Documentazione progetto Ingegneria del Software

Software di gestione dei pazienti diabetici

Università di Verona

Imbriani Paolo - VR500437
Irimie Fabio - VR501504

9 luglio 2025

Indice

1 Requisiti e Use Case

1.1 Note generali

Il software che si andrà a sviluppare è un sistema di telemedicina di un servizio clinico per la gestione di pazienti diabetici. Gli attori principali del sistema sono i *medici* (diabetologi) e i *pazienti*; questi ultimi hanno credenziali di accesso al sistema fornite dagli amministratori del servizio con cui possono autenticarsi. Se l'autenticazione va a buon fine allora l'utente verrà indirizzato alla propria *home page* in cui potrà visualizzare le informazioni relative al loro ruolo. Nel seguente diagramma dei casi d'uso sono rappresentati i principali attori e le loro interazioni con il sistema. Notare che diamo per scontato che tutti gli autori siano già autenticati per semplificare il diagramma.

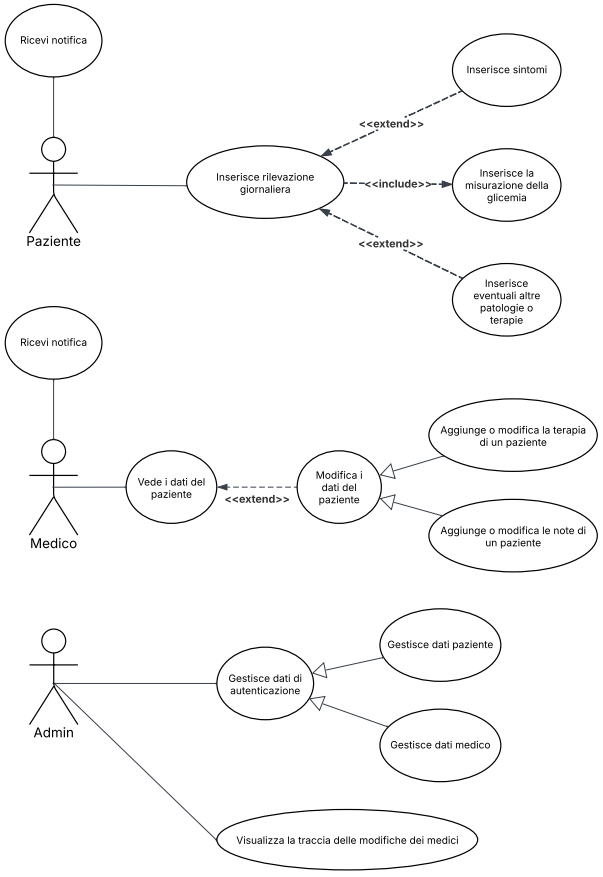

Figura 1: Diagramma dei casi d'uso

1.2 Casi d'uso del paziente

Una volta che il paziente si è autenticato ed è entrato all'interno della sua area riservata, egli può inserire i dati giornalieri relativi alla sua glicemia (prima e

dopo ogni pasto); per fare questo ha bisogno di inserire dati quali: sintomi, rilevazione glicemica, eventuali altre patologie o terapie.

1.2.1 Inserimento dei dati giornalieri

Attore: Paziente
Precondizioni: Il paziente deve essere autenticato
Passi:

1. Il paziente accede alla sua home page
2. Il paziente entra dentro l'area di inserimento dei dati giornalieri
3. Il paziente inserisce il dato della sua glicemia prima pasto e dopo pasto
    - Il paziente può aprire un ulteriore finestra per inserire i sintomi, le terapie e le patologie, con oppurtuna data
    - Può anche inserire le assunzioni di insulina o qualsiasi farmaco prescritto dal diabetologo, specificandone giorno, ora, farmaco e quantità assunta
4. Il paziente inserisce la data e ora di rilevazione
5. Il paziente conferma l'inserimento dei dati

Postcondizioni: La rilevazione è inserita

Andiamo a specificare come viene gestito **l'inserimento dei dati** del paziente: i dati glicemici sono quelli che il paziente inserisce giornalmente e obbligatori per inviare le rilevazioni. Dopo aver inserito i dati, l'utente può anche:

- inserire specifiche sulla patalogia, terapia o sintomi
- inserire le assunzioni di insulina o qualsiasi altro farmo prescritto dal diabetologo

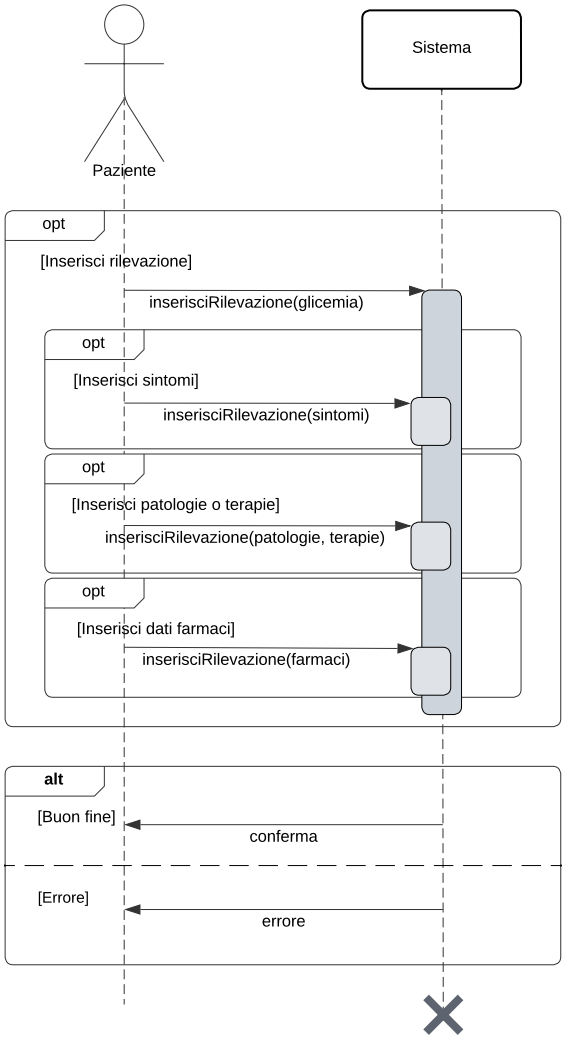

Figura 2: Sequence Diagram della rilevazione del paziente

1.3 Casi d'uso del medico

1.3.1 Visualizzare i dati del paziente

Attore: Medico
Precondizioni: Il medico deve essere autenticato
Passi:

1. Il medico accede alla sua area riservata
2. Il medico può accedere alla lista dei pazienti
3. Il medico può visualizzare i dati del paziente e lo storico delle rilevazioni

Postcondizioni: nessuna

Il medico una volta che ha acceduto alla sua area riservata può visualizzare i dati di tutti i pazienti.

1.3.2 Modificare i dati del paziente

Attore: Medico
Precondizioni: Il medico deve essere autenticato
Passi:

1. Il medico accede alla sua area riservata
2. Il medico accede alla lista dei pazienti
3. Il medico seleziona il paziente che vuole gestire
4. Il medico può decidere tra le seguenti opzioni:
    - Aggiungere o modificare la terapia del paziente
    - Aggiungere o modificare le note di un paziente
5. Il medico conferma le modifiche
6. Il sistema aggiorna i dati del paziente

Postcondizioni: La modifica è effettuata
Sequenza alternativa 1: il medico può in qualunque momento decidere di annullare le modifiche e ritornare alla lista dei pazienti
Postcondizioni: La modifica non è effettuata

