

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

**Documentazione di Ingegneria del Software
2024–2025**

Martusciello Giuseppe, Riccardo Vincenzo, Sindoni Giuseppe

Gruppo ID: INGSW2425_009

Indice

1	Glossario	4
2	Introduzione	8
2.1	Struttura del Documento	8
3	Documento dei Requisiti	9
3.1	Requisiti Funzionali	9
3.2	Requisiti Non Funzionali	11
3.3	Target utenti e Personas	12
3.3.1	Cliente Privato	13
3.3.2	Gestore di Agenzia	14
3.3.3	Dipendenti di agenzia	15
3.3.4	Agenti Immobiliari	16
3.4	Casi d'Uso	17
3.4.1	Use Case Diagram	18
3.4.2	Azioni per ciascun attore	19
3.5	Clockburn e Mock-up dell'Interfaccia	21
3.5.1	Figma	21
3.5.2	Inserimento offerta da parte di un cliente	22
3.5.3	Gestione offerte da parte di un agente	28
3.5.4	Eliminazione Annuncio	32
3.5.5	Gestione notifiche	36
3.6	Osservazioni fatte da stakeholder	39
4	Design del Sistema	41
4.1	Architettura del sistema	41
4.2	Distribuzione dei componenti lato server	42
4.3	Tecnologiche Utilizzate	43
4.4	Design pattern adottati	43
4.4.1	Backend: Controller – Service – Repository	43

4.4.2	Frontend: Model – View – ViewModel (MVVM)	44
4.5	Persistenza dei Dati	45
4.6	Diagramma UML	46
4.6.1	Entità principali	47
4.7	Design Interfaccia Utente	50
4.7.1	Home	52
4.7.2	Schermata Offerte	53
4.7.3	Schermata Ricerche	55
4.7.4	Notifiche	56
4.7.5	Schermata Profilo	57
4.8	Diagrammi di sequenza	58
4.8.1	Modifica delle preferenze di notifica	59
4.8.2	Inserimento di un'offerta da parte di un cliente	60
4.8.3	Gestione di un'offerta da parte di un agente	61
4.8.4	Eliminazione di un annuncio da parte di un agente	62
5	Documentazione e artefatti del processo di sviluppo	63
5.1	Versionamento	63
5.1.1	Github	63
5.2	SonarQube	64
5.2.1	Analisi della qualità del codice	65
5.3	Containerizzazione del Backend	67
5.3.1	Dockerfile	67
5.3.2	.dockerignore	67
5.3.3	docker-compose.yml	68
6	Testing e Valutazione dell'Usabilità del Sistema	69
6.1	Testing con Jest	70
6.1.1	Testing del metodo <code>changePassword</code>	71
6.1.2	Metodo <code>createExternalOffer</code>	74
6.1.3	Metodo <code>getLatestOffersByListingId</code>	79
6.1.4	Metodo <code>deleteAgentById</code>	82
6.2	Expert Review / Ispezione Esperta	85
6.2.1	Risultati principali	85
6.3	Esperimento con Utenti Reali	86

6.3.1	Metriche raccolte	86
6.4	Survey Post-Esperimento	87
6.4.1	Sintesi dei risultati	87

1 Glossario

.dockerignore File che specifica quali file e cartelle devono essere ignorati durante la creazione dell'immagine Docker, al fine di ridurre la dimensione dell'immagine e aumentare la sicurezza.

.gitignore File di configurazione usato da Git per ignorare file e cartelle specifiche nel processo di versionamento. Consente di escludere elementi non rilevanti come file temporanei, build locali o credenziali sensibili.

API REST Interfaccia di programmazione che consente la comunicazione tra client e server usando richieste HTTP.

Assert / Assertion Verifica che un risultato ottenuto durante il test sia uguale a quello atteso, tramite comandi come `expect(...).toEqual(...)`.

Backend Parte server-side dell'applicazione che gestisce la logica di business, le chiamate API e l'accesso ai dati.

BadRequestException Eccezione NestJS che segnala un errore nella richiesta dell'utente, ad esempio un input non valido o fuori dai vincoli attesi (es. prezzo negativo).

bcrypt Libreria per l'hashing e la verifica delle password, utilizzata comunemente per garantire la sicurezza delle credenziali. Nei test viene mockata per evitare operazioni costose.

Branch Ramo indipendente di sviluppo all'interno di una repository Git. Ogni collaboratore può lavorare su un branch separato per sviluppare nuove funzionalità senza interferire con il codice principale.

CE (Classe di Equivalenza) Raggruppamento di input equivalenti che si presume diano lo stesso comportamento nel sistema, utile per ridurre il numero di test mantenendo buona copertura.

class-validator Libreria TypeScript utilizzata per definire vincoli di validazione nei DTO (es. email valida, numero maggiore di zero, campo obbligatorio).

Clockburn Nome della schermata che mostra i mock-up relativi al flusso di interazione per l'inserimento e la gestione delle offerte da parte dell'utente. Rappresenta una fase centrale nell'esperienza utente e viene usata per visualizzare gli step chiave del processo.

Code Smell Difetto nel codice sorgente che non compromette direttamente il funzionamento, ma ne riduce la manutenibilità o la leggibilità.

Commit Operazione con cui si salva una modifica nel repository Git. È accompagnata da un messaggio descrittivo che documenta il cambiamento effettuato.

Container Istanza isolata di un’immagine Docker in esecuzione. Contiene l’applicazione e tutte le sue dipendenze.

Controller Componente backend che gestisce le richieste HTTP e coordina le risposte con il livello Service.

Database Sistema di memorizzazione persistente dei dati. In questo progetto si utilizza PostgreSQL.

Docker Piattaforma che consente di eseguire applicazioni all’interno di container isolati e portabili. Permette di replicare facilmente l’ambiente di produzione in fase di sviluppo.

Docker Compose Strumento per la definizione e l’esecuzione di applicazioni multi-container tramite file YAML.

Dockerfile File di configurazione che contiene le istruzioni necessarie per costruire un’immagine Docker personalizzata. Include comandi come COPY, RUN, EXPOSE, e CMD.

DTO (Data Transfer Object) Oggetto usato per trasferire dati tra client e server in modo strutturato e sicuro.

ENV File File di testo contenente variabili d’ambiente, tipicamente denominato .env. Viene caricato automaticamente in fase di avvio dal servizio tramite Docker Compose.

Extension In un diagramma dei casi d’uso o in un flusso di interazione, un’estensione rappresenta un percorso alternativo o aggiuntivo che si attiva solo al verificarsi di condizioni particolari (es. errore, conferma facoltativa, eccezioni).

Feedback Risposta del sistema all’utente in seguito a un’azione. Può essere visiva, sonora o testuale, e serve a guidare o confermare l’interazione.

Figma Strumento di progettazione grafica collaborativa utilizzato per creare interfacce utente e mock-up. Permette di simulare interazioni e flussi attraverso collegamenti tra i frame e componenti interattivi.

Frontend Interfaccia utente dell’applicazione, sviluppata in Kotlin per Android usando Jetpack Compose, che consente l’interazione dell’utente con il sistema.

Geoapify Servizio di geolocalizzazione utilizzato per effettuare ricerche immobiliari basate su coordinate o indirizzi.

Git Sistema di controllo versione distribuito, utilizzato per gestire il codice sorgente in modo collaborativo e tracciabile.

GitHub Piattaforma web per l'hosting di repository Git. Permette la gestione condivisa del codice, issue, pull request e revisione collaborativa.

Google OAuth Sistema di autenticazione federata che consente il login degli utenti tramite il loro account Google.

Hotspot di sicurezza (Security Hotspot) Punto critico nel codice che potrebbe introdurre vulnerabilità. Deve essere verificato manualmente per escludere rischi.

Immagine Docker Snapshot immutabile di un'applicazione e del suo ambiente di esecuzione. È generata a partire da un **Dockerfile**.

JWT (JSON Web Token) Metodo di autenticazione usato per verificare l'identità dell'utente in modo sicuro tramite token.

Listing Termine usato per indicare un annuncio immobiliare pubblicato nel sistema.

Main (branch) Branch principale della repository. Deve sempre contenere codice stabile, testato e pronto per la produzione.

Mock-up Simulazione grafica dell'interfaccia utente. Aiuta a visualizzare il layout e i flussi prima dello sviluppo effettivo.

MVVM (Model-View-ViewModel) Pattern architetturale usato nel frontend per separare la logica di presentazione dalla logica applicativa.

npm (Node Package Manager) Gestore di pacchetti per ambienti JavaScript/TypeScript. Viene usato per installare dipendenze e avviare script di build o esecuzione.

N-WECT Tecnica di testing (N-Way Weak Equivalence Class Testing) che prevede l'analisi delle combinazioni di classi di equivalenza deboli per individuare un set minimo di test significativi.

Package.json File di configurazione di un progetto Node.js. Contiene le dipendenze e gli script utili per compilare o eseguire l'applicazione.

PostgreSQL Sistema di gestione di database relazionali open source usato per la persistenza dei dati.

Prototipo Modello parziale o semplificato del sistema per testare funzionalità e interazioni prima dello sviluppo finale.

Pull Request Richiesta di revisione e fusione di un branch secondario nel branch principale. Consente la collaborazione e la validazione incrociata del codice.

Push Operazione che sincronizza il contenuto di un branch locale con il corrispondente branch remoto su GitHub.

Quality Gate Meccanismo di valutazione automatica su SonarQube che determina se il codice soddisfa gli standard minimi di qualità.

Repository Strato software che incapsula l'accesso al database, separando la logica di persistenza da quella di business.

Research Ricerca salvata da un cliente, basata su filtri e parametri geografici, usata per ricevere notifiche automatiche.

Script npm Comando definito nel file `package.json` per automatizzare operazioni di compilazione, avvio, test o sviluppo.

Service Componente backend che contiene la logica di business e coordina le operazioni sui dati tramite i repository.

SonarQube Piattaforma per l'analisi statica del codice, utile per rilevare bug, vulnerabilità, code smell, duplicazioni e problemi di manutenibilità.

Stakeholder Soggetto (interno o esterno) che ha un interesse diretto o indiretto nel progetto.

Step Fase o passaggio specifico all'interno di un processo, come una sequenza all'interno di uno use case o di un flusso utente.

TypeORM ORM (Object Relational Mapper) utilizzato per interagire con il database PostgreSQL tramite oggetti TypeScript.

TypeScript Compiler (tsc) Compilatore per TypeScript che trasforma il codice sorgente in JavaScript eseguibile.

UML Linguaggio di modellazione orientato agli oggetti, usato per descrivere graficamente la struttura e il comportamento di un sistema software.

UnauthorizedException Eccezione NestJS che indica che un utente ha tentato di accedere a una risorsa senza avere le autorizzazioni necessarie.

2 Introduzione

Il presente documento costituisce la documentazione tecnica del progetto sviluppato nell'ambito del corso di **Ingegneria del Software – A.A. 2024/2025**, promosso dal DIETI dell'Università degli Studi di Napoli Federico II. L'obiettivo generale dell'attività è la progettazione, implementazione e validazione di un sistema informatico complesso, articolato in più componenti software, secondo i principi dell'ingegneria del software moderna.

Il progetto affronta lo sviluppo della piattaforma *DietiEstates25*, un sistema per la gestione di annunci immobiliari che consente a clienti e agenzie immobiliari di interagire attraverso funzionalità avanzate.

2.1 Struttura del Documento

La documentazione è strutturata in più sezioni, ciascuna focalizzata su uno specifico aspetto del ciclo di vita del software:

- **Documento dei requisiti:** identificazione delle funzionalità fondamentali del sistema, modellazione UML dei casi d'uso, definizione degli utenti target e dei requisiti non funzionali.
- **Design del sistema:** illustrazione dell'architettura adottata, delle scelte tecnologiche, del design orientato agli oggetti e delle strategie di persistenza dei dati. Sono inclusi diagrammi delle classi, diagrammi di sequenza, schema di persistenza e design delle interfacce utente.
- **Documentazione e Artefatti del processo di sviluppo:** descrizione dell'ambiente di sviluppo, struttura del codice, tecnologie impiegate e utilizzo di strumenti di versionamento e containerizzazione. Sono descritti con precisione GitHub, Docker e SonarQube.
- **Testing e valutazione dell'usabilità:** pianificazione dei test automatici con framework jest, strategie di testing (classi di equivalenza, criteri di copertura) e valutazioni di usabilità tramite ispezioni e test con utenti reali.

3 Documento dei Requisiti

DietiEstates25 è una piattaforma per la gestione di annunci immobiliari. Essa permette a più agenzie di pubblicare annunci, permette agli utenti di esplorare le inserzioni, effettuare ricerche avanzate in base ai propri interessi e proporre eventuali offerte. Il sistema deve essere accessibile tramite applicazione mobile, offrendo un'interfaccia performante, intuitiva e piacevole.

3.1 Requisiti Funzionali

Gestione degli Account e Autenticazione

- Creazione di un account per il gestore dell'agenzia con credenziali predefinite.
- Possibilità di modificare la password iniziale dopo il primo accesso.
- Creazione di account di amministrazione di supporto da parte del gestore di un'agenzia.
- Creazione di account per agenti da parte del gestore dell'agenzia.
- Registrazione utenti tramite email e password.
- Autenticazione tramite provider esterni (Google).
- Login e salvataggio sicuro delle credenziali.

Inserimento e Gestione degli Immobili

- Gli agenti possono caricare nuovi immobili con: foto, descrizione, prezzo, dimensioni, indirizzo, piano, numero di stanze, classe energetica, servizi accessori.
- Ogni immobile deve essere categorizzato come "vendita" o "affitto".
- Ogni annuncio è associato ad una posizione geografica precisa.

Ricerca Avanzata di Immobili

- Possibilità di filtrare per: tipologia, fascia di prezzo, numero di stanze, classe energetica, posizione geografica ed altro.
- Supporto minimo al filtro per comune/città.
- Ricerca per raggio da un punto su mappa interattiva.

Modifica e Cancellazione Annunci

- Gli annunci possono essere modificati o cancellati solo dall'agente che li ha creati o da un amministratore.

Gestione Offerte

- Gli utenti possono inviare offerte per immobili.
- Gli agenti possono accettare, rifiutare o proporre una controfferta.
- Deve esserci un tracking delle offerte fatte e ricevute visibile sia dagli agenti che dai clienti.
- Gli agenti possono inserire offerte ricevute al di fuori del sistema.

Informazioni Contestuali sulla Posizione

- Il sistema deve verificare la presenza di scuole, parchi pubblici, fermate del trasporto pubblico tramite API esterne.
- L'inserzione deve riportare automaticamente indicatori come: "Vicino a scuole", "Vicino a parchi", ecc.

Storico delle Ricerche

- Il sistema tiene traccia delle ricerche effettuate da ciascun utente.
- Possibilità di rieseguire ricerche precedenti per trovare nuove inserzioni coerenti.

Gestione Notifiche

- Notifiche per: nuove inserzioni, messaggi promozionali o relative ad offerte.
- Gli utenti possono attivare/disattivare le diverse categorie di notifica.

3.2 Requisiti Non Funzionali

- **Sicurezza:** il sistema deve proteggere le informazioni sensibili degli utenti, in particolare credenziali e dati personali. Le password sono memorizzate in forma cifrata, e l'autenticazione può avvenire anche tramite provider esterni sicuri (es. Google).
- **Usabilità:** l'interfaccia dell'applicazione mobile deve essere semplice, coerente e facilmente utilizzabile anche da utenti con bassa familiarità digitale.
- **Accessibilità:** l'interfaccia deve essere ottimizzata per un'ampia gamma di dispositivi mobili (smartphone, tablet) e risoluzioni, garantendo una corretta visualizzazione e interazione ad un adeguato contrasto visivo.
- **Prestazioni:** il sistema deve garantire tempi di risposta rapidi per le operazioni di ricerca, caricamento e visualizzazione dei contenuti.
- **Scalabilità:** l'architettura deve essere in grado di sostenere un aumento progressivo del numero di utenti e annunci, senza degradare le prestazioni.
- **Modularità:** separazione chiara tra front-end e back-end tramite API REST.
- **Interoperabilità:** integrazione con servizi esterni (es. Geoapify, Google Auth).
- **Affidabilità:** il sistema deve garantire la tracciabilità e la persistenza di offerte, ricerche salvate e notifiche, anche in presenza di errori temporanei.

3.3 Target utenti e Personas

Per progettare in modo efficace la piattaforma **DietiEstates25**, è stato condotto uno studio preliminare volto a identificare le principali categorie di utenti che ne usufruiranno. L'obiettivo di tale analisi è comprendere in profondità i bisogni, le aspettative e i comportamenti degli utenti finali, così da orientare correttamente le scelte progettuali e funzionali.

Le *personas* individuate rappresentano profili tipici che incarnano le caratteristiche, le esigenze e gli obiettivi degli utenti reali. Attraverso queste rappresentazioni, è stato possibile definire in maniera più chiara i requisiti pratici e funzionali, sia generali che specifici, associati a ciascun tipo di utente. Queste personas non costituiscono una rappresentazione esaustiva di tutti gli utenti possibili, ma offrono un valido riferimento per modellare l'esperienza utente e guidare le decisioni progettuali. L'approccio adottato consente inoltre di valutare con maggiore precisione l'impatto delle funzionalità implementate sulle diverse tipologie di utilizzatori. Le personas sono state definite tenendo conto sia delle esigenze esplicite rilevate durante l'analisi dei requisiti, sia di potenziali criticità emerse in fase di progettazione preliminare. Questo approccio orientato all'utente garantisce che la piattaforma sia accessibile, usabile ed efficace nel soddisfare i diversi scenari d'uso previsti. Nelle sezioni seguenti vengono presentate le categorie principali e i rispettivi bisogni.

3.3.1 Cliente Privato

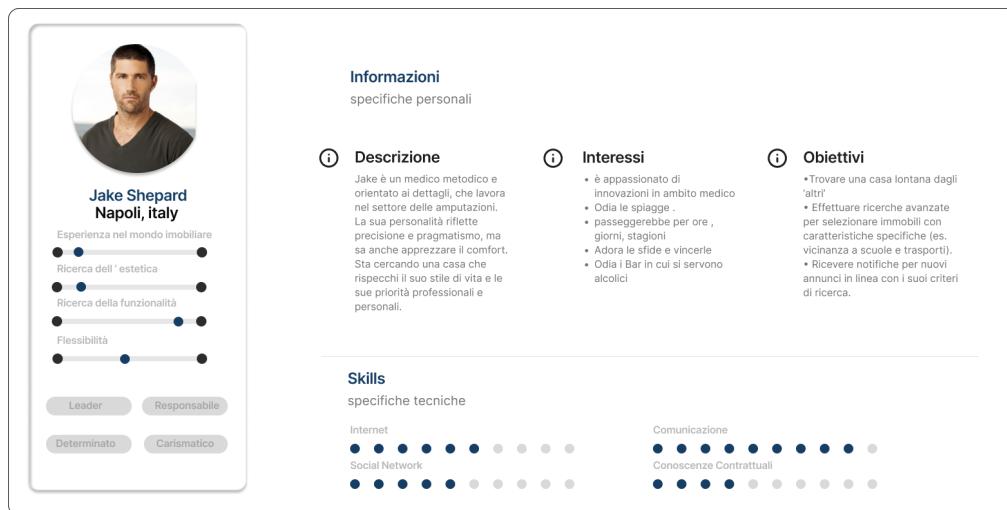


Figure 3.1: Cliente Privato

Il profilo rappresentato in Figura 3.1 descrive una tipologia di utente privato interessato alla ricerca di immobili residenziali in modo semplice, efficiente e personalizzato. Questi utenti hanno esigenze pratiche legate alla qualità dell'abitazione e alla sua collocazione nel contesto urbano, e richiedono strumenti che facilitino l'esplorazione del catalogo immobiliare.

