

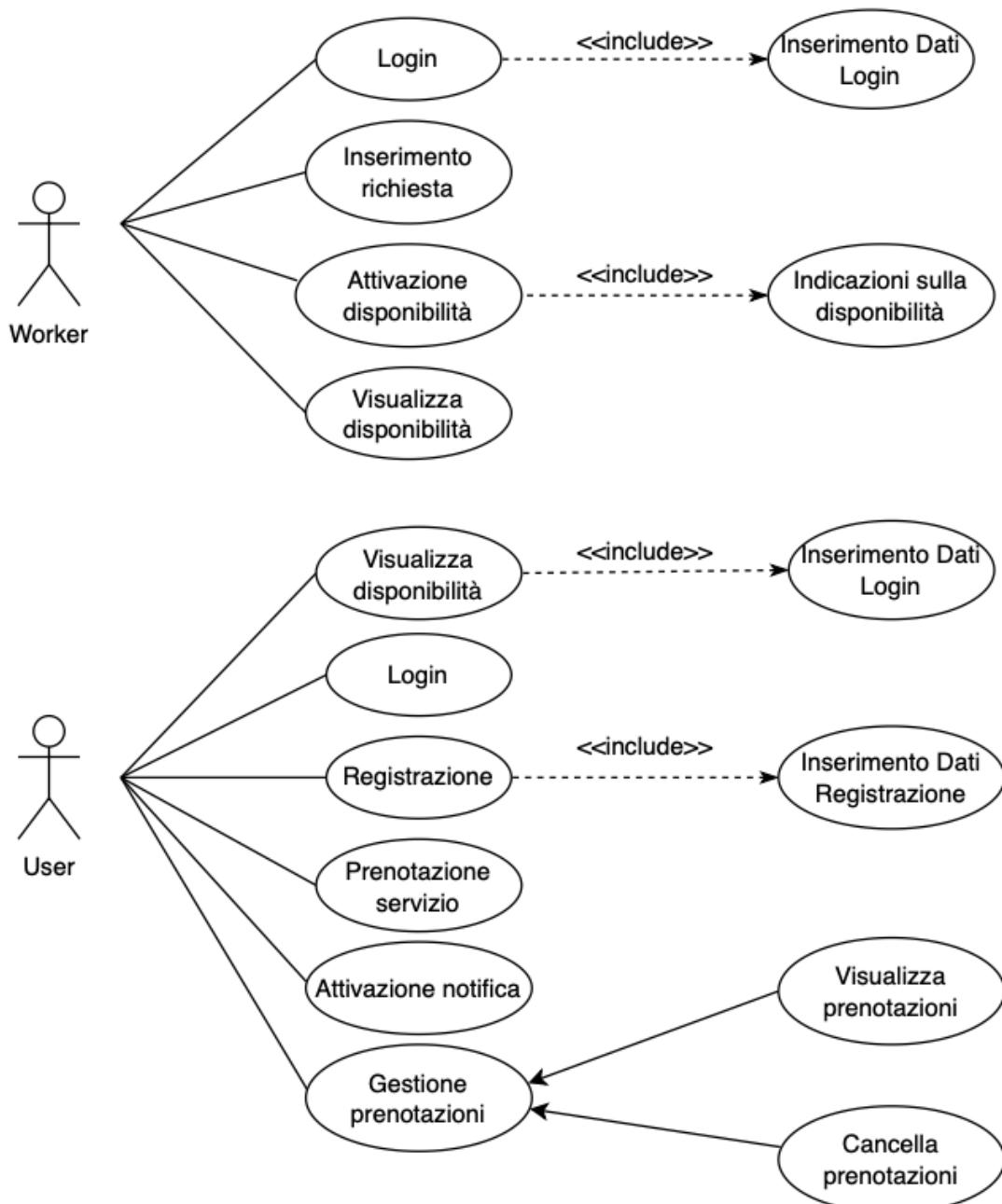
Requisiti ed interazioni utente-sistema

Specifiche casi d'uso

Note generali

Si vuole progettare un sistema informatico per gestire il servizio di prenotazione per il rilascio del passaporto per una Questura, che dispone di più sedi sul territorio. Il personale della Questura può inserire il tipo di richiesta e attivare di volta in volta le disponibilità per le varie richieste rispetto a giorni, orari, sedi e personale disponibile per soddisfarle. I cittadini possono registrarsi nel sistema e successivamente effettuare la prenotazione ad uno dei servizi disponibili o attivare una notifica se non ci sono disponibilità per il servizio scelto. È stata aggiunta anche la possibilità per il cittadino di gestire le prenotazioni confermate, con la possibilità di eliminarle o di visualizzare un pdf con le informazioni sulla prenotazione.

Use Case



Casi d'uso relativi al personale della Questura (Worker)

Login

Il personale può effettuare l'accesso con credenziali (nome utente e password) fornitegli in precedenza.

Attori: Worker

Precondizioni: Il worker possiede le credenziali d'accesso

Passi:

1. Il worker inserisce nome utente
2. Il worker inserisce la password

Postcondizioni: Il worker accede all'area riservata

Inserimento tipo di richiesta

Il personale può considerare vari tipi di operazioni sui passaporti (ritiro, rilascio per scadenza, furto ecc...) e inserire tali richieste nel sistema.

Attori: Worker

Precondizioni: Il worker deve essersi autenticato.

Passi:

1. Il worker seleziona "Nuova richiesta"
2. Il worker seleziona il tipo di richiesta

Postcondizioni: Il worker deve creare la disponibilità.

Attivazione disponibilità

Il personale può creare le disponibilità di volta in volta per le varie richieste.

Attori: Worker

Precondizioni: Il worker deve aver selezionato il tipo di richiesta.

Passi:

1. Il worker seleziona la sede
2. Il worker seleziona il giorno
3. Il worker seleziona la fascia oraria
4. Il worker conferma

Postcondizioni: Viene creata la nuova disponibilità il tipo di richiesta.

Casi d'uso relativi al cittadino (User)

Registrazione

A differenza del Worker, il cittadino prima di effettuare operazioni sul sito deve registrarsi.

Per registrarsi, l'utente deve inserire Nome, Cognome, Codice Fiscale, Email, Password e Luogo e Giorno di nascita.

Attori: User

Precondizioni: Il cittadino deve esistere nell'anagrafe.

Passi:

1. Lo user inserisce codice fiscale
2. Lo user inserisce il suo nome
3. Lo user inserisce il suo cognome
4. Lo user seleziona la data di nascita
5. Lo user inserisce il luogo di nascita
6. Lo user inserisce una mail di riferimento
7. Lo user inserisce una password

Postcondizioni: L'utente deve autenticarsi nel sito.

Login

Una volta creato il proprio profilo, è necessario effettuare il login prima di effettuare operazioni sul sito. Il login avviene in modo simile a quello del Worker e successivamente porta l'utente alla schermata di prenotazione.

Prenotazione

Dopo il login, l'utente è subito portato alla schermata di prenotazione. Eseguendo le operazioni necessarie, l'utente può cercare di prenotare un servizio, nel caso in cui esistano slot disponibili nella sede e nel giorno desiderato. Nel caso non esistano, l'utente attiva una notifica che lo avviserà quando si creerà una disponibilità per il servizio richiesto e tornerà successivamente ad effettuare l'operazione.

Attori: User

Precondizioni: L'utente deve essersi autenticato.

Passi:

1. Lo user seleziona il tipo di servizio desiderato
2. Lo user seleziona la sede in cui desidera recarsi per il servizio
3. Lo user visualizza un calendario con gli orari in cui è possibile effettuare il servizio
4. (a) Esistono slot orari in cui è possibile prenotare il servizio, lo user seleziona l'orario desiderato.

(b) Non esistono slot orari in cui è possibile prenotare il servizio, lo user chiede di ricevere una notifica quando sarà disponibile uno slot per il servizio richiesto
5. Lo user conferma

Postcondizioni: La prenotazione è effettuata e viene visualizzato un messaggio con i documenti da portare.

Cancellare una prenotazione

L'utente può gestire le prenotazioni che sono state confermate ed eventualmente cancellarle. Dopo il login esiste un menù che lo porta alle prenotazioni confermate dove può effettuare tale operazione.

Attori: User

Precondizioni: Lo user deve avere almeno una prenotazione confermata.

Passi:

1. Lo user accede al menù utente
2. Lo user accede alle prenotazioni
3. Lo user seleziona la prenotazione che desidera cancellare
4. Lo user conferma

Postcondizioni: La prenotazione è cancellata in modo irreversibile.

Visualizzare un pdf della prenotazione

Tra le funzionalità aggiuntive che abbiamo implementato, c'è la possibilità di visualizzare un pdf in qualsiasi momento di una delle prenotazioni confermate, il quale conterrà un riepilogo in una sorta di documento "ufficiale".

Attori: User

Precondizioni: Lo user deve possedere almeno una prenotazione confermata.

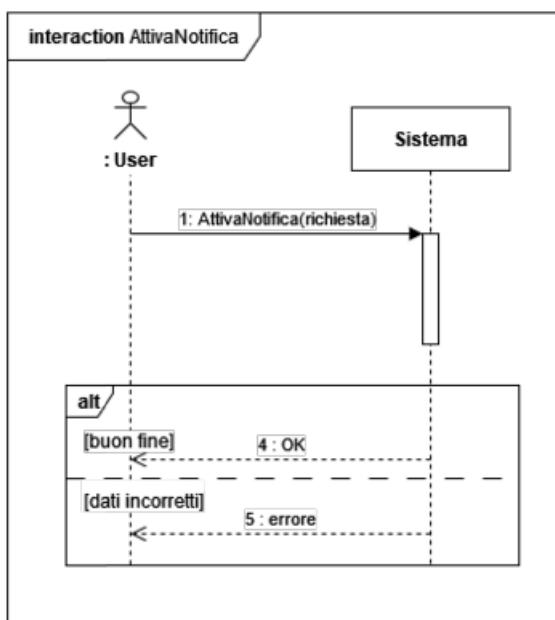
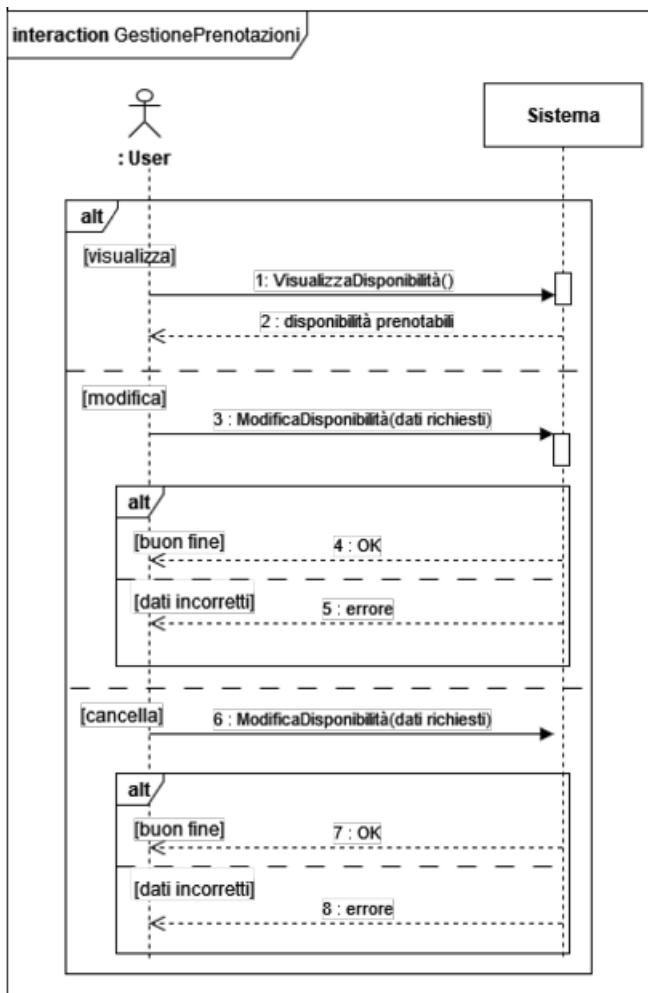
Passi:

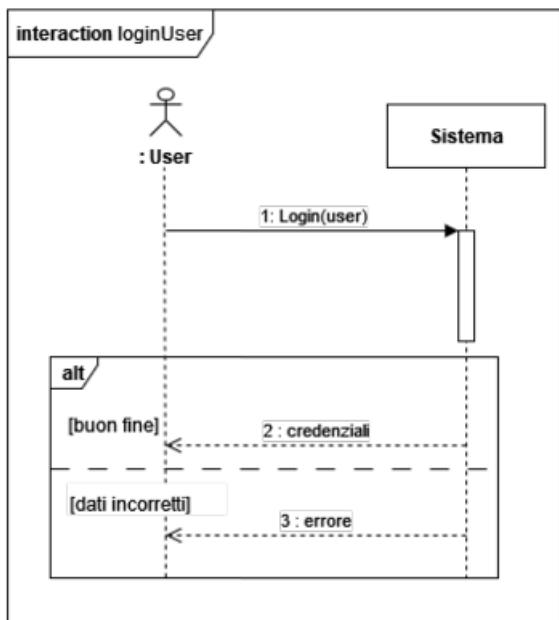
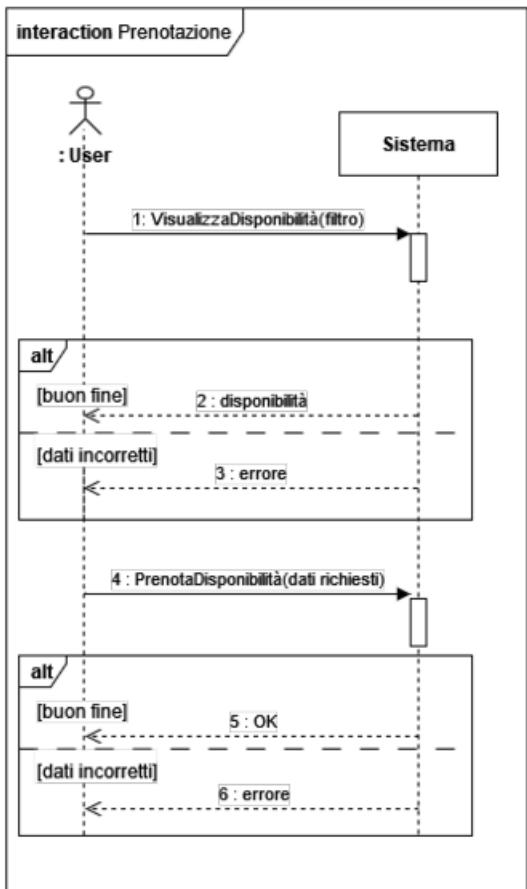
1. Lo user accede al menù utente
2. Lo user seleziona le prenotazioni confermate
3. Lo user seleziona il pdf della prenotazione desiderata

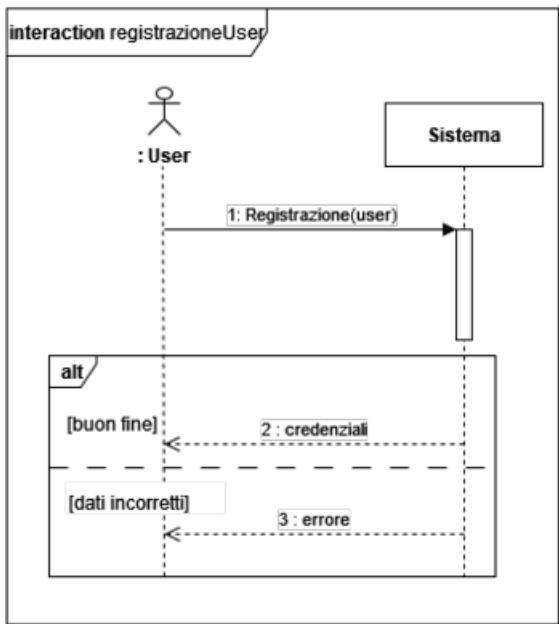
Postcondizioni: Il pdf viene aperto.

Sequence Diagram dei principali Use Case

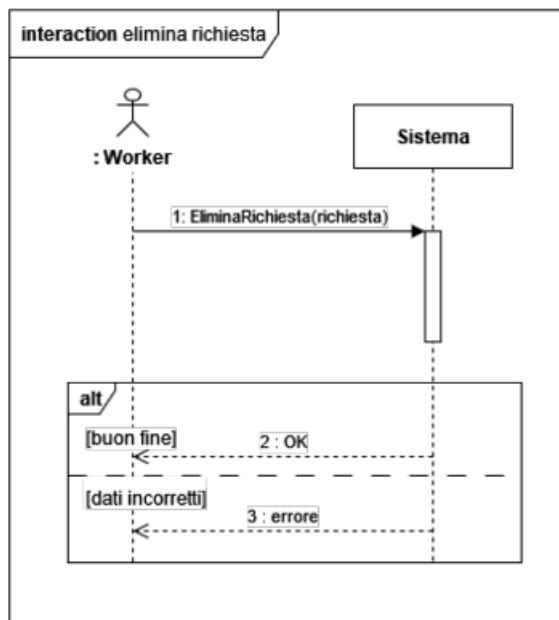
User

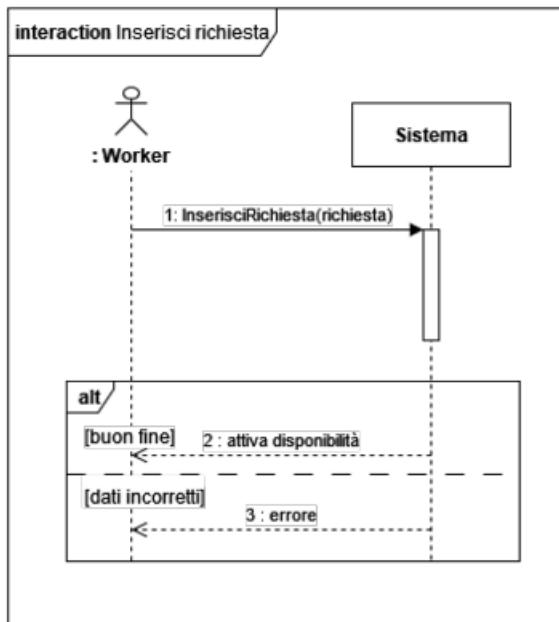
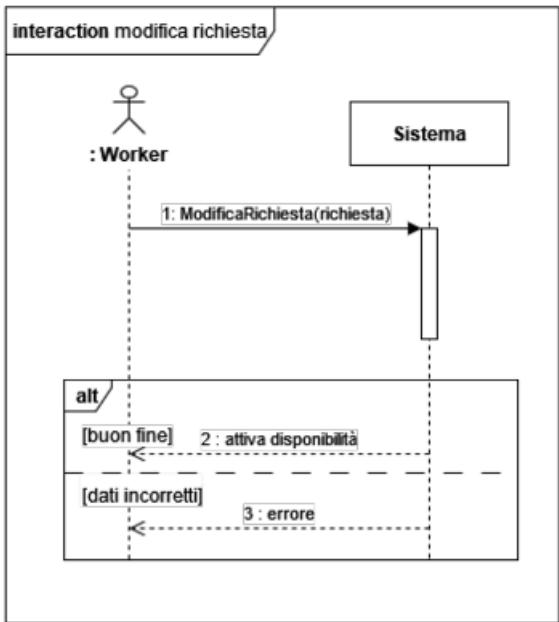


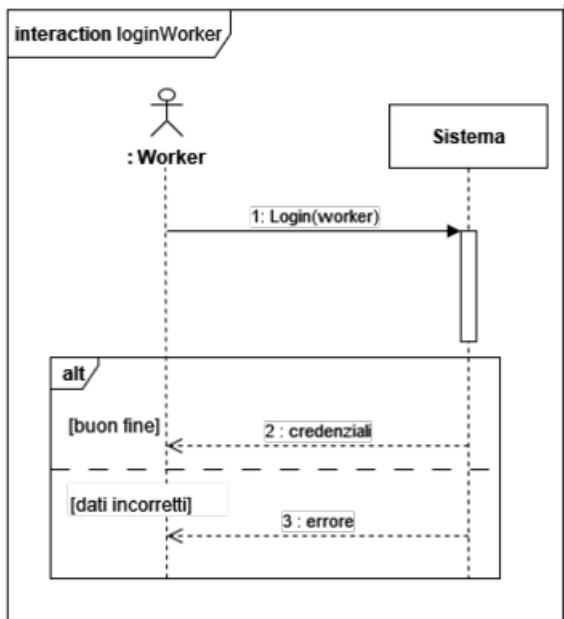
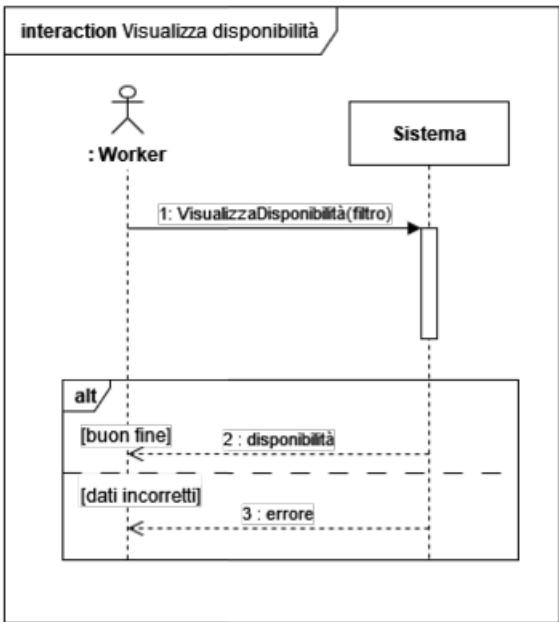




