



BSN WebApp

Report

Arturo Matteo Santarlasci

arturomatteo.santarlasci@mail.polimi.it

github.com/ArturoMatteoSantarlasci

Filippo Polvani

filippo.polvani@mail.polimi.it

github.com/filippo0000

Gennaio 2026

Indice

1. Abstract	1
2. Tecnologie Utilizzate	2
2.1 Backend	2
2.2 Persistenza dei Dati	2
2.3 Comunicazione e Streaming dei Dati	2
2.4 Frontend	2
2.5 Containerizzazione e DevOps	3
3. Organizzazione del Codice	4
3.1 Controller Layer	4
3.2 Service Layer	4
3.3 Persistence Layer (Repository)	5
3.4 Model e DTO	5
3.5 Configurazione e Sicurezza	5
4. MariaDB - Schema database relazionale	6
5. Funzionalità	7
5.1 Autenticazione e Accesso	7
5.2 Gestione Campagne	7
5.3 Manutenzione e Configurazione (DEV)	7
5.4 Link video funzionalità	8
6. Sicurezza	9
6.1 Autenticazione con Spring Security e JWT	9
6.2 Autorizzazione basata sui ruoli	10
6.3 Protezione delle password	10
7. Miglioramenti futuri possibili	11
7.1 Upgrade a InfluxDB 2.x	11
7.2 Modernizzazione UI front-end	11
7.3 Esperienza utente e interfaccia	11

1. Abstract

Overview Il codice sorgente del progetto è disponibile al seguente link: [BSN_WebApp](#).

Il progetto BSN WebApp è una piattaforma web integrata progettata per la gestione, l'acquisizione e la visualizzazione di dati provenienti da reti di sensori corporei (**Body Sensor Networks - BSN**).

L'obiettivo principale del sistema è fornire al personale medico e ai ricercatori uno strumento unificato che permetta sia il monitoraggio in tempo reale dei segnali fisiologici (come accelerometri e giroscopi) sia l'archiviazione efficiente delle serie temporali per analisi successive. L'architettura è stata sviluppata per garantire bassa latenza nella trasmissione dei dati e scalabilità nella gestione di volumi elevati di misurazioni inerziali.

Il sistema implementa una rigida separazione dei privilegi distinguendo il ruolo **USER_MED**, dedicato al personale clinico per la gestione dei pazienti e l'avvio delle campagne, dal ruolo **USER_DEV**, riservato agli admin per la configurazione degli script, dei database e la manutenzione dei sensori.

Tools L'infrastruttura tecnologica adotta un approccio moderno basato su microservizi logici e containerizzazione. Il core del backend è sviluppato in **Java** con il framework **Spring Boot**, che gestisce la logica di business e le **API REST**. La comunicazione real-time è garantita dal protocollo **MQTT** per l'ingestione dei dati dai sensori e dalla tecnologia **Server-Sent Events (SSE)** per lo streaming unidirezionale verso il frontend. Inoltre, il broker/server MQTT può persistere i messaggi ricevuti su database; l'infrastruttura consente di configurare anche un salvataggio aggiuntivo su un ulteriore database scelto dall'utente.

La persistenza dei dati segue un modello ibrido (Polyglot Persistence): un database relazionale **MariaDB** gestisce i metadati strutturati, mentre **InfluxDB** è utilizzato specificamente per l'archiviazione ad alte prestazioni delle serie temporali (Time-Series Data).

Il frontend è realizzato come applicazione web multi-pagina con rendering server-side tramite **Thymeleaf**. Si fa uso della libreria **HTMX** sul lato client per arricchire l'interattività. Per la visualizzazione in tempo reale viene impiegata la libreria **Chart.js**, **JavaScript (vanilla)** per creare script per la gestione della logica UI. Lo stile dell'applicazione sono basati su **Tailwind CSS** con componenti di **DaisyUI**.

L'intera documentazione del progetto è stata redatta tramite **Typst**.

I componenti del sistema vengono eseguiti come container e orchestrati tramite **Docker Compose**.

