Documentazione Progetto Ingegneria del Software: GlucoTrack

Nicolas Corini VR486086
Pietro Mori VR487996
Andrea Fanelli VR486083

A.A. 2024/2025

Contents

1 Requisiti e interazioni utente–sistema

1.1 Analisi dei requisiti

1.1.1 Introduzione

Il progetto prevede lo sviluppo di **GlucoTrack**, una piattaforma di telemedicina dedicata al monitoraggio dei pazienti affetti da diabete di tipo 2. L'obiettivo è garantire un controllo costante dei valori glicemici, favorire la comunicazione paziente–medico e migliorare l'aderenza terapeutica tramite funzioni automatiche di analisi e di alert.

1.1.2 Requisiti funzionali

Attore	Descrizione sintetica
Paziente	Persona con diabete di tipo 2 che inserisce dati clinici e interagisce con il proprio diabetologo.
Medico	Diabetologo responsabile della prescrizione e del monitoraggio della terapia.
Admin	Operatore autorizzato che gestisce anagrafiche e permessi degli utenti (*createUser*, *getUsers*, *deleteUser*).
Sistema	Servizio backend *GlucoTrack Monitoring Task* che analizza periodicamente i dati, genera alert e produce report.

Attori principali

Funzionalità per attore

Paziente

- Autenticazione con credenziali pre-caricate (no *self-registration*).
- Inserimento giornaliero delle misurazioni glicemiche:
  - pre-pasto (target: 80–130 mg/dL);
  - 2 h post-pasto (≤ 180 mg/dL).
- Registrazione di sintomi (es. nausea, spossatezza) e di assunzioni di insulina/-farmaci orali (farmaco, dose, data-ora).
- Segnalazione di patologie o terapie concomitanti specificando l'intervallo temporale.
- Invio di richieste al medico via e-mail.

Medico

- Autenticazione.
- Prescrizione e modifica di terapie (farmaco, numero dosi/giorno, dose, indicazioni).
- Analisi dei dati del paziente in forma dettagliata e sintetica (andamento settimanale/mensile delle glicemie).
- Aggiornamento della scheda clinica (fattori di rischio, comorbidità, patologie pregresse).
- Tracciabilità completa di ogni operazione eseguita.

Admin
- Creazione di nuovi utenti: *createUser*.
- Consultazione della lista utenti: *getUsers*.
- Eliminazione di un utente tramite ID: *deleteUser*.

Sistema – GlucoTrack Monitoring Task
1. **Controllo aderenza farmacologica**
   Verifica corrispondenza tra assunzioni registrate e programmazioni generando alert di:
   - mancata assunzione;
   - inadempienza prolungata (> 3 giorni).
2. **Analisi delle glicemie**
   Segnalazione di valori fuori soglia personalizzata o assenza di misurazioni in periodi critici.
3. **Monitoraggio sintomi critici**
   Evidenzia sintomi che richiedono attenzione urgente.
4. **Gestione automatica alert**
   Crea record in ALERT e ALERTRECIPIENTS, notificando pazienti e/o medici via e-mail o push in-app.
5. **Reporting giornaliero**
   Produzione di report suddivisi per tipologia di alert e destinatario.

Vincoli e note operative

- Tutti gli orari sono gestiti in UTC o sincronizzati con il fuso dell'utente.
- I dati iniziali di pazienti e medici sono caricati dal personale **Admin**.
- Ogni paziente ha un medico di riferimento; i medici possono comunque consultare e aggiornare qualsiasi paziente, con log delle attività.
- Le notifiche e-mail verso lo stesso medico sono raggruppate (max 1 al giorno) per evitare spam.
- I messaggi contengono *deep link* (es. `myapp://doctor/non-adherent-patients`) per l'apertura diretta delle schermate nella app.

1.1.3 Conclusione

La suddivisione fra **Paziente**, **Medico**, **Admin** e **Sistema** copre l'intero ciclo di cura, dal data-entry clinico all'automazione degli alert, garantendo controllo, tracciabilità e tempestività d'intervento. GlucoTrack rappresenta quindi uno strumento completo di telemonitoraggio finalizzato a ridurre le complicanze del diabete di tipo 2 e a migliorare l'aderenza terapeutica.

1.2 Casi d'Uso del Sistema

1.2.1 Casi d'Uso Generali

Login

Permette agli utenti (pazienti, medici, admin) di autenticarsi per accedere alle funzionalità dell'app.

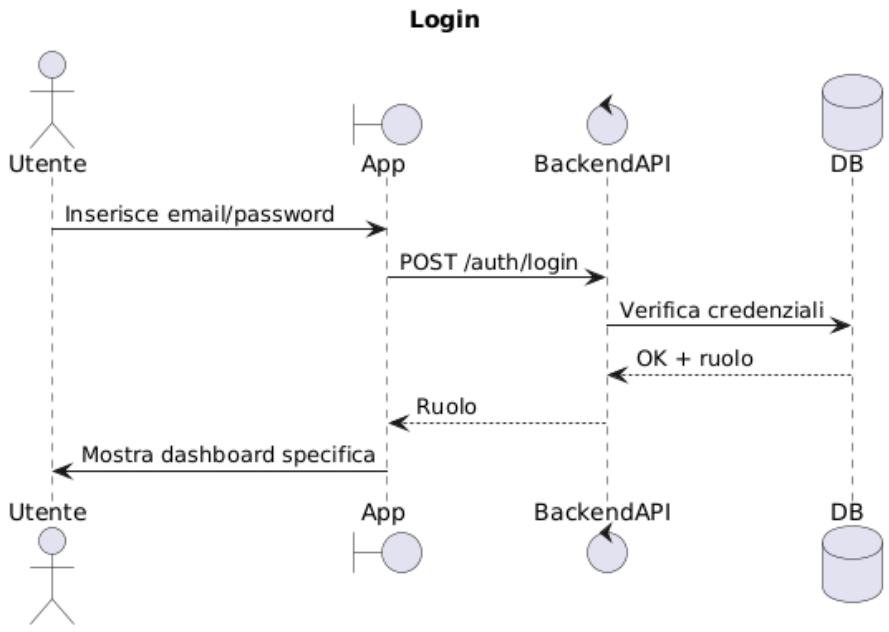
Attori: Utente non autenticato (medico, paziente, admin)
Scopo: Permettere l'accesso al sistema tramite credenziali
Precondizioni: L'utente possiede un account valido
Passi:

1. L'utente apre l'app e visualizza la schermata di login.
2. Inserisce e-mail e password.
3. Preme il pulsante di login.
4. Il sistema invia le credenziali al backend per la verifica.
5. Se le credenziali sono corrette, identifica il ruolo dell'utente.
6. Il sistema reindirizza alla dashboard corrispondente (medico, paziente, admin).

Estensioni:

- Credenziali errate → messaggio di errore.
- Server non disponibile → errore di connessione.

Postcondizioni: L'utente autenticato accede alla propria dashboard.



1.2.2 Casi d'Uso relativi ai Pazienti

Visualizzazione Profilo

Mostra al paziente i propri dati anagrafici e di contatto.

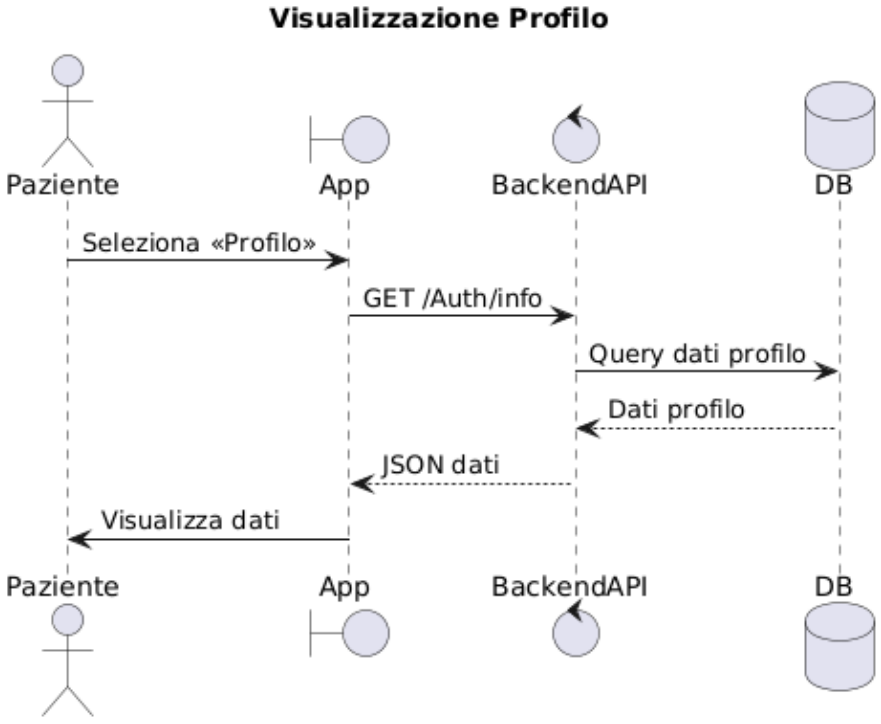
Attori: Paziente autenticato
Scopo: Consultare i dati personali
Precondizioni: Il paziente è autenticato
Passi:

1. Il paziente apre la sezione *Profilo*.
2. Il sistema recupera i dati dal backend.
3. I dati vengono visualizzati.

Estensioni:

- Recupero dati fallito → messaggio di errore.

Postcondizioni: Il paziente visualizza i propri dati.



Gestione Condizioni Segnalate

Consente di aggiungere, modificare o eliminare condizioni cliniche segnalate.