Worker

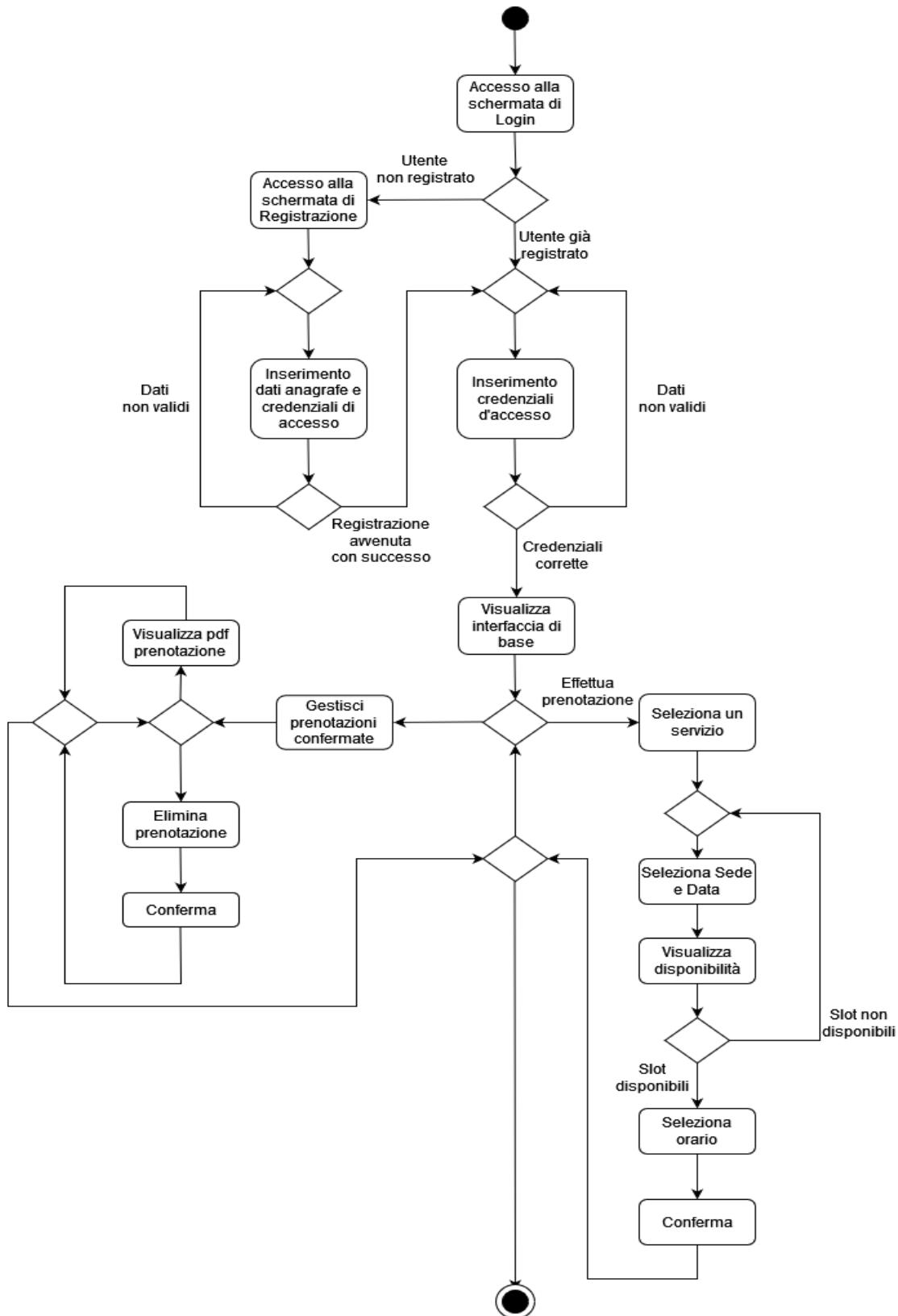




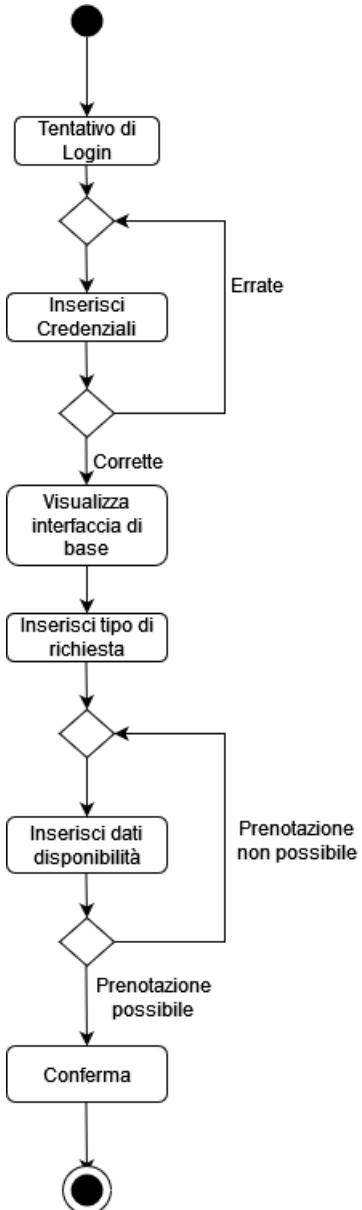


Activity Diagram

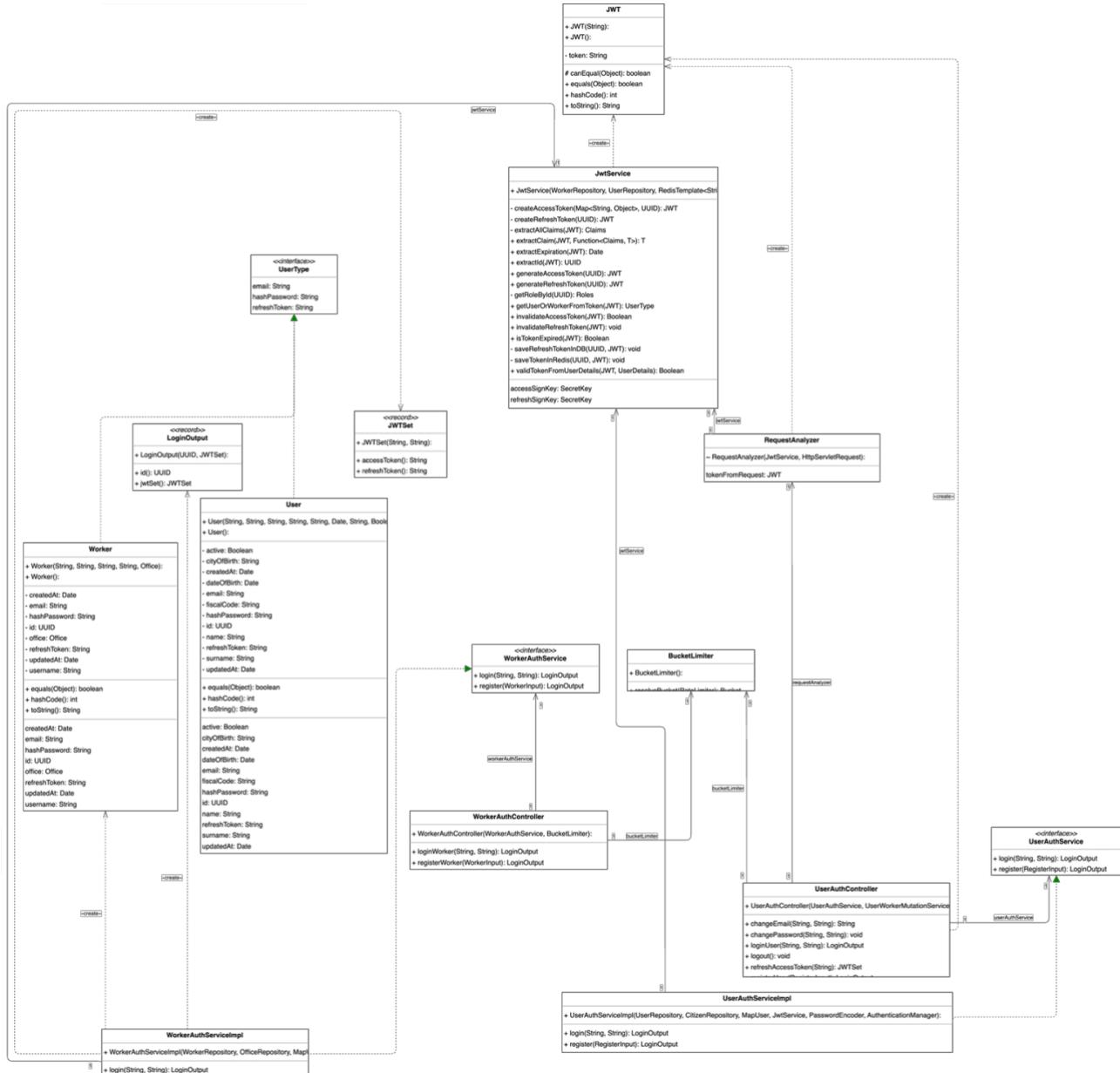
User



Worker

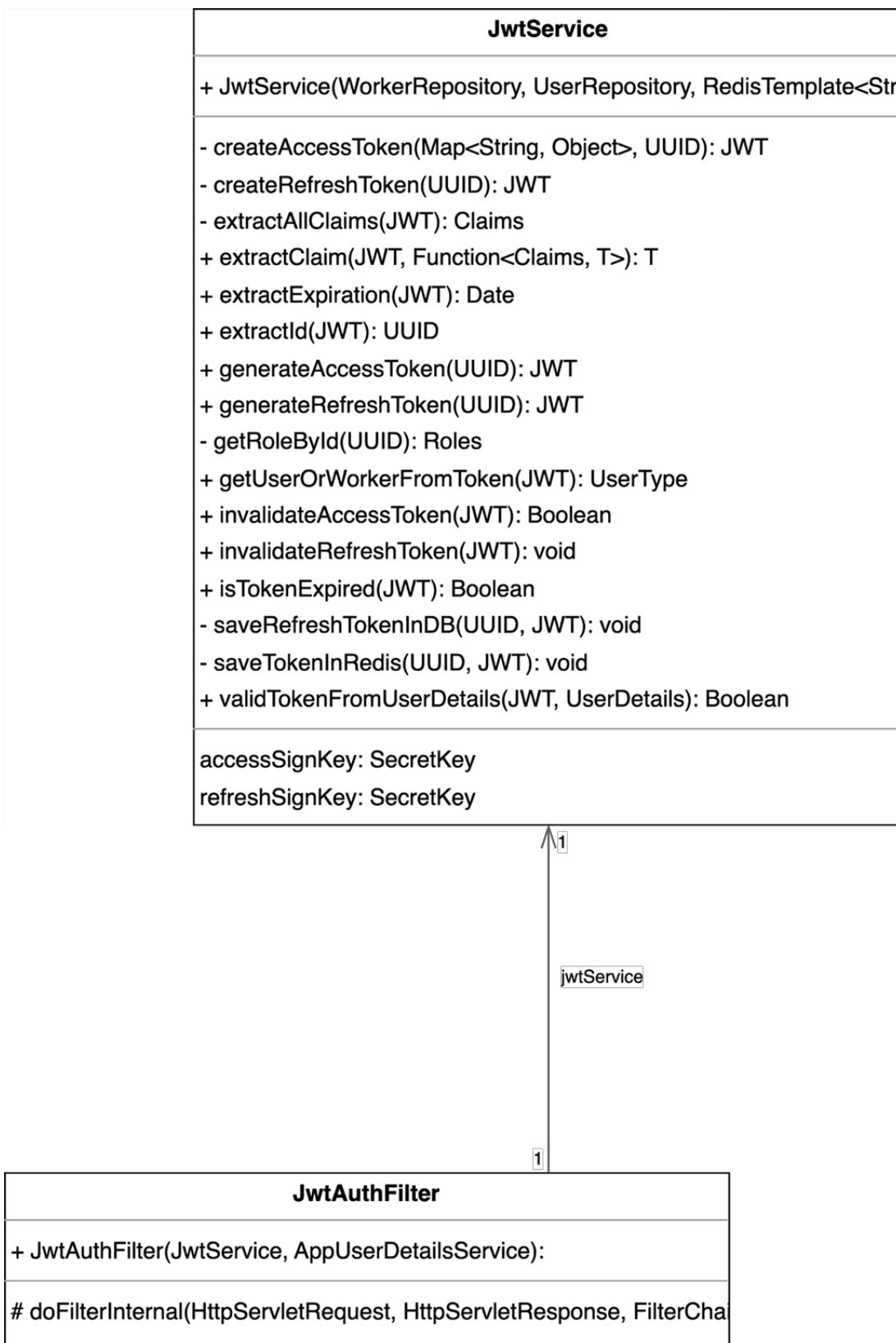


Class Diagram

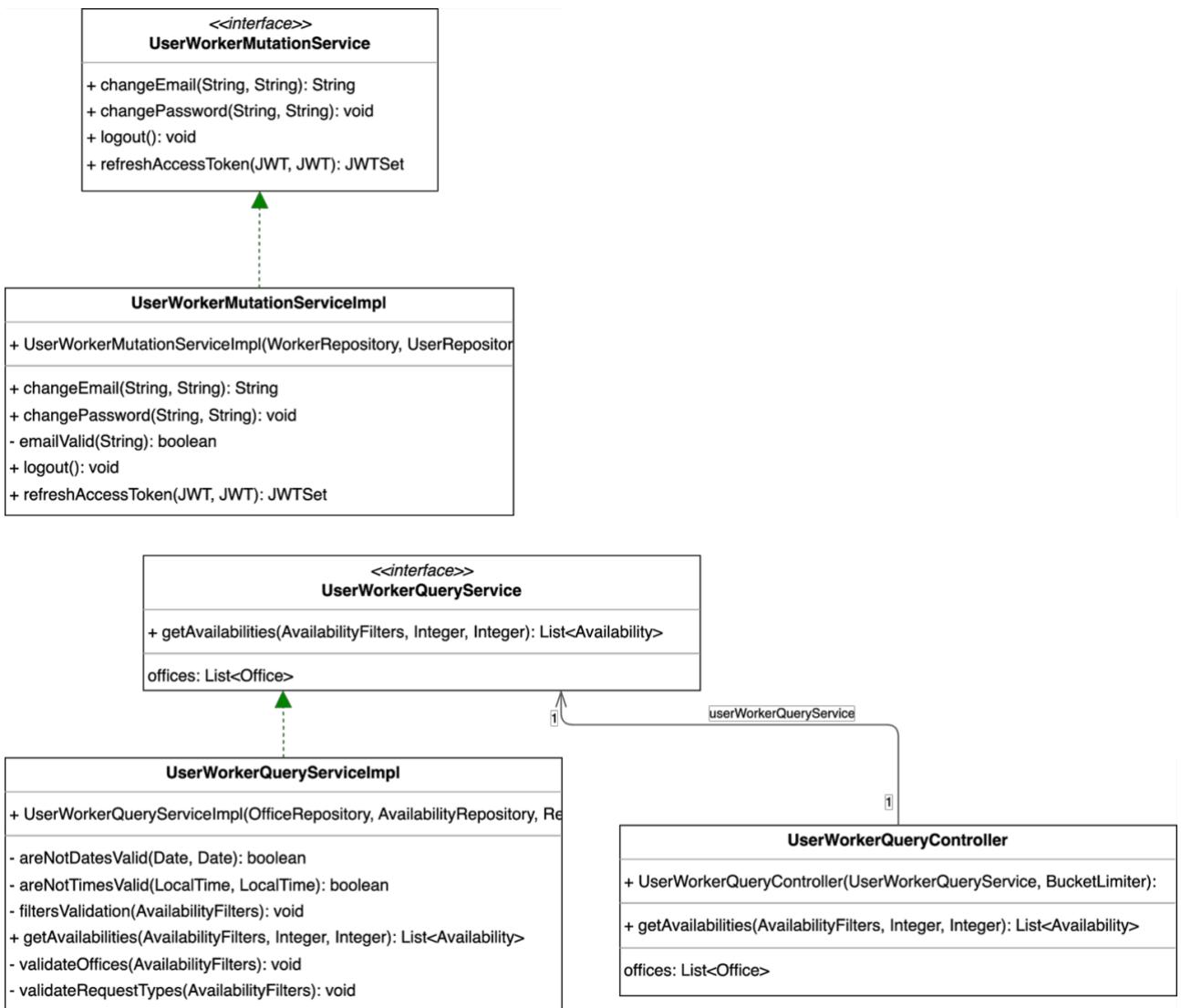


AuthTotale

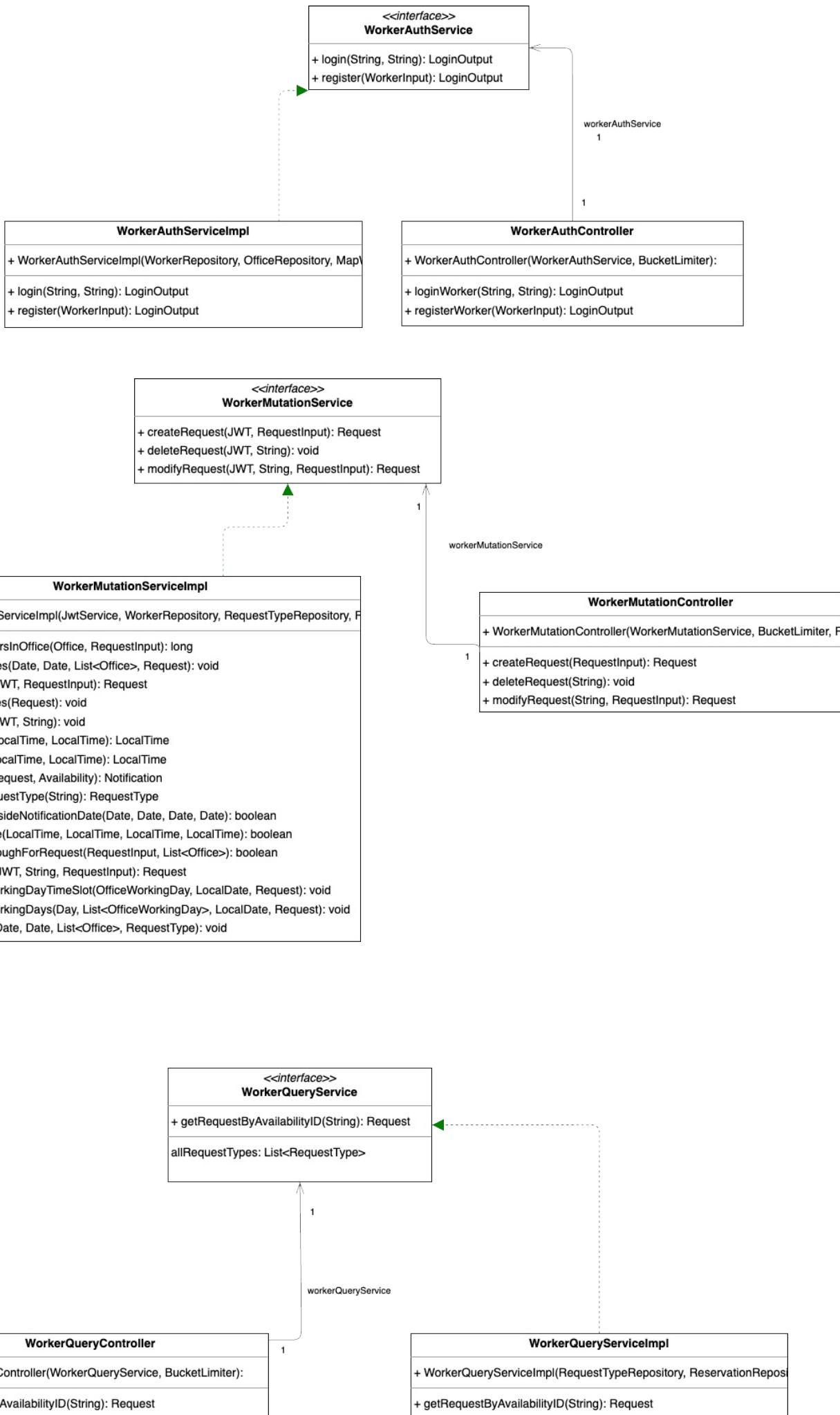
Class Diagram JWT



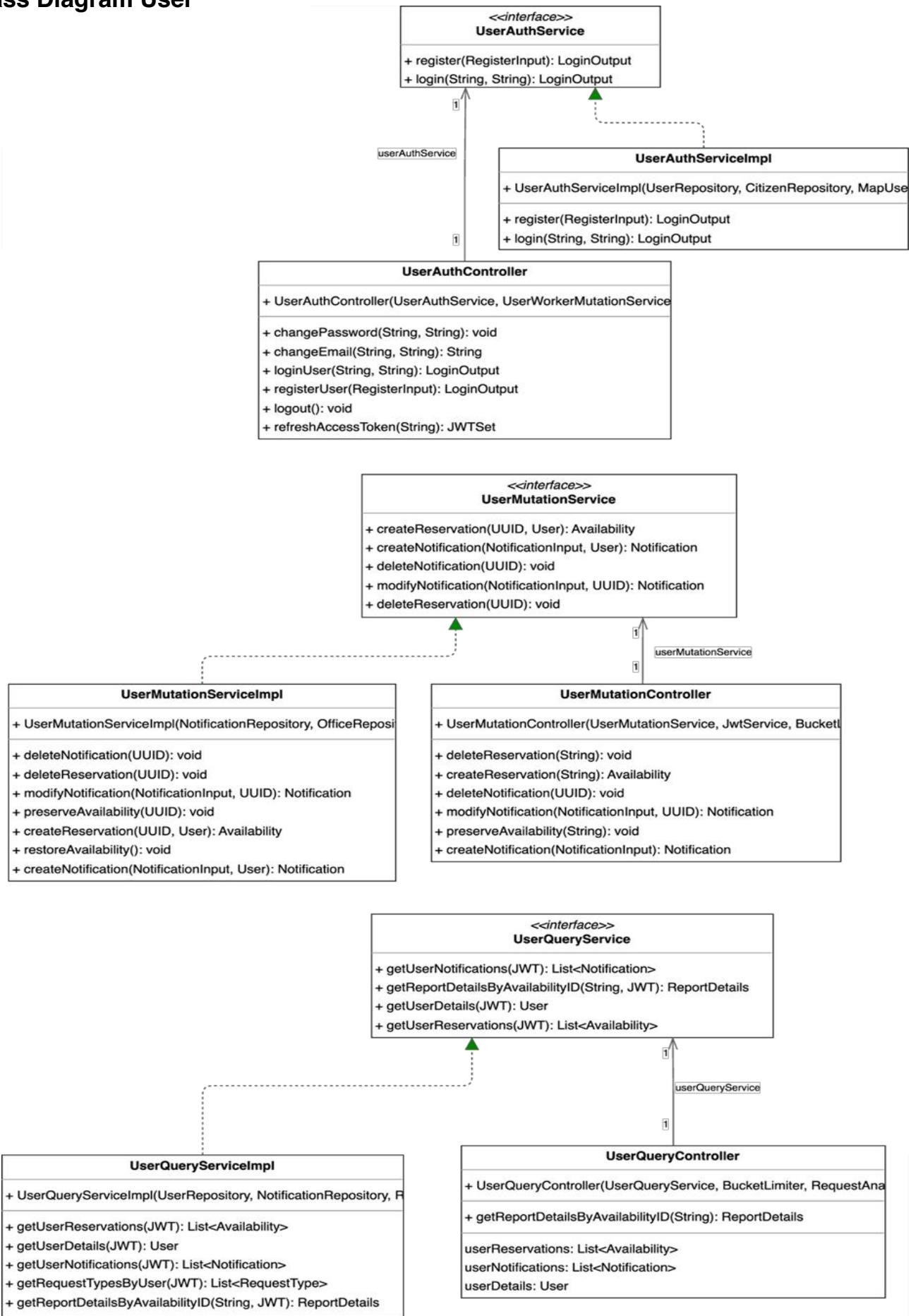
Class Diagram UserWorker



Class Diagram Worker



Class Diagram User



Sviluppo: progetto dell'architettura ed implementazione del sistema

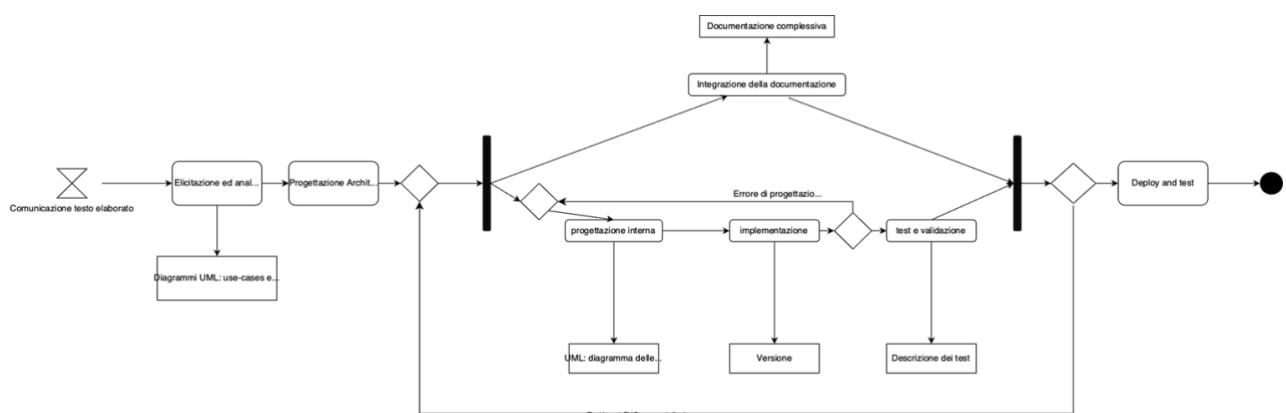
Note sul processo di sviluppo

Il processo di sviluppo è stato essenzialmente di tipo *Plan Driven e Incrementale*.

È stata eseguita un'analisi dei requisiti, nella quale sono stati generati gli use-case e i diagrammi delle attività.

Durante lo sviluppo è stata sfruttata una funzionalità di GitHub che permetteva di distribuire e organizzare le varie task, le quali erano suddivise tra Backend, Frontend e Testing, e a loro volta suddivise ulteriormente in TO DO, “in esecuzione” e “completate”.

Dopo ogni sostanziale implementazione/versione sono stati eseguiti gli appositi test. Non sempre le fasi di sviluppo sono state lineari, infatti sono state intervallate da numerose attività di refactoring.

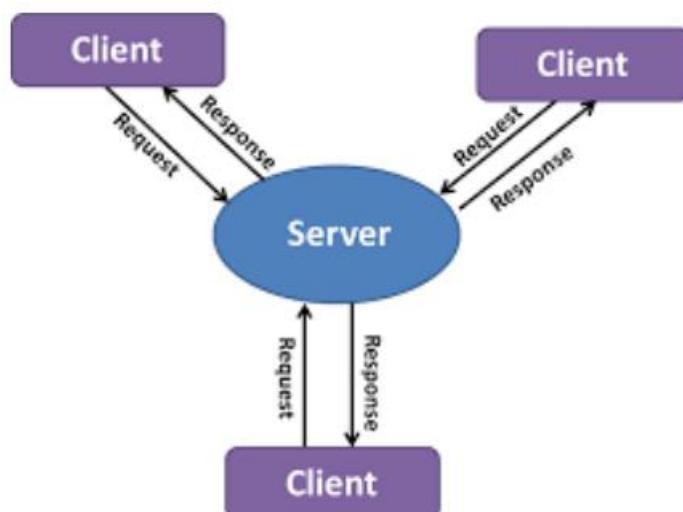


Progettazione e pattern architetturali usati

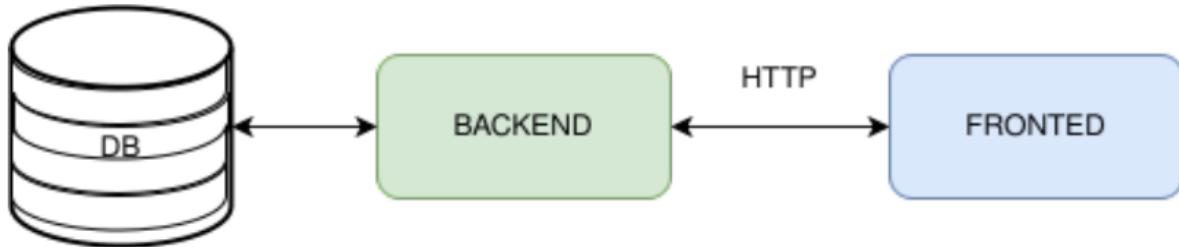
Il sistema è stato progettato utilizzando le tecniche di modellazione ad oggetti.

Abbiamo utilizzato un'architettura Client-Server.

Modello di sistema distribuito che presenta come i dati e l'elaborazione sono distribuiti attraverso un range di componenti. In questa architettura le funzionalità del sistema sono organizzate in servizi, con ogni servizio fornito da un server diverso. I Client sono gli utenti che possono accedere ai servizi attraverso i server.



Nel nostro abbiamo un solo server (BackEnd) e N client (FrontEnd).



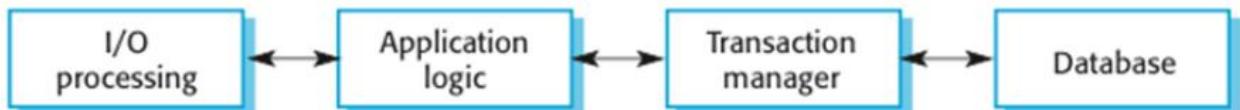
Backend

