIn("end\_time", resp\_dict)

        self.assertIn("This field is required.", [e for e in resp\_dict["start\_time"]][0])

        self.assertIn("This field is required.", [e for e in resp\_dict["end\_time"]][0])

    def test\_delete\_availability\_success(self):

        """

        Admin can delete a barber's availability by id.

        """

        avail = Availability.objects.create(

            barber=self.barber, date=datetime.date.today(), slots=["09:40"]

        )

        self.login\_as\_admin()

        url = reverse(

            "manage\_barber\_availability",

            kwargs={"barber\_id": self.barber.id, "availability\_id": avail.id},

        )

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_204\_NO\_CONTENT)

        self.assertFalse(Availability.objects.filter(pk=avail.id).exists())

    def test\_delete\_availability\_not\_found(self):

        """

        Returns error if availability does not exist.

        """

        self.login\_as\_admin()

        url = reverse(

            "manage\_barber\_availability",

            kwargs={"barber\_id": self.barber.id, "availability\_id": 123213},

        )

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("does not exist", str(resp.data["detail"]))

    def test\_admin\_get\_all\_appointments(self):

        """

        Admin can fetch all system appointments: should include all records.

        """

        service\_1 = Service.objects.create(

            barber=self.barber,

            name="Trim",

            price=20

        )

        appointment\_1 = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber,

            date=datetime.date.today(),

            slot=datetime.time(9, 0),

            status=AppointmentStatus.COMPLETED.value

        )

        AppointmentService.objects.create(

            appointment=appointment\_1,

            name=service\_1.name,

            price=service\_1.price,

            original\_service=service\_1

        )

        self.login\_as\_admin()

        response = self.client.get(self.all\_appointments\_url)

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        self.assertIn("appointments", response.data)

        self.assertTrue(any(a["id"] == appointment\_1.id for a in response.data["appointments"]))

### Test profilo barbiere

La classe **BarberProfileTest** raccoglie tutti gli unit test relativi alle funzionalità del profilo barbiere.  
In particolare, vengono verificati i seguenti aspetti:

* la possibilità per il barbiere di visualizzare, modificare ed eliminare il proprio profilo,
* la gestione dei servizi offerti (visualizzazione, creazione, modifica, cancellazione),
* la consultazione delle proprie disponibilità future e degli appuntamenti,
* la visualizzazione delle recensioni ricevute.

I test includono anche i controlli di permesso, assicurando che solo il barbiere autenticato possa accedere e modificare le proprie risorse, e che siano gestiti correttamente errori e vincoli (come la duplicazione dei nomi). La copertura di questi casi consente di individuare rapidamente anomalie e garantisce la sicurezza delle operazioni riservate al barbiere.

class BarberProfileTest(APITestCase):

    """

    Tests for the barber profile management and service/availability endpoints.

    """

    def test\_get\_profile\_success(self):

        """

        Authenticated barber can get their full profile including services and reviews.

        """

        self.login\_as\_barber()

        response = self.client.get(self.profile\_url)

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        profile = response.data["profile"]

        self.assertEqual(profile, self.barber\_user.to\_dict())

        self.assertEqual(profile['role'], Roles.BARBER.value)

    def test\_get\_profile\_requires\_auth\_and\_barber\_role(self):

        """

        Getting profile requires barber authentication; returns 403 for non-barber.

        """

        # Not authenticated at all

        response = self.client.get(self.profile\_url)

        self.assertEqual(response.status\_code, status.HTTP\_401\_UNAUTHORIZED)

        # Auth as client, not barber

        self.login\_as\_client()

        response = self.client.get(self.profile\_url)

        self.assertEqual(response.status\_code, status.HTTP\_403\_FORBIDDEN)

    def test\_update\_profile\_success\_partial(self):

        """

        Barber can update part(s) of their profile.

        """

        self.login\_as\_barber()

        patch = {"name": "Changed1", "username": "Changed2", "description": "Changed3"}

        resp = self.client.patch(self.profile\_url, patch, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_200\_OK)

        self.assertEqual(resp.data["detail"], "Profile info updated successfully.")

        self.barber\_user.refresh\_from\_db()

        self.assertEqual(self.barber\_user.name, "Changed1")

        self.assertEqual(self.barber\_user.username, "Changed2")

        self.assertEqual(self.barber\_user.description, "Changed3")

    def test\_update\_profile\_username\_unique\_constraint(self):

        """

        Username cannot be changed to one in use.

