

UNIVERSITÀ DI FEDERICO II

INGEGNERIA DEL SOFTWARE

---

# DietiEstates25

---

*Autori*

Antonio LEGNANTE

Vincenzo NOVIELLO

*Professori*

Prof. S. DI MARTINO

Prof. L.L.L. STARACE

Gennaio 2026



# Indice

<b>1</b>	<b>Analisi dei requisiti</b>	<b>3</b>
1.1	Glossario	3
1.2	Casi d'uso	4
1.2.1	Descrizione dei requisiti funzionali	4
1.2.2	Diagramma dei casi d'uso	5
1.3	Personas	5
1.4	Requisiti non funzionali	10
1.5	Diagrammi di cockburn	10
1.6	Mock-up	10
<b>2</b>	<b>Analisi di sistema</b>	<b>11</b>
2.1	Architettura di sistema	11
2.1.1	Motivazioni architetturali	11
2.2	Architettura del Server	12
2.2.1	Framework e Tecnologie del Backend	12
2.2.2	Architettura MVC del Backend	12
2.2.3	Il Model	13
2.3	Controller	13
2.4	API RESTful	14
2.5	Sistema di persistenza dei dati	15
2.5.1	PostgreSQL come Database Relazionale	15
2.5.2	Integrazione con SpringBoot tramite JPA/Hibernate	15
2.6	MiniIO per la gestione delle immagini	16
2.6.1	Vantaggi di MiniIO	16
2.6.2	Integrazione con SpringBoot	17
2.7	Sistema di Sicurezza	17
2.7.1	Autenticazione ed Autorizzazione con JWT	17
2.7.2	Flusso di autenticazione	17
2.8	Schema dei dati	18
2.9	Design interfaccia utente - non ho capito -	18
2.10	Diagramma delle classi	18
2.11	Diagramma di sequenza	18

<b>3</b>	<b>Testing</b>	<b>19</b>
3.1	Test del Controller . . . . .	19
3.2	Test del Service . . . . .	19
3.3	Test End-to-end . . . . .	19
3.4	Test Usabilita . . . . .	19

# Capitolo 1

## Analisi dei requisiti

### 1.1 Glossario

Di seguito sono elencati i concetti fondamentali del progetto

<b>Termine</b>	<b>Descrizione</b>
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 storico 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 Personas**

Sono descritti di seguito i tipici utenti dell'applicazione



**Name: Lidia Pascal**

#### Demografia

**Genere:** Femmina  
**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 vita/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 Bridgerton e prendere tè e biscotti mentre guarda la serie



**Name: Lidia Pascal**

#### Demografia

**Genere:** Femmina  
**Età:** 25  
**Stato coniugale:** Nubile  
**Titolo:** Studentessa  
**Tratti personali:** Ambiziosa, leale

#### Background

Lidia è un'attenta e dedicata 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 vita/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 Bridgerton e prendere tè e biscotti mentre guarda la serie





**Name: Lidia Pascal**

#### Demografia

**Genere:** Femmina  
**Età:** 25  
**Stato coniugale:** Nubile  
**Titolo:** Studentessa  
**Tratti personali:** Ambiziosa, leale

#### Background

Lidia è un'attenta e dedicata 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 Bridgerton e prendere tè e biscotti mentre guarda la serie



**Name: Lidia Pascal**

#### Demografia

**Genere:** Femmina  
**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 vita/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 Bridgerton e prendere tè e biscotti mentre guarda la serie

**1.4    Requisiti non funzionali**

**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 MinIO

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, eccetera.

## **2.8 Schema dei dati**

## **2.9 Design interfaccia utente - non ho capito -**

## **2.10 Diagramma delle classi**

## **2.11 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