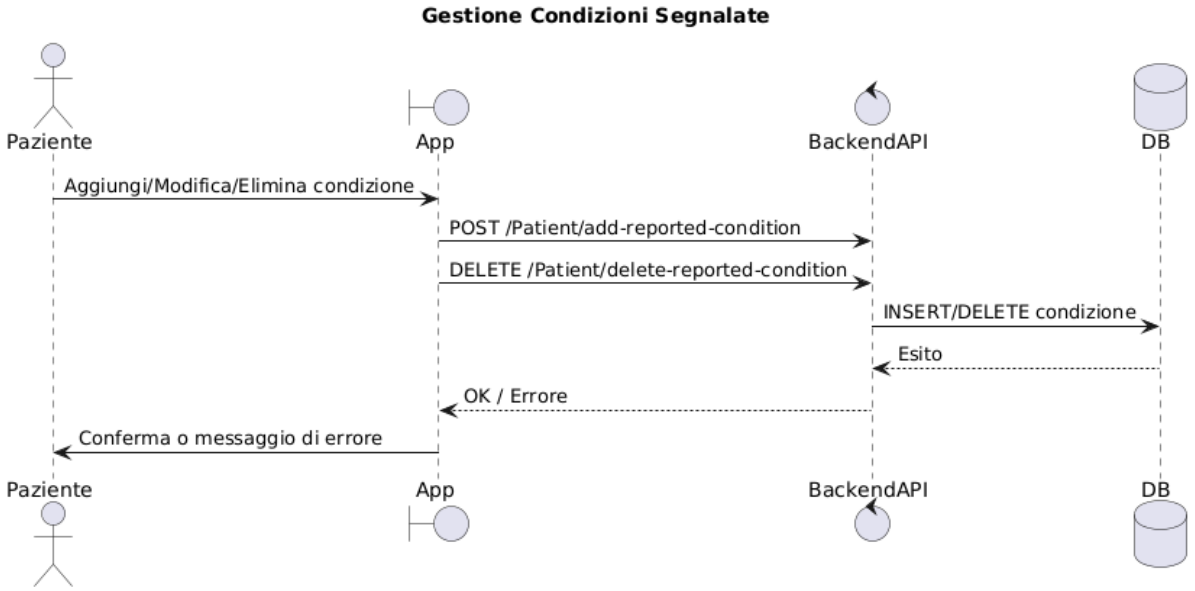
Attori: Paziente autenticato
Scopo: Tenere traccia delle condizioni cliniche
Precondizioni: Il paziente è autenticato
Passi:

1. Il paziente apre la sezione *Condizioni segnalate*.
2. Aggiunge, modifica oppure elimina una condizione.
3. Il sistema aggiorna i dati sul backend.

Estensioni:

- Operazione fallita → messaggio di errore.

Postcondizioni: Le condizioni risultano aggiornate.



Gestione Misurazioni Glicemiche

Consente di aggiungere, visualizzare o eliminare misurazioni glicemiche.

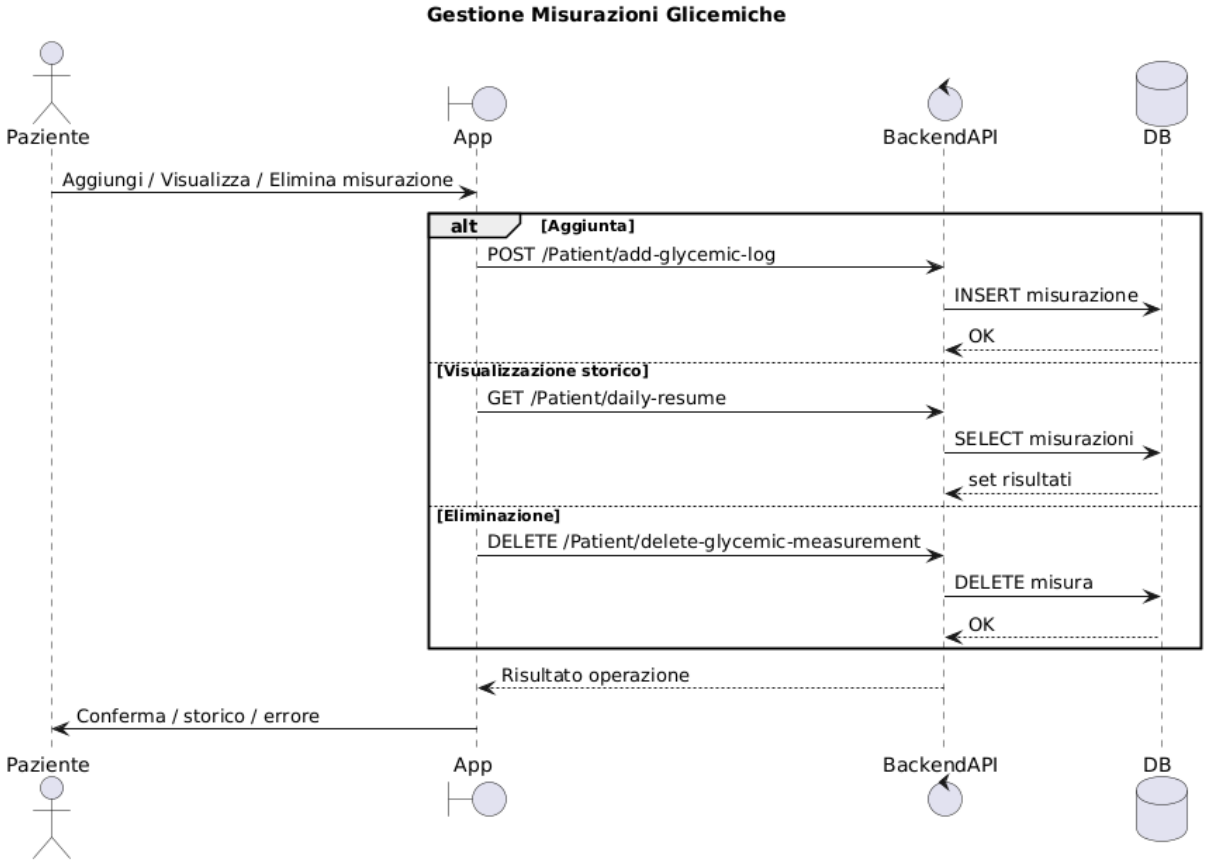
Attori: Paziente autenticato
Scopo: Monitorare i valori glicemici
Precondizioni: Il paziente è autenticato
Passi:

1. Il paziente apre la sezione *Glicemia*.
2. Aggiunge una nuova misurazione, consulta lo storico oppure elimina una misurazione.
3. Il sistema aggiorna i dati sul backend.

Estensioni:

- Operazione fallita → messaggio di errore.

Postcondizioni: Le misurazioni glicemiche sono aggiornate.



Gestione Sintomi

Consente di aggiungere, visualizzare o eliminare sintomi registrati.

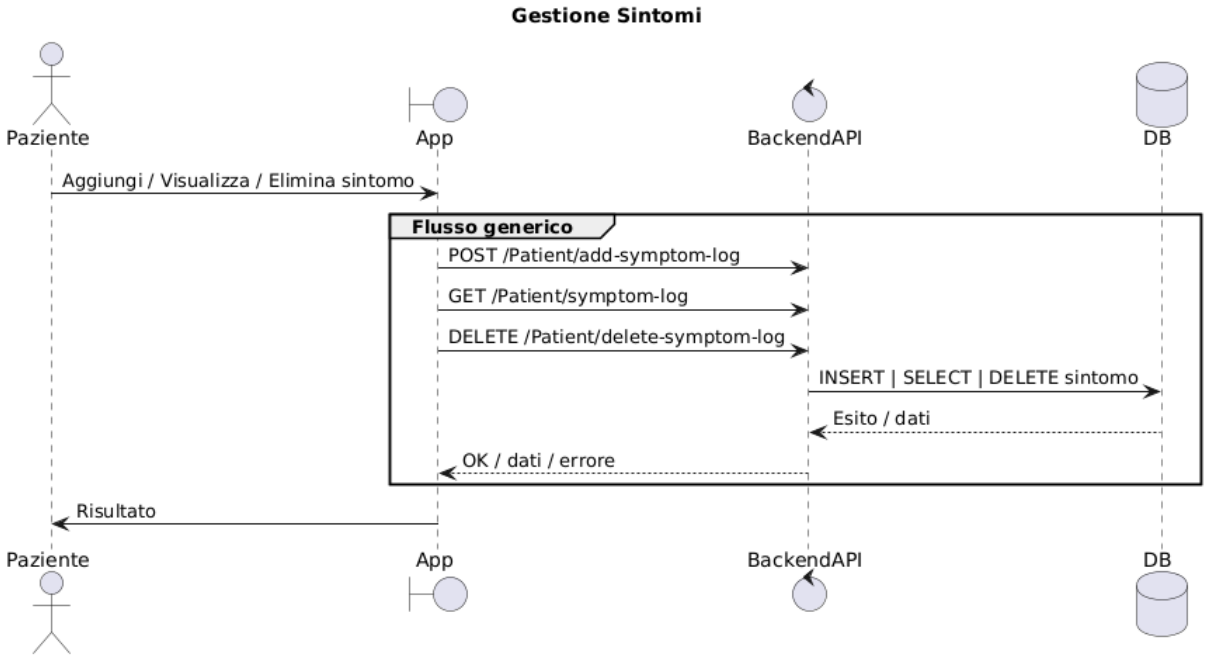
Attori: Paziente autenticato
Scopo: Tenere traccia dei sintomi
Precondizioni: Il paziente è autenticato
Passi:

1. Il paziente apre la sezione *Sintomi*.
2. Aggiunge, visualizza oppure elimina un sintomo.
3. Il sistema aggiorna i dati sul backend.

Estensioni:

- Operazione fallita → messaggio di errore.

Postcondizioni: I sintomi risultano aggiornati.



Gestione Terapie e Assunzione Farmaci

Permette di visualizzare le terapie prescritte e registrare l'assunzione dei farmaci.