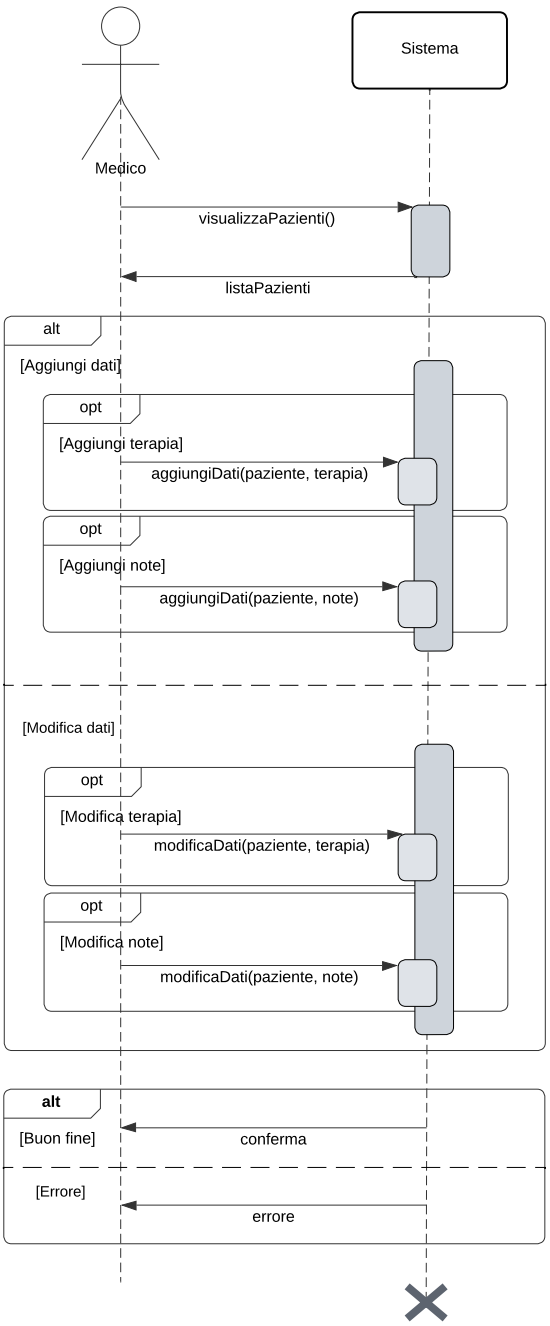

Figura 3: Sequence Diagram del medico

1.4 Ricezione delle notifiche

Il paziente può ricevere notifiche dal sistema; se il pazienta si dimentica di assumere i farmaci il sistema può inviare una notifica per ricordarglielo.

Attore: Paziente o Medico
Precondizioni: Il paziente o medico deve essere autenticato
Passi:

1. Il paziente o medico accede alla sua home page
2. Il paziente o medico entra dentro la sezione delle notifiche
3. Il paziente o medico può visualizzare:
    - Notifiche già lette
    - Notifiche non lette

Postcondizioni: Se viene visualizzata una notifica, questa viene marcata come letta

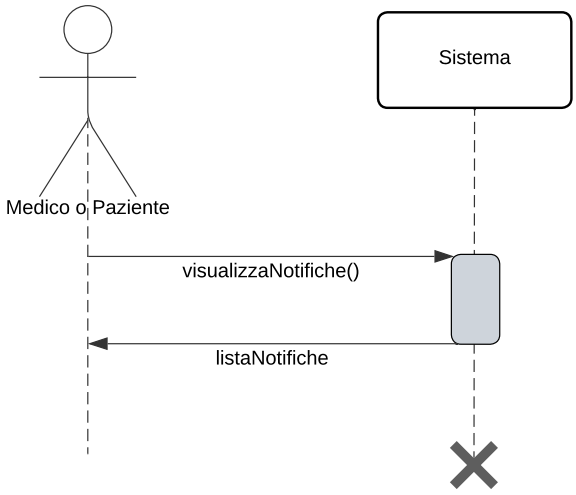

Figura 4: Sequence Diagram della notifica con lista

Il sistema invia delle notifiche ai seguenti attori:

- **Al paziente**: per ricordargli di inserire i dati giornalieri
- **Al medico di riferimento del paziente**: per avvisare che il paziente non ha seguito per più di tre giorni consecutivi le prescrizioni.
- **A tutti i medici**: per segnalare i pazienti che registrano livelli di glicemia sopra le soglie indicate.

1.5 Casi d'uso dell'amministratore

L'amministratore del servizio può gestire gli utenti del sistema. Si occupa di creare, modificare e cancellare gli account dei pazienti e dei medici. Per agevolare l'aggiunta di utenti, ci si può registrare tramite un form di registrazione (che specificherà il ruolo), ma l'amministratore deve comunque approvare la registrazione per rendere l'utente attivo nel sistema.

1.5.1 Gestione dell'utente

Attore: Amministratore
Precondizioni: L'amministratore deve essere autenticato
Passi:
1. L'amministratore accede alla sua area riservata
2. L'amministratore visualizza la lista degli utenti
3. L'amministratore può:
    - Creare un nuovo utente
    - Modificare un utente
    - Cancellare un utente

Postcondizioni: L'utente è creato, modificato o cancellato

1.5.2 Gestione delle richieste di registrazione

Attore: Amministratore
Precondizioni: L'amministratore deve essere autenticato
Passi:
1. L'amministratore accede alla sua area riservata
2. L'amministratore visualizza la lista delle richieste
3. Seleziona una richiesta di registrazione
4. L'amministratore può:
    - Accettare una registrazione
    - Rifiutare una registrazione

Postcondizioni: L'utente è accettato o rifiutato

1.6 Visualizzazione della traccia dei medici

Il sistema tiene traccia di ogni cambiamento o modifica che i medici effettuano sui pazienti, per ragioni di sicurezza.

- L'amministratore può visualizzare questa traccia per verificare che i medici non stiano effettuando operazioni non autorizzate.
- Il medico può visualizzare la traccia per verificare le operazioni effettuate precedentemente su un paziente da parte di altri medici

Attore: Amministratore
Precondizioni: L'amministratore deve essere autenticato
Passi:
1. L'amministratore accede alla sua area riservata
2. L'amministratore accede alla pagina di visualizzazione delle tracce

Postcondizioni: nessuna

Attore: Medico
Precondizioni: Il medico deve essere autenticato
Passi:
1. Il medico accede alla sua area riservata

2. Il medico accede alla pagina di visualizzazione dei pazienti
3. Il medico seleziona il paziente di cui vuole visualizzare la traccia
4. Il medico accede alla schermata di visualizzazione della traccia dell'utente

Postcondizioni: nessuna

1.7 Activity diagram

Per semplicità non verrà rappresentato il fatto di poter ripetere le operazioni più volte, ma si darà per scontato che l'utente possa tornare indietro e ripetere senza riavviare il software. Vengono quindi rappresentate soltanto le singole attività di operazione.

Si da inoltre per scontato che per ogni attività (tranne la registrazione) l'utente sia già autenticato e che non ci siano errori di autenticazione.