Di seguito sono riportate le funzionalità più ricercate da questa categoria di utenti:

- Ricerca rapida ed efficace di immobili tramite filtri avanzati e intuitivi.
- Visualizzazione degli annunci su mappa interattiva con foto e dettagli completi.
- Possibilità di inviare offerte direttamente dalla piattaforma in autonomia.
- Ricezione di notifiche personalizzate su nuove inserzioni compatibili con le ricerche salvate.
- Indicazione contestuale della presenza di scuole, parchi o mezzi pubblici nelle vicinanze dell'immobile.
- Accesso semplificato tramite account social (es. Google) per una gestione più comoda.

3.3.2 Gestore di Agenzia

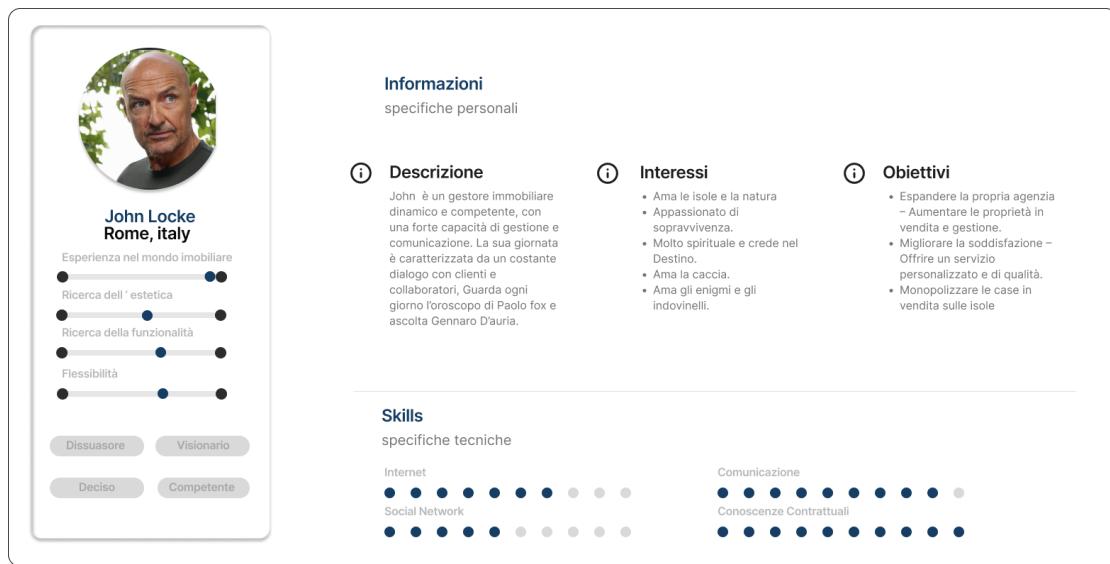


Figure 3.2: Amministratore

Il **Gestore di agenzia** rappresenta il profilo principale per la gestione organizzativa all'interno della piattaforma. È il responsabile che richiede l'inserimento della propria agenzia immobiliare su *DietiEstates25* e riceve, a seguito di approvazione, le credenziali di accesso per la gestione operativa del team.

Una volta autenticato, ha la possibilità di creare e amministrare gli account dei propri dipendenti, inclusi agenti immobiliari e profili di supporto (es. segreteria, assistenti). Il gestore ha inoltre accesso a una visione aggregata dell'attività dell'agenzia.

Di seguito sono riportate le funzionalità più rilevanti per questa tipologia di utente:

- Creazione e gestione degli account dei dipendenti dell'agenzia, in particolare agenti immobiliari e personale amministrativo.
- Visualizzare tutte le inserzioni, offerte della propria agenzia.
- Possibilità di intervenire nella modifica o rimozione degli annunci pubblicati dai propri dipendenti.
- Facoltà di inserire nuovi annunci immobiliari direttamente, in caso di necessità o gestione diretta.
- Possibilità di accettare, rifiutare o proporre controfferte, svolgendo le stesse operazioni disponibili per gli agenti.

3.3.3 Dipendenti di agenzia

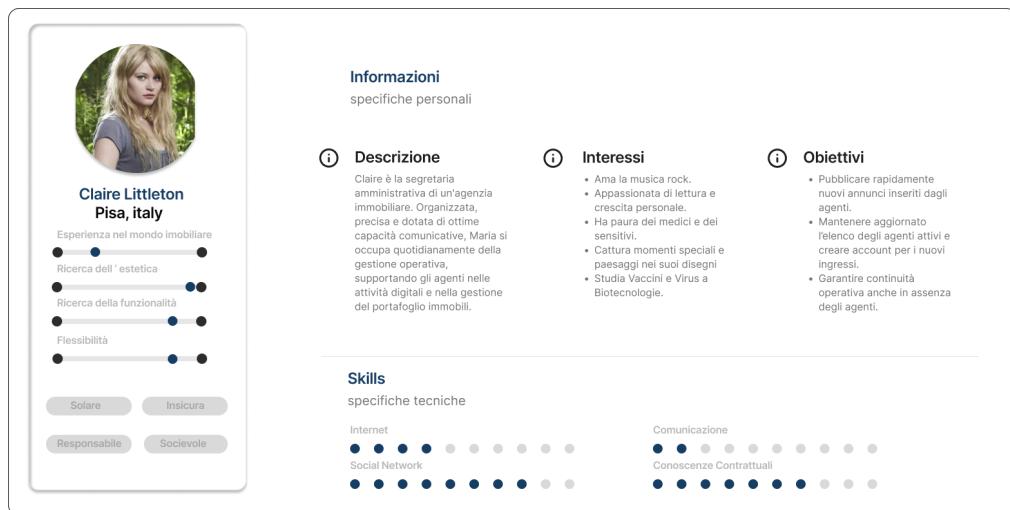


Figure 3.3: Amministrazione di supporto

I dipendenti dell'agenzia con ruolo amministrativo di supporto svolgono una funzione fondamentale nella gestione operativa quotidiana della piattaforma. Si tratta solitamente di figure come segretari o assistenti che, pur non avendo il ruolo di agenti, hanno accesso a numerose funzionalità per agevolare l'attività commerciale dell'agenzia.

Queste figure sono autorizzate ad operare sugli annunci immobiliari, gestire appuntamenti e offerte, ed effettuare operazioni amministrative come la creazione di nuovi account per gli agenti. L'obiettivo principale di questi utenti è garantire continuità e fluidità nei processi interni dell'agenzia, anche in assenza diretta del gestore o degli agenti.

Di seguito le funzionalità più ricercate da questa categoria di utenti:

- Inserimento, modifica e rimozione di annunci immobiliari per conto dell'agenzia.
- Ricezione e gestione delle offerte: accettazione, rifiuto o proposta di controfferte.
- Possibilità di inserire nel sistema offerte ricevute tramite canali esterni (es. telefono, email).
- Creazione di nuovi account per agenti immobiliari.
- Collaborazione diretta con gli agenti per supportare il processo di vendita o locazione.

3.3.4 Agenti Immobiliari

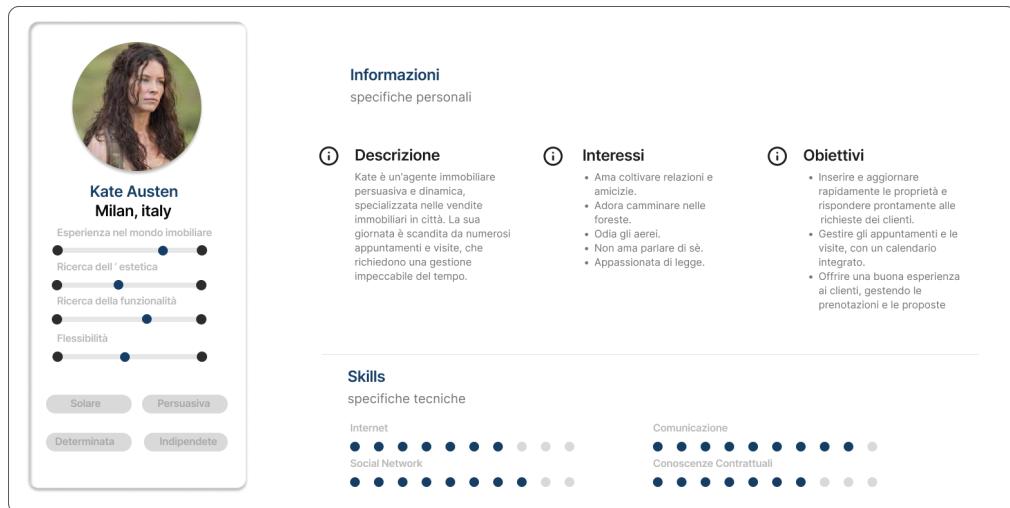


Figure 3.4: Agente Immobiliare

Gli agenti rappresentano un ruolo operativo fondamentale all'interno della piattaforma *DietiEstates25*. Hanno la responsabilità di inserire e gestire gli immobili, e le loro relative offerte. Le funzionalità pensate per questa categoria mirano a rendere il lavoro più efficiente, tracciabile e orientato alla qualità del servizio.

Di seguito sono elencate le principali esigenze funzionali espresse da questa tipologia di utenti:

- Inserimento rapido di nuovi annunci immobiliari, con compilazione guidata di tutte le informazioni richieste (foto, descrizione, caratteristiche tecniche).
- Possibilità di modificare o cancellare i propri annunci in maniera semplice ed intuitiva.
- Ricezione di notifiche relative a nuove offerte pervenute.
- Gestione centralizzata delle offerte ricevute: accettazione, rifiuto o invio di controposte.
- Possibilità di registrare anche offerte ricevute offline, mantenendo uno storico completo e aggiornato.

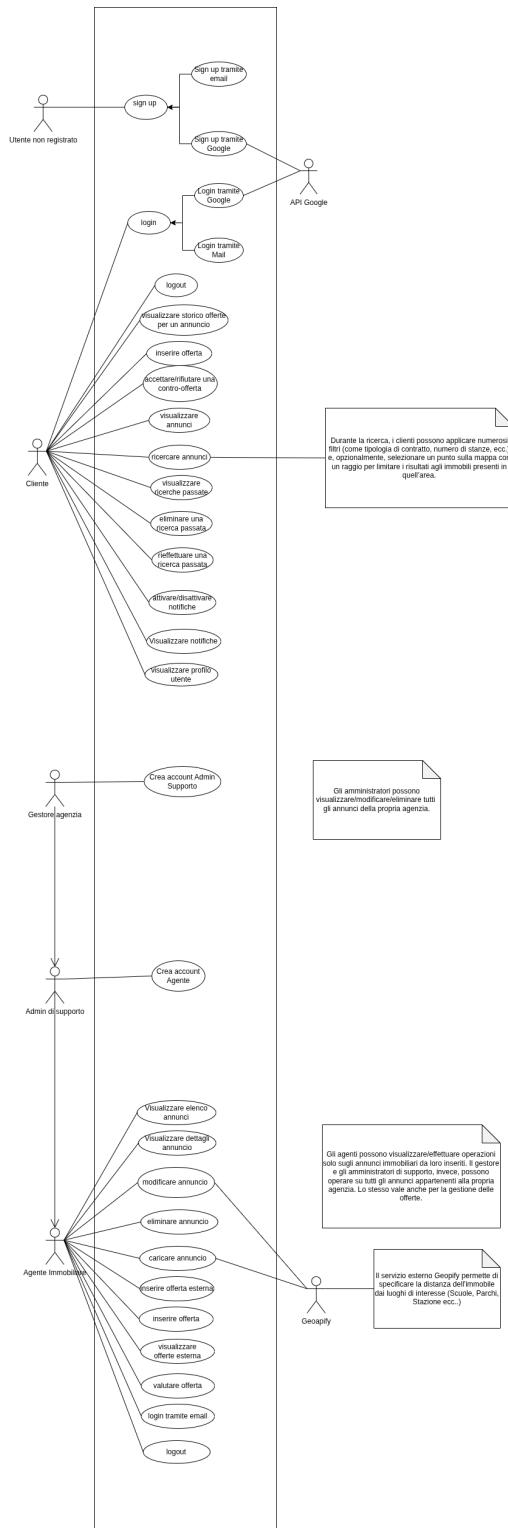
3.4 Casi d'Uso

In questa sezione vengono presentati i principali casi d'uso del sistema *Dieti-Estates25*, modellati secondo la metodologia UML. I casi d'uso descrivono le interazioni tra gli attori principali (utenti, agenti immobiliari e amministratori) e il sistema, evidenziando le funzionalità offerte e i flussi di comportamento attesi.

L'analisi dei casi d'uso costituisce un passaggio fondamentale per comprendere i requisiti funzionali, facilitare la progettazione dei componenti software e garantire una corrispondenza chiara tra le aspettative degli utenti e le funzionalità implementate.

Ogni caso d'uso è corredata da una descrizione testuale e da un diagramma che ne illustra visivamente gli attori coinvolti e le interazioni previste.

3.4.1 Use Case Diagram



3.4.2 Azioni per ciascun attore

Di seguito sono elencate le funzionalità accessibili ai diversi attori del sistema *DietiEstates25*, così come rappresentate nel diagramma dei casi d'uso.

Utente non registrato

- Sign up tramite email
- Sign up tramite Google
- Login tramite Google

Cliente

- Logout
- Visualizzare lo storico delle offerte per un annuncio
- Presentare un'offerta
- Accettare/rifiutare una contro-offerta
- Visualizzare annunci
- Ricercare annunci con filtri
- Visualizzare ricerche passate
- Eliminare una ricerca passata
- Effettuare una ricerca passata
- Attivare/disattivare notifiche
- Visualizzare notifiche
- Visualizzare il profilo utente

Gestore dell'agenzia

- Creare account di amministrazione o supporto

Admin di supporto

- Creare account per agenti immobiliari

Agente immobiliare

- Login tramite email
- Logout
- Cambiare password (obbligatorio al primo accesso)
- Visualizzare annunci propri
- Caricare nuovi annunci
- Modificare un annuncio
- Eliminare un annuncio
- Inserire offerte ricevute esternamente
- Valutare offerte ricevute
- Inserire una contro-offerta

Servizi esterni (Google e Geoapify)

- Autenticazione Google tramite OAuth2 (sign up/login)
- Fornitura di dati geolocalizzati (es. vicinanza a scuole, parchi, trasporti)

3.5 Clockburn e Mock-up dell'Interfaccia

In questa sezione analizzeremo alcuni **casi d'uso rappresentativi**, descrivendo il modo in cui vengono gestiti dal sistema e dall'interazione con l'utente. Per la loro stesura abbiamo adottato un formato ispirato allo stile proposto da **Alistair Cockburn**.

Durante la modellazione iniziale, abbiamo affiancato alla stesura dei casi d'uso la realizzazione di **mock-up cartacei**, che ci hanno aiutato a simulare l'interazione tra utente e sistema. Questo approccio ha facilitato l'identificazione dei **flussi principali** e la definizione delle **funzionalità attese**, permettendo una più chiara comprensione degli scenari d'uso.

Una volta completata questa prima fase esplorativa, è stato avviato un processo di **digitalizzazione dei prototipi** al fine di ottenere una rappresentazione più fedele dell'interfaccia grafica.

3.5.1 Figma

Per la digitalizzazione dei mock-up è stato utilizzato il software **Figma**, che ha permesso di trasformare gli sketch iniziali in schermate **ad alta fedeltà**, utili per visualizzare il comportamento previsto dell'interfaccia. I modelli ottenuti hanno costituito una **guida concreta** durante le fasi di sviluppo, pur non coincidendo perfettamente con il risultato finale.

I mock-up digitali hanno consentito di mantenere una visione coerente del **flusso delle azioni previste per l'utente**, supportando la **validazione dei requisiti** e facilitando la comunicazione tra i membri del team. L'approccio adottato ha quindi favorito un'**integrazione efficace** tra progettazione funzionale e design grafico, rafforzando l'allineamento tra interfaccia e logica applicativa.

3.5.2 Inserimento offerta da parte di un cliente

Use Case		Inserire offerta	
Obiettivo		Inserire un'offerta per uno specifico annuncio.	
Precondizione		Il cliente dovrà essere loggato.	
Condizione Finale di Successo		Il cliente inserisce una nuova offerta per l'annuncio desiderato.	
Condizione Finale di fallimento		Il cliente non inserisce la propria offerta.	
Attore principale		Cliente	
Trigger		Cliente clicca su " Proponi offerta "	
Main Scenario	Step	Cliente	System
	1		Mostra "Dettagli Immobile"
	2	Clicca su "Proponi offerta"	
	3		Mostra "Storico Offerte"
	4	Clicca su "Proponi offerta", inserisce l'offerta e preme invio	
	5		Mostra "Offerta inserita con successo"

Mockup: Inserimento Offerta da parte di un cliente

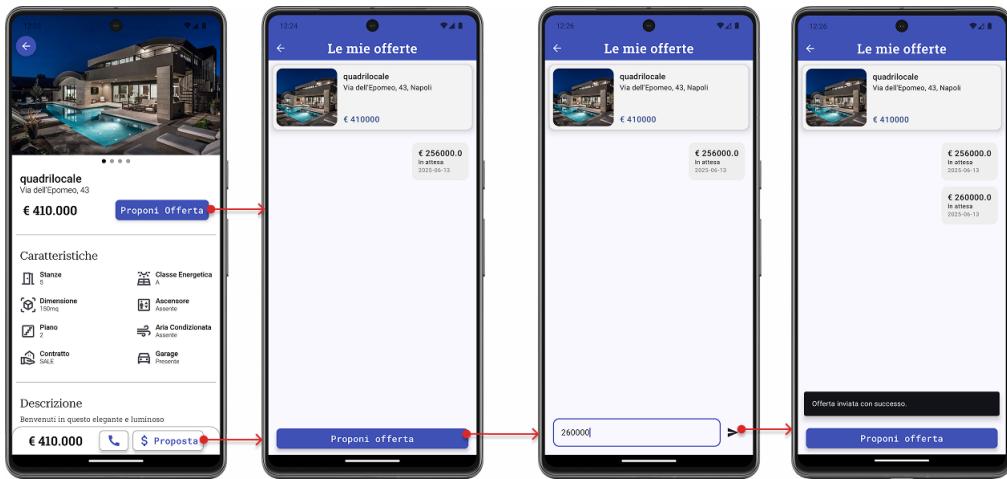


Figure 3.5

Questo caso d'uso si avvia a partire dalla schermata dei dettagli di un annuncio immobiliare.

