



Corso di Laurea Triennale in Ingegneria Informatica
Anno accademico 2024 - 2025



MANUTENZIONE COLLEGE

CANDIDATI:
Maria Nicola Goranova [214096]
Dario Nardella [218040]

RELATORE:
Prof. Salnitri Mattia

Indice

1. Manutenzione College	
1.1 Panoramica	3
1.2 Obiettivo	3
1.3 Analisi dei requisiti	3
1.3.1 Requisiti funzionali	3
1.3.2 Use Case diagrams.....	4
1.3.3 Requisiti non funzionali	5
1.3.4 Design strutturale	7
1.3.5 Ulteriori tecnologie utilizzate.....	8
2. Database	
2.1 Diagramma ER	10
2.2 Note implementative	11
3. Back-end	
3.1 Progettazione architetturale	11
3.1.1 Pattern MVCS	11
3.1.2 Packaging	12
3.2 Exception	14
3.2.1 Exception Class diagram	14
3.3 Dominio	15
3.3.1 Class diagram	16
3.4 Design patterns	17
3.4.1 Builder	17
3.4.1.1 Builder Class Diagram	18
3.4.2 State Pattern	19
3.4.2.1 State Pattern Class Diagram.....	20
3.5 Sicurezza e autorizzazione	22
3.5.1 Security Class diagram	22
3.5.2 Security Sequence diagram	24
3.5.3 Security Activity diagram	25
3.6 Mailing	26

3.6.1 Mailing Sequence diagram	27
3.7 Chat	27
3.8 Testing	29
3.8.1 Test Coverage	30
3.9 Docker	31
3.10 Avvio progetto	31
 4. Front-end	
4.1 Tecnologie utilizzate	32
4.1.1 Angular	32
4.1.2 Angular Material.....	35
4.2 Design patterns	35
4.2.1 DTO.....	35
4.2.2 Dependency Injection & Singleton	36
4.2.3 Observer	36
4.2.4 Composite	37
4.2.5 Validator	37
4.2.6 Decorator	38
4.2.7 Builder	39
4.3 Guida alla navigazione	40
5. Conclusione e Sviluppi futuri	43

1 Manutenzione College

1.1 Panoramica

La gestione delle manutenzioni nel nostro college è sempre stata complessa. In passato, ci affidavamo a gruppi WhatsApp e intermediari, ma questo portava spesso a problemi: guasti non tracciati, manutentori ai piani dei residenti senza preavviso e comunicazione inefficace.

Per risolvere queste difficoltà, abbiamo sviluppato il sistema di gestione delle manutenzioni del college, che centralizza le segnalazioni e facilita la comunicazione tra residenti, supervisori e manutentori. Grazie a notifiche automatiche e una chiara assegnazione dei compiti, il sistema garantisce un'organizzazione più efficiente e un ambiente funzionale per tutti gli utenti.

1.2 Obiettivo

L'obiettivo è semplificare la gestione delle richieste di manutenzione, riducendo i tempi di attesa e migliorando la comunicazione tra le parti coinvolte. Il sistema consente di tracciare ogni intervento fino alla sua risoluzione, ottimizzando il flusso di lavoro.

1.3 Analisi dei requisiti

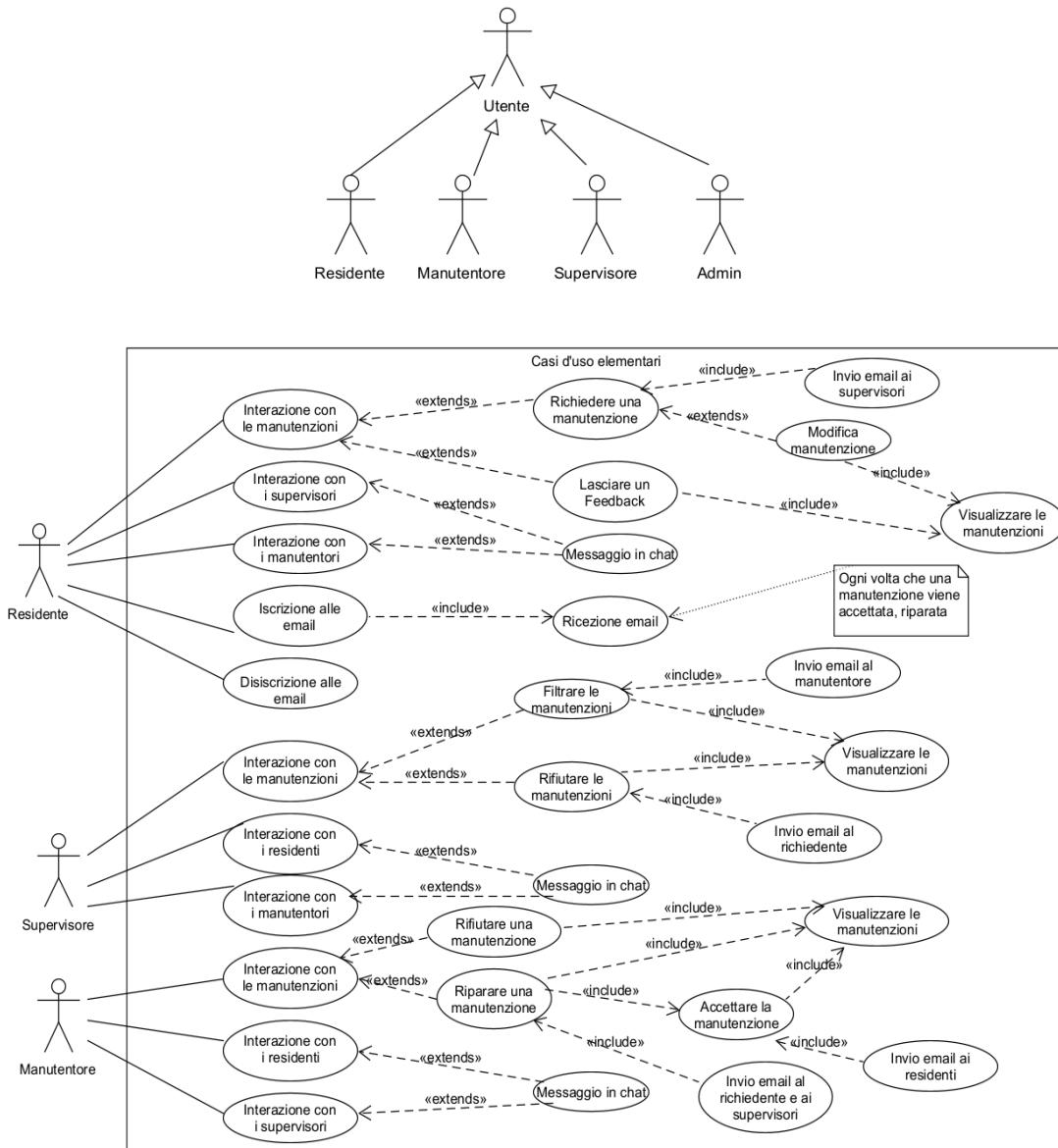
1.3.1 Requisiti funzionali

I requisiti funzionali definiscono le principali operazioni disponibili per ciascun ruolo, garantendo un sistema efficiente e intuitivo:

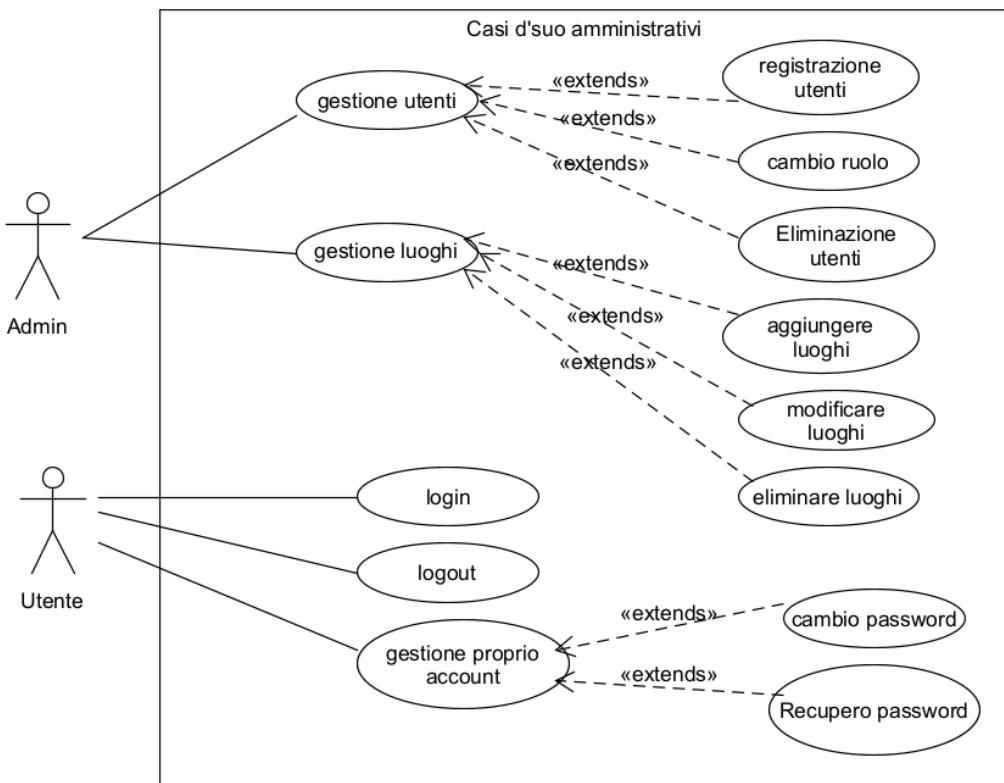
- **Residente:** Invia segnalazioni di guasti e riceve notifiche sullo stato della richiesta. Dopo l'intervento, può lasciare un feedback della riparazione.
- **Supervisore:** Gestisce le richieste, decidendo se accettarle, eliminarle o assegnare loro una priorità. Può inoltrare i ticket ai manutentori e consultare lo storico interventi.
- **Manutentore:** Pianifica e notifica le riparazioni delle manutenzioni, può rifiutare richieste da rivedere o accettarle, informando tutti gli interessati.
- **Admin:** Gestisce utenti e assegnazioni, aggiungendo o rimuovendo profili e definendo i luoghi comuni del college.
- **Chat:** Tutti gli utenti, tranne l'admin, hanno la possibilità di interfacciarsi con le altre utenze tramite un chat.

1.3.2 Use Case diagrams

Di seguito due Use Case diagrams che sintetizzano chiaramente tutte le azioni tramite cui gli attori possono interagire con l'applicazione. Si tenga sempre conto della seguente generalizzazione, omessa dai due diagrammi per ovvie questioni di spazio e ordine:



Use Case diagram 1 – Schema dei casi d'uso elementari, cioè le principali azioni basate sul topic dell'applicazione. Comprende l'interazione tra gli utenti stessi e l'interazione tra utenti e manutenzioni. Descrive quelli che sono i principali requisiti funzionali richiesti.



Use Case diagram 2 – Schema dei casi d'uso d'amministrazione. Comprende creazione, eliminazione e modifica di luoghi e utenti.

1.3.3 Requisiti non funzionali

Oltre alle funzionalità specifiche, un'applicazione web di successo deve garantire elevati standard di qualità, prestazioni e sicurezza. Questi aspetti, definiti "requisiti non funzionali", sono essenziali per un'esperienza utente positiva e il rispetto delle normative di settore. Particolare attenzione è stata dedicata a:

Manutenibilità:

La manutenibilità si riferisce alla facilità con cui il software può essere modificato, aggiornato e riparato nel tempo. Un codice manutenibile è essenziale per garantire la longevità dell'applicazione e per facilitare l'implementazione di nuove funzionalità o la correzione di eventuali bug. Per raggiungere questo obiettivo, abbiamo adottato le seguenti strategie:

- **Chiarezza e leggibilità del codice:** Il codice sorgente è stato scritto seguendo uno stile chiaro e conciso, con un'attenzione particolare alla nomenclatura e all'organizzazione del codice, per renderlo facilmente comprensibile da altri sviluppatori.

- **Documentazione completa:** Il codice è accompagnato da una documentazione dettagliata, che include commenti esplicativi, diagrammi di flusso e guide all'utilizzo, per facilitare la comprensione del funzionamento del sistema.
- **Test unitari rigorosi:** Sono stati implementati test unitari per verificare il corretto funzionamento di ogni componente del software, garantendo la qualità del codice e prevenendo regressioni durante le modifiche future.
- **Controllo di versione efficace:** L'utilizzo di un sistema di controllo di versione come Git consente di tracciare ogni modifica al codice, facilitando la collaborazione tra gli sviluppatori e permettendo di ripristinare versioni precedenti in caso di necessità.
-

2. Sicurezza: la protezione dei dati al primo posto

La sicurezza è un pilastro fondamentale per proteggere i dati sensibili degli utenti e garantire l'integrità del sistema. Abbiamo implementato diverse misure di sicurezza per mitigare i rischi e proteggere l'applicazione da potenziali minacce:

- **Crittografia:** Le password degli utenti vengono crittografate prima di essere memorizzate nel database. Questo impedisce che, in caso di accesso non autorizzato al database, le password vengano esposte in chiaro.
- **Controlli di accesso rigorosi:** L'accesso alle risorse e alle funzionalità dell'applicazione è regolato da un sistema di autenticazione e autorizzazione basato su ruoli, che garantisce che solo gli utenti autorizzati possano accedere a determinate aree o eseguire determinate azioni.
- **Validazione degli input:** Tutti i dati provenienti dall'utente vengono validati per prevenire attacchi di tipo injection (come SQL injection o cross-site scripting) che potrebbero compromettere l'integrità del sistema.
- **Autenticazione tramite JWT:** L'utilizzo di JSON Web Token (JWT) per l'autenticazione offre un meccanismo sicuro e affidabile per la gestione delle sessioni utente e la verifica delle autorizzazioni.