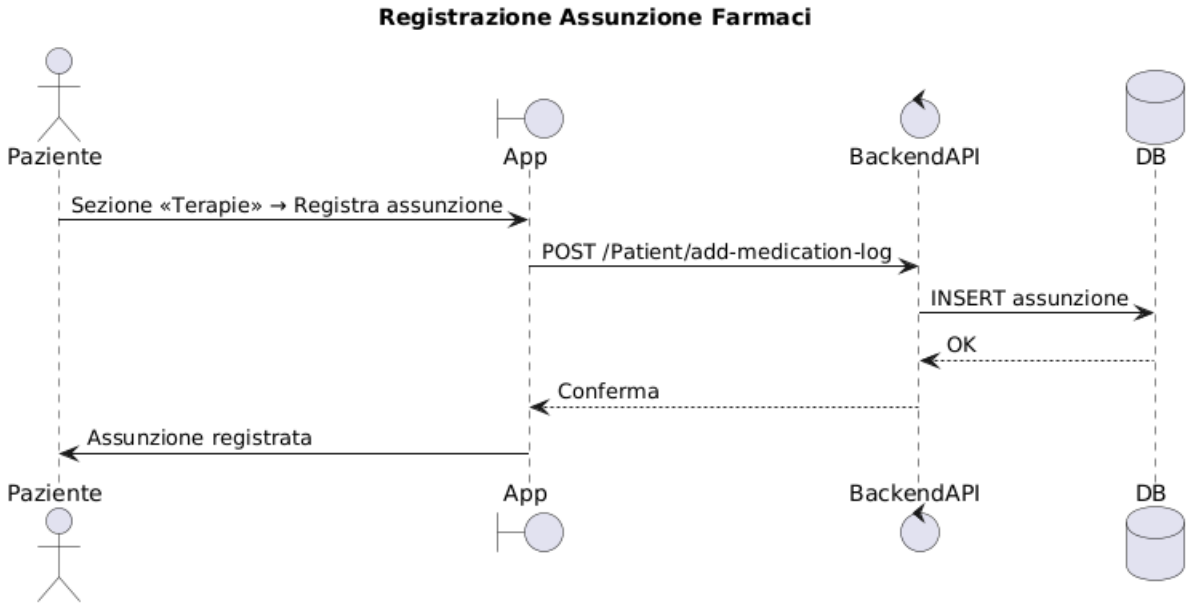
Attori: Paziente autenticato
Scopo: Monitorare l'aderenza terapeutica
Precondizioni: Il paziente è autenticato
Passi:

1. Il paziente apre la sezione *Terapie*.
2. Consulta la lista di terapie e farmaci prescritti.
3. Registra l'assunzione di un farmaco (data-ora, dose).
4. Il sistema aggiorna il backend.

Estensioni:

- Operazione fallita → messaggio di errore.

Postcondizioni: L'assunzione del farmaco è registrata.



Creazione Alert Glicemico Manuale

Consente di segnalare manualmente un valore glicemico fuori soglia al medico.

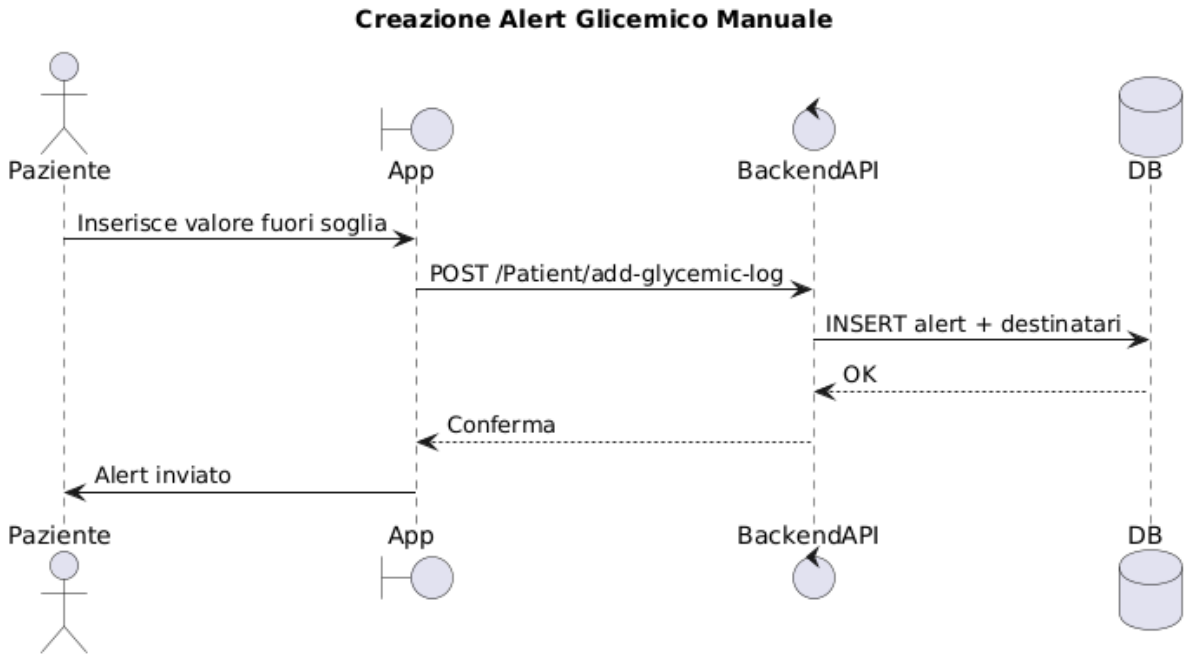
Attori: Paziente autenticato
Scopo: Avvisare il medico di un valore critico
Precondizioni: Il paziente è autenticato
Passi:

1. Il paziente inserisce un valore glicemico fuori soglia.
2. Il sistema invia un alert al backend.

Estensioni:

- Operazione fallita → messaggio di errore.

Postcondizioni: L'alert glicemico è registrato.



1.2.3 Casi d'Uso relativi ai Medici

Visualizzazione Dashboard Medico

Mostra al medico un riepilogo sui propri pazienti, trend glicemici e alert.

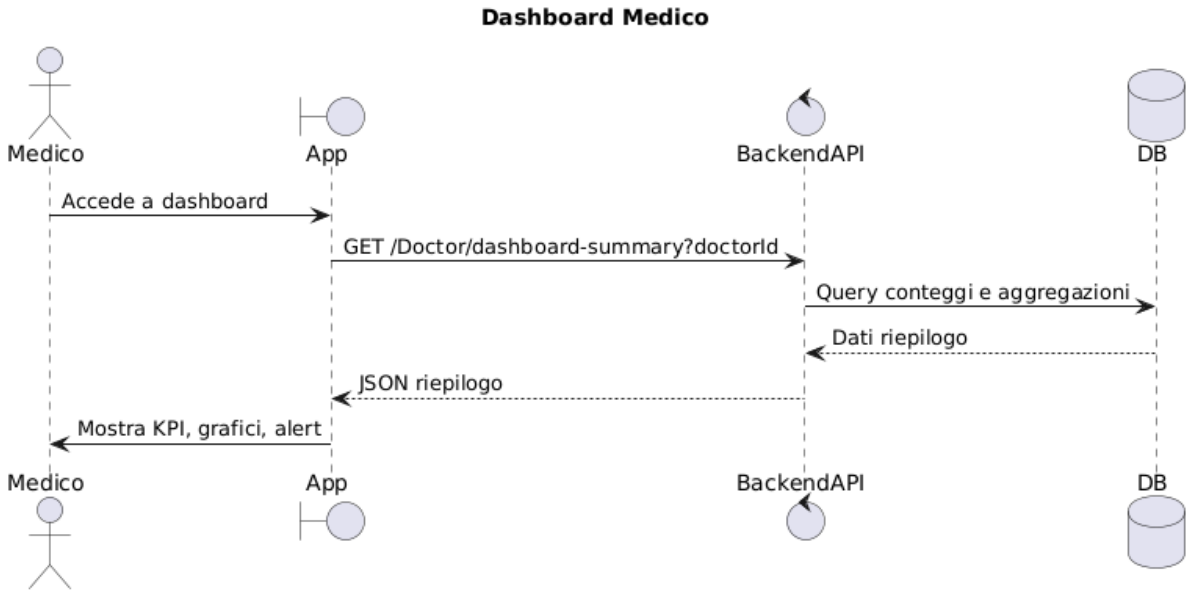
Attori: Medico autenticato
Scopo: Fornire una panoramica dello stato dei pazienti
Precondizioni: Il medico è autenticato
Passi:

1. Il medico accede alla dashboard.
2. Il sistema recupera i dati dal backend.
3. I dati vengono mostrati al medico.

Estensioni:

- Recupero dati fallito → messaggio di errore.

Postcondizioni: Il medico visualizza la dashboard.



Gestione Comorbidità e Fattori di Rischio

Consente di aggiungere, aggiornare o eliminare comorbidità e fattori di rischio di un paziente.

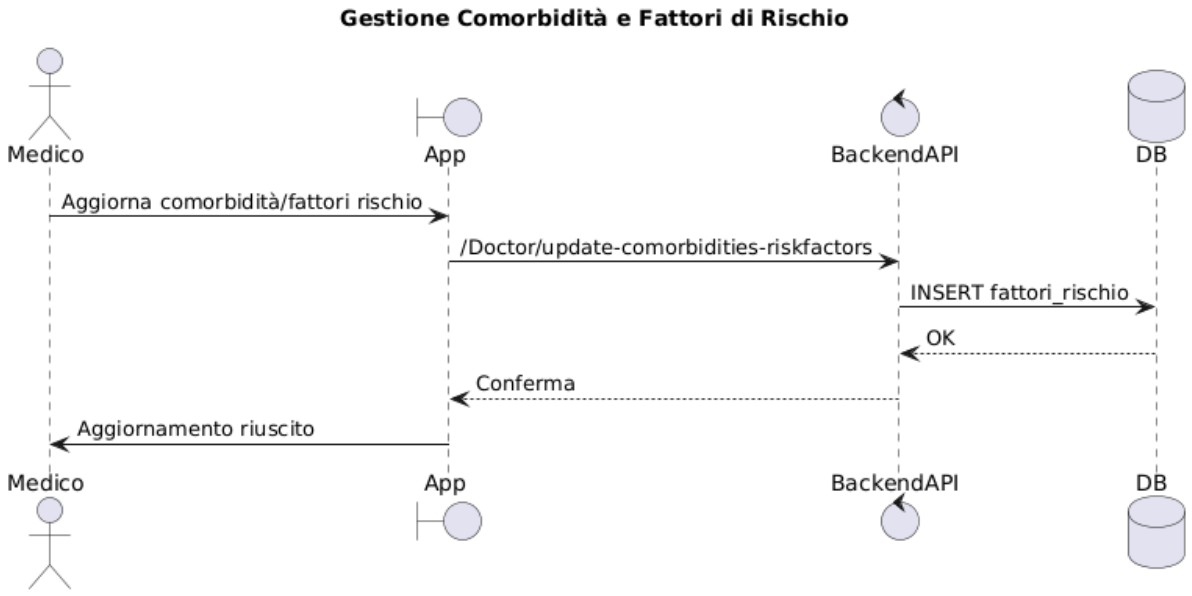
Attori: Medico autenticato
Scopo: Mantenere aggiornato il quadro clinico
Precondizioni: Il medico è autenticato
Passi:

1. Il medico seleziona un paziente.
2. Aggiorna, aggiunge o elimina comorbidità/fattori di rischio.
3. Il sistema aggiorna il backend.

Estensioni:

- Operazione fallita → messaggio di errore.

Postcondizioni: I dati clinici sono aggiornati.



Analisi Paziente

Mostra analisi dettagliate di trend glicemici, aderenza terapeutica, sintomi e alert.