Il Backend è la parte server dell'applicazione responsabile della gestione dei dati.

È stato realizzato in Java utilizzando il **Framework Spring** e un **Database Postgres**. Si è scelto di usare nello specifico le componenti Spring Boot, Spring Web e Spring Data Jpa. Spring Boot serve a ridurre il codice di configurazione; garantisce un livello maggiore di astrazione il quale permette al programmatore di concentrarsi di più sull'aspetto logico del programma.

Spring Web, come suggerisce il nome, è una componente di Spring specifica per la creazione di applicazioni Web. È stato utilizzato per produrre un API Graphql con servizi per la gestione dei dati. L'ultimo componente, Spring Data Jpa, serve per la creazione di database relazionali con generazione automatica delle query.

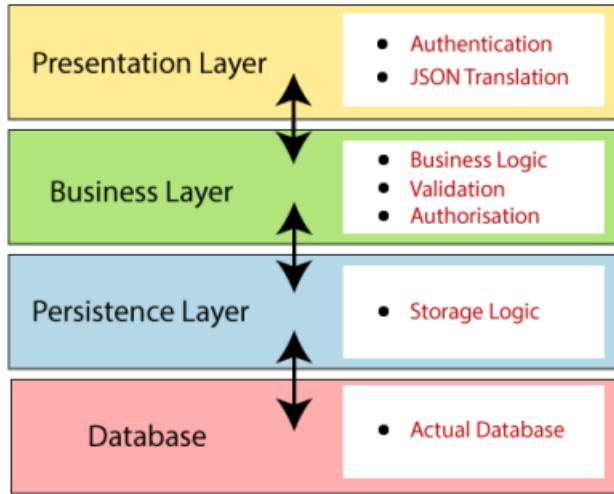
Il backend del progetto consiste in un'applicazione incentrata sui dati che processa le richieste degli utenti e aggiorna le informazioni di un database, anche chiamata Transaction processing application.



Architettura a Strati

Nello specifico, nel backend è stata utilizzata un'Architettura a Strati.

Nell'architettura a strati il sistema viene organizzato in un insieme di strati (layers o macchine astratte) ciascuno dei quali provvede ad un insieme di servizi. Le funzionalità sono correlate di strato in strato. Ciascuno fornisce servizi allo strato sottostante, fino all'ultimo livello che rappresenta il nucleo delle funzionalità che facilmente verranno usate da tutto il sistema.



Presentation Layer: implementato dalle classi Controller ha il compito di gestire le richieste HTTP, tradurre i parametri e il JSON in oggetti. Infine tutte le informazioni vengono passate al Business Layer.

Business Layer: gestisce la logica di business del programma e la validazione delle richieste, è implementato dalle classi Service e usa i servizi forniti dal Data Access Layer(Persistence Layer).

Persistence Layer: gestisce la logica dei dati, in altre parole effettua le query e trasforma gli oggetti di business(entità) da e in righe del database.