        """

        # Create another barber

        other\_barber = Barber.objects.create\_user(

            username="barber2",

            email="barber2@example.com",

            password="BarberPass456!",

            name="B2",

            surname="X",

        )

        self.login\_as\_barber()

        resp = self.client.patch(self.profile\_url, {"username": "barber2"}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("taken", str(resp.data["detail"]).lower())

    def test\_update\_profile\_requires\_at\_least\_one\_field(self):

        """

        Updating profile with no fields returns validation error.

        """

        self.login\_as\_barber()

        resp = self.client.patch(self.profile\_url, {}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("must provide at least one field", str(resp.data["detail"]))

    def test\_delete\_profile\_success(self):

        """

        Barber can delete their profile account.

        """

        self.login\_as\_barber()

        resp = self.client.delete(self.profile\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_204\_NO\_CONTENT)

        # Account should not exist anymore

        self.assertFalse(Barber.objects.filter(pk=self.barber\_user.pk).exists())

    def test\_list\_services(self):

        """

        Authenticated barber can list all their services.

        """

        Service.objects.create(barber=self.barber\_user, name="Cut", price=Decimal("10.99"))

        Service.objects.create(barber=self.barber\_user, name="Shave", price=Decimal("7.99"))

        self.login\_as\_barber()

        resp = self.client.get(self.services\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_200\_OK)

        self.assertEqual(len(resp.data["services"]), 2)

        self.assertEqual(resp.data["services"][0]["name"], "Cut")

    def test\_list\_services\_barber\_only(self):

        """

        Only authenticated barbers may list services.

        """

        resp = self.client.get(self.services\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_401\_UNAUTHORIZED)

        self.login\_as\_client()

        resp2 = self.client.get(self.services\_url)

        self.assertEqual(resp2.status\_code, status.HTTP\_403\_FORBIDDEN)

    def test\_create\_service\_success(self):

        """

        Barber can create a new service with unique name.

        """

        self.login\_as\_barber()

        data = {"name": "Buzz Cut", "price": "20.50"}

        resp = self.client.post(self.services\_url, data, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_201\_CREATED)

        self.assertIn("Service added successfully", resp.data["detail"])

        service = Service.objects.get(barber=self.barber\_user, name="Buzz Cut")

        self.assertEqual(str(service.price), data["price"])

    def test\_create\_service\_duplicate\_name\_case\_insensitive(self):

        """

        Creating service fails if name exists (case-insensitive).

        """

        Service.objects.create(barber=self.barber\_user, name="beard trim", price=Decimal("15.00"))

        self.login\_as\_barber()

        resp = self.client.post(self.services\_url, {"name": "Beard Trim", "price": "25.00"}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("already exists", str(resp.data["detail"]))

    def test\_update\_service\_success(self):

        """

        Barber can patch (partial update) a given service.

        """

        service = Service.objects.create(barber=self.barber\_user, name="Trim", price=Decimal("9.99"))

        self.login\_as\_barber()

        url = reverse("manage\_barber\_service", kwargs={"service\_id": service.id})

        resp = self.client.patch(url, {"price": "11.11"}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_200\_OK)

        service.refresh\_from\_db()

        self.assertEqual(str(service.price), "11.11")

    def test\_update\_service\_name\_duplicate(self):

        """

        Cannot update a service's name to one already in use.

        """

        s1 = Service.objects.create(barber=self.barber\_user, name="a", price=1)

        s2 = Service.objects.create(barber=self.barber\_user, name="b", price=2)

        self.login\_as\_barber()

        url = reverse("manage\_barber\_service", kwargs={"service\_id": s2.id})

        resp = self.client.patch(url, {"name": "a"}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("already exists", str(resp.data["detail"]))

    def test\_update\_service\_requires\_field(self):

        """

        Update service requires at least a name or price.

        """

        service = Service.objects.create(barber=self.barber\_user, name="Skin", price=10)

        self.login\_as\_barber()

        url = reverse("manage\_barber\_service", kwargs={"service\_id": service.id})

        resp = self.client.patch(url, {}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("at least one field", str(resp.data["detail"]))

    def test\_update\_service\_not\_found(self):

        """

        Updating a non-existent service for this barber returns error.