1.7.1 Registrazione di un utente

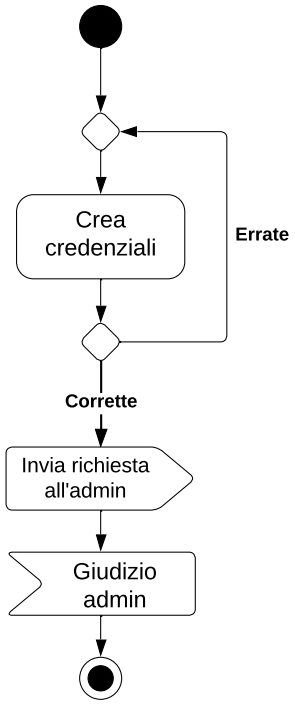

Figura 5: Activity diagram della registrazione di un utente

1.7.2 Attività del paziente

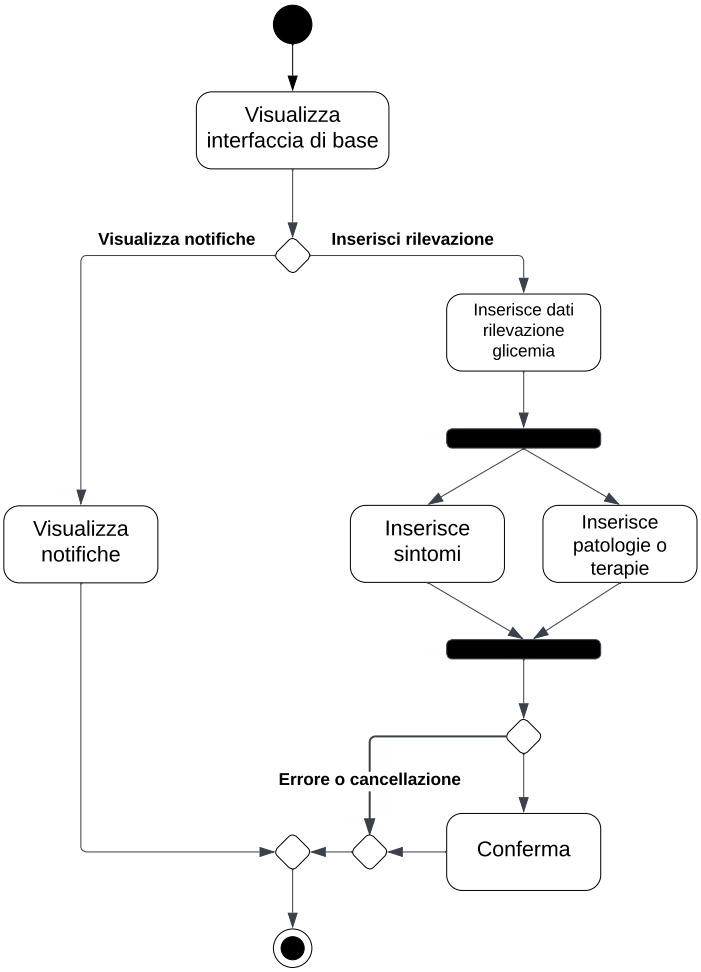

Figura 6: Activity diagram del paziente

1.7.3 Attività del medico

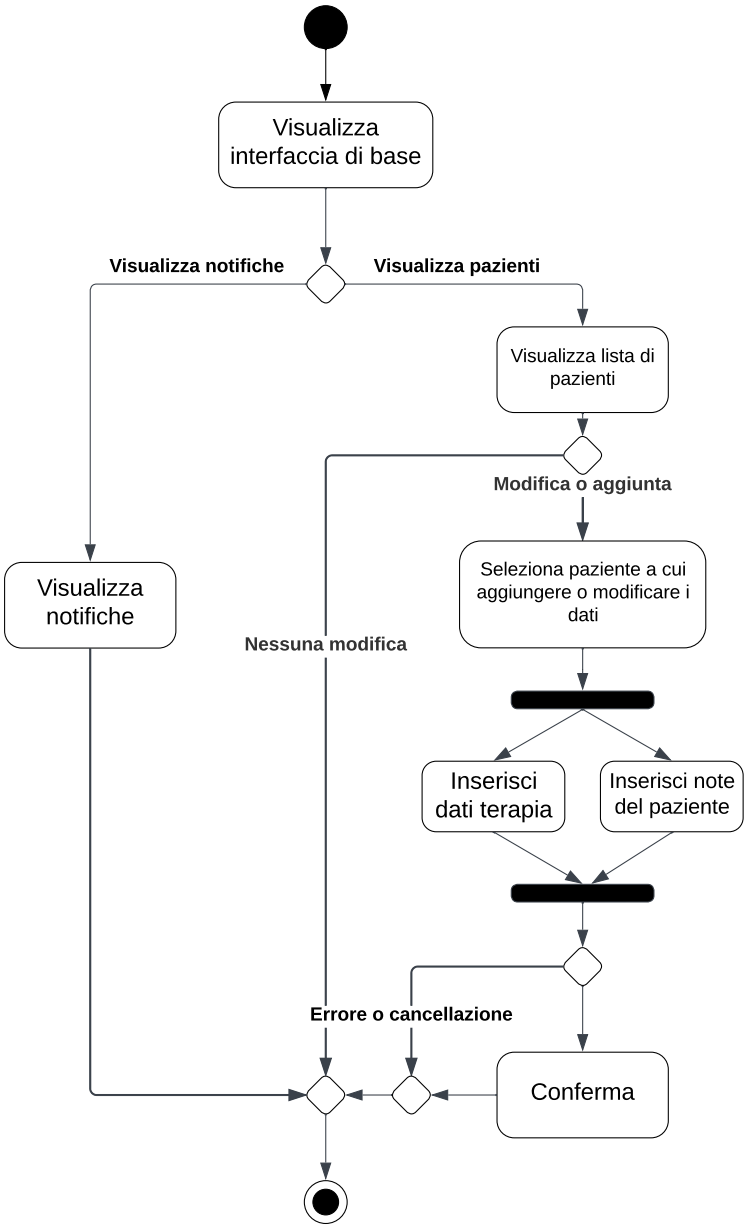

Figura 7: Activity diagram del medico

1.7.4 Attività dell'amministratore

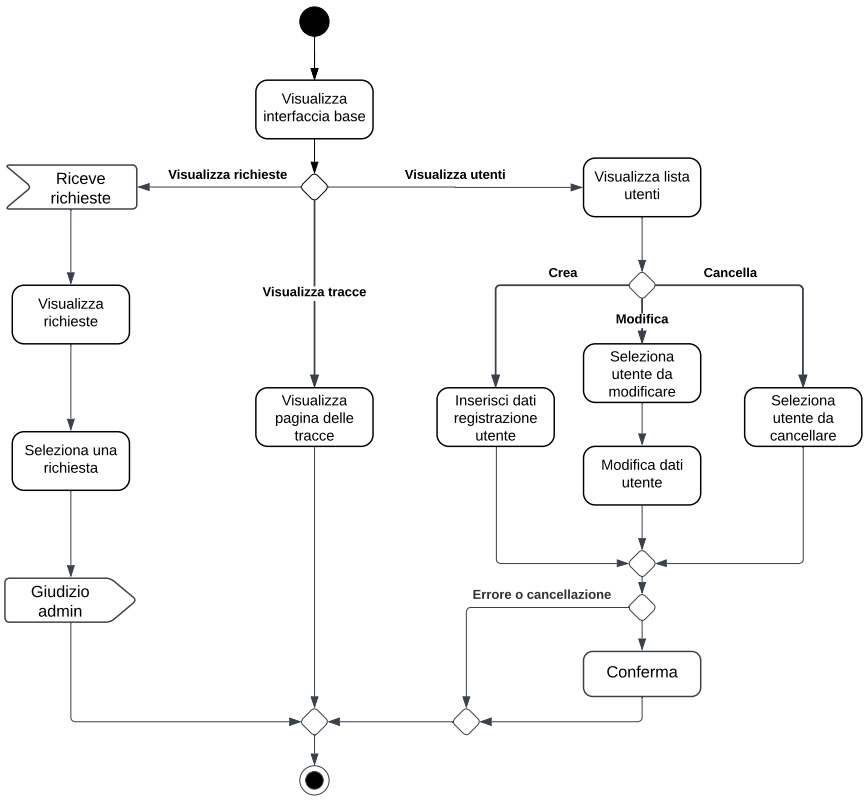

Figura 8: Activity diagram dell'amministratore

2 Sviluppo

2.1 Processo di sviluppo

Il processo di sviluppo del software è stato realizzato seguendo il modello *Agile* con metodologia *Scrum*. Il team è composto da due sviluppatori, che hanno lavorato in parallelo su diverse funzionalità del software. Lo Scrum ci ha permesso di organizzare il lavoro in sprint precisi, durante i quali abbiamo sviluppato le funzionalità richieste, testandole e integrandole nel software. Gli sprint venivano pianificati in base alle funzionalità da implementare e alle priorità stabilite all'inizio, facendo una scrematura delle funzionalità da implementare in base al tempo a disposizione. Alla fine di ogni sprint, abbiamo effettuato una revisione del lavoro svolto, testando le funzionalità implementate e correggendo eventuali bug. *Product Backlog* e *Sprint Backlog* non sono nient'altro che una lista di "to-do" task che bisogna affrontare per la scelta delle funzionalità da implementare e su come implementarle rispettivamente. Tuttavia non vi è un *Scrum Master*

in quanto il team è composto da due persone e non vi è la necessità di avere una figura che coordini il lavoro.