La documentazione fornisce sia una panoramica generale, sia un'analisi tecnica approfondita rivolta a sviluppatori e manutentori del progetto.

2. Tecnologie Utilizzate

Il progetto BSN WebApp adotta un insieme di tecnologie moderne per la realizzazione dell’interfaccia utente, della logica applicativa e dell’infrastruttura di supporto. Di seguito vengono descritte le principali tecnologie utilizzate, organizzate per ambito.

2.1 Backend

Il backend dell’applicazione è sviluppato in **Java** utilizzando il framework **Spring Boot** (versione 3.x), che consente una rapida configurazione e un’architettura modulare basata sul paradigma MVC.

Le principali componenti Spring impiegate sono:

- **Spring MVC**: gestione del routing web e dei controller;
- **Spring Data JPA**: accesso ai dati e persistenza su database relazionale;
- **Spring Security**: autenticazione e autorizzazione degli utenti.

2.2 Persistenza dei Dati

La persistenza è suddivisa in due livelli distinti, in base alla natura dei dati gestiti.

- **Database Relazionale**: Viene utilizzato **MariaDB** per l’archiviazione dei dati strutturati e persistenti (utenti, sensori, pazienti, campagne, configurazioni). Il database è eseguito all’interno di un container Docker e l’accesso avviene tramite **JDBC** utilizzando il driver `mariadb-java-client`.
- **Database Time-Series**: I dati grezzi dei sensori sono memorizzati in **InfluxDB** (versione 1.8), un database ottimizzato per serie temporali. Questa soluzione consente query efficienti su intervalli temporali ed è particolarmente adatta alla visualizzazione di grafici storici.

2.3 Comunicazione e Streaming dei Dati

La comunicazione tra sensori e server avviene tramite il protocollo **MQTT**. Gli script Python pubblicano i dati su un broker MQTT esterno, mentre il backend si sottoscrive ai topic tramite la libreria **Eclipse Paho MQTT**.

Per la trasmissione dei dati in tempo reale al browser viene utilizzato il meccanismo di **Server-Sent Events (SSE)**, che permette al server di mantenere una connessione persistente con il client e inviare aggiornamenti continui.

2.4 Frontend

L’interfaccia utente è realizzata come applicazione web multi-pagina con rendering server-side tramite **Thymeleaf**.

Le principali tecnologie frontend utilizzate sono:

- **HTMX**: aggiornamento dinamico di frammenti HTML senza ricaricare l’intera pagina;
- **Chart.js**: visualizzazione dei dati sensoriali tramite grafici interattivi;
- **JavaScript (Vanilla)**: gestione della logica client-side personalizzata;
- **Tailwind CSS con DaisyUI**: definizione dello stile grafico e del layout responsivo.

2.5 Containerizzazione e DevOps

Il progetto include una configurazione **Docker Compose** per l'orchestrazione dei servizi di supporto (database, servizi esterni).

Il sistema di build è gestito tramite **Maven**, che cura la gestione delle dipendenze e il ciclo di build dell'applicazione. Tra le dipendenze principali si segnalano:

- driver JDBC per MariaDB;
- client Java per InfluxDB;
- libreria Eclipse Paho MQTT;
- supporto a **JSON Web Token (JWT)** per la sicurezza.

3. Organizzazione del Codice

L'organizzazione interna del codice segue il paradigma **Model–View–Controller (MVC)** e una chiara separazione in strati, al fine di garantire modularità, manutenibilità e una netta distinzione delle responsabilità.

3.1 Controller Layer

Il package **Controller** contiene i controller Spring responsabili della gestione delle richieste HTTP in ingresso.

Nel progetto sono presenti:

- **Controller MVC** (annotati con `@Controller`), che restituiscono viste renderizzate tramite **Thymeleaf**;
- **Controller REST** (annotati con `@RestController`), che espongono API JSON utilizzate per funzionalità dinamiche basate su **AJAX/HTMX** o per servizi dati.

