ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

ANALISI, PROGETTAZIONE E PROTOTIPAZIONE DI UN SISTEMA PER LA MAPPATURA DI PUNTI DELLA SALUTE PER AUSL ROMAGNA

Elaborato in
SISTEMI EMBEDDED E INTERNET OF THINGS

Relatore
Prof. ALESSANDRO RICCI

Correlatore
Dott. SAMUELE BURATTINI
Ing. ANGELO CROATTI

Presentata da
FILIPPO MASSARI

III Sessione di Laurea
Anno Accademico 2024 – 2025

A Nino, il primo a sognarmi qui

Indice

Introduzione

La spinta alla digitalizzazione dei servizi ha impattato profondamente, come tanti settori, anche quello sanitario, evidenziando la necessità di strumenti moderni per la gestione e la fruizione delle informazioni territoriali. Il presente elaborato si inserisce in questo contesto e, in collaborazione con l'AUSL Romagna, si pone l'obiettivo di analizzare, progettare e prototipare un'applicazione web per rendere facilmente raccoglibili, fruibili e manutenibili le informazioni relative ai "Punti della Salute", destinata a un pubblico eterogeneo di operatori interni e cittadini. Per governare la complessità del dominio è stato adottato un approccio metodologico Agile, affiancato ai principi del Domain-Driven Design (DDD) per la modellazione del software. L'architettura implementata è strutturata su tre livelli: un backend RESTful in Spring Boot per la gestione capillare della logica di business; un frontend a componenti in React per garantire il riuso sistematico del codice; e un data tier su database PostgreSQL, potenziato dall'estensione PostGIS per l'esecuzione di query geospaziali. Il risultato è un prototipo funzionante che valida la fattibilità tecnica dell'architettura proposta e costituisce una solida base per lo sviluppo futuro di un sistema informativo territoriale completo per l'AUSL Romagna.

Capitolo 1

Contesto ed Analisi del Problema

Il presente elaborato di tesi prende avvio in risposta a una concreta esigenza emersa nel contesto operativo dell'Azienda Unità Sanitaria Locale (AUSL) della Romagna. L'AUSL della Romagna è l'ente pubblico del Servizio Sanitario Regionale che ha il compito di garantire la tutela della salute e l'erogazione di servizi sanitari e sociosanitari per i cittadini delle province di Forlì-Cesena, Ravenna e Rimini. La finalità del progetto, discusso in questa tesi, è l'analisi, l'ideazione e lo sviluppo di una piattaforma software per la mappatura e la gestione strategica dei servizi sanitari territoriali.

1.1 Dominio Applicativo

Il dominio applicativo si focalizza sulle attività degli operatori sanitari direttamente schierati sul territorio. Un virtuoso esempio di tali figure è rappresentato dagli *Infermieri di Famiglia e Comunità* (IFEC), figure professionali la cui missione consiste nell'erogare servizi di assistenza e prevenzione in modo capillare attraverso interventi domiciliari. È proprio dall'operatività quotidiana di figure come queste che scaturisce la necessità di uno strumento in grado di mappare e rendere accessibili, in modo strutturato e georeferenziato, le informazioni relative ai *Punti Territoriali della Salute* (PTS).

Per la fase iniziale di prototipazione, il perimetro del progetto è stato circoscritto al distretto sanitario di Rimini. Tale scelta non è casuale, ma fortemente motivata dal fatto che la provincia di Rimini rappresenta un esempio virtuoso nel percorso di digitalizzazione dei processi sanitari, evoluzione all'interno della quale l'Università di Bologna ha svolto un ruolo attivo in questi anni. Questo scenario preesistente, caratterizzato da una spiccata sensibilità verso l'inno-

vazione tecnologica, ha fornito un terreno fertile e un contesto collaborativo ideale per la raccolta dei requisiti e la validazione di un prototipo.

1.2 Identificazione del Problema

Dall'analisi dello stato dell'arte emerge un quadro di frammentazione tecnologica e informativa. Fenomeno dovuto al fatto che, negli anni, ogni dipartimento di suddivisione dell'azienda ha adottato strategie diverse per documentare e catalogare i dati. Si sono dunque immagazzinate, nel tempo, informazioni non sempre concordi, condivisibili e spesso conservate su supporti eterogenei che disincentivano la messa in condivisione delle stesse. La diretta conseguenza di ciò è una criticità strategica legata alla dispersione dei dati, che ostacola l'accesso ai contenuti sia da parte degli operatori interni sia da parte dei cittadini.

Sebbene i servizi erogati direttamente dall'ente siano documentati sul portale istituzionale, un vasto ecosistema di prestazioni fornite da attori terzi, come associazioni di volontariato, enti privati e terzo settore, rimane escluso da tale mappatura. L'asimmetria informativa, che ne risulta, genera una serie di inefficienze operative che impattano direttamente sulla qualità del servizio in molteplici modi:

- **Frammentazione dell'Accesso per il Cittadino:** L'assenza di un canale informativo unificato e autorevole, comunemente detta *single source of truth*, indirizza i cittadini verso una navigazione complessa tra molteplici fonti non coordinate, con conseguente difficoltà nel reperimento delle informazioni e sottoutilizzo dei servizi disponibili.

- **Inefficienza Operativa per il Personale Sanitario:** Per gli operatori AUSL, l'assenza di un catalogo di servizi centralizzato trasforma il compito di orientare l'utenza in un processo dispendioso in termini di tempo e basato sulla conoscenza individuale piuttosto che su dati strutturati e aggiornati.

- **Rischio di Sovrapposizione e Inefficiente Allocazione delle Risorse:** La mancanza di una visione d'insieme rende difficile identificare aree territoriali scoperte o con un'eccessiva concentrazione di prestazioni simili, portando a una potenziale allocazione inefficiente delle risorse.

- **Obsolescenza e Mancata *Governance* dei Dati:** La natura decentralizzata degli strumenti attuali porta a una rapida obsolescenza delle

informazioni. L'assenza di un processo di *governance* centralizzato genera un rischio concreto di disservizio, indirizzando gli utenti verso risorse non più disponibili e minando la fiducia nell'ente stesso.

Oltre alla gestione delle informazioni pubbliche, emerge una seconda dimensione critica legata allo scambio di dati sensibili interni. Gli operatori necessitano di canali di comunicazione sicuri e tracciabili per scambiare, con altri professionisti, informazioni relative ai pazienti. L'assenza di una piattaforma dedicata rappresenta un ulteriore ostacolo all'efficienza e alla sicurezza delle operazioni.

1.3 Analisi delle Soluzioni in Uso

Il primo passaggio intrapreso nell'approccio all'analisi del dominio è stato quello di studiare con cura le soluzioni già in uso presso l'ente e le tecnologie a supporto. Questa fase risulta di fondamentale importanza nel definire le proprietà del sistema in elaborazione. Partendo infatti dalle criticità emerse nelle soluzioni preesistenti, sarà possibile sviluppare accorgimenti volti a sopperire a tali mancanze e a valorizzare le funzionalità cardine già esistenti.

Lo studio degli strumenti in uso ha messo in luce una moltitudine eterogenea di tecnologie, spesso non interoperabili e prive di un modello di riferimento. Tale mancanza impedisce l'acquisizione di dati in maniera: strutturata, standardizzata e sistematica. Le soluzioni impiegate sono molteplici e la loro differente complessità, talvolta rudimentale, evidenzia come le necessità di una mappatura centralizzata rappresentino una sfida di lunga data per l'azienda ed i suoi operatori.

Sono state catalogate, ai fini della nostra analisi, le seguenti tipologie di strumenti:

- **Documentazione Cartacea non Strutturata:** Soluzione ormai quasi obsoleta, che prevede la raccolta e la distribuzione di informazioni su supporto cartaceo. Tale soluzione, sebbene di facile utilizzo iniziale, è intrinsecamente statica, di difficile aggiornamento, non ricercabile e non consente alcuna forma di collaborazione o condivisione centralizzata dei dati.

- **Documenti Digitali Statici:** Rappresentano la digitalizzazione del modello cartaceo. Documenti come PDF o presentazioni sono facilmente condivisibili, ma ereditano molte delle criticità del cartaceo: i dati non sono strutturati, l'aggiornamento richiede la ridistribuzione di una nuova versione del file, generando problemi di *versioning*, e le funzionalità di ricerca sono limitate al testo libero.

- **Fogli di Calcolo:** Questa soluzione rappresenta un primo passo verso la strutturazione dei dati. L'organizzazione tabellare permette di definire attributi e di applicare filtri di base. Le criticità risiedono nella mancanza di garanzie sull'integrità dei dati, nella gestione problematica degli accessi concorrenti e nella difficoltà di modellare relazioni complesse o dati geospaziali in modo nativo.
- **Mappe Basate su Piattaforme Web:** Strumenti che offrono un notevole vantaggio grazie al supporto nativo per la georeferenziazione. I limiti emergono rapidamente in un contesto aziendale: opzioni di personalizzazione scarse, meccanismi di controllo degli accessi rudimentali e dati che risiedono su piattaforme di terze parti, sollevando questioni di governance, costi e conformità normativa. Ne è un esempio *Google Maps*, attualmente utilizzato per tali scopi ma con limiti nella granularità dei permessi che rendono lo strumento non fruibile dalla cittadinanza.
- **Applicativo GeoNote:** Sistema del Comune di Rimini costruito su un *Geographic Information System* (GIS) per tracciare situazioni di disagio sociale. Pur offrendo funzionalità simili a quelle desiderate, esso garantisce accesso a dati privati non condivisibili con l'AUSL ed è limitato al solo territorio comunale.

Appare evidente come nessuna delle soluzioni in uso riesca a soddisfare simultaneamente tutti i requisiti critici emersi. I documenti testuali sono inadeguati per la loro staticità; i fogli di calcolo mancano di garanzie sull'integrità dei dati e sulla gestione degli accessi; le piattaforme web di terze parti presentano limiti di *governance* e sicurezza; infine, l'applicativo GeoNote risulta inapplicabile per la sua limitata interoperabilità e il perimetro circoscritto.

Il progetto sviluppato in questa tesi si propone di colmare questo divario, sintetizzando i punti di forza delle diverse tecnologie: l'interattività di una piattaforma cartografica, il rigore di un database strutturato e la sicurezza di un sistema con controllo degli accessi basato sui ruoli. L'obiettivo è superare la logica dello strumento tattico e frammentato per fornire una piattaforma strategica, centralizzata e sicura, la cui necessità è stata dimostrata dalle criticità analizzate.

1.4 Obiettivi del Progetto

Alla luce delle criticità emerse, il presente progetto si prefigge di analizzare, progettare e prototipare una piattaforma software per costituire un punto di accesso unificato sia per gli operatori sanitari sia per i cittadini. Nello specifico, i risultati attesi sono:

- **Analizzare, progettare e implementare un'applicazione cartografica interattiva:** Sviluppare un'interfaccia utente accessibile e intuitiva per la visualizzazione georeferenziata dei punti di servizio, integrando strumenti di ricerca e filtraggio avanzato.
- **Sviluppare un sistema di gestione e *auditing* dei dati:** Fornire strumenti per la gestione continua dei dati, garantendone l'integrità e la coerenza al fine di trasformare l'applicativo nella *single source of truth* per l'offerta sanitaria territoriale. Il sistema dovrà inoltre implementare un meccanismo di auditing per tracciare ogni operazione significativa, ponendo le basi per un sistema di notifiche proattive.
- **Integrare una gestione sicura dei dati sensibili:** Progettare un modello di controllo degli accessi basato sui ruoli per gestire la visibilità differenziata delle informazioni relative ai pazienti, unificando la gestione di dati pubblici e sensibili all'interno di un'unica piattaforma sicura.

1.5 Struttura della Tesi

Il presente elaborato è organizzato per guidare il lettore attraverso tutte le fasi del progetto, dall'analisi del contesto fino alla discussione dei risultati. I capitoli successivi affronteranno:

- **Metodi e Strumenti di Sviluppo:** Descrive l'approccio metodologico Agile e il paradigma di progettazione *Domain-Driven Design* (DDD).
- **Analisi e Progettazione del Sistema:** Dettaglia i requisiti funzionali e non funzionali, i casi d'uso e i vincoli del sistema. Illustra inoltre le decisioni architetturali, il modello dati, la progettazione delle API e l'architettura del frontend.
- **Sviluppo del Prototipo:** Descrive le fasi salienti dello sviluppo prototipale, focalizzandosi sulle soluzioni ideate per rispettare al meglio i dettami progettuali, le sfide tecniche affrontate e le strategie messe in campo per una scrittura intelligente del codice.
- **Conclusioni:** Valuta criticamente il lavoro svolto attraverso la sua scomposizione nelle fasi principali. Tale passaggio permette di identificare punti di forza e di debolezza in relazione agli obiettivi discutendo così, in fine, i limiti e i possibili sviluppi futuri.

Capitolo 2

Metodi e Strumenti di Sviluppo

L'ideazione e lo sviluppo di un sistemi software complessi richiede l'utilizzo di metodologie robuste, in grado di governare il progressivo aumento di complessità dovuto alla incrementale maturazione del progetto. Questo capitolo illustra le due principali metodologie che hanno guidato il presente lavoro.

In primo luogo, per affrontare l'incertezza dei requisiti iniziali e promuovere una collaborazione efficace con gli stakeholder, è stato adottato il paradigma di sviluppo *Agile*. Questo approccio si è rivelato fondamentale per gestire un contesto dinamico e caratterizzato da grande incertezza funzionale.

Parallelamente alla gestione del processo, è stata affrontata la sfida della complessità concettuale del dominio sanitario. A tal fine, il progetto ha tratto ispirazione dai principi strategici del *Domain-Driven Design* (DDD), utilizzandoli come bussola per la fase di analisi.

2.1 Il Paradigma di Sviluppo Agile

2.1.1 Limiti dei Modelli Sequenziali

I modelli tradizionali di gestione del ciclo di vita del software, come quelli sequenziali, si basano su un approccio per fasi distinte e ordinate. Il più noto tra questi paradigmi è il modello a cascata, il quale presuppone che il processo di sviluppo possa essere scomposto in una sequenza lineare di stadi ciascuno dei quali, debba essere completato prima dell'inizio del successivo.

Questa impostazione, sebbene rigorosa, presenta due criticità fondamentali in contesti caratterizzati da elevata incertezza. In primo luogo, assume che i requisiti del sistema siano completamente noti, stabili e immutabili fin dall'inizio del progetto, un'ipotesi che si rivela spesso irrealistica. In secondo luogo, il feedback da parte degli *stakeholder* e degli utenti finali viene posticipato alla fine del ciclo di sviluppo, quando un prototipo funzionante è finalmente

disponibile. Questa dilatazione dei tempi di verifica introduce un alto rischio che il prodotto finale non sia allineato con le reali esigenze del dominio, ormai evolute.

2.1.2 Il Manifesto Agile

La risposta alle rigidità dei modelli sequenziali non è stata la creazione di un singolo modello più dinamico, ma l'emergere di un nuovo paradigma filosofico, formalizzato nel 2001 con la pubblicazione del *Manifesto per lo Sviluppo Agile del Software* [1]. Questo documento, redatto da diciassette influenti sviluppatori e metodologi, non descrive un processo, ma definisce un insieme di valori e principi fondamentali. Il cuore del Manifesto risiede nei suoi quattro valori fondanti, che rappresentano una decisa inversione di priorità rispetto all'approccio tradizionale:

- **Gli individui e le interazioni** più che i processi e gli strumenti.
- **Il software funzionante** più che la documentazione esaustiva.
- **La collaborazione con il cliente** più che la negoziazione dei contratti.
- **Rispondere al cambiamento** più che seguire un piano.

È importante prestare attenzione alla formulazione utilizzata. Il Manifesto non nega il valore di aspetti consolidati come i processi e la documentazione, ma afferma la priorità di quelli legati al dinamismo, alla collaborazione e al pragmatismo. L'intento, quindi, non è eliminare la pianificazione, ma ridimensionarne il ruolo: da prerequisiti rigidi e vincolanti a strumenti flessibili, posti al servizio dell'obiettivo primario, ovvero la produzione di valore tangibile per il cliente.

Il Manifesto enuncia inoltre dodici principi che forniscono una guida più concreta. Tra questi, emergono concetti chiave come la consegna frequente di software funzionante al termine di brevi cicli di sviluppo, l'accoglienza dei cambiamenti anche a fasi avanzate del progetto, la stretta collaborazione quotidiana tra referenti di dominio e sviluppatori, mantenendo sempre costante attenzione all'eccellenza tecnica e alla buona progettazione.

2.1.3 L'Approccio Iterativo-Incrementale

I modelli agili traducono i principi del Manifesto in un processo di sviluppo iterativo e incrementale. Pensati per mettere al centro del lavoro la collaborazione continua con il cliente, offrono la possibilità di rispondere tempestivamente alle variazioni delle esigenze, con l'obiettivo operativo di fornire software funzionante per raffinamenti e ampliamenti successivi.

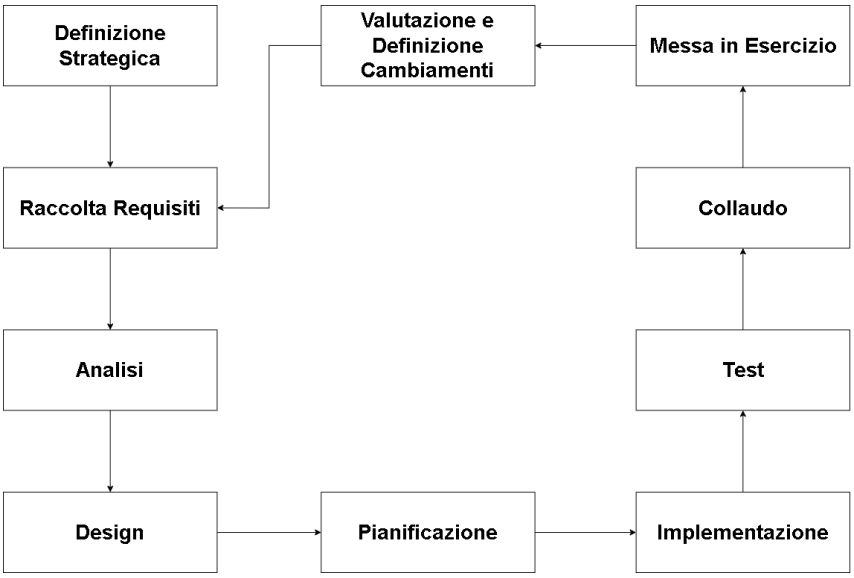

Figura 2.1: Diagramma del ciclo di sviluppo iterativo-incrementale.