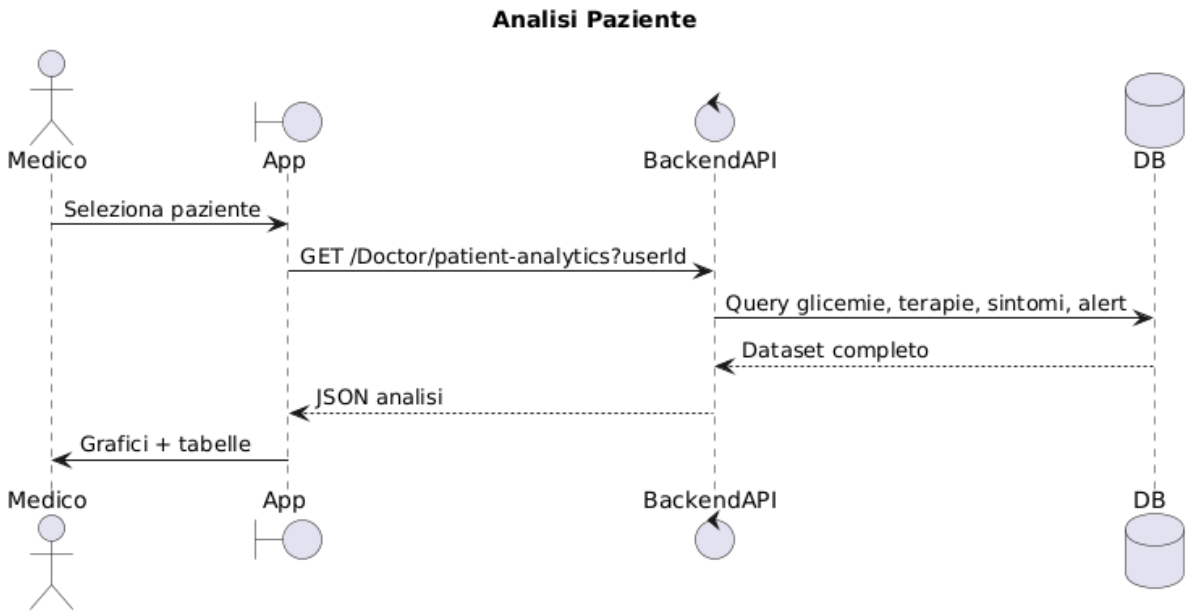
Attori: Medico autenticato
Scopo: Valutare l'andamento clinico di un paziente
Precondizioni: Il medico è autenticato
Passi:

1. Il medico seleziona un paziente.
2. Visualizza analisi dettagliate (grafici, tabelle, alert).

Estensioni:

- Recupero dati fallito → messaggio di errore.

Postcondizioni: Il medico consulta le analisi.



Gestione Terapie

Consente di aggiungere, modificare o eliminare terapie per i pazienti.

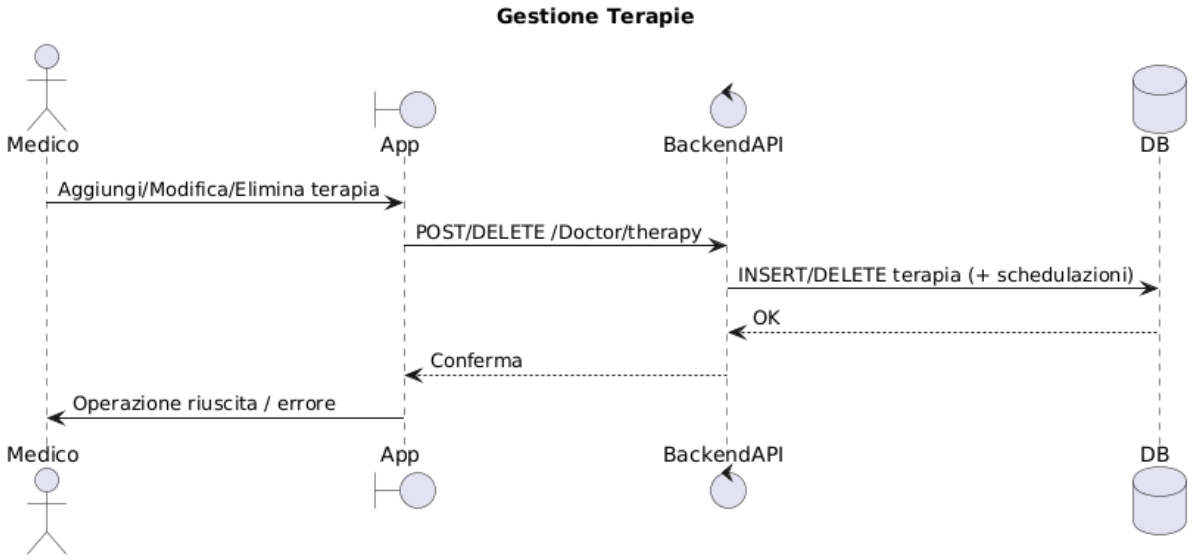
Attori: Medico autenticato
Scopo: Gestire le terapie prescritte
Precondizioni: Il medico è autenticato
Passi:

1. Il medico seleziona un paziente.
2. Aggiunge, modifica o elimina una terapia.
3. Il sistema aggiorna il backend.

Estensioni:

- Operazione fallita → messaggio di errore.

Postcondizioni: Le terapie risultano aggiornate.



Ricerca Pazienti

Permette di cercare pazienti tramite filtri (nome, età, genere, ecc.).

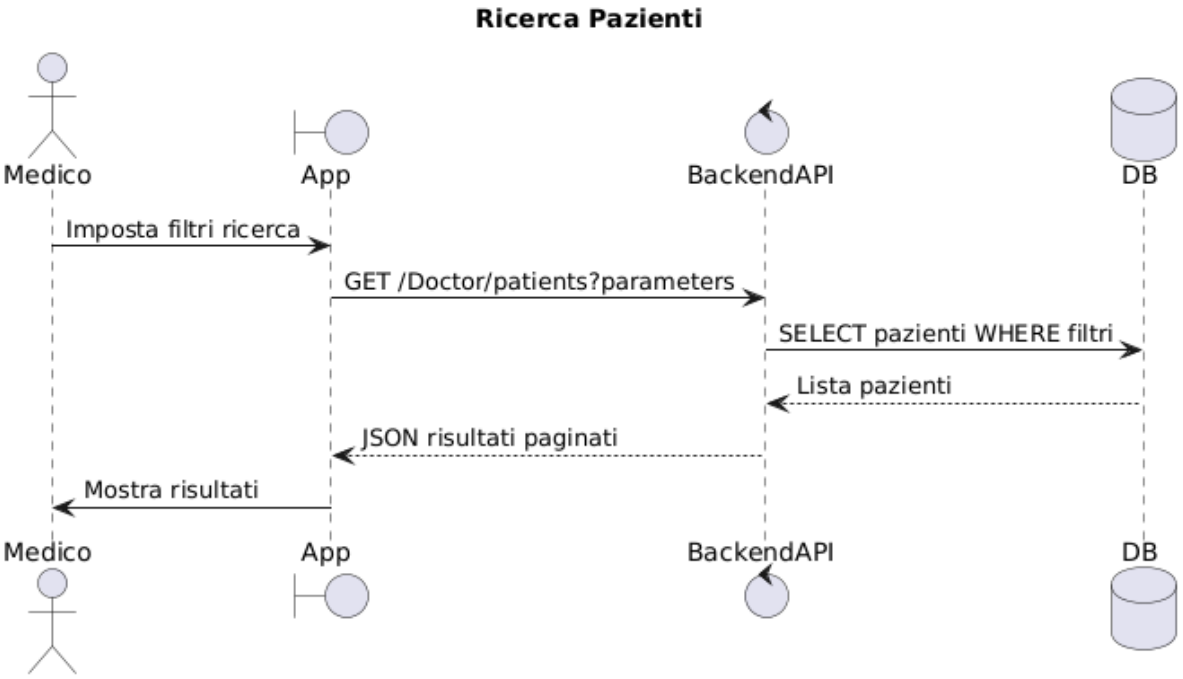
Attori: Medico autenticato
Scopo: Individuare rapidamente i pazienti
Precondizioni: Il medico è autenticato
Passi:

1. Il medico apre la sezione *Ricerca*.
2. Imposta i filtri desiderati.
3. Il sistema mostra i risultati.

Estensioni:

- Ricerca fallita → messaggio di errore.

Postcondizioni: I risultati sono visualizzati.



Gestione Alert

Permette al medico di visualizzare e risolvere alert clinici dei pazienti.

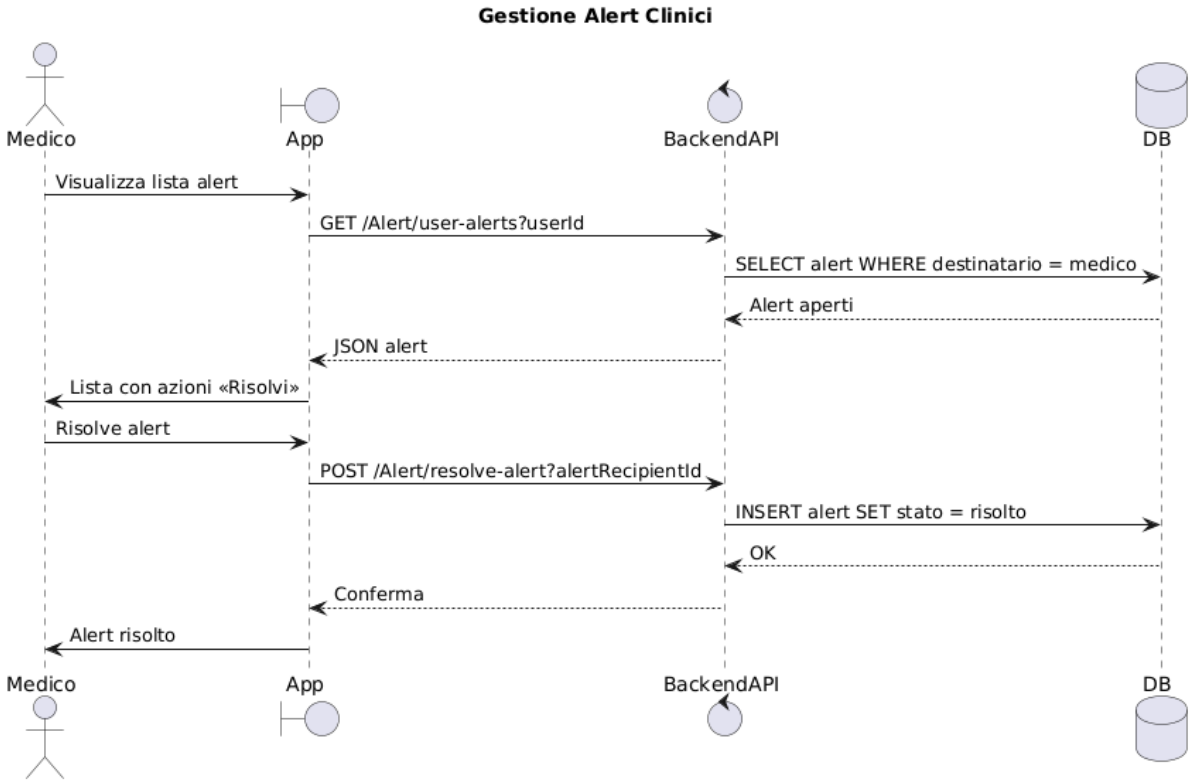
Attori: Medico autenticato
Scopo: Monitorare e gestire situazioni critiche
Precondizioni: Il medico è autenticato
Passi:

1. Il medico apre la lista degli alert.
2. Seleziona un alert e lo risolve.
3. Il sistema aggiorna lo stato dell'alert sul backend.