        """

        self.login\_as\_barber()

        url = reverse("manage\_barber\_service", kwargs={"service\_id": 9999})

        resp = self.client.patch(url, {"price": "5.00"}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("does not exist", str(resp.data["detail"]))

    def test\_delete\_service\_success(self):

        """

        Barber can delete one of their own services.

        """

        service = Service.objects.create(barber=self.barber\_user, name="Old", price=3)

        self.login\_as\_barber()

        url = reverse("manage\_barber\_service", kwargs={"service\_id": service.id})

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_204\_NO\_CONTENT)

        self.assertFalse(Service.objects.filter(pk=service.id).exists())

    def test\_delete\_service\_not\_found(self):

        """

        Deleting a non-existent service for this barber returns error.

        """

        self.login\_as\_barber()

        url = reverse("manage\_barber\_service", kwargs={"service\_id": 2299})

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("does not exist", str(resp.data["detail"]))

    def test\_list\_availabilities(self):

        """

        Only future availabilities and future slots for today are listed.

        """

        now = timezone.now().astimezone(ZoneInfo('Europe/Rome'))

        today = now.date()

        yesterday = today - datetime.timedelta(days=1)

        tomorrow = today + datetime.timedelta(days=1)

        past\_hour = (now - datetime.timedelta(hours=1)).time().strftime('%H:%M')

        future\_hour = (now + datetime.timedelta(hours=1)).time().strftime('%H:%M')

        # Correct assignment

        past\_hour = (now - datetime.timedelta(hours=1)).time().strftime('%H:%M')

        future\_hour = (now + datetime.timedelta(hours=1)).time().strftime('%H:%M')

        # For tomorrow (should show up with both slots)

        availability\_tomorrow = Availability.objects.create(

            barber=self.barber\_user,

            date=tomorrow,

            slots=[past\_hour, future\_hour]

        )

        # For yesterday (should not show up)

        availability\_yesterday = Availability.objects.create(

            barber=self.barber\_user,

            date=yesterday,

            slots=[past\_hour, future\_hour]

        )

        # For today (only future\_hour should be visible)

        availability\_today = Availability.objects.create(

            barber=self.barber\_user,

            date=today,

            slots=[past\_hour, future\_hour]

        )

        self.login\_as\_barber()

        response = self.client.get(self.availabilities\_url)

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        # Get returned availabilities

        availabilities = response.data["availabilities"]

        # Yesterday shouldn't show up

        self.assertNotIn(availability\_yesterday.to\_dict(), availabilities)

        # For today: only the future slot should show

        expected\_today\_dict = availability\_today.to\_dict()

        expected\_today\_dict['slots'] = [future\_hour]

        self.assertIn(expected\_today\_dict, availabilities)

        # Both slots should show for tomorrow

        self.assertIn(availability\_tomorrow.to\_dict(), availabilities)

    def test\_list\_availabilities\_only\_barber(self):

        """

        Only barbers can list availabilities.

        """

        resp = self.client.get(self.availabilities\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_401\_UNAUTHORIZED)

        self.login\_as\_client()

        resp2 = self.client.get(self.availabilities\_url)

        self.assertEqual(resp2.status\_code, status.HTTP\_403\_FORBIDDEN)

    def test\_list\_appointments(self):

        """

        Barber can list all their appointments (ongoing only).

        """

        appointment\_1 = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            date=datetime.date.today(),

            slot=datetime.time(14, 0),

            status=AppointmentStatus.ONGOING.value

        )

        client = Client.objects.create\_user(username="c2", password="foo", email="c2@e.com", is\_active=True)

        appointment\_2 = Appointment.objects.create(

            client=client,

            barber=self.barber\_user,

            date=datetime.date.today(),

            slot=datetime.time(15, 0),

            status=AppointmentStatus.COMPLETED.value

        )

        self.login\_as\_barber()

        resp = self.client.get(self.appointments\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_200\_OK)

        appointments = resp.data["appointments"]

        self.assertIn(appointment\_1.to\_dict(), appointments)

        self.assertIn(appointment\_2.to\_dict(), appointments)

    def test\_list\_reviews(self):

        """

        Barber can list all reviews received.