Questo cambio di prospettiva si traduce operativamente in un processo di sviluppo suddiviso in cicli brevi a durata fissa, noti come iterazioni o *sprint*. Come illustrato in Figura 2.1, ogni ciclo parte da una selezione di requisiti prioritari, attraversa le fasi di progettazione, sviluppo e test, e si conclude con la produzione di un incremento di software funzionante e potenzialmente rilasciabile [2].

A conclusione di ogni ciclo interviene il meccanismo di feedback continuo, integrato attraverso incontri di revisione con gli esperti del dominio. Questo elemento, cardine del paradigma, permette di adattare e ridefinire le priorità in corso d'opera, garantendo che il prodotto finale sia costantemente allineato con le reali esigenze dell'utente.

2.1.4 Motivazioni dell'Adozione del Modello Agile

L'adozione del paradigma Agile per il presente progetto non è stata una mera scelta metodologica, ma una necessità strategica emersa dalle complesse dinamiche del contesto operativo. Le fasi preliminari di interfacciamento con gli esperti del dominio dell'AUSL hanno infatti rivelato un significativo disallineamento su aspetti fondanti del sistema da sviluppare, tra cui la definizione del pubblico di riferimento primario e l'identificazione delle funzionalità prioritarie.

In uno scenario caratterizzato da visioni così divergenti, un approccio tradizionale basato su un'analisi dei requisiti iniziale ed esaustiva, tipico dei mo-

delli a cascata, si sarebbe rivelato inefficace e controproducente. Tale processo avrebbe con ogni probabilità portato alla stesura di documenti contraddittori e a una paralisi progettuale, con un'inevitabile dissipazione delle risorse destinate al progetto.

È risultato pertanto imperativo adottare un modello iterativo che permettesse lo sviluppo rapido di un *Minimum Viable Product* (MVP). Con questo termine si intende la versione di un prodotto dotata del minimo insieme di funzionalità sufficienti a renderla utilizzabile da un primo nucleo di utenti. In questo contesto, l'MVP non è stato concepito solo come una prima versione del software, ma primariamente come uno strumento tangibile per facilitare il dialogo e guidare gli esperti verso una visione condivisa. Il prototipo funzionante ha agito come un catalizzatore, trasformando concetti astratti in funzionalità concrete e consentendo al team di progetto di utilizzare il software stesso come strumento di selezione e validazione dei requisiti. In sintesi, il modello Agile è stato l'unico approccio in grado di gestire l'incertezza iniziale, promuovendo la convergenza delle idee attraverso cicli di feedback rapidi e basati su un prodotto tangibile.

2.2 Il Domain-Driven Design

Parallelamente alla metodologia di gestione del progetto, per affrontare la complessità del dominio applicativo si è tratto ispirazione da un approccio noto come *Domain-Driven Design* (DDD). I suoi principi strategici sono stati utilizzati primariamente nella fase di analisi per guidare la comprensione della logica di business e delle regole specifiche del contesto. Il DDD non è una tecnologia, ma una filosofia di sviluppo che pone al centro della progettazione la complessità del dominio applicativo, ovvero la logica di business e le regole specifiche del contesto in cui il software opera.

2.2.1 Principi Fondamentali del DDD

L'obiettivo primario del DDD è creare un modello software che sia un'astrazione fedele del suo dominio di riferimento. Per raggiungere tale scopo, il DDD si fonda su due pilastri strategici [3]:

- **Linguaggio Ubiquo (*Ubiquitous Language*):** Si tratta della creazione di un linguaggio comune, condiviso e rigoroso, sviluppato in collaborazione tra gli esperti del dominio e il team di sviluppo. Tale linguaggio viene utilizzato in tutte le forme di comunicazione: nel codice sorgente, nei diagrammi e nella documentazione. L'adozione di un Linguaggio Ubi-

quo elimina le ambiguità e le traduzioni errate tra la logica di business e la sua implementazione tecnica.

- **Contesto Delimitato (*Bounded Context*):** Il DDD riconosce che un dominio complesso non può essere rappresentato da un unico modello unificato. Un Contesto Delimitato definisce un confine esplicito all'interno del quale un particolare modello di dominio è valido e consistente. Ogni Contesto Delimitato ha il suo Linguaggio Ubiquo e la sua logica specifica. Questo principio di separazione è fondamentale per gestire la complessità, consentendo a diverse parti del sistema di evolvere in modo indipendente.

All'interno di un Contesto Delimitato, il modello viene costruito utilizzando una serie di componenti tattici, noti come *building blocks*. Questi componenti sono gli strumenti con cui si dà forma alla logica di business in modo esplicito e manutenibile. I principali sono:

- **Entity:** Un oggetto del dominio la cui caratteristica distintiva non risiede nei suoi attributi, ma in un'identità unica e continua nel tempo. L'identità di un'Entità persiste anche se i suoi attributi cambiano.

- **Value Object:** Un oggetto definito esclusivamente dal valore dei suoi attributi, privo di un'identità propria. Due *Value Object* sono considerati uguali se i loro attributi sono identici. Sono tipicamente immutabili, ovvero per modificarne uno, se ne crea una nuova istanza.

- **Aggregate:** Un raggruppamento di Entità e Oggetti Valore correlati, trattato come un'unica unità coesa ai fini delle modifiche dei dati. Ogni Aggregato ha una radice, detta *Aggregate Root*, che è l'unica Entità accessibile dall'esterno e responsabile di garantire l'integrità e la coerenza dell'intero gruppo. Qualsiasi modifica agli oggetti interni deve passare attraverso la radice, che funge da confine transazionale.

- **Repository:** Un componente che astrae l'accesso ai dati, mediando tra il modello di dominio e la tecnologia di persistenza. Fornisce l'illusione di una collezione di Aggregati in memoria, esponendo un'interfaccia con metodi come `findById` o `save`. In questo modo, la logica di business contenuta nei servizi rimane agnostica e non inquinata da dettagli implementativi legati alla persistenza.

2.2.2 Motivazioni dell'Adozione del DDD in Fase di Analisi

La volontà di ispirarsi ai principi del *Domain-Driven Design* è nata dalla necessità di gestire l'elevata complessità del dominio sanitario territoriale. Le criticità da modellare non erano banali: gerarchie di categorie, un complesso sistema di permessi e visibilità, e la distinzione tra dati pubblici e dati sensibili richiedevano un approccio che andasse oltre la semplice mappatura dei dati su un database.

Da questo approccio sono stati tratti benefici concreti, in particolare durante la fase di analisi. L'elaborazione di un Linguaggio Ubiquo condiviso con il personale AUSL si è rivelata cruciale per superare le ambiguità terminologiche, presenti non solo nella comunicazione tra il team tecnico e gli esperti, ma anche tra i diversi dipartimenti interni all'azienda. La formalizzazione terminologica ha garantito una modellazione puntuale e ha reso il modello software concettualmente accessibile anche agli esperti del dominio privi di competenze informatiche. Il DDD, dunque, ha fornito gli spunti metodologici per tradurre un dominio di business ricco e articolato in un modello software robusto, manutenibile e fedele alle esigenze degli operatori.

Capitolo 3

Analisi e Progettazione del Sistema

Il lavoro di creazione dell'applicativo ha preso avvio con una fase iniziale di analisi, condotta attraverso una serie di incontri telematici con gli esperti del dominio e alcuni responsabili tecnici esterni all'AUSL, coinvolti per fornire assistenza riguardo ad alcuni dei sistemi preesistenti. Tale processo è iniziato parallelamente al lavoro, interno all'azienda, volto a valutare la necessità dello strumento stesso. Il contesto, data l'assenza di un'idea unificata tra gli stakeholder, non permetteva un'analisi capillare con gli operatori; ha richiesto, al contrario, un lento lavoro di astrazione affiancando gli stessi operatori nei colloqui interni per la valutazione del progetto.

Fin da subito è apparso evidente come la vastità del dominio e l'eterogeneità delle esigenze imponessero l'adozione di un modello Agile. Tale scelta si è resa necessaria per mitigare le complesse dinamiche emerse. Prima tra tutte l'impossibilità del cliente di fornire una visione unificata delle necessità a cui lo strumento avrebbe dovuto far fronte. Data l'adozione del modello Agile, non è stato necessario disporre di un'analisi organica e completamente strutturata del dominio fin dal principio. Al contrario è stato possibile procedere per raffinamenti successivi, migliorando progressivamente la conoscenza del dominio stesso.

3.1 Analisi del Dominio Applicativo

La fase iniziale di analisi del dominio è stata caratterizzata da un'elevata complessità, dovuta principalmente alla diversità degli stakeholder coinvolti. Il primo confronto, aperto a una moltitudine di referenti dei dipartimenti AUSL, ha rivelato visioni differenti e talvolta contraddittorie, rendendo difficile l'estrapolazione di un modello unificato. Da questo primo incontro è emer-

sa un'unica esigenza condivisa: la necessità di una mappatura topografica di elementi georeferenziati, dotati di attributi per il filtraggio.

Un secondo e fondamentale passaggio di analisi ha permesso di unificare la visione e di delineare i pilastri del dominio. È emersa la necessità di un sistema flessibile, capace di tracciare entità georeferenziate, gestire la loro cronologia di modifica e consentire un filtraggio capillare sia su base semantica sia geografica.

Questa fase esplorativa ha evidenziato il valore di alcuni principi strategici del *Domain-Driven Design* per governare la complessità. Si è quindi deciso di adottarne lo strumento più efficace per la fase di analisi: la formalizzazione della conoscenza del dominio attraverso la definizione di un Linguaggio Ubiquo.

3.1.1 Definizione del Linguaggio Ubiquo

Per superare le ambiguità emerse e creare un modello software fedele alla realtà applicativa, è stato sviluppato un Linguaggio Ubiquo in collaborazione con gli esperti del dominio. Questo vocabolario condiviso costituisce la base per tutta la comunicazione, la documentazione e l'implementazione del software. I termini chiave del dominio sono definiti nella Tabella 3.1.

Tabella 3.1: Definizione dei termini del Linguaggio Ubiquo.

Termine	**Definizione**	**Sinonimi**
Punto della Salute	L'entità centrale del sistema. Rappresenta un qualsiasi luogo, servizio o risorsa fisica di interesse per l'AUSL, caratterizzato da una posizione geografica e da un insieme di attributi descrittivi.	• HealthPoint • Struttura • Punto Sanitario • Punto
Servizio	Un'offerta o prestazione specifica fornita presso un Punto della Salute. Un singolo Punto può erogare più Servizi.	• Service • Prestazione • Offerta
Categoria	Strumento di classificazione gerarchica per Punti della Salute e Servizi. Ogni categoria può avere una categoria genitore.	• Category • Classificazione • Tipologia
Distretto Sanitario	Una suddivisione territoriale e amministrativa dell'AUSL che rappresenta un'area geografica specifica di competenza.	• Department • Zona

Tabella 3.1 – segue dalla pagina precedente

Termine	**Definizione**	**Sinonimi**
Paziente	Individuo destinatario di assistenza sanitaria, a cui sono associate informazioni sensibili soggette a stringenti normative sulla privacy.	• Assistito • Utente del SSN • Caso
Permesso	Un'autorizzazione granulare per eseguire una specifica azione su una risorsa contenuta nel dominio.	• Permission • Autorizzazione • Diritto • Privilegio
Ruolo	Un insieme di permessi che definisce un profilo di autorizzazione.	• Role • Gruppo
Utente	Rappresenta un operatore del sistema. Entità a cui sono associati ruoli e permessi che ne determinano le capacità operative.	• Profile • Account
Visibilità	Attributo che definisce il livello di accesso a una risorsa. Può essere Pubblico, Privato o Ristretto.	• Livello di accesso • Scope
Modifica	Rappresenta un evento articolato in proposta ed accettazione o rigetto di alterazione di un Punto della Salute, garantendo la storicità e la tracciabilità delle informazioni.	• ModificationLog • Aggiornamento • Revisione
Evento di Dominio	Rappresenta un fatto significativo che si è verificato all'interno del dominio.	• Event • Trigger
Notifica	Un messaggio generato dal sistema per informare un Utente su un Evento di Dominio rilevante.	• Notification • Avviso • Messaggio

3.1.2 Componenti Logici Principali del Dominio

L'analisi ha portato a identificare due principali aree concettuali, ispirandosi all'enfasi che il *Domain-Driven Design* pone sui *Bounded Context*. Essi non rappresentano sottodomini destinati a una separazione fisica come nei microservizi, ma di componenti logici coesi, ciascuno con responsabilità ben definite.

Questi due componenti nascono dalle entità centrali del dominio: il Punto della Salute, che rappresenta il "cosa", e l'Utente, che rappresenta il "chi". Tutto il resto del modello di dominio serve ad arricchire e a contestualizzare queste due entità primarie. Data la loro centralità, è proprio l'interazione tra questi due componenti a definire la natura delle funzionalità del sistema.

Il Punto della Salute

Il primo fulcro concettuale si sviluppa attorno all'entità del Punto della Salute, che rappresenta l'oggetto centrale del dominio, il "cosa". L'analisi si è concentrata non solo sulla singola entità, ma su tutto l'ecosistema di informazioni che le conferisce significato. Un Punto, infatti, non è un dato isolato, ma esiste in un ricco contesto definito da:

- **Classificazione**, attraverso l'associazione a una o più *Categorie* gerarchiche che ne definiscono la natura.
- **Funzionalità**, tramite i *Servizi* specifici che è in grado di erogare.
- **Posizione**, l'attributo geografico che lo colloca fisicamente sul territorio.

Questo insieme di concetti interconnessi costituisce il dominio statico dei dati. L'obiettivo dell'analisi è stato quello di rispondere a domande fondamentali come: "Qual è la struttura minima di informazioni per descrivere un Punto della Salute in modo efficace?" e "Come modellare le relazioni gerarchiche e funzionali che lo legano ad altre entità?".

L'Utente

Il secondo fulcro concettuale è costruito attorno alla figura dell'Utente, che rappresenta l'agente attivo del sistema, il "chi". L'analisi di questo aspetto non si è focalizzata tanto sull'identità dell'operatore, quanto sulle sue capacità e sulle conseguenze delle sue azioni. Il modello definisce l'interazione dell'Utente con i dati attraverso:

- **Ruoli**, definiti da un insieme di *Permessi* granulari che specificano le singole azioni consentite.
- **Conseguenze**, rappresentate dagli *Eventi* che vengono scatenati dalle sue azioni e che possono dare origine a *Notifiche* per altri utenti.

Questo insieme di concetti costituisce il dominio dinamico delle azioni. Il focus dell'analisi è stato quello di definire le regole di business che governano l'accesso e la manipolazione delle informazioni, rispondendo a domande come: "Chi può

fare cosa?", "Come garantire che ogni azione sia tracciata e autorizzata?" e "Cosa deve accadere nel sistema in risposta a un'azione dell'utente?".

Questi due fulcri del dominio, il "cosa" e il "chi", pur essendo concettualmente distinti, non sono isolati ma al contrario, la loro interdipendenza è fondamentale per il corretto funzionamento del sistema. Il modello del Punto della Salute definisce le risorse informative, mentre il modello dell'Utente stabilisce le regole per interagire con esse. La relazione fondamentale tra i due può essere descritta in questo modo: la capacità di un *Utente* di visualizzare o manipolare un *Punto della Salute* è direttamente governata dai *Ruoli* e dai *Permessi* che gli sono stati assegnati. Questa interdipendenza logica è il pilastro su cui si fondano i requisiti di sicurezza e di accesso differenziato, garantendo che ogni operatore agisca esclusivamente entro i confini delle proprie autorizzazioni.

3.2 Analisi dei Requisiti

In linea con l'approccio Agile, la raccolta dei requisiti ha seguito un percorso evolutivo, specchio delle dinamiche del progetto. In una fase iniziale, caratterizzata da un'elevata incertezza da parte dell'azienda committente, ci è stato possibile partecipare come osservatori alle riunioni interne dell'azienda. Questo ha permesso di assorbire passivamente la conoscenza del dominio, mentre gli *stakeholder* stessi definivano il perimetro delle necessità.

Il ruolo di osservatori passivi è stato poi abbandonato. Una volta acquisita una sufficiente padronanza del dominio, è diventato fondamentale infatti svolgere un ruolo di facilitazione tecnica, capace di intravedere le conversazioni potenzialmente più proficue e di indirizzarle verso esiti concreti e attuabili. Questo ruolo, seppur non decisionale, ha permesso di iniziare ad abbozzare i primi requisiti, risolvendo le iniziali ambiguità e contraddizioni emerse.

Il contributo dell'Ing. Angelo Croatti, Responsabile per la Transizione Digitale di AUSL Romagna, è stato cruciale per trasformare le discussioni ad alto livello in specifiche più concrete. Grazie alla documentazione redatta in tali occasioni è stato possibile delineare con chiarezza i requisiti estratti in precedenza, studiarli e procedere alle prime fasi implementative.

A partire dal primo rilascio dell'MVP, il processo di raccolta dati è cambiato radicalmente, diventando più strutturato e pragmatico: il software stesso è diventato lo strumento per l'elaborazione di feedback mirati, permettendo di affinare e incrementare le funzionalità sulla base di riscontri tangibili. Gli stakeholder sono passati così dal supporre funzionalità sulla base della loro immaginazione a proporre modifiche su componenti e aspetti tangibili ed oggettivi. Questo passaggio è risultato fondamentale per molteplici ragioni:

anzitutto ha avvicinato i pensieri più divergenti all'interno del team di esperti, fornendogli una base comune di ragionamento basata sull'esistenza di un prodotto, ed ha reso i riscontri mirati alla modifica di qualcosa di tangibile e non più alla produzione di qualcosa di inesistente.

Il volume e la specificità dei feedback emersi in questa fase hanno reso necessaria la centralizzazione della raccolta dei dati per mezzo di un canale univoco. Per formalizzare questo canale di comunicazione, sono state infine introdotte interviste strutturate. La loro somministrazione agli esperti e la successiva sintesi dei risultati è stata gestita dalla Dott.ssa Melissa Corradi, Dirigente Medico di AUSL Romagna. Questo ha permesso di istituire un canale di comunicazione autorevole e unificato, garantendo che i requisiti raccolti attraverso cicli iterativi fossero validati, coerenti e rappresentativi di una visione condivisa. Tutto ciò ha portato all'affinamento progressivo dei requisiti di seguito descritti.

3.2.1 Requisiti Funzionali

I Requisiti Funzionali (RF) descrivono le funzionalità che il sistema deve fornire per rispondere alle esigenze operative e informative degli esperti del dominio.

RF-1 **Gestione dei Punti della Salute:** il sistema deve consentire le operazioni CRUD (Create, Read, Update, Delete) sui punti di interesse georeferenziati.