Estensioni:

- Operazione fallita → messaggio di errore.

Postcondizioni: L'alert è stato gestito o risolto.



1.2.4 Casi d'Uso relativi all'Admin

Creazione Utente

Consente all'admin di creare un nuovo utente (medico, paziente o admin).

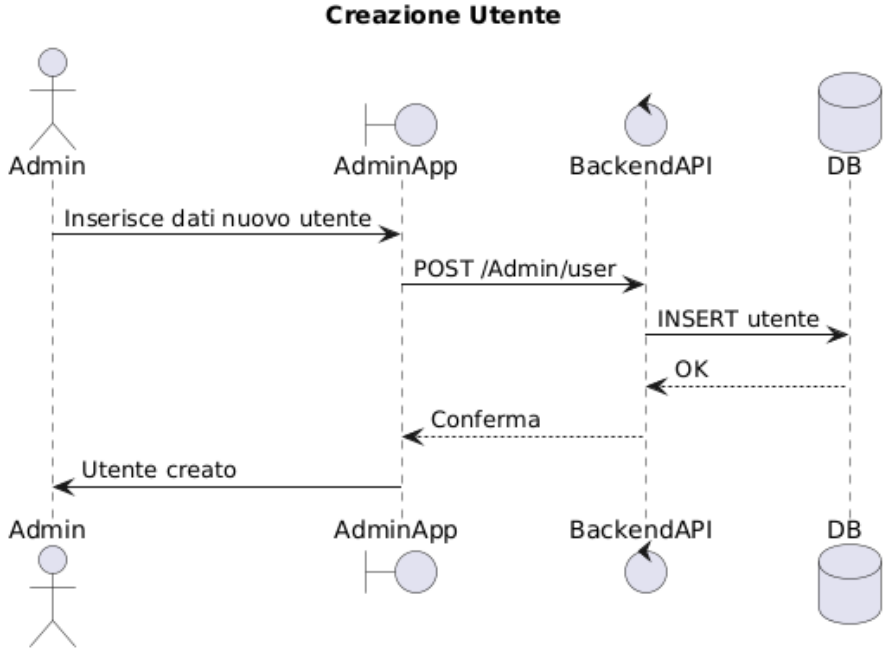
Attori: Admin autenticato
Scopo: Gestire le anagrafiche utenti
Precondizioni: L'admin è autenticato
Passi:

1. L'admin apre la sezione dedicata.
2. Inserisce i dati del nuovo utente.
3. Il sistema invia la richiesta di creazione al backend.

Estensioni:

- Creazione fallita → messaggio di errore.

Postcondizioni: Il nuovo utente è creato.



Visualizzazione Utenti

Mostra all'admin la lista degli utenti registrati.

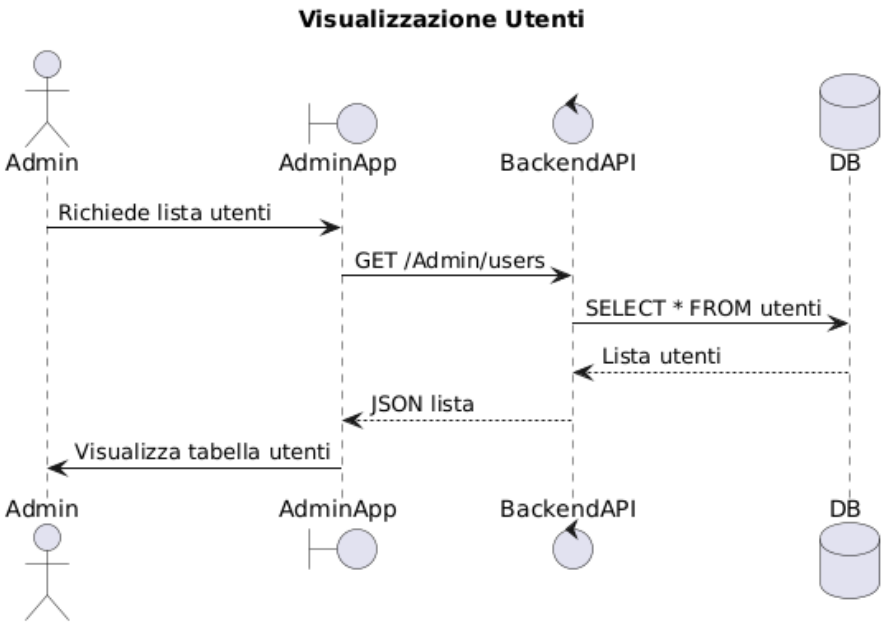
Attori: Admin autenticato
Scopo: Monitorare gli utenti del sistema
Precondizioni: L'admin è autenticato
Passi:

1. L'admin apre la sezione dedicata.
2. Il sistema recupera la lista degli utenti dal backend.
3. La lista viene visualizzata.

Estensioni:

- Recupero fallito → messaggio di errore.

Postcondizioni: La lista degli utenti è visualizzata.



Eliminazione Utente

Consente all'admin di eliminare un utente esistente.

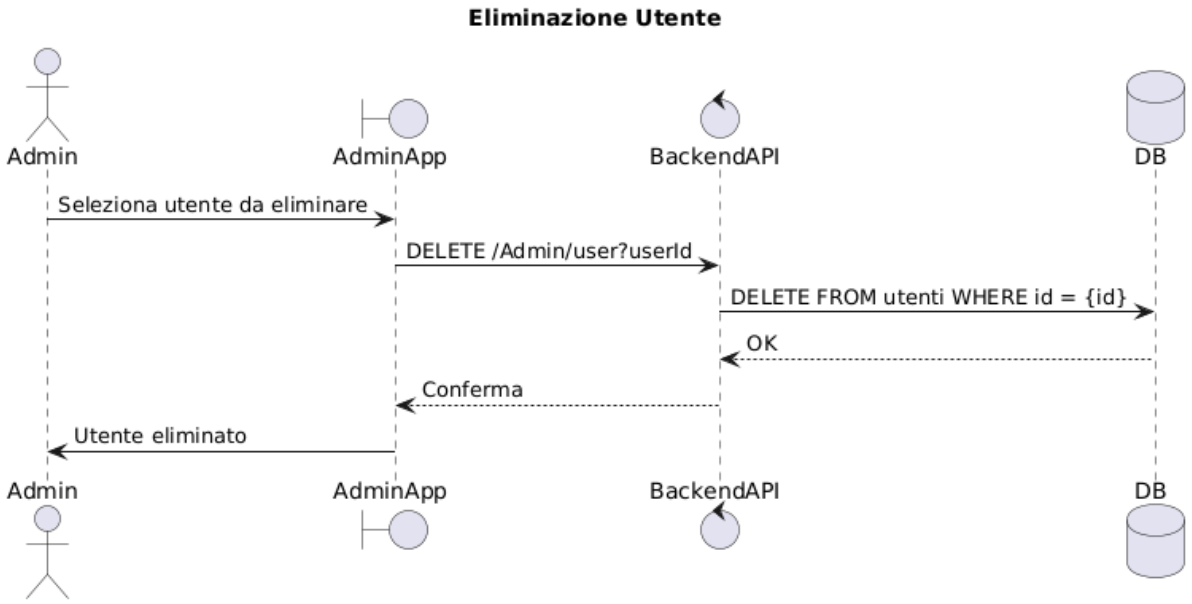
Attori: Admin autenticato
Scopo: Rimuovere utenti non più attivi
Precondizioni: L'admin è autenticato
Passi:

1. L'admin seleziona un utente dalla lista.
2. Conferma l'eliminazione.
3. Il sistema elimina l'utente dal backend.

Estensioni:

- Eliminazione fallita → messaggio di errore.

Postcondizioni: L'utente è stato eliminato.



1.3 Diagrammi di Attività

In aggiunta alla descrizione testuale dei casi d'uso, questa sezione propone una serie di diagrammi UML che illustrano visivamente i principali processi e flussi operativi dell'applicazione. L'utilizzo di questi strumenti grafici consente di chiarire in modo immediato e strutturato le dinamiche tra utenti e sistema, riducendo le possibilità di fraintendimenti e facilitando la comunicazione tra sviluppatori e analisti. I diagrammi rappresentano inoltre un valido supporto per la comprensione, la manutenzione e l'evoluzione futura del software, offrendo una panoramica sintetica ma esaustiva delle logiche implementate.