I nomi dei controller riflettono chiaramente il loro ambito funzionale:

- `HomePageController` gestisce la homepage e le pagine generali;
- `AuthController` si occupa delle operazioni di autenticazione e registrazione;
- controller come `CampagnaMedController` e `PersonaMedController` gestiscono le funzionalità dedicate all'utente medico;
- `SensoreDevController` espone operazioni sui sensori per l'utente sviluppatore.

È inoltre presente un `ImuStreamController`, dedicato allo streaming dei dati dei sensori tramite **Server-Sent Events (SSE)**. IMU (**Inertial Measurement Unit**) indica l'insieme di dati trasmessi in tempo reale.

3.2 Service Layer

Il package **Service** incapsula la logica di business dell'applicazione, coordinando le operazioni complesse e transazionali e fungendo da livello intermedio tra controller e persistenza.

Alcuni servizi rilevanti sono:

- `CampagnaService`, che gestisce l'avvio e la terminazione delle campagne di monitoraggio;
- `MqttSseService`, che si connette al broker MQTT e inoltra i dati in ingresso sia verso InfluxDB (tramite `InfluxWriteService`) sia verso i client SSE;
- `InfluxQueryService`, che fornisce metodi per interrogare i dati storici delle campagne conclusive;
- `AuthService`, responsabile della validazione e registrazione degli utenti;
- `PersonaService`, `SensoreService`, `TipoCampagnaService`, che incapsulano la logica CRUD delle rispettive entità.
- `CampagnaWatchdog`: job schedulato che verifica la coerenza tra lo stato campagna a DB e i processi attivi. Se una campagna risulta IN_CORSO ma non ha processi vivi, la termina e aggiorna le note.

3.3 Persistence Layer (Repository)

Il layer di persistenza è gestito tramite **Spring Data JPA** nel package **Repository**. Per ciascuna entità principale è presente un'interfaccia repository dedicata (ad esempio `UtenteRepository`, `PersonaRepository`, `CampagnaRepository`, `SensoreRepository`).

Queste interfacce estendono `JpaRepository` e vengono implementate automaticamente da Spring al runtime, consentendo operazioni CRUD e query personalizzate senza scrivere manualmente il codice di accesso al database.

—

3.4 Model e DTO

Il package **Model** contiene le entità JPA che rappresentano il modello dati dell'applicazione e mappano le tabelle del database relazionale.

Accanto alle entità, il package **DTO** include oggetti di trasferimento dati utilizzati per:

- inviare informazioni aggregate al frontend;
- disaccoppiare il modello di persistenza dalle API esposte;
- ottimizzare il payload delle risposte HTTP.

—

3.5 Configurazione e Sicurezza

La configurazione di alto livello è gestita nei package **Config** e **Security**. In particolare:

- `SecurityConfig` configura **Spring Security**, inclusi filtri JWT, autorizzazioni sulle rotte e gestione delle sessioni;
- eventuali configurazioni custom (ad esempio client MQTT) sono definite nel package **Config**.

I parametri applicativi (credenziali DB, URL del broker MQTT, configurazione InfluxDB) sono caricati tramite `application.properties` oppure memorizzati nel database e gestiti tramite interfaccia amministrativa.

4. MariaDB - Schema database relazionale

Il database relazionale (MariaDB) gestisce tutte le informazioni strutturate della piattaforma, come utenti e ruoli, persone monitorate, sensori disponibili e campagne (con le relative associazioni). Lo schema seguente riassume le principali entità e relazioni, evidenziando il legame tra campagne, soggetti e dispositivi utilizzati.