3. Usabilità: un'esperienza intuitiva per tutti

Un'applicazione usabile è facile da navigare, intuitiva e permette agli utenti di raggiungere i propri obiettivi in modo efficiente. Abbiamo dedicato particolare attenzione alla progettazione dell'interfaccia utente per garantire un'esperienza utente positiva:

- **Design intuitivo:** L'interfaccia utente è stata progettata seguendo principi di design centrati sull'utente, utilizzando elementi visivi chiari e coerenti e organizzando le informazioni in modo logico e intuitivo.
- **Efficienza:** L'interfaccia è stata ottimizzata per consentire agli utenti di completare le attività in modo rapido e semplice, riducendo al minimo il numero di clic e le interazioni necessarie.

1.3.4 Design strutturale

L'applicazione è stata strutturata tramite un'architettura a tre livelli: un approccio comune alla progettazione di applicazioni web che mira a suddividere l'applicazione in tre componenti distinti per garantire una struttura ben organizzata e scalabile. Ogni livello svolge un ruolo specifico e si occupa di determinati aspetti dell'applicazione.



Client – Front-end sviluppato in Angular, che permette di creare applicazioni web dinamiche e scalabili. Basato su TypeScript, offre un'architettura a componenti. Ottiene le informazioni comunicando con il server (verrà approfondito in seguito).

Server – Il back-end è stato sviluppato con **Spring**, framework di java.

Java Spring Framework (Spring Framework) è uno dei framework open source più diffusi a livello aziendale per la creazione di applicazioni Java. Il framework è adatto agli ambienti di produzione che vengono eseguiti sulla Java Virtual Machine (JVM).

Spring Boot è uno strumento che semplifica e accelera lo sviluppo di applicazioni web e microservizi basati su Spring Framework, grazie a tre funzionalità principali:

1. **Configurazione automatica:** Spring Boot può configurare automaticamente le dipendenze necessarie (utilizzando Maven o Gradle) per l'applicazione in modo da evitare la necessità di configurare manualmente ogni singolo componente.
2. **Approccio categorico alla configurazione:** Spring Boot rende più facile e intuitivo organizzare le impostazioni in base alle esigenze dell'applicazione, soprattutto la sezione relativa alla sicurezza.
3. **Capacità di creare applicazioni autonome:** le app possono essere eseguite senza la necessità di un server applicativo esterno.

Tutte queste funzionalità si combinano per fornire uno strumento potente e flessibile che semplifica notevolmente la creazione di applicazioni backend riducendo la necessità di configurazione e installazione.

Inoltre, offre un'architettura modulare, con integrato supporto per **Dependency Injection**. Include moduli come **Spring MVC**, **Spring Data** e **Spring security** (verranno

approfonditi in seguito). Fornisce risorse o servizi quando richiesto dal client. Responsabile, inoltre, della comunicazione con il database.

Database – è stato usato **MySQL** (trattato nella prossima sezione). Sistema di archiviazione e gestione strutturato che consente di organizzare, memorizzare e recuperare dati in modo efficiente.

1.3.5 Ulteriori tecnologie utilizzate

Hibernate e Jpa – Per la **persistenza** dei dati registrati nell'applicativo, ho scelto di utilizzare Hibernate e JPA.

Hibernate è un ampio ecosistema di librerie, un framework open source di Object Relational Mapping (ORM).

È uno strumento di sviluppo Java che consente di mappare i modelli di entità orientati agli oggetti su un database relazionale, consente di salvare i dati in modo permanente dal context di Java al database.

Le specifiche JPA (Java Persistence API) per la persistenza dei dati, garantiscono una maggiore portabilità delle applicazioni.

Docker – Docker è una piattaforma che consente di **containerizzare** tutte le componenti dell'applicazione, garantendo un ambiente di esecuzione **coerente e portabile** su qualsiasi sistema operativo che supporti Docker (verrà approfondito in seguito).

Nel nostro caso, Docker verrà utilizzato per containerizzare:

- **Front-end (Angular)** → Per garantire un ambiente stabile e riproducibile.
- **Back-end (Spring Boot)** → Isolando le dipendenze e semplificando il deployment.
- **RabbitMQ** → Per la gestione affidabile della messaggistica tra i servizi.
- **Database (MySQL)** → Evitando problemi di configurazione locale.
- **PHPMyAdmin** → Per una gestione più intuitiva del database.

Grazie all'uso di **Docker Compose**, sarà possibile avviare e gestire facilmente più container in un'unica configurazione, semplificando lo sviluppo e il deployment.

L'approfondimento successivo includerà dettagli sulla creazione dei **Dockerfile**, la gestione dei **volumi** e la configurazione dei **network** tra i container.

GitHub – Piattaforma di hosting per il controllo di versione e la collaborazione sul codice, basata su **Git**. Consente agli sviluppatori di **gestire** repository, **tracciare** modifiche, **collaborare** su progetti open-source o privati e **automatizzare** flussi di lavoro tramite GitHub Actions.

GitHub offre strumenti avanzati come:

- **Branching e Pull Requests** → Per una gestione efficiente del codice e delle revisioni.
- **Issues e Project Boards** → Per tracciare bug, feature e organizzare il lavoro.
- **GitHub Actions** → Per l'integrazione e il deployment continuo (CI/CD).
- **Security Features** → Per la scansione delle vulnerabilità nel codice e la protezione dei repository.

Maven - uno strumento open source per la gestione e l'automazione della build di progetti Java. La sua funzione principale è quella centralizzare le dipendenze del progetto, aiutando in fase di compilazione.

Il progetto è organizzato in una struttura predefinita con una serie di file di configurazione, tra cui il file **pom.xml** (Project Object Model). Questo file contiene informazioni sul progetto, come la versione, le dipendenze, i plugin, i test e le risorse necessarie per la compilazione.

2 Database

2.1 Diagramma ER

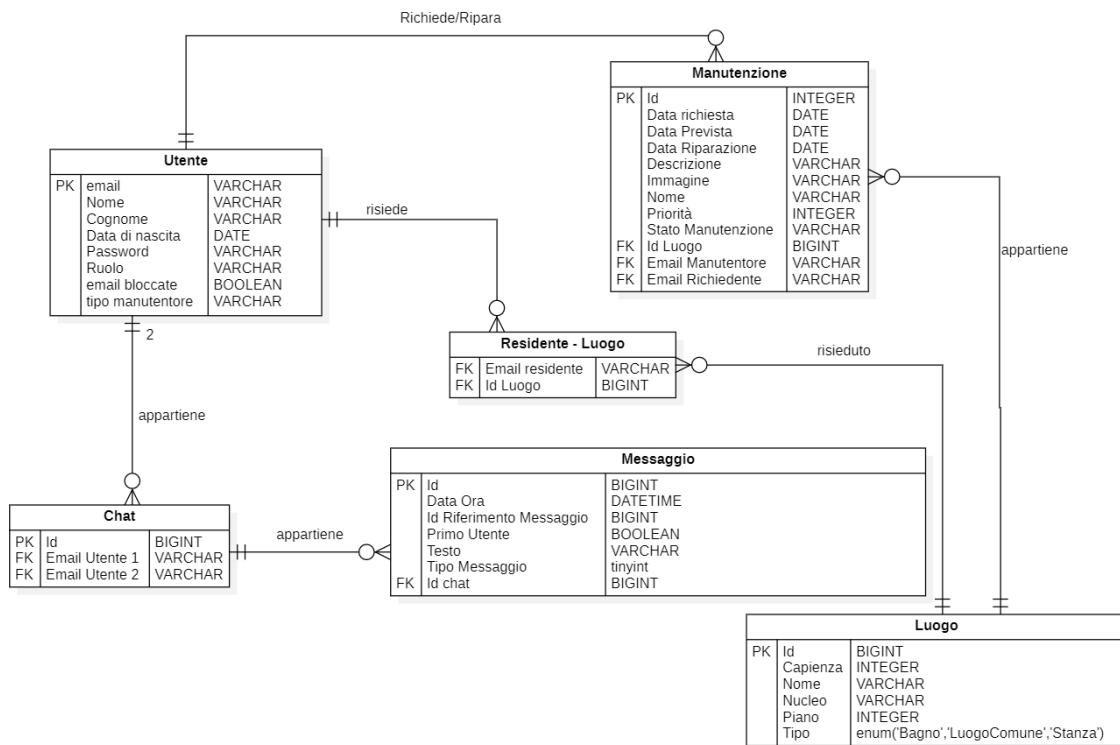


Diagramma ER – Contiene le entità (tabelle), gli attributi (campi) e le associazioni tra le tabelle stesse.

L'associazione molti a molti è stata risolta introducendo, come da prassi, un'entità aggiuntiva che risolve il problema interponendosi tra le 2 entità in questione e trasformando l'associazione in 2 associazioni 1 a molti. Si ha dunque l'entità **Residente - Luogo**.

Il campo **Ruolo** all'interno dell'utente è di tipo varchar e riferisce alla classe enum chiamata **Ruolo**. A differenza dell'enum **Tipo** (o a **Tipo Messaggio**) all'interno di **Messaggio** che è di tipo tinyint in cui ogni valore numerico viene associato a un enum) presente all'interno del luogo (che è di tipo enum), il ruolo è varchar perché il campo viene inserito automaticamente in base all'istanza creata. Perciò, se si crea un utente di tipo Residente, JPA automaticamente assegna il valore al campo Ruolo tenendo in considerazione il tipo di utente creato.

L'associazione Utente – Chat è N-2 perché un utente può avere N chat ma una chat è solo per 2 persone (infatti c'è l'annotazione 2 affianco all'associazione).

2.2 Note implementative

È stato scelto di affidarsi **all'RDBMS** (relational database management system) MySQL. Questa scelta può essere motivata da diverse ragioni:

- essendo open source, gode di un **ampio supporto** della comunità e di una vasta documentazione;
- è rinomato per la sua **velocità** ed **efficienza** nelle operazioni di lettura e scrittura;
- offre una facile **scalabilità**, rendendolo adatto a progetti di diverse dimensioni e complessità;
- Inoltre, grazie alla sua **compatibilità** con molte piattaforme e linguaggi di programmazione, rappresenta una soluzione versatile per lo sviluppo di applicazioni moderne.

3 Back-end

3.1 Progettazione architetturale

3.1.1 Pattern MVCS

È stato deciso di implementare il pattern architetturale **MVCS** (Model-ViewController-Service), ovvero una variante del più comune pattern MVC. È utilizzato per **organizzare** e **separare** le responsabilità all'interno di un'applicazione software complessa, consentendo una migliore manutenibilità, scalabilità e riusabilità del codice.

Di seguito una breve descrizione di ciascun componente del pattern:

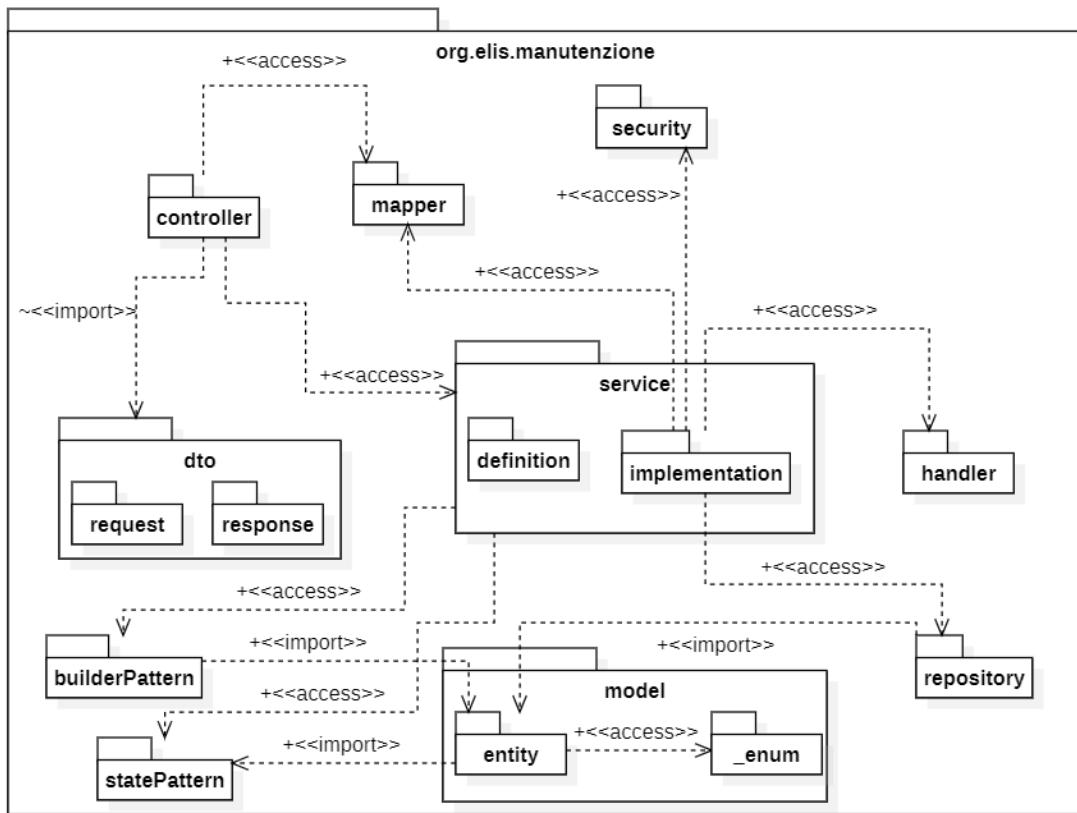
Model – Contiene le entità di **dominio** (opportunamente mappate sul database) e i metodi di accesso ai dati (rappresentati dalle **repository**). Utilizza JPA e il framework **Hibernate** per mappare le entità Java alle tabelle del database. Le repository sono la componente che servono per accedere e manipolare i dati.