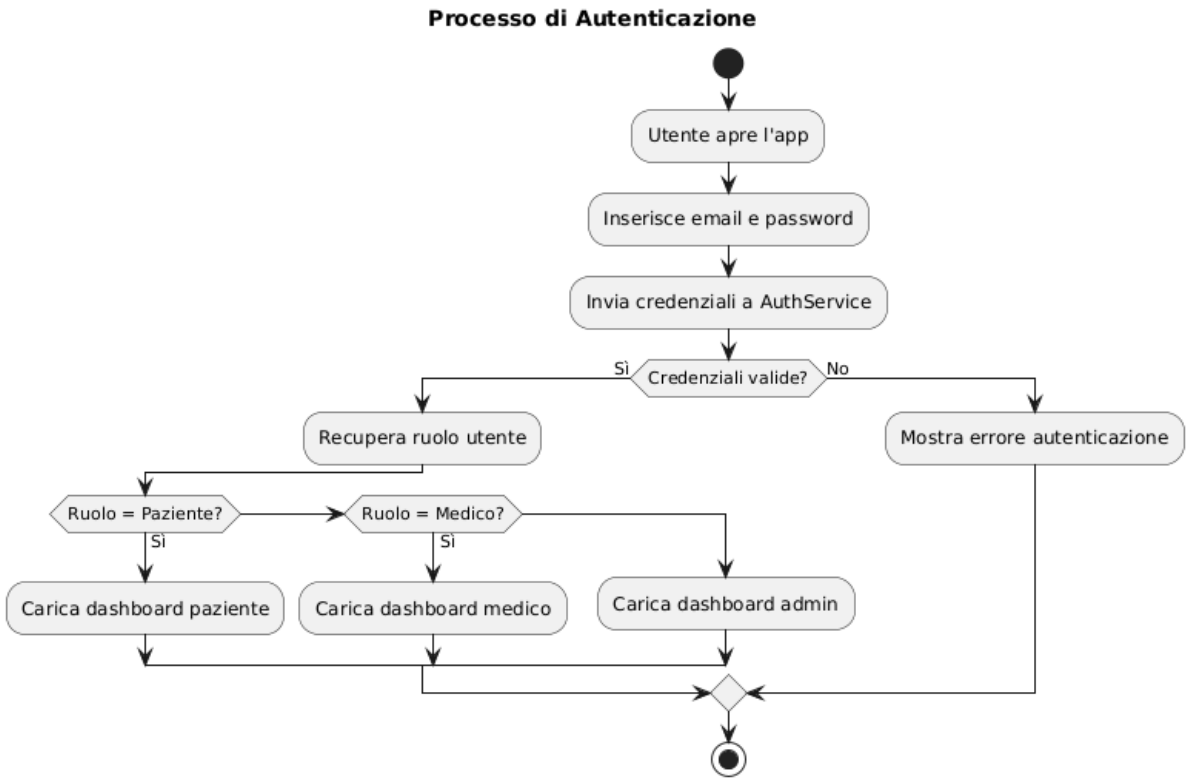


Figure 1: Diagramma di Attività per il processo di autentificazione. Evidenzia il flusso di autenticazione dell'utente all'avvio dell'app.

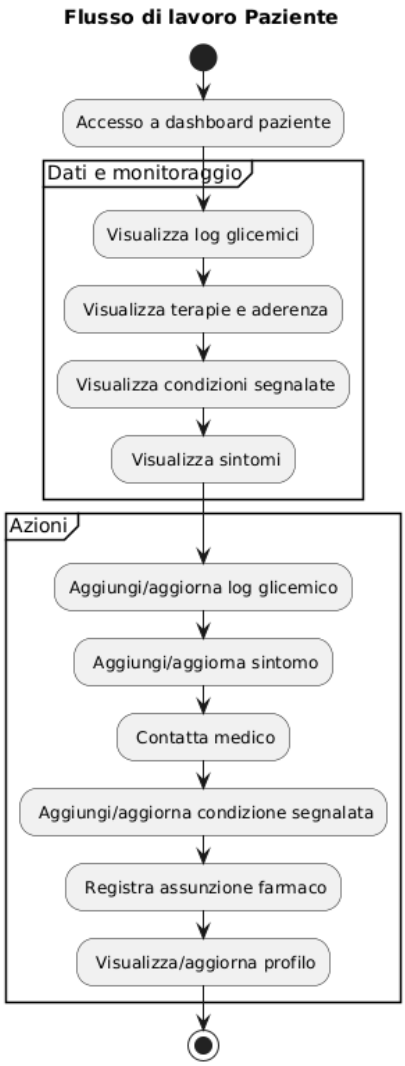


Figure 2: Diagramma di Attività per il flusso di lavoro dell'attore Paziente. Partendo dalla dashboard, il diagramma illustra le diverse funzionalità a sua disposizione.

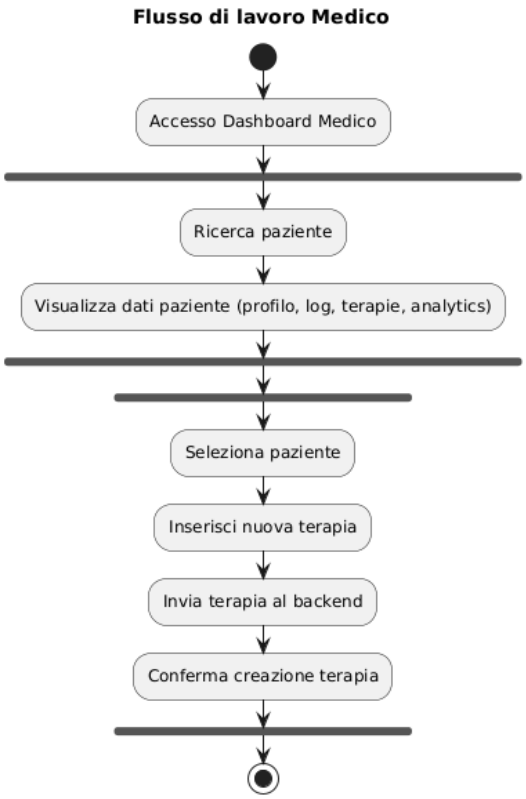


Figure 3: Diagramma di Attività per il flusso di lavoro dell'attore Medico. Il flusso evidenzia la natura ciclica del lavoro del medico.

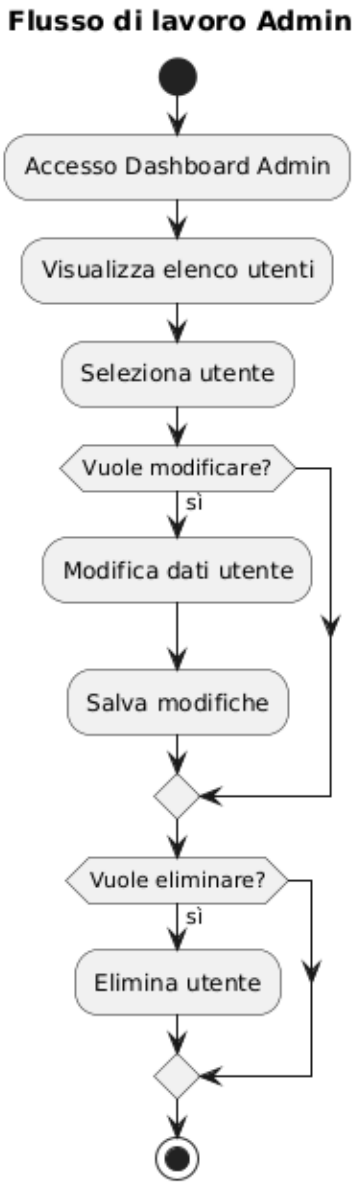
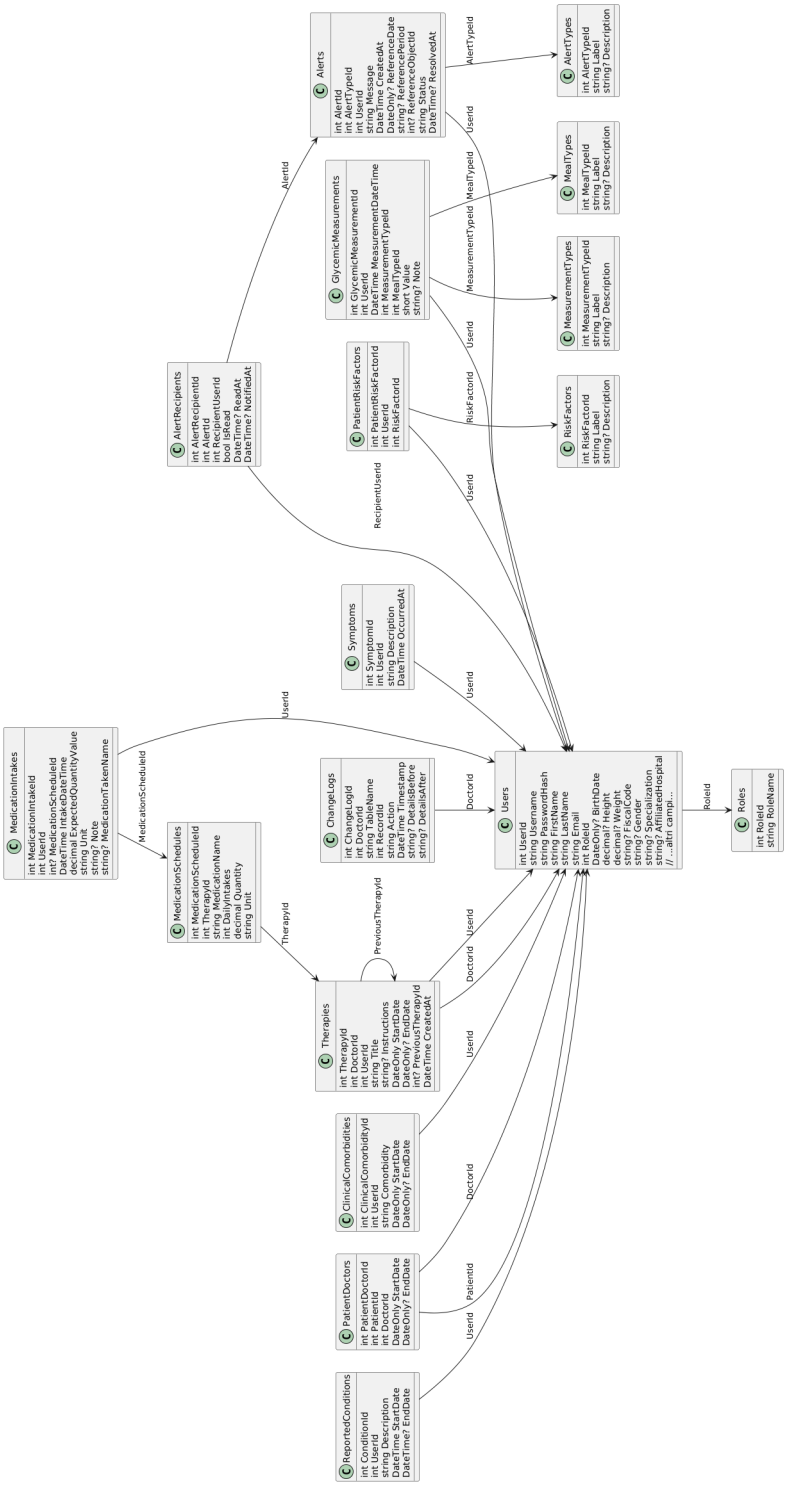


Figure 4: Diagramma di Attività per il flusso di lavoro dell'attore Admin.

