Università di Federico II

Ingegneria del Softaware

DietiEstates25

Autori
Antonio Legnante
Vincenzo Noviello

Professori
Prof. S. Di martino
Prof. L.L.L. Starace

Gennaio 2026

Indice

Capitolo 1

Analisi dei requisiti

1.1 Glossario

Di seguito sono elencati i concetti fondamentali del progetto

Termine	**Descrizione**
Agente Immobiliare	Professionista appartenente a un'agenzia immobiliare che utilizza la piattaforma per pubblicare annunci, gestire i contatti e comunicare con gli utenti.
Utente Registrato	Persona che crea un account sulla piattaforma e può consultare gli annunci, contattare gli agenti immobiliari e avviare conversazioni.
Agenzia Immobiliare	Organizzazione composta da uno o più agenti immobiliari, responsabile della gestione e pubblicazione degli immobili sulla piattaforma.
Immobile	Proprietà immobiliare pubblicata sulla piattaforma (appartamento, casa, locale commerciale, ecc.) con descrizione, immagini e caratteristiche tecniche.
Annuncio Immobiliare	Scheda informativa di un immobile pubblicata da un agente, contenente dettagli come prezzo, descrizione, foto e disponibilità.
Conversazione / Chat	Canale di comunicazione interna tra utente registrato e agente immobiliare, utilizzato per richieste di informazioni o appuntamenti.
Autenticazione	Processo tramite cui un utente o un agente immobiliare accede alla piattaforma utilizzando le proprie credenziali.
Registrazione	Procedura con cui un nuovo utente crea un account sulla piattaforma inserendo i propri dati personali.

1.2 Casi d'uso

1.2.1 Descrizione dei requisiti funzionali

Requisito 1: Gestione Utenti e Autenticazione Attori coinvolti

- Amministratore
- Agente immobiliare
- Utente

Descrizione:
Il sistema deve permettere la registrazione di nuovi utenti (clienti) e di agenti immobiliari. Tutti gli attori devono essere in grado di effettuare il login con credenziali sicure. L'Amministratore deve poter modificare le credenziali di accesso predefinite e gestire gli account degli agenti.

Requisito 2: Inserimento di Inserzioni Immobiliari Attori coinvolti

- Agente immobiliare

Descrizione:
Gli agenti immobiliari possono caricare nuove inserzioni di immobili, complete di dettagli quali foto, descrizione, prezzo, dimensioni, indirizzo, numero di stanze, classe energetica, ecc. Le inserzioni devono essere classificate per tipologia: vendita o affitto.

Requisito 3: Ricerca Avanzata di Immobili Attori coinvolti

- Utente
- Geopify

Descrizione:
Il sistema deve permettere la ricerca avanzata di immobili tramite filtri multipli: tipologia, prezzo, posizione geografica (con supporto mappa), numero di stanze, classe energetica, ecc. La ricerca deve essere efficiente e visualizzare i risultati anche tramite mappa interattiva.

Requisito 4: Gestione delle Offerte sugli Immobili Attori coinvolti

- Agente immobiliare
- Utente registrato

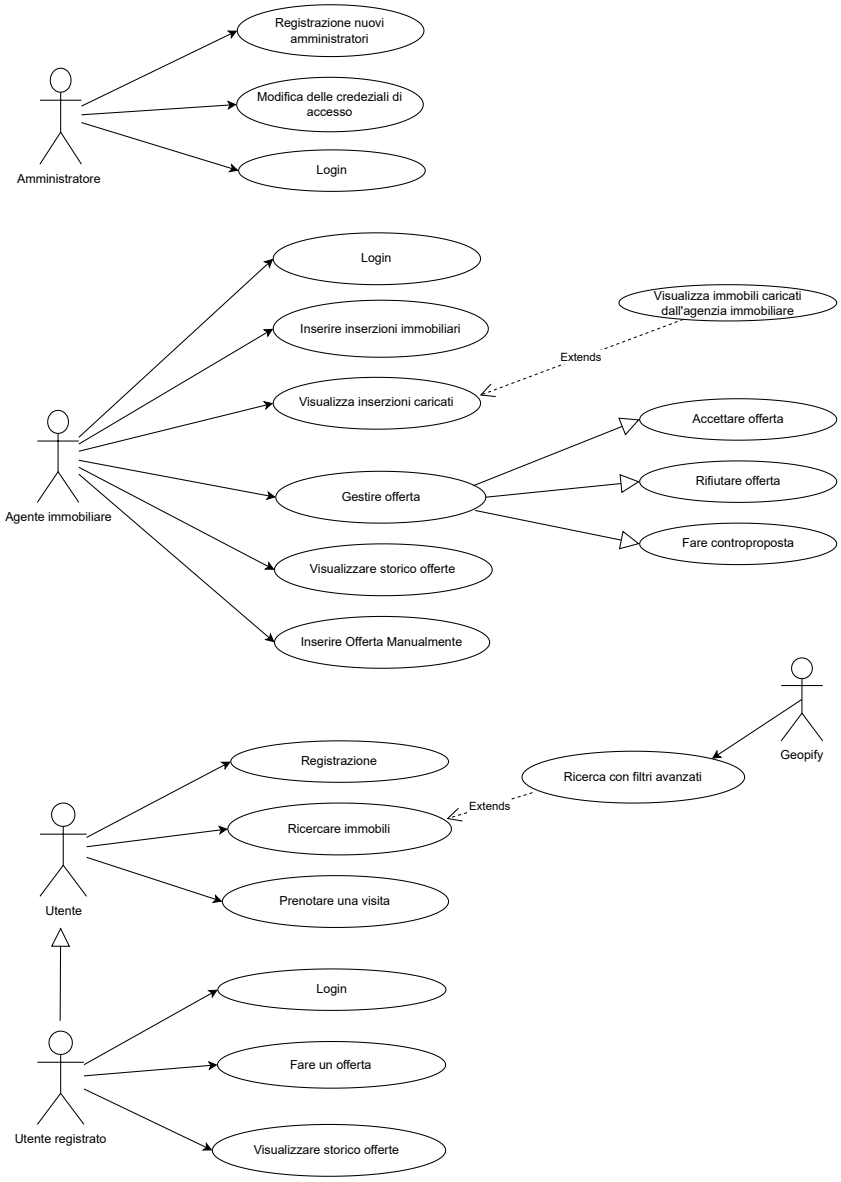
Descrizione:
Gli utenti registrati possono fare offerte per immobili specificando un prezzo. Gli agenti possono accettare, rifiutare o fare controproposte. Deve essere possibile visualizzare uno s torico delle offerte sia per l'utente che per l'agente. Gli agenti possono inserire manualmente offerte ricevute esternamente al sistema.

Requisito 5: Integrazione con Servizi Esterni (Geoapify) Attori coinvolti

- Geopify

Descrizione:
All'atto della creazione di un'inserzione immobiliare, il sistema deve interrogare il servizio esterno Geoapify per verificare la presenza di scuole, parchi o trasporto pubblico nelle vicinanze dell'immobile. Se presenti, verranno mostrati appositi indicatori ("Vicino a scuole", ecc.).

1.2.2 Diagramma dei casi d'uso



1.3 Requisiti non funzionali e di dominio

I requisiti non funzionali descrivono vincoli di qualità applicabili all'intero sistema. I requisiti di dominio derivano dal contesto reale in cui il sistema opera: leggi, regolamenti, abitudini, regole del settore. Sono di seguito elencati i requisiti non funzionali e di dominio

Usabilità

- Un utente non esperto deve poter registrare un immobile senza consultare documentazione esterna.
- L'interfaccia deve essere accessibile tramite browser web moderni (Chrome, Firefox).

Sicurezza

- L'accesso al sistema deve avvenire tramite autenticazione con credenziali.
- Le password devono essere memorizzate in forma cifrata (hash).
- Un utente può accedere solo ai dati per cui è autorizzato

Prestazioni

- Il sistema deve rispondere alle operazioni principali (login, visualizzazione immobili, registrazione contratto) entro 2 secondi nel 95% dei casi.
- Il sistema deve supportare almeno 100 utenti contemporanei senza degrado significativo delle prestazioni.

Manutenibilità

- Il sistema deve essere progettato in modo modulare per facilitare l'aggiunta di nuove funzionalità.
- Le componenti devono essere debolmente accoppiate per consentire modifiche locali senza impatti sull'intero sistema.
- Il codice deve essere organizzato secondo un'architettura che favorisca la manutenibilità.

Aspetti economici

- Il canone di locazione/ prezzo di vendità deve essere espresso in euro.

Gestione immobili

- Ogni immobile deve essere identificato in modo univoco all'interno del sistema.

Aggiungiamo una nota anche legata ai vincoli e assunzioni di progetto; Specifichiamo che questi non sono requisiti non funzionali puri

Vincoli di progetto

- Il sistema deve essere sviluppato utilizzando tecnologie note al team.
- L'architettura deve rimanere semplice per facilitare manutenzione e sviluppo.