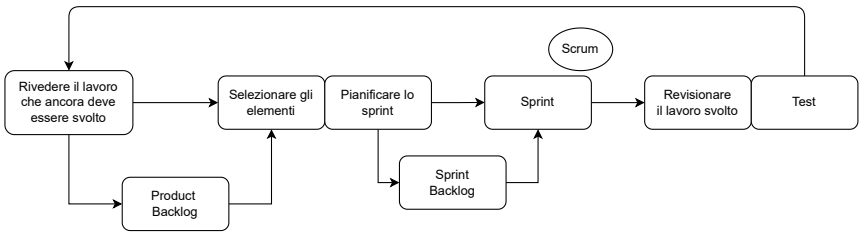

Figura 9: Diagramma del processo di sviluppo

Prima di cominciare il ciclo di Scrum però, è stata effettuata una fase di *analisi dei requisiti*, dove tutto il team di Scrum si è unito per analizzare le specifiche del progetto e poi sono stati definiti i casi d'uso (vedi figura 1). Man mano che venivano implementate le classi e le funzionalità, venivano costruiti gli UML per rappresentare le classi e le relazioni tra di esse. Per la gestione del codice sorgente, è stato utilizzato *Git* come sistema di versionamento, con un repository su *GitHub* per facilitare la collaborazione tra i membri del team. Abbiamo utilizzato anche *Github Planner* per tenere traccia delle attività da svolgere e dello stato di avanzamento del progetto.

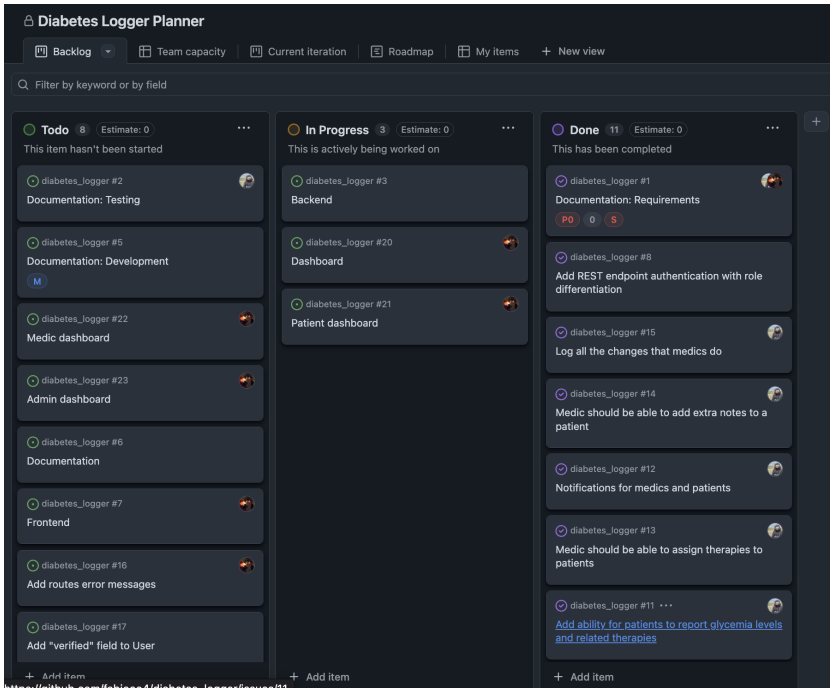

Figura 10: Github Planner del progetto

2.2 Progettazione e pattern architetturali usati

Il software è stato progettato seguendo il pattern architetturale *Model-View-Controller* (MVC), che separa la logica di business, la presentazione e la gestione degli eventi. Questo tipo di pattern ci ha permesso di mantenere il codice ben organizzato e facilmente manutenibile. Abbiamo scelto questo pattern poiché *Spring Boot* (framework di Java utilizzato per il backend) ha gli strumenti necessari per implementare questo pattern in modo semplice e veloce:

- **Model**: rappresenta la logica di business e i dati del software. In questo caso, il model è composto da classi che rappresentano lo user, i pazienti, i medici, le rilevazioni, le notifiche e il change log.
- **View**: rappresenta la parte di presentazione del software. In questo caso, la view è composta da pagine costruite con SvelteKit, un framework per fare siti web.
- **Controller**: rappresenta la parte di gestione degli eventi e della logica di business. Risponde alle chiamate e alle sollecitazioni della View e gestisce le richieste degli utenti.

Tramite richieste HTTP, il controllera comunica con il model e la view. Quindi il controller riceve la richiesta dalla view e poi interagisce con il model per ottenere i dati necessari oppure modificarli in base alla richiesta dell'utente. Nelle prossime sezioni verrà analizzato in dettaglio il modello Model poiché contiene le classi che rappresentano gli attori del sistema e le loro interazioni.

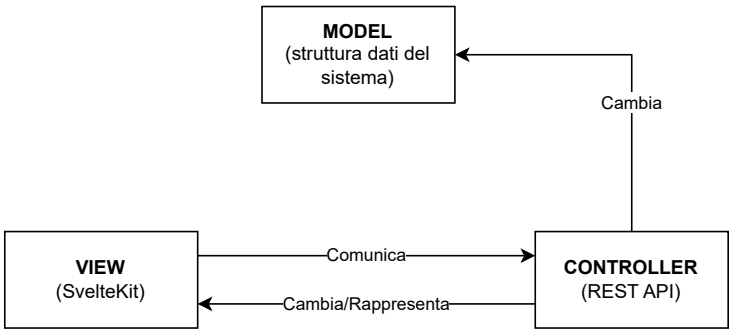

Figura 11: Architettura MVC del software

Seguono i diagrammi UML delle classi del Model e del Controllore, che rappresentano le entità principali del software e le loro relazioni.