RF-2 **Gestione dei Servizi:** ogni punto può erogare uno o più servizi; il sistema deve permetterne la definizione, modifica e associazione ai punti.

RF-3 **Classificazione gerarchica:** i punti e i servizi devono poter essere organizzati in categorie e sottocategorie, con strutture gerarchiche annidate.

RF-4 **Gestione dei livelli di visibilità e persistenza:** ciascun punto deve essere contrassegnato come pubblico, privato o riservato, determinando i limiti di accesso alle informazioni. Deve essere inoltre possibile inserire punti con un arco temporale di vita predefinito.

RF-5 **Gestione utenti e ruoli:** gli utenti devono essere profilati con ruoli che a loro volta possiedono permessi specifici, in modo da regolare le azioni consentite.

RF-6 **Ricerca e filtraggio:** il sistema deve consentire ricerche testuali e filtri avanzati per categoria, fascia d'età, area geografica, livello di visibilità, ecc.

RF-7 **Mappatura geografica interattiva:** l'interfaccia deve fornire una visualizzazione su mappa dei punti, con possibilità di selezione, zoom e filtraggio.

RF-8 **Gestione multi-livello territoriale:** il sistema deve supportare la visualizzazione e l'analisi dei dati a livello di quartiere, comune.

RF-9 **Integrazione con dati AUSL:** il sistema deve poter recuperare e aggiornare automaticamente le informazioni pubblicate sul sito dell'AUSL.

RF-10 **Caricamento dati da banche dati esistenti:** devono poter essere importati dataset preesistenti o esterni in formati strutturati.

RF-11 **Tracciamento delle modifiche:** ogni modifica a un punto o servizio deve essere storicizzata, mantenendo traccia dell'autore, del timestamp e dei campi modificati.

RF-12 **Notifiche:** il sistema deve inviare notifiche automatiche agli utenti in caso di eventi rilevanti che essi siano sincroni o asincroni.

RF-13 **Accesso pubblico:** i dati non riservati devono essere consultabili pubblicamente tramite un'interfaccia web accessibile.

RF-14 **Espandibilità funzionale:** il sistema deve prevedere l'aggiunta di nuove categorie, servizi o tipi di punto senza necessità di modifiche strutturali.

3.2.2 Requisiti Non Funzionali

I Requisiti Non Funzionali (RNF) definiscono le proprietà qualitative del sistema, ovvero le caratteristiche che ne determinano la solidità, l'efficienza e la sostenibilità nel tempo.

RNF-1 **Scalabilità:** il sistema deve poter gestire un progressivo incremento di dati e utenti, garantendo prestazioni adeguate.

RNF-2 **Flessibilità:** l'architettura deve consentire modifiche e ampliamenti senza impatti significativi sulle funzionalità esistenti.

RNF-3 **Interoperabilità:** il sistema deve rispettare standard di scambio dati e protocolli di interoperabilità con piattaforme sanitarie.

RNF-4 **Tracciabilità:** ogni operazione significativa deve essere registrata, garantendo la possibilità di audit e controllo successivo.

RNF-5 **Sicurezza:** devono essere implementati meccanismi di autenticazione, autorizzazione e cifratura dei dati, nel rispetto del GDPR.

RNF-6 **Usabilità:** l'interfaccia deve essere intuitiva, coerente con standard grafici noti e facilmente utilizzabile da personale non tecnico.

RNF-7 **Manutenibilità:** il codice deve essere documentato e strutturato in modo da agevolare interventi futuri di manutenzione o estensione.

3.3 Progettazione del Sistema

La fase di progettazione traduce i requisiti funzionali e non funzionali, emersi dall'analisi, in una struttura architetturale concreta. Attraverso la modellazione del sistema a diversi stadi di astrazione, formalizzata da diagrammi Entità-Relazione (E/R) e diagrammi UML, è possibile definire con capillarità crescente la struttura della soluzione. L'obiettivo è definire un'architettura robusta, scalabile e manutenibile, che risponda efficacemente alle complesse esigenze del dominio. Le scelte progettuali sono state guidate da pattern consolidati per lo sviluppo di applicazioni web moderne, seguendo un approccio top-down: dall'architettura generale fino al dettaglio dei singoli componenti.

3.3.1 Progettazione Architetturale

Lo studio dell'architettura è iniziato non appena i requisiti fondanti del dominio sono stati sufficientemente chiari da permettere l'individuazione dei componenti strutturali del progetto. Il sistema è stato progettato seguendo un'architettura a tre livelli solitamente detto *Three-Tier Architecture*, un modello consolidato che garantisce una rigorosa separazione delle responsabilità tra i componenti logici dell'applicazione.

Questa scelta strategica risponde direttamente ai requisiti non funzionali di manutenibilità e flessibilità. Disaccoppiando i livelli, si ottiene la facoltà di far evolvere, o persino sostituire, le tecnologie di un singolo livello senza impattare sugli altri. Ciò garantisce non solo la possibilità di ampliamento futuro, ma anche la migrazione verso scelte tecnologiche più innovative o versatili.

I tre livelli sono:

- **Presentation Tier:** Rappresenta il punto di contatto tra l'utente e il sistema. La sua responsabilità primaria è quella di tradurre i dati e le funzionalità del sistema in un'interfaccia utente interattiva e comprensibile. Questo livello si occupa di presentare le informazioni, acquisire l'input dell'utente attraverso moduli e controlli, e fornire un feedback immediato, eseguendo validazioni preliminari a livello client. Non contiene logica di business, ma agisce come un interprete tra il mondo umano e quello digitale.

- **Application Tier:** Costituisce il cuore logico del sistema, operando come un cervello centrale. Questo livello è l'unico custode delle regole di business e della logica di dominio. Le sue responsabilità includono l'elaborazione delle richieste provenienti dal livello di presentazione, l'orchestrazione di processi complessi, la gestione della sicurezza attraverso autenticazione ed autorizzazione e l'applicazione di tutte le policy che garantiscono il corretto funzionamento del sistema. Espone le sue capacità attraverso un'interfaccia di servizi ben definita, che funge da unico punto di accesso controllato alla logica applicativa.

- **Data Tier:** Funge da memoria a lungo termine del sistema. La sua responsabilità non si limita alla semplice memorizzazione dei dati, ma si estende a garantirne l'integrità, la coerenza e la sicurezza nel tempo. Questo livello gestisce l'accesso fisico e logico ai dati, assicura che le relazioni tra le entità siano mantenute e fornisce uno strumento di interrogazione per la ricerca e la manipolazione dei dati. È inoltre responsabile della gestione delle transazioni per assicurare che le operazioni sulle informazioni siano atomiche e affidabili.

3.3.2 Progettazione del Database

La progettazione del database ha preso avvio dalla traduzione del modello concettuale del dominio in un modello logico relazionale, formalizzato attraverso un diagramma E/R. Tale sfida, per quanto fondamentale ai fini della progettazione, ha rappresentato un passaggio tutt'altro che banale. La criticità prima e fondante è stata l'assenza di un modello definito. Ciò ha reso complessa sia l'estrapolazione dei dati utili a modellare le singole entità, sia le relazioni tra le stesse. L'assenza, infatti, di limpidità a livello dei requisiti ha reso arduo immaginare la struttura che dunque, inizialmente, ha assunto una forma estremamente articolata. Ancora una volta il lavoro di raffinamento capillare svolto nel corso delle iterazioni agili ha fornito la possibilità di affinare il modello attraverso l'eliminazione di ridondanze ed elementi con grado di complessità eccedentemente elevato. Il lavoro si è svolto a partire dai due noccioli

semantici di interesse, quali Punti della Salute e Utenti. Successivamente, con un lavoro ad anelli concentrici, è stato possibile allargarsi alla progettazione di tutti quegli elementi capaci di arricchire semanticamente i due noccioli, assicurando così un uso parsimonioso delle risorse ed una conformità alle specifiche richieste.

Punto della Salute

La progettazione dell'entità `Punto della Salute` ha rappresentato il fulcro della modellazione del database. La sua centralità è confermata dal fatto che il suo schema è stato oggetto di continue revisioni durante tutto il processo iterativo, evolvendo fino alle fasi finali dello sviluppo.

Una delle prime sfide è stata quella di catturare la natura duale di questa entità. L'analisi esplorativa ha infatti rivelato che il termine Punto della Salute poteva essere usato per descrivere due concetti distinti: da un lato, presidi fisici che erogano servizi diretti al cittadino, come per esempio un punto prelievi; dall'altro, strutture con funzioni puramente burocratiche o rappresentative, come le sedi amministrative di un distretto.

Per gestire questo binomio, la prima ipotesi di lavoro prevedeva una modellazione gerarchica basata su una specializzazione. Si ipotizzò una superclasse generica, che possiamo definire `Luogo di Interesse Sanitario`, specializzata in due sottoclassi distinte: la `Sede Amministrativa`, per rappresentare le entità con funzione organizzativa, e il `Punto di Erogazione`, per identificare i luoghi in cui vengono effettivamente forniti servizi sanitari. La logica dietro questa scomposizione era quella di semplificare l'esperienza per i due principali profili di utente: gli operatori interni, spesso interessati a interagire con le sedi amministrative, e i cittadini, la cui ricerca è focalizzata sui punti di erogazione di servizi concreti. Dal punto di vista del modello E/R, questa ipotesi si sarebbe tradotta in una gerarchia con specializzazione totale e disgiunta: ogni `Luogo di Interesse Sanitario` sarebbe stato necessariamente o una `Sede Amministrativa` o un `Punto di Erogazione`, ma mai entrambi. Inoltre, si prevedeva che una `Sede Amministrativa` potesse agire come entità dirigenziale, gestendo uno o più `Punti di Erogazione` ad essa collegati.

I riscontri ottenuti successivamente hanno reso evidente che la modellazione così ideata raggiungesse un grado di capillarità eccessivo e non attinente alle necessità funzionali. Si è dunque proceduto a smantellare la gerarchia. La modellazione del `Punto della Salute` è diventata monolitica e ci si è concentrati sull'arricchimento semantico per mezzo di attributi ed associazioni con entità apposite, come ci sarà possibile vedere di seguito nel capitolo.

Il modello finale parte dunque da una struttura di base, illustrata in Figura 3.1, che si concentra sugli attributi intrinseci dell'entità. Le associazioni

HEALTH_POINT		
Integer	id	PK
String	name	
JSONB	address	
Double	latitude	
Double	longitude	
String	phone	
String	email	
String	pec_email	
String	website	
JSONB	schedule	
Text	notes	
String	visibility	
Timestamp	expires_at	
Boolean	active	
Integer	municipality_id	FK
Integer	owner_profile_id	FK
Integer	owner_department_id	FK

Figura 3.1: Entità che modella nel diagramma E/R il Punto della Salute.

con le altre entità e gli attributi più specifici verranno descritte nei paragrafi successivi.

Categoria

L'entità `Categoria` rappresenta un elemento fondante dell'architettura dei dati, in quanto costituisce il meccanismo primario per le funzionalità di filtraggio e classificazione. Lo studio di questo concetto ha portato alla modellazione di un'unica entità in relazione sia con i `Punti della Salute` sia con i Servizi.

Questa scelta progettuale è stata dettata da un principio di efficienza e coerenza. Poiché una moltitudine di categorie è applicabile in modo paritetico sia alle strutture sia alle prestazioni, un modello unificato evita la duplicazione di entità e garantisce un sistema di classificazione omogeneo in tutto il dominio.

Una funzionalità fondamentale da poter implementare è la capacità di creare gerarchie di categorie, ovvero avere la possibilità di creare e visualizzare categorie e sottocategorie ad esse annesse. A tale scopo è stata definita una relazione ricorsiva sull'entità `Categoria` stessa, facendo uso del pattern noto come *Adjacency List* come mostrato in Figura 3.2. In questo schema, ogni istanza di Categoria può avere un riferimento opzionale a un'altra istanza della stessa entità, che funge da genitore. Questo realizza una relazione uno-a-molti padre-figlio, permettendo di costruire alberi di classificazione di profondità ar-

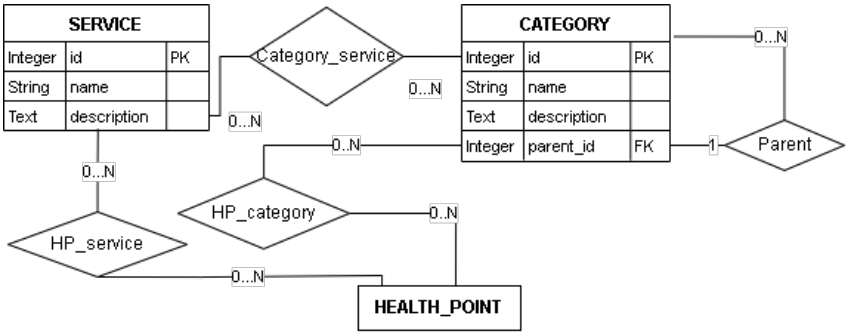

Figura 3.2: Diagramma E/R della categorizzazione.

bitraria, modellando così in modo efficace la struttura indentata tipica di un sistema di categorizzazione complesso.

Georeferenziazione

La georeferenziazione dei `Punto della Salute` è stata modellata non come un singolo attributo, ma come un sistema composito che integra diversi livelli di informazione per rispondere a esigenze sia funzionali che di usabilità.

In primo luogo, è stata definita una gerarchia amministrativa territoriale attraverso la creazione delle entità `Provincia` e `Comune`. Queste sono legate da una relazione uno-a-molti: ogni `Provincia` è associata a una moltitudine di `Comuni`, e ogni `Punto della Salute` è localizzato in un unico `Comune`. Questa struttura offre due vantaggi strategici:

- **Abilita filtri gerarchici:** Permette agli utenti di effettuare ricerche e aggregazioni di dati a livello comunale e provinciale in modo efficiente e consistente.
- **Garantisce scalabilità:** Il modello può essere facilmente esteso per includere livelli territoriali superiori permettendo al sistema di scalare a livello regionale o nazionale secondo le necessità future.

In secondo luogo, per la localizzazione puntuale, l'entità `Punto della Salute` è stata arricchita con gli attributi `latitudine` e `longitudine`. Queste coordinate numeriche sono essenziali per il sistema, in quanto permettono la visualizzazione precisa del punto su una mappa interattiva, il calcolo delle distanze e l'esecuzione di interrogazioni spaziali complesse.

Infine, per garantire la comprensibilità da parte dell'utente finale, è stato incluso un attributo testuale `indirizzo`. Questo fornisce una rappresentazione della posizione in linguaggio naturale, familiare e immediatamente interpretabile.

Questi tre elementi ovvero: la gerarchia amministrativa, le coordinate geometriche e l'indirizzo testuale lavorano in sinergia per fornire una modellazione della posizione completa, robusta e flessibile.

Gestione della Visibilità

Un requisito fondamentale, emerso durante i cicli di sviluppo agile, è stata la necessità di gestire la visibilità dei dati a un livello granulare. L'introduzione di questo concetto ha richiesto una parziale ristrutturazione del modello per garantire un controllo degli accessi flessibile e sicuro.

La soluzione individuata si basa su un approccio a due livelli. In primo luogo, all'entità `Punto della Salute` è stato aggiunto un attributo `visibilità`, che funge da regola di accesso generale e può assumere tre valori:

- **Pubblico:** Il punto è visibile a tutti gli utenti del sistema, inclusi quelli non autenticati.
- **Privato:** Il punto è visibile unicamente al personale interno all'azienda.
- **Ristretto:** La visibilità è limitata a un elenco specifico di utenti autorizzati esplicitamente dal proprietario dello stesso Punto.

Per gestire quest'ultimo caso, è stata introdotta unaspecifica entità associativa, ovvero `Autorizzazione Accesso Punto`. Questa tabella mette in relazione un `Punto della Salute` con visibilità ristretta a uno o più `Utenti`, specificando quali operatori sono autorizzati alla sua visualizzazione. In questo modo, l'attributo `visibilità` agisce come un selettore di *policy*, mentre la tabella `Autorizzazione Accesso Punto` implementa una vera e propria *Access Control List* (ACL) a livello di singolo record, fornendo il massimo grado di granularità nel controllo degli accessi.

Gestione del Ciclo di Vita del Dato

Oltre alla visibilità, è stato necessario modellare la validità temporale di un `Punto della Salute`. Invece di prevedere un'eliminazione fisica dei dati, che comporterebbe una perdita di informazioni storiche, si è optato per un approccio di tipo *soft-delete* ovvero di eliminazione logica.

Per implementare questo pattern, è stato aggiunto all'entità un attributo opzionale `Data di Scadenza`. Se all'attributo viene assegnata una data, il punto rimane valido fino a tale data, dopodiché viene considerato scaduto e filtrato dalle visualizzazioni operative standard. Qualora invece non fosse inserita alcuna data il punto sarebbe considerato permanentemente attivo. Questo meccanismo garantisce che nessun dato venga mai perso pur mantenendo pulita e aggiornata l'interfaccia utente.

Utente

L'entità Utente costituisce il secondo fulcro concettuale del dominio, rappresentando l'agente attivo del sistema. Lo studio dedicato alla sua modellazione è stato capillare e ha portato a una soluzione che bilancia la semplicità anagrafica con la complessità delle interazioni. La progettazione si è sviluppata su due livelli distinti: la definizione della sua identità e la modellazione del suo ruolo operativo.

A livello di attributi intrinseci, la modellazione dell'entità `Utente` è volutamente lineare e pragmatica. La sua struttura è stata progettata per raccogliere le informazioni strettamente necessarie all'identificazione univoca dell'operatore e alla gestione dei suoi contatti, senza introdurre complessità superflue.

La vera complessità progettuale, tuttavia, non risiede negli attributi, bensì nel modo in cui l'entità `Utente` si relaziona con il resto del dominio. Il suo ruolo non è quello di un'entità statica, ma di un attore le cui capacità operative sono definite dalle sue associazioni. Il focus dell'analisi si sposta quindi dalla sua identità alle sue autorizzazioni, che vengono modellate attraverso le entità `Ruolo` e `Permesso`, come verrà descritto di seguito.

Autorizzazione

Per tradurre la complessa struttura organizzativa aziendale in un sistema di autorizzazioni sicuro e flessibile, è stato adottato il modello di controllo degli accessi basato sui ruoli, noto come *Role-Based Access Control* (RBAC). Questo pattern disaccoppia l'identità dell'utente dalle autorizzazioni specifiche, garantendo una gestione centralizzata e scalabile dei permessi. La progettazione si articola su tre livelli concettuali: il Permesso, il Ruolo e il Dipartimento.

Alla base del sistema si trova l'entità `Permesso`, che rappresenta il diritto di eseguire una singola e specifica operazione all'interno del sistema. La modellazione di permessi così granulari permette di definire con precisione assoluta il perimetro operativo di ogni azione.