View – Responsabile **dell'interfaccia** utente dell'applicazione. In un'architettura con Angular come front-end, la View è costituita da **componenti Angular** che gestiscono la presentazione dei dati e l'interazione dell'utente. I dati vengono recuperati tramite chiamate agli endpoint **REST** esposti dal back-end e aggiornati in tempo reale grazie al **data binding**. Il back-end restituisce dei **DTO** (Data Transfer Objects), oggetti utilizzati per trasferire dati tra i diversi livelli dell'applicazione, senza esporre direttamente le entità del database. I **mapper** si occupano della conversione tra oggetti di tipi diversi, come ad esempio tra entità del database e DTO.

Controller – Classi che rappresentano l’interfaccia di accesso all’applicazione. Rimangono in ascolto e **intervettano le richieste HTTP** da parte del client e **indirizzano** la richiesta al service adatto. Ogni controller espone un endpoint RESTful, tramite il framework **Spring Web MVC**, che risponde a richieste di tipo HTTP (GET, POST, PUT, DELETE, PATCH).

Service – Insieme di classi contenente la **business logic**. Sollevano i controller da alcuni compiti, come **flussi logici** complessi o **comunicazione con risorse esterne**.

3.1.2 Packaging



Package diagram – Sono riportati tutti i package del progetto. I collegamenti (rilevanti) rappresentano diversi tipi di dipendenze tra i package: un package importa le classi dell’altro oppure un package accede ai metodi delle classi dell’altro package.

Controller – il package rappresenta l’insieme di classi controller che sono **responsabili** della gestione delle **richieste** dell’utente tramite chiamata **HTTP**. Chiama i Service e restituisce poi la risposta.

Service – package che contiene due sotto package (definition e implementation). Nel package **definition** si trovano le **interfacce** che poi saranno implementate dalle classi presenti nel package implementation. L’obiettivo delle classi Service è quello di avere all’interno tutta la **logica**. Vengono chiamati dai controller e ritornano ad essi il risultato delle operazioni effettuate sugli oggetti o sul db.

DTO - oggetti utilizzati per **trasferire** dati tra i diversi livelli dell’applicazione, senza esporre direttamente le entità del database. All’interno troviamo due sotto package: in **request** troviamo gli oggetti che arrivano dal front-end e poi devono essere elaborati; in **response** ci sono gli oggetti che, dopo essere stati elaborati, vengono ritornati al front-end.

Mapper - si occupano della **conversione** tra oggetti di tipi diversi, come ad esempio tra entità del database e DTO.

BuilderPattern – all’interno si trovano le classi del Pattern Builder. Vengono usate dai service per **creare** gli oggetti che rappresentano i model.

StatePattern – all’interno si trovano le classi dello State Pattern. Vengono usate per definire lo **stato** della manutenzione.

Model – rappresentato da due sotto package, chiamati **model** e **_enum**. All’interno del **model** ci sono le entità di **dominio**. Invece, all’interno di **_enum** ci sono le classi di tipo enum che servono per definire un insieme fisso di **costanti**.

Repository – contiene le interfacce che gestiscono **l’accesso ai dati** nel database. Vengono utilizzate per operazioni **CRUD** (create, read, update, delete). Inoltre, c’è anche una classe che si occupa di **filtrare** gli oggetti nel db tramite i filtri inseriti dall’utente.

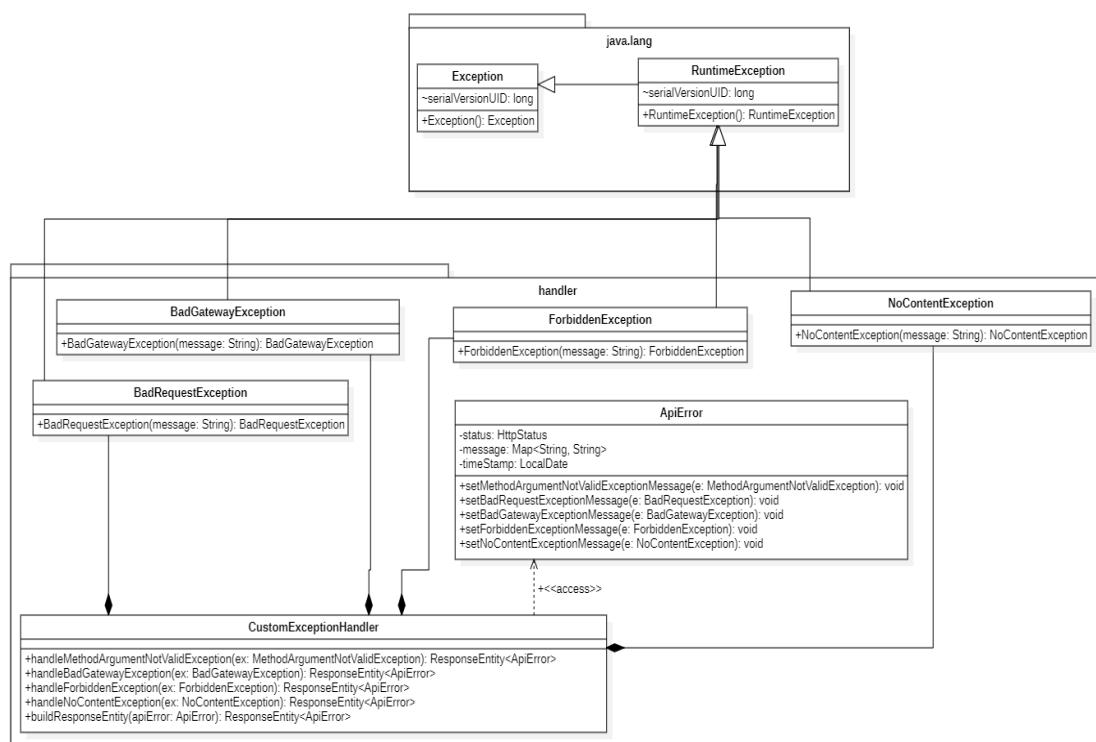
Security – contiene le classi che servono ad implementare la **sicurezza** dell’applicazione. Esse intervengono per ogni chiamata fatta al back-end.

Handler – contiene le classi usate per implementare le **eccezioni custom**. Infatti, ad ogni eccezione che potrebbe essere lanciata all’interno dell’applicazione è associata una classe custom di Exception. Inoltre, ci sono le classi **ApiError** (che contiene i dati dell’eccezione lanciata) e **CustomExceptionHandler** (che si occupa di “rintracciare” le eccezioni lanciate per poi creare l’exception custom).

3.2 Exception

Durante l'uso dell'applicazione, può avvenire che una richiesta HTTP vada incontro ad anomalie o errori. È una buona pratica **catturare gli errori** e notificare il client di quanto accaduto. È stato, quindi, implementato un sistema di gestione delle eccezioni composto da **un'entità con messaggio, status e data** come attributi, un **controller** responsabile dell'invio della risposta al client e un insieme di **classi figlie di RuntimeException** fatte ad hoc in base al tipo di errore rilevato.

3.2.1 Exception Class Diagram



Ogni classe di custom exception è figlia di RuntimeException (eccezioni che si verificano durante l'esecuzione), a sua volta figlia della classe **Exception**, che è la classe base per tutte le eccezioni. Ognuna di queste classi ha un costruttore a cui viene passata la stringa di errore generato.

La classe **ExceptionHandler** rintraccia ogni eccezione lanciata e ognuna di queste ha un metodo proprio, il quale va a richiamare, nella classe **ApiError**, il metodo associato a quell'eccezione.

ApiError andrà a **costruire** il messaggio di errore che sarà poi ritornato a CustomExceptionHandler e successivamente sarà ritornato all'applicazione. Il messaggio di errore comprenderà le informazioni più importanti: status HTTP, messaggio e data.

3.3 Dominio

Il dominio dell'applicazione è rappresentato dal package model, che contiene gli oggetti fondamentali che descrivono il problema affrontato dal sistema. Questo package è suddiviso in sotto-package per organizzare meglio le entità e i tipi di dati utilizzati.

entity – contiene le **classi Java** che modellano gli oggetti del dominio dell'applicazione, come Utente, Chat, Messaggio, Manutenzione, ecc. Queste classi sono mappate sul database utilizzando **Hibernate**, il framework **ORM** (Object-Relational Mapping) che permette di lavorare con entità Java senza dover scrivere SQL manualmente.

_enum – Il sotto-package _enum contiene **tipi di dati enumerativi**, utilizzati per rappresentare insiemi finiti di valori predefiniti. Vantaggi nell'utilizzo:

- **Miglior leggibilità** → Definiscono chiaramente i valori possibili;
- **Evitano errori di input** → Non permettono valori non previsti;
- **Efficienza** → Sono implementati come costanti statiche, senza necessità di lookup nel database.

```
Luogo.java


```
@Enumerated(EnumType.STRING)
private Tipo tipo;
```


```

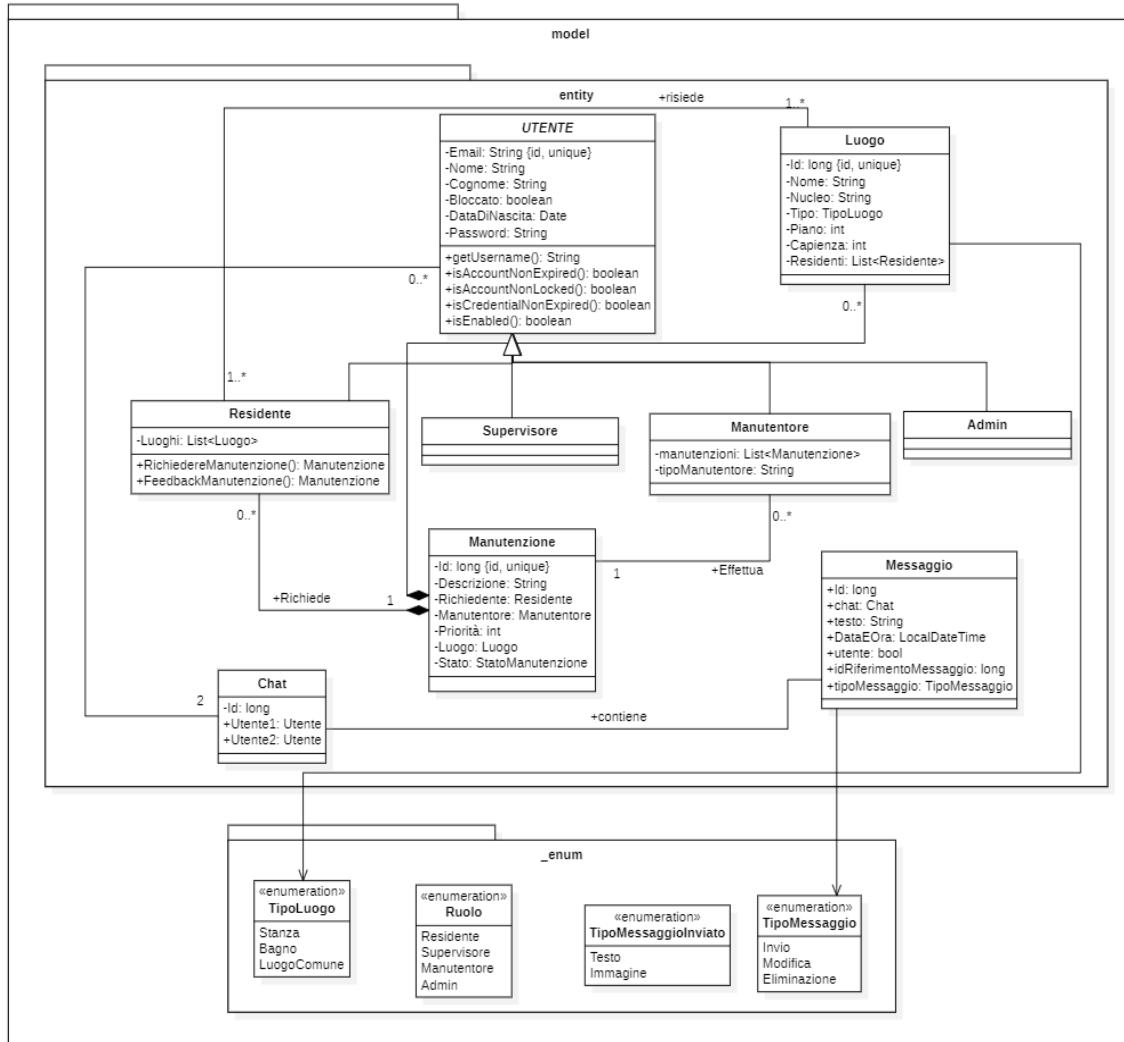
```
Tipo.java


```
public enum Tipo{
 Bagno,
 Stanza,
 LuogoComune;
}
```


```

Questo è un esempio dell'uso dell'enum Tipo, che definisce i tipi di Luogo.

3.3.1 Class Diagram



Class diagram del package model. L'utente è una classe astratta che contiene i dati generici per ogni utente dell'applicazione. Ci sono 4 tipi diversi di utente e ognuno di essi è una classe che estende la classe Utente; questi sono:

Residente – è associato alla classe manutenzione, con un'associazione di tipo **composizione** visto che la manutenzione non esiste senza un residente, perchè può richiedere una o più manutenzioni. È anche associato alla classe luogo perchè all'interno del college avrà a disposizione una stanza e un bagno.