1. Lo use case viene attivato quando il cliente seleziona il pulsante “*Proponi offerta*”, disponibile nella parte superiore della schermata o nella barra di navigazione inferiore.
2. A seguito della selezione, l’utente viene reindirizzato alla schermata dello **storico delle offerte** relative all’annuncio selezionato. Da qui, può cliccare nuovamente su “*Proponi offerta*” per avviare la procedura di inserimento.
3. Si apre quindi un campo di testo (text area) dedicato all’inserimento del prezzo o dei termini dell’offerta. Una volta completata la compilazione, l’utente conferma cliccando su *Invia*.
4. Al termine dell’invio, viene visualizzato un messaggio di conferma e lo storico delle offerte viene automaticamente aggiornato con la nuova proposta.

CockBurn Offerta Cliente - Extension 1

Extension #1	Clicca annulla	
Step	Cliente	System
4/4a	Annulla l'inserimento	
5/5a		Torna al passo 3 di Main Scenario

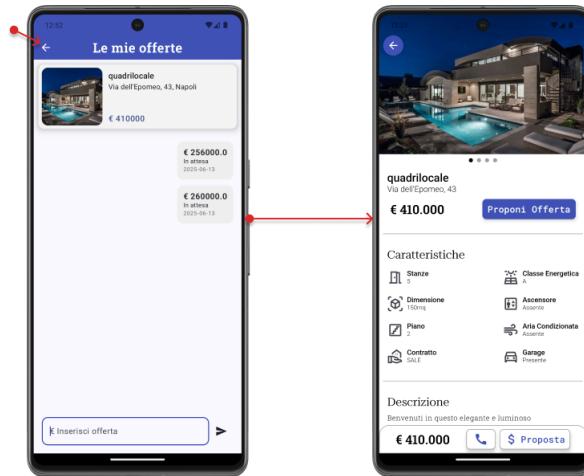


Figure 3.6

- Questa estensione si verifica nel momento in cui, durante l'inserimento dell'offerta, l'utente seleziona il pulsante “Torna indietro” anziché il pulsante “Invia”.
- Il sistema annulla l'inserimento e l'utente viene reindirizzato alla schermata “Dettagli di un Immobile”, corrispondente allo Step 3 dello scenario principale.

CockBurn Offerta Cliente - Extension 2

Extension #2	Inserimento valore troppo basso	
Step	Cliente	System
4/4a	Inserisce un'offerta troppo bassa	
5/5a		Mostra "Offerta troppo bassa" (Torna al passo 3 del Main scenario)

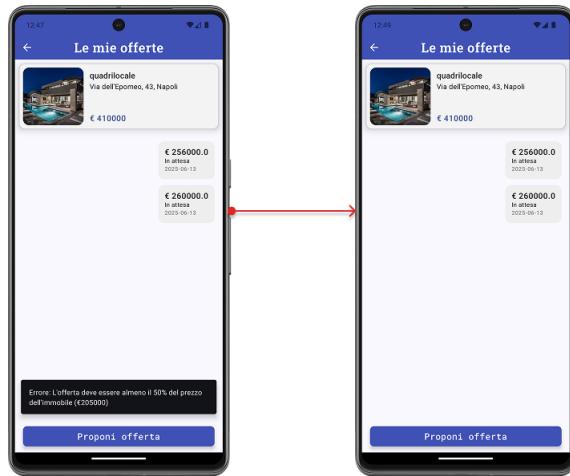


Figure 3.7

1. Questa estensione si attiva nel momento in cui, durante l'inserimento dell'offerta, l'utente propone un importo inferiore al 50% prezzo richiesto per l'annuncio.
2. Il sistema blocca l'invio dell'offerta, annulla l'operazione e visualizza un messaggio di errore che informa l'utente dell'importo non valido.

CockBurn Offerta Cliente - Extension 3

Extension #3	Inserimento valore più alto del prezzo del annuncio	
Step	Cliente	System
4/4a	Inserisce un'offerta troppo alta	
5/5a		Mostra "Importo inserito troppo alto" (Torna al passo 3 del Main scenario)

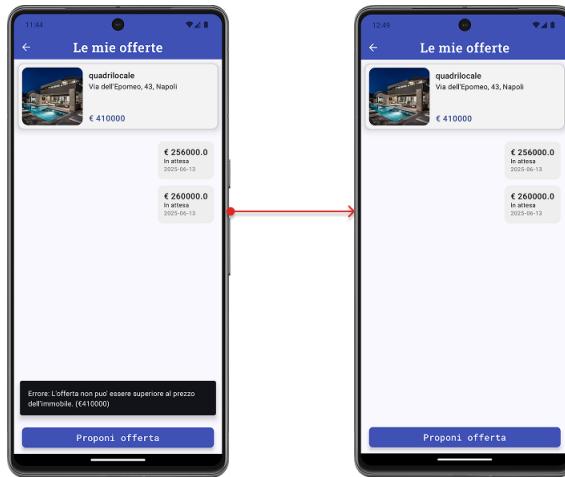


Figure 3.8

1. Questa estensione si attiva nel momento in cui, durante l'inserimento dell'offerta, l'utente propone un importo superiore al prezzo richiesto per l'annuncio.
2. Il sistema blocca l'invio dell'offerta, annulla l'operazione e visualizza un messaggio di errore che informa l'utente dell'importo non valido.

CockBurn Offerta Cliente - Extension 4

Extension #4		Proposta non possibile
Step	Cliente	System
5/5a		Mostra "Impossibile inserire offerta" (Torna al passo 3 del Main scenario)

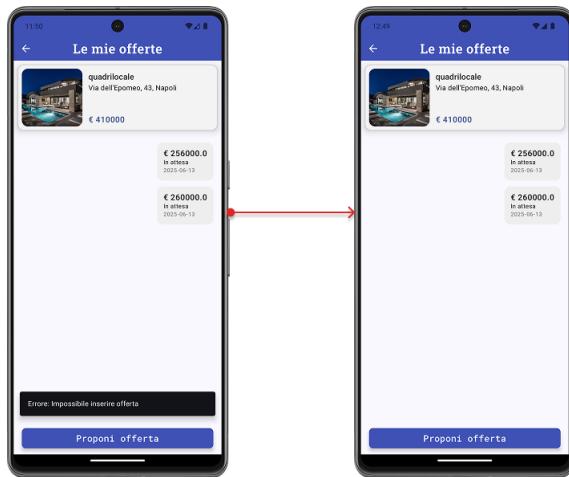


Figure 3.9

- Questa estensione si attiva nel caso in cui si verifichi un errore generico di sistema subito dopo che il cliente ha selezionato il pulsante “*Invia*” per confermare l’offerta.
- Il sistema blocca l’operazione, annulla l’invio dell’offerta e visualizza un messaggio di errore che informa l’utente dell’impossibilità di completare l’azione.

3.5.3 Gestione offerte da parte di un agente

Use Case		Accettare offerta	
Obiettivo		Accettare un'offerta per un annuncio.	
Precondizione		L'agente dovrà essere loggato.	
Condizione Finale di Successo		L'agente accetta un'offerta per l'immobile	
Condizione Finale di fallimento		L'agente non accetta nessuna offerta per l'immobile.	
Attore principale		Agente	
Trigger		Agente clicca su "Offerte"	
Main Scenario	Step	Agente	System
	1		Mostra "Annunci pubblicati"
	2	Selezione l'annuncio desiderato	
	3		Mostra "Elenco Clienti"
	4	Selezione un cliente	
	3		Mostra "Chat"
	4	Clicca su "Accetta" su un'offerta del cliente	
5		Mostra "Offerta inserita con successo"	

Mockup: Gestione offerte da parte di un agente

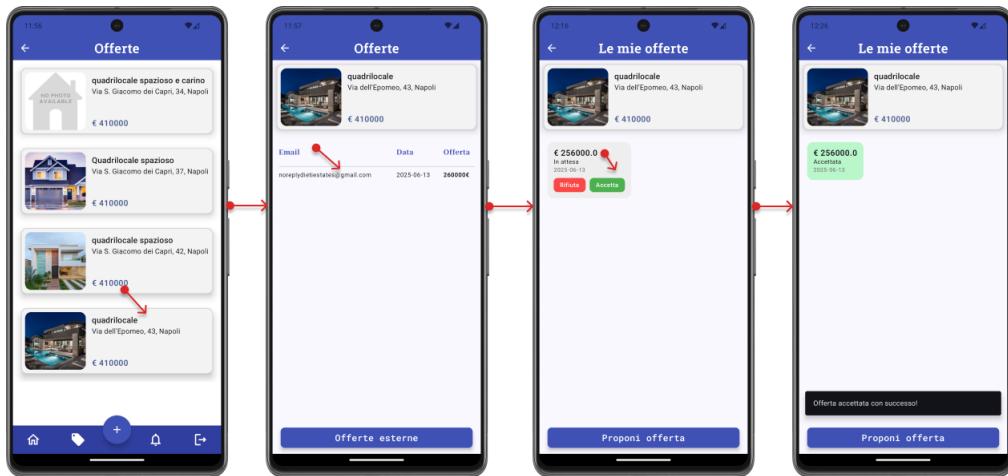


Figure 3.10

Questo caso d'uso si avvia a partire dalla schermata dei propri annunci pubblicati.

1. Il caso d'uso viene attivato quando l'utente seleziona l'immobile per il quale desidera valutare e accettare un'offerta ricevuta.
2. A seguito della selezione, l'utente viene reindirizzato alla schermata dell'**elenco delle offerte** relative all'annuncio selezionato. In questa schermata viene visualizzata una tabella contenente l'elenco dei clienti che hanno presentato un'offerta per quell'immobile, unitamente all'ultima proposta fatta da ciascuno. L'utente seleziona quindi uno dei clienti presenti nell'elenco.
3. Dopo aver selezionato il cliente, l'utente accede alla schermata dello **storico delle offerte**, dove sono visibili tutte le proposte fatte da quel cliente e le eventuali controfferte precedentemente inviate. Da qui, l'utente seleziona il pulsante “Accetta” relativo all'offerta che intende accettare.
4. A seguito della conferma, il sistema visualizza un messaggio di avvenuta accettazione e aggiorna lo storico, modificando lo stato dell'offerta selezionata.

CockBurn Offerta Agente - Extension 1

Extension #1	Clicca annulla	
Step	Agente	System
4/4a	Clicca il tasto "indietro"	
5/5a		Torna al passo 3 di Main Scenario

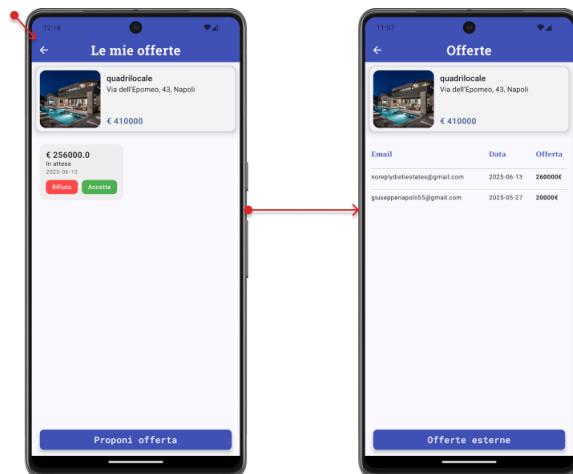


Figure 3.11

1. Questa estensione si verifica nel momento in cui, durante l'accettazione dell'offerta, l'utente seleziona il pulsante “Torna indietro” anziché il pulsante “Accetta”.
2. L'utente viene reindirizzato alla schermata “Elenco delle offerte”, corrispondente allo **Step 3** dello scenario principale.

CockBurn Offerta Agente - Extension 2

Extension #2		Offerta già accettata esistente
Step	Agente	System
5/5a		Mostra "Impossibile accettare offerta" (Torna allo step 3 del Main scenario)

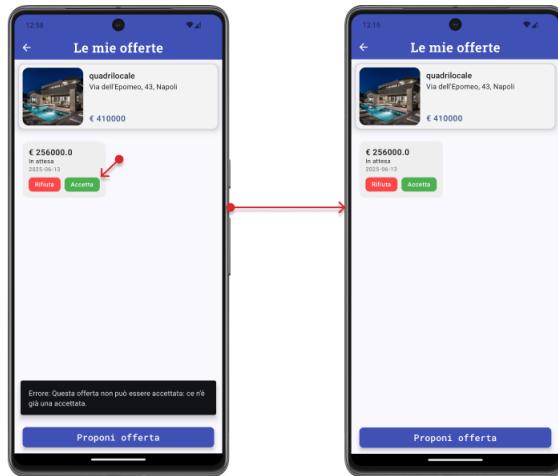


Figure 3.12

- Questa estensione si attiva nel caso in cui si verifichi un errore subito dopo che l'utente ha selezionato il pulsante “Accetta” per accettare un'offerta.
- Il sistema interrompe l'operazione, annulla l'accettazione e visualizza un messaggio di errore che informa l'utente dell'impossibilità di completare l'azione. L'errore è dovuto al fatto che esiste già un'altra offerta accettata per lo stesso immobile.

3.5.4 Eliminazione Annuncio

Use Case		Elimina annuncio	
Obiettivo		Eliminare	
Precondizione		Il cliente deve essere loggato	
Condizione Finale di Successo		L'agente elimina correttamente l'annuncio	
Condizione Finale di fallimento		L'agente non elimina l'annuncio	
Main Scenario	Step	Cliente	System
	1		Mostra "Home Page"
	2	Clicca su ":" sulla card di un immobile	
	3		Mostra "Actions"
	4	Clicca su "Elimina"	
	5		Mostra "Annuncio Eliminato"

Mockup: Eliminazione Annuncio

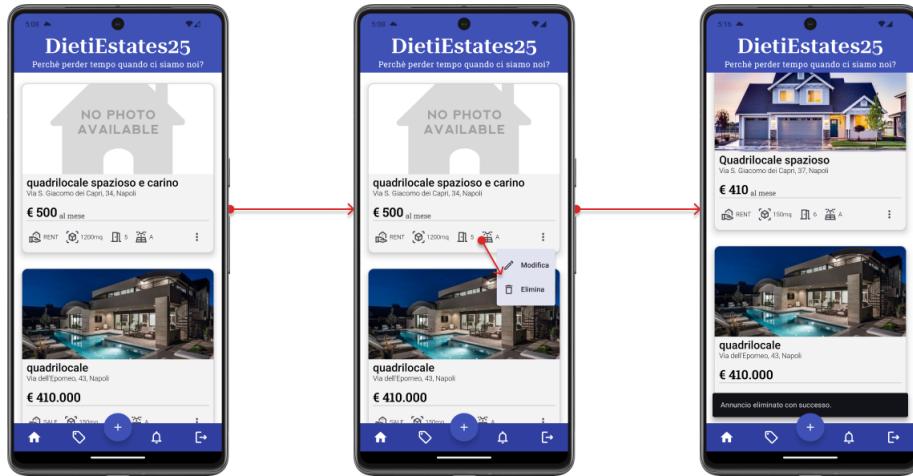


Figure 3.13

Questo caso d'uso si avvia a partire dalla schermata **Home Page**.

1. Il caso d'uso viene attivato quando l'utente seleziona il pulsante “*Azioni*” presente sulla card di uno specifico annuncio pubblicato.
2. Dal menu contestuale, l'utente seleziona l'opzione “*Elimina*” per rimuovere l'annuncio.
3. Il sistema elimina l'annuncio selezionato, visualizza un messaggio di conferma dell'operazione e aggiorna dinamicamente la **Home Page** per riflettere la modifica.

CockBurn Elimina Annuncio - Extension 1

Extension #1	Clicca annulla	
Step	Agente	System
4/4a	Clicca il tasto "indietro"	
5/5a		Mostra "Home Page" (Torna al passo 1 di Main Scenario)

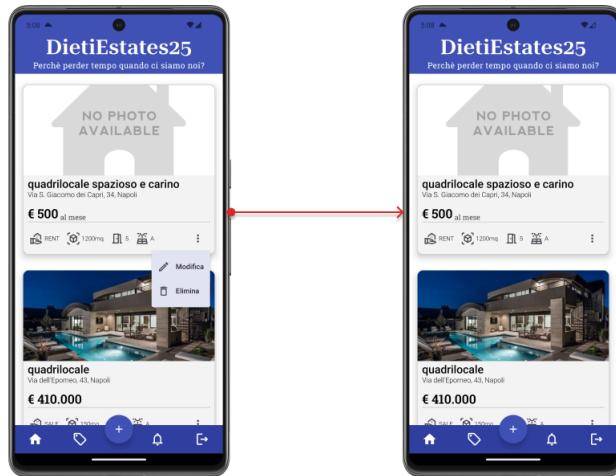


Figure 3.14

- Questa estensione si verifica nel momento in cui, durante la selezione dell'azione sull'annuncio, l'utente sceglie il pulsante “*Torna indietro*” anziché “*Elimina*”.
- Il menu contestuale viene chiuso e il sistema riporta l'utente allo Step 1 dello scenario principale.

CockBurn Offerta Agente - Extension 2

Extension #2	Impossibile eliminare annuncio	
Step	Agente	System
5/5a		Mostra "Impossibile eliminare annuncio" (Torna allo step 1 del Main scenario)

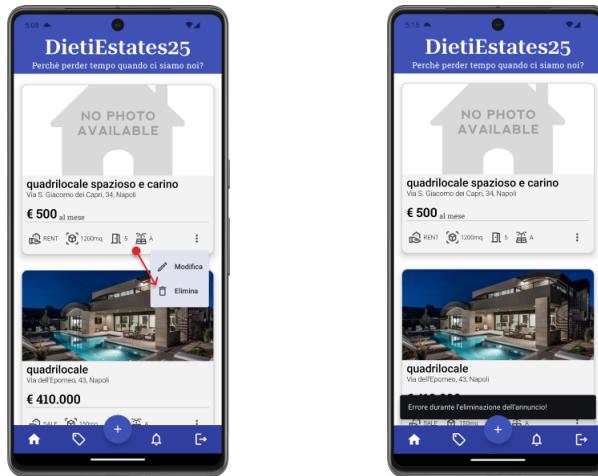


Figure 3.15

1. Questa estensione si attiva nel caso in cui si verifichi un errore generico di sistema subito dopo che l'agente ha selezionato il pulsante “*Elimina*”.
2. Il sistema visualizza un messaggio di errore che informa l'utente dell'impossibilità di completare l'azione.