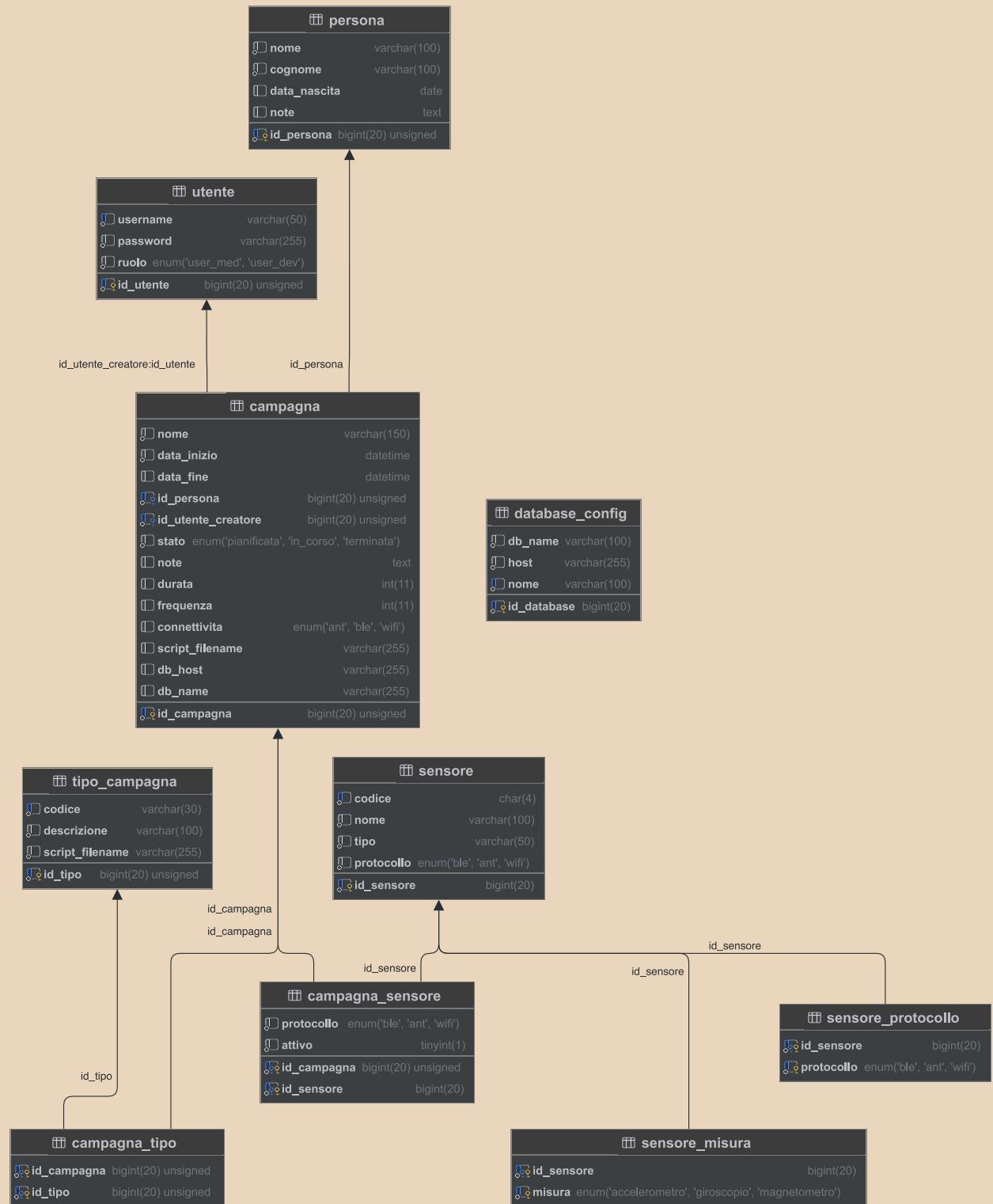


Figura 1: Schema del database relazionale

5. Funzionalità

Le funzionalità della piattaforma sono suddivise in base al ruolo assegnato all'utente:

- **MED**: **Profilo Medico**, focalizzato sulla gestione clinica, dei pazienti e dell'acquisizione dati.
- **DEV**: **Profilo Tecnico**, con accesso completo alle configurazioni di sistema, sensori e script.

5.1 Autenticazione e Accesso

Il sistema garantisce un accesso sicuro e profilato.

- **Registrazione Utente**: Creazione di un nuovo account selezionando il ruolo appropriato (**MED** o **DEV**).
- **Login/Logout**: Accesso tramite credenziali e terminazione sicura della sessione.