Il secondo livello è rappresentato dall'entità `Ruolo`. Un Ruolo non è altro che un insieme di `Permessi`, raggruppati per definire una funzione lavorativa o un profilo specifico. Questa astrazione è cruciale; invece di assegnare decine di permessi individuali a ogni utente, si assegna un unico `Ruolo`, semplificando drasticamente la gestione e garantendo coerenza. Un `Utente` può essere associato a più `Ruoli`, ereditando l'unione di tutti i `Permessi` in essi contenuti, permettendo così di modellare profili con responsabilità trasversali.

Infine, l'entità `Dipartimento` introduce un ulteriore livello di contestualizzazione. Ogni `Utente` è associato a un Dipartimento che rappresenta la sua collocazione all'interno dell'organigramma aziendale. Sebbene non faccia par-

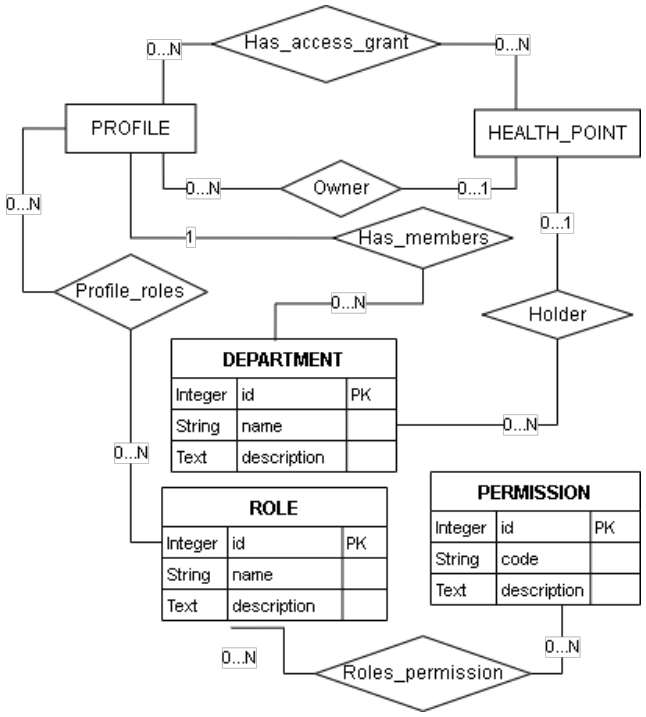

Figura 3.3: Diagramma E/R della gestione RBAC dei permessi di accesso.

te strettamente della gerarchia RBAC, il Dipartimento fornisce un contesto essenziale per future *policy* di accesso ai dati. Permette, ad esempio, di implementare regole in cui un operatore può modificare solo i `Punti della Salute` di competenza del proprio dipartimento, aggiungendo così una dimensione di controllo basata sulla proprietà e sulla pertinenza dei dati. Tutto ciò prende forma nel diagramma E/R rappresentato in Figura 3.3.

Notifica

Per trasformare l'applicazione da un semplice repository di dati a una piattaforma collaborativa e reattiva, è stata progettata un'architettura di notifica basata sul concetto di Evento. Questo approccio garantisce che gli utenti siano informati in tempo reale sulle modifiche e sugli avvenimenti rilevanti, promuovendo la consapevolezza e la collaborazione.

La progettazione si fonda su una netta separazione di responsabilità tra la registrazione di un fatto e la sua comunicazione. Ogni azione significativa che avviene all'interno del sistema, l'approvazione di una modifica o la revoca di un permesso genera un record immutabile: l'`Evento`. Questa entità non è un semplice messaggio, ma una registrazione formale e strutturata di un fatto avvenuto. L'approccio di creare un log immutabile di eventi si ispira a pattern

architetturali reattivi e garantisce un completo disaccoppiamento tra i produttori di eventi ovvero i componenti che eseguono l'azione e i loro consumatori che nel nostro caso è il sistema di notifiche. Il produttore, ovvero il componente che genera l'evento, non arresta il proprio operato in attesa di una reazione, ma delega al consumatore il compito di agire in modo asincrono. L'`Evento`, tuttavia, rappresenta il fatto accaduto, non il messaggio da comunicare. Per questo, è stata introdotta l'entità `Notifica`, la cui responsabilità è quella di tradurre un evento grezzo in un messaggio significativo e contestualizzato per l'utente finale. Un singolo `Evento` di Dominio può dare origine a zero, una o molteplici `Notifiche`, a seconda delle regole di business. Ad esempio, la modifica di un `Punto della Salute` potrebbe generare una `Notifica` per il proprietario del `Punto` e una diversa `Notifica` per gli amministratori di sistema.

Infine, il sistema modella esplicitamente la relazione tra una `Notifica` e i suoi destinatari. La progettazione prevede un meccanismo di distribuzione che associa una singola `Notifica` a uno o più `Utenti`. Per ogni coppia `Utente-Notifica`, il sistema traccia l'avvenuta lettura, permettendo di costruire un'interfaccia utente funzionale simile a un centro notifiche o a una casella di posta in arrivo.

Questa architettura disaccoppiata, che separa l'evento dalla notifica e dalla sua consegna, garantisce non solo la tracciabilità completa delle azioni, ma anche una notevole flessibilità. Permette di modificare le regole di notifica o di introdurre nuovi canali di comunicazione in futuro, senza dover alterare la logica di business che genera gli eventi fondamentali del dominio.

Modifica

La gestione del ciclo di vita dei dati non si esaurisce con la loro creazione o cancellazione logica, ma deve includere un controllo rigoroso sul processo di aggiornamento, un requisito fondamentale per garantire l'affidabilità delle informazioni. Per rispondere a questa esigenza, si è adottato il *Proposal Pattern*, che evita la manipolazione diretta dei dati attraverso un'entità dedicata, `Modifica`, la quale agisce come strumento di storicizzazione per le alterazioni non ancora validate.

L'implementazione di questo pattern prevede che, invece di alterare istantaneamente un record, ogni cambiamento venga incapsulato in un'istanza di `Modifica`. Questa entità agisce come una proposta di modifica con un proprio ciclo di vita. Essa viene generata in uno stato iniziale di attesa (`PENDING`) e non altera il dato reale finché non riceve la supervisione di un utente autorizzato. Quest'ultimo può esaminare la proposta e decidere se approvarla (`APPROVED`), rendendo la modifica effettiva e persistente, o rifiutarla (`REJECTED`), scartando l'operazione ma conservandone traccia.

Per ogni proposta, il sistema registra un insieme completo di metadati, tra cui l'autore, l'eventuale revisore, i *timestamp* di ogni fase del processo e, soprattutto, un dettaglio strutturato di tutti i campi oggetto dell'alterazione. Questo design garantisce non solo una cronologia completa e immutabile, ma istituisce un *workflow* di approvazione formale, essenziale per mantenere l'integrità e l'accuratezza dei dati in un contesto collaborativo.

E/R Completo

La sintesi delle scelte progettuali descritte nei paragrafi precedenti è formalizzata nel diagramma Entità-Relazione completo del database, illustrato in Figura 3.4. Questo schema visualizza tutte le entità, i loro attributi chiave e le relazioni che le legano, offrendo una visione d'insieme dell'architettura dei dati che supporta l'intera applicazione.

3.3.3 Progettazione del Backend

La progettazione del backend traduce la struttura logica dell'*Application Tier* in un'architettura software concreta. L'architettura interna segue rigorosamente il pattern *Layered Architecture*. Esso articola il componente in livelli logici coesi e a basso accoppiamento, ognuno con un compito nettamente definito. I Layer tipicamente identificati sono: il *Repository Layer*, il *Service Layer* e il *Controller Layer*.

Repository Layer

Il livello più interno è il *Repository Layer*, strutturato sulla base del *Repository Pattern*. Questo strato ha la responsabilità esclusiva di comunicare con il database, astraendo completamente la tecnologia di persistenza sottostante. Ogni entità *Java Persistence API* (JPA) è associata a un *repository* specifico, che espone un'interfaccia con metodi per la manipolazione dei dati.

Questa astrazione è una delle scelte più potenti dell'architettura poiché nasconde la complessità delle interrogazioni al database, permettendo al *Service Layer* di interagire con la persistenza attraverso un vocabolario di alto livello. Di conseguenza, la logica di business rimane pulita e completamente svincolata rispetto alla tecnologia di memorizzazione, libera dunque di evolvere in maniera totalmente indipendente.

Service Layer

Il componente cardine dell'applicazione risiede nel *Service Layer*, custode di tutta la logica di business e delle regole di dominio. Questo livello orchestra le

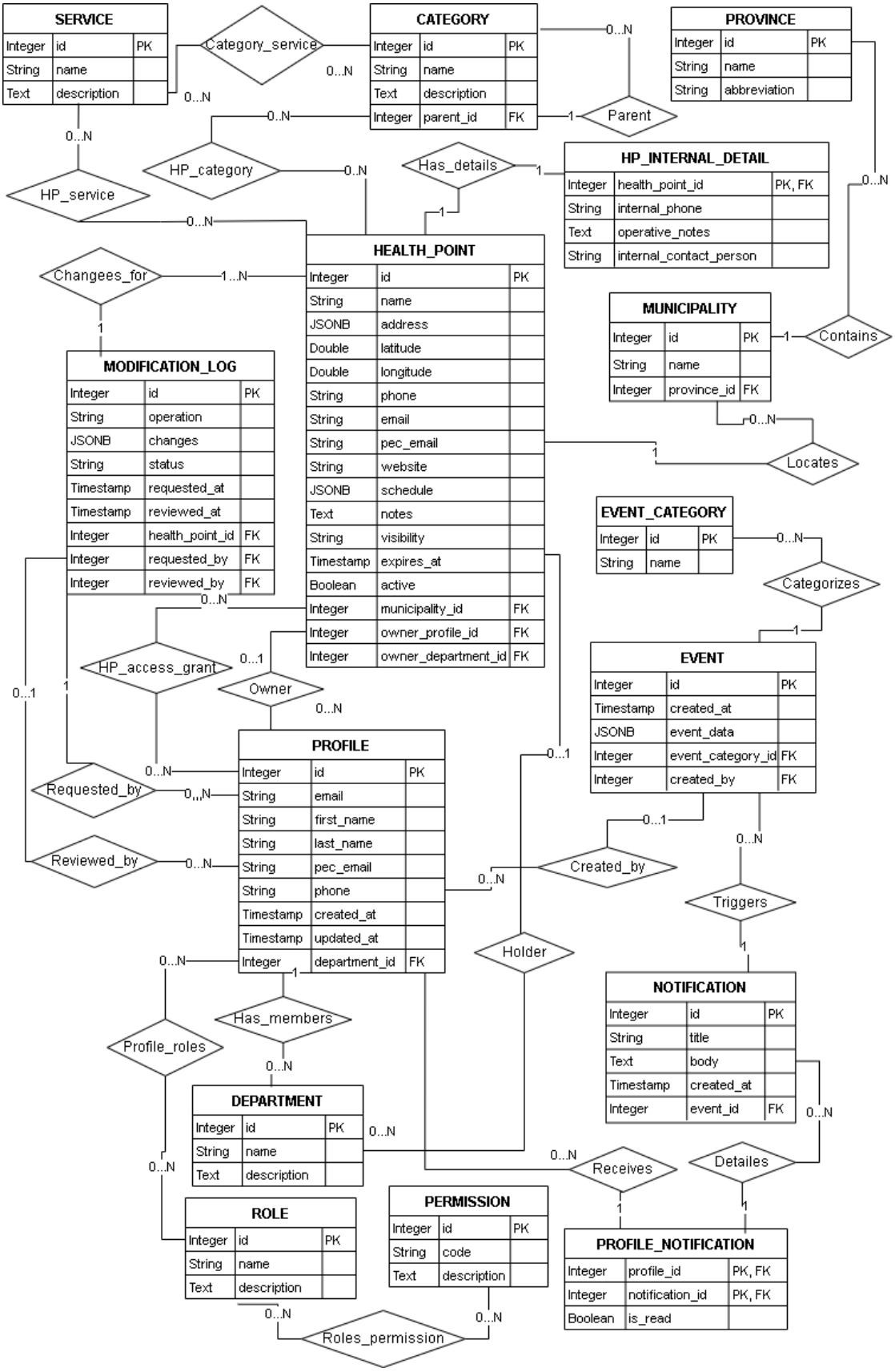

Figura 3.4: Diagramma E/R del database.

operazioni sui dati senza mai interagire direttamente con la persistenza. Sfrutta infatti le interfacce offerte dal *Repository Layer* per manipolare le entità. L'interazione tra i livelli è governata dal principio di *Inversione di Controllo* (IoC), attuato attraverso il pattern *Dependency Injection* (DI). Un servizio non crea le proprie dipendenze, come possono risultare essere i repository, ma le riceve tramite iniezione esterna da parte del sistema. Questo disaccoppiamento non è un mero dettaglio implementativo, ma una scelta architetturale che facilita la testabilità unitaria dei servizi in isolamento e promuove un'architettura estremamente modulare e manutenibile.

Controller Layer

Il livello più esterno del backend è il *Controller Layer*, che agisce come facciata del sistema, esponendone le funzionalità attraverso un'interfaccia *API RESTful*. La sua responsabilità è interpretare le richieste HTTP in arrivo, eseguire una prima e fondamentale validazione dell'input e orchestrare le chiamate ai servizi sottostanti.

Una scelta progettuale cardine, a questo livello, è stata l'adozione sistematica del pattern *Data Transfer Object* (DTO). Invece di esporre direttamente le entità del dominio, che rappresentano la struttura interna e talvolta sensibile del database, vengono utilizzati oggetti DTO. Questi definiscono un contratto API stabile e sicuro. Tale disaccoppiamento è cruciale per due motivi:

- **Sicurezza e Ottimizzazione:** Protegge il modello di dominio interno da esposizioni involontarie e permette di modellare risposte minimali, che contengono solo i dati strettamente necessari a soddisfare la richiesta. Questo riduce il *payload* di dati trasmessi e semplifica l'utilizzo dell'API da parte del client.
- **Flessibilità:** Consente al contratto API di evolvere in modo indipendente dalla struttura del database, garantendo che modifiche interne non impattino i consumatori del servizio stesso.

L'interazione sinergica di questi tre livelli definisce un flusso di controllo unidirezionale e robusto come rappresentato in Figura 3.5. Ogni richiesta proveniente dal client attraversa ordinatamente questi strati, come visualizzato nel diagramma seguente, garantendo che ogni componente agisca esclusivamente entro i confini della propria responsabilità.

3.3.4 Progettazione del Frontend

La progettazione del frontend traduce il concetto astratto di *Presentation Tier* in un'interfaccia utente interattiva, dinamica e manutenibile. Si è adot-

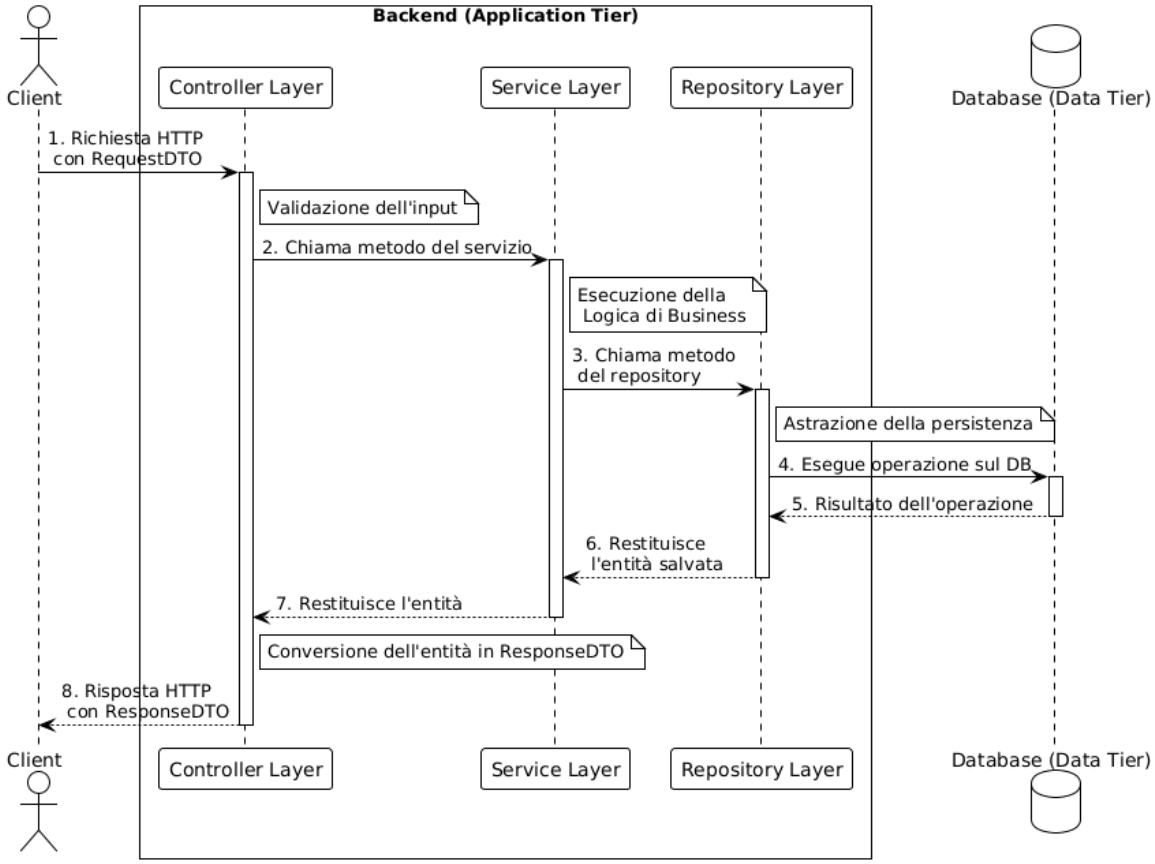

Figura 3.5: Diagramma UML di flusso di una richiesta attraverso l'Architettura a Strati del Backend.

tato un approccio basato su elementi assemblabili detti componenti. Tale tecnica, volta alla modularità e al riutilizzo del codice, è di uso comune per la costruzione di interfacce complesse. L'architettura non si limita alla sola resa visiva, ma si fonda su una rigorosa separazione delle responsabilità, articolata su tre pilastri fondamentali: la gerarchia dei componenti, il livello di servizio per la comunicazione API e una strategia definita per la gestione dello stato.