3.5.5 Gestione notifiche

Use Case		Disattivare notifiche offerte	
Obiettivo		Disattivare notifiche offerte.	
Precondizione		Il cliente dovrà essere loggato.	
Condizione Finale di Successo		Il cliente disattiva le notifiche per le offerte	
Condizione Finale di fallimento		Il cliente non disattiva le notifiche per le offerte	
Attore principale		Cliente	
Trigger		Cliente clicca su "Notifiche offerte"	
Main Scenario	Step	Cliente	System
	1	Clicca su "Profilo Utente"	
	2		Mostra "Profilo Utente"
	3	Clicca su "Notifiche offerte"	
	4		Modifica il toggle

Mockup: Gestione notifiche

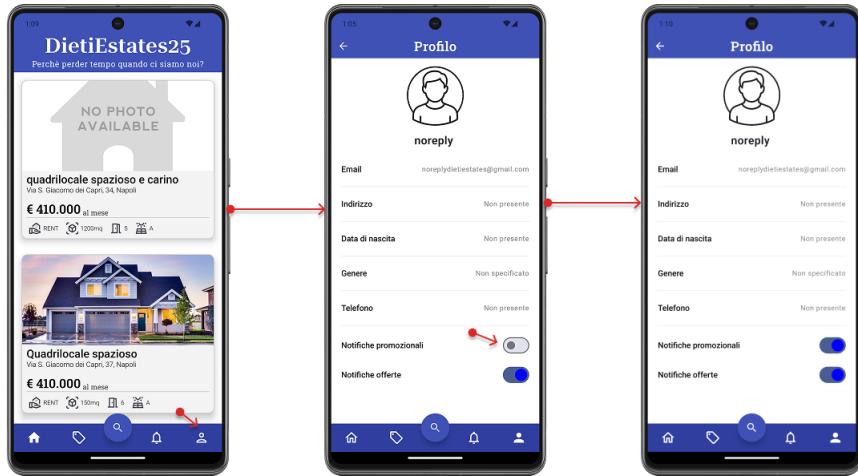


Figure 3.16

Questo caso d'uso si avvia a partire dalla schermata **Home Page**.

1. Il caso d'uso viene attivato quando il cliente seleziona il pulsante “*Profilo utente*”, situato nella parte inferiore della schermata principale.
2. Viene aperta la schermata del profilo utente, nella quale sono visualizzate le informazioni personali dell'utente, insieme a una serie di interruttori (toggle) che rappresentano le categorie di notifiche attualmente attive.
3. Quando l'utente interagisce con uno dei toggle, il sistema aggiorna dinamicamente la preferenza di ricezione per la specifica categoria di notifica selezionata.

CockBurn Elimina Annuncio - Extension 1

Extension #1	Impossibile disattivare le notifiche	
Step	Utente	System
4/4a		Mostra "Errore aggiornamento preferenza" Torna al passo 2 del Main scenario

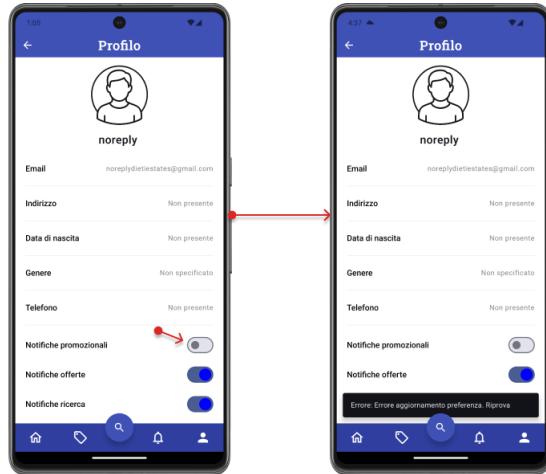


Figure 3.17

1. Questa estensione si attiva nel caso in cui si verifichi un errore generico di sistema subito dopo che l'utente ha interagito con uno dei toggle per la gestione delle notifiche.
2. Il sistema visualizza un messaggio di errore che informa l'utente dell'impossibilità di completare l'azione e ripristina lo stato precedente del toggle.

3.6 Osservazioni fatte da stakeholder

Durante la fase di analisi dei requisiti, abbiamo condotto una serie di colloqui e sessioni di confronto con utenti reali e potenziali stakeholder del sistema, tra cui clienti interessati all'acquisto o affitto di immobili, agenti immobiliari e gestori di agenzie. Tali interazioni ci hanno permesso di raccogliere spunti concreti e osservazioni utili per orientare le decisioni progettuali in modo mirato.

Di seguito sono riportati i principali punti emersi da queste interazioni:

1. **Intuitività dell'interfaccia:** Gli utenti finali, in particolare quelli con poca esperienza digitale, hanno ribadito l'importanza di un'interfaccia semplice e coerente. Elementi come la ricerca guidata, la visualizzazione su mappa e la chiarezza dei pulsanti sono stati indicati come fondamentali per garantire un utilizzo fluido dell'applicazione.
2. **Accesso immediato alle informazioni rilevanti:** Le persone coinvolte desiderano poter consultare in modo diretto e senza passaggi superflui le informazioni chiave di un immobile: prezzo, superficie, posizione, numero di stanze e classe energetica. La presenza di icone sintetiche e indicatori visivi è stata considerata molto utile.
3. **Esperienza reattiva nella presentazione di offerte:** Durante la simulazione del flusso di offerta, è emersa la richiesta di un'interazione fluida e immediata con il sistema. Eventuali ritardi o incertezze nei feedback dell'app potrebbero compromettere l'esperienza utente.
4. **Personalizzazione e profilo utente:** Alcuni clienti hanno espresso il desiderio di personalizzare il proprio profilo, visualizzare lo storico delle ricerche e delle offerte, e salvare preferenze di ricerca che possano essere riutilizzate nel tempo.
5. **Supporto contestuale alla posizione:** È stato valutato positivamente il supporto automatico alla geolocalizzazione tramite API esterne, che consente di indicare se un immobile è vicino a servizi pubblici come scuole, fermate del trasporto o aree verdi.

Le osservazioni raccolte sono state tenute in considerazione nella fase di definizione dei requisiti e hanno contribuito a migliorare la coerenza tra aspettative degli utenti e funzionalità offerte dalla piattaforma.

questa pagina è stata lasciata volutamente bianca

4 Design del Sistema

In questa sezione viene descritto il design del sistema **DietiEstates25**, a partire dall'architettura generale, fino alle scelte tecnologiche e all'interfaccia utente. La progettazione è stata guidata da principi di modularità, riusabilità, semplicità ed efficienza, in linea con i requisiti funzionali e non funzionali precedentemente analizzati.

4.1 Architettura del sistema

Si è deciso di adottare un'architettura a **tre livelli** suddivisa in:

- **Frontend**: responsabile della presentazione dei dati e dell'interazione con l'utente.
- **Backend**: un'applicazione server-side che espone un set di API REST per la gestione dei dati e la logica di business.
- **Database**: sistema di persistenza dei dati relazionale

Docker: È stato inoltre dockerizzato il backend NestJS, al fine di semplificare il processo di deploy e rendere l'applicazione facilmente eseguibile in ambienti differenti.

Questa architettura garantisce scalabilità, facilità di manutenzione e separazione delle responsabilità.

4.2 Distribuzione dei componenti lato server

Per ospitare l'infrastruttura lato server, è stata adottata una soluzione autonoma e flessibile tramite l'utilizzo di una macchina Linux dedicata, configurata come nodo server permanente e accessibile da rete pubblica. Tale macchina funge da ambiente di produzione per il sistema, simulando l'infrastruttura di un'azienda in fase di avvio che gestisce in proprio le risorse cloud. Nello specifico, i principali componenti del sistema sono distribuiti come segue:

- **Backend:** il server NestJS, responsabile dell'esposizione delle API REST e della gestione della logica applicativa, è in esecuzione sulla macchina Linux e in ascolto su porta 3000.
- **Database:** il database relazionale PostgreSQL risiede anch'esso sulla stessa macchina, protetto da configurazioni di firewall e autenticazione basata su credenziali sicure.
- **Storage immagini:** le immagini caricate dagli utenti (es. fotografie degli immobili) sono salvate in una directory del file system della macchina, organizzate in directory con un identificativo univoco e accessibili tramite endpoint.
- **Docker:** il backend NestJS è stato containerizzato utilizzando Docker, permettendo una gestione più semplice del ciclo di build e deploy. L'immagine è costruita a partire da un `Dockerfile` personalizzato e viene eseguita in modo isolato sulla macchina Linux. Questo approccio assicura portabilità, riproducibilità dell'ambiente e semplifica il rilascio di nuove versioni.
- **Dominio pubblico:** l'intero sistema è esposto su Internet tramite il dominio `dietestates.duckdns.org`, aggiornato automaticamente ogni 5 minuti. Questo consente di mantenere il servizio disponibile anche in presenza di IP dinamico, simulando un ambiente server sempre raggiungibile come avverrebbe in un contesto aziendale o su cloud privato.

Questa configurazione ha permesso di sviluppare e testare l'intero sistema in un contesto realistico, garantendo al tempo stesso controllo completo sull'ambiente di esecuzione.

4.3 Tecnologie Utilizzate

- **Frontend:** Kotlin + Jetpack Compose (Android) + Gradle
- **Backend:** NestJS + TypeOrm + Npm
- **Docker, Docker Compose**
- **Database:** PostgreSQL
- **Autenticazione:** Google OAuth e JWT
- **Dominio:** DuckDNS
- **Geolocalizzazione:** API Geoapify e Google Map
- **Mockup UI:** Figma

4.4 Design pattern adottati

Nel progetto **DietiEstates25** sono stati adottati design pattern consolidati sia per il backend che per il frontend, al fine di garantire modularità, testabilità e manutenzione efficiente del codice.

4.4.1 Backend: Controller – Service – Repository

L’architettura backend, realizzata con NestJS, segue il pattern a tre livelli:

- **Controller:** gestisce le richieste HTTP in ingresso, si occupa della validazione preliminare dei parametri e delega la logica applicativa al livello sottostante (Service). È responsabile della definizione degli endpoint REST e della restituzione delle risposte HTTP.
- **Service:** contiene la logica di business del sistema. Esegue operazioni complesse, aggrega dati da più repository, gestisce flussi e applica le regole di dominio.
- **Repository:** incapsula l’accesso al database. Espone metodi di persistenza e recupero dei dati (CRUD), astratti dalla logica applicativa, sfruttando l’ORM (TypeORM) per interagire con il database PostgreSQL.

Questa suddivisione consente una netta separazione delle responsabilità, facilita l’attività di testing unitario e rende possibile l’evoluzione indipendente dei singoli livelli.

4.4.2 Frontend: Model – View – ViewModel (MVVM)

L'applicazione mobile è sviluppata in **Kotlin** con **Jetpack Compose** e adotta il pattern MVVM, largamente utilizzato nell'ecosistema Android moderno.

- **Model:** rappresenta i dati e le entità del dominio (ad es. `Listing`, `Offer`).
- **ViewModel:** gestisce lo stato dell'interfaccia e l'interazione con il Model. Espone flussi reattivi (`StateFlow`) alla View e si occupa della logica di trasformazione dei dati, gestione degli eventi e orchestrazione delle chiamate API.
- **View:** realizzata tramite `@Composable`, osserva gli stati esposti dal ViewModel e visualizza dinamicamente i dati. È completamente stateless, delegando la logica al ViewModel.

Il pattern MVVM è stato scelto per favorire la separazione tra logica di presentazione e business logic, migliorare la reattività dell'interfaccia utente e supportare la persistenza dello stato.

Integrazione dei pattern

Il flusso complessivo tra frontend e backend, basato sui design pattern adottati, può essere sintetizzato come segue:

- L'utente interagisce con la **View**, che invia eventi al **ViewModel**.
- Il **ViewModel** effettua chiamate alle API esposte dai **Controller** nel backend.
- Il **Controller** riceve la richiesta e delega al **Service**, che applica la logica di business.
- Il **Service** interagisce con il **Repository** per recuperare o salvare dati.
- I dati viaggiano all'indietro fino alla **View**, che li visualizza aggiornando l'interfaccia.

Criteri di scelta del design

Le scelte progettuali sono state guidate dai seguenti criteri:

- **Separation of Concerns:** ogni componente ha responsabilità ben definite.
- **Scalabilità:** l'architettura consente di integrare nuove funzionalità e microservizi in futuro.
- **Responsiveness:** lato mobile, particolare attenzione è stata data alla reattività e fluidità dell'interfaccia.

Questo approccio garantisce un'elevata manutenibilità del sistema, facilità di testing e riutilizzabilità dei componenti.

4.5 Persistenza dei Dati

Il sistema *DietiEstates25* utilizza un database relazionale **PostgreSQL**, ospitato su una **macchina linux dedicata**, per la gestione persistente delle informazioni. La scelta di un RDBMS è stata motivata da esigenze di *consistenza*, *integrità referenziale* e capacità di effettuare *query complesse* in modo efficiente.

Modello concettuale

I dati sono organizzati secondo un modello relazionale che garantisce:

- **Ridondanza minima:** grazie a relazioni ben definite tra tabelle.
- **Integrità dei dati:** tramite vincoli di chiave primaria, esterna e regole di validazione.
- **Manutenibilità:** schema facilmente estendibile per futuri sviluppi.

4.6 Diagramma UML

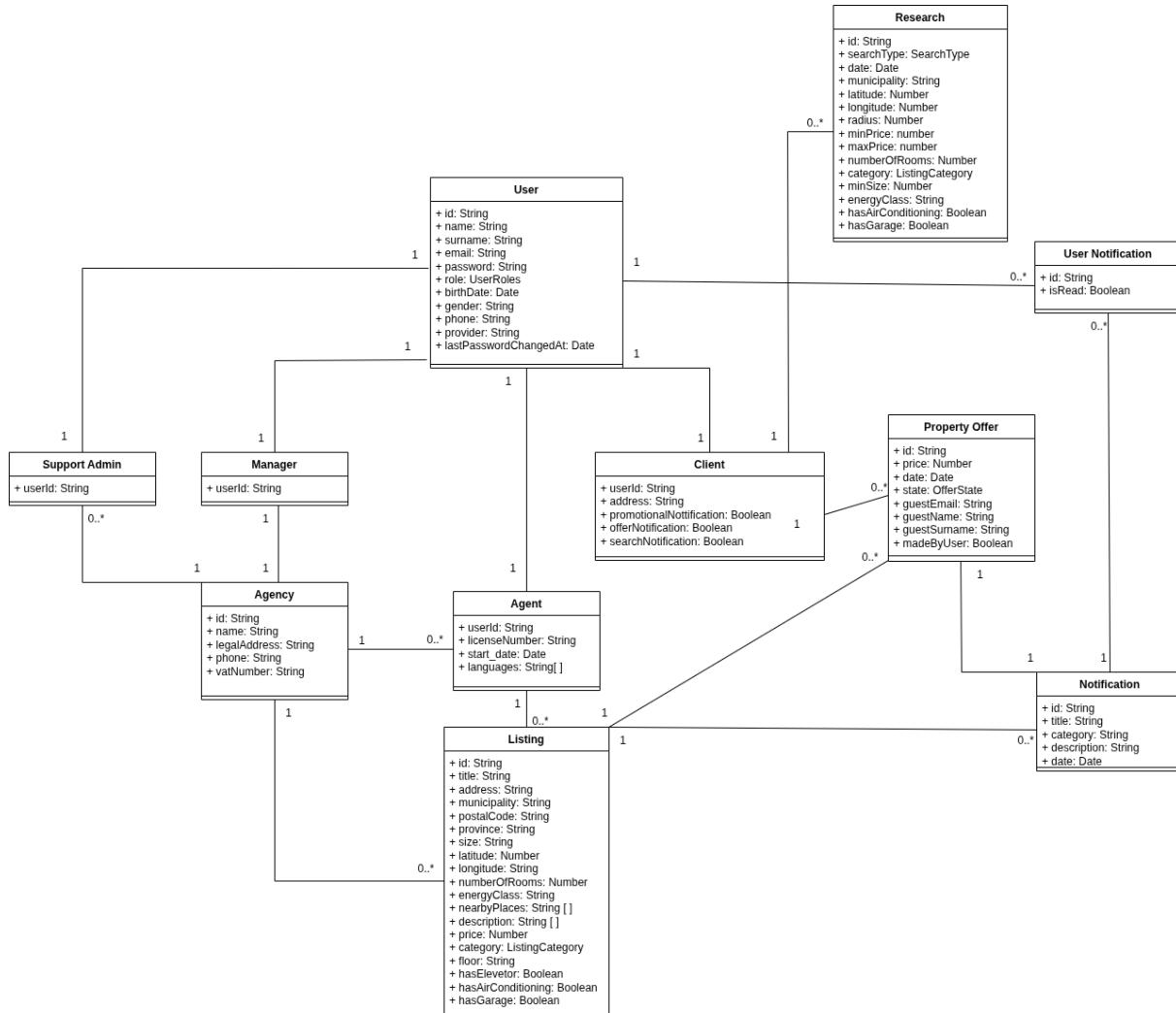


Figure 4.1: Diagramma UML

4.6.1 Entità principali

Le principali entità gestite dal sistema sono:

- **Utente (User)**: rappresenta un individuo registrato (Cliente, Agente, Admin di supporto o Gestore), con attributi comuni come nome, cognome, email, password, data di nascita, ruolo, ecc...
- **Cliente (Client)**: rappresenta l'entità cliente, cioè l'utilizzatore finale dell'app con attributi come l'indirizzo e le scelte per la ricezione delle notifiche
- **Agente, Admin Di Supporto, Gestore (Agent, SupportAdmin, Manager)**: rappresentano i possibili ruoli dei dipendenti all'interno di un'agenzia
- **Annuncio (Listing)**: rappresenta un immobile pubblicato, con attributi come titolo, descrizione, prezzo, posizione, ecc...
- **Offerta (Offer)**: rappresenta una proposta di acquisto o affitto da parte di un utente. Quest ultimo può essere autenticato oppure fare un offerta che verrà poi inserita dall'agente referente.
- **Agenzia (Agency)**: rappresenta una struttura organizzativa che raggruppa più agenti, support admin e un gestore.
- **Notifica (Notification)**: tiene traccia delle notifiche presenti, per le quali verranno poi create delle UserNotification per ogni utente che la dovrà ricevere.
- **Notifica Utente (UserNotification)**: entità che rappresenta la notifica per lo specifico utente, con l'attributo isRead per verificare se quella notifica è già stata letta o meno.
- **Ricerca (Research)**: rappresenta le ricerche effettuate dagli utenti, con attributi come tipo di ricerca (per comune o coordinate e raggio), e i vari filtri.