Supervisore – è l'utente incaricato di filtrare le manutenzioni, infatti dovrà decidere se rifiutare una manutenzione richiesta da un residente, oppure potrà filtrarla e indirizzarla al manutentore più adatto.

Manutentore – è l'utente incaricato ad eseguire le manutenzioni all'interno del college. È collegato con la manutenzione per ovvi motivi.

Admin – è l'utente le cui uniche mansioni che svolge sono amministrative e sono: gestione degli utenti e gestione dei luoghi.

L'utente, inoltre, è anche collegato alla classe chat. Ogni istanza della classe chat avrà 2 utenti. La chat è collegata alla classe messaggio perchè ogni messaggio apparirà ad una chat.

Alcuni enum non sono collegati a classi perchè servono semplicemente per il back-end.

3.4 Design Pattern

L'uso dei design pattern aiuta a **ridurre la complessità** del software, migliorando la **manutenibilità** e **l'efficienza**. Tuttavia, è importante utilizzarli in modo appropriato, evitando l'implementazione quando non necessario. Di seguito la descrizione dei pattern applicati e i vantaggi di cui gode l'applicazione di conseguenza.

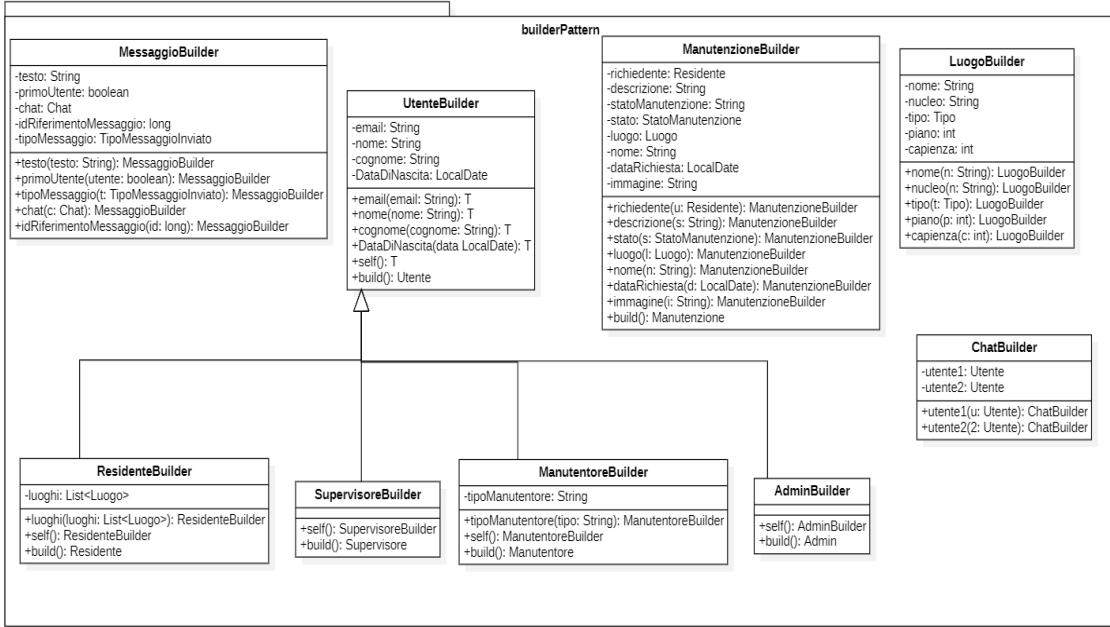
3.4.1 Builder

Pattern **creazionale** il cui scopo è creare oggetti, passo dopo passo. Nello specifico, invece di avere il singolo costruttore con molti parametri, il pattern Builder separa il processo di costruzione dell'oggetto in una serie di passi, consentendo di personalizzare l'oggetto in modo flessibile. I principali motivi per cui è stato scelto di implementarlo sono:

Testabilità – Poiché vengono separati chiaramente i passi di costruzione, è più semplice scrivere test unitari che coprano ciascun passo senza dover eseguire l'intero processo di costruzione.

Flessibilità – Il pattern Builder consente di costruire oggetti complessi con configurazioni flessibili. È possibile specificare solo le opzioni necessarie, evitando parametri opzionali confusi e valori nulli.

3.4.1.1 Builder Class Diagram



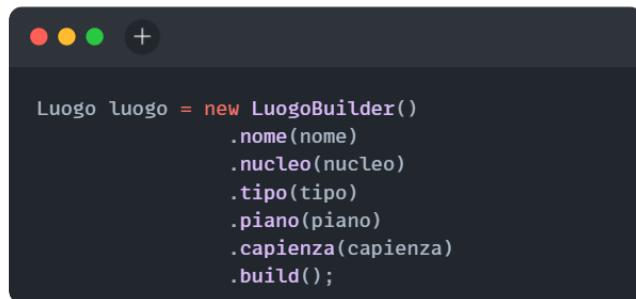
Class diagram del package builder. Sono presenti le entity e ognuna di queste ha all'interno gli attributi che servono in fase di creazione. Infatti, per esempio, nella tabella **LuogoBuilder** manca la lista di residenti, questo perché in fase di creazione del luogo ancora non gli saranno associati dei residenti, cosa che ovviamente sarà fatta in seguito.

```
public MessaggioBuilder testo(String testo) {
    this.testo = testo;
    return this;
}
```

Setter dell'attributo testo che **ritorna l'istanza** del builder stesso, così da poter concatenare altri metodi.

```
public Messaggio build() {
    return new Messaggio(testo, primoUtente, chat, idRiferimentoMessaggio, tipoMessaggio);
}
```

Metodo **build** che si occupa del **casting** da builder a entity. Questo metodo va a richiamare il costruttore della classe di riferimento passandogli tutti i parametri necessari.



```
Luogo luogo = new LuogoBuilder()
    .nome(nome)
    .nucleo(nucleo)
    .tipo(tipo)
    .piano(piano)
    .capienza(capienza)
    .build();
```

Esempio di creazione di un oggetto Luogo usando LuogoBuilder.

Questo, appunto, avviene perché ogni metodo ritorna l'istanza del builder stesso e dopo aver settato tutti i parametri richiama il metodo build, che, come detto precedentemente, richiama il costruttore della classe Luogo.

3.4.2 State Pattern

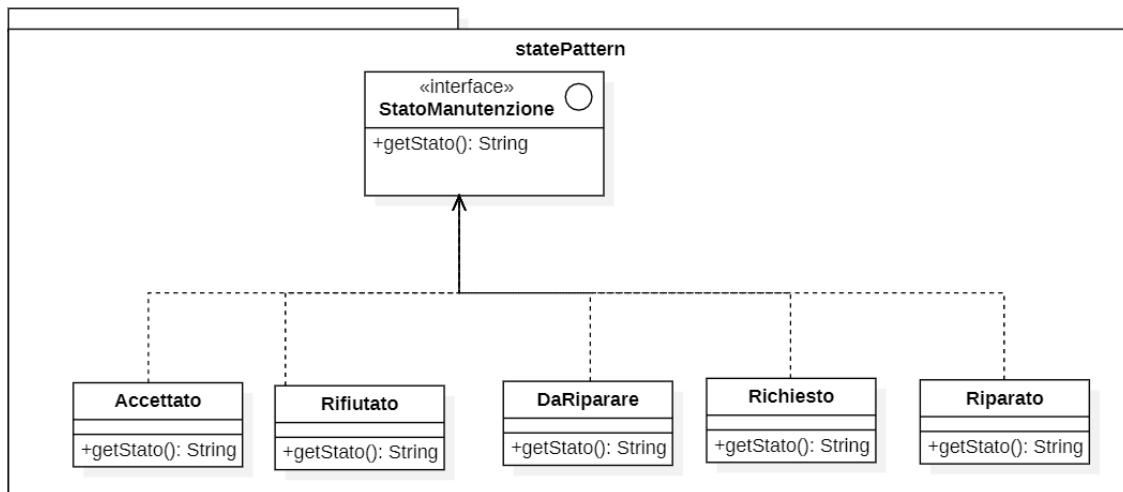
Pattern di tipo **comportamentale** il cui obiettivo è quello di **cambiare** il comportamento del programma in base allo stato di un oggetto (in questo caso della manutenzione). I principali vantaggi per cui è stato scelto questo pattern sono:

Scalabilità: è possibile aggiungere e rimuovere stati senza dover modificare l'oggetto. Questo consente una facile estensione del programma senza dover modificare troppe parti del codice.

Manutenibilità: Il pattern separa la logica di ogni stato in classi diverse, rendendo il codice più chiaro, organizzato e facile da modificare.

Cambi di stato dinamici e controllati: L'oggetto manutenzione può cambiare stato in modo dinamico senza che il resto dell'applicazione debba preoccuparsene. Il cambio di stato è controllato e ben definito.

3.4.2.1 State Pattern Class Diagram



Class diagram del package statePattern.

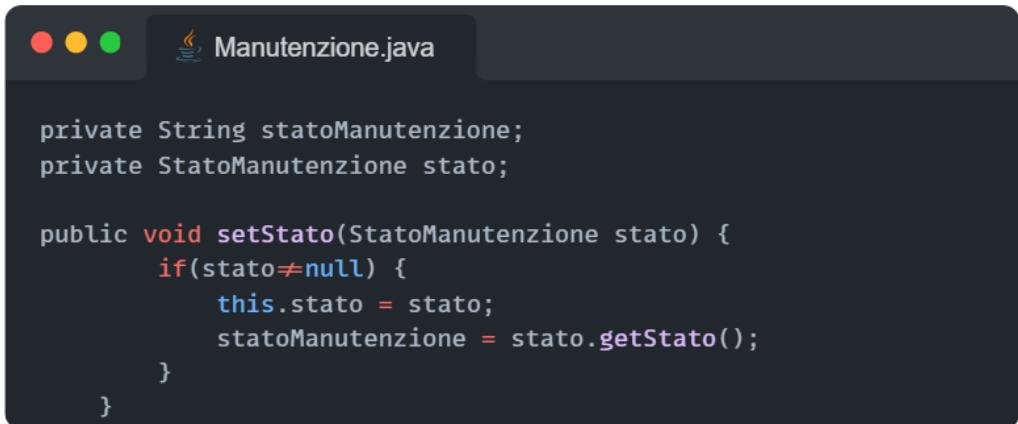
All'interno è presente un'interfaccia **StatoManutenzione** la quale contiene il metodo **getStato**, il quale verrà poi usato dalle classi concrete.

Ogni classe concreta prende il nome dello stato della manutenzione e all'interno ha semplicemente il getter dello stato, questo servirà ad assegnare lo stato corretto alla manutenzione.

A screenshot of a Java code editor window titled "Accettato.java". The code contains a single method:

```
public String getStato() {
    return "Accettato";
}
```

Getter dello stato all'interno dell'istanza di uno stato. Questo ritorna una stringa, successivamente sarà spiegato il perchè.



```
Manutenzione.java

private String statoManutenzione;
private StatoManutenzione stato;

public void setStato(StatoManutenzione stato) {
    if(stato!=null) {
        this.stato = stato;
        statoManutenzione = stato.getStato();
    }
}
```

Setter dello stato della manutenzione. Prende in input un'istanza di StatoManutenzione



```
Manutenzione.java

private String statoManutenzione;
private StatoManutenzione stato;

public StatoManutenzione getStato(){
    if(stato == null && statoManutenzione != null){
        switch (statoManutenzione) {
            case "Accettato" → stato = new Accettato();
            case "Rifiutato" → stato = new Rifiutato();
            case "Riparato" → stato = new Riparato();
            case "Da riparare" → stato = new DaRiparare();
            default → stato = new Richiesto();
        };
    }
    return stato;
}
```

Getter dello stato della manutenzione. Questo controlla ciò che gli ritorna il metodo **getStato** (ricordo essere una stringa) dello statoManutenzione e per ogni ritorno crea un'istanza di quel tipo di stato della manutenzione.

3.5 Sicurezza e autorizzazione

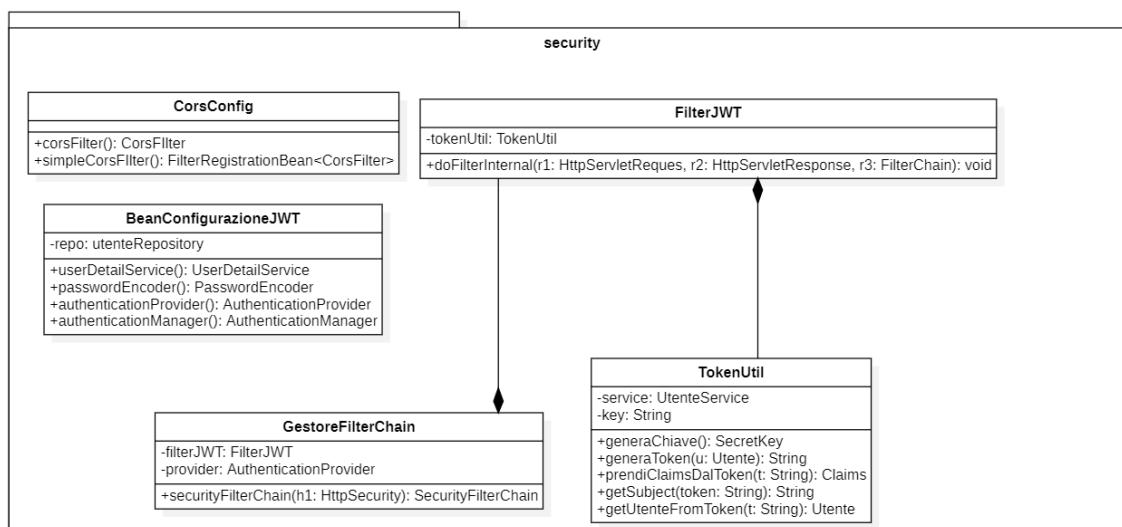
Per garantire un elevato livello di sicurezza nell'applicazione web, è stato implementato il modulo **Spring Security**, che gestisce l'autenticazione e l'autorizzazione degli utenti. Il sistema assicura che solo gli utenti autorizzati possano accedere alle risorse protette, adottando un meccanismo basato su **JSON Web Token (JWT)** per l'autenticazione stateless.