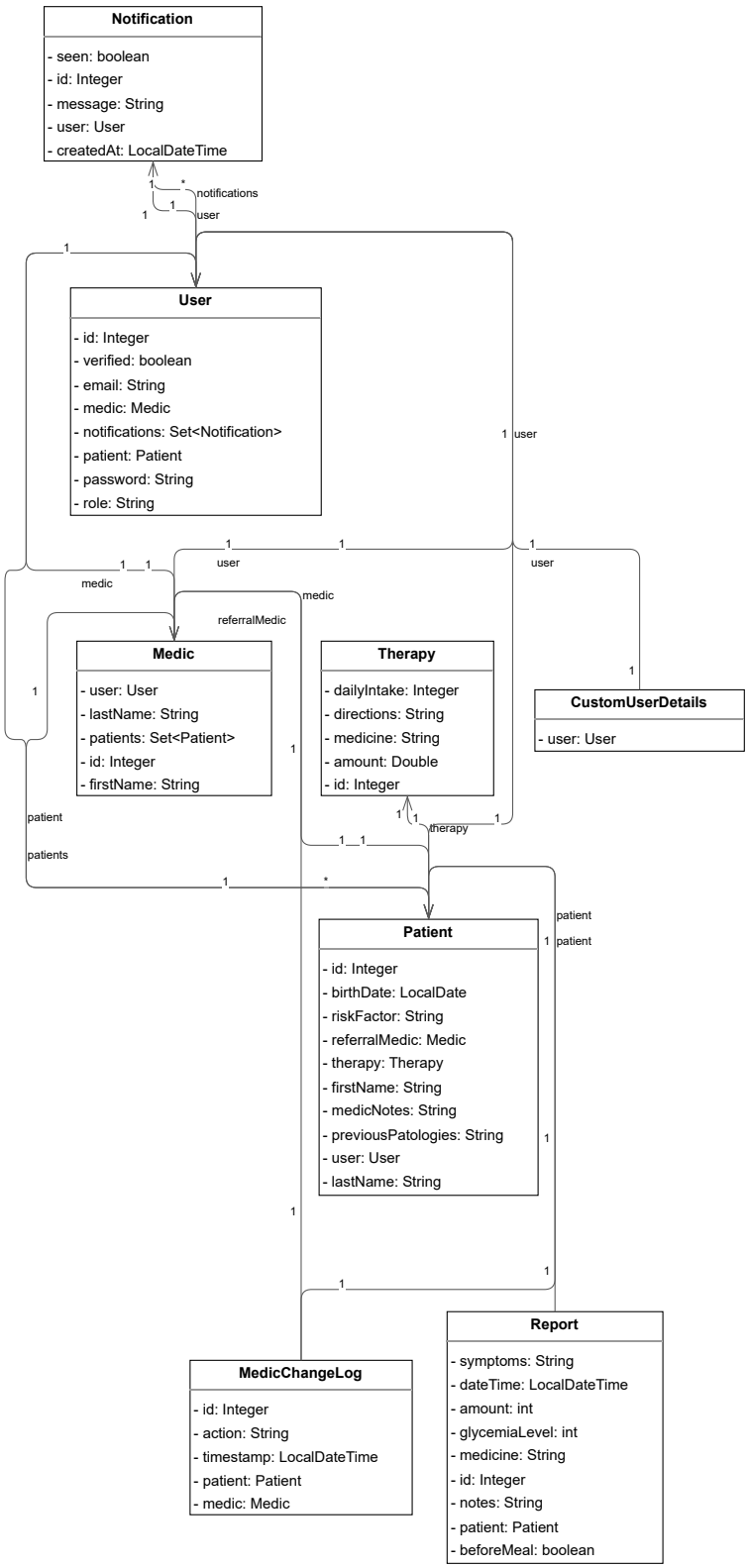

Figura 12: Diagramma UML delle classi del Model

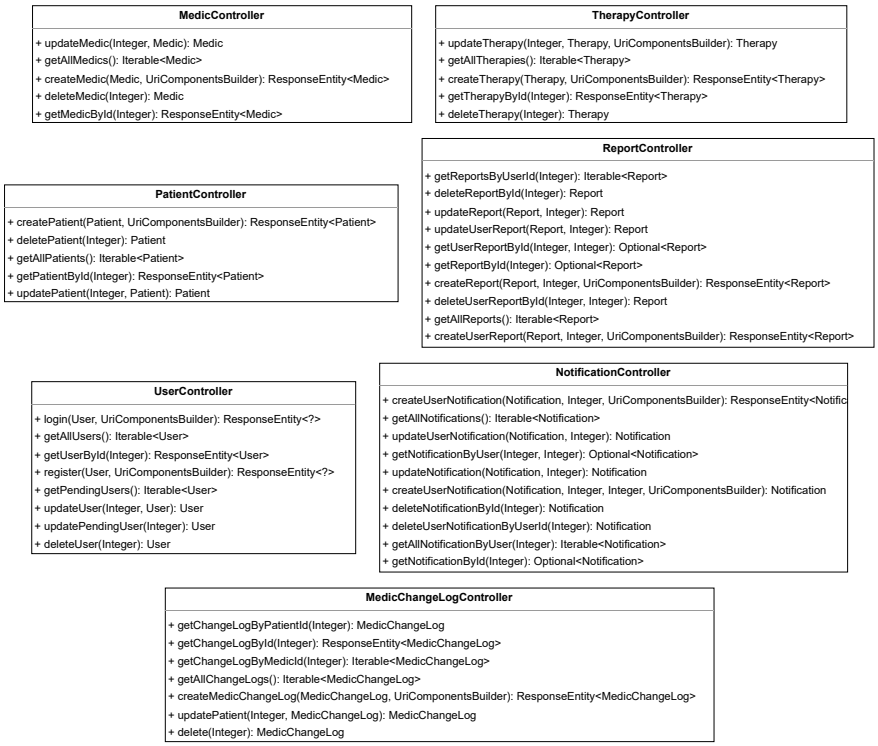

Figura 13: Classi UML dei Controller

2.3 Comunicazione MVC e gestione delle richieste

Per gestire la comunicazione nel modello MVC sono state effettuate le seguenti scelte:

- Per la comunicazione tra il Model e il Controller, sono stati utilizzati i *Repository* di Spring Boot, che permettono di interagire con il database in modo semplice e veloce. Dopodiché abbiamo implementato i *Service* (Service Layer Pattern) che contengono la logica di business e le operazioni da effettuare sui dati. Abbiamo creato un interfaccia *CrudService* che contiene i metodi CRUD (Create, Read, Update, Delete) che ogni servizio implementa.

- Per la comunicazione tra il Controller e la View, sono stati utilizzati i *REST Controller* di Spring Boot, che permettono di gestire le richieste HTTP e di restituire le risposte in formato JSON.

- Per la comunicazione tra il Model e la View in realtà, non c'è una comunicazione diretta, ma il Controller si occupa di prendere i dati dal Model e passarli alla View tramite le richieste HTTP. Questo ci permette di poter decidere quando vogliamo di cambiare modalità di View senza dover modificare il Model.

2.4 Implementazione e design dei pattern usati

2.4.1 Generalità

Ora andremo a vedere quali sono i pattern implementati nel software e come sono stati utilizzati.

- I **pattern iterator** sono stati utilizzati per iterare sulle liste di tutti i nostri modelli, proprietà intrinseca di Java.
- È stato utilizzato molteplici volte il **pattern Observer** grazie alle dipendenze fornite da Spring Boot tramite Bean (In Spring Boot, i Bean sono oggetti gestiti dal container di Spring, questo ci permette di usare *Dependency Injection* per iniettare le dipendenze nei nostri componenti, Gestione del ciclo della vita e una configurazione centralizzata) chiamati *@ManyToOne* e *@OneToMany*: quindi quando un oggetto viene modificato, tutti gli oggetti che dipendono da esso vengono notificati e aggiornati automaticamente.
- Per gestire l'autenticazione e la sicurezza degli endpoints, è stato utilizzato il **pattern Security** di Spring Boot, che permette di gestire l'autenticazione e l'autorizzazione degli utenti. Tuttavia, in concomitanza abbiamo dovuto usare il **pattern Adapter** per adattare le nostre classi di autenticazione a quelle di Spring Boot e alle nostre esigenze (più approfondimenti sul pattern adapter nella sezione *Autenticazione e autorizzazione*).
- Il **pattern Factory Method** è stato utilizzato per comporre le classi del Model come le repository e tutto il service layer (che implementano l'interfaccia *CrudService*) e implementano i loro metodi in base alle esigenze del progetto.
- A sua volta, all'interno della configurazione per la sicurezza, è stato utilizzato il **Filter Chain Pattern** dove una serie di filtri (autenticazione, autorizzazione, CORS, ecc.) processano la richiesta in sequenza.

2.4.2 Autenticazione e autorizzazione

Per gestire l'autenticazione e l'autorizzazione degli utenti, è stato utilizzata la libreria *Spring Security*, che permette di gestire l'autenticazione e l'autorizzazione degli utenti.

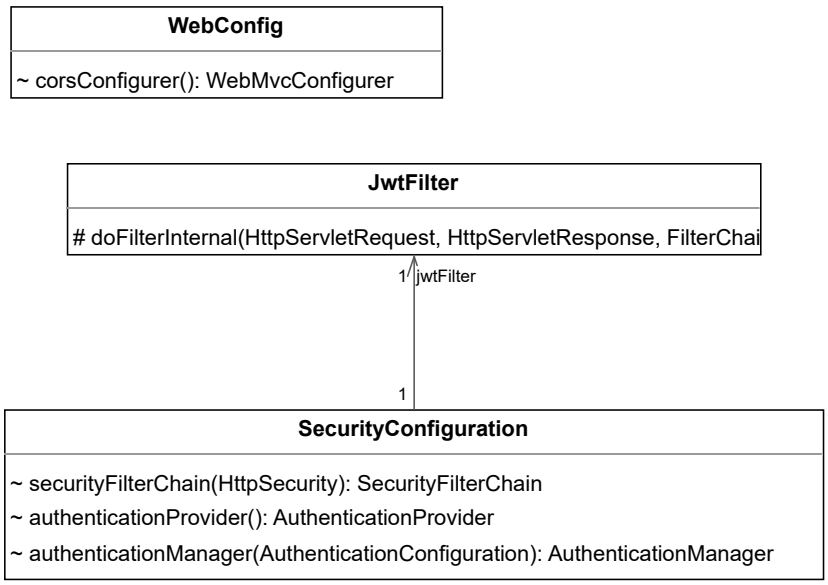

Figura 14: Classi UML della configurazione della sicurezza

Di seguito sono riportate le classi principali che compongono la configurazione della sicurezza:

- La classe **WebConfig** si occupa di configurare le impostazioni di sicurezza del web, come i CORS e le risorse statiche. Nel nostro caso ci permette di effettuare richieste dallo stesso IP del server, ma su porte diverse (per esempio, il frontend).
- La classe **JwtFilter** si occupa di filtrare le richieste HTTP e di verificare se l'utente è autenticato tramite un token JWT. I token JWT (JSON Web Token) sono un modo per rappresentare le informazioni di autenticazione in modo sicuro e compatto. In poche parole estraiamo il token JWT dalla richiesta HTTP e lo verifichiamo per vedere se è valido.
- La classe **SecurityConfiguration** è stato implementato usando il **builder Pattern** e si occupa di configurare la sicurezza del progetto, infatti possiamo definire una serie di filtri che processano la richiesta in sequenza. Quindi, nella funzione *securityFilterChain* definiamo i filtri che vogliamo utilizzare, come il filtro di autenticazione e il filtro di autorizzazione. Questo ci permette di definire i permessi di ogni ruolo, quindi per esempio, i pazienti possono accedere solo alle loro informazioni, i medici possono accedere alle informazioni dei pazienti e l'amministratore può accedere a tutte le informazioni (vedere 1).

Stiliamo ora un breve elenco degli endpoints a cui possono accedere i vari ruoli:

- Admin:

  - **GET+POST+PUT+DELETE/****

- Pazienti:
  - UsersController:
    - **GET+PATCH /users/loggedId**
  - ReportsController:
    - **GET+POST+PUT+DELETE /reports/loggedId**
  - NotificationsController:
    - **GET+DELETE /notifications/loggedId**

- Medici:
  - UsersController:
    - **GET+PATCH /users/loggedId**
  - ReportsController:
    - **GET /reports/anyUserId**
    - **GET /reports/anyPatientId**
    - **GET /reports/**
  - NotificationsController:
    - **GET+DELETE /notifications/loggedId**
  - TherapiesController:
    - **GET+POST+PUT+DELETE /therapies**
    - **GET /therapies/userId**
  - PatientsController:
    - **GET /patients**
    - **GET+PATCH /patients/userId**

2.4.3 Model: repository e service layer

Il nostro Model è composto da diverse classi che rappresentano gli attori del sistema e le loro interazioni. Esso ricopre un ruolo essenziale perché è il modello che definisce e manipola i dati del software. Ovviamente ognuno di questi sottomodelli sono contenute dentro **package** per garantire una buona organizzazione del codice e una facile navigazione, insieme ad un controllo degli errori più semplice. Nella nostra codebase, il Model è composto dalle seguenti classi:

- **Model**: è il package principale che contiene tutte le classi del Model. Quest'ultima contiene tutte le definizioni delle entità del sistema, come *User*, *Patient*, *Medic*, *MedicChangeLog*, *Report*, *Notification* e *Therapy*.

- **Repository**: è il package che contiene le interfacce dei repository, che sono utilizzate per interagire con il database. Abbiamo utilizzato JPA (Jakarta Persistance API) Repository di Spring Boot, che ci ha permesso di immagazinare e ottenere i dati da un database relazionale. Le interfacce dei repository estendono *JpaRepository* e forniscono i metodi per effettuare le operazioni CRUD sui dati.

- **Service**: è il package che contiene le classi dei servizi, che sono utilizzate per gestire la logica di business del software. Infatti, le classi dei servizi implementano l'interfaccia *CrudService* che contiene i metodi CRUD che ogni servizio deve implementare. I servizi sono utilizzati dai controller per gestire le richieste degli utenti e interagire con i repository.

Il controller (vedi figura 13) come già stato detto, utilizza il Model per gestire le richieste degli utenti e interagire con i dati e passa i dati alla View tramite le richieste HTTP grazie all'architettura REST. Seguono sequence diagram di alcune interazioni e dinamiche importanti del software:

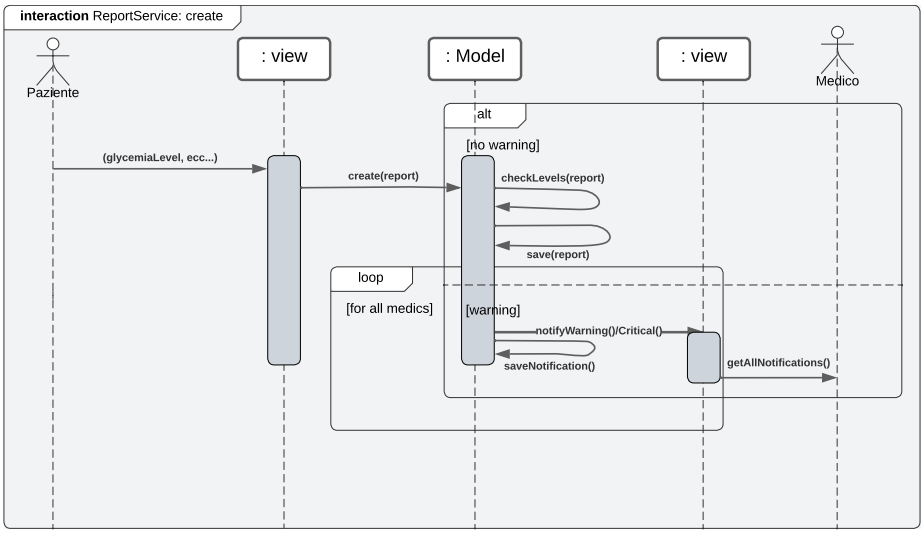

Figura 15: Sequence Diagram della creazione di un report + makeNotification

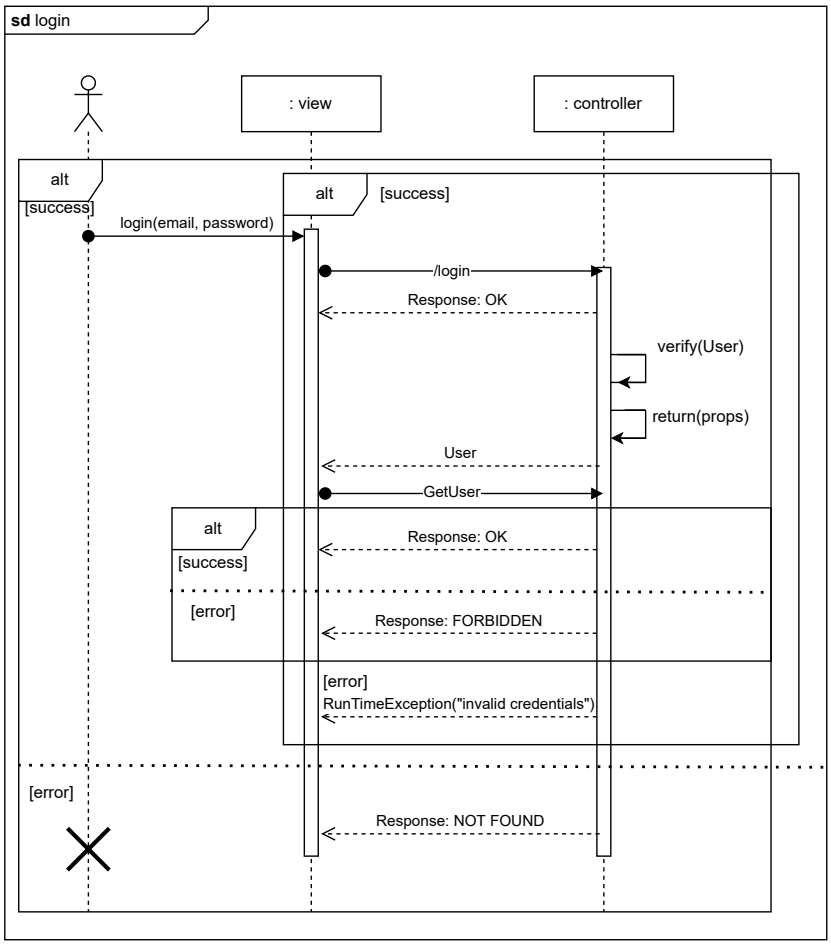

Figura 16: Sequence Diagram di un login e della creazione di un token JWT (props)