Relazioni principali

- Ogni **User** è associato esattamente a una tra le entità **Client**, **Agent**, **Manager** o **Support Admin**, mediante una relazione 1:1. Il tipo di utente viene determinato dal ruolo (**role**) e dalla presenza della relativa entità secondaria.
- Ogni **Client** può eseguire più **Research**, con una relazione uno-a-molti (1 -- 0..*).
- Ogni **Client** può effettuare più **Property Offer**, ma ciascuna offerta è legata ad un solo **Listing**. Ogni **Listing** può quindi ricevere più offerte (relazione molti-a-uno).
- Ogni **Property Offer** può essere generata da un utente registrato oppure inserita manualmente come offerta esterna (guest), identificabile tramite i campi **guestName**, **guestEmail**, ecc. Il campo **madeByUser** indica se l'offerta è stata fatta da un agente o un cliente.
- Ogni **User** può ricevere più **User Notification**, ognuna delle quali è anche collegata ad una singola **Notification** generale.
- Ogni **Manager** appartiene a una singola **Agency**, con una relazione 1:1.
- Ogni **Agency** può avere più **Agent**, mentre ciascun agente è associato a una singola agenzia (1 -- 0..*).
- Ogni **Agent** può pubblicare più **Listing**, ma ciascun annuncio è gestito da un solo agente. Anche in questo caso la relazione è uno-a-molti.
- Ogni **Listing** può avere più offerte (**Property Offer**) e può essere oggetto di più ricerche da parte dei client (in base ai criteri di ricerca).
- Ogni **Notification** rappresenta un messaggio generico del sistema (es. “Hai ricevuto una nuova offerta”), con attributi come **title**, **description**, **category** e **date**.
- Ogni **Notification** può essere indirizzata a più utenti tramite la tabella intermedia **User Notification**, che modella una relazione molti-a-molti tra **User** e **Notification**. La tabella contiene anche il campo **isRead**, che specifica se l'utente ha visualizzato la notifica.
- Ogni **User** può quindi ricevere più **Notification** (tramite **User Notification**), e ogni notifica può essere inviata a più utenti. Si tratta di una classica relazione **molti-a-molti con attributi**.

Accesso ai dati

L'accesso al database è gestito tramite uno strato di **Repository**, implementato con il framework *TypeORM* nel backend *NestJS*. Questo approccio favorisce una chiara separazione delle responsabilità tra i diversi livelli dell'applicazione e consente di:

- Separare la logica di business dalla logica di accesso ai dati, migliorando la manutenibilità del codice.
- Gestire in modo trasparente il mapping tra oggetti TypeScript e le tabelle del database relazionale (ORM - Object Relational Mapping).
- Favorire la modularità e il riuso dei metodi di accesso ai dati, centralizzando le operazioni CRUD.

Sicurezza e affidabilità

- Tutti i dati sensibili, come le password, sono **criptati utilizzando l'algoritmo bcrypt** prima del salvataggio nel database, garantendo un elevato livello di sicurezza.
- Le operazioni critiche che coinvolgono più passaggi (es. creazione di entità collegate o aggiornamenti concatenati) sono gestite tramite **transazioni**. Una transazione è un'unità atomica di lavoro che garantisce che tutte le operazioni al suo interno vengano completate con successo: se anche solo una fallisce, tutte le modifiche vengono annullate (rollback), evitando stati inconsistenti del database.
- Nella maggior parte dei casi, i dati vengono restituiti al frontend come oggetti completi, ad eccezione di alcuni campi sensibili che vengono esclusi tramite appositi filtri o annotazioni (**exclude** o selezione dei campi). In alcuni endpoint specifici, vengono utilizzati **DTO** (Data Transfer Object) per strutturare in modo più controllato la risposta, ma non in maniera sistematica su tutte le entità.

4.7 Design Interfaccia Utente

L’interfaccia utente dell’applicazione mobile è stata progettata per garantire una navigazione fluida, un’interazione immediata e un aspetto coerente con le moderne linee guida di progettazione, adottando lo stile **Material Design 3** e sfruttando il framework **Jetpack Compose** per Android.

L’applicazione presenta una struttura a schermate principali, raggiungibili attraverso una **barra di navigazione inferiore** (*bottom bar*) visibile in modo persistente in quasi tutte le viste. La top bar, invece, fornisce contesto alla schermata corrente tramite titolo e branding dell’app.

Schermata iniziale

All’apertura dell’app, se non già precedentemente autenticato, l’utente accede a una schermata di **login**, dove può:

- effettuare l’accesso con credenziali (email e password);
- registrarsi tramite un apposito form di iscrizione;
- utilizzare l’accesso semplificato tramite **Google Sign-In**.

Questa schermata è progettata per essere chiara, con pulsanti ben visibili e feedback in tempo reale per input errati, al fine di minimizzare errori in fase di autenticazione.

Navigazione principale

Dopo il login, l’utente accede alla **home**, dove viene mostrato un elenco di *listing* (annunci immobiliari), ognuno rappresentato da una card informativa con immagine, titolo, indirizzo e prezzo. In alto è presente una top bar con il nome dell’applicazione e un testo che varia a seconda del contesto. La navigazione tra le diverse sezioni avviene tramite una **bottom navigation bar**, sempre visibile, che include le seguenti icone:

- **Home** – elenco degli annunci ;
- **Ricerca** – accesso ai filtri e alla ricerca geolocalizzata;
- **Notifiche** – sezione dedicata alle comunicazioni ricevute;
- **Profilo** – informazioni personali e preferenze;
- **Offerte** – offerte effettuate o ricevute, a seconda del ruolo.

Bottom bar

La **bottom bar** è adattiva: la disposizione e la visibilità delle icone possono variare in base al tipo di utente autenticato. Ad esempio:

- un **cliente** visualizzerà solo le voci relative alla propria esperienza utente;
- un **agente, admin di supporto o gestore** vedrà voci aggiuntive o modificate, coerenti con le funzionalità a lui riservate, ad esempio non vedrà il tasto di ricerca ma vedrà un tasto per aggiungere un nuovo listing.

4.7.1 Home

La schermata **Home** presenta un elenco dinamico di annunci immobiliari (*listing*). Il contenuto di questa schermata varia in base alla tipologia di utente:

- **Cliente:** visualizza l'intero catalogo degli annunci pubblicati all'interno della piattaforma, provenienti da qualunque agenzia. Questa modalità consente ai clienti di esplorare liberamente l'offerta immobiliare.
- **Agente:** visualizza esclusivamente gli annunci creati da sé stesso. Questa schermata funge da pannello di gestione dei propri immobili, consentendo eventualmente la modifica o eliminazione di ciascun annuncio.
- **Gestore/Admin di supporto:** visualizzano tutti gli annunci pubblicati all'interno dell'agenzia di appartenenza, indipendentemente dall'agente che li ha creati. Questo consente un controllo completo sull'insieme dei listing della propria organizzazione.

Ogni annuncio è presentato tramite una card che include immagine principale, titolo, indirizzo, prezzo e stato, con possibilità di accedere a una schermata di dettaglio completa.

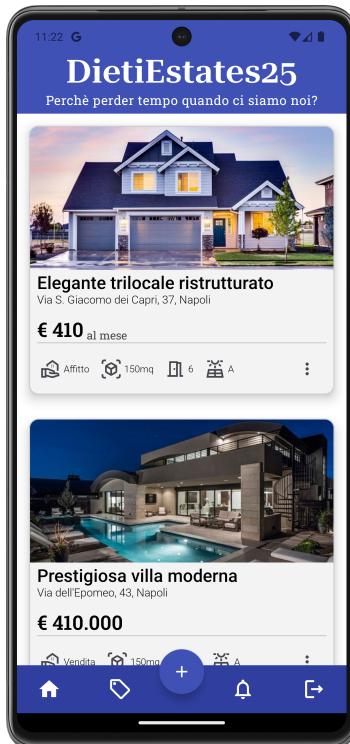


Figure 4.2: Home

4.7.2 Schermata Offerte

La schermata **Offerte** è una sezione centrale dell'applicazione, il cui contenuto e comportamento si adatta dinamicamente in base al ruolo dell'utente autenticato. In particolare, essa permette di consultare e gestire lo storico delle trattative per ogni immobile, sotto forma di conversazione strutturata in offerte inviate o ricevute. Le modalità di interazione variano come segue:

- **Cliente:** l'utente visualizza un elenco di tutti gli *immobili* per i quali ha inviato almeno un'offerta. Cliccando su un immobile, viene reindirizzato a una schermata che simula una **chat** tra lui e l'agente referente dell'annuncio. Ogni messaggio della chat rappresenta un'offerta (inviata dal cliente o dall'agente), e contiene:
 - l'importo dell'offerta;
 - il mittente (cliente o agente);
 - lo stato dell'offerta: *in attesa*, *accettata*, o *rifiutata*.

Questa struttura consente una visione cronologica e trasparente della negoziazione.

- **Agente:** visualizza un elenco di tutti gli *immobili di sua proprietà*. Selezionando uno specifico immobile, accede a una schermata che mostra una **tabella** riepilogativa, dove ogni riga rappresenta un cliente che ha effettuato almeno un'offerta per quell'annuncio. Per ciascun cliente viene mostrata l'ultima offerta ricevuta.

Se l'agente clicca su una riga della tabella, viene reindirizzato alla stessa schermata di **chat** accessibile dal cliente, dove è possibile visualizzare l'intero scambio di offerte. Inoltre, nella schermata tabellare è presente un pulsante per **inserire manualmente un'offerta esterna**, utile per registrare proposte ricevute al di fuori della piattaforma.

- **Gestore / Admin di supporto:** accedono a una schermata identica a quella degli agenti, ma con una differenza sostanziale: invece di visualizzare solo gli immobili dell'agente autenticato, hanno accesso all'intero elenco degli *immobili appartenenti alla propria agenzia*. Anche in questo caso, cliccando su un immobile si accede alla tabella dei clienti con offerte attive, e da lì alla conversazione dettagliata.

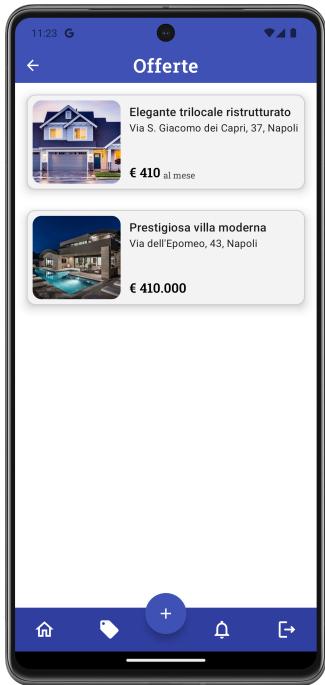


Figure 4.3: Immobili con offerte

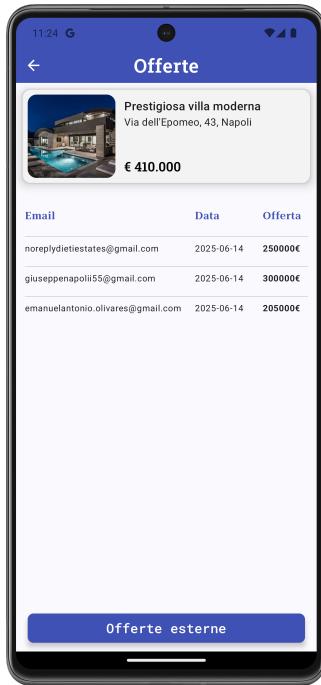


Figure 4.4: Offerte per immobile



Figure 4.5: Chat Offerte

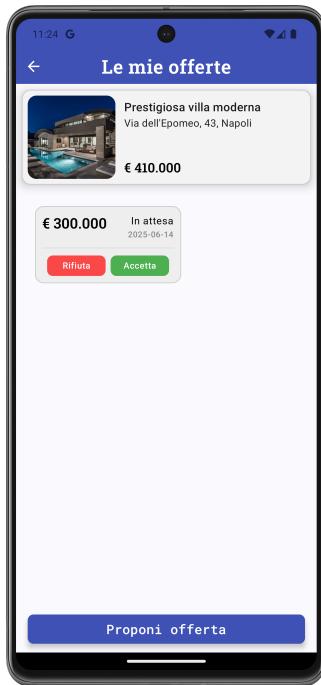


Figure 4.6: Chat Offerte

4.7.3 Schermata Ricerche

La schermata **Ricerche** è disponibile esclusivamente per gli utenti con ruolo di **cliente**. Consente di effettuare nuove ricerche o ripetere ricerche precedenti, selezionabili da una cronologia. Quando si effettua una nuova ricerca si possono specificare parametri come l'**area geografica** (con possibilità di specificare un raggio di interesse), la **fascia di prezzo**, il **numero di stanze**, la **tipologia dell'immobile**, ecc.

Ogni ricerca viene salvata per ricevere notifiche automatiche in caso di nuovi annunci compatibili. Gli altri ruoli non accedono a questa sezione, ma dispongono di una funzionalità alternativa: un tasto rapido per la **creazione di un nuovo immobile**, accessibile direttamente dalla barra di navigazione.

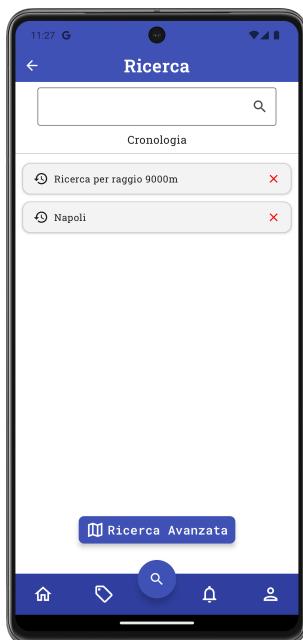


Figure 4.7: Ricerca

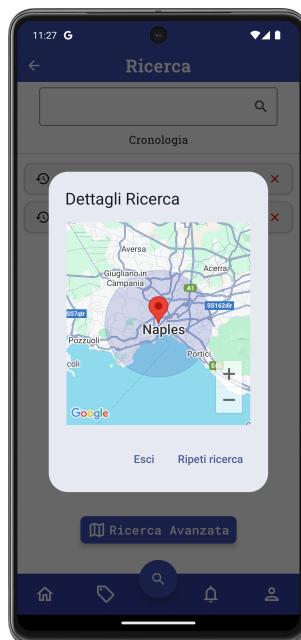


Figure 4.8: Ricerca Passata



Figure 4.9: Campi Ricerca

4.7.4 Notifiche

La schermata **Notifiche** è condivisa tra tutti i tipi di utenti e mostra un elenco ordinato cronologicamente di messaggi rilevanti, tra cui:

- ricezione o esito di un'offerta;
- nuove proposte ricevute da clienti (per agenti);
- notifiche promozionali o informative;
- aggiornamenti sugli immobili in osservazione.

Ogni notifica può essere contrassegnata come letta. I dati sono sincronizzati in tempo reale con il backend e riflettono le informazioni presenti nella tabella **User Notification**.

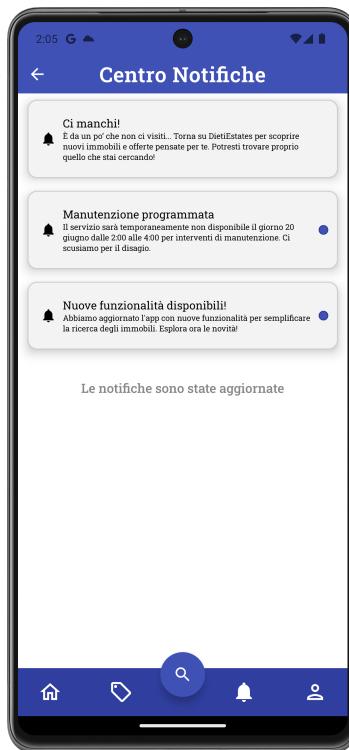


Figure 4.10: Notifiche

4.7.5 Schermata Profilo

La schermata **Profilo** varia in base al tipo di utente loggato:

- **Cliente:** ha accesso ai propri dati personali (nome, cognome, email, telefono, genere, data di nascita) e anche tre interruttori (*toggle*) per attivare o disattivare la ricezione delle notifiche:
 - promozionali,
 - relative alle ricerche salvate,
 - relative alle offerte.
- **Agente:** non dispone di una vera e propria schermata profilo, ha direttamente il tasto logout per disconnettersi.
- **Gestore/Admin di supporto:** visualizzano le informazioni della propria agenzia e dispongono di due pulsanti per la **gestione dello staff**:
 - aggiunta di *agenti immobiliari*,
 - aggiunta di *admin di supporto*.



Figure 4.11: Profilo utente

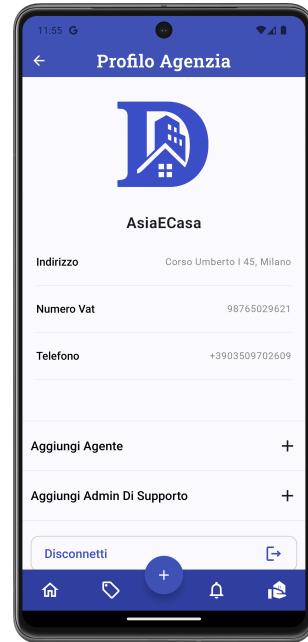


Figure 4.12: Profilo agenzia

4.8 Diagrammi di sequenza

I diagrammi di sequenza presentati in questa sezione derivano direttamente dalla Sezione 3.5. Essi rappresentano, secondo una prospettiva temporale, le interazioni tra gli attori esterni e i componenti principali del sistema, con particolare attenzione alla sequenza di messaggi scambiati per realizzare specifici scenari funzionali.

Questi diagrammi risultano fondamentali per esplicitare la dinamica dei processi, rendendo chiaro non solo chi interagisce con chi, ma anche in quale ordine. Vengono utilizzati durante la fase di progettazione per validare la coerenza delle operazioni previste dai casi d'uso e per guidare la successiva implementazione dei moduli applicativi. Ciascun diagramma rappresenta:

- Gli attori coinvolti;
- I componenti software responsabili dell'elaborazione;
- I messaggi o chiamate scambiati nel tempo;
- Eventuali condizioni o alternative tramite costrutti come `alt` o `opt`.

Nel complesso, i diagrammi di sequenza forniscono una visione dettagliata del comportamento del sistema, aiutando a individuare dipendenze, responsabilità e possibili punti critici nella comunicazione tra le varie parti del sistema.

4.8.1 Modifica delle preferenze di notifica

Il diagramma di sequenza in Figura 4.13 rappresenta il flusso di interazioni che si verifica quando un utente modifica una preferenza di notifica (es. notifiche promozionali, di ricerca o relative alle offerte) all'interno della schermata del profilo.