        """

        client = Client.objects.create\_user(username="c3", password="x", email="c3@e.com", is\_active=True)

        app = Appointment.objects.create(

            client=client,

            barber=self.barber\_user,

            date=datetime.date.today(),

            slot=datetime.time(10, 0),

            status=AppointmentStatus.COMPLETED.value)

        rev = Review.objects.create(client=client, barber=self.barber\_user, rating=5, comment="Great!")

        self.login\_as\_barber()

        resp = self.client.get(self.reviews\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_200\_OK)

        # At least one review present with right data

        out\_reviews = resp.data["reviews"]

        self.assertTrue(any(r["id"] == rev.id for r in out\_reviews))

        self.assertEqual(out\_reviews[0]["rating"], 5)

    def test\_list\_reviews\_only\_barber(self):

        """

        Only barbers can list their reviews.

        """

        resp = self.client.get(self.reviews\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_401\_UNAUTHORIZED)

        self.login\_as\_client()

        resp2 = self.client.get(self.reviews\_url)

        self.assertEqual(resp2.status\_code, status.HTTP\_403\_FORBIDDEN)

### Test profilo cliente

La classe **ClientProfileTest** raccoglie tutti gli unit test relativi alle funzionalità del profilo cliente.  
In particolare, vengono verificati i seguenti aspetti:

* la possibilità per il cliente di visualizzare, modificare ed eliminare il proprio profilo,
* la visualizzazione, creazione e cancellazione degli appuntamenti con i barbieri,
* la gestione delle recensioni sui barbieri, incluse creazione, modifica, eliminazione e restrizioni di unicità,
* i controlli sui permessi e sui vincoli di integrità (ad esempio validità del numero di telefono e imposizione di regole sui ruoli e proprietà delle risorse).

I test assicurano che soltanto il cliente autenticato possa accedere alle proprie operazioni, individuando tempestivamente anomalie e violazioni di sicurezza nelle principali funzionalità lato cliente.

class ClientProfileTest(APITestCase):

    """

    Tests for the client profile management and appointment/review endpoints.

    """

    def test\_get\_profile\_success(self):

        """

        Authenticated client can get their full profile including appointments and reviews.

        """

        self.login\_as\_client()

        response = self.client.get(self.profile\_url)

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        profile = response.data["profile"]

        self.assertEqual(profile, self.client\_user.to\_dict())

        self.assertEqual(profile['role'], Roles.CLIENT.value)

    def test\_get\_profile\_requires\_auth\_and\_client\_role(self):

        """

        Getting profile requires client authentication; returns 403 for non-clients.

        """

        # Not authenticated

        response = self.client.get(self.profile\_url)

        self.assertEqual(response.status\_code, status.HTTP\_401\_UNAUTHORIZED)

        # Auth as barber, not client

        self.login\_as\_barber()

        response = self.client.get(self.profile\_url)

        self.assertEqual(response.status\_code, status.HTTP\_403\_FORBIDDEN)

    def test\_update\_profile\_success\_partial(self):

        """

        Client can update part(s) of their profile.

        """

        self.login\_as\_client()

        patch = {"username": "newclientname", "name": "CName", "surname": "CSurname", "phone\_number": "+15715550111"}

        resp = self.client.patch(self.profile\_url, patch, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_200\_OK)

        self.assertEqual(resp.data["detail"], "Profile info updated successfully.")

        self.client\_user.refresh\_from\_db()

        self.assertEqual(self.client\_user.username, patch["username"])

        self.assertEqual(self.client\_user.name, patch["name"])

        self.assertEqual(self.client\_user.surname, patch["surname"])

        self.assertEqual(self.client\_user.phone\_number, patch["phone\_number"])

    def test\_update\_profile\_username\_unique\_constraint(self):

        """

        Username cannot be changed to one in use.

        """

        # Make "jane" also a taken username

        self.login\_as\_client()

        response = self.client.patch(self.profile\_url, {"username": "jane"}, format="json")

        self.assertEqual(response.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("taken", str(response.data["detail"]).lower())

    def test\_update\_profile\_requires\_field(self):

        """

        Updating profile with no fields returns validation error.

        """

        self.login\_as\_client()

        resp = self.client.patch(self.profile\_url, {}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("must provide at least one field", str(resp.data["detail"]))

    def test\_update\_profile\_phone\_number\_invalid(self):

        """

        Invalid phone number format is not accepted.

        """

        self.login\_as\_client()

        resp = self.client.patch(self.profile\_url, {"phone\_number": "123abc"}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("phone number must be entered", str(resp.data["phone\_number"][0]).lower())

    def test\_delete\_profile\_success(self):

        """

        Client can delete their profile.