1.4 Personas

Sono descritti di seguito i tipici utenti dell'applicazione

Nome: Lidia Pascal

Demografia

Genere: Femmina

Luogo: San Giuseppe Vesuviano

Età: 25

Stato coniugale: Nubile

Titolo: Studentessa

Tratti personali: Ambiziosa, leale

Background

Lidia è un attenta e dedita studentessa di scienze politiche. È una ragazza estroversa e piena di risorse che non vede l'ora di completare il percorso di studi per potersi dedicare a tempo pieno nel lavoro.

GOAL

- Avere più tempo da dedicare allo studio.
- Migliorare il rapporto vità/studio.
- Migliorare le proprie relazioni interpersonali.

INTERESSI

- Fisica: Le piace la natura e legge tanto su questo argomento
- Sport: Le piace mantenere una linea perfetta e si esercita spesso anche a casa
- Serie Tv: Adora guardare Bridgaton e prendere tè e biscotti mentre guarda la serie

Nome: Romeo Giuliettini

Demografia

Genere: Maschio

Luogo: Roma

Età: 35

Stato coniugale: Celibe

Titolo: Software Engineer

Tratti personali: Curioso, Testardo

Background

Romeo è un lavoratore attento e preciso. È felice della vità che ha e sta cercando una stabilità. Ha fatto un percorso di crescità personale che lo ha portato ad avere un ottima gestione del suo tempo, permettendogli di riuscire a dedicare tempo per tutti i suoi hobby e impegni.

GOAL

- Cercare un luogo con cui convivere con la compagna
- Migliorare il rapporto vità/lavoro.

INTERESSI

- Videogiochi: Gli piace giocare con i videogiochi e rimane costantemente aggiornato sulle ultime notivà.
- Sport: Correre è essenziale per Romeo, che ha una routine con orari molto stretti.
- Film: Grande cinefilo, guarda spesso film.

Nome: Mario Rossi

Demografia

Genere:	Maschio
Luogo:	Milano
Età:	47
Stato coniugale:	Sposato
Titolo:	Commercialista
Tratti personali:	Responsabile, affidabile

Background

Dopo aver conseguito il titolo di studi, Mario si è dedicato a tempo pieno al lavoro e alla costruzione di una famiglia. Spesso, secondo lui purtroppo, la città lo assorbe troppo, e sempre più cerca modi per evader il caos quotidiano. È felicemente sposato, e il suo pensiero e la sua attenzione sono rivolti ai figli, nel pieno della loro adolescenza.

GOAL

- Trovare un luogo per fuggire dallo stress della città
- Trovare un luogo che permetta anche ai suoi figli di poter soggiornare con i genitori

INTERESSI

- Politica: Segue con accesso interesse al dibattito pubblico
- Natura: Adora I paesaggi naturalistici
- Musica: Collezionista di dischi della sua adolescenza

Nome: Roberta Franchini

Demografia

Genere: Femmina

Luogo: Gallipoli

Età: 42

Stato coniugale: Sposata

Titolo: CEO Agenzia Immobiliare

Tratti personali: Puntigliosa, affabile

Background

Roberta è un enconiabile lavoratrice nel settore immobiliare. Sposata e con 3 figli, è una persona poliedrica e sempre pronta a stupire in positive chiunque la incontri.

GOAL

- Individuare il modo di raggiungere più persone possibili per mostrare le proprietà per venderle o affittarle

INTERESSI

- Immobili: Le piace il mondo immobiliare
- Sport: Adora praticare e seguire equitazione
- Salute e benessere: È sempre informata sulle ultime uscite per i prodotti della cura del corpo

1.5 Diagrammi di cockburn

1.6 Mock-up

Capitolo 2

Analisi di sistema

2.1 Architettura di sistema

L'applicazione adotta un'architettura client–server organizzata secondo il modello a 3 livelli (3-tier), strutturata come Single Page Application (SPA) e progettata seguendo il pattern MVC. Al primo accesso il server invia al client l'intera applicazione Web (HTML, CSS, JavaScript). Da quel momento l'applicazione viene eseguita interamente nel browser, e il server entra in gioco soltanto per fornire o aggiornare i dati tramite API RESTful, che restituiscono JSON. Il rendering dell'interfaccia utente avviene lato client (Client-Side Rendering).

2.1.1 Motivazioni architetturali

La scelta di un'architettura a 3 livelli è stata determinata dalla necessità di separare nettamente le responsabilità tra presentazione, logica applicativa e persistenza dei dati. Questa separazione consente una maggiore manutenibilità del codice, facilita l'evoluzione indipendente di ciascun livello e permette una distribuzione scalabile delle risorse computazionali. L'adozione del pattern Single Page Application risponde all'esigenza di offrire un'esperienza utente fluida e reattiva, simile a quella delle applicazioni desktop native, eliminando i tempi di attesa dovuti al ricaricamento completo delle pagine web durante la navigazione.

2.2 Architettura del Server

2.2.1 Framework e Tecnologie del Backend

Il backend dell'applicazione è implementato utilizzando Spring Boot 3, un framework Java che rappresenta l'evoluzione della piattaforma Spring, ampiamente riconosciuta nell'ecosistema enterprise per la sua robustezza, maturità e completezza. Spring Boot 3 introduce numerosi miglioramenti rispetto alle versioni precedenti, tra cui il supporto nativo per Java 17 e successive versioni LTS (Long Term Support), l'integrazione con Jakarta EE 9 che sostituisce le specifiche Java EE, e ottimizzazioni significative nelle prestazioni e nel consumo di memoria.

Spring Boot semplifica drasticamente la configurazione e il deployment delle applicazioni Spring tradizionali attraverso il principio della "convention over configuration". Questo approccio permette agli sviluppatori di concentrarsi sulla logica di business piuttosto che sulla configurazione infrastrutturale. Il framework fornisce starter dependencies pre-configurate che aggregano le dipendenze comuni necessarie per scenari specifici, come lo sviluppo di API REST, l'integrazione con database relazionali, la gestione della sicurezza.

Spring Boot 3 include un server applicativo embedded, tipicamente Tomcat, Jetty o Undertow, che elimina la necessità di deployare l'applicazione su un application server esterno. Questo semplifica notevolmente il processo di distribuzione e rende l'applicazione facilmente containerizzabile mediante Docker o altre tecnologie di containerizzazione. L'applicazione può essere eseguita come un semplice JAR eseguibile, contenente tutte le dipendenze necessarie, incluso il server web embedded.

2.2.2 Architettura MVC del Backend

Il lato server adotta un'architettura ispirata al pattern Model-View-Controller, ma limitata alle componenti Model e Controller, poiché la View è completamente delegata al client. Questa variante dell'architettura MVC è particolarmente adatta per applicazioni che espongono servizi RESTful e che separano il frontend dal backend.

2.2.3 Il Model

Il Model comprende due componenti fondamentali che lavorano sinergicamente per gestire i dati e la logica di business dell'applicazione.

- **Entità (Entities):** rappresentano le strutture dati persistenti dell'applicazione e costituiscono il nucleo del livello dati. Nel contesto di un'applicazione per la gestione di immobili, le entità principali includono Immobile, Utente, Chat, Messaggio e altre strutture dati necessarie per modellare il dominio applicativo. Ogni entità è mappata su una tabella corrispondente nel database relazionale PostgreSQL attraverso l'uso di annotazioni JPA (Jakarta Persistence API), che definiscono la corrispondenza tra gli attributi della classe Java e le colonne della tabella database.
  Le entità incapsulano non solo i dati ma anche le relazioni tra di esse. Per esempio, un Immobile può avere una relazione uno-a-molti con le Immagini associate, una relazione molti-a-uno con l'agente immobiliare, e una relazione molti-a-molti con gli Utenti interessati. Queste relazioni sono gestite in modo trasparente dall'ORM, che si occupa della generazione delle query SQL appropriate e della gestione della consistenza referenziale.

- **Logica di Business (Servizi):** implementano le regole applicative e la logica di business complessa che governa il comportamento dell'applicazione. Ogni servizio è responsabile di un'area funzionale specifica, come la gestione degli immobili (ImmobileService), l'autenticazione e gestione degli utenti (UtenteService, AuthService), la gestione delle comunicazioni (ChatService, MessaggioService), e così via.
  I servizi fungono da ponte tra i Controller e il database, incapsulando la logica applicativa in modo che i controller possano rimanere leggeri e focalizzati esclusivamente sulla gestione delle richieste HTTP. I servizi coordinano le operazioni che coinvolgono multiple entità, gestiscono le transazioni database attraverso l'annotazione @Transactional di Spring, applicano regole di validazione del business e implementano algoritmi complessi come la ricerca avanzata di immobili con filtri multipli, il calcolo di statistiche e metriche, la generazione di notifiche e molto altro.