Architettura a Componenti

Il fondamento dell'architettura è la scomposizione dell'interfaccia utente in *Componenti* isolati e riutilizzabili. È stata adottata una distinzione strategica tra due tipologie di componenti, seguendo il pattern *Container-Presentational*:

- **Componenti Presentazionali:** La loro unica responsabilità è la resa visiva e la coerenza stilistica. Ricevono dati e funzioni esclusivamente tramite le loro proprietà, dai rispettivi componenti padri, e si limitano a visualizzarli. Questa purezza li rende estremamente riutilizzabili.

- **Componenti Contenitore:** Agiscono come orchestratori. Il loro scopo è recuperare e gestire i dati, contenere la logica di *business frontend* e passarli ai componenti presentazionali figli. Essi rappresentano in nucleo funzionale, separando la logica dalla sua rappresentazione.

Questa strategia permette di costruire interfacce complesse attraverso la composizione di elementi semplici, garantendo che la logica applicativa sia disaccoppiata dalla sua struttura visiva.

Separazione della Logica dalla Vista

Una delle scelte architetturali più comuni è l'estrazione sistematica della logica di business e della gestione dello stato dai componenti visivi. Il meccanismo adottato per questa separazione è l'uso dei *Custom Hooks*. Questi elementi agiscono come contenitori di logica riutilizzabile. Provvedono infatti all'incapsulazione delle chiamate API o delle sottoscrizioni a eventi e manipolano lo stato come conseguenza diretta degli stessi avvenimenti. Così facendo, i componenti diventano puramente dichiarativi, ovvero si occupano di esprimere cosa visualizzare in base allo stato corrente, senza preoccuparsi di come tale stato viene ottenuto o modificato. Questo pattern non solo aumenta la leggibilità e la testabilità del codice, ma favorisce anche la riusabilità della logica di business in diverse parti dell'applicazione.

Gestione dei Dati

Il flusso dei dati è governato da due meccanismi complementari. In primo luogo, in modo analogo al backend, tutta la comunicazione con l'API RESTful è stata centralizzata in un *Service Layer* dedicato. Questa astrazione funge da barriera protettiva tra il frontend e le specificità del backend, centralizzando la logica di chiamata e la gestione degli errori in un unico punto.

In secondo luogo, la gestione dello stato interno all'applicazione segue un flusso di dati unidirezionale. Si distingue tra:

- **Stato Locale**, confinato a un singolo componente e utilizzato per dati effimeri della *UI*.
- **Stato Condiviso**, accessibile da diverse parti dell'albero dei componenti, gestito attraverso meccanismi come la *Context API* di `React`.

Questa strategia garantisce che lo stato dell'applicazione sia prevedibile e facile da tracciare, evitando la complessità di flussi di dati incontrollati.

3.4 Selezione dello Stack Tecnologico

Il *workflow* operativo delle prime fasi di progettazione si è concluso con la selezione dello stack tecnologico che meglio si adattasse all'architettura ideata. Le specifiche emerse hanno reso necessario l'utilizzo di *framework* e tecnologie moderne, capaci di soddisfare le proprietà architetturali definite. Di conseguenza, per ognuno dei livelli individuati ovvero: *Data*, *Application* e *Presentation*; è stata selezionata una o più tecnologie atte a supportarne al meglio i requisiti specifici.

3.4.1 Tecnologie per il Data Tier

Per la selezione della tecnologia più opportuna nell'ambito del database, è stato necessario considerare non solo i risultati della progettazione dello stesso, ma anche la totalità dei requisiti funzionali e non funzionali. Il sistema, infatti, richiedeva una notevole flessibilità nella modellazione di alcuni attributi, una proprietà non sempre garantita dai sistemi puramente relazionali data la loro rigidità strutturale. D'altra parte, l'integrità dei dati, la loro tracciabilità e la coerenza relazionale dovevano rimanere una priorità, come richiesto dagli stakeholder, allontanando l'ipotesi di un database documentale di natura *NoSQL*.

Ci si è dunque interrogati su quale potesse essere il compromesso tecnologico in grado di assecondare queste esigenze ibride e che, al contempo, facilitasse

operazioni complesse come *query* geospaziali. La soluzione è stata individuata in `PostgreSQL` con la sua estensione `PostGIS`. Questa scelta rappresenta una sintesi ideale per molteplici ragioni:

- Offre la robustezza, l'affidabilità e la coerenza transazionale di un database relazionale, tutelando l'integrità dei dati.
- Supporta nativamente tipi di dato complessi come `JSON/JSONB`, garantendo la flessibilità necessaria per modellare attributi non strutturati.
- L'estensione `PostGIS` lo trasforma in un potente database geospaziale, fornendo un supporto nativo ed efficiente per l'archiviazione e l'interrogazione di dati geografici, requisito fondamentale del progetto.

3.4.2 Tecnologie per l'Application Tier

La valutazione delle tecnologie da adottare per l'*Application Tier* è stata articolata e ha seguito l'evoluzione del progetto. Le prime fasi esplorative, basate su prototipi in `Java` senza l'ausilio di un *framework*, hanno rapidamente rivelato che tale approccio era troppo semplicistico per gestire la complessità del modello e l'architettura a strati che si stava delineando.

L'attenzione si è dunque spostata su `Spring Boot`, identificato come il *framework* ottimale per l'implementazione dell'architettura a livelli ideata. La sua adozione si è rivelata una scelta virtuosa, in quanto ha fornito un supporto nativo e robusto per:

- L'implementazione rigorosa dei *pattern* discussi in precedenza, come la *Dependency Injection (DI)* per disaccoppiare i *layer* e i *Data Transfer Objects (DTO)*.
- La creazione rapida e standardizzata di *API RESTful*.
- La gestione integrata e dichiarativa delle transazioni, essenziale per la logica del *Service Layer*.

Il lavoro è stato coadiuvato da librerie di supporto come `Lombok`, che ha permesso di snellire la stesura del codice *boilerplate* delle classi di dominio, riducendo la verbosità, minimizzando la possibilità di errori e liberando risorse temporali da dedicare allo sviluppo dei componenti più complessi.

3.4.3 Tecnologie per il Presentation Tier

L'architettura a componenti ideata per il livello di presentazione ha reso imperativo l'utilizzo di uno strumento che la supportasse nativamente. La

scelta è ricaduta su `React`, libreria ampiamente consolidata e supportata da una vasta comunità, ideale per la creazione di *Single-Page Application* moderne. `React` ha permesso di tradurre fedelmente il design architetturale in una base di codice modulare, dove la distinzione tra componenti di logica e di presentazione è un principio fondante.

A sostegno dello sviluppo dell'interfaccia, sono state utilizzate tecnologie specifiche per la gestione della componente cartografica, che rappresenta il cuore dell'applicazione. Per garantire un'esperienza fluida e ad alte prestazioni, è stato scelto un moderno stack di mappatura vettoriale basato su `MapLibre GL`, un potente motore di rendering *open-source* per mappe vettoriali, utilizzato in questo progetto per visualizzare dati cartografici basati su `OpenStreetMap`. Per integrare questa tecnologia in modo armonico nell'ecosistema `React`, è stata utilizzata la libreria `React Map GL`, che espone un set di componenti dichiarativi per il controllo della mappa.

A tale scopo, si è fatto ricorso a fonti dati esterne in formato `GeoJSON`, uno standard aperto per la rappresentazione di dati geografici. La scelta è ricaduta su *dataset* autorevoli curati dalla fondazione Openpolis, disponibili tramite repository GitHub pubblico [4]. Questi file non sono semplici immagini, ma contengono dati vettoriali strutturati che descrivono con precisione la geometria poligonale dei confini di tutte le province e i comuni italiani. Questa scelta metodologica garantisce l'utilizzo di dati aperti, affidabili e standardizzati, assicurando la qualità e la trasparenza dell'informazione geografica presentata all'utente.

Capitolo 4

Sviluppo del Prototipo

Questo capitolo documenta la fase di sviluppo del prototipo, illustrando come il processo implementativo sia stato non solo un'esecuzione tecnica, ma una componente strategica integrante del dialogo con gli esperti di dominio e della progressiva definizione dei requisiti.

4.1 Scopo della Prototipazione

A seguito della fase di progettazione iniziale, è stata avviata la prototipazione. Coerentemente con il paradigma Agile, questa è stata articolata in cicli iterativi, durante i quali le funzionalità sono state progressivamente affinate. L'obiettivo primario non era la produzione di un sistema finale, ma lo sviluppo di un artefatto strategico, un prototipo funzionale utile a guidare gli esperti del dominio nel processo di acquisizione dei dati e a validare le ipotesi progettuali. Si è scelto deliberatamente di non mirare a uno strumento direttamente fruibile dal cliente, poiché i rigorosi standard di privacy e sicurezza, imposti dalla sensibilità dei dati, sarebbero stati inconciliabili con l'approccio di sviluppo rapido richiesto in questa fase esplorativa.

L'adozione di questo approccio ha innescato un virtuoso processo di co-progettazione. Le demo periodiche del prototipo hanno agito da catalizzatore per il confronto, trasformando gli incontri con gli stakeholder in sessioni di lavoro interattive. Questo ha permesso uno scambio bilaterale di competenze. Gli esperti del dominio infatti hanno affinato la loro pragmaticità tecnica, imparando a formulare requisiti in modo più strutturato, mentre il team di sviluppo ha consolidato la propria comprensione del dominio applicativo. Tale sinergia ha reso la comunicazione progressivamente più efficiente e l'analisi stessa più profonda.

Il processo implementativo si è articolato in cicli iterativi ben definiti. Ogni iterazione è iniziata con il testing di tecnologie e componenti specifiche per la

funzionalità da realizzare. Seguiva la creazione minimale del modello di dati necessario. Il parallelo lavoro di implementazione frontend che ne derivava si prefiggeva il duplice scopo di rendere la *feature* disponibile all'utente e di sopperire temporaneamente a eventuali mancanze dell'*application tier*, la cui complessità era talvolta inconciliabile con le scadenze. Ogni ciclo si è concluso con una sessione di revisione, durante la quale il team riportava i progressi agli stakeholder, raccogliendo feedback cruciali per pianificare l'iterazione successiva.

4.2 Implementazione dei Servizi Backend

Il lavoro implementativo, volto alla creazione dei servizi *backend*, prende avvio dalla documentazione prodotta e raffinata nelle fasi di progettazione, con lo scopo di dare forma concreta all'architettura a strati precedentemente descritta. Per il raggiungimento di tale obiettivo, sono stati sfruttati i molteplici strumenti che `Spring Boot` mette a disposizione per la realizzazione di architetture di questo tipo.

La traduzione del modello progettuale in codice ha seguito un percorso metodico, partendo dal livello più vicino ai dati per poi salire progressivamente fino a quello di esposizione delle API. Questa scelta strategica non è stata casuale, ma ha permesso di strutturare il lavoro seguendo livelli di astrazione crescenti. In questo modo è stato possibile isolare con cura i singoli strati, garantendone la testabilità in maniera autonoma e assicurando che la logica dei livelli superiori poggiasse su fondamenta già verificate e stabili.

4.2.1 Implementazione del Data Layer

Il lavoro cardine svolto dal *Data Layer* è quello di astrazione dei dati provenienti dal database e delle operazioni su di essi attuate. La criticità di questa operazione è legata al rischio di accoppiare strettamente la tecnologia su cui si basa la persistenza dei dati e quella del *backend* stesso. Per ovviare a questa criticità, la persistenza viene gestita per mezzo dello standard *Java Persistence API* (JPA).

Esso, in quanto standard e non implementazione, non si occupa direttamente di come risolvere il problema di disaccoppiamento, ma definisce un'interfaccia e un approccio convenzionale per farlo. L'attuazione pratica dello standard proposto è `Hibernate` ovvero l'implementazione di *JPA* integrata in `Spring Boot`, che risolve il problema attuando una *Mappatura Oggetto-Relazionale* (ORM). Ciò significa che esiste una relazione paritetica tra le entità del layer applicativo e le tabelle della base di dati.

Tale approccio permette al sistema di eseguire operazioni sulla persistenza dei dati attraverso semplici chiamate a metodi su specifiche classi, appositamente segnalate con l'annotazione `@Entity`, svincolandosi dalla specificità delle query SQL. Questo sistema è inoltre messo in sinergia con `Spring Data JPA`, un modulo dell'ecosistema `Spring` che semplifica ulteriormente l'implementazione del livello di accesso ai dati. La combinazione dei due permette di definire interfacce di *Repository* capaci di generare automaticamente le query a partire dalla firma dei metodi.

Entity

Le classi *Entity* sono componenti del modello di dominio la cui struttura è mappata direttamente su una tabella della base di dati. Ciascuna istanza di una classe *Entity* rappresenta una singola tupla all'interno della tabella corrispondente. Hanno di conseguenza campi identici agli attributi delle entità del database e non si occupano di null'altro se non di esporre metodi per l'accesso e la gestione di questi ultimi come *setter* e *getter*. Per facilitare la produzione di queste classi, spesso ripetitive, si è sfruttato `Lombok`, una libreria che interviene in fase di compilazione per generare automaticamente codice *boilerplate*. Essa ha permesso di semplificare la scrittura, creando costruttori, *setter* e *getter* in maniera automatizzata per mezzo di apposite annotazioni.

Un esempio emblematico di questa implementazione è la classe `HealthPoint`. Questa classe rappresenta un punto della salute all'interno del *Data Layer*: i suoi campi modellano gli attributi della tabella corrispondente e, tramite le annotazioni di `Lombok`, vengono forniti i metodi per la loro gestione.

Un aspetto critico nell'ORM è la gestione delle relazioni, in particolare quelle multiple come la relazione molti a molti, annotata con il tag `@ManyToMany`, tra `HealthPoint` e `Category`. Per garantire la coerenza dei dati su entrambi i lati della relazione, è buona norma implementare appositi *helper methods* che incapsulino la logica di associazione e dissociazione. Questa gestione deve essere simmetrica: quando una categoria viene aggiunta a un punto della salute, anche il riferimento inverso nel grafo degli oggetti deve essere aggiornato.

Per tale ragione entrambe le entità coinvolte adottano questo pattern. La classe `Category`, che rappresenta il lato proprietario della relazione tramite l'annotazione `@JoinTable`, è colei che gestisce in maniera prioritaria la relazione attraverso il campo `healthPoints` che rappresenta la relazione stessa. Per far ciò espone i metodi `addHealthPoint` e `removeHealthPoint` come ci è possibile osservare nel Codice 4.1).

In modo speculare, la classe `HealthPoint`, che definisce il lato inverso della relazione tramite l'attributo `mappedBy`, implementa i metodi `addCategory` e `removeCategory` come mostrato dal Codice 4.2.

```java
    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(
        name = "hp_category",
        joinColumns = @JoinColumn(name = "category_id"),
        inverseJoinColumns = @JoinColumn(name = "health_point_id")
    )
    private Set<HealthPoint> healthPoints = new HashSet<>();

    public void addHealthPoint(HealthPoint healthPoint) {
        this.healthPoints.add(healthPoint);
        healthPoint.getCategories().add(this);
    }
```

Codice 4.1: Helper methods lato proprietario in `Category.java`.

```java
    @ManyToMany(mappedBy = "healthPoints", fetch = FetchType.LAZY)
    private Set<Category> categories = new HashSet<>();

    public void addCategory(Category category) {
        this.categories.add(category);
        category.getHealthPoints().add(this);
    }
```

Codice 4.2: Helper methods lato inverso in `HealthPoint.java`.

L'adozione di questa strategia su entrambe le entità garantisce che lo stato del modello a oggetti rimanga sempre consistente, indipendentemente da quale lato della relazione venga utilizzato per avviare l'operazione di associazione.

Repository

Una volta definite le *Entity*, il passo successivo consiste nel creare un meccanismo per la loro interrogazione. Questo ruolo è affidato alle interfacce *Repository*, realizzazioni pratiche dell'omonimo design pattern atto a mediare tra il dominio dell'applicazione e la logica di accesso ai dati. Per realizzare questo strato, si è utilizzato il modulo `Spring Data JPA`. Questo componente semplifica drasticamente il processo, fornendo diversi meccanismi per definire le operazioni di accesso ai dati, dalla convenzione sulla configurazione fino alla scrittura di query complesse.

La base per la creazione di un repository è l'estensione dell'interfaccia generica `JpaRepository<T, ID>` fornita da `Spring`, come mostrato all'interno della classe `HealthPointRepository.java` in corrispondenza di riga 2 del

Codice 4.3. In questo modo, le operazioni *CRUD* di base vengono ereditate automaticamente, senza la necessità di scrivere alcuna implementazione concreta.

Per query semplici, `Spring Data JPA` permette di definire interrogazioni semplicemente dichiarando la firma di un metodo nell'interfaccia. Seguendo una specifica convenzione, infatti, il framework astrae il nome del metodo e genera la query corrispondente. Come illustrato alle righe 4-6 del Codice 4.3, questo approccio, noto come *Query Methods*, riduce il codice e aumenta la leggibilità per le operazioni standard.

Per esigenze più complesse, non esprimibili tramite i *Query Methods*, è possibile definire query personalizzate utilizzando l'annotazione `@Query`. Questo approccio offre due alternative:

- ***Java Persistence Query Language*(JPQL):** Una sintassi simile a SQL ma che opera sulle entità e i loro attributi. È una scelta ottimale in termini di disaccoppiamento e portabilità. Un esempio è visibile alla riga 8, dove viene definita una ricerca testuale case-insensitive.

- ***SQL Nativo*:** Utile per sfruttare funzionalità specifiche del dialetto SQL del database. Un caso emblematico per questo progetto è l'interrogazione di dati geospaziali tramite `PostGIS`, ecosistema di funzioni non accessibile via *JPQL*. La *query* implementata a partire dalla riga 11 è un esempio di questa necessità, utilizzando una *Common Table Expression* (CTE) ricorsiva per interrogare le categorie gerarchiche.