1.4 Diagramma delle Classi

Il diagramma delle classi rappresenta la struttura statica principale dell'applicazione, mostrando le entità fondamentali (classi), i loro attributi e le relazioni che intercorrono tra di esse. Questo tipo di diagramma fornisce una visione d'insieme delle componenti del dominio, facilitando la comprensione dell'architettura del sistema e delle connessioni tra i diversi oggetti. È uno strumento essenziale sia in fase di progettazione che di manutenzione, poiché aiuta a individuare le responsabilità delle varie classi e a chiarire come interagiscono tra loro.



2 Sviluppo: progetto dell'architettura ed implementazione del sistema

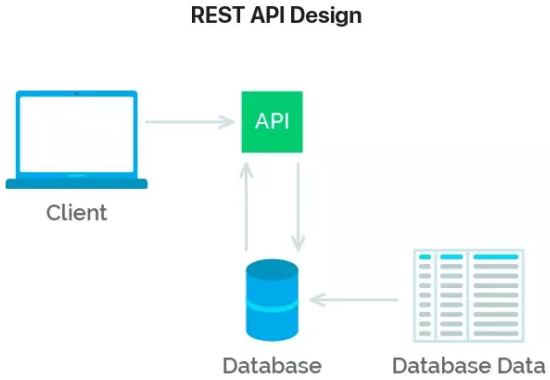
2.1 Note sul processo di sviluppo

Il processo di sviluppo dell'applicazione si è basato su una progettazione modulare e scalabile, con particolare attenzione alla separazione delle responsabilità e alla manutenibilità del codice. La suddivisione in cartelle tematiche (`models`, `components`, `services`, `screens`, `hooks`) ha permesso di isolare le diverse aree funzionali, facilitando sia lo sviluppo parallelo da parte di più sviluppatori sia l'estensione futura del sistema. L'adozione di pattern architetturali e di progettazione consolidati ha guidato le scelte implementative, riducendo la complessità e favorendo la riusabilità dei componenti. L'utilizzo di interfacce TypeScript per la definizione dei modelli dati e di DTO C# per il backend ha garantito tipizzazione forte e coerenza tra i due livelli.

2.2 Pattern Usati

2.2.1 Pattern Architetturali

L'architettura del progetto segue un approccio client–server, in cui l'app mobile è sviluppata in React Native (TypeScript) e comunica tramite API REST con un backend .NET. Il backend gestisce la logica di business, l'autenticazione e la persistenza dei dati su database relazionali, mentre il frontend si concentra sull'interfaccia e sull'esperienza utente.



L'app mobile adotta una struttura a strati ispirata al pattern Model–View–Controller (MVC), adattato al contesto React Native:

- **Model**: le interfacce in `models/` rappresentano le entità del dominio (es. `User`, `GlycemicMeasurement`, `Therapy`) e sono utilizzate per la serializzazione/deserializzazione dei dati scambiati con il backend.
- **View**: le cartelle `components/` e `screens/` contengono i componenti React responsabili della presentazione e dell'interazione con l'utente.
- **Controller/Service**: la logica di interazione con le API REST e la gestione dello stato applicativo sono affidate ai servizi (`services/`) e agli hook personalizzati (`hooks/`), che fungono da intermediari tra UI e backend.

Questa architettura consente una chiara separazione delle responsabilità, facilitando manutenzione, estendibilità e testabilità. L'approccio client–server permette di scalare indipendentemente frontend e backend e di adottare tecnologie diverse per ciascun livello. L'integrazione tra mobile e backend avviene tramite contratti API ben definiti descritti in formato OpenAPI e documentati automaticamente con Swagger, che ne gestisce anche il versionamento (`/v1`, `/v2`...), con attenzione alla retro compatibilità.
La gestione dello stato e della sincronizzazione dei dati tra client e server è supportata da hook come `useRefreshOnFocus`, che garantiscono che l'interfaccia utente sia sempre coerente con lo stato reale dei dati.

2.2.2 Pattern Strutturali

Dal punto di vista strutturale, il progetto adotta diversi pattern per migliorare modularità e manutenibilità:

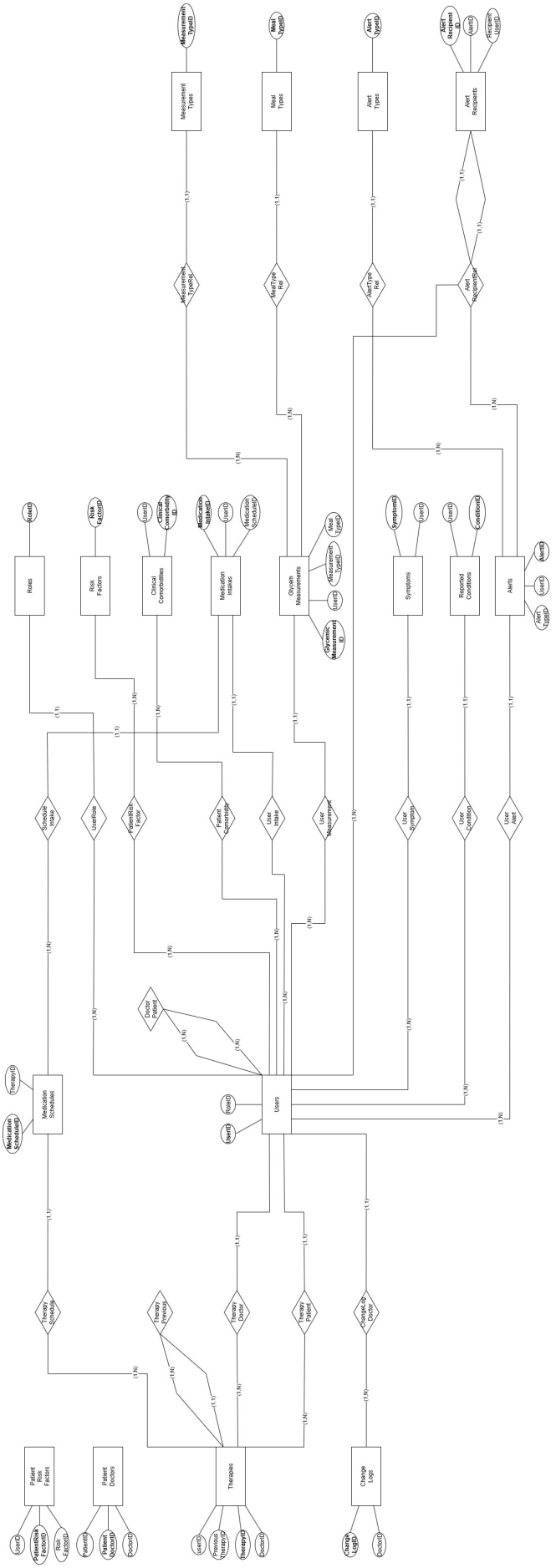
- **Data Transfer Object (DTO)**: le interfacce in `models/` (frontend) e le classi in `DTOs/` (backend) fungono da DTO, facilitando il trasferimento tipizzato e strutturato dei dati tra frontend e backend.
- **Singleton**: i servizi come `AuthService`, `PatientService` e `TherapyService` sono implementati come singleton, centralizzando la gestione delle chiamate API e dello stato condiviso.
- **Facade**: i servizi agiscono anche da facade, offrendo un'interfaccia semplificata verso la logica di business e l'accesso ai dati, nascondendo la complessità delle chiamate API e della gestione degli errori ai componenti della UI.
- **Observer**: l'utilizzo di hook personalizzati, come `useRefreshOnFocus`, implementa il pattern observer, permettendo ai componenti di reagire automaticamente ai cambiamenti di stato o agli eventi di sistema (es. ritorno in primo piano dell'app).

L'architettura attuale, pur solida, presenta le tipiche sfide dei sistemi distribuiti, come la gestione della sincronizzazione offline e degli errori di rete. In prospettiva, l'introduzione di pattern come il *Repository* (per astrarre l'accesso ai dati locali/remoti) e di un *Service Layer* più strutturato potrebbe migliorare ulteriormente modularità e scalabilità, facilitando anche l'integrazione con altri servizi o piattaforme.
L'adozione di questi pattern ha permesso di realizzare un'applicazione robusta, estendibile e manutenibile, in cui ogni componente ha responsabilità chiare e la comunicazione tra frontend e backend avviene in modo sicuro e standardizzato tramite API REST.

3 Progettazione del Database

GlucoTrackDB è un database relazionale progettato per supportare la gestione clinica e l'autogestione dei pazienti diabetici. Il sistema consente il monitoraggio di terapie, assunzioni di farmaci (programmate e non), misurazioni glicemiche, sintomi, comorbidità, fattori di rischio e allerte cliniche.
L'API funge da intermediario tra l'applicazione e il database, implementando una validazione di primo e secondo livello sui dati ricevuti, garantendo che solo dati coerenti e validi vengano scritti nel database.

3.1 Diagramma Entità–Relazione (ER)



3.2 Schema Relazionale

Lo schema relazionale rappresenta la traduzione del diagramma ER in una struttura tabellare ottimizzata per l'implementazione in un database relazionale.