2.3 Controller

Il Controller rappresenta il punto di ingresso delle richieste HTTP provenienti dai client. Ogni controller è annotato con @RestController, un'annotazione Spring che combina @Controller e @ResponseBody, indicando che i metodi del controller restituiranno direttamente oggetti serializzati in JSON piuttosto che nomi di view da renderizzare.
I controller ricevono le richieste HTTP dai client attraverso endpoint REST chiaramente definiti, interpretano i parametri e i payload delle richieste, invocano i metodi appropriati dei servizi di business, gestiscono le eccezioni e gli errori, e restituiscono risposte HTTP con payload JSON e status code appropriati.

I controller non contengono logica di business complessa ma si limitano a orchestrare le chiamate ai servizi e a trasformare le risposte in formato appropriato per i client.
Spring Boot fornisce il DispatcherServlet, che implementa il pattern Front Controller, intercettando tutte le richieste HTTP in ingresso e delegandole al controller appropriato basandosi sul mapping degli URL definiti attraverso annotazioni come @GetMapping, @PostMapping, @PutMapping, @DeleteMapping e @RequestMapping. Questo meccanismo centralizzato di routing permette una gestione uniforme di aspetti cross-cutting come la gestione degli errori, la validazione, la serializzazione/deserializzazione JSON e l'applicazione di filtri e interceptor.

2.4 API RESTful

Il backend espone esclusivamente servizi RESTful, aderendo ai principi dell'architettura REST (Representational State Transfer). Le API sono progettate seguendo le best practices REST, utilizzando i verbi HTTP in modo semanticamente corretto: GET per recuperare risorse, POST per creare nuove risorse, PUT per aggiornare risorse esistenti in modo completo, PATCH per aggiornamenti parziali, e DELETE per rimuovere risorse.
Gli endpoint seguono una struttura gerarchica e intuitiva che riflette le relazioni tra le risorse. Per esempio, /api/immobili per accedere alla collezione di immobili, /api/immobili/id per accedere a un immobile specifico, /api/immobili/id/immagini per accedere alle immagini associate a un immobile. Gli status code HTTP sono utilizzati in modo appropriato per comunicare l'esito delle operazioni: 200 OK per operazioni riuscite, 201 Created per risorse create con successo, 204 No Content per operazioni che non restituiscono dati, 400 Bad Request per errori di validazione, 401 Unauthorized per problemi di autenticazione, 403 Forbidden per problemi di autorizzazione, 404 Not Found per risorse non trovate, e 500 Internal Server Error per errori del server.
Le risposte delle API sono sempre in formato JSON, scelto per la sua leggerezza, leggibilità umana e ampio supporto in tutti i linguaggi di programmazione moderni. Spring Boot utilizza Jackson come libreria di serializzazione/deserializzazione JSON, configurabile attraverso ObjectMapper per gestire casi specifici come la formattazione delle date, l'esclusione di proprietà null, la gestione di riferimenti circolari e la customizzazione della serializzazione di tipi complessi.

2.5 Sisitema di persistenza dei dati

2.5.1 PostgreSQl come Database Relazionale

PostgreSQL offre conformità completa agli standard SQL e supporta funzionalità avanzate come transazioni ACID (Atomicity, Consistency, Isolation, Durability), che garantiscono l'integrità dei dati anche in presenza di fallimenti di sistema o operazioni concorrenti. Il database supporta vincoli referenziali complessi, trigger, stored procedure, viste materializzate, indici di vari tipi (B-tree, Hash, GiST, GIN) per ottimizzare le query, e funzionalità di full-text search integrate.
PostgreSQL eccelle nella gestione di carichi di lavoro complessi con query elaborate che coinvolgono join multipli, aggregazioni, subquery correlate e common table expressions (CTE). Supporta nativamente tipi di dato avanzati come JSON e JSONB per dati semi-strutturati, array, tipi geometrici per dati spaziali, range types, e permette la definizione di tipi di dato custom. Questa flessibilità lo rende ideale per applicazioni che necessitano di modellare domini complessi con requisiti eterogenei.
Il database implementa il controllo della concorrenza multi-versione (MVCC - Multi-Version Concurrency Control), che permette a lettori e scrittori di operare contemporaneamente sulla stessa tabella senza bloccarsi a vicenda, massimizzando la concorrenza e le prestazioni in scenari multi-utente. PostgreSQL gestisce automaticamente il vacuum dei dati vecchi e l'ottimizzazione delle tabelle per mantenere prestazioni elevate nel tempo.

2.5.2 Integrazione con SpringBoot tramite JPA/Hibernate

L'integrazione tra Spring Boot e PostgreSQL avviene attraverso l'uso di JPA (Jakarta Persistence API) con Hibernate come implementazione dell'ORM (Object-Relational Mapping). Questo stack tecnologico assicura una netta separazione tra la logica applicativa e il database relazionale, permettendo agli sviluppatori di lavorare con oggetti Java piuttosto che con query SQL dirette.
Spring Data JPA, parte dell'ecosistema Spring, fornisce un ulteriore livello di astrazione sopra JPA, offrendo repository interface che eliminano la necessità di scrivere implementazioni boilerplate per operazioni CRUD comuni. Attraverso la definizione di semplici interfacce che estendono JpaRepository o CrudRepository, Spring Data JPA genera automaticamente le implementazioni concrete a runtime, includendo metodi per salvare, aggiornare, eliminare e recuperare entità, oltre a supportare la derivazione di query dai nomi dei metodi.
Hibernate si occupa della traduzione delle operazioni sugli oggetti Java in query SQL ottimizzate per PostgreSQL, gestisce il caching di primo e secondo livello per ridurre gli accessi al database, implementa lazy loading e eager loading delle relazioni tra entità, e gestisce automaticamente la generazione e l'aggiornamento dello schema database in fase di sviluppo attraverso la proprietà **spring.jpa.hibernate.ddl-auto.**

L'ORM consente di modellare le entità come classi Java annotate, gestire le relazioni (uno-a-uno, uno-a-molti, molti-a-molti) in modo trasparente attraverso annotazioni come @OneToMany, @ManyToOne, @ManyToMany, e semplificare le operazioni CRUD attraverso metodi ad alto livello. Le query possono essere scritte usando JPQL (Java Persistence Query Language), un linguaggio di query orientato agli oggetti che astrae dal SQL specifico del database, o attraverso la Criteria API per query dinamiche costruite programmaticamente.

2.6 MiniIO per la gestione delle immagini

Il sistema di persistenza delle immagini è gestito tramite MinIO, un object storage server open-source ad alte prestazioni, compatibile con l'API Amazon S3. MinIO rappresenta una soluzione moderna ed efficiente per la gestione di file binari di grandi dimensioni, offrendo scalabilità, affidabilità e prestazioni superiori rispetto alla memorizzazione diretta nel database relazionale.

2.6.1 Vantaggi di MiniIO

La separazione della gestione delle immagini dal database principale garantisce numerosi benefici architetturali e operativi. Le prestazioni del sistema migliorano significativamente poiché il database relazionale non deve gestire il carico di lettura e scrittura di file binari di grandi dimensioni, che potrebbero variare da poche centinaia di kilobyte a diversi megabyte per immagine. Questo riduce drasticamente le dimensioni del database, velocizza le operazioni di backup e ripristino, e ottimizza l'utilizzo della memoria cache del database per i dati realmente relazionali.
MinIO offre scalabilità orizzontale attraverso la distribuzione dei file su più server in modalità distribuita, permettendo di gestire petabyte di dati e miliardi di oggetti. Supporta la replica dei dati per garantire alta disponibilità e disaster recovery, implementa erasure coding per proteggere i dati dalla corruzione e dalla perdita senza la necessità di replica completa, e fornisce API RESTful per l'accesso programmatico agli oggetti memorizzati.
Il sistema di bucket di MinIO permette di organizzare logicamente i file in container separati, ognuno con proprie policy di accesso, versioning, lifecycle management e encryption. Per un'applicazione di gestione immobili, è possibile creare bucket separati per immagini di proprietà, documenti degli utenti, immagini di profilo, thumbnail generati automaticamente e così via.

2.6.2 Integrazione con SpringBoot

L'integrazione di MinIO con Spring Boot avviene attraverso l'SDK Java ufficiale di MinIO, che fornisce un client completo per tutte le operazioni di object storage. Il backend implementa servizi dedicati alla gestione delle immagini che si occupano dell'upload di nuove immagini, del download di immagini esistenti, della generazione di URL presigned per l'accesso temporaneo, della gestione del versioning e della cancellazione di immagini obsolete.
Quando un utente carica una nuova immagine per un immobile, il backend riceve il file come multipart/form-data, lo valida (verificando formato, dimensioni, tipo MIME), genera un nome file univoco utilizzando UUID per evitare collisioni, e lo carica su MinIO nel bucket appropriato. Il server può inoltre generare thumbnail di diverse dimensioni utilizzando librerie di image processing come Thumbnailator o ImageIO, memorizzando le versioni ridimensionate accanto all'originale per ottimizzare i tempi di caricamento nell'interfaccia utente.
Nel database PostgreSQL viene mantenuto solo il riferimento all'immagine. Questo riferimento è memorizzato come campo della tabella Immobile. Quando il frontend richiede un immobile, il backend restituisce nel JSON le URL complete per accedere alle immagini, costruite dinamicamente concatenando l'endpoint base di MinIO con il percorso memorizzato nel database.