        """

        self.login\_as\_client()

        resp = self.client.delete(self.profile\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_204\_NO\_CONTENT)

        self.assertFalse(Client.objects.filter(pk=self.client\_user.pk).exists())

    def test\_list\_appointments(self):

        """

        Client can list all their appointments.

        """

        s1 = Service.objects.create(barber=self.barber\_user, name="Hair", price=10)

        app\_1 = Appointment.objects.create(

            client=self.client\_user, barber=self.barber\_user,

            date=datetime.date.today(), slot=datetime.time(10, 0),

            status=AppointmentStatus.ONGOING.value,

        )

        self.add\_services(app\_1, [s1])

        app\_2 = Appointment.objects.create(

            client=self.client\_user, barber=self.barber\_user,

            date=datetime.date.today() + datetime.timedelta(days=1), slot=datetime.time(11, 0),

            status=AppointmentStatus.COMPLETED.value,

        )

        self.add\_services(app\_2, [s1])

        self.login\_as\_client()

        resp = self.client.get(self.appointments\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_200\_OK)

        self.assertEqual(len(resp.data["appointments"]), 2)

        self.assertTrue(any(app["id"] == app\_1.id for app in resp.data["appointments"]))

    def test\_list\_appointments\_requires\_auth\_client(self):

        """

        Only authenticated clients may list appointments.

        """

        resp = self.client.get(self.appointments\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_401\_UNAUTHORIZED)

        self.login\_as\_barber()

        resp2 = self.client.get(self.appointments\_url)

        self.assertEqual(resp2.status\_code, status.HTTP\_403\_FORBIDDEN)

    def test\_create\_appointment\_success(self):

        """

        Client can create a valid appointment with a barber and services.

        """

        service\_1 = Service.objects.create(

            barber=self.barber\_user,

            name="Haircut",

            price=10

        )

        service\_2 = Service.objects.create(

            barber=self.barber\_user,

            name="Shave",

            price=15

        )

        Availability.objects.create(

            barber=self.barber\_user,

            date=datetime.date.today(),

            slots=["09:00", "11:00"]

        )

        self.login\_as\_client()

        url = reverse("create\_client\_appointment", kwargs={"barber\_id": self.barber\_user.id})

        data = {

            "date": datetime.date.today(),

            "slot": "09:00",

            "services": [service\_1.id, service\_2.id],

        }

        response = self.client.post(url, data, format="json")

        self.assertEqual(response.status\_code, status.HTTP\_201\_CREATED)

        self.assertIn("Appointment added successfully", response.data["detail"])

        appointment = Appointment.objects.get(client=self.client\_user)

        self.assertEqual(appointment.barber, self.barber\_user)

        self.assertEqual(str(appointment.date), str(datetime.date.today()))

        self.assertEqual([x.id for x in appointment.services.all()], [service\_1.id, service\_2.id])

    def test\_create\_appointment\_requires\_barber\_exists(self):

        """

        Cannot create appointment with non-existent, inactive, or wrong barber.

        """

        self.login\_as\_client()

        url = reverse("create\_client\_appointment", kwargs={"barber\_id": 9999})

        data = {

            "date": datetime.date.today(),

            "slot": "09:00",

            "services": [],

        }

        resp = self.client.post(url, data, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("does not exist", str(resp.data["detail"]).lower())

    def test\_create\_appointment\_rejects\_conflicts(self):

        """

        Rejects double bookings for same client/date or same barber/slot or ongoing.

        """

        service\_1 = Service.objects.create(barber=self.barber\_user, name="Cut", price=9)

        today = datetime.date.today()

        slot = datetime.time(9, 0)

        # Set available

        Availability.objects.create(barber=self.barber\_user, date=today, slots=["09:00"])

        # Prior appointment (ONGOING for client)

        appointment = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            date=today,

            slot=slot,

            status=AppointmentStatus.ONGOING.value

        )

        AppointmentService.objects.create(

            appointment=appointment,

            name=service\_1.name,

            price=service\_1.price,

            original\_service=service\_1

        )

        self.login\_as\_client()

        url = reverse("create\_client\_appointment", kwargs={"barber\_id": self.barber\_user.id})

        data = {

            "date": today,

            "slot": "09:00",

            "services": [service\_1.id],

        }

        # Should fail: client already has ONGOING appt

        resp = self.client.post(url, data, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("already has an ONGOING appointment".lower(), str(resp.data["detail"]).lower())

    def test\_create\_appointment\_rejects\_wrong\_service(self):

        """

        Cannot create appointment with a service not provided by the barber.