Implementazione e design pattern usati

Generalità

Si sorvola sui vari **pattern iterator** ed affini, poiché sono alla base della programmazione in Java.

Inversion of Control

Una caratteristica di Spring, è l'**IoC** (Inversion of Control), un design pattern nel quale le porzioni di codice fatte su misura ricevono il controllo di flusso da un framework generico.

Nella programmazione procedurale, il codice di un programma fatto su misura chiamerebbe delle librerie riutilizzabili per portare a termine delle task generiche, mentre nell'IoC, è il framework che si rifà a tali parti di codice per il medesimo fine.

Dependencies Injection

Nel contesto in cui abbiamo lavorato noi, ovvero il framework Spring, IoC si riferisce al DI (**D**ependencies **I**njection) che avviene con i “container IoC” in Java.

In questo specifico contesto, IoC si applica garantendo al framework il controllo sull'implementazione delle dipendenze che sono usate dagli oggetti, invece di garantirgli il controllo di flusso.

Gli “oggetti”, nel framework di Spring, sono chiamati “**Beans**”; essi formano la struttura dell’applicazione e, a seguito di quanto detto sopra, sono quindi gestiti dall’IoC container.

Nel nostro programma i Bean corrispondono a Controller, Service e Repository.

Singleton

Ogni Bean implementa il pattern **Singleton** (con configurazione specificata).

Abbiamo una cartella configurazione con la configurazione per ogni specifico Bean. Il Singleton è usato per assicurare che una classe abbia una sola istanza per tutto il ciclo di vita del

software e un unico punto di accesso globale. Le classi Singleton hanno costruttori privati per evitare la possibilità di istanziare più oggetti della stessa classe. Esiste un metodo statico che fa in modo che nessuna istanza venga creata oltre alla prima restituendo un riferimento a quella esistente.

Nel caso del Singleton dei Beans, Spring limita un singleton a un oggetto per Spring IoC Container, creando un solo Bean per ogni tipo di contesto dell'applicazione.