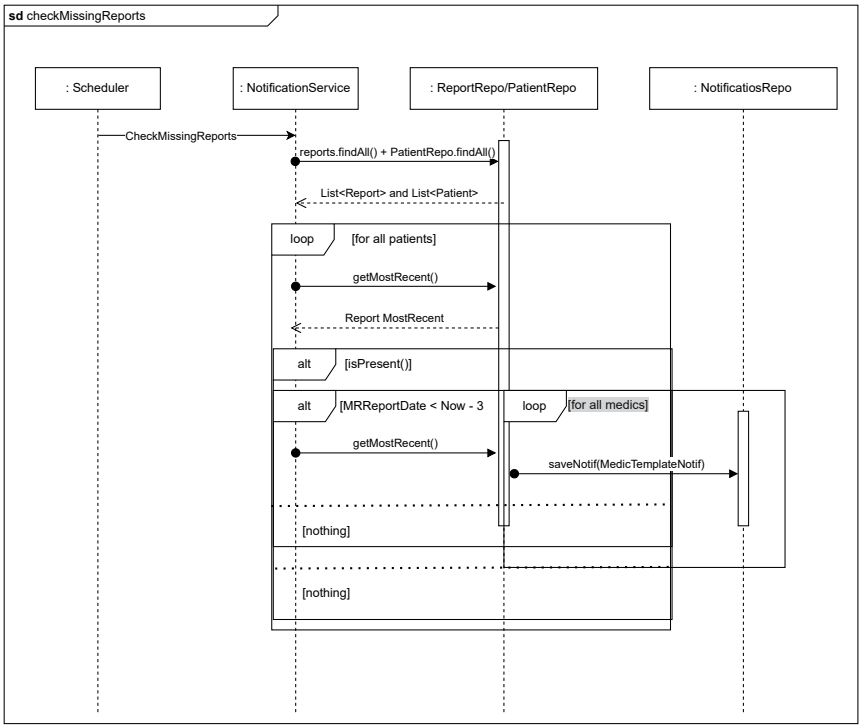

Figura 17: Sequence Diagram del controllo delle rilevazioni mancanti

3 Test e validazione

Per testare il software sono stati utilizzati diversi tipi di test, tra cui:

- Revisione del codice: il codice è stato revisionato da entrambi i membri del team per garantire che fosse scritto in modo che rispettasse le specifiche del progetto e che fosse facilmente comprensibile.
- Verifica della consistenza: sono stati effettuati test per verificare che il software fosse coerente con le specifiche del progetto e che funzionasse come previsto, tramite i Bean @WebMvcTest di Spring Boot che permettono di testare i controller e le loro interazioni con il Model. Questo tipo di test è automatizzato grazie all'API di JUnit.
- È stato usato @DataJpaTest per testare le interazioni con il database e verificare che le operazioni CRUD funzionassero correttamente.
- È stato fatto gran uso di Postman per testare le API REST del software, verificando che le richieste HTTP restituissero i risultati attesi.
- Il software è stato testato dagli sviluppatori
- Il software è stato testato da un utente generico

3.1 Revisione del codice

La fase di revisione del codice è stata effettuata da entrambi i membri del team, che hanno esaminato il codice per garantire che fosse scritto in modo chiaro e comprensibile e che soprattutto rispettasse le specifiche del progetto. Si è più volte controllato che gli use case, activity diagram, design pattern e sequence diagram fossero rispettati e che il codice fosse ben strutturato e organizzato. È stato rivisto il codice anche per cercare di trovare eventuali bug o errori di logica, in modo da correggerli prima di procedere con i test veri e propri. Abbiamo dato più attenzione specialmente agli use case del paziente e del medico (insieme a quello che può fare solo l'amministratore), in quanto sono le funzionalità principali del software.

3.2 Test con @WebMvcTest e @DataJpaTest

Per testare il software sono stati utilizzati i Bean di Spring Boot @WebMvcTest e @DataJpaTest, che permettono di testare i controller e le interazioni con il database. Quello che segue è un esempio di test per il repository del paziente, che verifica le operazioni CRUD (Create, Read, Update, Delete). Le funzioni con il tag @Test sono i test veri e propri e ad ogni test è stato assegnato un ordine tramite l'annotazione @Order. Effettuiamo l'azione e verifichiamo che il risultato sia quello atteso, utilizzando le asserzioni di JUnit.

```
@DataJpaTest
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class PatientRepositoryTest {
  @Autowired
  private PatientRepository patientRepository;
```

```java
@Test
@Order(1)
@Rollback(value = false)
public void createPatientTest() {

  // Action
  Patient patient = new Patient(new User("testmail", "pass", Role.PATIENT,true),
          "TestFirstName", "TestLastName",
      LocalDate.of(2000, 1, 1), new Medic(new User("medicmail",
          "pass", Role.MEDIC, true), "TestMedic", "lastname"));
  patientRepository.save(patient);

  // Verify
  System.out.println(patient);
  assertThat(patient.getId()).isGreaterThan(0);
}

@Test
@Order(2)
public void getPatientByIdTest() {
  // Action
  Patient found = patientRepository.findById(1).get();

  // Verify
  System.out.println(found);
  assertThat(found.getId()).isEqualTo(1);
}

@Test
@Order(3)
public void getAllPatientsTest() {
  // Action
  List<Patient> patients = patientRepository.findAll();

  // Verify
  System.out.println(patients);
  assertThat(patients.size()).isGreaterThan(0);
}

@Test
@Order(4)
@Rollback(value = false)
public void updatePatientTest() {
  // Action
  Patient patient = patientRepository.findById(1).get();
  patient.setFirstName("UpdatedFirstName");
  patient.setLastName("UpdatedLastName");
  Patient updated = patientRepository.save(patient);
```

```
    // Verify
    System.out.println(updated);
    assertThat(updated.getFirstName()).isEqualTo("UpdatedFirstName");
    assertThat(updated.getLastName()).isEqualTo("UpdatedLastName");
  }

  @Test
  @Order(5)
  @Rollback(value = false)
  public void deletePatientTest() {
    // Action
    Patient patient = patientRepository.findById(1).get();
    patientRepository.delete(patient);

    // Verify
    Patient deleted = patientRepository.findById(1).orElse(null);
    assertThat(deleted).isNull();
  }
}
```