        """

        their\_barber = Barber.objects.create\_user(

            username="otherbarber", email="x@x.com", password="p", name="t", surname="t",

            is\_active=True

        )

        s\_other = Service.objects.create(barber=their\_barber, name="Wax", price=10)

        s\_our = Service.objects.create(barber=self.barber\_user, name="Buzz", price=9)

        dt = datetime.date.today()

        Availability.objects.create(barber=self.barber\_user, date=dt, slots=["13:00"])

        self.login\_as\_client()

        url = reverse("create\_client\_appointment", kwargs={"barber\_id": self.barber\_user.id})

        data = {

            "date": dt,

            "slot": "13:00",

            "services": [s\_other.id],  # not from this barber!

        }

        resp = self.client.post(url, data, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("does not exist", str(resp.data["detail"]))

    def test\_create\_appointment\_rejects\_if\_unavailable(self):

        """

        Cannot create appointment if barber not available (no availability or slot missing).

        """

        s1 = Service.objects.create(barber=self.barber\_user, name="Mass", price=8)

        dt = datetime.date.today()

        # No Availability for barber

        self.login\_as\_client()

        url = reverse("create\_client\_appointment", kwargs={"barber\_id": self.barber\_user.id})

        data = {

            "date": dt,

            "slot": "08:00",  # not available

            "services": [s1.id],

        }

        resp = self.client.post(url, data, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("not available", str(resp.data["detail"]).lower())

        Availability.objects.create(barber=self.barber\_user, date=dt, slots=["13:00"])

        # Now date exists, but slot is missing

        data["slot"] = "08:00"

        resp2 = self.client.post(url, data, format="json")

        self.assertEqual(resp2.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("not available", str(resp2.data["detail"]).lower())

    def test\_cancel\_appointment\_success(self):

        """

        Client can cancel their ONGOING appointment.

        """

        s1 = Service.objects.create(barber=self.barber\_user, name="B", price=7)

        appt = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            date=datetime.date.today(),

            slot=datetime.time(8,0),

            status=AppointmentStatus.ONGOING.value

        )

        self.add\_services(appt, [s1])

        self.login\_as\_client()

        url = reverse("delete\_client\_appointment", kwargs={"appointment\_id": appt.id})

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_200\_OK)

        appt.refresh\_from\_db()

        self.assertEqual(appt.status, AppointmentStatus.CANCELLED.value)

    def test\_cancel\_appointment\_not\_found\_or\_not\_ongoing(self):

        """

        Only ONGOING appts can be cancelled; not found or wrong client returns error.

        """

        s1 = Service.objects.create(barber=self.barber\_user, name="V", price=5)

        appt = Appointment.objects.create(

            client=self.client\_user, barber=self.barber\_user,

            date=datetime.date.today(), slot=datetime.time(8,0),

            status=AppointmentStatus.COMPLETED.value

        )

        self.add\_services(appt, [s1])

        self.login\_as\_client()

        # Not ONGOING

        url = reverse("delete\_client\_appointment", kwargs={"appointment\_id": appt.id})

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("Only ONGOING appointments can be cancelled", str(resp.data["detail"]))

        # Not your appointment

        appt2 = Appointment.objects.create(

            client=self.client\_user\_other, barber=self.barber\_user,

            date=datetime.date.today(), slot=datetime.time(9,0),

            status=AppointmentStatus.ONGOING.value

        )

        self.add\_services(appt2, [s1])

        url2 = reverse("delete\_client\_appointment", kwargs={"appointment\_id": appt2.id})

        resp2 = self.client.delete(url2)

        self.assertEqual(resp2.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("does not exist", str(resp2.data["detail"]))

    def test\_list\_reviews(self):

        """

        Client can list all the reviews they've posted.

        """

        # Need at least one completed appointment and review created

        service\_1 = Service.objects.create(barber=self.barber\_user, name="BE", price=6)

        appointment = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            date=datetime.date.today(),

            slot=datetime.time(22,0),

            status=AppointmentStatus.COMPLETED.value

        )

        self.add\_services(appointment, [service\_1])

        review = Review.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            rating=4,

            comment="Nice cut!"