Processo di Autenticazione

Quando un utente tenta di accedere all'applicazione fornendo le proprie credenziali (username e password), il sistema segue i seguenti passaggi:

1. **Ricerca dell'utente** → Viene cercato l'utente con l'e-mail inserita
2. **Verifica della password** → Spring Security confronta la password inserita con quella salvata nel database, che è memorizzata in formato cifrato mediante BCrypt.
3. **Generazione del Token JWT** → Se l'autenticazione ha successo, il server genera un token JWT contenente informazioni chiave sull'utente (es. ID, e-mail, ruoli).
4. **Invio del Token al Client** → Il token viene restituito al client, che lo utilizza per autenticarsi nelle richieste successive senza dover reinserire le credenziali.
5. **Protezione delle API** → Le richieste future del client includono il token nell'header Authorization (Bearer <token>), e il server lo valida per concedere o negare l'accesso alle risorse protette.

3.5.1 Security Class diagram



Il diagramma di classe illustra l'architettura del modulo di sicurezza dell'applicazione, basato su **Spring Security** e **JWT** per l'autenticazione degli utenti. Di seguito, vengono descritte le principali classi e le loro responsabilità.

CorsConfig - Questa classe si occupa della configurazione delle **CORS** (Cross-Origin Resource Sharing) per consentire al front-end di comunicare con il back-end senza restrizioni di dominio.

- **corsFilter()**: Restituisce un oggetto **CorsFilter** che definisce le regole per le richieste HTTP cross-origin.
- **simpleCorsFilter()**: Registra il filtro CORS come un **Bean** in Spring.

BeanConfigurazioneJWT - Gestisce la configurazione dei componenti necessari per la sicurezza e l'autenticazione.

- **repo**: utenteRepository → Accesso al database degli utenti.
- **userDetailsService()**: Carica i dettagli dell'utente per l'autenticazione.
- **passwordEncoder()**: Definisce il **PasswordEncoder** (BCrypt) per la cifratura delle password.
- **authenticationProvider()**: Fornisce il meccanismo di autenticazione.
- **authenticationManager()**: Gestisce il processo di autenticazione.

FilterJWT - È un **filtro** che intercetta le richieste HTTP per validare il token JWT.

- **tokenUtil**: TokenUtil → Classe di supporto per la gestione dei token JWT.
- **doFilterInternal(request, response, filterChain)**:
 1. **Estrae il token** dalla richiesta HTTP.
 2. **Lo valida** e recupera l'utente corrispondente.
 3. **Imposta l'autenticazione nel SecurityContext**.

TokenUtil - Classe di utilità per la gestione dei JSON Web Token (JWT).

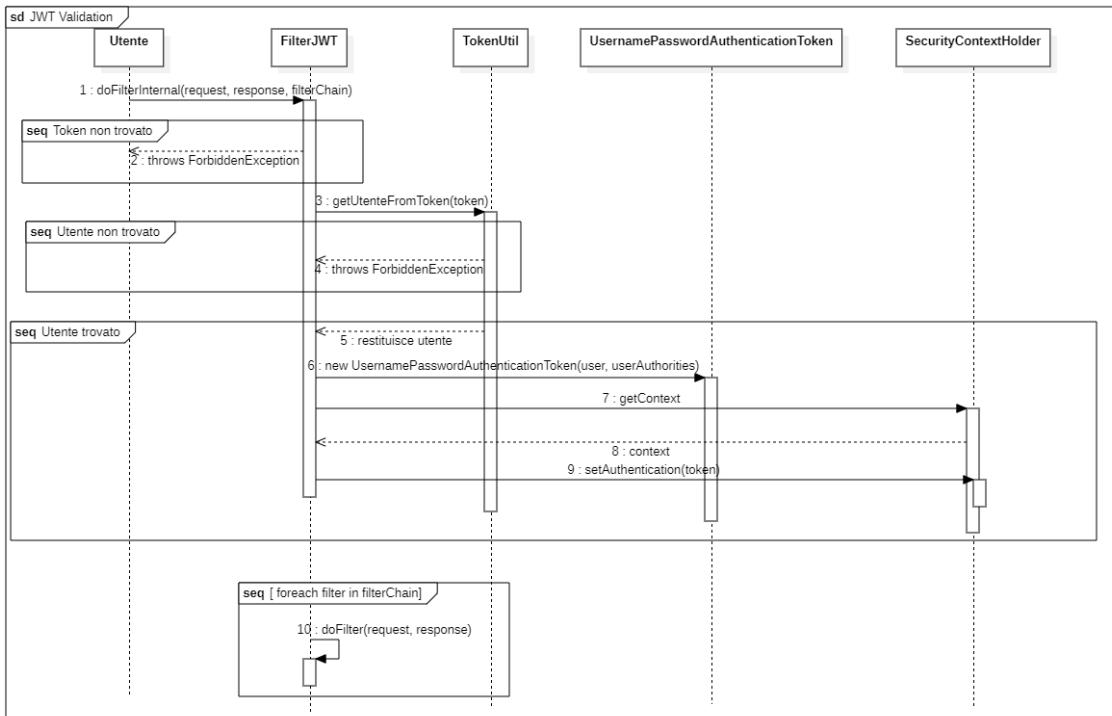
- **service**: UtenteService → Servizio che gestisce gli utenti.
- **generaChiave()**: Genera una chiave **SecretKey** per la firma del token.
- **generaToken(utente)**: Crea un token JWT a partire dai dati dell'utente.
- **prendiClaimsDalToken(token)**: Estraie le informazioni (claims) dal token.
- **getSubject(token)**: Recupera il nome utente dal token.

- **getUtenteFromToken(token):** Decodifica il token e restituisce l'utente corrispondente.

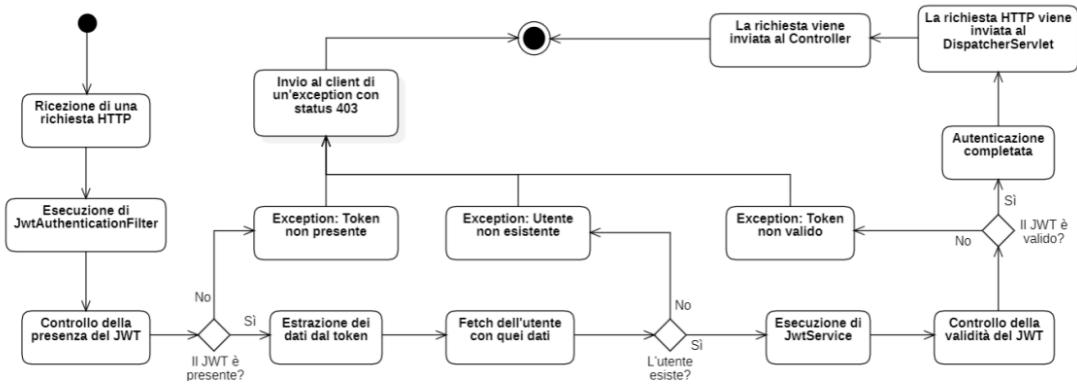
GestoreFilterChain - Configura la **catena dei filtri di sicurezza** in Spring Security.

- **filterJWT:** FilterJWT → Integra il filtro JWT nella catena di sicurezza.
- **provider:** AuthenticationProvider → Definisce il provider di autenticazione.
- **securityFilterChain(httpSecurity):** Configura la catena di sicurezza, definendo quali endpoint richiedono autenticazione e abilitando il filtro JWT.

3.5.2 Security Sequence diagram



3.5.3 Security Activity diagram

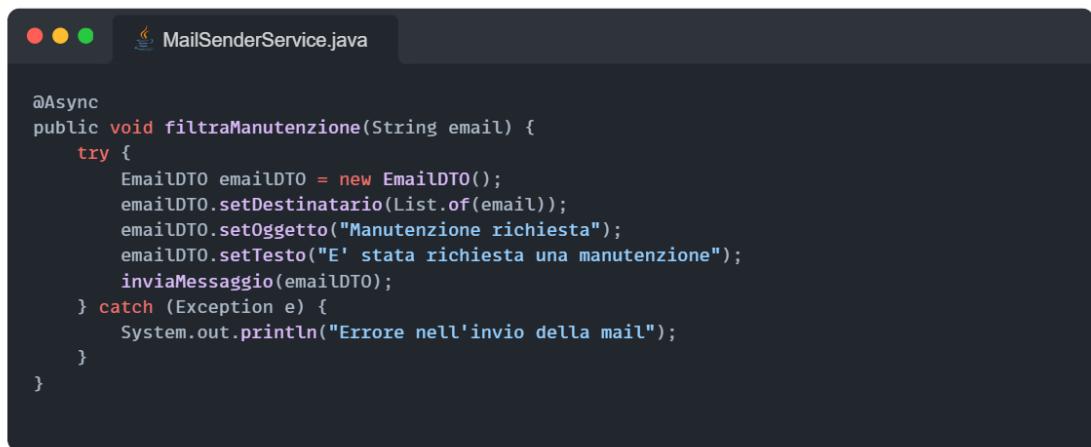


Activity diagram che rappresenta l'esecuzione di Spring Security all'arrivo di ogni richiesta HTTP. Questo tipo di diagramma si focalizza maggiormente su attività, decisioni e flussi di controllo.

L'activity diagram in alto spiega graficamente il flusso principale gestito da Spring Security, cioè quello per cui ogni richiesta effettuata al backend passa prima da una serie di controlli atti a determinare se il client che fa la richiesta è autorizzato e può proseguire oppure se un codice 403 Forbidden debba esser inviato come risposta. Fondamentalmente, Spring Security impone che, ad per ogni richiesta inviata ai controller dell'applicazione venga verificata la presenza di un'informazione specifica nell'header della richiesta stessa: il **JWT** (JSON Web Token - un sistema di cifratura e scambio di informazioni in formato JSON tra i servizi esposti, generato come token cifrato e trasmesso come stringa). Se questa stringa particolare è presente, Spring Security può procedere con ulteriori controlli che consentono di ottenere informazioni sull'utente che ha effettuato la richiesta, poiché è l'utente loggato. Se tali informazioni non possono essere ottenute, significa che la richiesta proviene da un utente non loggato e quindi non autorizzato ad effettuare tale richiesta. È importante notare che login e password dimenticata sono "esclusi" da questo flusso poiché possono essere richiamati da qualsiasi visitatore esterno, per i quali non è necessario eseguire questi controlli per determinare le autorizzazioni di accesso necessarie.

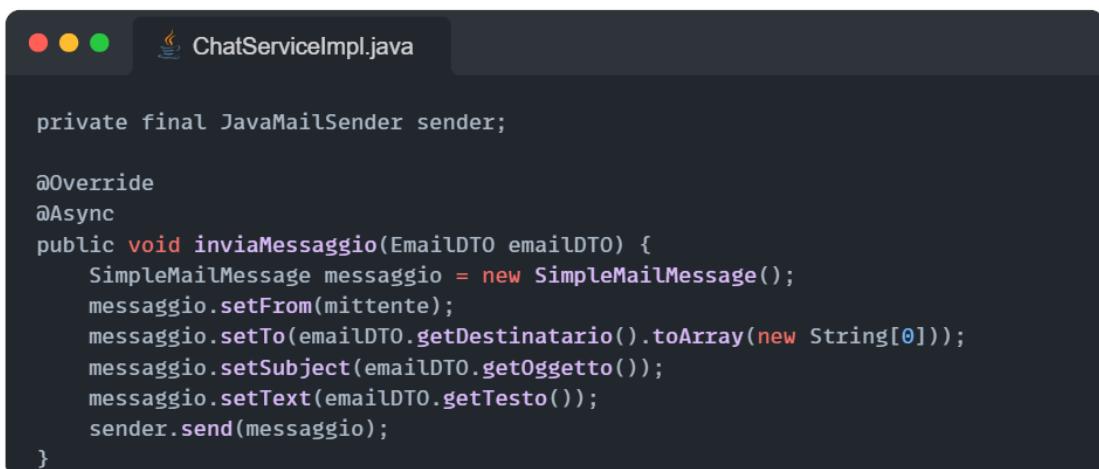
3.6 Mailing

Si è optato per l'implementazione di un sistema di mailing al fine di comunicare con gli utenti in occasione di situazioni specifiche. Tra gli esempi rientrano la creazione e-mail quando un residente richiede una manutenzione e la generazione di una nuova password.



```
@Async  
public void filtramanutenzione(String email) {  
    try {  
        EmailDTO emailDTO = new EmailDTO();  
        emailDTO.setDestinatario(List.of(email));  
        emailDTO.setOggetto("Manutenzione richiesta");  
        emailDTO.setTesto("E' stata richiesta una manutenzione");  
        inviaMessaggio(emailDTO);  
    } catch (Exception e) {  
        System.out.println("Errore nell'invio della mail");  
    }  
}
```