4.2.2 Implementazione del Service Layer

Salendo di livello dal *Data Layer*, si incontra il *Service Layer*. Esso funge da collante tra la persistenza dei dati e la loro esposizione all'esterno, rendendolo anzitutto una componente fondamentale per il disaccoppiamento tra queste due realtà. Mette infatti in relazione le classi *Controller*, tipiche del *Presentation Layer*, con i *Repository* già visti nel *Data Layer*. Questo compito è affidato a classi denominate *Service* e segnalate con l'omonima annotazione `@Service`, utile a qualificarle come componenti gestiti dal framework Spring. Il loro ciclo di vita e le loro dipendenze sono governate dal *container Inversion of Control* (IoC). Tramite il pattern *Dependency Injection* (DI), il framework provvede a iniettare automaticamente le istanze necessarie, come un `HealthPointRepository` all'interno di un `HealthPointService`. Questo approccio promuove un basso accoppiamento e un'elevata testabilità dei componenti.

```java
@Repository
public interface HealthPointRepository extends
    JpaRepository<HealthPoint, Integer> {

    List<HealthPoint> findByActiveTrue();

    List<HealthPoint> findByActiveTrueOrderByNameAsc();

    @Query("SELECT h FROM HealthPoint h WHERE h.active = true AND
        h.name ILIKE %:searchTerm%")
    List<HealthPoint> findActiveByNameContaining(String
        searchTerm);

    @Query(value = """
        WITH RECURSIVE category_tree AS (
            SELECT id FROM category WHERE id IN (:categoryIds)
            UNION ALL
            SELECT c.id FROM category c JOIN category_tree ct ON
                c.parent_id = ct.id
        )
        SELECT DISTINCT hp.* FROM health_point hp
        JOIN hp_category pc ON hp.id = pc.health_point_id
        WHERE hp.active = true AND pc.category_id IN (SELECT id
            FROM category_tree)
        """, nativeQuery = true)
    List<HealthPoint>
        findActiveByCategoriesAndChildren(@Param("categoryIds")
        Set<Integer> categoryIds);

    //... restanti query methods ...
}
```

Codice 4.3: Esempio di implementazione di `HealthPointRepository` con diverse strategie di query.

I *service* hanno un compito primario di importanza fondamentale. Essi, rappresentando la logica di business nella sua interezza, hanno la responsabilità di orchestrare le operazioni sui dati, affinché questi risultino coerenti e attendibili nel tempo. Spesso un *service* si occupa della gestione di una moltitudine di *repository* e, affinché tale compito venga svolto in maniera sicura, è richiesto l'uso dell'annotazione `@Transactional`. Grazie ad essa si specificano le operazioni che richiedono atomicità nell'essere svolte. Se, per esempio, fosse necessario eliminare un utente e di conseguenza modificare i riferimenti ai punti di sua proprietà, è cruciale che l'intera sequenza avvenga come un'unica operazione. Se si dovesse incorrere in un errore, l'intera transazione verrebbe annullata per mantenere la persistenza, l'affidabilità e la coerenza dei dati.

Un aspetto architetturale cruciale implementato in questo strato è l'uso del pattern *Data Transfer Object*. Per disaccoppiare il modello di dominio interno, composto da *Entity JPA*, dal contratto pubblico esposto dalle API, il Service Layer non opera direttamente con le entità verso l'esterno. Esso riceve i dati dai *Controller* sotto forma di *DTO* e, analogamente, restituisce *DTO* come risultato delle sue operazioni. Questo approccio offre molteplici vantaggi:

- **Stabilità dell'API:** La struttura delle API non risulta legata rigidamente a quella del database, permettendo a quest'ultima di evolvere senza impattare i *client*.
- **Sicurezza:** Si evita di esporre accidentalmente dati sensibili presenti nelle *Entity* ma non necessari all'esterno.
- **Performance:** Si trasferiscono solo i dati strettamente necessari, ottimizzando il *payload* e prevenendo problemi legati al caricamento *lazy* delle relazioni *JPA*.

Data Transfer Object (DTO)

Componenti fondamentali per garantire la separazione tra i *layer* sono i DTO. Si tratta di un pattern architetturale che prevede l'uso di classi il cui unico scopo è trasportare dati tra i confini del sistema, tipicamente tra il *Service* e il *Presentation Layer*. Per loro natura, devono essere strutture dati semplici e immutabili. In questo progetto, tale requisito è stato soddisfatto attraverso l'adozione dei *record Java*, un costrutto del linguaggio che permette di definire classi in modo conciso e sicuro, riducendo drasticamente il codice *boilerplate*.

Il loro utilizzo permette di disaccoppiare il modello di dominio interno, composto da *Entity JPA*, dal contratto pubblico esposto dalle *API*. Ciò offre molteplici vantaggi, tra cui la stabilità dell'API, una maggiore sicurezza,

e un miglior controllo sulle performance, prevenendo problemi di caricamento massivo dei dati. Inoltre, una singola entità può essere rappresentata da più DTO, ciascuno modellato su uno specifico caso d'uso. Questa strategia consente di progettare API precise e performanti, modellate sulle reali esigenze dell'interfaccia utente. Per esempio, l'entità `HealthPoint` è servita da due DTO distinti:

- `HealthPointDto`: una rappresentazione completa e dettagliata, utilizzata quando il *client* richiede tutte le informazioni di un singolo punto della salute come mostrato nel Codice 4.4.
- `HealthPointSimpleDto`: una versione minimale contenente solo ID e nome, ottimizzata per popolare liste o menu a tendina in cui mostrare l'intera anagrafica sarebbe inefficiente come mostrato nel Codice 4.5.

```
public record HealthPointDto(
        Integer id,
        String name,
        JsonNode address,
        Integer municipalityId,
        // ... altri campi dettagliati ...
        Boolean active,
        List<CategorySimpleDto> categories
) {
    public static HealthPointDto from(HealthPoint entity) {
        // ... logica di mappatura da Entity a DTO ...
    }
}
```

Codice 4.4: DTO dettagliato per l'entità `HealthPoint`.

```
public record HealthPointSimpleDto(
    Integer id,
    String name
) {
    public static HealthPointSimpleDto from(HealthPoint entity) {
        return new HealthPointSimpleDto(entity.getId(), entity.getName());
    }
}
```

Codice 4.5: DTO semplificato per l'entità `HealthPoint`.

Service

Per illustrare come il *Service Layer* orchestri le interazioni tra i componenti del dominio, si analizza il caso d'uso della creazione di un nuovo Punto della

Salute. Questa operazione, sebbene comune, incarna perfettamente il ruolo del *service* nel farsi mediatore e garante della coerenza dei dati, come illustrato nel frammento di Codice 4.6 estratto dalla classe `HealthPointServiceImpl`.

L'intero processo è racchiuso in un metodo annotato con `@Transactional`, che assicura l'atomicità dell'operazione tale per cui o tutti i passaggi hanno successo, o l'intera transazione viene annullata. Il flusso logico ha inizio quando il *Service* riceve un DTO dal *Controller*, contenente i dati grezzi del nuovo punto. Da questo momento, il *Service* avvia la sua attività di orchestrazione. Per prima cosa, interroga il `MunicipalityRepository` per recuperare e validare l'entità del Comune a cui il punto appartiene. Successivamente, si rivolge al `CategoryRepository` per ottenere le istanze delle entità `Categoria` basandosi sugli identificativi forniti. Una volta che il *Service* si trova con l'oggetto `HealthPoint` completamente assemblato e arricchito delle sue relazioni invoca l'`HealthPointRepository` per persistere il nuovo grafo di oggetti nel database. Infine, l'entità salvata viene riconvertita in un DTO e restituita al chiamante, completando il ciclo e mantenendo il modello di dominio interno disaccoppiato dalle API.

4.2.3 Implementazione del Controller Layer

Ultimando la nostra scalata all'interno dell'architettura backend, giungiamo al layer di controllo, lo strato responsabile dell'esposizione delle funzionalità di sistema con l'esterno. Esso si occupa di raccogliere le richieste HTTP provenienti dal frontend e di rigirarle agli appositi *Service*. Tale compito viene assolto attraverso l'esposizione di API RESTful, gestendo il protocollo HTTP e delegando la logica di business al *Service Layer*.

Controller

Le classi che danno il nome al livello stesso, i *Controller*, costituiscono i componenti cardine di questo strato. Sono identificati dall'annotazione `@RestController`, che li qualifica come componenti del modulo `Spring MVC`, responsabile della gestione delle richieste web, e indica che i valori di ritorno dei metodi devono essere serializzati direttamente nel corpo della risposta. Per adempiere a tale scopo risulta fondamentale, ancora una volta, l'utilizzo della ormai nota *Dependency Injection*: questo pattern ci permette di istanziare specifici *service* all'interno dei *controller*, assicurando che la logica di business rimanga completamente disaccoppiata dai dettagli del protocollo di comunicazione.

I *controller* fungono dunque da "postini", mappano il tipo, il verbo e l'*URI* delle richieste HTTP pervenute e, seguendo *path* preventivamente de-

```java
@Service
@Transactional(readOnly = true)
public class HealthPointServiceImpl implements HealthPointService {

    private final HealthPointRepository healthPointRepository;
    private final CategoryRepository categoryRepository;
    private final MunicipalityRepository municipalityRepository;

    // ... costruttore per la Dependency Injection ...

    @Override
    @Transactional
    public HealthPointDto create(HealthPointDto dto) {
        HealthPoint newHealthPoint = new HealthPoint();

        // ... mappatura dei campi semplici dal DTO all'entit (name,
            address, etc.) ...

        Municipality municipality =
            municipalityRepository.findById(dto.municipalityId())
                .orElseThrow(() -> new EntityNotFoundException("Municipality
                    not found"));
        newHealthPoint.setMunicipality(municipality);

        if (dto.categories() != null && !dto.categories().isEmpty()) {
            Set<Integer> categoryIds = // ... estrazione ID dal DTO ...
            List<Category> categories =
                categoryRepository.findAllById(categoryIds);
            for (Category category : categories) {
                newHealthPoint.addCategory(category);
            }
        }

        HealthPoint savedEntity =
            healthPointRepository.save(newHealthPoint);

        return HealthPointDto.from(savedEntity);
    }
}
```

Codice 4.6: Orchestrazione di tre repository nel metodo `create` di `HealthPointServiceImpl.java`.

finite, le associano univocamente a un metodo che si interfaccia con il *service layer*, perché possa provvedere al compito impartitogli. L'annotazione `@RequestMapping` definisce l'*URI* di base per una risorsa, come per esempio `/api/health-points`, che fa riferimento ai Punti della Salute. Mentre annotazioni come `@GetMapping` e `@PostMapping` specificano il verbo HTTP a cui un metodo risponde, associando dunque `GET` e `POST` agli specifici *endpoint* e definendo di fatto le operazioni consentite su quella risorsa.

Il *Controller* è dunque un layer volutamente agnostico rispetto alla logica di business. Il suo compito è orchestrare il ciclo di vita della richiesta HTTP. A tale scopo, utilizza annotazioni come `@RequestBody` per convertire i *payload JSON* in DTO e `@RequestParam` per leggere i parametri dalle *query URL*, attivando meccanismi di validazione con `@Valid`. Avviene poi la fase di delega, in cui invoca i metodi del *service* appropriato, passando i dati validati e disaccoppiati dalla richiesta HTTP. Una volta pervenutegli le risposte dal *service*, passa alla costruzione della risposta: utilizza la classe `ResponseEntity` per costruire una risposta HTTP completa, definendo con precisione lo *status code*, gli *header* e il corpo della risposta, che viene automaticamente serializzato in *JSON*. Il Codice 4.7 mostra un'implementazione concreta di questo flusso nel `HealthPointController`.

Il percorso implementativo descritto, che si muove dal *Data Layer* fino al *Controller Layer*, delinea un'architettura *backend* robusta e scalabile. Ogni strato possiede responsabilità precise e confini netti. Il *Data Layer* gestisce la persistenza astraendo il database, il *Service Layer* incapsula la logica di business orchestrando le operazioni ed infine, il *Controller Layer* espone queste funzionalità in modo sicuro e standardizzato attraverso un'*API RESTful*.

Questa rigorosa separazione delle responsabilità, resa possibile dagli strumenti offerti da `Spring` come la *Dependency Injection* e la gestione delle transazioni, non è un mero esercizio stilistico. Essa si traduce in vantaggi concreti. Ogni componente può essere testato in isolamento, le modifiche a uno strato non impattano gli altri, e il sistema nel suo complesso risulta più facile da comprendere, manutenere ed estendere. Avendo così definito l'interfaccia dati del sistema, il passo successivo è analizzare l'implementazione del fruitore di questa parte del sistema ovvero il *Frontend*.

4.3 Implementazione dell'Interfaccia Frontend

Come discusso nella sezione iniziale di questo capitolo, il prototipo ha svolto un ruolo strategico nel processo di coprogettazione con gli esperti di dominio. Sebbene il sistema sia un'unione coesa di più componenti, l'interfaccia *frontend* si è distinta come il principale catalizzatore di feedback. Il *frontend*,

```java
@RestController
@RequestMapping("/api/health-points")
public class HealthPointController {

    private final HealthPointService healthPointService;

    ... costruttore per la Dependency Injection ...

    @GetMapping
    public ResponseEntity<List<HealthPointDto>> getAllHealthPoints(
            @RequestParam(required = false) String search,
            @RequestParam(required = false) Set<Integer> categoryIds){
        // ... logica per delegare la ricerca al service in base ai
            parametri ...
    }

    @PostMapping
    public ResponseEntity<HealthPointDto> addHealthPoint(@Valid
        @RequestBody HealthPointDto dto) {
        HealthPointDto created = healthPointService.create(dto);

        URI location = ServletUriComponentsBuilder
                .fromCurrentRequest().path("/{id}")
                .buildAndExpand(created.id()).toUri();

        return ResponseEntity.created(location).body(created);
    }
}
```

Codice 4.7: Esempio di Controller REST per la gestione della risorsa `HealthPoint`.

infatti, dando una forma grafica e interattiva a concetti altrimenti astratti, ha permesso di ottenere riscontri pragmatici e mirati.

La sua capacità di evolvere in modo relativamente indipendente dallo stato di avanzamento del Backend si è rivelata un vantaggio cruciale. Seguendo un approccio Agile, i requisiti sono mutati significativamente nel corso delle iterazioni. Una modifica che appariva semplice a livello di interfaccia avrebbe potuto richiedere un intervento complesso e trasversale sull'intero *stack* implementativo. Grazie al disaccoppiamento, è stato possibile implementare tali modifiche prima solo a livello grafico, verificandone l'utilità, la correttezza e la reale volontà degli stakeholder di adottare l'accorgimento proposto. Solo dopo una validazione chiara e definitiva, si è potuto procedere con l'implementazione completa su backend e database, minimizzando così il rischio di sviluppare funzionalità superflue o errate.

L'implementazione del *Frontend* è stata dunque lo strumento principe per l'acquisizione di feedback e la validazione dei requisiti. Lo sviluppo si è basato su un nucleo di componenti fondamentali realizzati in `React`, utilizzati come mattoni per costruire progressivamente interfacce più complesse. Il processo ha seguito un andamento a spirale, volto ad arricchire il sistema con funzionalità sempre più capillari, partendo da solide fondamenta.

La traduzione dei principi dettati dalla progettazione in una base di codice manutenibile ha richiesto l'adozione di una struttura rigorosa e convenzionale per la gestione dei file. L'organizzazione del codice sorgente non è un dettaglio secondario, ma un pilastro che garantisce la leggibilità, la scalabilità e la facilità di navigazione del progetto.

La struttura adottata suddivide il codice per responsabilità funzionale, raggruppando i file in cartelle distinte sulla base della loro attinenza semantica:

- `/pages`: Contiene i componenti di primo livello, ognuno dei quali rappresenta una pagina o una rotta principale dell'applicazione. Nell'implementazione attuale, vi si trovano `HomePage` e `DashboardPage`, pagine fondamentali per la rappresentazione grafica delle funzionalità principali del sistema. Questi componenti agiscono come contenitori, che orchestrano i dati e gli elementi *UI* specifici per la vista di appartenenza.

- `/components`: È la libreria di elementi *UI* riutilizzabili. Contiene tutti i componenti presentazionali, da quelli più atomici, come i pulsanti, a quelli più complessi, come le *card* dei punti ed i pulsanti. Gli elementi di questa cartella sono accomunati dal loro scopo: visualizzare dati ricevuti tramite *props*, rimanendo completamente agnostici alla logica di business.

- `/hooks`: Incapsula la logica di business e la gestione dello stato riutilizzabile. I *Custom Hooks* sono il meccanismo primario per estrarre la logica

di *data-fetching* e di interazione dai componenti, rendendoli più snelli e dichiarativi.

- `/services`: Contiene il *Service Layer* del *Frontend*, un'astrazione responsabile esclusivamente della comunicazione con le API REST del Backend.

- `/styles`: Centralizza gli stili globali e le variabili CSS, garantendo coerenza visiva in tutta l'applicazione.

Questa suddivisione è l'espressione concreta del principio cardine dell'architettura: la separazione tra i componenti contenitore conservati in `/pages` e presentazionali contenuti in `/components`. Tale separazione garantisce il disaccoppiamento tra la logica di business e la sua rappresentazione visiva, rendendo il sistema più modulare e facile da manutenere.

4.3.1 Gestione del Flusso Dati

Per evitare la dispersione della logica e garantire un flusso di dati prevedibile, è stato implementato un pattern architetturale a tre stadi che disaccoppia nettamente l'interfaccia utente dalla gestione dello stato e dalla comunicazione di rete. Così facendo si assicura che ogni parte del sistema abbia una singola e ben definita responsabilità. Il ciclo ha inizio quando un componente necessita di dati. Esso invoca un *Custom Hook* dedicato, il quale a sua volta orchestra la chiamata al *Service API* appropriato, gestendo l'intero ciclo di vita della richiesta.

Custom Hooks

Il cuore della logica applicativa del Frontend risiede nei *Custom Hooks*. Si tratta di un meccanismo fondamentale di `React` che consiste in funzioni `JavaScript` riutilizzabili, il cui nome, per convenzione, inizia sempre con il prefisso `use`. La loro caratteristica principale è la capacità di chiamare al loro interno altri *hooks*, permettendo di incapsulare e condividere logica *stateful* tra più componenti.

Il nocciolo della loro funzionalità risiede proprio nella composizione degli *hooks* nativi di `React`:

- `useState` viene utilizzato per dichiarare e gestire le variabili di stato interne all'*hook*.

- `useEffect` viene impiegato per gestire i *side effects*, come le chiamate ad API in risposta a specifici cambiamenti o al montaggio del componente.

Questa architettura permette una divisione capillare della logica di business e promuove un elevato riutilizzo del codice, evitando ridondanze.

Il principio sul quale si basano i *custom hooks* è quello di evitare il sovraccarico di componenti visivi con la gestione dello stato di caricamento, degli errori e del recupero dati. Tale logica viene estratta in funzioni riutilizzabili, contenute negli *hooks* stessi. Un componente che necessita dei dati descritti non deve far altro che invocare l'*hook*, ottenendo in cambio un'interfaccia completa per accedere non solo ai dati specifici dell'interrogazione, ma anche allo stato della richiesta e a funzioni per manipolarla. Di conseguenza, il componente rimane puramente dichiarativo. Il suo unico compito, dunque, è quello di renderizzare la *UI* in base allo stato fornito dall'*hook*, senza conoscere i dettagli di come i dati vengono recuperati o di come gli errori vengano gestiti.