L'interazione ha inizio quando l'utente seleziona uno degli switch presenti nella UI, provocando la chiamata al metodo `updateNotification(type, value)` del `ProfileViewModel`. Quest'ultimo aggiorna immediatamente lo stato locale per offrire una risposta reattiva nell'interfaccia, dopodiché inoltra la richiesta di aggiornamento al repository dell'utente, attraverso il contenitore di dipendenze `AppContainer`.

Il `UserRepository` invia quindi una chiamata HTTP al backend (`PUT /user/notification`). Il diagramma rappresenta due possibili esiti:

- In caso di successo, il backend restituisce un codice HTTP 200 e l'operazione si conclude correttamente.
- In caso di errore (HTTP 400 o 500), il sistema esegue un *rollback* locale delle modifiche e notifica l'utente tramite un messaggio di errore (Toast).

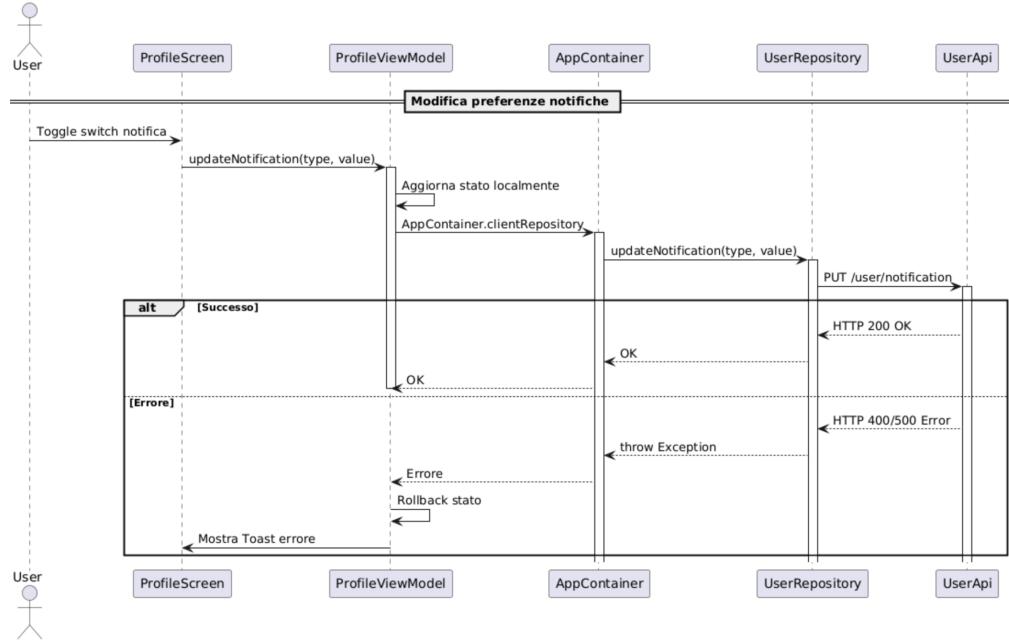


Figure 4.13: Modifica preferenze notifiche

4.8.2 Inserimento di un'offerta da parte di un cliente

Il diagramma di sequenza riportato in Figura 4.14 descrive il processo che consente a un utente con ruolo **CLIENT** di proporre un'offerta per un immobile.

L'interazione ha inizio quando il cliente, attraverso l'interfaccia **OfferScreen**, preme il pulsante per inviare l'offerta. Il **ListingOfferViewModel** esegue un controllo sul valore inserito: se l'importo è inferiore al 50% o superiore al prezzo dell'immobile, l'operazione viene interrotta e viene mostrato un messaggio d'errore.

In caso di validazione positiva, la richiesta viene inoltrata al backend tramite la catena **AppContainer** → **OfferRepository** → **OfferApi**, che espone l'endpoint `POST /offers/listing/{id}`. A seconda del risultato:

- In caso di successo (HTTP 201), la lista delle offerte viene aggiornata e l'utente riceve un messaggio di conferma.
- In caso di errore (HTTP 400/500), viene sollevata un'eccezione e il ViewModel aggiorna lo stato della UI con il messaggio d'errore.

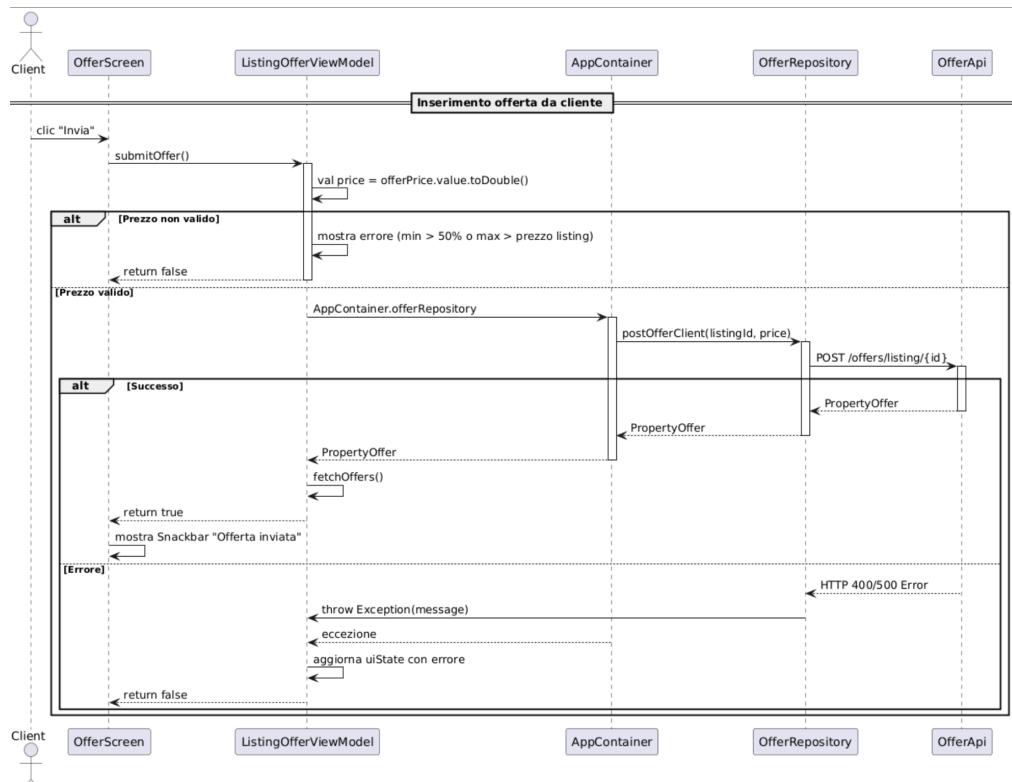


Figure 4.14: l'inserimento di un'offerta

4.8.3 Gestione di un'offerta da parte di un agente

Il diagramma in Figura 4.15 rappresenta la sequenza di interazioni che si verifica quando un agente seleziona un'offerta e ne modifica lo stato, accettandola o rifiutandola.

L'interazione parte dalla schermata `OfferScreen`, da cui l'agente attiva l'azione di modifica tramite il metodo `updateOfferStatus`. Il `ViewModel` aggiorna localmente lo stato dell'offerta nella UI tramite una modifica ottimistica (*optimistic update*), ovvero immediata e temporanea in attesa della conferma da parte del backend.

Successivamente, viene invocata la funzione `updateOfferState` nel `OfferRepository`, che a sua volta invia una richiesta PATCH all'endpoint `/offer/{id}`, con payload contenente il nuovo stato (es. `ACCEPTED` o `DECLINED`).

- In caso di successo, il backend restituisce l'offerta aggiornata, che viene propagata al `ViewModel`, il quale provvede a sincronizzare nuovamente la lista con una chiamata a `fetchOffers()`.
- In caso di errore (ad esempio conflitto HTTP 409), viene sollevata un'eccezione, lo stato della UI viene aggiornato con un messaggio d'errore e le offerte vengono ricaricate per ripristinare i dati corretti.

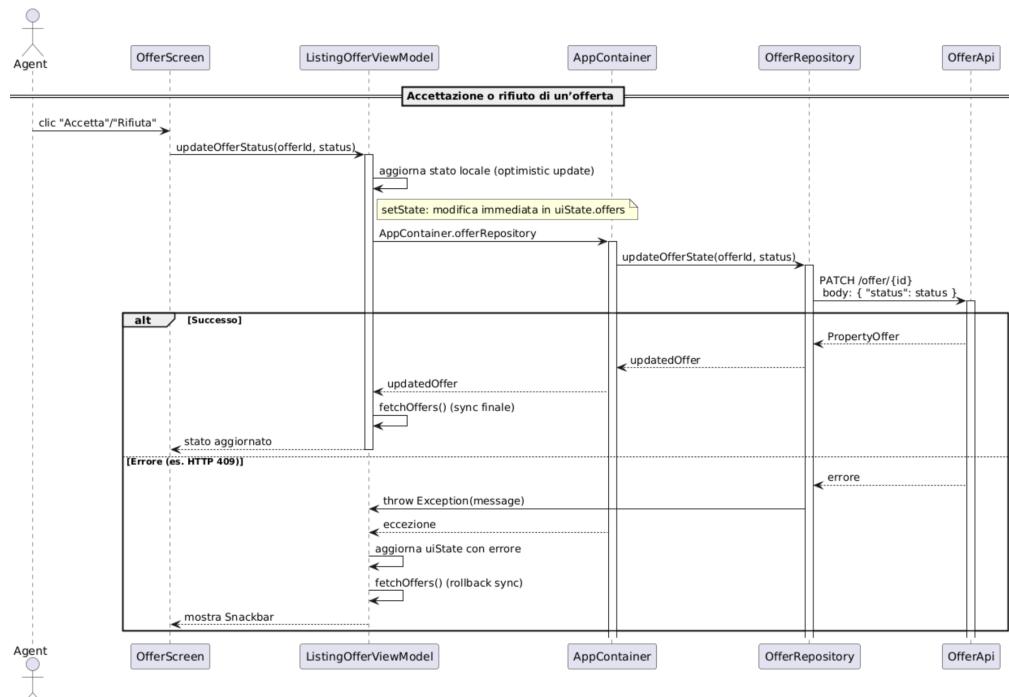


Figure 4.15: Gestione di un'offerta

4.8.4 Eliminazione di un annuncio da parte di un agente

Il diagramma in Figura 4.16 rappresenta la sequenza delle operazioni che si verificano quando un agente elimina un proprio annuncio immobiliare dalla schermata HomeScreen.

Il flusso ha inizio con il click sul pulsante "Elimina", che attiva il metodo `deleteListing` nel `HomeViewModel`. Questo invoca il repository tramite `AppContainer`, il quale inoltra la richiesta HTTP `DELETE` verso l'endpoint `/listing/{id}`.

- In caso di risposta positiva (HTTP 200), l'elenco degli annunci viene ricaricato con `fetchListings()` e viene notificato l'esito positivo all'utente.
- In caso di errore, viene sollevata un'eccezione e l'utente riceve un messaggio di errore tramite la funzione `onError`.

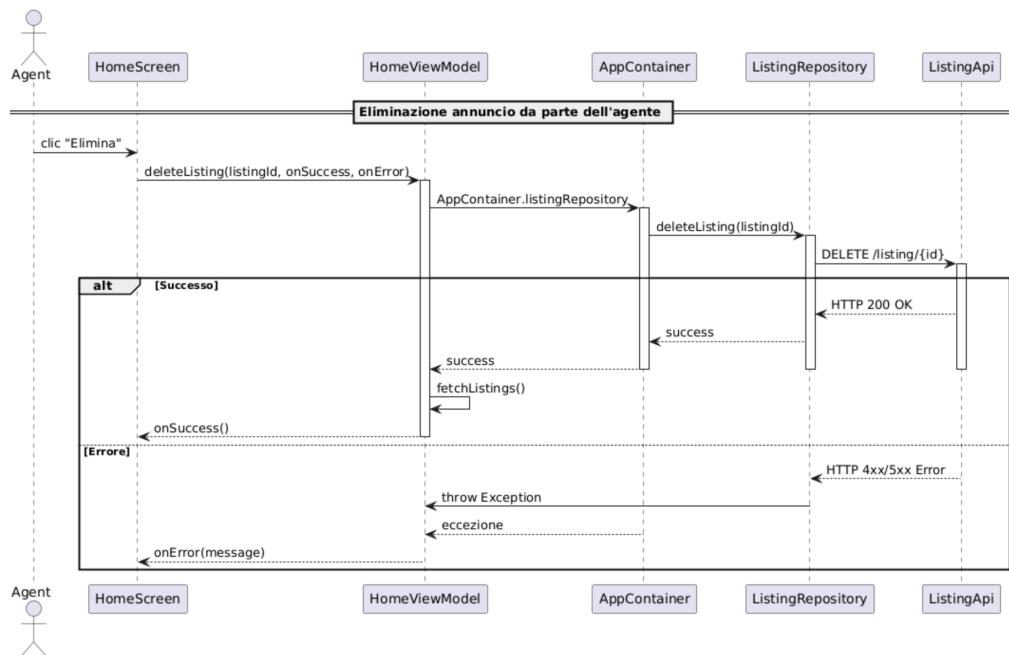


Figure 4.16: Eliminazione di un annuncio

5 Documentazione e artefatti del processo di sviluppo

Nel contesto della realizzazione di un sistema software strutturato, la produzione e gestione della documentazione assume un ruolo centrale per garantire trasparenza, tracciabilità e qualità durante l'intero ciclo di vita del progetto.

Durante lo sviluppo, sono inoltre prodotti numerosi artefatti software, ovvero output tangibili generati nel corso delle attività ingegneristiche.

5.1 Versionamento

Particolare attenzione è stata dedicata all'impiego di strumenti di controllo versione come **Git** e **GitHub**, che hanno permesso di mantenere una cronologia completa e dettagliata delle modifiche apportate, facilitando il lavoro collaborativo e garantendo una gestione efficace delle versioni. Inoltre, tali strumenti hanno reso possibile la generazione automatica di report statistici utili a documentare in modo oggettivo l'impegno del gruppo: numero di commit, distribuzione dei contributi e frequenza degli aggiornamenti costituiscono indicatori significativi della progressione dello sviluppo.

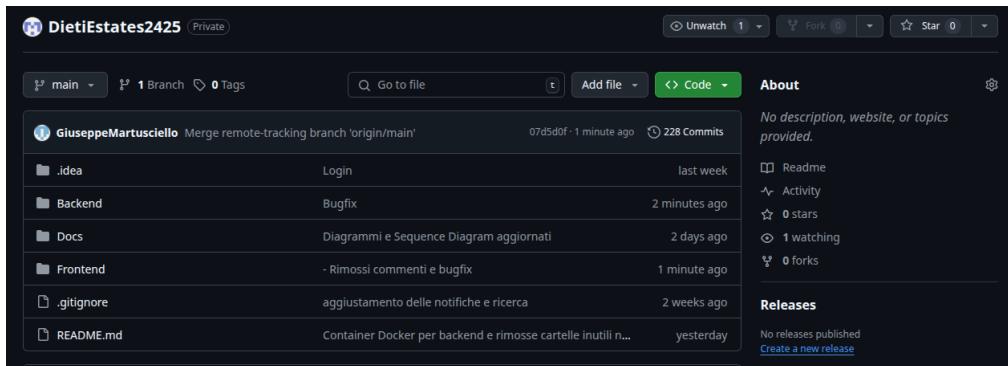
5.1.1 Github

Durante lo sviluppo del progetto è stato adottato un flusso di lavoro basato su **Git** e **GitHub**, con l'obiettivo di garantire un processo collaborativo efficace, ordinato e tracciabile.

La repository ufficiale del progetto è disponibile pubblicamente al seguente indirizzo:

<https://github.com/GiuseppeSindoni03/DietiEstates2425>

Ogni collaboratore ha operato all'interno del proprio branch, sviluppando funzionalità specifiche in maniera indipendente. Questo approccio ha permesso di evitare conflitti diretti sul codice e di mantenere il branch principale (**main**) stabile e privo di modifiche non testate.



Il ciclo di lavoro ha previsto le seguenti operazioni:

1. **Commit frequenti** per tracciare ogni modifica significativa, accompagnati da messaggi descrittivi e coerenti con le convenzioni (`feat:`, `fix:`, `docs:`).
2. **Push regolari** verso i branch remoti, così da sincronizzare il lavoro e mantenere aggiornati gli altri membri del team.
3. **Pull request** verso il branch `main`, con revisione incrociata tra collaboratori prima di procedere con il merge.
4. **Merge** effettuato solo dopo approvazione e verifica dell'assenza di conflitti, assicurando una qualità costante del codice integrato.

File `.gitignore`

Nel progetto è stato definito un file `.gitignore` per escludere dal versionamento Git file e cartelle non pertinenti al codice sorgente, come dipendenze, artefatti di compilazione, file temporanei, configurazioni locali e file di ambiente. Questo consente di mantenere il repository pulito e privo di elementi specifici dell'ambiente di sviluppo individuale. La presenza di questo file è fondamentale per evitare che file temporanei, di sistema o sensibili (come variabili d'ambiente) vengano accidentalmente inclusi nel controllo versione e condivisi nel repository remoto.

5.2 SonarQube

Al fine di garantire uno standard elevato nella qualità del codice, è stato integrato nel progetto anche uno strumento di analisi automatica statico: **SonarQube**. Tale piattaforma ha permesso di monitorare metriche fondamentali contribuendo concretamente al miglioramento continuo del backend applicativo.

5.2.1 Analisi della qualità del codice

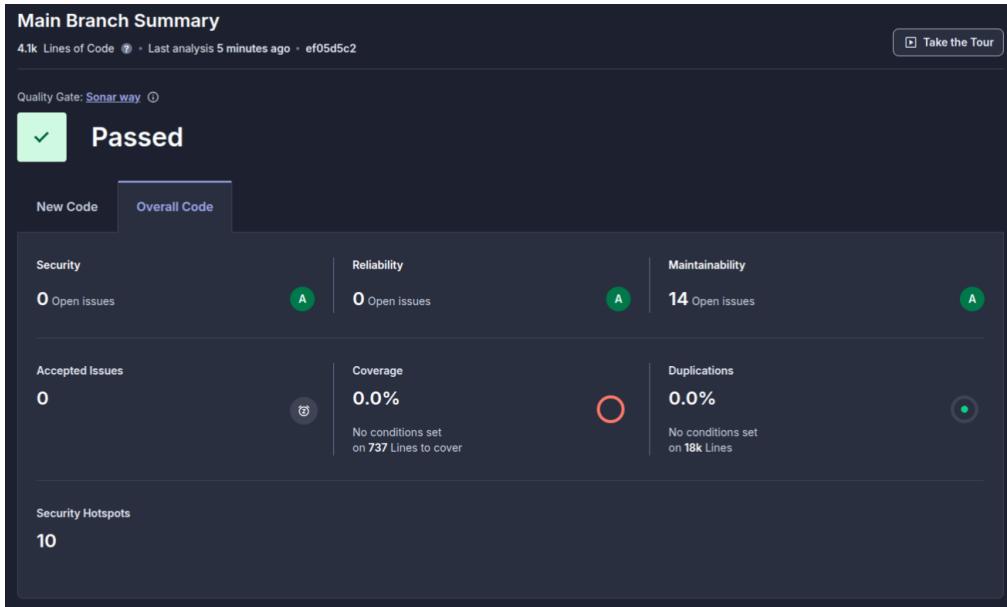


Figure 5.1: Analisi del Codice

L’analisi è stata effettuata sul branch `main` del progetto, e il risultato ha evidenziato un superamento positivo del *Quality Gate*, come mostrato in Figura 5.1. Di seguito si riportano le principali metriche emerse:

- **Security:** Nessuna issue rilevata in termini di vulnerabilità.
- **Reliability:** Nessun bug individuato nel codice.
- **Maintainability:** Presenti 14 *code smells*, ovvero difetti che non compromettono il funzionamento ma ne peggiorano la manutenzione.
- **Duplications:** Assenza di codice duplicato (0.0%), indicatore di buona qualità architetturale.
- **Security Hotspots:** Sono stati rilevati 10 hotspot da verificare manualmente, relativi a possibili comportamenti critici.