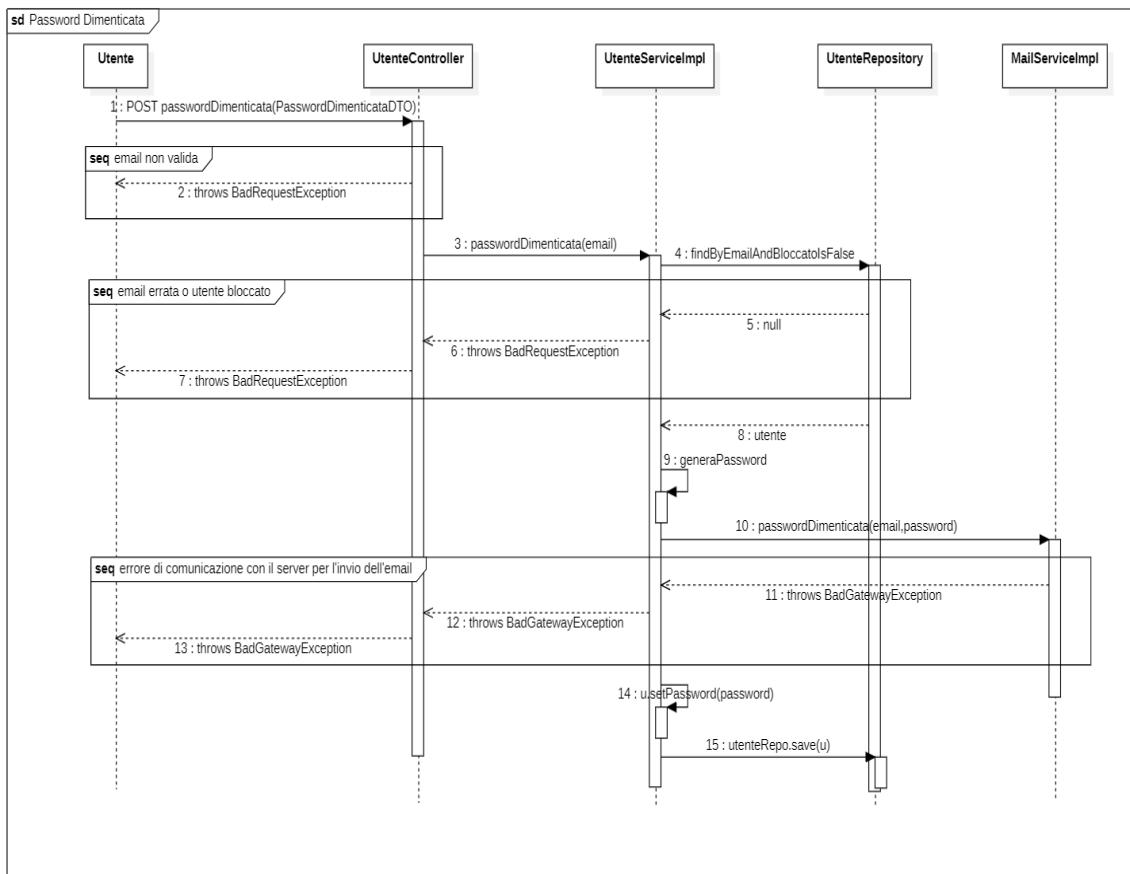
Metodo che prepara i campi da inserire nell'email. Si può notare che il metodo è asincrono, questo fa sì che l'utente non debba aspettare l'invio dell'e-mail continuando la navigazione all'interno dell'applicazione con più fluidità. Nel caso ci fosse un problema durante l'invio dell'e-mail, questa operazione non viene né segnalata all'utente, né rieseguita; questo perché è una email di poca rilevanza (dato che per questioni lavorative il supervisore sarà molto spesso impegnato all'interno dell'applicazione e si accorgerebbe della manutenzione richiesta), a differenza di email come la registrazione, la quale se non fosse inviata non permetterebbe all'utente di sapere le sue credenziali.



```
private final JavaMailSender sender;  
  
@Override  
@Async  
public void inviaMessaggio(EmailDTO emailDTO) {  
    SimpleMailMessage messaggio = new SimpleMailMessage();  
    messaggio.setFrom(mittente);  
    messaggio.setTo(emailDTO.getDestinatario().toArray(new String[0]));  
    messaggio.setSubject(emailDTO.getOggetto());  
    messaggio.setText(emailDTO.getTesto());  
    sender.send(messaggio);  
}
```

Metodo che effettivamente invia l'e-mail.

3.6.1 Mailing Sequence Diagram



3.7 Chat

Una funzionalità che permette tanta **interazione** tra gli utenti è la chat. Il servizio è in **tempo reale** ed è stato utilizzato il broker di messaggistica **RabbitMQ**. RabbitMQ permette di inviare dei messaggi che vengono inviati in una coda e a cui l'utente farà accesso per ricevere il messaggio. La chat è molto ricca di funzionalità, infatti permette agli utenti di inviare messaggi, modificarli, eliminarli, indicarli per rispondere direttamente a quel messaggio, inviare immagini e taggare manutenzioni.

I vantaggi di RabbitMQ sono:

- **Inviare e ricevere i messaggi in tempo reale:** questo è possibile grazie al fatto che utilizza una web socket, chiamata Stomp.
- **Sicurezza:** ogni messaggio viene inviato a un Exchange che tramite la routing key decide a quale coda inviare il messaggio. Avendo impostato come routing key l'e-mail del destinatario, solo quest'ultimo può vedere i messaggi a lui inviati.

- **Semplicità:** semplice da integrare perchè non bisogna comunicare direttamente con le web socket



```
CustomMessage customMessage = new CustomMessage();
customMessage.setContent(testo);
customMessage.setSender(mittente);
customMessage.setType(TipoMessaggio.INVIO);
customMessage.setIdRiferimentoMessaggio(idMessaggioRiferimento);
customMessage.setTipoMessaggio(tipoMessaggio);
rabbitTemplate.convertAndSend(
    "TopicTestExchange", emailDestinatario,
    objectMapper.writeValueAsString(customMessage));
```

Esempio di come viene inviato il messaggio. Viene costruito il messaggio da inviare. L'ultima riga di codice è il vero e proprio invio con RabbitMQ. Il primo parametro è **'exchange** ("l'intradattore"), il secondo è la **routing key** (indica la coda a cui l'exchange deve inviare i messaggi), il terzo è **'l'oggetto** del messaggio inviato.



```
CustomMessage customMessage = new CustomMessage();
customMessage.setIdMessage(m.getId());
customMessage.setType(TipoMessaggio.ELIMINAZIONE);
rabbitTemplate.convertAndSend("TopicTestExchange", u2.getEmail(),
    objectMapper.writeValueAsString(customMessage));
```

Questo invece è un esempio della **cancellazione** del messaggio. Dopo aver eliminato il messaggio dal db, viene inviata una notifica di tipo eliminazione e il front-end eliminerà il messaggio dalla chat.

3.8 Testing

Il **testing unitario** è un processo che verifica il comportamento corretto delle unità fondamentali del codice (come **classi** e **metodi**) in **isolamento**. L'obiettivo principale è garantire che ogni componente funzioni indipendentemente, senza dipendere dal resto del sistema.

Questa pratica porta numerosi vantaggi:

- **Identificazione precoce dei bug** → Le anomalie vengono rilevate subito, riducendo i costi di correzione.
- **Miglioramento della qualità del codice** → Scrivere test promuove un design più modulare e meno accoppiato.
- **Facilità di refactoring** → Se il codice è coperto da test, è possibile modificarlo con maggiore sicurezza.

Nell'implementazione dei test unitari, vengono utilizzati strumenti come **JUnit** e **MockMvc**, particolarmente utili nelle applicazioni **Spring Boot**, mentre per simulare il database viene utilizzato H2, un database in-memory.

- **JUnit** è il framework di testing unitario più diffuso in Java, utilizzato per scrivere ed eseguire test automatizzati. È uno strumento fondamentale per garantire la qualità del codice, facilitare il debugging e supportare il refactoring sicuro. Caratteristiche principali:
 - Semplicità** → Fornisce annotazioni intuitive per definire i test.
 - Isolamento** → Ogni test viene eseguito separatamente senza influenzare gli altri.
 - Integrazione con Spring Boot** → Utilizzato per testare **servizi**, **repository** e **controller** in combinazione con MockMvc e H2.

- **MockMvc**, Nel contesto di **Spring Boot**, oltre ai test unitari sui servizi e repository, è fondamentale testare i **controller**, ossia i punti di ingresso delle API REST. **MockMvc** consente di simulare le richieste HTTP senza avviare un server, garantendo test veloci ed efficaci. Il funzionamento di MockMVC è spiegato in questi semplici caratteristiche:
 - Simula richieste HTTP** (GET, POST, PUT, DELETE).
 - Verifica la risposta** (status, header, body).

Evita il bisogno di un server reale → Il test è eseguito interamente in memoria.

- **H2** è un database **relazionale leggero**, scritto in **Java**, che può essere eseguito in modalità **in-memory** o su **file**. È particolarmente utile nei test perché non richiede configurazioni complesse e viene eliminato automaticamente alla fine dell'esecuzione. Caratteristiche principali di h2:

Database in-memory → Perfetto per test veloci senza dipendenza da un database esterno.

Compatibile con JDBC e JPA → Integrabile facilmente in applicazioni **Spring Boot**.

Modalità persistente → Se necessario, può anche salvare i dati su file.

Compatibilità SQL → Supporta gran parte delle funzionalità SQL standard.

Avvio rapido e nessuna installazione richiesta → Viene eseguito direttamente all'interno dell'applicazione.

3.8.1 Test Coverage

Element	Class	Method	Line, %	Branch, %
org.elis.manutenzione	97% (10... 98% (55... 95% (15... 13% (32...			
config	50% (1/2) 100% (3... 100% (1... 100% (0...			
handler	83% (5/6) 80% (17... 82% (2... 0% (0/3...			
dto	97% (48... 99% (24... 99% (2... 0% (0/1...			
repository	100% (1... 100% (2... 92% (3... 64% (2...			
security	100% (5... 100% (1... 100% (7... 75% (3...			
controller	100% (5... 100% (4... 100% (1... 100% (1...			
statePattern	100% (5... 100% (5... 100% (5... 100% (0...			
mapper	100% (5... 100% (2... 98% (2... 72% (5...			
service	100% (8... 100% (5... 89% (4... 76% (2...			
builderPattern	100% (9... 100% (5... 100% (8... 0% (1/...			
model	100% (1... 98% (82... 98% (12... 6% (23...			

✓ Tests passed: 185 of 185 tests – 1 min 16 sec

Come si può vedere dalle immagini, sono stati scritti 185 test che coprono la maggior parte delle righe di codice di tutto il progetto (1510/1587).

3.9 Docker

Piattaforma di **virtualizzazione** leggera che consente di **creare, distribuire ed eseguire** applicazioni in containers. I **container** Docker contengono tutto il necessario per eseguire un'applicazione, inclusi il codice, le librerie e le dipendenze, isolati dal sistema operativo sottostante. Questa tecnologia è utilizzata per semplificare il processo di sviluppo, distribuzione e gestione delle applicazioni. Di seguito alcune nomenclature utili per comprendere l'architettura:

Immagine – Pacchetto leggero che contiene tutto il necessario per eseguire un'applicazione, inclusi il codice, le librerie, le dipendenze e le configurazioni. Le immagini sono utilizzate per creare containers.

Container – Istanza eseguibile di un'immagine Docker. Sono leggeri, portabili e possono essere eseguiti in qualsiasi ambiente Docker compatibile.

Dockerfile – File di configurazione che contiene le istruzioni passo-passo per la creazione di un'immagine Docker. Specifica le dipendenze, le configurazioni e le azioni da eseguire durante la costruzione dell'immagine.

3.10 Avvio progetto

Prerequisiti:

- Git
- Docker

Avvio

1. Clonare il progetto in locale: usare il comando "git clone <https://github.com/Narderio/Manutenzione-College>"
2. Avviare il processo docker
3. Posizionarsi nella cartella back-end e digitare il comando "docker compose up -build"
4. Raggiungere il sito: "localhost:4200"
5. Fare il login con le credenziali dell'admin (viene salvato automaticamente nel db una volta avviato il progetto)
6. Username: nar@03.com Password: prova
7. Ora è possibile usare l'admin per aggiungere altri utenti e luoghi all'interno dell'applicazione

4 Front-end

4.1 Tecnologie utilizzate

4.1.1 Angular

Angular è un framework open-source sviluppato principalmente da Google. Implementato nella versione utilizzata (v.15.2.0) con TypeScript, adotta un approccio modulare, optionalmente orientato agli oggetti e si basa sull'utilizzo di componenti per la creazione di applicazioni web dinamiche. Prima di addentrarci nelle competenze del progetto è doveroso spiegare gli elementi principali di un progetto angular.

4.1.1.1 Components

Un componente Angular è un costrutto fondamentale che incapsula logica(.ts), template (.html) e stili(.css) per una specifica parte dell'interfaccia utente. Viene definito tramite il decoratore **@Component()**, il quale permette di:

4.1.1.1.1 Specificare i metadata(@Component({...})):

- **selector**: indica come il componente sarà usato nel template
(es. `selector: 'app-segnala-manutenzione'`)
- **templateUrl** : definisce la parte HTML in un file separato.
(es. `templateUrl: './segnala-manutenzione.component.html'`)
- **styleUrls**: permette di associare stili CSS specifici.
(es. `styleUrls: ['./segnala-manutenzione.component.css']`)

4.1.1.1.2 Classe **TypeScript**:

- Contiene proprietà e metodi utili al rendering e alla **logica** del componente.
- Integra i **lifecycle hooks** (es. `ngOnInit()`, `ngOnDestroy()`) per gestire comportamenti nelle varie fasi di vita (creazione, aggiornamento, distruzione).

4.1.1.1.3 Dependency **Injection**:

- Può **ricevere servizi** (o altre dipendenze) tramite il "constructor", Angular poi si occuperà di fornirli automaticamente.