2.7 Sistema di Sicurezza

2.7.1 Autenticazione ed Autorizzazione con JWT

L'applicazione adotta un meccanismo stateless di autenticazione e autorizzazione basato su JWT (JSON Web Token), uno standard aperto (RFC 7519) che definisce un modo compatto e autonomo per trasmettere informazioni tra le parti come oggetto JSON firmato digitalmente.

2.7.2 Flusso di autenticazione

Al momento del login, l'utente invia le proprie credenziali (username/email e password) al backend tramite una richiesta POST all'endpoint **/api/auth/login.** Il controller di autenticazione delega la verifica delle credenziali al servizio di autenticazione, che recupera l'utente dal database PostgreSQL tramite username/email, verifica l'hash della password utilizzando un algoritmo di hashing robusto come BCrypt (che implementa salting automatico e iterazioni multiple per resistere agli attacchi brute-force), e in caso di successo procede con la generazione del token JWT.
Il token JWT è composto da tre parti separate da punti: header, payload e signature. L'header contiene metadati sul token, come il tipo (JWT) e l'algoritmo di firma utilizzato (tipicamente HS256 per HMAC con SHA-256 o RS256 per RSA con SHA-256). Il payload, chiamato anche claims, contiene le informazioni sull' utente e metadata del token, come subject (identificatore univoco dell'utente), roles (ruoli dell'utente per l'autorizzazione), issued at timestamp

(momento di emissione del token), expiration timestamp (momento di scadenza del token, tipicamente impostato a poche ore o giorni dal momento dell'emissione), e eventualmente altre informazioni custom come nome utente, email, tenant ID in applicazioni multi-tenant, etc.
La signature è generata applicando l'algoritmo di firma specificato nell'header al contenuto concatenato di header e payload, utilizzando una chiave segreta (nel caso di HMAC) o una chiave privata (nel caso di RSA). Questa firma garantisce che il token non possa essere alterato senza invalidare la signature, fornendo integrità e autenticità. Chiunque possieda la chiave segreta (HMAC) o la chiave pubblica corrispondente (RSA) può verificare che il token sia stato emesso dal server autorizzato e non sia stato modificato.
Il server restituisce il token JWT generato al client in risposta al login, tipicamente nel corpo della risposta JSON con status 200 OK. Il client memorizza il token localmente per utilizzarlo nelle richieste successive. È fondamentale notare che, diversamente dalle sessioni tradizionali gestite lato server, il token JWT è completamente autonomo e non richiede che il server mantenga stato alcuno riguardo alle sessioni attive. Questo rende l'architettura intrinsecamente scalabile poiché non esiste un single point of failure legato alla gestione delle sessioni.

2.7.3 Memorizzazione del token sul client

2.7.4 Autorizzazione delle richieste API

Per ogni richiesta successiva al login che richiede autenticazione, il client include il token JWT nell'header HTTP Authorization usando lo schema Bearer. L'header assume la forma Authorization: Bearer ¡token¿, dove ¡token¿ è la stringa del JWT ricevuto durante il login.
Nel backend Spring Boot, questa verifica è implementata attraverso un filtro personalizzato che estende OncePerRequestFilter o attraverso Spring Security Filter Chain. Il filtro intercetta ogni richiesta HTTP in ingresso prima che raggiunga il controller, estrae il token dall'header Authorization, lo valida verificando la signature per assicurarsi che non sia stato alterato e che sia stato realmente emesso dal server, verifica che il token non sia scaduto controllando il claim exp, e in caso di validazione riuscita, estrae le informazioni sull'utente dal payload del token (subject, roles) e le inserisce nel SecurityContext di Spring Security.
Una volta che le informazioni di autenticazione sono state caricate nel SecurityContext, Spring Security gestisce automaticamente l'autorizzazione verificando che l'utente abbia i permessi necessari per accedere all'endpoint richiesto. I controller e i metodi possono essere protetti utilizzando annotazioni come @PreAuthorize, @Secured, o @RolesAllowed per specificare i ruoli richiesti. Ad esempio, @PreAuthorize("hasRole('ADMIN')") su un endpoint garantisce che solo utenti con ruolo ADMIN possano accedervi.

In caso di token mancante, invalido o scaduto, il filtro restituisce un errore HTTP 401 Unauthorized senza invocare il controller. Se il token è valido ma l'utente non ha i permessi necessari per l'operazione richiesta, Spring Security restituisce un errore 403 Forbidden. Questi meccanismi garantiscono che tutte le API siano protette in modo uniforme e che la logica di autorizzazione sia centralizzata e facilmente manutenibile.

2.7.5 Vantaggi dell'approccio Stateless

Il meccanismo stateless basato su JWT offre numerosi vantaggi architetturali. La scalabilità orizzontale è enormemente semplificata poiché non esiste stato di sessione da condividere tra i nodi del cluster: ogni istanza del backend può validare autonomamente i token JWT senza necessità di coordinazione con altre istanze o accesso a uno storage condiviso delle sessioni. Questo elimina la necessità di sticky sessions nei load balancer e permette di aggiungere o rimuovere dinamicamente istanze in base al carico.
La sicurezza è garantita dalla firma crittografica del token, che impedisce la contraffazione, e dalla possibilità di impostare scadenze brevi per minimizzare il danno in caso di token compromesso. L'inclusione dei ruoli e permessi nel token elimina la necessità di interrogare il database per ogni richiesta per recuperare le informazioni di autorizzazione, riducendo significativamente il carico sul database e migliorando i tempi di risposta.
La flessibilità dell'approccio JWT permette l'implementazione di pattern avanzati come i refresh token per rinnovare l'accesso senza richiedere un nuovo login, token blacklisting per invalidare token specifici prima della loro scadenza naturale (utile per logout o revoca di accesso), e la possibilità di utilizzare algoritmi di firma asimmetrica (RS256) che permettono la verifica del token da parte di servizi terzi senza condividere la chiave segreta.

2.7.6 Spring Security

Spring Security è il framework de facto per la gestione della sicurezza nelle applicazioni Spring e Spring Boot. Fornisce un'implementazione completa ed estensibile di autenticazione, autorizzazione, protezione da attacchi comuni (CSRF, session fixation, clickjacking), gestione delle password con algoritmi di hashing robusti, e integrazione con standard industriali come OAuth2, SAML e OpenID Connect.
Nel contesto di questa applicazione, Spring Security è configurato per supportare l'autenticazione JWT stateless, disabilitando le sessioni HTTP tradizionali attraverso la configurazione
sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).
Il framework gestisce la configurazione dei filtri che intercettano le richieste, valida i token JWT, popola il SecurityContext con le informazioni di autenticazione, e applica le regole di autorizzazione definite nelle configurazioni o nelle annotazioni.

La configurazione di Spring Security definisce quali endpoint sono pubblici (accessibili senza autenticazione, come login, registrazione, visualizzazione pubblica degli immobili) e quali richiedono autenticazione o ruoli specifici. La protezione CSRF può essere disabilitata per le API RESTful stateless poiché non si utilizzano cookie di sessione tradizionali, mentre la protezione contro altri attacchi come XSS, clickjacking e injection viene gestita attraverso header di sicurezza HTTP appropriati e validazione rigorosa degli input.

2.8 Deployment e infrastruttura

2.8.1 NGINX come Reverse Proxy e Web Server

NGINX è utilizzato come reverse proxy e web server davanti all'applicazione Spring Boot e ai file statici del frontend React. NGINX è rinomato per le sue prestazioni eccezionali, l'efficienza nell'uso delle risorse, e la capacità di gestire decine di migliaia di connessioni concorrenti con un footprint di memoria ridotto grazie alla sua architettura event-driven asincrona.