Figure 5.2: Grafiche delle valutazioni

Nel complesso, i risultati confermano una buona affidabilità e sicurezza del codice, suggerendo tuttavia margini di miglioramento nella copertura dei test e nella manutenibilità.

5.3 Containerizzazione del Backend

Per garantire portabilità, isolamento dell’ambiente e semplicità nella distribuzione, il backend è stato containerizzato utilizzando **Docker** e **Docker Compose**. La configurazione prevede tre elementi principali: un **Dockerfile**, un file **.dockerignore** e un **docker-compose.yml**.

5.3.1 Dockerfile

L’immagine viene costruita a partire dalla versione `node:20-alpine`, particolarmente leggera. I file vengono copiati `package.json` e `package-lock.json`, ed eseguita l’installazione delle dipendenze:

```
COPY package*.json ./
RUN npm install
```

Successivamente, viene copiato il resto del codice sorgente, compilato con `npm run build`, esposta la porta 3000 e avviata l’applicazione in modalità produzione:

```
COPY . .
RUN npm run build

EXPOSE 3000
CMD ["npm", "run", "start:prod"]
```

5.3.2 .dockerignore

Per evitare di includere file inutili o sensibili nell’immagine, è stato configurato un file `.dockerignore`, contenente:

```
node_modules
dist
.env
uploads
```

Questo permette di ridurre la dimensione dell’immagine e aumentare la sicurezza.

5.3.3 docker-compose.yml

L'esecuzione del container è stata orchestrata tramite Docker Compose. Il servizio **nest-backend** è definito come segue:

```
services:  
  nest-backend:  
    container_name: nest-backend  
    build:  
      context: .  
      dockerfile: Dockerfile  
    ports:  
      - "3000:3000"  
    env_file:  
      - .env  
    volumes:  
      - ./uploads:/usr/src/app/uploads  
    restart: always
```

In questo modo, il backend può essere avviato con un semplice `docker-compose up --build`, esponendo la porta 3000, leggendo le variabili d'ambiente da `.env`, e montando la cartella `uploads` per il caricamento file.

6 Testing e Valutazione dell’Usabilità del Sistema

Nel processo di sviluppo del sistema **DietiEstates25**, una particolare attenzione è stata riservata alla fase di verifica e validazione, elementi fondamentali per assicurare l'affidabilità e la qualità del software. In questo capitolo vengono presentate le strategie adottate per definire, eseguire e documentare i test realizzati, insieme alle metodologie impiegate per valutare l'usabilità dell'applicazione da parte degli utenti finali. La definizione accurata dei casi di test, unita all'adozione di strumenti automatizzati per l'esecuzione di test unitari, ha permesso di individuare tempestivamente eventuali malfunzionamenti e incongruenze rispetto ai requisiti iniziali. Parallelamente, la valutazione dell'interfaccia utente e dell'esperienza d'uso è stata condotta attraverso un duplice approccio: analisi da parte di esperti in usabilità e test controllati con utenti reali. Questo ha consentito di raccogliere feedback qualitativi e quantitativi fondamentali per perfezionare l'interazione con il sistema e garantire un'esperienza coerente, accessibile ed efficace per tutti i ruoli previsti.

La valutazione dell'usabilità dell'applicazione **DietiEstates25** è stata condotta seguendo un approccio strutturato su due livelli: **analisi da parte di esperti** (expert review) e **esperimento controllato con utenti reali**. L'obiettivo era identificare punti di forza e criticità dell'interfaccia utente, migliorare l'esperienza di utilizzo e garantire accessibilità.

6.1 Testing con Jest

Nel progetto è stato adottato il framework **Jest** per la scrittura dei test unitari e di integrazione, in combinazione con il framework backend **NestJS**. Jest è un framework di testing moderno, sviluppato da Facebook, che offre funzionalità complete come mocking, test coverage e snapshot testing.

Struttura di un test con Jest

Un test in Jest è generalmente strutturato in tre fasi:

- **Setup**: viene creato l'ambiente di test, instanziando i moduli necessari e mockando le dipendenze.
- **Act**: viene eseguito il metodo o la funzione da testare, eventualmente passando parametri di input.
- **Assert**: si verificano i risultati attesi tramite le asserzioni (`expect(...)`).

Testing con NestJS

NestJS fornisce un modulo chiamato `@nestjs/testing` che permette di costruire facilmente moduli di test isolati tramite la classe `TestingModule`. Con esso si possono mockare i provider, i repository, e ogni altra dipendenza, senza necessità di eseguire un database reale.

Le dipendenze vengono sostituite con versioni mockate tramite `jest.fn()`, e si utilizza `getRepositoryToken()` per ottenere correttamente il token dei repository TypeORM da iniettare nel contesto di test.

Vantaggi del testing con Jest

- **Rapidità**: esecuzione veloce dei test grazie all'isolamento e al mocking.
- **Affidabilità**: ogni unità viene testata separatamente, facilitando l'individuazione di errori.
- **Copertura**: è possibile generare facilmente report di copertura (`jest --coverage`).
- **Integrazione**: compatibilità completa con NestJS, TypeORM e TypeScript.

6.1.1 Testing del metodo `changePassword`

Il metodo `changePassword` del servizio `UserService` consente di aggiornare la password di un utente, previa verifica delle credenziali. Il metodo riceve in input un oggetto DTO contenente la password attuale e quella nuova, e l'identificativo dell'utente.

Nel test sono state mockate le dipendenze principali:

- il repository di `User`, tramite `getRepositoryToken(User)`;
- la libreria `bcrypt`, per simulare la verifica e l'hashing delle password.

Classi di equivalenza

Sono state identificate le seguenti classi di equivalenza per i parametri del metodo:

- `userId`:
 - **CE1 valida**: esiste nel sistema;
 - **CE2 non valida**: non esiste .
- `currentPassword`:
 - **CE1 valida**: corrisponde alla password salvata (autenticazione corretta);
 - **CE2 non valida**: non corrisponde.
- `newPassword`:
 - per l'attributo `newPassword` non ci siamo soffermati a definire classi di equivalenza in quanto la validazione è stata svolta precedentemente all'interno del dto tramite l'utilizzo di ***class-validator***.

Strategia di test

È stata adottata la tecnica **N-WECT** (Testing con Classi di Equivalenza Debolì), selezionando un insieme minimo di test che copre tutte le combinazioni significative delle classi sopra indicate:

- **Caso 1 (CE1, CE1):** l'utente esiste e la password è corretta ⇒ aggiornamento riuscito.



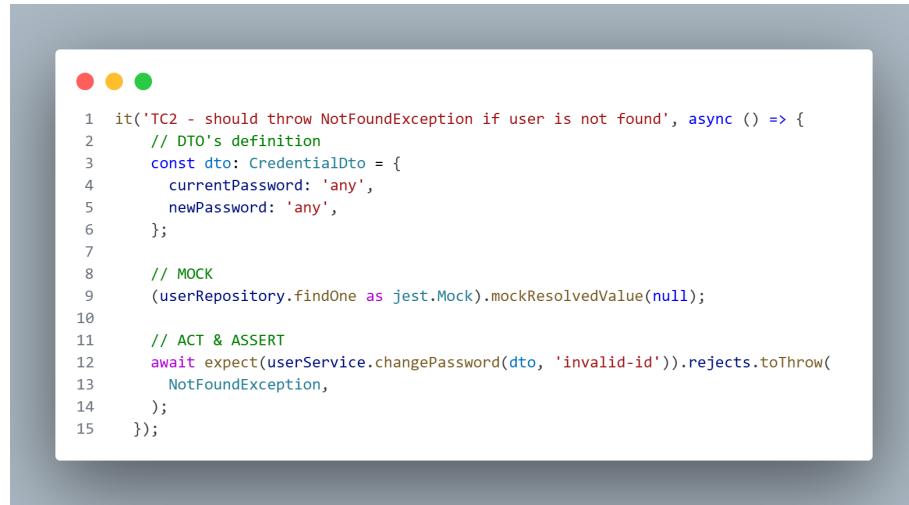
The screenshot shows a code editor with a dark theme. At the top left, there are three circular icons: red, yellow, and green. Below them is a block of 41 numbered lines of Jest test code. The code is written in JavaScript and uses Jest's expect and spyOn functions to test a userService.changePassword method. It involves mocking userRepository.findOne and save, and brypt.compare.

```

1  it('TC1 - should update password if currentPassword matches and user exists', async
2    c () => {
3      // DTO's definition
4      const dto: CredentialDto = {
5        currentPassword: 'oldPass123',
6        newPassword: 'newPass456!',
7      };
8
9      const newHashedPassword = 'newHashedPassword123';
10
11     const mockUserCopy = JSON.parse(JSON.stringify(mockUser));
12
13     // MOCK
14     (userRepository.findOne as jest.Mock).mockResolvedValue(mockUserCopy);
15     (userRepository.save as jest.Mock).mockResolvedValue(mockUserCopy);
16
17     jest.spyOn(bcrypt, 'compare').mockResolvedValue(true);
18     jest
19       .spyOn(userService as any, 'hashPassword')
20       .mockResolvedValue(newHashedPassword);
21
22     // ACT
23     const result = await userService.changePassword(dto, mockUser.id);
24
25     // ASSERT
26     expect(userRepository.findOne).toHaveBeenCalledWith({
27       where: { id: mockUser.id },
28     });
29
30     expect(bcrypt.compare).toHaveBeenCalledWith(
31       dto.currentPassword,
32       mockUser.password,
33     );
34     expect(userRepository.save).toHaveBeenCalledWith(
35       expect.objectContaining({
36         password: newHashedPassword,
37       }),
38     );
39     expect(result).toEqual({ message: 'Password updated successfully' });
40   });
41

```

- **Caso 2 (CE2, -):** l'utente non esiste \Rightarrow `NotFoundException`.



```

1 it('TC2 - should throw NotFoundException if user is not found', async () => {
2   // DTO's definition
3   const dto: CredentialDto = {
4     currentPassword: 'any',
5     newPassword: 'any',
6   };
7
8   // MOCK
9   (userRepository.findOne as jest.Mock).mockResolvedValue(null);
10
11  // ACT & ASSERT
12  await expect(userService.changePassword(dto, 'invalid-id')).rejects.toThrow(
13    NotFoundException,
14  );
15});

```

- **Caso 3 (CE1, CE2):** l'utente esiste ma la password è errata \Rightarrow `UnauthorizedException`.



```

1 it('TC3 - should throw UnauthorizedException if currentPassword is wrong', async
() => {
2   // DTO's definition
3   const dto: CredentialDto = {
4     currentPassword: 'wrongPass',
5     newPassword: 'newPass',
6   };
7
8   // MOCK
9   (userRepository.findOne as jest.Mock).mockResolvedValue(mockUser);
10  jest.spyOn(bcrypt, 'compare').mockResolvedValue(false);
11
12  // ACT & ASSERT
13  await expect(userService.changePassword(dto, mockUser.id)).rejects.toThrow(
14    UnauthorizedException,
15  );
16});

```

Esiti dei test

```

PASS  src/auth/user-service.spec.ts
UserService - changePassword
  ✓ TC1 - should update password if currentPassword matches and user exists (8 ms)
  ✓ TC2 - should throw NotFoundException if user is not found (7 ms)
  ✓ TC3 - should throw UnauthorizedException if currentPassword is wrong (2 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:  0 total
Time:        1.977 s, estimated 3 s
Ran all test suites matching /user-service.spec.ts/i.

```

6.1.2 Metodo `createExternalOffer`

Il metodo `createExternalOffer(createExternalOfferDto, userId, listingId)` del servizio `OfferService` consente di inserire un'offerta (`PropertyOffer`) proposta esternamente al sistema, riguardo uno specifico annuncio(`listingId`).

Il DTO `CreateExternalOfferDto` contiene i seguenti attributi:

- `price` - ammontare della proposta dell'annuncio in euro
- `guestEmail` - l'email di chi ha fatto la proposta
- `guestName` - nome di chi ha fatto la proposta
- `guestSurname` - cognome di chi ha fatto la proposta

Classi di equivalenza

Sono state identificate le seguenti classi di equivalenza per i parametri del metodo:

- `listingId`:
 - **CE1 valida:** l'identificativo dell'annuncio esiste nel sistema;
 - **CE2 non valida:** l'identificativo non è associato ad alcun annuncio esistente.
- `price`:
 - **CE1 valida:** $0 < price \leq listing.price$;
 - **CE2 non valida:** $price \leq 0$;
 - **CE3 non valida:** $price > listing.price$.
- `guestEmail` - `guestName` - `guestSurname`:
 - Non sono state definite classi di equivalenza per questi attributi, poiché la validazione è già gestita tramite annotazioni del pacchetto `class-validator` all'interno del DTO.
- `user`:
 - Questo parametro rappresenta l'utente autenticato. Non essendo un tipo primitivo, non si presta a una suddivisione in classi di equivalenza tradizionali.
 - È stato comunque necessario verificare con casi di test se il sistema, quando l'utente non è autorizzato, permette o meno di creare offerte per l'annuncio indicato.

Strategia di test

È stata adottata la tecnica **N-WECT** (Testing con Classi di Equivalenza Deboli Non Sovraposte), selezionando un insieme minimo di test in grado di coprire tutte le combinazioni significative delle classi di equivalenza individuate.

- **Caso 1 (listingId: CE1, price: CE1):** l'inserimento dell'offerta avviene correttamente.



```

1  it('TC1 - should create an external offer successfully', async () => {
2    // DTO's definition
3    const dto: CreateExternalOfferDto = {
4      price: 1000,
5      guestEmail: 'guest@example.com',
6      guestName: 'Mario',
7      guestSurname: 'Rossi',
8    };
9
10   const mockOffer = {
11     id: 'offer-1',
12     ...dto,
13     listing: mockListing,
14     state: OfferState.PENDING,
15     madeByUser: true,
16     date: new Date(),
17   };
18
19   // MOCK
20   (listingRepository.findOne as jest.Mock).mockResolvedValue(mockListing);
21
22   (offerRepository.create as jest.Mock).mockReturnValue(mockOffer);
23   (offerRepository.save as jest.Mock).mockResolvedValue(mockOffer);
24
25   // ACT
26   const result = await service.createExternalOffer(
27     dto,
28     mockUser,
29     mockListing.id,
30   );
31
32   // ASSERT
33   expect(result).toEqual(mockOffer);
34   expect(offerRepository.create).toHaveBeenCalledWith(
35     expect.objectContaining({
36       price: dto.price,
37       guestEmail: dto.guestEmail,
38       guestSurname: dto.guestSurname,
39       guestName: dto.guestName,
40       madeByUser: true,
41     }),
42     );
43   });

```

- **Caso 2 (listingId: CE2, price: -):** annuncio non esistente ⇒ restituisce una `NotFoundException`.



```

1  it('TC2 - should throw NotFoundException if listing does not exist', async () => {
2    // DTO's definition
3    const dto: CreateExternalOfferDto = {
4      price: 1000,
5      guestEmail: 'guest@example.com',
6      guestName: 'Mario',
7      guestSurname: 'Rossi',
8    };
9
10   // MOCK
11   (listingRepository.findOne as jest.Mock).mockResolvedValue(null);
12
13   // ACT & ASSERT
14   await expect(
15     service.createExternalOffer(dto, mockUser, 'invalid-listing'),
16   ).rejects.toThrow(NotFoundException);
17 });

```

- **Caso 3 (user non autorizzato):** l'utente autenticato non ha i permessi per creare l'offerta ⇒ viene sollevata una `UnauthorizedException`.