5.2 Gestione Campagne

Questa sezione rappresenta il **core** dell'applicazione, permettendo di configurare, avviare e monitorare le sessioni di acquisizione dati.

Creazione Nuova Campagna

Il wizard di creazione guida l'utente attraverso i seguenti passaggi:

1. **Selezione Persona**: Scelta del soggetto dall'anagrafica o creazione rapida di una **nuova persona**.
2. **Tipo di Campagna**: Selezione dello script di acquisizione (protocollo medico o test).
3. **Sensori**: Associazione dei sensori necessari per la misurazione.
4. **Database Aggiuntivo** (Opzionale): Configurazione di un database Time-Series esterno per la ridondanza dei dati (il salvataggio sul DB principale è sempre garantito).

Ciclo di Vita e Monitoraggio

Gli stati principali di una campagna sono evidenziati nell'interfaccia:

- **Avvio**: Inizio dell'esecuzione dello script Python sottostante.
- **Monitoraggio Live**: Visualizzazione in tempo reale dei dati streaming (grafici e valori).
- **Terminazione**: Chiusura formale della sessione e salvataggio definitivo.

Consultazione

- **Campagne Attive**: Vista dedicata alle sessioni attualmente in corso.
- **Storico Campagne**: Archivio filtrabile per **stato**, **paziente** o **data**, per analizzare le attività passate.

5.3 Manutenzione e Configurazione (DEV)

Funzionalità avanzate per la gestione dell'infrastruttura IoT e dati.

Gestione Sensori

- **Creazione**: Inserimento di nuovi dispositivi nel sistema.
- **Lista**: Elenco completo dei sensori configurati.
- **Eliminazione**: Rimozione sicura (inibita se il sensore è legato a campagne storiche).

Database Time-Series

- **Configurazione**: Definizione dei parametri di connessione (Host, Porta, Nome DB).

- **Rimozione:** Gestione delle configurazioni.

Script e Tipi Campagna

Consultazione e upload dei tipi di campagna (script Python). Gli script sono fisicamente residenti nella cartella bsn/script/ e modificabili direttamente se necessario.

5.4 Link video funzionalità

- Registrazione, login, manutenzione: [BSN_WebApp_1](#)
- Avvio campagna, aggiunta paziente, monitoraggio tempo reale: [BSN_WebApp_2](#)
- Terminazione campagna, visualizzazione dati storici: [BSN_WebApp_3](#)

6. Sicurezza

La **BSN WebApp** adotta misure di sicurezza standard per proteggere applicazione e dati, con particolare attenzione a **autenticazione e segregazione dei permessi**, dato che esistono ruoli distinti.

6.1 Autenticazione con Spring Security e JWT

L'implementazione segue un'architettura **stateless** basata su JWT: il server non mantiene una sessione applicativa “persistente” tra una richiesta e la successiva. Ogni richiesta HTTP viene autenticata (o rifiutata) esclusivamente in base al token presentato dal client.

Il meccanismo si basa su due concetti fondamentali:

- una **catena di filtri (SecurityFilterChain)** che intercetta ogni richiesta HTTP in ingresso e applica controlli di sicurezza prima di raggiungere i controller;
- un **SecurityContext** (accessibile tramite `SecurityContextHolder`) che contiene l'utente autenticato per la **singola richiesta** corrente, oppure `null/anonimo` se non autenticato.

A livello di configurazione, la classe di **SecurityConfig** definisce principalmente:

- la `SecurityFilterChain`, cioè l'ordine e la composizione dei filtri applicati alle richieste;
- l'oggetto `HttpSecurity`, tramite cui si impostano regole di accesso, CSRF/CORS, gestione delle sessioni (in stateless), ecc.
- **Login**: il client invia le credenziali. Se valide, il server genera un JWT tramite `JwtUtil` e lo restituisce al client.
- **Richieste successive**: il client include il token nell'header `Authorization: Bearer <token>`.
- Il filtro `JwtAuthFilter` intercetta la richiesta, valida il token (firma e scadenza) ed estrae le informazioni principali (es. `username` e ruoli).
- Se il token è valido, viene creata un'istanza di `Authentication` e inserita nel **SecurityContext**; da quel momento Spring può applicare le regole di autorizzazione su quella richiesta.