2.8.2 Funzioni di NGINX

Come web server, NGINX serve i file statici del frontend React (HTML, CSS, JavaScript, immagini, font) direttamente, sfruttando la sua velocità ottimizzata per lo static content serving e riducendo il carico sull'applicazione Spring Boot. NGINX può essere configurato per implementare cache HTTP aggressive per i file statici, impostando header di caching appropriati per massimizzare l'efficacia delle cache del browser e delle CDN intermedie.
Come reverse proxy, NGINX intercetta tutte le richieste HTTP/HTTPS in arrivo e le instrada verso il backend appropriato. Le richieste per API (/api/*) vengono inoltrate (proxied) all'applicazione Spring Boot in esecuzione su una porta interna (tipicamente 8080), mentre le richieste per file statici vengono servite direttamente da NGINX. Questo pattern permette di presentare un'unica interfaccia esterna all'utente evitando problemi CORS (Cross-Origin Resource Sharing), poiché frontend e backend sembrano risiedere sullo stesso dominio dal punto di vista del browser.
NGINX implementa connection pooling e keep-alive verso il backend, riutilizzando le connessioni TCP e riducendo l'overhead della creazione di nuove connessioni per ogni richiesta. Supporta load balancing con vari algoritmi (round-robin, least connections, IP hash) per distribuire il traffico tra multiple istanze del backend, health checks per rilevare e bypassare istanze non responsive, e failover automatico per garantire alta disponibilità.
Il reverse proxy può inoltre implementare rate limiting per proteggere il backend da abusi e attacchi DoS, caching di risposte API per ridurre il carico sul backend, compressione gzip/brotli delle risposte per ridurre la banda utilizzata e migliorare i tempi di caricamento, e gestione di header HTTP di sicurezza come HSTS,

Content-Security-Policy, X-Frame-Options, X-Content-Type-Options per proteggere da vari attacchi web.

2.8.3 Cloudflare per SSL/TLS e CDN

Cloudflare è utilizzato come layer davanti a NGINX per fornire terminazione SSL/TLS, Content Delivery Network (CDN) globale, protezione DDoS, e altri servizi di sicurezza e performance.

2.8.4 SSL/TLS

Cloudflare gestisce i certificati SSL/TLS utilizzando Let's Encrypt o certificati proprietari, fornendo HTTPS per il dominio dell'applicazione senza che l'amministratore debba gestire manualmente rinnovi e configurazioni dei certificati. Cloudflare supporta protocolli moderni come TLS 1.3 e cipher suite ottimizzate per bilanciare sicurezza e prestazioni, implementa HSTS preloading, e offre certificati wildcard per proteggere tutti i sottodomini.
La connessione tra client e Cloudflare avviene sempre su HTTPS, garantendo la crittografia dei dati in transito e proteggendo da intercettazioni man-in-the-middle. Cloudflare offre diverse modalità di connessione tra i suoi edge server e il server di origine (NGINX): Flexible (HTTPS verso Cloudflare, HTTP verso origine), Full (HTTPS su entrambi i lati ma senza verifica del certificato di origine), e Full Strict (HTTPS su entrambi i lati con verifica completa del certificato).

2.8.5 Content Delivery Network (CDN)

Cloudflare opera una rete globale di data center distribuiti geograficamente che fungono da cache per i contenuti dell'applicazione. Quando un utente richiede una risorsa, la richiesta viene instradata al data center Cloudflare più vicino geograficamente, che serve il contenuto dalla propria cache se disponibile, riducendo drasticamente la latenza e migliorando i tempi di caricamento per utenti distribuiti globalmente.
I file statici del frontend React (JavaScript bundles, CSS, immagini, font) sono particolarmente adatti per il caching CDN poiché cambiano raramente e possono essere cachati aggressivamente. Cloudflare permette di configurare regole di cache personalizzate basate su path, header, query string, e altri criteri, controllando cosa viene cachato, per quanto tempo, e in quali circostanze la cache deve essere invalidata.

2.8.6 Containerizzazione e Orchestrazione

Un'applicazione moderna con questa architettura trae enormi benefici dalla containerizzazione usando Docker e dall'orchestrazione tramite Docker Compose (come nel nostro caso) o Kubernetes per ambienti più semplici.

2.8.7 Docker per la containerizzazione

Ogni componente dell'applicazione (backend Spring Boot, frontend React, NGINX, PostgreSQL, MinIO) viene eseguito in container Docker indipendenti. Il backend Spring Boot utilizza un'immagine base Java ottimizzata (eclipse-temurin), copia il JAR buildato dell'applicazione nel container, e lo espone sulla porta 8080. Il Dockerfile implementa multi-stage builds per compilare l'applicazione all'interno di un container builder e copia solo gli artifact necessari nel container finale, riducendo drasticamente le dimensioni dell'immagine.
NGINX utilizza l'immagine ufficiale nginx:alpine come base, copiando i file statici del frontend React nella directory appropriata e includendo file di configurazione custom per il reverse proxying verso il backend. PostgreSQL e MinIO hanno immagini Docker ufficiali pronte all'uso che possono essere configurate tramite variabili d'ambiente.
I container offrono isolamento, portabilità tra ambienti (development, staging, production), versioning preciso delle dipendenze, e deployment consistente. Docker Compose permette di definire l'intero stack dell'applicazione in un file YAML, orchestrando l'avvio, il networking e il volume management di tutti i container con un singolo comando.

2.9 Architettura del Frontend

2.9.1 React come Framework Frontend

Il frontend dell'applicazione è implementato utilizzando React, una libreria JavaScript sviluppata e mantenuta da Meta (Facebook) che ha rivoluzionato lo sviluppo di interfacce utente moderne. React adotta un approccio component-based, dove l'interfaccia è composta da componenti riutilizzabili e componibili che incapsulano markup, stile e logica in unità coese.

2.9.2 Paradigma Dichiarativo e Virtual DOM

React utilizza un paradigma dichiarativo dove gli sviluppatori descrivono come l'interfaccia dovrebbe apparire dato un certo stato, e React si occupa di aggiornare efficacemente il DOM reale per riflettere quello stato. Questo approccio contrasta con la manipolazione imperativa del DOM tipica del JavaScript vanilla o di librerie come jQuery, risultando in codice più leggibile, prevedibile e facile da debuggare.
Il Virtual DOM è una rappresentazione in memoria della struttura del DOM che React mantiene. Quando lo stato di un componente cambia, React crea un nuovo Virtual DOM, lo confronta con la versione precedente attraverso un algoritmo di diffing ottimizzato, determina il set minimo di cambiamenti necessari per aggiornare il DOM reale, e applica solo quelle modifiche. Questo processo, chiamato reconciliation, garantisce prestazioni eccellenti anche con interfacce complesse e frequenti aggiornamenti.

2.9.3 Component-Based Architecture

L'applicazione React è strutturata come un albero di componenti gerarchico. Componenti di alto livello orchestrano la struttura generale dell'applicazione (layout, routing, gestione dello stato globale), mentre componenti di livello intermedio rappresentano sezioni funzionali specifiche (lista immobili, dettaglio immobile, chat, profilo utente) e componenti di basso livello sono elementi UI riutilizzabili (bottoni, card, form input, modal, navbar).
Ogni componente può essere implementato come function component (approccio moderno preferito) o class component (legacy). I function components utilizzano React Hooks per gestire stato locale (useState), effetti collaterali (useEffect), contesto (useContext), memoizzazione (useMemo, useCallback), e riferimenti (useRef). Gli Hooks permettono di riutilizzare logica stateful tra componenti attraverso custom hooks, evitando duplicazione del codice e promuovendo la composizione.
I componenti comunicano attraverso props (proprietà passate dal parent al child) e callback functions (funzioni passate come props che il child può invocare per notificare eventi al parent). Per gestire stato condiviso tra componenti non direttamente correlati nell'albero, React offre Context API che permette di evitare prop drilling, passando dati attraverso l'albero dei componenti senza doverli passare esplicitamente a ogni livello.

2.9.4 Single Page Application (SPA)

L'applicazione è implementata come Single Page Application, dove l'intero codice JavaScript, CSS e HTML viene caricato al primo accesso, e successivamente solo i dati vengono scambiati con il server tramite chiamate API asincrone. Questo approccio elimina i refresh completi della pagina durante la navigazione, offrendo un'esperienza utente fluida e reattiva simile alle applicazioni desktop native.

2.9.5 React Router per la navigazione

La navigazione nell'applicazione è gestita da React Router, la libreria de facto per il routing lato client in applicazioni React. React Router permette di definire route dichiarative che mappano URL specifici a componenti React da renderizzare. Per esempio, / potrebbe renderizzare la home page con ricerca immobili, /immobili/:id il dettaglio di un immobile specifico, /profilo il profilo dell'utente autenticato, /chat l'interfaccia di messaggistica, e così via.
React Router implementa routing lato client intercettando i click sui link e manipolando la History API del browser per cambiare l'URL senza causare un page reload. Quando l'URL cambia, React Router identifica la route corrispondente e renderizza il componente associato, smontando i componenti delle route precedenti e montando i nuovi. Questo processo è estremamente veloce poiché avviene interamente in memoria senza comunicazione con il server.
Il router supporta nested routes per creare layout gerarchici dove componenti

di layout comuni (header, sidebar, footer) rimangono montati mentre solo il contenuto centrale cambia, route parameters per passare dati dinamici tramite l'URL, query parameters per filtri e stato opzionale, programmatic navigation per navigare da codice in risposta a eventi o condizioni, route guards per proteggere route che richiedono autenticazione o autorizzazione, e lazy loading di route per code splitting e ottimizzazione delle prestazioni.