Roles(*RoleId*, RoleName)

Users(*UserId*, Username, PasswordHash, FirstName, LastName, Email, RoleId, BirthDate, Height, Weight, FiscalCode, Gender, Specialization, AffiliatedHospital, CreatedAt, LastAccess)
FK: RoleId → **Roles(RoleId)**

PatientDoctors(*PatientDoctorId*, PatientId, DoctorId, StartDate, EndDate)
FK: PatientId → **Users(UserId)**
FK: DoctorId → **Users(UserId)**

RiskFactors(*RiskFactorId*, Label, Description)

PatientRiskFactors(*PatientRiskFactorId*, UserId, RiskFactorId)
FK: UserId → **Users(UserId)**
FK: RiskFactorId → **RiskFactors(RiskFactorId)**

ClinicalComorbidities(*ClinicalComorbidityId*, UserId, Comorbidity, StartDate, EndDate)
FK: UserId → **Users(UserId)**

Therapies(*TherapyId*, DoctorId, UserId, Title, Instructions, StartDate, EndDate, PreviousTherapyId, CreatedAt)
FK: DoctorId → **Users(UserId)**
FK: UserId → **Users(UserId)**
FK: PreviousTherapyId → **Therapies(TherapyId)**

MedicationSchedules(*MedicationScheduleId*, TherapyId, MedicationName, DailyIntakes, Quantity, Unit)
FK: TherapyId → **Therapies(TherapyId)**

MedicationIntakes(*MedicationIntakeId*, UserId, MedicationScheduleId, IntakeDateTime, ExpectedQuantityValue, Unit, Note, MedicationTakenName)
FK: UserId → **Users(UserId)**
FK: MedicationScheduleId → **MedicationSchedules(MedicationScheduleId)**

MeasurementTypes(*MeasurementTypeId*, Label, Description)

MealTypes(*MealTypeId*, Label, Description)

GlycemicMeasurements(*GlycemicMeasurementId*, UserId, MeasurementDateTime, MeasurementTypeId, MealTypeId, Value, Note)
FK: UserId → **Users(UserId)**
FK: MeasurementTypeId → **MeasurementTypes(MeasurementTypeId)**
FK: MealTypeId → **MealTypes(MealTypeId)**

Symptoms(*SymptomId*, UserId, Description, OccurredAt)
FK: UserId → **Users(UserId)**

ReportedConditions(*ConditionId*, UserId, Description, StartDate, EndDate)
FK: UserId → **Users(UserId)**

AlertTypes(*AlertTypeId*, Label, Description)

Alerts(*AlertId*, AlertTypeId, UserId, Message, CreatedAt, ReferenceDate, ReferencePeriod, ReferenceObjectId, Status, ResolvedAt)
FK: AlertTypeId → **AlertTypes(AlertTypeId)**
FK: UserId → **Users(UserId)**

AlertRecipients(*AlertRecipientId*, AlertId, RecipientUserId, IsRead, ReadAt, NotifiedAt)
FK: AlertId → **Alerts(AlertId)**
FK: RecipientUserId → **Users(UserId)**

ChangeLogs(*ChangeLogId*, DoctorId, TableName, RecordId, Action, Timestamp, DetailsBefore, DetailsAfter)
FK: DoctorId → **Users(UserId)**

4 Attività di Test del Software

La qualità e l'affidabilità dell'applicazione sono state garantite attraverso una strategia di test multilivello, che comprende *unit test*, test di interazione e test di consistenza e integrità dei dati. Di seguito vengono descritte le principali attività svolte, con esempi rappresentativi e riferimenti agli ambienti di test implementati.

4.1 Unit Test

Gli *unit test* sono stati implementati per verificare il corretto funzionamento delle singole unità di codice, come funzioni, metodi e componenti isolati.

- **Backend API**: I test unitari per l'API sono localizzati nella cartella `GlucoTrack_api.Tests/`. Questi test coprono la logica dei controller, dei servizi e delle funzioni di utilità, assicurando che ogni metodo restituisca i risultati attesi anche in presenza di input anomali o errori.
- **App Mobile**: I test unitari per l'applicazione mobile sono presenti nella cartella `src/tests/`. Qui vengono testati i componenti React, gli hook personalizzati e le funzioni di utilità, verificando la corretta gestione dello stato e delle props.

Gli unit test sono eseguiti automaticamente ad ogni modifica del codice, garantendo la regressione e la stabilità delle funzionalità di base.

```
// Esempio di unit test per il controller delle allerte
[Fact]
public void GetAlertById_ReturnsAlert_WhenIdIsValid()
{
    var controller = new AlertController(...);
    var result = controller.GetAlertById(1);
    Assert.NotNull(result);
    Assert.Equal(1, result.Id);
}
```

Listing 1: Esempio di Unit Test (API, C)

```
// Esempio di unit test per un hook personalizzato
import { renderHook } from '@testing-library/react-hooks';
import useAuth from '../hooks/useAuth';

test('useAuth restituisce utente autenticato', () => {
  const { result } = renderHook(() => useAuth());
  expect(result.current.user).toBeDefined();
});
```

Listing 2: Esempio di Unit Test (App, TypeScript)

4.2 Test di Interazione

I test di interazione verificano il corretto funzionamento dei flussi applicativi e delle interazioni tra i diversi componenti del sistema.

- **App Mobile**: Sono stati implementati test di interazione per simulare l'utilizzo reale dell'applicazione, come la navigazione tra schermate, l'inserimento di dati, la ricezione di notifiche e la gestione degli errori. Questi test assicurano che l'esperienza utente sia fluida e priva di malfunzionamenti.
- **API**: I test di interazione sull'API verificano la corretta gestione delle richieste HTTP, la validazione degli input e la coerenza delle risposte, anche in presenza di condizioni di errore o dati mancanti.

I test di interazione sono fondamentali per garantire che le funzionalità principali siano accessibili e funzionanti in scenari realistici.

```typescript
// Esempio di test di interazione su una schermata di login
import { render, fireEvent } from '@testing-library/react-native';
import LoginScreen from '../screens/LoginScreen';

test('mostra errore con credenziali errate', () => {
  const { getByText, getByPlaceholderText } = render(<LoginScreen />);
  fireEvent.changeText(getByPlaceholderText('Email'), 'wrong@example.com');
  fireEvent.changeText(getByPlaceholderText('Password'), 'wrongpass');
  fireEvent.press(getByText('Accedi'));
  expect(getByText('Credenziali non valide')).toBeTruthy();
});
```

Listing 3: Esempio di Test di Interazione (App, TypeScript)

4.3 Test di Consistenza e Integrità dei Dati

L'integrità e la consistenza dei dati sono garantite da una combinazione di validazioni applicative e vincoli a livello di database:

- **Validazione a due livelli**: L'API funge da intermediario tra l'applicazione e il database, implementando una validazione di primo livello (formale/sintattica) e secondo livello (logica/coerenza) sui dati ricevuti. Solo dati coerenti e validi vengono scritti nel database.
- **Chiavi primarie ed esterne**: Ogni tabella del database possiede una chiave primaria (`PRIMARY KEY`) e, dove necessario, chiavi esterne (`FOREIGN KEY`) che garantiscono l'integrità referenziale tra le entità. I vincoli di unicità (`UNIQUE`) sono applicati su campi come username, email e label.
- **Integrità referenziale**: Il database impedisce la cancellazione di record "padre" se esistono record "figli" che li referenziano, bloccando operazioni che violerebbero i vincoli di integrità. Tutte le operazioni critiche sono tracciate tramite la tabella `ChangeLogs` per garantire auditabilità.

- **Validazione e sicurezza**: L'API effettua controlli formali e logici prima di ogni operazione, mentre il database applica vincoli come `NOT NULL`, `UNIQUE` e `CHECK` per impedire l'inserimento o la modifica di dati non validi.

Esempi di integrità:

- Un utente non può essere eliminato se esistono terapie, misurazioni, allerte o altre entità collegate.
- Le relazioni tra pazienti e medici, fattori di rischio, terapie e assunzioni sono sempre garantite tramite chiavi esterne.
- Le operazioni di modifica e cancellazione sono tracciate tramite la tabella `ChangeLogs`.

4.4 Automazione e Ambiente di Test

L'automazione dei test è stata implementata per garantire rapidità e affidabilità nel processo di verifica:

- **Backend API**: I test sono eseguiti tramite framework come xUnit o NUnit, con script di automazione che permettono l'esecuzione continua dei test ad ogni build.
- **App Mobile**: I test sono gestiti tramite framework come Jest e Testing Library per React Native, con integrazione nei processi di CI/CD.
- **Ambiente di Test**: È stato predisposto un ambiente di test isolato, con database di test e dati fittizi per garantire la ripetibilità dei test.

4.5 Conclusioni

La strategia di test multilivello adottata ha permesso di individuare e correggere tempestivamente eventuali anomalie, garantendo un'elevata qualità del software. L'integrazione tra test automatici, validazione dei dati e vincoli di integrità a livello di database assicura che l'applicazione sia robusta, affidabile e pronta per l'utilizzo in scenari reali.

5 Ulteriori note e miglioramenti futuri

Il progetto *GlucoTrack*, pur presentando un'architettura solida, modulare e orientata alla scalabilità, offre margini di miglioramento e potenziali sviluppi futuri in diverse direzioni. Di seguito si propongono alcune considerazioni e linee guida per un'evoluzione funzionale e tecnologica del sistema:

- **Gestione della modalità offline**: attualmente l'applicazione richiede una connessione di rete attiva per la maggior parte delle operazioni. L'introduzione di un meccanismo di sincronizzazione offline/online, supportato da un sistema di cache locale, permetterebbe all'utente di continuare ad operare anche in assenza temporanea di connettività, migliorando la robustezza complessiva del sistema.

- **Notifiche push native**: l'invio di notifiche è attualmente affidato principalmente a canali e-mail. L'integrazione con un sistema di notifiche push native (ad es. Firebase Cloud Messaging) consentirebbe una comunicazione più tempestiva ed efficace tra sistema e utente, in particolare in caso di alert clinici urgenti.

- **Integrazione con dispositivi medicali**: un'estensione naturale del progetto consisterebbe nell'abilitare l'interoperabilità con dispositivi di misurazione glicemica (glucometri intelligenti, wearable devices), mediante connessioni Bluetooth o API standard, al fine di automatizzare l'acquisizione dei dati clinici e ridurre l'errore umano.

- **Canale di comunicazione interno**: al fine di migliorare la comunicazione tra medico e paziente, potrebbe essere integrato un modulo di messaggistica in-app, con protocolli di sicurezza adeguati (es. crittografia end-to-end), evitando il ricorso a strumenti esterni meno controllati.

- **Logging e audit avanzati**: la tracciabilità delle modifiche è attualmente garantita tramite la tabella `ChangeLogs`. In prospettiva, l'adozione di strumenti esterni per il monitoraggio centralizzato dei log (es. ELK Stack o servizi cloud dedicati) potrebbe facilitare l'analisi delle anomalie, la diagnostica e il mantenimento del sistema in esercizio.

- **Applicazioni di intelligenza artificiale**: infine, l'introduzione progressiva di modelli predittivi basati su machine learning, ad esempio per identificare pazienti a rischio di ipoglicemia o per suggerimenti terapeutici personalizzati, potrebbe rappresentare un valore aggiunto rilevante, mantenendo comunque un approccio evidence-based e validato clinicamente.