L'oggetto Authentication in Spring Security

Spring Security lavora con un oggetto `Authentication`, che rappresenta l'identità (e i permessi) associati alla richiesta corrente. In particolare:

- **principal**: rappresenta l'utente autenticato; è un oggetto che implementa `UserDetails`, cioè l'interfaccia con cui Spring “vede” un utente;
- **authorities**: insieme di ruoli e/o permessi (`USER_DEV`, `USER_MED`) usati per l'autorizzazione;
- **authenticated**: booleano che vale `true` quando l'autenticazione è stata completata con successo.

In pratica, il `JwtAuthFilter` costruisce/riempie questo oggetto a partire dai dati presenti nel JWT e lo associa al `SecurityContext` della richiesta.

Contenuto e verifica del JWT

Il JWT è un token firmato che include:

- lo **username** (o identificativo utente);
- i **ruoli/authorities** necessari per autorizzare le operazioni;

In fase di verifica, il server controlla l'integrità del token (firma) e la validità temporale (scadenza). Se uno dei controlli fallisce, la richiesta viene trattata come non autenticata e può essere bloccata dalle regole di autorizzazione.

6.2 Autorizzazione basata sui ruoli

L'applicazione distingue almeno due ruoli: USER_MED e USER_DEV. Il backend applica restrizioni su rotte e funzionalità:

- Le rotte “dev” (tipicamente prefissate da /api/v1/dev/** e/o pagine admin) sono accessibili solo a USER_DEV. Un utente medico che tenti l'accesso riceverà un errore di **accesso negato**.
- Le rotte “med” sono destinate a USER_MED. USER_DEV rimane comunque in grado di poter svolgere il 100% delle funzionalità.

6.3 Protezione delle password

Le password non vengono memorizzate in chiaro: sono salvate come **hash**. Si fa uso di **BCrypt** in Spring Security:

- in registrazione: la password viene codificata prima del salvataggio;
- in login: la verifica avviene confrontando l'hash (BCrypt gestisce anche il **salt** in modo sicuro).

7. Miglioramenti futuri possibili

Nonostante nel progetto siano state utilizzate tecnologie inizialmente nuove per noi, e quindi non completamente conosciute all'avvio dello sviluppo, rimangono, come in ogni progetto software, diverse possibilità di miglioramento. Alcune delle principali idee e direzioni evolutive sono riportate di seguito.

7.1 Upgrade a InfluxDB 2.x

InfluxDB 1.8, sebbene stabile, è una versione legacy. Un miglioramento sarebbe passare a **InfluxDB 2 (o superiore)**, che introduce il concetto di bucket, API unificate e richiede token per la sicurezza. L'app andrebbe aggiornata per usare il nuovo client e gestire l'autenticazione, migliorando la sicurezza del dato.

7.2 Modernizzazione UI front-end

Sebbene l'attuale scelta di **CSS + HTMX + JS** sia efficiente, in futuro si potrebbe valutare di passare ad un front-end interamente single-page application (**React/Vue/Angular...**) per offrire esperienze utente ancora più ricche, soprattutto se si vogliono grafici interattivi avanzati, multitouch, ecc. Ciò richiederebbe di trasformare le attuali pagine in API REST complete e separare front-end e back-end. In alternativa l'utilizzo di Web Components o librerie come Stimulus potrebbe migliorare l'organizzazione del codice JS mantenendolo discreto.

7.3 Esperienza utente e interfaccia

Potenziamenti possibili includono:

1. **Funzionalità di annotazione:** permettere al medico di inserire note o event marker durante una campagna (es. "paziente ha riferito sintomo X ora") che vengono registrate e visualizzate sul grafico.
2. **Multi-utenza avanzata:** se in futuro più medici useranno il sistema, introdurre il concetto di team o assegnazione paziente-medico così che ciascun utente veda solo i propri pazienti. Attualmente, tutti gli `USER_MED` possono vedere tutte le persone.