Spring esce quindi leggermente dalla definizione rigorosa di singleton perché un'applicazione può avere più container Spring e quindi ammettere l'esistenza di più oggetti di una stessa classe nel caso in cui vi siano più container.

DTO

Quando l'invocazione tra oggetti è remota è più conveniente permettere al client di gestire molti valori in un'unica operazione piuttosto che avere invocazioni distinte.

Il pattern DTO è usato quindi per trasferire dati tra un client e un server e permette di gestire più oggetti con una singola invocazione piuttosto che singolarmente.

L'oggetto DTO contiene un sottoinsieme dei dati del BusinessObject remoto, non ha metodi comportamentali e permette l'accesso diretto ai dati senza fare get/set.

Nel nostro caso, è stato utilizzato per gestire i tipi di oggetti in input, quali i filtri delle richieste e, in generale, tutti i valori inseriti dall'utente tramite la schermata del Frontend.

All'input vengono successivamente aggiunti dei campi per permettere l'inserimento dell'oggetto nel database.

Per quel che riguarda la gestione dell'output, abbiamo i valori di ritorno dell'utente:

- LoginOutput: contiene la coppia di JWT e l'id dell'utente appena autenticato
- ReportDetails: contiene tutte le cose che servono per il report alla fine di una registrazione (es. dati quali Fiscal Code, Name, Username, Office name ecc...)
- RequestAndOffices: è un oggetto che ritorniamo all'API che unisce per ogni oggetto una lista di uffici associati a quella Request. È un DTO di output fatto solo per il Frontend.

Dentro Helper/Map abbiamo le funzioni di map per mappare i valori di input del DTO in valori di output e viceversa.

Factory

Consiste nel creare un oggetto senza esporre la logica della creazione al client e si riferisce all'oggetto nuovo creato utilizzando un'interfaccia comune.

Il pattern Factory Method è utilizzato per fornire un'interfaccia unificata per la creazione di oggetti di connessione e di template per interagire con il server Redis, mentre la logica specifica di creazione degli oggetti è delegata ai metodi factory redisConnectionFactory() e redisTemplate().

Builder

Il design pattern Builder è un pattern creazionale utilizzato per costruire un oggetto complesso passo dopo passo. Questo approccio consente di separare la creazione di un oggetto complesso dalla sua rappresentazione, consentendo variazioni flessibili e facilitando la costruzione di oggetti con diverse configurazioni.

In Java, il pattern Builder viene implementato attraverso la creazione di una classe Builder dedicata per costruire l'oggetto desiderato. Questa classe Builder contiene metodi per impostare i diversi attributi dell'oggetto e restituisce infine l'oggetto completo una volta che tutte le impostazioni sono state specificate.

Noi l'abbiamo usato per il rate limiter.

Decorator

Il pattern Decorator permette di aggiungere funzionalità ad un oggetto già esistente senza modificarne le proprietà o la sua struttura. Si crea una classe Decorator che fa da wrapper alla classe originale e aggiunge funzionalità mantenendo i metodi della classe intatti. Il pattern decorator è utilizzato in tutto il progetto perché è un concetto nativo del framework Spring. Abbiamo utilizzato questi decoratori tramite Lombok, per i metodi costruttore, setter e getter, e tramite il log4j2.

Frontend

Il Frontend, anche detto Client è la parte grafica del software responsabile dell'interazione con l'utente. È stato realizzato in Dart mediante l'utilizzo delle librerie Flutter per la parte grafica.

Nel nostro caso l'applicazione è un **SPA** (Single Page Application), ovvero un'applicazione web o un sito che interagisce con l'utente riscrivendo dinamicamente riscrivendo la pagina web corrente con le nuove informazioni dal web server invece del metodo di default di caricare interamente pagine nuove. L'obiettivo sono transizioni più veloci che permettono al sito di assomigliare di più a una native app.

In Flutter, il pattern BLoC (Business Logic Component) è uno dei più diffusi per gestire lo stato dell'applicazione, separando la logica di business dall'interfaccia utente. Tuttavia, all'interno del pattern BloC abbiamo utilizzato il pattern **Stream/Observer**:

Uno **Stream** in Flutter è un flusso di dati asincrono.

Un flusso può emettere un valore in qualsiasi momento e può essere ascoltato da un numero arbitrario di osservatori. I flussi sono utili per gestire dati in tempo reale, come aggiornamenti di stato o cambiamenti nel database.

Un **Observer** è un widget che ascolta i cambiamenti su un flusso di dati. Quando il flusso emette un nuovo valore, l'Observer si aggiorna per riflettere quel valore nell'interfaccia utente. Abbiamo usato gli Observer per aggiornare dinamicamente l'interfaccia utente in base ai cambiamenti nello stato dell'applicazione.

Abbiamo creato classi e BLoC che contengono la logica di business dell'applicazione., All'interno del BLoC, si definiscono gli Stream che rappresentano lo stato dell'applicazione.

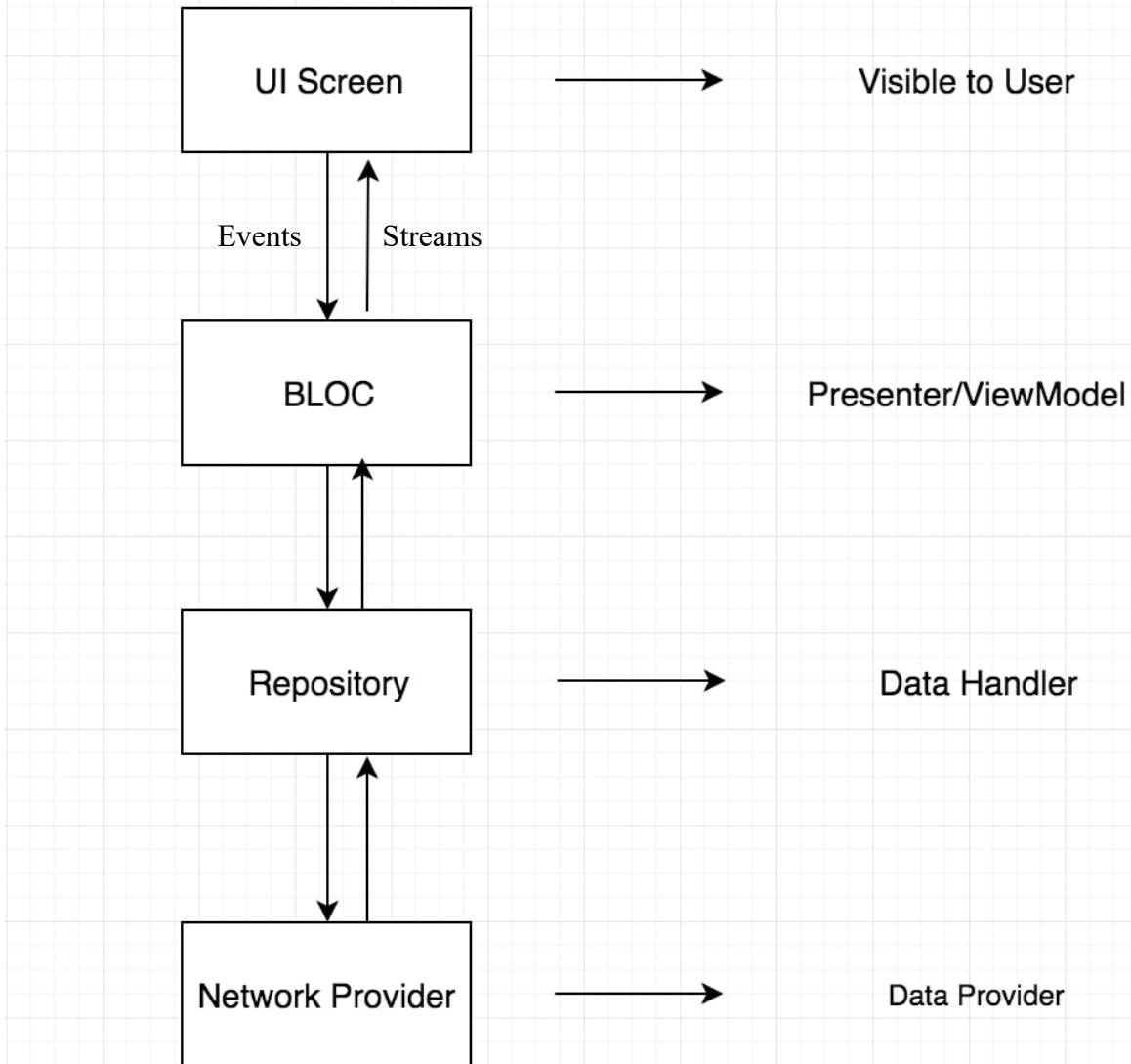
I flussi definiti nel BLoC vengono esposti all'esterno, generalmente tramite getter. Gli altri widget dell'applicazione possono ascoltare questi flussi per ricevere aggiornamenti sullo stato dell'applicazione.

Nei widget dell'interfaccia utente che devono rispondere ai cambiamenti di stato, si utilizzano Observer per ascoltare i flussi esposti dal BLoC. Quando il flusso emette un nuovo valore, gli Observer si aggiornano e riflettono questi cambiamenti nell'interfaccia utente.

Questo pattern è una sottocategoria del pattern Observer, in Java.

All'interno del Frontend abbiamo usato anche il pattern DTO, già spiegato precedentemente.