Un esempio emblematico di questo approccio è l'*hook* `useHealthPoints`, responsabile di fornire l'elenco filtrato dei punti della salute. L'implementazione di tale *hook*, contenuta nel Codice 4.8, va oltre un semplice recupero dati e mette in luce tre concetti chiave, primo tra tutti l'incapsulamento dello stato. L'*hook* gestisce internamente tutte le variabili di stato necessarie tramite `useState`, ovvero: `points`, `selectedPoint`, `loading` e `error`, nascondendo questa complessità al componente. Risulta fondamentale anche nella gestione reattiva degli effetti. Nel caso presentato l'*hook* è infatti reattivo ai cambiamenti dei filtri. Grazie a `useCallback` con l'array di dipendenze `[searchQuery, selectedCategoryIds]`, la funzione `fetchPoints` viene memoizzata e ricreata solo quando i parametri di ricerca cambiano. A sua volta, `useEffect` si attiva in base alla modifica di `fetchPoints`, implementando una logica di *debounce* di 500ms. Questa tecnica è fondamentale per le performance, in quanto evita di inondare il Backend di richieste API a ogni singolo tasto premuto dall'utente, avviando la ricerca solo dopo una breve pausa. Infine, l'*hook* espone un'interfaccia pubblica. Il valore di ritorno dell'*hook* non è solo un dato, ma un oggetto che costituisce un'*API* completa per il componente: i dati, lo stato associato a un suo gestore e un'azione esplicita di *reload*, che permette al componente di forzare un nuovo caricamento dei dati quando necessario.

Il Service Layer del Frontend

In modo del tutto speculare all'architettura Backend, anche il Frontend implementa un *Service Layer*, il cui unico scopo è agire da intermediario con le API REST. I *Custom Hooks*, pur orchestrando la logica di business, non eseguono direttamente le chiamate HTTP. Essi delegano questo compito a moduli specifici, come il file `services/api/healthPointsAPI.js`, che funge da unico punto di contatto tra l'applicazione e l'endpoint dei Punti della Salute.

```
import { useState, useEffect, useCallback } from 'react';
import healthPointsAPI from '../../services/api/healthPointsAPI.js';

export const useHealthPoints = (searchQuery, selectedCategoryIds)
    => {
const [points, setPoints] = useState([]);
const [selectedPoint, setSelectedPoint] = useState(null);
const [loading, setLoading] = useState(true);
// ... altri stati ...


const fetchPoints = useCallback(async () => {
    try {
        setLoading(true);
        const data = await healthPointsAPI.getFilteredHealthPoints({
            searchQuery,
            categoryIds: selectedCategoryIds
        });
        setPoints(data);
    } catch (err) {
        setError(err.message);
    } finally {
        setLoading(false);
    }
}, [searchQuery, selectedCategoryIds]); // Dipendenze del callback

useEffect(() => {
    const debounceTimer = setTimeout(() => {
        fetchPoints();
    }, 500);

    return () => clearTimeout(debounceTimer);
}, [fetchPoints]);

return { points, selectedPoint, setSelectedPoint, loading, error,
    reload: fetchPoints };
};
```

Codice 4.8: Implementazione dell'hook `useHealthPoints` con logica di debounce e memoizzazione.

Questa astrazione è un punto di snodo critico dell'architettura per diverse ragioni:

- **Centralizzazione:** Tutta la logica relativa alla comunicazione di rete per una specifica risorsa è confinata in un unico modulo. Se l'*URL* di un'*API* dovesse cambiare o se fosse necessario aggiungere un *header* di autenticazione a tutte le richieste, la modifica sarebbe localizzata in un singolo file.

- **Astrazione della tecnologia:** Il resto dell'applicazione non conosce i dettagli implementativi delle chiamate di rete. Che si utilizzi la *fetch API* nativa del browser o una libreria di terze parti l'interfaccia esposta dal *service* non cambia.

- **Separazione delle Responsabilità:** I *Custom Hooks* si occupano della gestione dello stato, mentre il *Service Layer* si occupa esclusivamente della meccanica della comunicazione HTTP: nello specifico attua la costruzione della richiesta, l'invio e la basilare gestione della risposta.

L'implementazione del service `HealthPointsAPI`, che possiamo vedere nel Codice 4.9, adotta diverse best practice moderne di `JavaScript`. L'utilizzo della sintassi `async/await` permette di gestire le operazioni asincrone in modo leggibile e sequenziale. Ogni metodo del service è una funzione `async`, che restituisce implicitamente una `Promise`. Nel metodo `getByCategories`, ad esempio, si osserva l'uso dell'interfaccia nativa `URLSearchParams` per costruire in modo robusto e sicuro la *query string* da allegare all'*URL*, evitando i rischi legati alla concatenazione manuale di stringhe. Al contrario, il metodo `createHealthPoint` mostra la configurazione necessaria per una richiesta `POST`. L'oggetto `options` passato a `fetch` specifica il metodo, ovvero `POST`, gli *header*, atti ad informare il *server* del formato dei dati inviati, e il *body*, che contiene i dati del nuovo punto della salute, serializzati in una stringa *JSON* tramite `JSON.stringify`.

Infine, una gestione, seppur basilare, degli errori è implementata in ogni metodo. Essa ha svolto un ruolo di fondamentale importanza in un'ottica di implementazione agile, permettendo un'analisi dell'errore capillare senza introdurre un livello di complessità implementativa articolato.

Inoltre risulta imperativa la presenza del controllo `if (!response.ok)` quando si usa `fetch`, poiché, a differenza di altre librerie, essa non rigetta la `Promise` in caso di risposte con *status HTTP* di errore. Questa verifica esplicita assicura che tali risposte vengano gestite come eccezioni, propagando l'errore al *Custom Hook* chiamante, che potrà così aggiornare lo stato dell'applicazione e informare l'utente.

```
1
2  class HealthPointsAPI {
3
4    async getFilteredHealthPoints(filters = {}) {
5      // ... logica di smistamento in base ai filtri ...
6    }
7
8    async getAllHealthPoints() {
9      try {
10       const response = await fetch(/* ... URL ... */);
11       if (!response.ok) {
12         throw new Error(`HTTP error! status: ${response.status}`);
13       }
14       return await response.json();
15     } catch (error) {
16       // ... gestione errore ...
17     }
18   }
19
20   async getByCategories(categoryIds = []) {
21     try {
22       const params = new URLSearchParams({ categoryIds:
             categoryIds.join(',') });
23       const url = /*... costruttore dinamico di URL per endpoint API ...*/ ;
24       const response = await fetch(url);
25
26       if (!response.ok) {
27         // ... gestione errore ...
28       }
29       return await response.json();
30     } catch (error) {
31       // ... gestione errore ...
32     }
33   }
34
35   async createHealthPoint(healthPointData) {
36     try {
37       const response = await fetch(
38         `${API_CONFIG.BASE_URL}${API_CONFIG.ENDPOINTS.HEALTH_POINTS}`,
39         {
40           method: 'POST',
41           headers: {
42             'Content-Type': 'application/json',
43           },
44           body: JSON.stringify(healthPointData),
45         }
46       );
47       if (!response.ok) {
48         // ... gestione errore ...
```

```
49        }
50        return await response.json();
51      } catch (error) {
52        // ... gestione errore ...
53      }
54    }
55  }
56
57  export default new HealthPointsAPI();
```

Codice 4.9: Implementazione del Service Layer per la risorsa `HealthPoint`.

4.3.2 Il Componente Dashboard

L'analisi della pagina `Dashboard` offre un'interessante spaccato della filosofia incrementale che ha guidato l'intero progetto. Questa sezione dell'applicazione, infatti, funge da microcosmo del processo di co-progettazione. Le diverse viste che la compongono non sono nate da un elenco di requisiti statici e predefiniti, ma sono state implementate strategicamente per dare una forma tangibile a quelle funzionalità che, durante il dialogo con gli stakeholder, rimanevano più ambigue o complesse da definire a parole.

Attraverso l'implementazione di queste interfacce, è stato possibile trasformare concetti astratti, come la gestione strategica delle modifiche o il flusso delle notifiche, in artefatti visivi e interattivi. Questo ha permesso di raccogliere feedback precisi e di validare le logiche di business prima ancora che il loro supporto nel Backend fosse completo, incarnando pienamente il principio della prototipazione come strumento di scoperta e dialogo.

La pagina è strutturata come un contenitore principale che agisce da orchestratore. Tramite lo stato interno, gestito dall'*hook* `useState`, e una semplice logica di rendering condizionale, esso si occupa di visualizzare la vista attiva selezionata dall'utente attraverso il componente `DashboardSidebar`. Questa architettura a singolo contenitore permette di simulare una navigazione multipagina pur rimanendo all'interno di un'unica rotta, una tecnica comune nelle *Single Page Application* (SPA) che garantisce una transizione fluida e immediata tra le diverse sezioni funzionali. Le viste renderizzate da questo componente possono essere classificate in due categorie, che riflettono il loro stadio di maturità nel ciclo di prototipazione:

- **Viste di Prototipazione Rapida con Uso di *Mock Data*:** I componenti `HealthPointsView`, `ModificationsView` e `NotificationsView` sono stati inizialmente implementati utilizzando dati fittizi detti *mock data*. Questo approccio ha permesso di definire e validare rapidamente

la struttura dell'interfaccia utente e l'esperienza utente con gli stakeholder. In particolare, la `ModificationsView`, in Figura 4.1, ha fornito una rappresentazione visiva e immediata del *Proposal Pattern*, volto a rendere rigorose e storicizzabili le modifiche. Esso, discusso in fase di progettazione, mostra all'utente lo stato delle proprie proposte permettendo, in maniera intuitiva, di valutarne lo stato di avanzamento. Analogamente, la `NotificationsView` ha materializzato il sistema di notifiche, aiutando a definirne il contenuto e l'aspetto prima di implementare l'infrastruttura *Event-Driven* nel Backend.

- **Viste Funzionali Complete:** La vista `AddHealthPointView` rappresenta il caso di studio più completo all'interno della `Dashboard`, essendo un'implementazione matura e funzionante. Questo componente è un esempio emblematico dell'architettura *Component/Custom Hook* descritta in precedenza. La vista ignora quasi totalmente la logica. Si occupa di renderizzare gli elementi del *form* e di collegare gli eventi dell'utente come `onChange` ed `onSubmit` alle funzioni fornite dall'esterno. Tutta la complessità logica ovvero la gestione dello stato del *form*, è interamente incapsulata e gestita dall'*hook* `useAddHealthPoint`. Questa separazione netta delle responsabilità rende il componente di vista estremamente leggibile e focalizzato sulla presentazione, mentre la logica di business rimane riutilizzabile e testabile in modo isolato.

La `Dashboard` dunque non è solo un pannello di controllo per l'utente, ma anche un potente strumento del processo di sviluppo. Essa dimostra come, attraverso di prototipazione rapida, sia stato possibile navigare la complessità dei requisiti, promuovendo una collaborazione efficace e garantendo che il prodotto finale fosse allineato con le reali necessità degli esperti di dominio.

4.3.3 Applicazioni del Componente Map

Il componente `Mappa` rappresenta senza dubbio l'elemento più emblematico dell'intero progetto. La sua centralità non è solo di natura funzionale, ma anche implementativa, dato che è stato declinato in due contesti distinti per soddisfare esigenze specifiche dell'applicazione. Sfruttando i dati geografici forniti da `OpenStreetMap` (OSM) e la potente libreria di rendering `MapLibre GL`, il componente è stato integrato nel prototipo per realizzare due funzionalità chiave:

- **Mappa principale nella** `HomePage`: qui la mappa viene utilizzata per l'esplorazione interattiva e la visualizzazione georeferenziata dei punti sanitari, costituendo il cuore dell'esperienza utente.

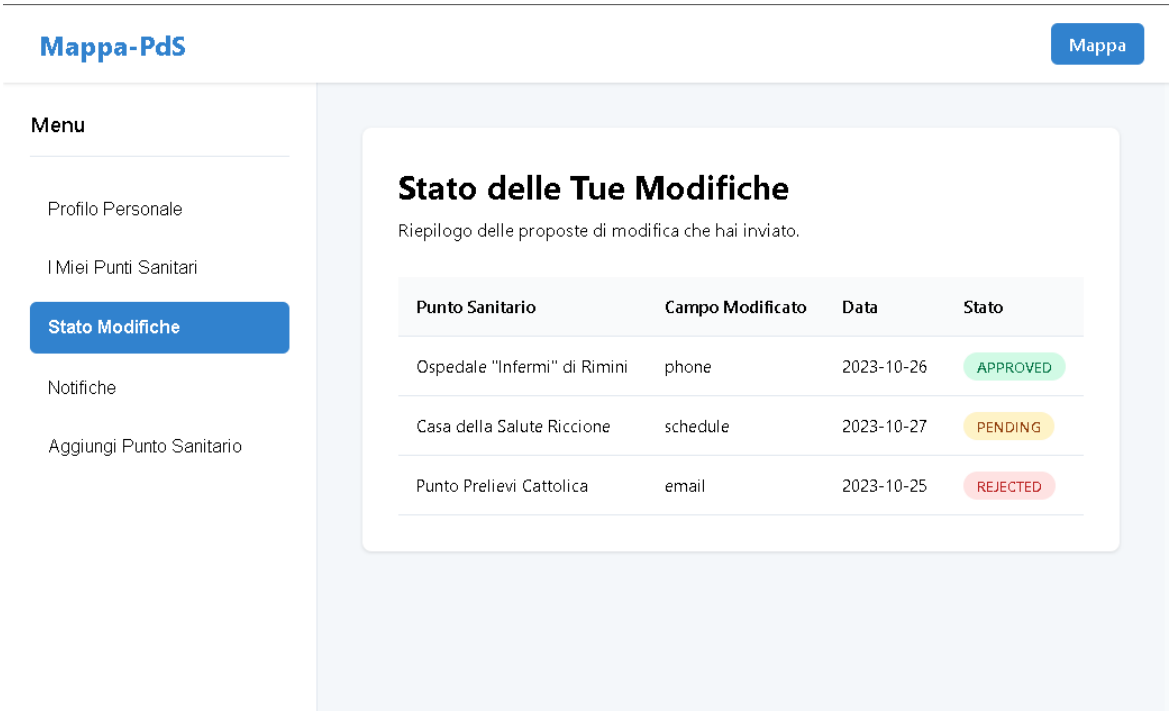

Figura 4.1: Vista dello Stato delle modifiche nella DashBoardPage.

- **Mappa nel *form* di inserimento**: in questo contesto, la mappa funge da strumento di precisione per consentire all'utente di definire le coordinate geografiche, ovvero latitudine e longitudine, di un nuovo punto sanitario in modo intuitivo e privo di errori.

Applicazione nella HomePage

Nella sua applicazione principale, il componente `Map` agisce come un'interfaccia di visualizzazione dinamica per l'intero *dataset* di punti sanitari come è rappresentato dalla Figura 4.2. La sua implementazione è stata studiata per garantire sia un'ottima performance che un'elevata usabilità, anche in presenza di un gran numero di dati.

Essa implementa infatti un sistema di *clustering* dinamico. Per evitare la sovrapposizione di indicatori e mantenere la mappa leggibile a bassi livelli di zoom, i punti vicini vengono raggruppati in *cluster*. Ogni *cluster* mostra il numero di punti che aggrega e adotta una colorazione differente in base alla sua densità, fornendo un'immediata percezione visiva della distribuzione dei servizi sul territorio.

Una seconda importante prerogativa che viene rispettata è l'intuitiva interattività e utilizzabilità del prodotto. L'utente può interagire con la mappa

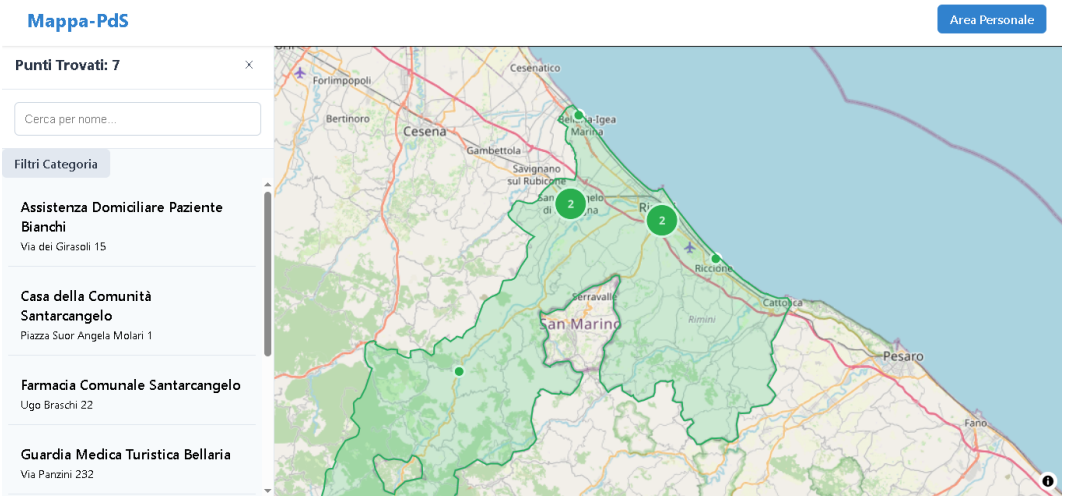

Figura 4.2: Mappatura presente nella HomePage con visualizzazione a cluster dei punti sanitari.

in modo fluido. Un clic su un *cluster* provoca uno zoom progressivo verso l'area di interesse, rivelando i punti individuali o *cluster* più piccoli. Un clic su un singolo punto sanitario lo seleziona, facendo apparire, sulla *sidebar*, le informazioni di riepilogo e centrandolo nella finestra. Infine gli strumenti di *layering* geografico permettono a livelli appropriati di zoom sul componente di visualizzare i confini provinciali e comunali fornendo uno strumento che aiuta ulteriormente l'orientamento e la leggibilità della carta.

La mappa è inoltre sincronizzata con lo stato applicativo e dunque reattiva ai cambiamenti esterni. Ad esempio, quando un utente seleziona un punto da una lista testuale presente nella *sidebar*, la mappa si sposta e zooma automaticamente su di esso, garantendo una perfetta coerenza tra i diversi componenti dell'interfaccia.

Applicazione nel Form di Inserimento di un Nuovo Punto Sanitario

Nel contesto del *form* di creazione di un nuovo punto, la mappa abbandona la sua funzione di visualizzazione dati per trasformarsi in uno strumento di *input*, denominato *Location Picker* e riportato in Figura 4.3. L'obiettivo è semplificare e rendere più accurato il processo di acquisizione delle coordinate geografiche, un'operazione che sarebbe altrimenti complessa e soggetta a errori se demandata all'inserimento manuale.