        )

        self.login\_as\_client()

        resp = self.client.get(self.reviews\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_200\_OK)

        self.assertTrue(any(r["id"] == review.id for r in resp.data["reviews"]))

        self.assertEqual(resp.data["reviews"][0]["comment"], "Nice cut!")

    def test\_list\_reviews\_requires\_client\_only(self):

        """

        Only clients can list their reviews.

        """

        resp = self.client.get(self.reviews\_url)

        self.assertEqual(resp.status\_code, status.HTTP\_401\_UNAUTHORIZED)

        self.login\_as\_barber()

        resp2 = self.client.get(self.reviews\_url)

        self.assertEqual(resp2.status\_code, status.HTTP\_403\_FORBIDDEN)

    def test\_create\_review\_success(self):

        """

        Client can create a review on a completed appointment (appointment->barber).

        """

        s1 = Service.objects.create(barber=self.barber\_user, name="Mow", price=10)

        appt = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            date=datetime.date.today(),

            slot=datetime.time(12,0),

            status=AppointmentStatus.COMPLETED.value

        )

        self.add\_services(appt, [s1])

        self.login\_as\_client()

        url = reverse("create\_client\_review", kwargs={"barber\_id": self.barber\_user.id})

        data = {"rating": 5, "comment": "Spectacular"}

        resp = self.client.post(url, data, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_201\_CREATED)

        review = Review.objects.get(client=self.client\_user, barber=self.barber\_user)

        self.assertEqual(review.rating, 5)

        self.assertEqual(review.comment, "Spectacular")

    def test\_create\_review\_fail\_non\_completed\_appointment(self):

        """

        Cannot review a non-completed appointment.

        """

        s1 = Service.objects.create(barber=self.barber\_user, name="Face", price=6)

        appt = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            date=datetime.date.today(),

            slot=datetime.time(15,0),

            status=AppointmentStatus.ONGOING.value

        )

        self.add\_services(appt, [s1])

        self.login\_as\_client()

        url = reverse("create\_client\_review", kwargs={"barber\_id": self.barber\_user.id})

        resp = self.client.post(url, {"rating": 4, "comment": ""}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("ou may only review a barber if you have completed at least one appointment with them.", str(resp.data["detail"]))

    def test\_create\_review\_cannot\_duplicate(self):

        """

        Client can only review a given barber once.

        """

        s1 = Service.objects.create(barber=self.barber\_user, name="Legs", price=6)

        completed = Appointment.objects.create(

            client=self.client\_user, barber=self.barber\_user,

            date=datetime.date.today(), slot=datetime.time(17,0),

            status=AppointmentStatus.COMPLETED.value

        )

        self.add\_services(completed, [s1])

        review = Review.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            rating=3,

            comment="zzz"

        )

        self.login\_as\_client()

        url = reverse("create\_client\_review", kwargs={"barber\_id": self.barber\_user.id})

        resp = self.client.post(url, {"rating": 5, "comment": ""}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("review for the barber", str(resp.data["detail"]))

    def test\_create\_review\_fail\_wrong\_barber(self):

        """

        Can only review your own appointments.

        """

        self.login\_as\_client()

        url = reverse("create\_client\_review", kwargs={"barber\_id": self.barber\_user\_other.id})

        resp = self.client.post(url, {"rating": 2, "comment": ""}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("does not exist", str(resp.data["detail"]))

    def test\_update\_review\_success(self):

        """

        Client can update their own review's rating/comment.

        """

        service\_1 = Service.objects.create(barber=self.barber\_user, name="BU", price=1)

        appointment = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            date=datetime.date.today(),

            slot=datetime.time(20,0),

            status=AppointmentStatus.COMPLETED.value

        )

        self.add\_services(appointment, [service\_1])

        review = Review.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            rating=3,

            comment="Old comment"

        )

        self.login\_as\_client()

        url = reverse("manage\_client\_reviews", kwargs={"review\_id": review.id})

        resp = self.client.patch(url, {"rating": 2, "comment": "Better"}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_200\_OK)

        review.refresh\_from\_db()

        self.assertEqual(review.rating, 2)

        self.assertEqual(review.comment, "Better")

        self.assertIsNotNone(review.edited\_at)

    def test\_update\_review\_requires\_field(self):

        """

        Cannot update a review with no fields specified.