4.1.1.1.4 Input/Output:

- Proprietà con **decoratori** @Input() / @Output() consentono di comunicare dati ed eventi fra componenti tramite relazione **padre-figlio**.

Nel flusso di esecuzione dell'app, Angular istanza e collega i componenti per costruire **l'albero dell'interfaccia**: ogni componente diventa un nodo specializzato che aggiorna la **vista in modo reattivo** sulla base delle sue proprietà e dei servizi iniettati.

4.1.1.2 Services

Un servizio Angular è una classe che contiene logiche riutilizzabili e funzionalità condivise tra tutti i componenti (ad es. chiamate HTTP, gestione dei dati, logica di business) che può essere iniettata in componenti o in altri servizi. Le sue caratteristiche principali sono:

4.1.1.2.1 Decoratore @Injectable():

- Marca la classe come idonea all'iniezione delle dipendenze (vedi X.X.X Dependency Injection).
- Include la proprietà **providedIn** (es. 'root'), che specifica l'**ambito** di fornitura delle dipendenze (tipicamente il root injector, per avere un'unica istanza condivisa).

4.1.1.2.2 Organizzazione del codice:

- Ideale per chiamate a **REST API** (HttpClient), gestione dello stato, ecc.
- Rende la logica più manutenibile e testabile, riducendo duplicazioni di codice nei componenti.

4.1.1.2.3 Gerarchia di injector:

- Se un servizio è "fornito" (**provided**) nel **root injector**, si ottiene di default un **singleton** in tutta l'applicazione (vedi X.X.X Singleton).

4.1.1.3 Routes

In Angular, una rottura è una definizione che **mappa un URL** specifico dell'applicazione a un componente che deve essere caricato e mostrato quando quell'URL viene richiesto. Alcune caratteristiche tecniche chiave sono:

4.1.1.3.1 Configurazione tramite **RouterModule**:

- Le rotte si dichiarano in un array di oggetti passato a **RouterModule.forRoot()**.
- Ogni definizione contiene almeno una proprietà "**path**" (ossia la porzione di URL), un "**component**" (il componente che si vuole caricare), un parametro "**canActivate**" (per il controllo degli accessi e dei ruoli tramite "Guard") e **data** (ossia dati da passare alla rottura).

4.1.1.3.2 **Route Guards** (AuthGuard, CanActivate):

- Sono meccanismi che consentono di **controllare l'accesso** a una rottura, eseguendo logiche prima di permettere il caricamento del componente.

4.1.1.3.3 **RouterLink** e **RouterOutlet**:

- **<router-outlet>** è la direttiva nel template che ospita il componente corrispondente alla rottura attiva.
- **routerLink** e **routerLinkActive** sono direttive che permettono di navigare tra rotte e gestire lo stato di selezione del componente mostrato in "**router-outlet**".

4.1.2 Angular Material

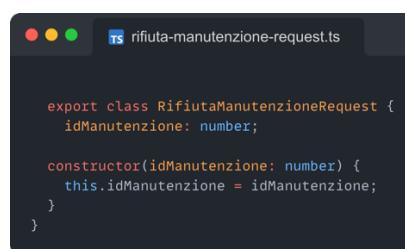
Angular Material è una libreria UI ufficiale per Angular basata sulle linee guida **Material Design di Google**. Fornisce componenti pre-costruiti, già ottimizzati per l'accessibilità e la responsività, e facilmente integrabili in un progetto Angular. Ogni componente come pulsanti tabelle o tab risiede in un **modulo separato** (es. MatButtonModule, MatTableModule), basta importare i moduli necessari all'interno del file “**app.module.ts**” per ridurre il peso del bundle.

4.2 Design patterns

4.2.1 DTO

Il pattern DTO (Data Transfer Object) si riferisce all'uso di oggetti specializzati per **trasferire dati tra i componenti** di un'applicazione. Questi oggetti contengono solo dati e non includono logica di business o metodi. L'obiettivo principale del pattern DTO è semplificare la comunicazione e lo scambio di dati all'interno dell'applicazione, come in questo progetto tra il front-end e il back-end. È ampiamente utilizzato all'interno del codice come per esempio in “RifiutaManutenzioneRequest”(1) che è un DTO creato ed utilizzato per comunicare con il backend nella richiesta di “rifiutaManutenzione”(2).

Foto di esempio 1:



```
export class RifiutaManutenzioneRequest {
    idManutenzione: number;

    constructor(idManutenzione: number) {
        this.idManutenzione = idManutenzione;
    }
}
```

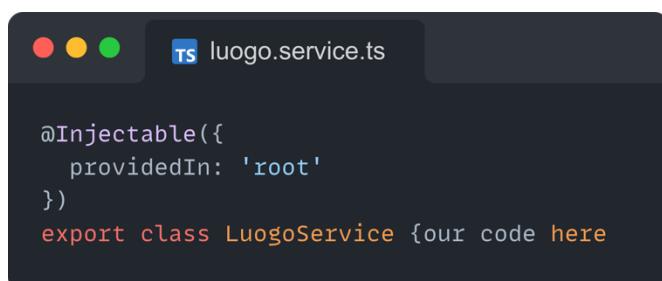
Foto di esempio 2:



```
/**
 * Invia la richiesta di rifiuta manutenzione al BE
 */
rifiutaManutenzione(rifiutaManutenzioneRequest: RifiutaManutenzioneRequest): Observable<RifiutaManutenzioneResponse> {
    return this.http.post<RifiutaManutenzioneResponse>(environment.baseUrl + environment.Manutenzione.rifiutaManutenzione, rifiutaManutenzioneRequest);
}
```

4.2.2 Dependency Injection & Singleton (DI)

Il Dependency Injection pattern promuove il concetto di **Inversion of Control** (IoC). E' un pattern direttamente implementato da Angular che **separa la creazione di un oggetto dalle sue dipendenze** favorendo future manutenzioni e modularità del codice. Allo stesso tempo tramite il “**@Injectable**” Angular **implementa** anche il **Singleton** pattern fornendo un punto di accesso unico all'istanza di una classe. Sostanzialmente l'istanza creata la prima volta che il servizio viene richiamato rimane “unica” per tutta la durata dell'applicazione.



```
❶ ❷ ❸ luogo.service.ts
❹
❺ @Injectable({
❻   providedIn: 'root'
❼ })
❽ export class LuogoService {our code here}
```

4.2.3 Observer

L'Observer pattern è un modello di **programmazione asincrona**, noto per gestire sequenze di eventi o dati in modo reattivo. Stabilisce una relazione **uno-a-molti** tra oggetti, dove:

- **Subject** (o Observable): è l'entità “osservata” che è in grado di emettere notifiche su modifiche di stato o nuovi dati.
- **Observer** (o Subscriber): che sono gli oggetti che si registrano per ricevere notifiche (di solito tramite un metodo subscribe() come si può vedere dall'esempio di codice(1)).

Nel caso particolare di questo progetto è stato utilizzato per tutte le **richieste Http** e nel caso della **chat** per sottoscriversi alle notifiche di rabbitMq (2).

Foto di esempio 1:



```
❶ ❷ ❸ auth.service.ts
❹
❺ login(loginRequest: LoginRequest): boolean {
❻   this.http.post<LoginResponse>(environment.baseUrl + environment.Utente.login, loginRequest).subscribe({
❼     next: result => {
❽ }
```

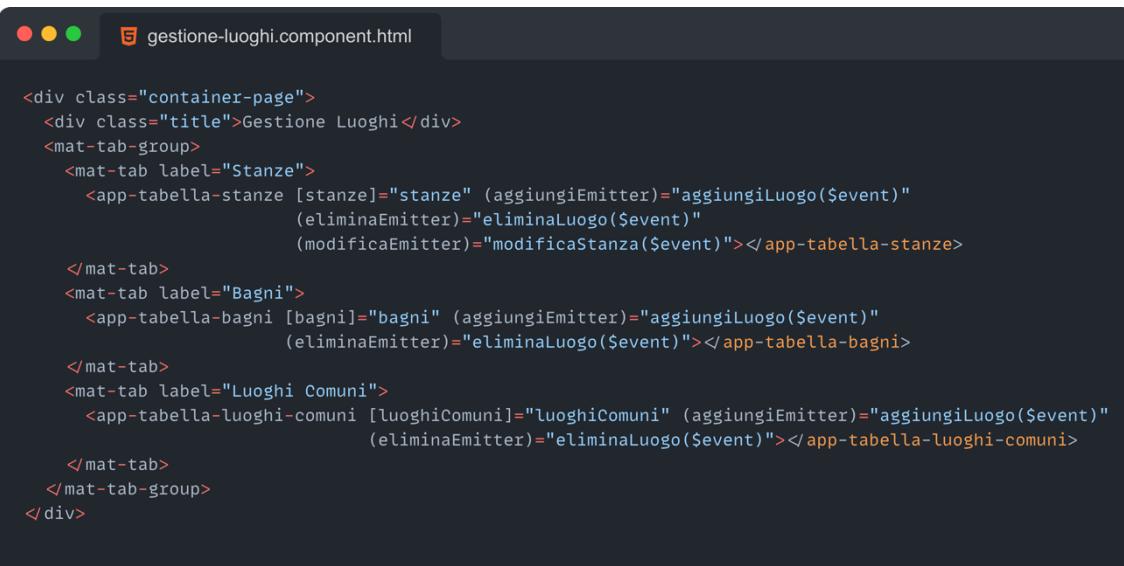
Foto di esempio 2:



```
// se non è connesso si connette al topic
const topic = `/exchange/TopicTestExchange/${email}`; // Topic basato sull'email
this.client.subscribe(topic, (message) => callback(message));
```

4.2.4 Composite

Il Composite è un design pattern strutturale che permette di comporre oggetti in **strutture ad albero** (gerarchie), in modo che il codice possa trattare in modo uniforme sia gli oggetti singoli sia i gruppi di oggetti. In Angular è implementato tramite l'utilizzo dei componenti. In particolare nel progetto possiamo vedere l'utilizzo di questo pattern nell'immagine dove richiamando il componente "gestione-luoghi" automaticamente vengono richiamati al suo interno altri 3 componenti: "tabella-stanze", "tabella-bagni", "tabella-luoghi-comuni".

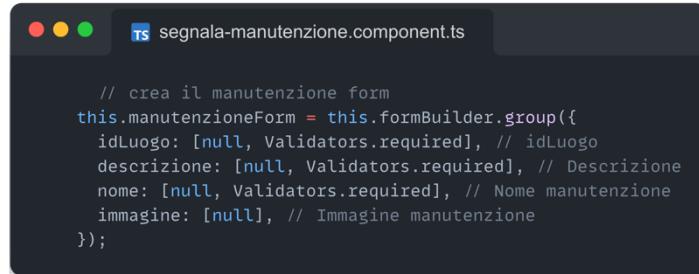


```
<div class="container-page">
  <div class="title">Gestione Luoghi</div>
  <mat-tab-group>
    <mat-tab label="Stanze">
      <app-tabella-stanze [stanze]="stanze" (aggiungiEmitter)="aggiungiLuogo($event)"
        (eliminaEmitter)="eliminaLuogo($event)"
        (modificaEmitter)="modificaStanza($event)"></app-tabella-stanze>
    </mat-tab>
    <mat-tab label="Bagni">
      <app-tabella-bagni [bagni]="bagni" (aggiungiEmitter)="aggiungiLuogo($event)"
        (eliminaEmitter)="eliminaLuogo($event)"></app-tabella-bagni>
    </mat-tab>
    <mat-tab label="Luoghi Comuni">
      <app-tabella-luoghi-comuni [luoghiComuni]="luoghiComuni" (aggiungiEmitter)="aggiungiLuogo($event)"
        (eliminaEmitter)="eliminaLuogo($event)"></app-tabella-luoghi-comuni>
    </mat-tab>
  </mat-tab-group>
</div>
```

4.2.5 Validator

Il Validator è un pattern architetturale con l'obiettivo di **separare la logica di validazione da quella di business**, incapsulandola in funzioni o oggetti che verifichino se un certo dato rispetta regole specifiche (es. un campo non vuoto, un

formato email corretto, un range numerico, ecc.). È ampiamente implementato nel codice, un esempio può essere trovato in “segnala-manutenzione” dove i campi: “idLuogo”, “descrizione” e “nome” devono essere obbligatoriamente inseriti.



```
// crea il manutenzione form
this.manutenzioneForm = this.formBuilder.group({
  idLuogo: [null, Validators.required], // idLuogo
  descrizione: [null, Validators.required], // Descrizione
  nome: [null, Validators.required], // Nome manutenzione
  immagine: [null], // Immagine manutenzione
});
```

4.2.6 Decorator

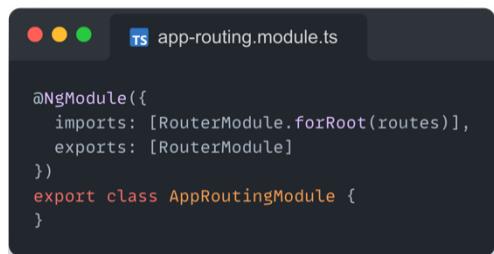
Il Decorator è un pattern strutturale che consente di **aggiungere dinamicamente funzionalità** a un oggetto avvolgendolo in un oggetto “decoratore”, che implementa la stessa interfaccia. Invece di ereditare o modificare direttamente la classe, si compone con un **wrapper che intercetta le chiamate** e aggiunge comportamenti prima o dopo di delegarle all’oggetto originale. In angular è ampiamente utilizzato tramite i **decorator**: @Component (1) @NgModule (2) @Input (3) @Output (4) come nelle immagini sottostanti:

Foto di esempio 1:



```
@Component({
  selector: 'app-segnala-manutenzione',
  templateUrl: './segnala-manutenzione.component.html',
  styleUrls: ['./segnala-manutenzione.component.css']
})
export class SegnalaManutenzioneComponent implements OnInit {
```

Foto di esempio 2:



```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {
```

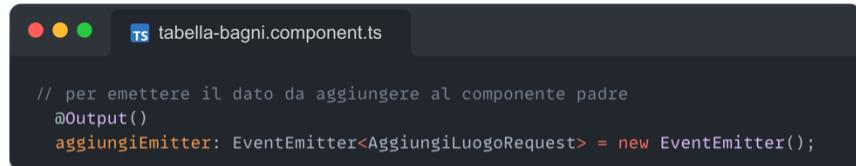
Foto di esempio 3:



```
tabella-luoghi-comuni.component.ts

@Input()
luoghiComuni: GetLuoghiComuniResponse[]; // per visualizzare i luoghi comuni presenti
```

Foto di esempio 4:



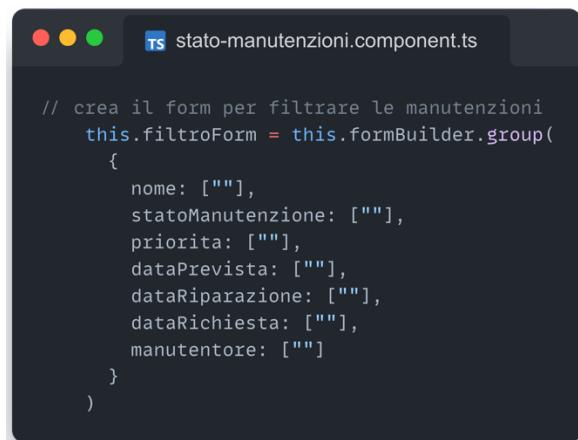
```
tabella-bagni.component.ts

// per emettere il dato da aggiungere al componente padre
@Output()
aggiungiEmitter: EventEmitter<AggiungiLuogoRequest> = new EventEmitter();
```

4.2.7 Builder

Il Builder è un **pattern creazionale** che separa la costruzione di un oggetto complesso dalla sua rappresentazione, consentendo di realizzare differenti rappresentazioni dello stesso processo di costruzione. In Angular e in particolare nel progetto è ampiamente utilizzato per creare tutti i form presenti tramite una tecnica chiamata ReactiveForm.

Foto di esempio:



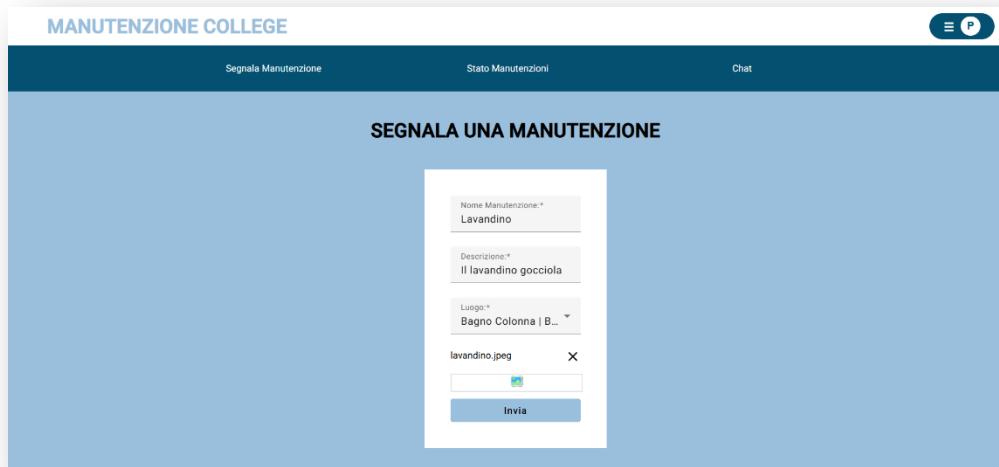
```
stato-manutenzioni.component.ts

// crea il form per filtrare le manutenzioni
this.filtroForm = this.formBuilder.group(
{
  nome: [""],
  statoManutenzione: [""],
  priorita: [""],
  dataPrevista: [""],
  dataRiparazione: [""],
  dataRichiesta: [""],
  manutentore: []
})
```

4.3 Navigazione lato utente

4.3.1 Segnala manutenzione

Per segnalare una manutenzione bisogna andare nella corrispettiva pagina e segnalare un guasto inserendo i campi richiesti, e se necessaria, aggiungere una foto del guasto.



MANUTENZIONE COLLEGE

Segnala Manutenzione Stato Manutenzioni Chat

SEGNALA UNA MANUTENZIONE

Nome Manutenzione: *
Lavandino

Descrizione: *
Il lavandino gocciola

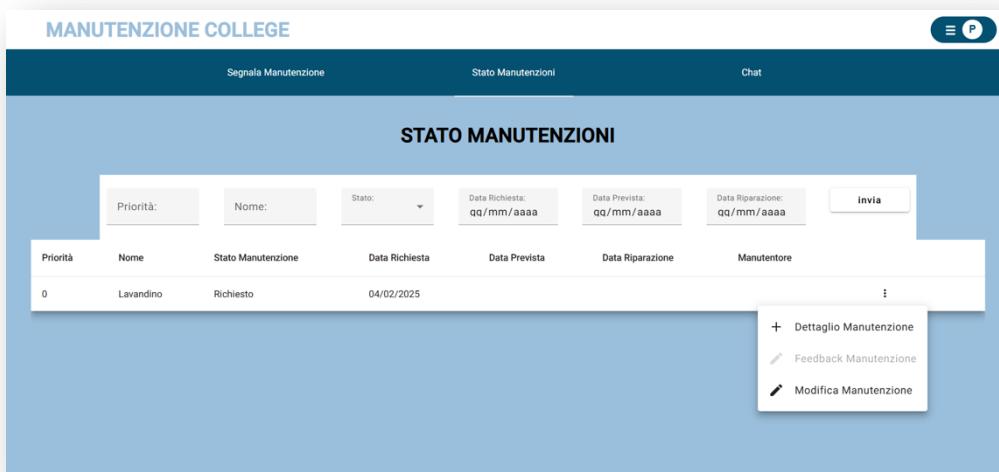
Luogo: *
Bagno Colonna | B...

lavandino.jpeg

Invia

4.3.2 Segnala manutenzione

In stato manutenzioni è possibile visualizzare lo stato delle proprie manutenzioni, ovvero se sono state riparate, se devono essere ancora accettate, se c'è una data di previsione...



MANUTENZIONE COLLEGE

Segnala Manutenzione Stato Manutenzioni Chat

STATO MANUTENZIONI

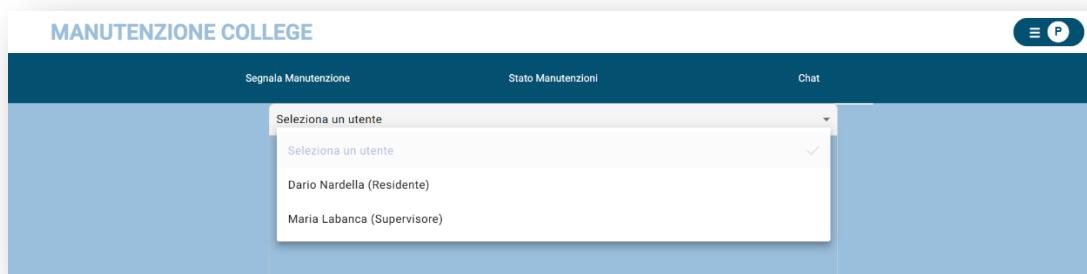
Priorità	Nome	Stato Manutenzione	Data Richiesta	Data Prevista	Data Riparazione	Manutentore
0	Lavandino	Richiesto	04/02/2025			

+ Dettaglio Manutenzione
✍ Feedback Manutenzione
✍ Modifica Manutenzione

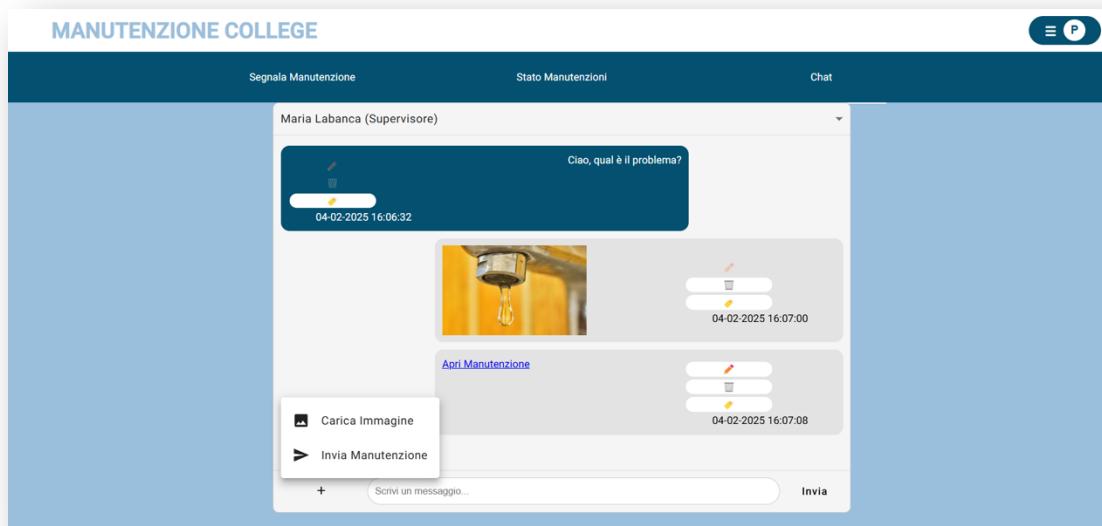
Inoltre è possibile entrare nel dettaglio della manutenzione, per visualizzare meglio i campi e la descrizione, ed è possibile modificare alcuni campi della manutenzione, in caso di errori.

4.3.3 Chat

Per prima cosa è necessario scegliere la persona con cui messaggiare (è possibile farlo anche tramite tastiera).



Successivamente si entra nella chat con la persona selezionata in cui è possibile: inviare messaggi, modificare i propri messaggi, eliminare i propri messaggi, rispondere taggando i messaggi di ognuno, inviare immagini e taggare una manutenzione propria.



4.3.4 Gestione manutenzioni

Il supervisore è colui che filtrerà le manutenzioni, assegnando un livello di priorità che va da 0 a 5, inoltrandole al manutentore scelto.

The screenshot shows the 'GESTIONE MANUTENZIONI' page. At the top, there is a search bar with fields for Priorità, Nome, Stato, Data Richiesta, Data Prevista, Data Riparazione, and Manutentore, followed by a 'Invia' button. Below the search bar is a table listing two maintenance requests:

Priorità	Nome	Stato Manutenzione	Data Richiesta	Data Prevista	Data Riparazione	Manutentore
0	Lavandino	Richiesto	04/02/2025			
0	Maniglia	Richiesto	04/02/2025			

A context menu is open over the second row (Maniglia), containing three options: 'Dettaglio Manutenzione', 'Filtra', and 'Rifiuta'. The 'Filtra' option is highlighted with a blue background.

Inoltre ha la possibilità di modificare il nome della manutenzione, precedentemente assegnato dal residente.

The screenshot shows the 'GESTIONE MANUTENZIONI' page with a modal dialog titled 'Filtrala manutenzione: "Lavandino"'. The dialog contains fields for Manutentore (manutentore1@gmail.com), Priorità (5), and Nome (Lavandino). There are 'Indietro' and 'Filtra' buttons at the bottom of the dialog. The background table remains visible.

4.3.5 Gestione utenze e luoghi

Il compito dell'admin sarà principalmente gestire gli utenti e i luoghi, quindi ogni volta sarà necessario, popolare il db e effettuare modifiche su persone o luoghi, come aggiornare ruoli, cambiare la stanza di un residente o cambiare la capienza di un luogo.

Nome	Nucleo	Piano	Capienza	Residenti
Colonna 1	Colonna	4	4	0
Colonna 2	Colonna	4	4	Modifica Capienza Elimina Aggiungi

Residenti	Manutentori	Supervisori	Admin
residente1@gmail.com	Nicol	Goranova	2003-01-04
residente2@gmail.com	Dario	Nardella	2003-05-17

5 Conclusioni e Sviluppi futuri

Questo progetto ci ha permesso di crescere sotto diversi aspetti, non solo dal punto di vista tecnico nello sviluppo dell'applicativo e nell'approfondimento delle tecnologie utilizzate, ma anche nell'analisi delle esigenze dei vari ruoli coinvolti e nella gestione delle diverse fasi del processo.

Nel breve termine, prevediamo di integrare questo sistema di gestione delle manutenzioni all'interno dei nostri college, sia maschile che femminile. Successivamente, puntiamo ad estenderlo all'interno dell'ecosistema Elis, permettendo anche ai dipendenti interni di segnalare guasti e necessità di manutenzione, contribuendo così a migliorare l'ambiente lavorativo.

Per rendere l'interazione con il sistema ancora più fluida e accessibile, stiamo valutando l'integrazione di un chatbot che possa assistere gli utenti nella navigazione e nella gestione delle segnalazioni, semplificando ulteriormente l'utilizzo della piattaforma.

Riteniamo che questa soluzione possa davvero ottimizzare la gestione delle manutenzioni e migliorare l'efficienza dei processi, favorendo una comunicazione più diretta tra utenti e manutentori. Il progetto continuerà a evolversi e ad adattarsi alle esigenze specifiche dei vari contesti in cui verrà implementato, con l'obiettivo di garantire un servizio sempre più efficace e innovativo.