2.9.6 Gestione dello stato

La gestione dello stato è un aspetto critico delle SPA complesse. Lo stato include dati temporanei dell'interfaccia (form input, modal aperti/chiusi, tabs selezionati), dati dell'applicazione (immobili caricati, profilo utente, messaggi chat), stato di autenticazione (token JWT, informazioni utente corrente), e stato delle richieste API (loading, error, success).
React offre diversi livelli di gestione dello stato. Lo stato locale dei componenti è gestito con useState per dati che appartengono a un singolo componente e non necessitano di essere condivisi. Context API permette di condividere stato tra componenti senza prop drilling, ideale per tema UI, lingua, informazioni utente autenticato, e altre configurazioni globali che cambiano raramente.
Per applicazioni più complesse con stato globale esteso e logica di aggiornamento complessa, librerie di state management come Redux, Zustand, Jotai o Recoil offrono soluzioni più strutturate. Redux, il più maturo e adottato, implementa un store centralizzato con aggiornamenti gestiti tramite actions e reducers, offrendo predicibilità, debugging avanzato con time-travel tramite Redux DevTools, e middleware per gestire side effects asincroni.
React Query (TanStack Query) o SWR sono librerie specializzate nella gestione dello stato server-side, gestendo automaticamente caching, invalidazione, refetching, sincronizzazione background, e ottimizzazioni come deduplicazione delle richieste. Queste librerie eliminano gran parte del boilerplate per la gestione delle chiamate API e migliorano significativamente le prestazioni e l'esperienza utente.

2.9.7 Comunicazione con il backend

Il frontend comunica con il backend esclusivamente tramite API RESTful, effettuando richieste HTTP asincrone utilizzando Fetch API nativa del browser o librerie wrapper come Axios che offrono funzionalità aggiuntive come interceptor, timeout automatici, cancellazione delle richieste, e trasformazione automatica dei dati.

2.9.8 Axios e Interceptor

Axios è tipicamente configurato con una istanza base che include l'URL di base del backend (es. https://api.tuodominio.com o semplicemente /api se il frontend è servito dallo stesso dominio tramite NGINX), timeout predefiniti, e header comuni. Gli interceptor di Axios permettono di intercettare e modificare richieste

e risposte globalmente, implementando logica cross-cutting come l'aggiunta automatica del token JWT all'header Authorization di ogni richiesta autenticata, la gestione centralizzata degli errori con reindirizzamento automatico al login in caso di 401 Unauthorized, il refresh automatico del token quando scaduto, il logging delle richieste per debugging, e la gestione di loading spinner globali.

2.9.9 Pattern di chiamata API

Le chiamate API seguono pattern asincroni gestiti con Promise, async/await, o attraverso librerie di state management. Un pattern comune prevede la definizione di servizi API che incapsulano le chiamate specifiche, separando la logica di networking dalla logica dei componenti UI. Ogni servizio espone metodi che corrispondono agli endpoint backend, gestendo la costruzione delle richieste, la gestione degli errori, e la trasformazione delle risposte.
I componenti React invocano questi servizi tipicamente all'interno di useEffect per caricare dati al mount del componente, o in risposta ad azioni utente (submit di form, click su bottoni). Durante la chiamata API, il componente mostra uno stato di loading, e al completamento aggiorna lo stato con i dati ricevuti o mostra un messaggio di errore in caso di fallimento.

2.9.10 Tailwind CSS per lo Styling

Il frontend utilizza Tailwind CSS, un framework CSS utility-first che ha rivoluzionato l'approccio allo styling nelle applicazioni moderne. A differenza di framework component-based tradizionali come Bootstrap o Material-UI che forniscono componenti pre-stilizzati, Tailwind fornisce classi utility atomiche a basso livello che possono essere combinate per costruire qualsiasi design direttamente nel markup.

2.9.11 Filosofia Utility-First

L'approccio utility-first significa che invece di scrivere CSS custom o utilizzare componenti pre-fatti, gli sviluppatori compongono gli stili applicando classi utility direttamente agli elementi HTML/JSX.
Questo approccio offre numerosi vantaggi: eliminazione del context-switching tra file HTML/JSX e CSS, nessun problema di naming conventions o collisioni di classi, design system consistente attraverso scale predefinite per spacing, colori, typography, responsive design semplificato con prefissi per breakpoint (sm:, md:, lg:, xl:), pseudo-classi e pseudo-elementi gestiti con prefissi (hover:, focus:, active:, first:, last:), e file CSS finali estremamente ottimizzati grazie al tree-shaking automatico che include solo le classi effettivamente utilizzate.

2.9.12 Configurazione e Customizzazione

Tailwind è altamente configurabile attraverso il file tailwind.config.js, che permette di estendere o sovrascrivere la configurazione predefinita. È possibile

definire palette di colori custom che riflettono il brand dell'applicazione, scale di spacing personalizzate, font families, breakpoint responsive, durate e timing functions per animazioni, e molto altro. La configurazione supporta anche temi multipli (light/dark mode) attraverso la variante dark: che applica stili alternativi basati sulla preferenza di sistema o su uno stato dell'applicazione.
Il sistema di plugin di Tailwind permette di estendere le funzionalità base, aggiungendo nuove utility, varianti, componenti, o integrando librerie complementari. Tailwind UI offre componenti pre-costruiti utilizzando le utility di Tailwind, fornendo un punto di partenza per UI complesse come form elaborati, tabelle con sorting e filtering, dashboard layouts, e-commerce interfaces, e molto altro.

2.9.13 Integrazione con React

In progetti React, Tailwind si integra perfettamente attraverso PostCSS, processando le direttive Tailwind (@tailwind base, @tailwind components, @tailwind utilities) nel file CSS principale e generando il file CSS finale ottimizzato. Durante il build process, il PurgeCSS integrato in Tailwind scansiona tutti i file JSX/TSX specificati nella configurazione, identifica le classi utilizzate, e genera un file CSS che contiene solo quelle classi, risultando in bundle CSS minuscoli anche per applicazioni complesse.
I componenti React possono utilizzare conditional classes attraverso librerie come classnames o clsx per applicare dinamicamente classi basate su props o stato.

2.9.14 Responsive Design e Accessibilità

Tailwind eccelle nel responsive design attraverso il mobile-first approach. Le classi senza prefisso si applicano a tutte le dimensioni di schermo, mentre prefissi come md:, lg:, xl: applicano stili solo a breakpoint specifici. Questo permette di creare layout complessi che si adattano perfettamente a diverse dimensioni di schermo con classi come flex flex-col md:flex-row lg:grid lg:grid-cols-3.
Per l'accessibilità, Tailwind fornisce utility per focus states visibili (focus:ring, focus:outline), screen reader utilities (sr-only), e supporta tutte le best practices ARIA. Combinato con componenti React accessibili, si possono costruire interfacce che rispettano gli standard WCAG garantendo usabilità per tutti gli utenti.

2.10 Containerizzazione con Docker

L'intera applicazione è containerizzata utilizzando Docker, permettendo deployment consistente, isolamento tra componenti, e facilità di sviluppo con ambienti identici in development, staging e production.

2.10.1 Architettura Docker Multi-Container

L'applicazione è composta da sei container Docker, ognuno incapsulante un componente specifico dello stack tecnologico:

- `Container Backend (Spring Boot):` contiene l'applicazione Java Spring Boot con tutte le sue dipendenze;
- `Container Frontend (React):` Contiene il frontend React e tutte le sue dipendenze;
- `Container NGINX:` Serve i file statici di React e funge da Reverse Proxy verso il backend;
- `Container Database (PostgreSQL):` Gestisce la persistenza dei dati relazionali;
- `Container Object Storage (MiniIO):` Gestisce la persistenza delle immagini e file multimediali;
- `Container Network Setting (Cloudflared):` Consente l'impiego del servizio Cloudflared per connessioni sicure.