        """

        service\_1 = Service.objects.create(barber=self.barber\_user, name="XY", price=8)

        appointment = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            date=datetime.date.today(),

            slot=datetime.time(18,0),

            status=AppointmentStatus.COMPLETED.value

        )

        self.add\_services(appointment, [service\_1])

        review = Review.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            rating=2,

            comment="hi"

        )

        self.login\_as\_client()

        url = reverse("manage\_client\_reviews", kwargs={"review\_id": review.id})

        response = self.client.patch(url, {}, format="json")

        self.assertEqual(response.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("at least one field", str(response.data["detail"]))

    def test\_update\_review\_not\_found(self):

        """

        Cannot update a review not owned or not found.

        """

        self.login\_as\_client()

        url = reverse("manage\_client\_reviews", kwargs={"review\_id": 9999})

        resp = self.client.patch(url, {"rating": 4}, format="json")

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("does not exist", str(resp.data["detail"]))

    def test\_delete\_review\_success(self):

        """

        Client can delete their review.

        """

        service\_1 = Service.objects.create(barber=self.barber\_user, name="XC", price=4)

        appointment = Appointment.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            date=datetime.date.today(),

            slot=datetime.time(19,0),

            status=AppointmentStatus.COMPLETED.value

        )

        self.add\_services(appointment, [service\_1])

        review = Review.objects.create(

            client=self.client\_user,

            barber=self.barber\_user,

            rating=1,

            comment="x"

        )

        self.login\_as\_client()

        url = reverse("manage\_client\_reviews", kwargs={"review\_id": review.id})

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_204\_NO\_CONTENT)

        self.assertFalse(Review.objects.filter(pk=review.id).exists())

    def test\_delete\_review\_not\_found(self):

        """

        Fails gracefully when deleting a review that does not exist or not owned.

        """

        self.login\_as\_client()

        url = reverse("manage\_client\_reviews", kwargs={"review\_id": 34901})

        resp = self.client.delete(url)

        self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

        self.assertIn("does not exist", str(resp.data["detail"]))

### Test upload immagini

La classe **ImageUploadTest** raccoglie tutti gli unit test relativi all’upload, visualizzazione ed eliminazione dell’immagine profilo per admin, barbieri e clienti.

Vengono verificati:

* il caricamento e la rimozione corretti dell’immagine profilo da parte di ciascun utente,
* la risposta ai casi errati (file non immagine, richiesta senza autenticazione),
* la reale persistenza e cancellazione dei file associati.

I test assicurano che solo l’utente autenticato possa modificare la propria immagine e che vengano gestiti gli errori di input e permessi, garantendo integrità e sicurezza dell’operazione.

class ImageUploadTest(APITestCase):

    """

    Tests for the client profile management and appointment/review endpoints.

    """

    def test\_upload\_profile\_image\_success(self):

        """

        Users can upload and retrieve their own profile image url.

        """

        self.upload\_profile\_image('admin')

        self.upload\_profile\_image('barber')

        self.upload\_profile\_image('client')

    def test\_delete\_profile\_image\_success(self):

        """

        Users can upload and then delete their profile image.

        """

        self.delete\_profile\_image('admin')

        self.delete\_profile\_image('barber')

        self.delete\_profile\_image('client')

    def test\_upload\_profile\_image\_bad\_request(self):

        """

        Users can't upload a non-image.

        """

        self.login\_as("admin")

        with tempfile.NamedTemporaryFile(suffix='.txt') as fake\_file:

            fake\_file.write(b'not an image')

            fake\_file.seek(0)

            payload = {'profile\_image': fake\_file}

            resp = self.client.post(self.manage\_image\_url, payload, format='multipart')

            self.assertEqual(resp.status\_code, status.HTTP\_400\_BAD\_REQUEST)

    def test\_upload\_requires\_auth(self):

        """

        Uploading a profile image without authentication is rejected.

        """

        # Do not log in

        with tempfile.NamedTemporaryFile(suffix='.jpg') as image\_file:

            img = Image.new('RGB', (10, 10))

            img.save(image\_file, format='JPEG')

            image\_file.seek(0)

            payload = {'profile\_image': image\_file}

            response = self.client.post(self.manage\_image\_url, payload, format='multipart')

            self.assertEqual(response.status\_code, status.HTTP\_401\_UNAUTHORIZED)

    def test\_delete\_requires\_auth(self):

        """

        Deleting a profile image without authentication is rejected.

        """

        response = self.client.delete(self.manage\_image\_url)

        self.assertEqual(response.status\_code, status.HTTP\_401\_UNAUTHORIZED)