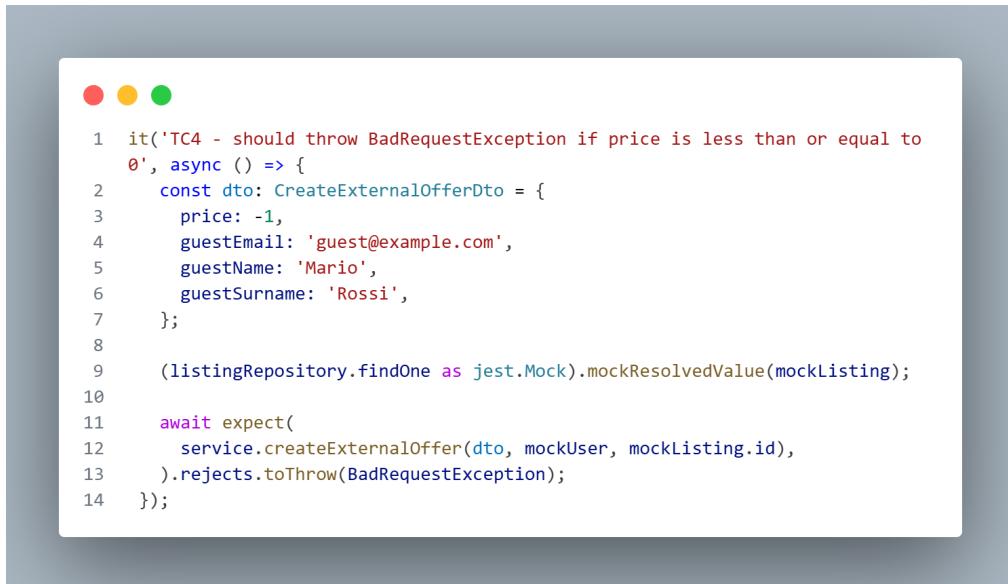


```

1  it('TC3 - should throw UnauthorizedException if user is not authorized', async c () => {
2    // DTO's definition
3    const dto: CreateExternalOfferDto = {
4      price: 1000,
5      guestEmail: 'guest@example.com',
6      guestName: 'Mario',
7      guestSurname: 'Rossi',
8    };
9
10   // MOCK
11   (listingRepository.findOne as jest.Mock).mockResolvedValue(mockListing);
12
13   const unauthorizedUser: UserItem = {
14     ...mockUser,
15     agent: mockWrongAgent,
16   };
17
18   // ACT && ASSERT
19   await expect(
20     service.createExternalOffer(dto, unauthorizedUser, mockListing.id),
21   ).rejects.toThrow(UnauthorizedException);
22 });

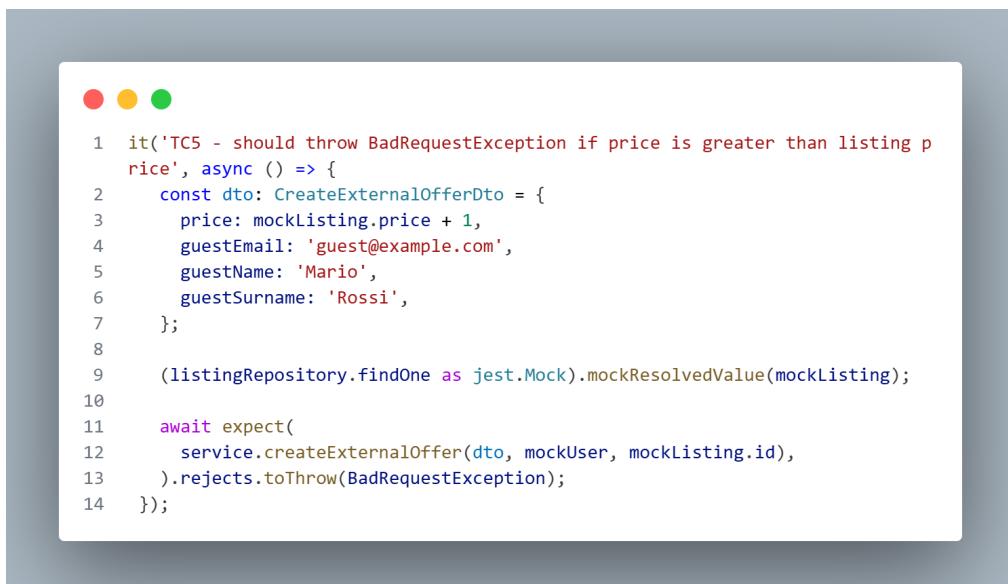
```

- Caso 4 (listingId: CE1, price: CE2): il prezzo inserito è minore o uguale a zero ⇒ viene sollevata una `BadRequestException`.



```
● ● ●  
1 it('TC4 - should throw BadRequestException if price is less than or equal to  
0', async () => {  
2   const dto: CreateExternalOfferDto = {  
3     price: -1,  
4     guestEmail: 'guest@example.com',  
5     guestName: 'Mario',  
6     guestSurname: 'Rossi',  
7   };  
8  
9   (listingRepository.findOne as jest.Mock).mockResolvedValue(mockListing);  
10  
11  await expect(  
12    service.createExternalOffer(dto, mockUser, mockListing.id),  
13  ).rejects.toThrow(BadRequestException);  
14});
```

- Caso 5 (listingId: CE1, price: CE3): il prezzo inserito è superiore a quello dell'annuncio ⇒ viene sollevata una `BadRequestException`.



```
● ● ●  
1 it('TC5 - should throw BadRequestException if price is greater than listing p  
rice', async () => {  
2   const dto: CreateExternalOfferDto = {  
3     price: mockListing.price + 1,  
4     guestEmail: 'guest@example.com',  
5     guestName: 'Mario',  
6     guestSurname: 'Rossi',  
7   };  
8  
9   (listingRepository.findOne as jest.Mock).mockResolvedValue(mockListing);  
10  
11  await expect(  
12    service.createExternalOffer(dto, mockUser, mockListing.id),  
13  ).rejects.toThrow(BadRequestException);  
14});
```

Esiti dei test

```
PASS  src/property_offer/property-offer.spec.ts
OfferService - createExternalOffer
  ✓ TC1 - should create an external offer successfully (23 ms)
  ✓ TC2 - should throw NotFoundException if listing does not exist (15 ms)
  ✓ TC3 - should throw UnauthorizedException if user is not authorized (4 ms)
  ✓ TC4 - should throw BadRequestException if price is less than or equal to 0 (3 ms)
  ✓ TC5 - should throw BadRequestException if price is greater than listing price (5 ms)
OfferService - getLatestOffersByListingId
  ✓ TC1 - should return latest offer for each client on success (4 ms)
  ✓ TC2 - should throw UnauthorizedException if listing not found (2 ms)
  ✓ TC3 - should throw UnauthorizedException if user not authorized (3 ms)

Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        2.48 s
Ran all test suites matching /property-offer.spec.ts/i.
```

6.1.3 Metodo `getLatestOffersByListingId`

Il metodo `getLatestOffersByListingId(listingId, user)` del servizio `OfferService` consente di recuperare tutti i clienti che hanno effettuato un'offerta ad un specifico annuncio (`listingId`), insieme all'offerta più recente. Viene inoltre verificato che l'utente loggato abbia i permessi per accedere alle offerte dell'annuncio specificato.

Il metodo esegue le seguenti operazioni:

- controllo dell'esistenza dell'annuncio tramite repository;
- verifica dei permessi di accesso dell'utente sull'annuncio;
- recupero dei clienti insieme alla loro offerta più recente

Classi di equivalenza

Sono state individuate le seguenti classi di equivalenza per i parametri e le condizioni operative del metodo:

- **listingId:**
 - **CE1 valida:** l'annuncio esiste nel sistema;
 - **CE2 non valida:** annuncio inesistente \Rightarrow `UnauthorizedException`.
- **user:**
 - Questo parametro rappresenta l'utente autenticato. Non essendo un tipo primitivo, non si presta a una suddivisione in classi di equivalenza tradizionali.
 - È stato comunque necessario verificare con casi di test se il sistema, quando l'utente non è autorizzato, permette o meno di creare offerte per l'annuncio indicato.

Strategia di test

È stata adottata la tecnica **N-WECT** (Testing con Classi di Equivalenza Deboli), selezionando un insieme minimo di test che copra le combinazioni più significative:

- **Caso 1 (listingId: CE1):** annuncio esistente, utente autorizzato, nessuna offerta ⇒ ritorna array vuoto.



```

1  it('TC1 - should return latest offer for each client on success', async () => {
2    // MOCK
3    (listingRepository.findOne as jest.Mock).mockResolvedValue(mockListing);
4
5    const mockQueryBuilder = {
6      distinctOn: jest.fn().mockReturnThis(),
7      innerJoinAndSelect: jest.fn().mockReturnThis(),
8      leftJoinAndSelect: jest.fn().mockReturnThis(),
9      where: jest.fn().mockReturnThis(),
10     andWhere: jest.fn().mockReturnThis(),
11     orderBy: jest.fn().mockReturnThis(),
12     addOrderBy: jest.fn().mockReturnThis(),
13     getMany: jest.fn().mockResolvedValue([
14       {
15         id: 'offer-1',
16         price: 1000,
17         date: new Date('2023-01-01'),
18         state: OfferState.PENDING,
19         madeByUser: true,
20         client: {
21           userId: 'user-client-1',
22           user: {
23             name: 'Mario',
24             surname: 'Rossi',
25             email: 'mario@example.com',
26             phone: '123456789',
27           },
28         },
29       },
30     ]),
31   };
32
33   (offerRepository.createQueryBuilder as jest.Mock).mockReturnValue(
34     mockQueryBuilder,
35   );
36
37   // ACT
38   const result = await service.getLatestOffersByListingId(
39     mockListing.id,
40     mockUser,
41   );
42
43   // ASSERT
44   expect(result).toEqual([
45     {
46       userId: 'user-client-1',
47       name: 'Mario',
48       surname: 'Rossi',
49       email: 'mario@example.com',
50       phone: '123456789',
51       lastOffer: {
52         id: 'offer-1',
53         price: 1000,
54         date: new Date('2023-01-01'),
55         state: OfferState.PENDING,
56         madeByUser: true,
57       },
58     },
59   ]);
60 });

```

- Caso 2 (listingId: CE2): annuncio non trovato \Rightarrow UnauthorizedException.



```

● ● ●

1 it('TC2 - should throw UnauthorizedException if listing not found', async () => {
2   // MOCK
3   (listingRepository.findOne as jest.Mock).mockResolvedValue(null);
4
5   // ACT && ASSERT
6   await expect(
7     service.getLatestOffersByListingId('invalid-id', mockUser),
8   ).rejects.toThrow(UnauthorizedException);
9 });

```

- Caso 3 (listingId: _): utente non autorizzato \Rightarrow UnauthorizedException



```

● ● ●

1 it('TC3 - should throw UnauthorizedException if user not authorized', async () => {
2   // MOCK
3   (listingRepository.findOne as jest.Mock).mockResolvedValue(mockListing);
4
5   jest.spyOn(service as any, 'checkAuthorization').mockImplementation(() => {
6     throw new UnauthorizedException();
7   });
8
9   // ACT && ASSERT
10  await expect(
11    service.getLatestOffersByListingId(mockListing.id, mockUser),
12  ).rejects.toThrow(UnauthorizedException);
13 });

```

Esiti dei test

```

PASS  src/property_offer/property-offer.spec.ts
OfferService - createExternalOffer
  ✓ TC1 - should create an external offer successfully (23 ms)
  ✓ TC2 - should throw NotFoundException if listing does not exist (15 ms)
  ✓ TC3 - should throw UnauthorizedException if user is not authorized (4 ms)
  ✓ TC4 - should throw BadRequestException if price is less than or equal to 0 (3 ms)
  ✓ TC5 - should throw BadRequestException if price is greater than listing price (5 ms)
OfferService - getLatestOffersByListingId
  ✓ TC1 - should return latest offer for each client on success (4 ms)
  ✓ TC2 - should throw UnauthorizedException if listing not found (2 ms)
  ✓ TC3 - should throw UnauthorizedException if user not authorized (3 ms)

Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:  0 total
Time:        2.48 s
Ran all test suites matching /property-offer.spec.ts/i.

```

6.1.4 Metodo `deleteAgentById`

Il metodo `deleteAgentById(agentId, agencyId)` del servizio `AgencyService` consente di eliminare un agente dal sistema, a condizione che l'agente esista e sia effettivamente associato all'agenzia identificata da `agencyId`.

Il metodo esegue le seguenti operazioni:

- verifica dell'esistenza dell'utente identificato da `agentId`;
- verifica della presenza dell'agente associato all'utente;
- controllo che l'agente appartenga all'agenzia con identificativo `agencyId`;
- rimozione dell'utente tramite il repository.

Classi di equivalenza

Sono state identificate le seguenti classi di equivalenza per i parametri e le condizioni del metodo:

- `agentId`:
 - **CE1 valida**: utente esistente nel sistema;
 - **CE2 non valida**: utente inesistente ⇒ `NotFoundException`.
- `agencyId`:
 - **CE1 valida**: corrisponde a quella dell'agente;
 - **CE2 non valida**: non corrisponde a quella dell'agente ⇒ `UnauthorizedException`.

Strategia di test

È stata adottata la tecnica **N-WECT** (Testing con Classi di Equivalenza Deboli), selezionando un insieme minimo di test che copre tutte le combinazioni significative:

- Caso 1 (agentId: CE1, agencyId: CE2): l'utente e l'agente esistono, e l'agente appartiene alla giusta agenzia ⇒ eliminazione riuscita.

```
● ● ●

1 it('TC1 - It should properly delete an agent', async () => {
2   // MOCK
3   (userRepository.findOneBy as jest.Mock).mockResolvedValue(user);
4   (agentRepository.findOne as jest.Mock).mockResolvedValue(agent);
5
6   // ACT
7   const result = await agencyService.deleteAgentById(agent.userId, agency.id);
8
9   // ASSERT
10  expect(userRepository.delete).toHaveBeenCalledWith(agent.userId);
11  expect(result).toEqual({ message: 'Agent delete successfully' });
12});
```

- Caso 2 (agentId: CE2, _): l'utente non esiste ⇒ NotFoundException.

```
● ● ●

1 it('TC2 - Should throw NotFoundException if user does not exist', async () => {
2   // MOCK
3   (userRepository.findOneBy as jest.Mock).mockResolvedValue(null);
4
5   // ACT && ASSERT
6   await expect(
7     agencyService.deleteAgentById('invalid-agent', agency.id),
8   ).rejects.toThrow(NotFoundException);
9});
```

- **Caso 3 (agentId: CE2, agencyId: _):** l'utente esiste ma non è associato a un agente ⇒ `NotFoundException`.



```

1 it('TC3 - Should throw NotFoundException if agent does not exist', async () => {
2   // MOCK
3   (userRepository.findOneBy as jest.Mock).mockResolvedValue(user);
4   (agentRepository.findOne as jest.Mock).mockResolvedValue(null);
5
6   // ACT && ASSERT
7   await expect(
8     agencyService.deleteAgentById(agent.userId, agency.id),
9   ).rejects.toThrow(NotFoundException);
10 });

```

- **Caso 4 (agentId: CE1, agencyId: CE2):** l'agente esiste ma è associato a un'altra agenzia ⇒ `UnauthorizedException`.



```

1 it('TC4 - Should throw UnauthorizedException if the agency does not correspond', async () => {
2   // MOCK
3   (userRepository.findOneBy as jest.Mock).mockResolvedValue(user);
4   (agentRepository.findOne as jest.Mock).mockResolvedValue({
5     ...agent,
6     agency: { id: 'agency-2' },
7   });
8
9   // ACT && ASSERT
10  await expect(
11    agencyService.deleteAgentById(agent.userId, agency.id),
12  ).rejects.toThrow(UnauthorizedException);
13 });
14 });

```

Esiti dei test

```

PASS  src/agency/agency-service.spec.ts
AgencyService - deleteAgentById
  ✓ TC1 - It should properly delete an agent (7 ms)
  ✓ TC2 - Should throw NotFoundException if user does not exist (7 ms)
  ✓ TC3 - Should throw NotFoundException if agent does not exist (2 ms)
  ✓ TC4 - Should throw UnauthorizedException if the agency does not correspond (2 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        2.03 s
Ran all test suites matching /agency-service.spec.ts/i.

```

6.2 Expert Review / Ispezione Esperta

Individuare difetti di usabilità e incoerenze nell'interfaccia prima della fase di test con utenti finali, utilizzando una checklist sistematica basata su principi noti di usabilità (es. Nielsen).

Metodologia

- **Checklist definita:** 16 criteri suddivisi in 5 categorie principali:
 - *Visibilità dello stato del sistema:* presenza di feedback chiari e immediati dopo ogni azione (es. caricamento annunci, conferma offerte).
 - *Corrispondenza tra sistema e mondo reale:* uso di terminologia comprensibile dagli utenti (es. *offerta*, *immobile*, *profilo*).
 - *Controllo e libertà dell'utente:* possibilità di annullare un'azione (es. tornare indietro, disconnettersi facilmente).
 - *Consistenza e standard:* uniformità nella posizione dei pulsanti, icone coerenti in tutte le schermate.
 - *Prevenzione degli errori:* messaggi d'errore descrittivi, disattivazione di pulsanti non utilizzabili.
- **Strumenti:** l'app Android è stata installata su dispositivi fisici (Xiaomi Mi 10T Pro, Samsung S23, OnePlus 9) e testata anche tramite emulatori Android Studio.
- **Chi ha effettuato la review:** due valutatori indipendenti con competenze in UI/UX design e sviluppo mobile.

6.2.1 Risultati principali

- **Punti di forza:**
 - Interfaccia coerente e aderente al Material Design 3.
 - Navigazione intuitiva tramite bottom bar adattiva.
 - Coerenza tra le varie schermate.
- **Criticità rilevate:**
 - Pulsanti troppo vicini su schermi molto piccoli.
 - Assenza di tooltip per icone meno intuitive.
 - Mancanza di notifiche push automatiche.

Le criticità sono state documentate e in parte corrette prima della fase di sperimentazione con utenti reali.

6.3 Esperimento con Utenti Reali

Valutare l'usabilità del sistema attraverso l'interazione con utenti rappresentativi del pubblico finale, misurando quantitativamente e qualitativamente la facilità d'uso e la soddisfazione.

Partecipanti

- Totale: 6 soggetti
- Età: 21–35 anni
- Profilo: studenti universitari e giovani professionisti nel settore informatico.
- Ruoli testati: 6 utenti "cliente", 3 "agente", 3 "gestore/admin" (per simulare tutte le possibili interazioni).

Procedura

1. Sessione guidata con scenario d'uso per ogni ruolo:
 - Cliente: login, ricerca immobile, invio offerta, modifica preferenze profilo.
 - Agente: accesso, pubblicazione di un nuovo annuncio, gestione offerte ricevute.
 - Gestore: login, visualizzazione profilo agenzia, aggiunta nuovo agente.
2. Ogni utente ha eseguito 5 compiti in autonomia (max 15 minuti).
3. Monitoraggio del tempo di completamento, errori commessi, richieste d'aiuto.
4. Al termine: compilazione survey.

6.3.1 Metriche raccolte

- Tempo medio per task: 42 secondi
- Task completati con successo: 91
- Errori medi per utente: 1.2
- Richieste d'aiuto: 0.5 per sessione
- Livello di soddisfazione generale (su scala 1-5): 4.4

6.4 Survey Post-Esperimento

Struttura del questionario

Hai trovato facilmente le funzionalità principali dell'app?

(Sì / No / In parte)

Quanto è stato facile orientarti all'interno dell'app?

(1 – Per niente 2 3 4 5 – Molto facile)

Hai mai avuto difficoltà a tornare alla schermata principale?

(Sì / No / Non saprei)

Hai trovato tutte le funzioni che ti aspettavi?

(Sì / No)

Com'è il design dell'app secondo te?

(Intuitivo / Normale / Poco curato / Confuso)

Hai avuto difficoltà a leggere testi, pulsanti o elementi grafici?

(Sì / No)

Hai riscontrato problemi di visualizzazione su schermo?

(Sì / No)

Quanto è stata reattiva l'app durante l'utilizzo?

(1 – 5)

Hai riscontrato crash o blocchi?

(Sì / No)

Valuta la tua esperienza complessiva con l'app

(1 – 5)

Consiglieresti questa app ad altri utenti?

(1 – 5)

6.4.1 Sintesi dei risultati

- Il 100% dei partecipanti ha completato almeno 4 su 5 task.
- Le domande 1–6 hanno ottenuto una media tra 4.1 e 4.7.
- Dai commenti aperti è emersa l'importanza di:
 - un onboarding iniziale per spiegare le icone meno note;
 - maggiore spazio tra pulsanti nelle schermate con molti dati;
 - possibilità di ricevere notifica anche via email.