2.10.2 Dockerfile del Backend

```
# Stage 1: Build
FROM maven:3.9.6-eclipse-temurin-21 AS builder
WORKDIR /app

COPY pom.xml .
RUN mvn -B dependency:resolve

COPY src ./src
RUN mvn -B clean package -DskipTests

# Stage 2: Runtime
FROM eclipse-temurin:21-jre
WORKDIR /app

COPY --from=builder /app/target/*.jar app.jar

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Il primo stage utilizza un'immagine Maven completa per compilare il codice sorgente, risolvere le dipendenze, ed eseguire il package Maven che produce il JAR eseguibile. Il secondo stage parte da un'immagine JRE minimale (significativamente più piccola del JDK completo), copia solo il JAR compilato dal primo stage, ed esegue l'applicazione. Questo approccio riduce l'immagine finale

da oltre 500MB a circa 200MB, migliorando i tempi di pull e deployment. L'immagine Alpine Linux è scelta per il runtime per la sua dimensione minuscola (circa 5MB base) e il suo profilo di sicurezza ridotto. L'applicazione Spring Boot è configurata tramite variabili d'ambiente passate al container per connessione database, configurazione MinIO, chiave segreta JWT, profilo Spring attivo (dev, prod), e altre configurazioni environment-specific.

2.10.3 Dockerfile del Frontend

Il Frontend React viene buildato in un multi-stage process simile:

```
# Stage 1: Build
FROM node:20-alpine AS builder
WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .
RUN npm run build

# Stage 2: Serve with nginx
FROM nginx:alpine

COPY --from=builder /app/dist /usr/share/nginx/html
```

2.10.4 Dockerfile e configurazione NGINX

```
FROM nginx:alpine

COPY nginx.conf /etc/nginx/nginx.conf

# Create directory for certificates (Lets Encrypt)
RUN mkdir -p /etc/letsencrypt/live \
    && mkdir -p /etc/letsencrypt/archive \
    && mkdir -p /var/www/certbot

EXPOSE 80
EXPOSE 443
```

Il file nginx.conf configura NGINX per servire i file statici React e reindirizzare le richieste API verso il backend:

```
worker_processes auto;

events {
    worker_connections 1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    sendfile        on;
    keepalive_timeout 65;
    client_max_body_size 200M;

    server {
        listen 80;

        # React frontend
        location / {
            proxy_pass http://frontend:80;
            proxy_http_version 1.1;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For
                $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # API backend
        location /api/ {
            proxy_pass http://backend:8080;
            proxy_http_version 1.1;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For
                $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # Auth backend
        location /auth/ {
            proxy_pass http://backend:8080;
            proxy_http_version 1.1;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For
                $proxy_add_x_forwarded_for;
```

```
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # Chat backend
        location /chat/ {
            proxy_pass http://backend:8080;
            proxy_http_version 1.1;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For
                $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # MinIO images
        location /images/ {
            proxy_pass http://minio:9000/images/;
            proxy_http_version 1.1;
            proxy_set_header Host minio:9000;
            proxy_set_header X-Real-IP $remote_addr;
        }

        add_header X-Content-Type-Options nosniff;
        add_header X-Frame-Options SAMEORIGIN;
    }
}
```

2.10.5 Docker Compose per l'Orchestrazione

Docker Compose orchestra l'intero stack dell'applicazione attraverso un file docker-compose.yml che definisce tutti i servizi, le loro configurazioni, le relazioni, i volumi per la persistenza dei dati, e le reti per la comunicazione inter-container.

```
services:
  backend:
    build:
      context: ./DietiEstates2025
      dockerfile: Dockerfile
    container_name: backend
    restart: unless-stopped
    environment:
      FRONTEND_URL: "https://stars-friends-instead-applies.
          trycloudflare.com"
      SPRING_DATASOURCE_URL: jdbc:postgresql://db:5432/
          dietiestates2025
      SPRING_DATASOURCE_USERNAME: antoniolegnante
      SPRING_DATASOURCE_PASSWORD: lama
      MINIO_URL: http://minio:9000
      MINIO_ACCESS_KEY: minioadmin
      MINIO_SECRET_KEY: minioadmin123
      MINIO_BUCKET_NAME: images
    depends_on:
      - db
      - minio

  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: frontend
    restart: unless-stopped

  nginx:
    build:
      context: ./nginx
      dockerfile: Dockerfile
    container_name: nginx_proxy
    restart: unless-stopped
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - backend
      - frontend

  db:
    image: postgres:16
```

```
    container_name: postgres_db
    environment:
      POSTGRES_USER: antoniolegnante
      POSTGRES_PASSWORD: lama
      POSTGRES_DB: dietiestates2025
    volumes:
      - postgres_data:/var/lib/postgresql/data
    restart: unless-stopped

  minio:
    image: minio/minio
    container_name: minio_storage
    restart: unless-stopped
    command: server /data --console-address ":9001"
    environment:
      MINIO_ROOT_USER: minioadmin
      MINIO_ROOT_PASSWORD: minioadmin123
    volumes:
      - minio_data:/data
    ports:
      - "9000:9000"
      - "9001:9001"

  cloudflared:
    image: cloudflare/cloudflared:latest
    container_name: cloudflared
    restart: unless-stopped
    command: tunnel --url http://nginx:80
    depends_on:
      - nginx


volumes:
  postgres_data:
  minio_data:
```

2.10.6 Deployment con Docker Compose

L'intero stack applicativo viene avviato con un singolo comando:

```
docker-compose up -d
```

Questo comando:

- Legge il file docker-compose.yml nella directory corrente;
- Crea i volumi postgres_data e minio_data per la persistenza;
- Builda le immagini per backend e frontend se non esistono;
- Pulla le immagini per PostgreSQL e MiniIO dal Docker Hub;
- Avvia i container nell'ordine corretto rispettando le dipendenze definite con depends_on;
- Esegue gli healthcheck per verificare che i servizi siano pronti prima di avviare i servizi dipendenti;
- Esegue i container in modalità detached (-d), permettendo che continuino a girare in background.

2.10.7 Gestione dei volumi per la persistenza

I volumi Docker garantiscono che i dati persistano tra riavvii dei container. Il volume postgres_data mappa la directory /var/lib/postgresql/data del container PostgreSQL, dove il database memorizza tutti i file di dati, indici, WAL logs e configurazioni. Similmente, minio_data mappa la directory /data di MinIO dove vengono memorizzati tutti gli oggetti (immagini).
I volumi sono gestiti da Docker e risiedono in una directory speciale del filesystem host (tipicamente /var/lib/docker/volumes/ su Linux). Questo isola completamente i dati dal lifecycle dei container: i container possono essere distrutti, ricreati, aggiornati senza perdita di dati. I volumi possono essere backuppati, migrati tra host, e gestiti indipendentemente dai container.

2.10.8 Health Checks e gestione dei fallimenti

Gli health checks definiti per PostgreSQL e MinIO permettono a Docker di verificare che i servizi siano non solo avviati ma effettivamente funzionanti e pronti ad accettare connessioni. Il backend, configurato con depends_on e condition: service_healthy, aspetta che PostgreSQL e MinIO passino i loro health check prima di avviarsi, evitando errori di connessione durante lo startup.

La politica di restart unless-stopped garantisce che i container vengano automaticamente riavviati in caso di crash o errori, implementando una forma base di self-healing. Se il backend crasha per un'eccezione non gestita, Docker lo riavvia automaticamente. Solo un esplicito docker-compose down o docker stop ferma permanentemente i container.

2.11 Integrazione con Cloudflared

Cloudflare si posiziona come layer più esterno dell'architettura, davanti a NGINX, gestendo il traffico globale verso l'applicazione.

2.11.1 Modalità Proxy e SSL/TLS

Con il proxy arancione attivo (proxy mode) in Cloudflare, tutto il traffico passa attraverso i server Cloudflare. La configurazione SSL/TLS in modalità "Full (Strict)" garantisce crittografia end-to-end: HTTPS tra client e Cloudflare, e HTTPS tra Cloudflare e il server di origine (NGINX).

2.11.2 Regole di Firewall e sicurezza

Cloudflare permette di configurare regole di firewall (WAF Rules) per bloccare traffico malevolo prima che raggiunga il server. Regole comuni includono il blocco di IP geografici specifici se l'applicazione serve solo determinate regioni, rate limiting per endpoint sensibili come login e registrazione per prevenire brute-force, challenge CAPTCHA per traffico sospetto, e blocco automatico di pattern noti di attacco SQL injection, XSS, e altri vettori dell'OWASP Top 10.

2.11.3 Ottimizzazioni delle prestazioni

Oltre al caching CDN, Cloudflare offre ottimizzazioni come Auto Minify per minimizzare automaticamente HTML, CSS e JavaScript, Brotli compression per compressione superiore rispetto a gzip, HTTP/2 e HTTP/3 support per multiplexing e prestazioni migliorate, Early Hints per inviare header di preload mentre il server sta generando la risposta, e Argo Smart Routing per instradare il traffico attraverso i percorsi di rete meno congestionati.

2.12 Ciclo completo delle richieste

2.12.1 Flusso di una richiesta API

Quando un utente autenticato effettua un'azione che richiede comunicazione con il backend (es. caricamento di un nuovo immobile), il flusso completo è:

1. `Client React:` l'utente compila un form e clicca "Pubblica". Il componente React invoca una funzione handler che previene il submit predefinito, valida i dati lato client, e chiama un servizio API;

2. `Servizio API (frontend):` il servizio costruisce una richiesta HTTP POST con i dati del form serializzati in JSON. Axios interceptor aggiunge automaticamente l'header Authorization: Bearer ¡token¿. La richiesta viene inviata a /api/immobili;

3. `NGINX:` riceve la richiesta sulla porta 80/443. Identifica che il path inizia con /api/ e, basandosi sulla configurazione di reverse proxy, inoltra la richiesta a http://backend:8080/api/immobili attraverso la rete Docker interna, preservando gli header originali e aggiungendo header X-Forwarded-*;

4. `Spring Boot - DispatcherServlet:` il DispatcherServlet (Front Controller) intercetta la richiesta in ingresso, identifica il metodo handler appropriato basandosi sul path e sul verbo HTTP (POST), e la passa attraverso la filter chain;

5. `JWT Authentication Filter:` un filtro custom estrae il token dall'header Authorization, lo valida verificando la signature e la scadenza, estrae le informazioni utente (ID, roles), e popola il SecurityContext di Spring Security;

6. `Controller:` il metodo controller annotato con @PostMapping("/api/immobili") viene invocato. Spring Boot automaticamente deserializza il JSON della richiesta in un oggetto DTO (Data Transfer Object) usando Jackson, applicando validazione con Bean Validation annotations (@NotNull, @Size, @Pattern, ecc);

7. `Service Layer:` il controller delega la logica di business al servizio appropriato (ImmobileService), passando il DTO validato. Il servizio applica ulteriori validazioni di business, converte il DTO in un'entità Immobile, gestisce il caricamento delle immagini su MinIO generando URL di riferimento, e persiste l'entità nel database tramite repository;

8. `Repository/JPA/Hibernate:` il repository (interfaccia che estende JpaRepository) delega a Hibernate, che genera la query SQL INSERT appropriata per PostgreSQL, gestisce la transazione, assegna l'ID generato automaticamente (sequence o identity), e committa la transazione;

9. `MinIO:` : parallelamente o sequenzialmente, il servizio carica le immagini dell'immobile su MinIO utilizzando l'SDK Java, che invia richieste HTTP PUT all'endpoint MinIO con i dati binari delle immagini;

10. `Response:` il servizio ritorna l'entità creata (o un DTO di risposta) al controller. Il controller restituisce un ResponseEntity con status 201 Created e il corpo contenente l'immobile creato serializzato in JSON da Jackson;

11. `NGINX:` riceve la risposta dal backend e la inoltra al client, aggiungendo eventualmente header di caching se appropriato;

12. `Client React:` Axios riceve la risposta con status 201. Il codice gestisce il successo aggiornando lo stato locale per riflettere il nuovo immobile, mostrando una notifica di successo, e navigando alla pagina di dettaglio del nuovo immobile o aggiornando la lista degli immobili dell'utente.

2.12.2 Flusso di caricamento iniziale dell'applicazione

Al primo accesso, il flusso è significativamente diverso:

1. `Browser:`l'utente inserisce l'URL del dominio (es. https://tuodominio.com) nella barra degli indirizzi;
2. `DNS Resolution:` il browser risolve il dominio interrogando i nameserver di Cloudflare, che restituiscono un IP Anycast puntando al data center Cloudflare più vicino geograficamente;
3. `Cloudflare:` la richiesta HTTPS raggiunge il data center Cloudflare. Se i file statici sono in cache e validi, Cloudflare li serve direttamente senza toccare il server di origine. Altrimenti, Cloudflare inoltra la richiesta al server di origine;
4. `NGINX:` riceve la richiesta per / (root). Basandosi sulla configurazione, serve il file index.html dalla directory /usr/share/nginx/html;
5. `Browser:` riceve l'HTML, lo parsa, e identifica le risorse referenziate (CSS, JavaScript bundles, immagini). Invia richieste parallele per ciascuna risorsa;
6. `NGINX/Cloudflare:` servono le risorse statiche, con Cloudflare che le cacha aggressivamente data l'intestazione Cache-Control impostata da NGINX;
7. `Browser:` una volta scaricati tutti i file JavaScript, esegue il codice React. React renderizza l'interfaccia iniziale, React Router inizializza il routing basandosi sull'URL corrente, e i componenti montati eseguono eventuali chiamate API per caricare dati (es. lista immobili in homepage);
8. `Chiamate API Iniziali:` seguono il flusso descritto precedentemente, caricando dati dal backend tramite NGINX verso Spring Boot verso PostgreSQL, e ritornando JSON al frontend;
9. `Render Finale:` React aggiorna il DOM con i dati ricevuti, mostrando l'interfaccia completa e interattiva all'utente.

Da questo momento, l'applicazione è completamente caricata nel browser e tutte le successive interazioni avvengono senza page reload, con solo scambi di dati JSON con il backend.

2.13 Considerazioni di sicurezza

2.13.1 Sicurezza del Backend

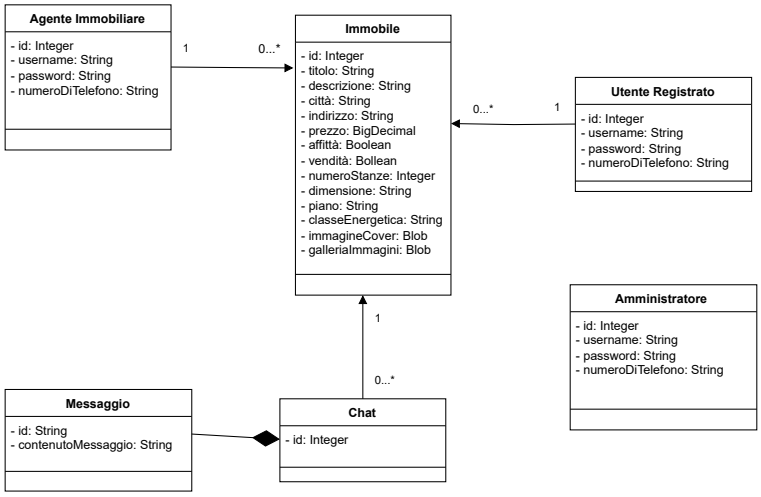
Spring Security fornisce protezione completa contro le vulnerabilità più comuni. CSRF protection è disabilitata per le API stateless ma può essere abilitata

se necessario per alcune route. XSS protection è fornita attraverso encoding automatico dell'output e Content-Security-Policy headers. SQL Injection è prevenuta automaticamente dall'uso di parametrized queries di JPA/Hibernate.
La validazione degli input è implementata a più livelli: lato client con React per feedback immediato, nel controller Spring con Bean Validation, e nella logica di business con validazioni custom. Le password sono hashate con BCrypt, un algoritmo robusto che include salting automatico e costo computazionale configurabile, resistente ad attacchi brute-force grazie alle iterazioni multiple.

2.13.2 Sicurezza del Frontend

Il frontend React beneficia dell'auto-escaping predefinito di React che previene XSS quando si renderizza contenuto dinamico. Il framework converte automaticamente stringhe in testo sicuro. L'uso di dangerouslySetInnerHTML è evitato a meno che assolutamente necessario e con sanitizzazione appropriata tramite librerie come DOMPurify.
I token JWT sono memorizzati in modo sicuro considerando i trade-off tra sicurezza e usabilità.

2.14 Schema dei dati



2.15 Progettazione interfaccia utente

In questa sezione si vuole descrivere come è stata pensata l'interfaccia e perchè sono state adottate determinate scelte piuttosto che altre. L'interfaccia di DietiEstates25 è strutturata come un'applicazione web con una barra di navigazione principale e un'area centrale dedicata alla gestione di immobili

Navigazione

La navigazione è organizzata tramite un menu principale che consente l'accesso diretto alle funzionalità principali del sistema. Quando l'utente non è autenticato, il menu è limitato alle pagine Home, dalla quale è possibile effettuare la ricerca degli immobili, e alle funzionalità di Registrazione e Accesso.

Un utente autenticato dispone di ulteriori voci di menu; in particolare, un agente immobiliare ha accesso alla funzionalità di inserimento di un nuovo immobile. Tutti gli utenti autenticati possono inoltre accedere allo storico delle chat associate ai singoli immobili tramite un'apposita voce del menu.

Motivazioni delle scelte

L'interfaccia privilegia una disposizione semplice e lineare al fine di ridurre il carico cognitivo degli utenti, che non sono necessariamente esperti informatici. La pagina principale presenta una barra di ricerca con filtri essenziali, quali città, tipologia dell'operazione (affitto o vendita), prezzo minimo e prezzo massimo; ricerche più specifiche possono essere effettuate tramite l'icona "Altri filtri".

L'utilizzo di maschere guidate e campi obbligatori riduce la possibilità di inserimento di dati non validi, come date incoerenti o importi errati, migliorando l'affidabilità complessiva del sistema.

La pagina principale, fino all'esecuzione di una ricerca, rimane intenzionalmente vuota per limitare la quantità di informazioni visualizzate e prevenire il disorientamento dei nuovi utenti. Ogni immobile è rappresentato tramite una scheda contenente le informazioni essenziali e una mappa dedicata; tale scelta evita l'uso di una singola mappa sovraccarica con tutti gli immobili della zona di riferimento, risultando di difficile consultazione. Ulteriori dettagli sono accessibili tramite la pagina specifica dell'immobile, raggiungibile selezionando la relativa scheda.

2.16 Diagramma delle classi

2.17 Diagramma di sequenza

Capitolo 3

Testing

3.1 Test del Controller

3.2 Test del Service

3.3 Test End-to-end

3.4 Test Usabilita