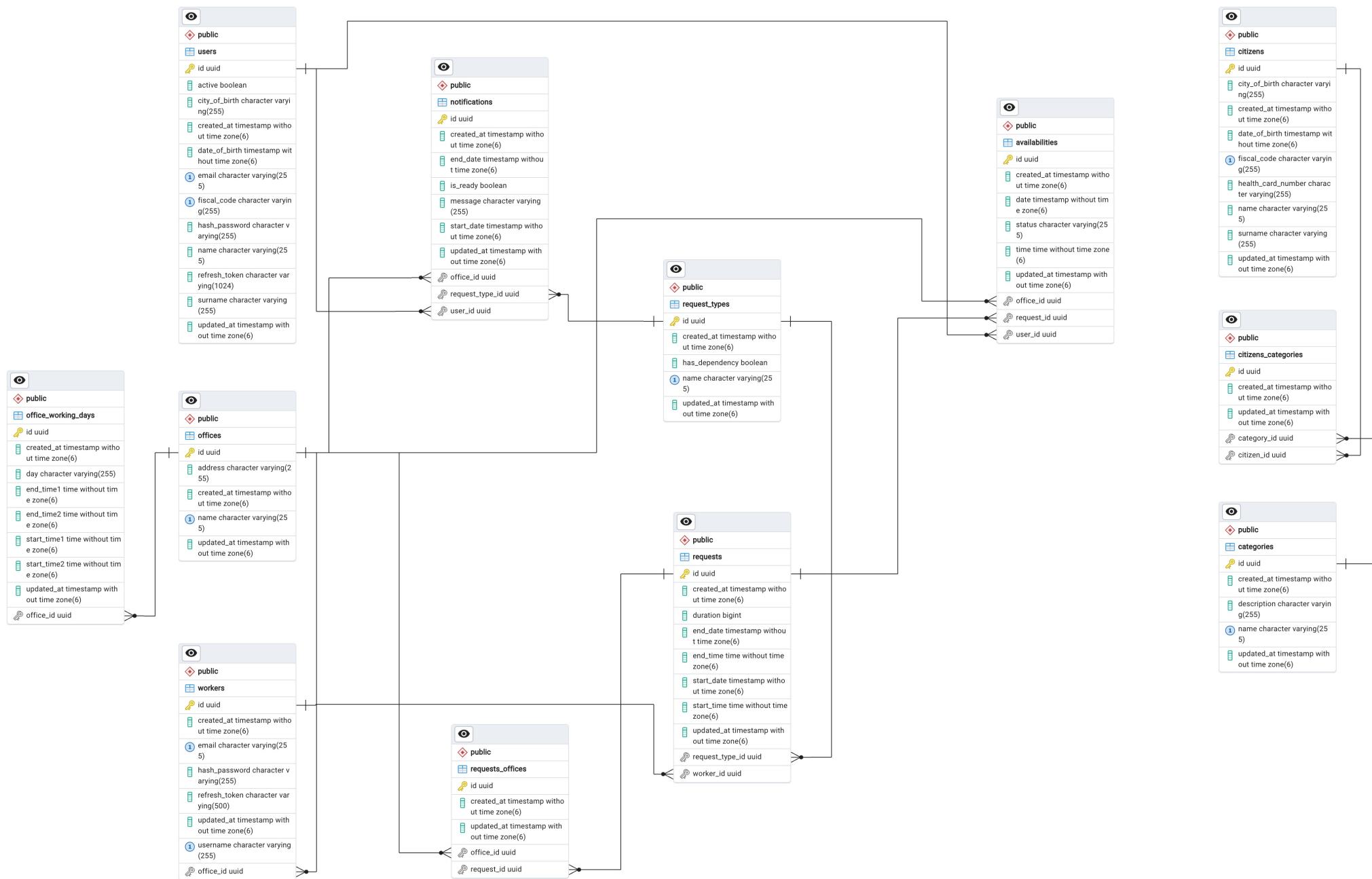
BLOC pattern for Flutter




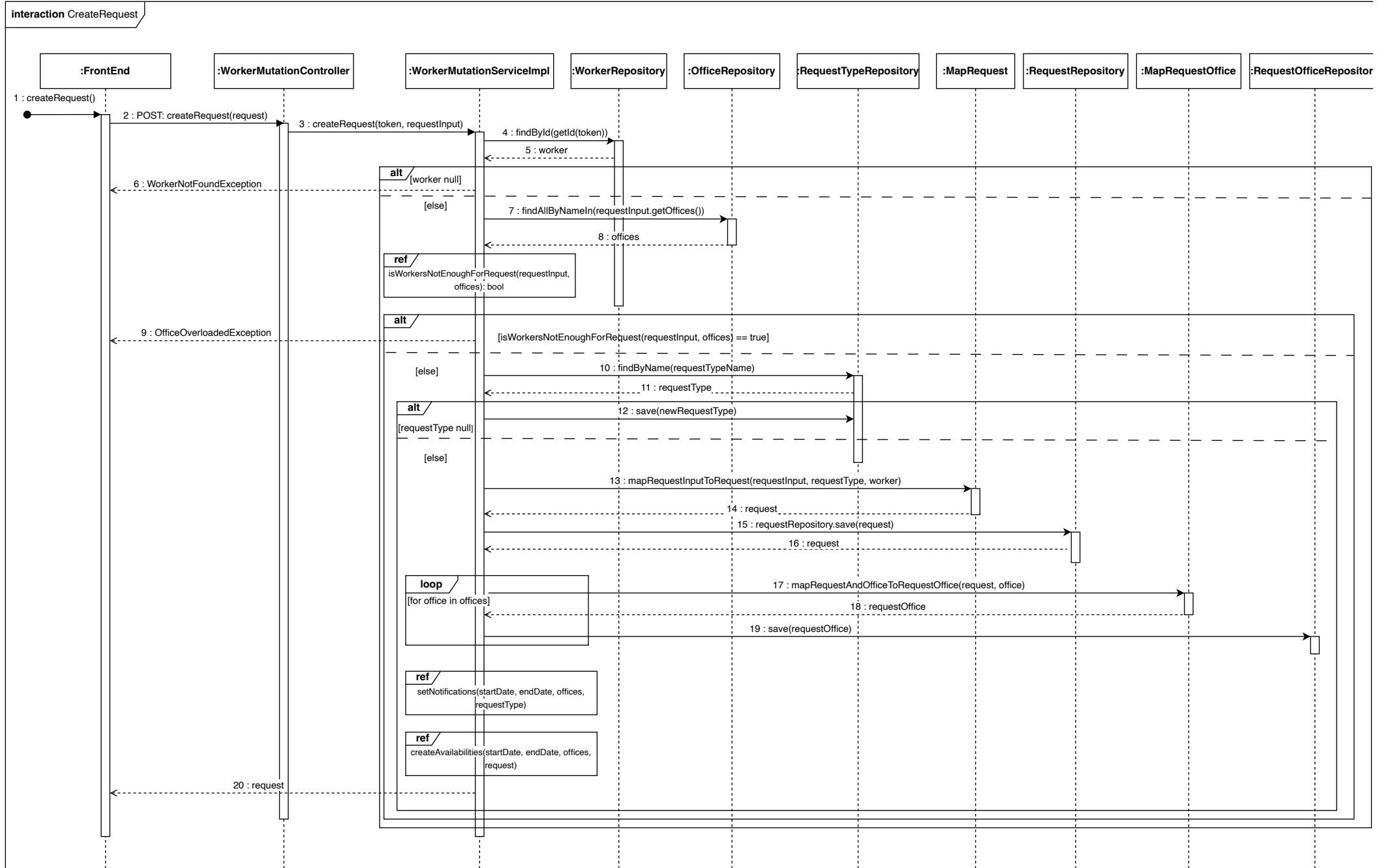
Progetto dell'architettura e implementazione del database

Per lo sviluppo del sistema è stato un database relazionale Postgres e la libreria Spring Data JPA per interfacciare il sistema al Database tramite un sistema di Repository. La logica del database è stata di tipo incrementale seguita da numerosi refactor per implementare nuove funzionalità, questo ha portato al non seguire uno schema logico. Di sotto possiamo trovare lo schema dell'attuale database.

SCHEMA FISICO:







Attività di test e validazione

Generalità

La fase di testing del software è pensata per dimostrare che il programma fa quanto richiesto e verificare che non ci siano difetti di sviluppo prima che il software venga rilasciato. Esistono tre fasi del processo di test, development testing, release testing, user testing.

Development Testing

Il test di sviluppo include tutte le attività di test che sono svolte dal team di sviluppo, nel nostro caso:

- Unit testing
- Component testing

Per effettuare questi test abbiamo utilizzato **Junit5** (è al centro dell'infrastruttura di testing e serve principalmente come base per definire le funzioni di test, funzioni di supporto e le classi di test mediante le apposite annotazioni) e una pipeline CI/CD integrata in GitHub. La pipeline CI/CD eseguiva automaticamente tutti i test a ogni implementazione di grandi features.

Abbiamo usato MockMVC tramite l'autoconfiguratore implementato in Spring. In ogni test viene creato l'utente “mockato” (fittizio) e una volta eseguiti tutti i test sul mock user, l'utente viene eliminato.

Per testare l'API GraphQL dall'esterno abbiamo usato GraphQLEstimator, che utilizza il design pattern di builder, e Web Application Context, che utilizza sempre il design pattern di builder.

Per effettuare questi test abbiamo usato il paradigma **given-when-then**; esso consiste nella suddivisione del codice di test in tre blocchi:

- **given**: in questo blocco mettiamo il sistema in uno stato noto definendo i dati di partenza e il comportamento delle componenti fittizie.
- **when**: in questo blocco vengono descritte le operazioni compiute dall'utente.
- **then**: questo è il blocco di verifica, osserviamo l'output del sistema e controlliamo se si è comportato come ci aspettiamo.

Adesso analizziamo alcuni dei test scritti, iniziando dalla classe da testare. Per lo scopo di questa documentazione il codice di seguito esposto è solo un'estrappolazione del codice reale.

```
@BeforeAll
void createMockUser() {
    // save user if not exists
    if (citizenRepository.findByFiscalCode("NTLZLD98R52G273R").isPresent()) {
        citizenRepository.deleteByFiscalCode("NTLZLD98R52G273R");
        userRepository.deleteByFiscalCode("NTLZLD98R52G273R");
    }

    Citizen citizen = new Citizen();

    citizen.setFiscalCode("NTLZLD98R52G273R");
    citizen.setName("Zelda");
    citizen.setSurname("NotLink");
    citizen.setCityOfBirth("Palermo");
    citizen.setDateOfBirth(
        new Date(new SimpleDateFormat(pattern: "yyyy-MM-dd").parse(source: "1998-10-12").getTime())
    );
    citizen.setHealthCardNumber("1234567890");

    citizenRepository.save(citizen);

    System.out.println("\u001B[33m");
    System.out.println("[!] Mock User Created");
    System.out.println("\u001B[0m");
}
```

Crea uno user fittizio e lo salva nel database se non esiste.

```
User user = new User();

user.setName("Zelda");
user.setSurname("NotLink");
user.setFiscalCode("NTLZLD98R52G273R");
user.setEmail("zeldanotlink@gmail.com");
user.setActive(true);
user.setUpdatedAt(new Date());
user.setCreatedAt(new Date());
user.setCityOfBirth("Palermo");
user.setDateOfBirth(
    new Date(new SimpleDateFormat(pattern: "yyyy-MM-dd").parse(source: "1998-10-12").getTime())
);
user.setHashPassword(passwordEncoder.encode(rawPassword: "ciao"));
user.setRefreshToken(
    jwtService.generateRefreshToken(
        userRepository.save(user).getId()
    ).getToken()
);

userRepository.save(user);

USER_ID = user.getId();

System.out.println("\u001B[33m");
System.out.println("[!] Mock User Created");
System.out.println("\u001B[0m");
```

Parte per registrare il citizen mockato, sempre nella fase di given.

```
Object makeGraphQLRequest(@NotNull String graphQLDocument, String pathResponse, Class<?> objectClass, JWT bearerToken) {
    if (bearerToken == null)
        return graphQLTester
            .mutate()
            .build()
            .document(graphQLDocument)
            .execute()
            .path(pathResponse)
            .entity(objectClass)
            .get();

    return HttpGraphQLTester.builder(
        MockMvcWebClient.bindToApplicationContext(context)
            .configureClient()
            .baseUrl(BASE_URL)
            .build()
            .mutate()
    )
        .headers(headers -> headers.setBearerAuth(bearerToken.getToken()))
        .build()
        .document(graphQLDocument)
        .execute();
}
```

Parametri:

graphQLDocument- la query GraphQL come stringa

pathResponse- il path della risposta

objectClass- la classe della risposta

bearerToken- il bearer token (opzionale)

Risposta:

La risposta della query, come oggetto della classe passata come parametro se il bearerToken è null, altrimenti come GraphQITesterResponse.

```
@Test
@Order(1)
void registerUser() {
    String registerUser = GraphQLOperations.registerUser.getGraphQL(
        ...args: "NTLZLD98R52G273R",
        "zeldanotlink@gmail.com",
        "Zelda",
        "NotLink",
        "Palermo",
        "1998-10-12",
        "ciao"
    );

    System.out.println("[!] registerUser query: \n" + registerUser + "\n");

    LoginOutput loginOutput = (LoginOutput) makeGraphQLRequest(
        registerUser,
        pathResponse: "data.registerUser",
        LoginOutput.class,
        bearerToken: null
    );

    System.out.println("[!] registerUser output: \n" + loginOutputToJson(loginOutput) + "\n");

    Assertions.assertNotNull(loginOutput);
    JWTSet jwtSet = loginOutput.jwtSet();

    Assertions.assertNotNull(jwtSet);
    Assertions.assertNotNull(loginOutput.id());
    MOCK_USER_ID = loginOutput.id();
    Assertions.assertTrue(jwtSet.accessToken().matches(JWT_REGEX));
    Assertions.assertTrue(jwtSet.refreshToken().matches(JWT_REGEX));
    REFRESH_TOKEN = new JWT(jwtSet.refreshToken());
}
}
```

Registra un nuovo user e controlla se la risposta è corretta. Salva lo user id e salva il refresh token per i prossimi test.

```

@Test
void loginUser() {
    if (MOCK_USER_ID == null)
        registerUser();

    final String JWT_REGEX = "^[\\w-]*\\. [\\w-]*\\. [\\w-]*$";

    String loginUser = GraphQLOperations.loginUser.getGraphQL(...args: "NTLZLD98R52G273R", "ciao");

    System.out.println("[!] Login query: \n" + loginUser + "\n");

    LoginOutput loginOutput = (LoginOutput) makeGraphQLRequest(
        loginUser,
        pathResponse: "data.loginUser",
        LoginOutput.class,
        bearerToken: null
    );

    System.out.println("[!] Login output: \n" + loginOutputToJson(loginOutput) + "\n");

    Assertions.assertNotNull(loginOutput);

    JWTSet jwtSet = loginOutput.jwtSet();

    Assertions.assertNotNull(jwtSet);

    Assertions.assertEquals(MOCK_USER_ID, loginOutput.id());

    Assertions.assertTrue(jwtSet.accessToken().matches(JWT_REGEX));
    Assertions.assertTrue(jwtSet.refreshToken().matches(JWT_REGEX));

    REFRESH_TOKEN = new JWT(jwtSet.refreshToken());
}

```

Logga uno user e controlla se la risposta è corretta. Salva il refresh token per i test successivi.

```
@Test
@WithMockUser(authorities = {"USER", "VALIDATED"})
void logoutUser() {
    if (MOCK_USER_ID == null)
        registerUser();

    System.out.println("[!] ID: " + MOCK_USER_ID);

    JWT token = jwtService.generateAccessToken(MOCK_USER_ID);

    System.out.println("[!] Generated token: " + token);

    String logout = GraphQLOperations.logout.getGraphQL();

    System.out.println("[!] Logout query: \n" + logout + "\n");

    GraphQLTester.Response response = (GraphQLTester.Response) makeGraphQLRequest(
        logout,
        pathResponse: "data.logout",
        objectClass: null,
        token
    );

    Assertions.assertNotNull(response);
    Assertions.assertDoesNotThrow(response.errors() :: verify);

    System.out.println("[!] Logout output: \n" + logoutJson() + "\n");
}
```

Effettua il logout dello user e controlla che la risposta sia corretta.

```
@Test
@WithMockUser(authorities = {"USER", "VALIDATED"})
void changePassword() {
    if (MOCK_USER_ID == null)
        registerUser();

    JWT token = jwtService.generateAccessToken(MOCK_USER_ID);

    System.out.println("[!] Generated token: " + token);

    String changePassword = GraphQLOperations.changePassword.getGraphQL(...args: "ciao1", "ciao");

    System.out.println("[!] changePassword query: \n" + changePassword + "\n");

    GraphQLTester.Response response = (GraphQLTester.Response) makeGraphQLRequest(
        changePassword,
        pathResponse: "data.changePassword",
        objectClass: null,
        token
    );

    Assertions.assertNotNull(response);
    Assertions.assertDoesNotThrow(response.errors()::verify);

    System.out.println("[!] changePassword output: \n" + changePasswordJson() + "\n");
}
```

Cambia la password di uno user e controlla se la risposta è corretta.

```
@Test
@WithMockUser(authorities = {"USER", "VALIDATED"})
void changeEmail() {
    if (MOCK_USER_ID == null)
        registerUser();

    JWT token = jwtService.generateAccessToken(MOCK_USER_ID);

    System.out.println("[!] Generated token: " + token);

    String changeEmail = GraphQLOperations.changeEmail.getGraphQL(...args: "new@email.com", "zeldanotlink@gmail.com");

    System.out.println("[!] changeEmail query: \n" + changeEmail + "\n");

    GraphQLTester.Response response = (GraphQLTester.Response) makeGraphQLRequest(
        changeEmail,
        pathResponse: "data.changeEmail",
        objectClass: null,
        token
    );

    Assertions.assertNotNull(response);
    Assertions.assertDoesNotThrow(response.errors() :: verify);

    String newEmail = response.path("data.changeEmail") Path
        .entity(String.class) Entity<String, capture of ?>
        .get();

    System.out.println("[!] changeEmail output: \n" + changeEmailJson(newEmail) + "\n");
}
```

Cambia l'email di uno user e controlla se la risposta è corretta.

```

@Test
@WithMockUser(authorities = {"USER", "VALIDATED"})
void refreshAccessToken() {
    if (MOCK_USER_ID == null)
        registerUser();
    if (REFRESH_TOKEN == null)
        loginUser();

    JWT token = jwtService.generateAccessToken(MOCK_USER_ID);
    System.out.println("[!] Generated token: " + token);

    String refreshToken = GraphQLOperations.refreshAccessToken.getGraphQL(REFRESH_TOKEN.getToken());
    System.out.println("[!] refreshAccessToken query: \n" + refreshToken + "\n");

    GraphQLTester.Response response = (GraphQLTester.Response) makeGraphQLRequest(
        refreshToken,
        pathResponse: "data.refreshAccessToken",
        objectClass: null,
        token
    );

    Assertions.assertNotNull(response);
    Assertions.assertDoesNotThrow(response.errors()::verify);

    JWTSet jwtSet = response.path("data.refreshAccessToken") Path
        .entity(JWTSet.class) Entity<JWTSet, capture of ?>
        .get();

    Assertions.assertTrue(jwtSet.accessToken().matches(JWT_REGEX));
    Assertions.assertTrue(jwtSet.refreshToken().matches(JWT_REGEX));

    System.out.println("[!] refreshAccessToken output: \n" + jwtSet.accessToken() + "\n");
}

```

Rinnova l'access token di uno user e controlla se la risposta è corretta.

```
@Test
@WithMockUser(authorities = {"USER", "VALIDATED"})
void getRequestTypesByUser() {

    JWT token = jwtService.generateAccessToken(USER_ID);

    System.out.println("[!] Generated token: \n" + token + "\n");

    String getRequestTypesByUser = GraphQLOperations.getRequestTypesByUser.getGraphQL();

    GraphQLTester.Response response = (GraphQLTester.Response) makeGraphQLRequest(getRequestTypesByUser,
        pathResponse: "data.getRequestTypesByUser", objectClass: null, token);

    Assertions.assertNotNull(response);

    Assertions.assertDoesNotThrow(response.errors()::verify);

    List<RequestType> responseType = response
        .path("data.getRequestTypesByUser") Path
        .entityList(RequestType.class) EntityList<RequestType>
        .get();

    responseType.forEach(System.out::println);

    System.out.println("\u001B[32m");
    System.out.println("[!] getRequestTypesByUser successful");
    System.out.println("\u001B[0m");
}
```

Controlla se la risposta non è null e se non ci sono errori.

```

@Test
@WithMockUser(authorities = {"USER", "VALIDATED"})
@sneakyThrows(ParseException.class)
void getUserNotifications() {

    UUID notificationID = userMutationService.createNotification(
        new NotificationInput(
            new Date(new SimpleDateFormat(pattern: "yyyy-MM-dd").parse(source: "2023-08-05").getTime()),
            new Date(new SimpleDateFormat(pattern: "yyyy-MM-dd").parse(source: "2023-09-01").getTime()),
            officeName: "Sede di Napoli",
            requestTypeNames: "ritiro passaporti"),
        userRepository.findById(USER_ID).orElseThrow().getId());

    JWT token = jwtService.generateAccessToken(USER_ID);

    System.out.println("![ Generated token: \n" + token + "\n");
    String getUserNotifications = GraphQLOperations.getUserNotifications.getGraphQL();

    GraphQLTester.Response response = (GraphQLTester.Response) makeGraphQLRequest(getUserNotifications,
        pathResponse: "data.getUserNotifications", objectClass: null, token);

    Assertions.assertNotNull(response);
    Assertions.assertDoesNotThrow(response.errors()::verify);
    List<Notification> notifications = response.path("data.getUserNotifications") Path
        .entityList(Notification.class) EntityList<Notification>
        .get();

    notifications.forEach(System.out::println);

    userMutationService.deleteNotification(notificationID);

    System.out.println("\u001B[32m");
    System.out.println("![ getUserNotifications successful");
    System.out.println("\u001B[0m");
}

```

Controlla se la risposta non è null e se non ci sono errori. Il test crea una notifica per lo user e poi la cancella.

```

@Test
@WithMockUser(authorities = {"USER", "VALIDATED"})
@sneakyThrows(ParseException.class)
void getUserDetails() {

    JWT token = jwtService.generateAccessToken(USER_ID);

    System.out.println("![ Generated token: \n" + token + "\n");

    String getUserDetails = GraphQLOperations.getUserDetails.getGraphQL();

    GraphQLTester.Response response = (GraphQLTester.Response) makeGraphQLRequest(getUserDetails,
        pathResponse: "data.getUserDetails", objectClass: null, token);

    Assertions.assertNotNull(response);
    Assertions.assertDoesNotThrow(response.errors()::verify);

    User user = response.path("data.getUserDetails").entity(User.class).get();

    Assertions.assertEquals(USER_ID, user.getId());
    Assertions.assertEquals(expected: "Zelda", user.getName());
    Assertions.assertEquals(expected: "NotLink", user.getSurname());
    Assertions.assertEquals(expected: "NTLZLD98R52G273R", user.getFiscalCode());
    Assertions.assertEquals(new Date(
        new SimpleDateFormat(pattern: "yyyy-MM-dd").parse(source: "1998-10-12").getTime(),
        user.getDateOfBirth()));
    Assertions.assertEquals(expected: "Palermo", user.getCityOfBirth());

    System.out.println(user);

    System.out.println("\u001B[32m");
    System.out.println("![ getUserDetails successful");
    System.out.println("\u001B[0m");
}

```

Prende le informazioni dello User per il report della registrazione. Controlla che la risposta non sia null e che non ci siano errori.

```
@EnableJpaRepositories
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class UserRepositoryTest {
    @Autowired
    private UserRepository userRepository;

    /**
     * This method tests the ability to find a user by its fiscal code.
     */
    @Test
    void createUser() {
        Assertions.assertNotNull(userRepository.findByFiscalCode("BLSCLL96D55E4630").orElse(null));
        System.out.println("[!] User found: " + userRepository.findByFiscalCode("BLSCLL96D55E4630") + "\n");
    }
}
```

Test della repository.

```
@Test
void contextLoads() {
    Assertions.assertNotNull(passportEaseApplication);
}
```

Controlla che l'applicazione sia caricata interamente.