Il prossimo codice invece è preso dal test di un controller, più precisamente il controller dei pazienti.

```
@WebMvcTest(PatientController.class)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class PatientControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockitoBean
  private PatientService patientService;

  @Autowired
  private ObjectMapper objectMapper;

  private Patient patient;

  @BeforeEach
  public void setup() {
    patient = new Patient(new User("usermail", "pass1", Role.PATIENT,true),
            "Nome", "Cognome", LocalDate.of(2000, 1, 1),
        new Medic(new User("medicmail", "medicpass", Role.MEDIC, true),
                "FirstName", "LastName"));
    patient.setId(1); // Set a mock ID for testing
  }

  // Post Controller
  @Test
  @Order(1)
  public void createPatientTest() throws Exception {
```

```
  // precondition
  given(patientService.create(any(Patient.class))).willReturn(patient);

  // action
  ResultActions response = mockMvc.perform(post("/patients")
      .contentType(MediaType.APPLICATION_JSON)
      .content(objectMapper.writeValueAsString(patient)));

  // verify
  response.andDo(print()).andExpect(status().isCreated())
      .andExpect(jsonPath("$.firstName",
          is(patient.getFirstName())))
      .andExpect(jsonPath("$.lastName",
          is(patient.getLastName())))
      .andExpect(jsonPath("$.birthDate",
          is(patient.getBirthDate())))
      .andExpect(jsonPath("$.referralMedic.id",
      is(patient.getReferralMedic().getId())));
}

// Get Controller
@Test
@Order(2)
public void getAllPatientsTest() throws Exception {
  // precondition
  List<Patient> patientsList = new ArrayList<>();
  patientsList.add(patient);
  patientsList
      .add(
              new Patient(new User("testmail", "testpass",
                      Role.PATIENT,true), "test",
                      "test", LocalDate.of(2000, 1, 1),
              new Medic(new User("testmedicmail", "testpass", Role.MEDIC, true),
                      "testMedic", "lastname")));
  given(patientService.getAll()).willReturn(patientsList);

  // action
  ResultActions response = mockMvc.perform(get("/patients"));

  // verify
  response.andExpect(status().isOk())
      .andDo(print())
      .andExpect(jsonPath("$.size()",
          is(patientsList.size())));
}

@Test
@Order(3)
public void getPatientByIdTest() throws Exception {
  // precondition
```

```
  given(patientService.getById(patient.getId())).willReturn(Optional.of(patient));

  // action
  ResultActions response = mockMvc.perform(get("/patients/{id}", patient.getId()));

  // verify
  response.andExpect(status().isOk())
      .andDo(print())
      .andExpect(jsonPath("$.firstName",
          is(patient.getFirstName())))
      .andExpect(jsonPath("$.lastName",
          is(patient.getLastName())))
      .andExpect(jsonPath("$.birthDate",
          is(patient.getBirthDate())))
      .andExpect(jsonPath("$.referralMedic.id",
      is(patient.getReferralMedic().getId())));
}

// Put Controller
@Test
@Order(4)
public void updatePatientTest() throws Exception {
  // precondition
  patient.setFirstName("UpdatedFirst");
  patient.setLastName("UpdatedLast");
  given(patientService.update(any(Integer.class),
  any(Patient.class))).willReturn(patient);

  // action
  ResultActions response = mockMvc.perform(put("/patients/{id}", patient.getId())
      .contentType(MediaType.APPLICATION_JSON)
      .content(objectMapper.writeValueAsString(patient)));

  // verify
  response.andExpect(status().isOk())
      .andDo(print())
      .andExpect(jsonPath("$.firstName",
          is(patient.getFirstName())))
      .andExpect(jsonPath("$.lastName",
          is(patient.getLastName())))
      .andExpect(jsonPath("$.birthDate",
          is(patient.getBirthDate())))
      .andExpect(jsonPath("$.referralMedic.id",
      is(patient.getReferralMedic().getId())));
}

// Delete Controller
@Test
@Order(5)
public void deletePatientTest() throws Exception {
```

```
        // precondition
        given(patientService.delete(patient.getId())).willReturn(patient);

        // action
        ResultActions response = mockMvc.perform(delete("/patients/{id}", patient.getId()));

        // verify
        response.andExpect(status().isOk())
            .andDo(print())
            .andExpect(jsonPath("$.firstName",
                is(patient.getFirstName())))
            .andExpect(jsonPath("$.lastName",
                is(patient.getLastName())))
            .andExpect(jsonPath("$.birthDate",
                is(patient.getBirthDate())))
            .andExpect(jsonPath("$.referralMedic.id",
            is(patient.getReferralMedic().getId())));
    }

}
```

3.3 Postman

Per **testare le API REST** del software, è stato utilizzato *Postman*, un tool che permette di inviare richieste HTTP e di visualizzare le risposte. Ne è stato fatto un uso intensivo per testare le API del software, poiché per testare le API REST è necessario inviare richieste HTTP e verificare le risposte. Il nostro environment di Postman permetteva di simulare l'autenticazione degli utenti e di verificarne le funzionalità. Quando un utente si autenticava, veniva restituito un token JWT che veniva utilizzato per autenticare le richieste successive. Il token veniva inserito nella cartella principale dell'environment di Postman e veniva utilizzato per autenticare le richieste da parte di tutti i controller.

Ovviamente per testare la correttezza generale del software in maniera veloce seguento come sempre uno stile *Agile*, abbiamo creato un file java (*LoadDatabase.java*) che caricava in automatico i dati di test nel database, in modo da poter testare le funzionalità del software senza dover inserire manualmente i dati. Per ogni controller, sono stati creati dei test che verificano le funzionalità principali di essi, come la creazione, la lettura, l'aggiornamento e la cancellazione dei dati. Postman ha aiutato incredibilmente anche a **testare che gli endpoints fossero protetti** correttamente e che le autorizzazioni funzionassero come previsto.

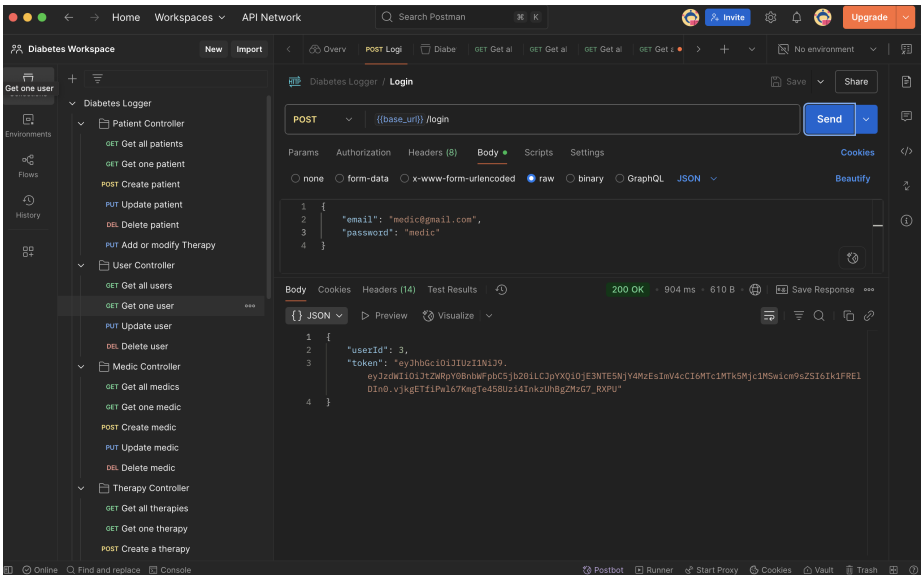

Figura 18: Postman environment del progetto

3.4 Test manuali degli sviluppatori

In questa fase ci siamo preoccupati di testare il software manualmente, verificando che tutte le funzionalità implementate funzionassero come previsto e che non ci fossero bug o errori di logica. Alcuni dei test effettuati sono stati:

- Verifica che il login funzionasse correttamente per tutti gli utenti;
- Verifica che la registrazione funzionasse correttamente per tutti gli utenti;
- Verifica che l'utente potesse modificare i propri dati;
- I campi obbligatori devono essere riempiti correttamente altrimenti la View avvisa di riempirli;
- Alcuni campi non possono essere modificati da un certo tipo di utente questo tipo di sicurezza è stata implementata tramite il pattern Security di Spring Boot e quindi è stato semplice accertarsi che funzionasse;
- Inserimento di un report con dati validi e verifica che venisse creato correttamente;
- Verifica che le notifiche venissero create correttamente quando un report "preoccupante" veniva inserito;
- Verifica che le notifiche venissero cancellate correttamente quando un utente le cancellava;
- Verificato il sistema di avvisi tramite notifiche;
- Verifica dei vari inserimenti con input errati oppure vuoti;

- Verifica di un assegnazione di una terapia ad un paziente;
- Inserimento di terapie, pazienti o medici duplicati
- Verifica che l'amministratore potesse gestire gli utenti e le richieste di registrazione;
- Verifica che l'amministratore potesse visualizzare la traccia dei medici e i medici l'ultima operazione effettuata su un paziente;
- Verificato che il paziente possa vedere solo i suoi report ma non quelli degli altri pazienti;

3.5 Test utente generico

Per testare il software da un utente generico, abbiamo chiesto ad un individuo esterno di provare il software e di darci un feedback sulle funzionalità implementate. Sono state fornite all'utente solo alcune linee guida di come utilizzare il software, senza spiegare nel dettaglio. L'utente ha registrato un nuovo account e ha provato a utilizzare il software come un paziente, provando ad inserire report e visualizzare le notifiche. Lo scopo di questo test era quello di verificare che il software fosse facile da usare e che le funzionalità implementate fossero intuitive. Il feedback ricevuto è stato positivo e allo stesso tempo ci ha dato spunti per migliorare ulteriormente l'usabilità del software.