Le caratteristiche di questa implementazione sono:

- **Acquisizione delle Coordinate tramite Clic**: L'utente può navigare la mappa e fare clic nel punto esatto in cui desidera posizionare la nuova

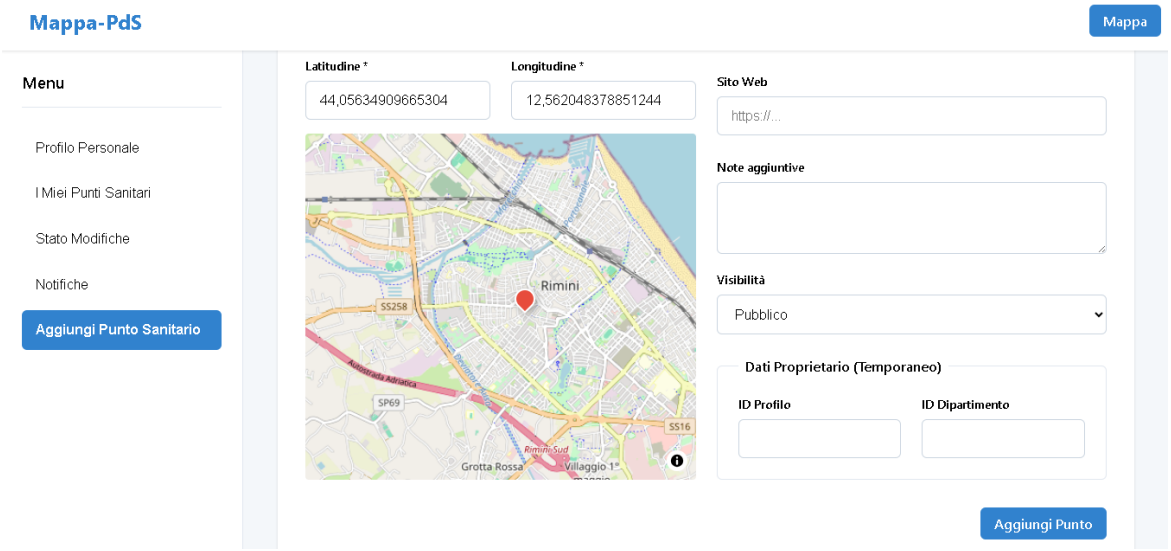

Figura 4.3: Il componente Mappa utilizzato come selettore di posizione nel form di inserimento.

struttura. L'interfaccia cattura immediatamente le coordinate di latitudine e longitudine, popolando automaticamente i campi corrispondenti nel *form*.

- ***Marker* di Posizione**: Un indicatore visivo detto *marker* viene posizionato sulla mappa nel punto selezionato, fornendo un feedback chiaro e immediato all'utente. Questo *marker* si sposta a ogni nuovo clic, permettendo di rifinire la posizione con facilità.
- **Interfaccia Semplificata**: L'interfaccia è minimale per focalizzare l'utente sull'unico compito richiesto: la selezione di una posizione geografica. In tal senso il cursore a forma di croce rinforza ulteriormente l'idea di uno strumento di puntamento.
- **Livelli Geografici di Contesto**: Per aiutare l'orientamento, la mappa mostra strati informativi con i confini delle province e dei comuni, che appaiono e scompaiono dinamicamente in base al livello di zoom, fornendo riferimenti geografici utili durante la selezione.

Conclusioni

Il presente elaborato di tesi si è posto l'obiettivo di analizzare, progettare e prototipare un sistema per la mappatura Punti eoreferenzati per l'AUSL Romagna. Il percorso, si è articolato attraverso un approccio metodologico Agile e grazie ai principi del Domain-Driven Design ha permesso di tradurre un insieme di requisiti inizialmente incerti in un artefatto software funzionante, capace di validare le ipotesi architetturali e di fungere da catalizzatore per il dialogo con gli esperti di dominio.

Questo capitolo finale si propone di tracciare un bilancio critico dell'intero progetto. Verranno analizzate e validate, in modo distinto, le tre fasi cruciali del workflow evidenziandone i punti di forza, le debolezze intrinseche e le scelte strategiche. Infine, partendo dalle lacune evidenziate nel prototipo, verrà delineata una roadmap pragmatica per i futuri sviluppi, trasformando il lavoro svolto nella solida base per un'evoluzione industriale del sistema.

Validazione della Fase di Analisi

La fase di analisi ha rappresentato il fondamento su cui si è costruito l'intero progetto, navigando un contesto caratterizzato da forte incertezza e requisiti non consolidati. La sua validazione rivela sia la robustezza delle metodologie adottate sia i limiti intrinseci di un'analisi puramente teorica in un ambiente dinamico.

Il successo principale di questa fase risiede nella scelta e nell'applicazione rigorosa delle metodologie. L'adozione del paradigma Agile si è rivelata non una semplice preferenza, ma una necessità strategica che ha permesso al progetto di avviarsi e progredire nonostante l'assenza di una visione unificata da parte degli stakeholder. Parallelamente, l'approccio DDD ha fornito gli strumenti per gestire la complessità intrinseca del dominio sanitario. L'elaborazione di un Linguaggio Ubiquo condiviso, si è dimostrata cruciale per superare le ambiguità terminologiche, agendo da catalizzatore non solo per il team di sviluppo, ma anche per gli stessi esperti del dominio nel formalizzare la propria conoscenza.

La principale debolezza del processo di analisi è stata la sua dipendenza dalla disponibilità e dall'allineamento degli stakeholder, aggravata da risorse temporali limitate. Gli incontri con gli esperti del dominio, infatti, sono stati dilazionati ampiamente nel tempo, generando lassi temporali durante i quali il progetto era a rischio. In queste fasi di attesa, la volontà di procedere ha introdotto il pericolo di svolgere lavoro speculativo, ovvero basato su assunzioni non ancora validate, con il conseguente rischio di disallineamento rispetto alle specifiche in evoluzione. Proprio questa criticità, tuttavia, funge da parametro di validazione della metodologia scelta. Dimostra infatti, inequivocabilmente come, in contesti organizzativi complessi, l'analisi puramente teorica raggiunga presto i propri limiti e come la produzione rapida di un artefatto tangibile sia indispensabile per accorciare i cicli di feedback e mitigare i rischi.

Validazione della Fase di Progettazione

La fase di progettazione ha tradotto le esigenze astratte emerse dall'analisi in un'architettura software robusta e scalabile.

La scelta di un'architettura a tre livelli si è confermata vincente, garantendo un disaccoppiamento netto tra presentazione, logica di business e accesso ai dati, in piena sinergia ai requisiti non funzionali di manutenibilità e flessibilità. Un altro punto di forza è la progettazione del database. Il modello Entità-Relazione non si è limitato a supportare le funzionalità del prototipo, ma ha incluso entità chiave per la realizzazione dei requisiti funzionali proposti, anticipando le esigenze future e garantendo che il sistema potesse evolvere senza richiedere una profonda ristrutturazione dello schema. Infine, l'adozione di pattern consolidati come DTO ha assicurato la definizione di un contratto API stabile e sicuro.

Il principale limite della progettazione risiede nella sua stessa ambizione. L'architettura ideata, in particolare per la gestione della sicurezza attraverso i pattern RBAC e ACL e delle notifiche, è complessa e la sua implementazione completa rappresenta una sfida significativa, che va oltre gli scopi di una singola prototipazione. Se da un lato questo garantisce la scalabilità futura, dall'altro ha creato un divario tra la completezza del disegno architetturale e la sua parziale realizzazione pratica.

Validazione dell'Implementazione Prototipale

Il prototipo sviluppato rappresenta la sintesi concreta delle scelte di analisi e progettazione. La sua validazione evidenzia sia i successi funzionali sia le lacune implementative deliberate.

A livello di persistenza, il prototipo presenta una fondazione solida. Il modello Entità-Relazione è stato tradotto fedelmente in uno schema `PostgreSQL`, sfruttando appieno le sue capacità relazionali e l'estensione `PostGIS`. Le entità centrali del dominio: `Punto della Salute`, `Servizio` e `Categoria`; sono pienamente operative. La principale lacuna risiede nell'utilizzo parziale dello schema: le tabelle progettate per supportare le funzionalità avanzate esistono ma sono in gran parte dormienti, agendo come *placeholder* architetturali in attesa di essere attivate dalla logica applicativa.

L'architettura a strati del Backend rappresenta uno dei successi più significativi, con un'implementazione robusta del pattern *Controller-Service-Repository* che valida le scelte tecnologiche basate su `Spring Boot`. Le mancanze sono una diretta conseguenza della strategia di prototipazione e si concentrano sulla logica di business non implementata:

- **Assenza di un Layer di Sicurezza:** Il *backend* è attualmente privo di qualsiasi meccanismo di protezione. Gli *endpoint API* sono pubblici e non implementano i controlli di autenticazione né di autorizzazione previsti dal modello *RBAC* e *ACL* progettato. Manca l'integrazione con *framework* dedicati come `Spring Security` per proteggere le risorse e gestire il contesto utente.

- **Logica di Dominio Avanzata non implementata:** I *Service* si limitano a orchestrare le operazioni *CRUD* di base. La logica di dominio più complessa, pur essendo stata progettata, non è stata tradotta in codice. Nello specifico, mancano i meccanismi per la gestione del ciclo di vita di un'entità `Modifica` e la logica *Event-Driven* per la pubblicazione di eventi di dominio e la conseguente creazione di `Notifiche`.

- **Mancanza dei Moduli di Interoperabilità:** Non sono stati sviluppati i moduli applicativi necessari per dialogare con sistemi esterni. Questo include sia le funzionalità di importazione e sincronizzazione dati, sia l'esposizione di *endpoint API* conformi allo standard sanitario HL7 FHIR, lasciando il sistema, allo stato attuale, come un silo informativo autonomo.

Il *frontend* è la componente dove il successo del prototipo come strumento di dialogo è più evidente. L'interfaccia, costruita su `React`, è reattiva e implementa eccellentemente il nucleo funzionale di visualizzazione e interazione con la mappa. Le lacune riflettono le mancanze del *backend*, con un'interfaccia che agisce spesso come una facciata per validare l'esperienza utente:

- **Simulazione di Funzionalità Collaborative:** Le viste per la gestione di modifiche e notifiche sono state implementate usando *mock data* per raccogliere feedback prima dello sviluppo della logica sottostante.

- **Assenza di un Contesto Utente:** L'applicazione è priva di un flusso di login e non ha cognizione di un utente autenticato, presentando un'esperienza utente unica e priva di personalizzazioni basate sui ruoli.

Sviluppi Futuri

Le lacune implementative identificate nella fase di validazione non rappresentano fallimenti, ma delineano una chiara e pragmatica *roadmap* per l'evoluzione futura del sistema. Gli sviluppi si articolano come il naturale completamento della visione progettuale:

- **Implementazione del Modello di Sicurezza Completo**: Il passo successivo più critico è l'attivazione del sistema di autorizzazioni progettato. Questo include l'implementazione del *backend* per la gestione di Utenti, Ruoli e Permessi, l'integrazione con i sistemi di autenticazione aziendali e l'applicazione delle *policy* di visibilità a livello di dato.

- **Attivazione dei Workflow Collaborativi**: Completare l'implementazione del *Proposal Pattern* per la gestione delle modifiche, sviluppando le interfacce per la revisione e l'approvazione. Contestualmente, attivare l'architettura *Event-Driven* per generare Notifiche in risposta agli eventi di dominio.

- **Realizzazione dell'Interoperabilità**: Sviluppare i moduli per l'integrazione con fonti dati esterne per trasformare il sistema nella *single source of truth* aziendale. Parallelamente, avviare l'implementazione di *endpoint API* conformi allo standard HL7 FHIR per garantire l'integrazione con l'ecosistema sanitario.

- **Consolidamento della Strategia di Testing**: Affiancare allo sviluppo una solida *pipeline* di test, includendo *unit test* per i *service*, test di integrazione per il database e test *end-to-end* per i flussi utente critici.

Ringraziamenti

Il percorso che oggi si sta concludendo ha coinvolto molti più ambiti della mia persona di quanto mai avrei potuto immaginare. È stata una sfida che mi ha portato a conoscere e scoprire molto più su me stesso che sull'ambito di studi intrapreso. Non fraintendetemi, mi è capitato anche di imparare qualcosa sull'informatica, ma la bellezza di questo viaggio l'ho colta pienamente in altri momenti. Primo tra tutti è stato lo spingersi ben oltre i miei limiti dimostrando a me stesso quanto sia facile cadere ed al contempo quanto sia gratificante sapersi rialzare per guardare in faccia le difficoltà. Quella capacità di rialzarsi è il frutto di un lavoro di presa di consapevolezza, lungo ed articolato, che fortunatamente ho potuto condividere con tutti voi. Sono proprio le persone meravigliose che oggi mi circondano a rendere eccezionalmente sorprendente questa giornata di festa: non un titolo, non una cerimonia, né tantomeno una corona, ma voi tutti qui riuniti.

Per primi voglio ringraziare la Nati ed Andre, i miei genitori. Siete da sempre i miei fari, primi punti di riferimento e modelli. Maestri indiscussi di umiltà, mi avete insegnato che nella vita non serve a nulla primeggiare, specialmente se ciò va a discapito di chi si ama. Seguendo il vostro esempio ho scoperto che è l'avere la possibilità di condividere le difficoltà con le persone a noi care a profumare la vita della stessa fragranza della primavera. Vi ringrazio per non aver mai smesso di credere in me. Anche quando il buio mi ha circondato e non sentivo di poter dare qualcosa al mondo siete rimasti lì a sostenermi. Lo avete fatto quel tanto che è bastato a non farmi sprofondare ed al contempo lo stretto indispensabile per darmi la possibilità di sbagliare e capire come vivere la vita senza lasciarsi trasportare caoticamente da essa.

Un secondo grazie va ai miei fratelli Linda ed Alessandro, i piccolini di casa ormai tutt'altro che piccoli, tanto nel corpo quanto nella mente. Siete persone splendide alle quali è toccato subire un fratellone spesso stanco e provato. Ciò nonostante mai una volta mi avete fatto mancare quella parola, quella battuta o quella spinta che sono valse più di mille gesti. Grazie per la dinamicità che contribuite a portare in casa, per gli stimoli e le emozioni con cui giorno dopo giorno avete riempito la mia vita.

Alle nonne Norma e Gabriella rivolgo un ringraziamento sincero come l'a-

more che mi avete sempre dimostrato. Siete due donne diverse, accumunate dall'affetto che sempre avete conservato per i vostri nipoti. Un affetto profondo fatto di cura e di dolcezza. Fatto di storie che raccontano il vissuto della nostra famiglia e di gesti che ne dimostrano l'unità. Siete persone splendide che mi hanno insegnato il rispetto e la dedizione sotto le forme più disparate. A voi rivolgo il mio grazie perché siete da sempre e per sempre testimoni di quanto sia bello e potente portare la gentilezza nel rapporto con l'altro senza aspettarsi nulla in cambio, se non un sorriso.

A Giuseppe, lo zio che in questi anni si è fatto un po' nonno, rivolgo la mia gratitudine. Ti sei preso tante responsabilità e, così facendo, hai contribuito in maniera essenziale al farmi essere qui oggi. A quell'uomo capace sempre di portare un sorriso a tavola e di vedere il bello anche quando si nasconde in fondo alla fatica ed alla difficoltà, grazie per i sacrifici che giorno dopo giorno dedichi alla tua famiglia.

Un caloroso abbraccio di riconoscenza lo dedico agli amici di sempre, quelle persone tanto pazze da aver deciso un giorno di volermi stare accanto. Se sono qui è perché ho sempre trovato una spalla su cui appoggiarmi, un sorriso su cui contare e un abbraccio più potente di mille belle parole. Siete tra le persone più importanti della mia vita e queste poche frasi non bastano a raccontare quanto significativo sia il vostro essereci al di là delle avversità. Alla Sofi, al Dire, all'Alo, a Frency, a Babe, alla Mati, ad Eli e a Pie rivolgo la mia riconoscenza, oggi e sempre, con un affetto sincero ed indissolubile costruito nel tempo e destinato a durare, solido come la passione che mi avete donato in questi anni.

Mi rivolgo ora alla massa informe e straordinaria di persone che ho conosciuto qui dentro. Quando ho detto che questo posto mi ha formato più come persona che come informatico non erano parole al vento, e ciò è tutto merito vostro. Avrei voluto ringraziarvi uno ad uno ma, nello scrivere queste parole, mi sono reso conto di essermi un po' troppo abbandonato alle chiacchiere in questi anni. Ho sottratto molti di voi allo studio un numero di volte che oramai è fin troppo complesso calcolare, il che mi rende pressoché impossibile ringraziarvi uno ad uno. Ricordo con emozione il primo giorno qui, quando con timore abbiamo chiesto dove mangiare in pausa pranzo e ci è stata menzionata la polivalente. Non sapevo ancora quanto quell'ambiente di studio a tutto sarebbe servito meno che a studiare. Ci abbiamo giocato, ci abbiamo discusso, ci abbiamo sognato ma soprattutto ci siamo incontrati. Devo a voi tutti la malinconia di lasciare un posto che mi fa stare bene, mi fa sentire accolto e spronato, ma anche l'orgoglio di uscire da qui carico di rapporti con persone meravigliosamente uniche.

Un ultimo pensiero va alla donna che ha sistemato questo completo per oggi. Da piccolino mi chiedevo come una persona così piena di meraviglia,

forza e generosità potesse non gestire un asilo. Al tempo lo vedevo come il mestiere per antonomasia delle persone buone nello spirito. Negli anni non hai mai aperto quell'asilo ma continuo a considerarti sempre la mia maestra. Insegnante di forza e coerenza, dimostri giorno dopo giorno come giocare al gioco della vita, osservando tutto con gli occhi meravigliati del bambino. Grazie Aldina.

Oggi non si conclude semplicemente una narrazione, ma si celebra ogni avventura vissuta assieme a voi. Questa tesi porta il mio nome, ma la sua vera forza risiede nelle fondamenta che, giorno dopo giorno, avete costruito per me: la vostra fiducia, il vostro affetto e la vostra infinita pazienza. Con questo zaino di lezioni e di amore, sono pronto a iniziare a camminare verso una nuova meta, sapendo di non essere mai solo.

A tutti voi, dal profondo del cuore, il mio grazie.

Bibliografia

[1] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001.

[2] Giulio Destri. Sviluppo software agile. Technical report, apr 2007.

[3] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[4] Fondazione openpolis. geojson-italy: Confini amministrativi italiani georeferenziati, 2023. Repository GitHub. Dati originali ISTAT rilasciati con licenza CC-BY.