POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Cartella Clinica Informatizzata

Progetto e implementazione di un backend a microservizi

Relatori
prof. Carla Fabiana Chiasserini
prof. Guido Pagana

Candidato
Giovanni Cravero

Anno accademico 2022-2023

Sommario

Il centro ipertensione arteriosa del reparto di Medicina Interna 4 dell'ospedale Molinette di Torino necessita di un software per la gestione dei pazienti che permetta di inserire informazioni cliniche di varia natura e la cui struttura può variare nel tempo. Inoltre, queste informazioni devono essere disponibili a più utenti diversi, medici e specializzandi, e su diversi dispositivi di accesso (computer con diversi sistemi operativi, tablet, ...).

L'applicazione utilizzata attualmente non soddisfa pienamente questi requisiti. In particolare, il software utilizzato è progettato per essere eseguito solo su macchine tra loro indipendenti equipaggiate con il sistema operativo Windows™. Questa soluzione limita la diffusione dell'utilizzo dell'applicazione e costringe gli utenti a condividere tramite supporti di archiviazione esterni gli aggiornamenti effettuati sul sistema affinché vengano allineati manualmente gli archivi presenti sui diversi computer in dotazione al reparto.

Lo scopo di questa tesi consiste nella progettazione di un prototipo per un **backend**, denominato *Cartella Clinica Informatizzata*, che permetta di gestire un archivio delle informazioni relative ai pazienti e sia disponibile in un ambiente distribuito accessibile tramite la rete internet (o una rete intranet) da applicazioni di frontend progettate per essere eseguite su qualsiasi tipo di dispositivo.

L'applicazione, sviluppata con il linguaggio `Kotlin` ed il framework `Spring`, è organizzata in microservizi, ognuno con un compito specifico. La suddivisione in microservizi garantisce maggiori scalabilità, flessibilità e affidabilità offrendo una modalità più semplice per apportare future implementazioni. I servizi vengono mantenuti il più possibile disaccoppiati per permettere di incrementare in futuro le opzioni offerte, aggiungendo nuove funzionalità senza dover modificare la parte restante del progetto.

Il progetto contiene il package "*it.polito.links.ccinfo.data*" sviluppato per permettere di utilizzare l'accesso al database senza conoscerne l'effettiva implementazione. I servizi offerti dal package, infatti, utilizzano delle API per trasferire i dati dal database ai microservizi e viceversa senza esporre direttamente l'accesso al database sottostante: questa soluzione permette di sostituire il database con la semplice modifica del package. Nelle conclusioni verrà evidenziato un possibile sviluppo futuro che preveda la sostituzione del package con alcuni microservizi che, sfruttando la semantica di `GraphQL`, possono fornire un ulteriore astrazione per rendere completamente indipendenti tutti i microservizi.

Per la distribuzione dell'applicativo, il progetto si affida a due strumenti di virtualizzazione e gestione dei processi: `Docker` per la creazione dei container e a `Kubernetes` per l'esecuzione ed il controllo dei servizi. La creazione dei container garantisce la possibilità di gestire la versione del software e la sua distribuzione in contesti diversi, sia per quanto riguarda l'hardware utilizzato, sia in relazione ai diversi sistemi operativi dei server utilizzati, sia per la possibilità di installare l'applicazione nelle infrastrutture presenti nel cloud.

La caratteristica principale di questo progetto consiste nella scelta di archiviare le informazioni su una database documentale (`MongoDB`) che rende possibile l'archiviazione delle informazioni senza una struttura definita a priori. Benché non fornisca i vantaggi di un database relazionale per la gestione dell'integrità referenziale e il reperimento delle informazioni complesse attraverso le viste, questa soluzione offre la possibilità di definire e salvare sul database stesso la struttura dei dati tramite degli schemi definiti con le specifiche di `JSON Schema, versione 2020-12`. Questa soluzione permette da un lato ai client di comporre dinamicamente l'interfaccia utente in base al formato dei dati che dovrà essere archiviato sul database, dall'altro di fornire un mezzo per validare i dati che vengono archiviati. In questo modo future variazioni ed implementazioni dei tipi di dato che si vogliono gestire con l'applicativo non richiederanno delle modifiche al progetto ma semplicemente la ridefinizione dell'insieme di schemi presenti sul database offrendo una maggiore capacità del sistema ad adattarsi a nuove esigenze, eventualmente anche per reparti dell'ospedale diversi dall'attuale committente. Seguendo le specifiche di `JSON Schema`, la definizione degli schemi permette di inserire i criteri di validazione dei singoli campi che possono così essere controllati senza codificare direttamente all'interno dei programmi le specifiche di verifica.

Per effettuare la validazione degli schemi è stata utilizzata la libreria `jsonsKema`. In questa prima versione del progetto la validazione avviene esclusivamente in base alle specifiche di `JSON Schema, versione 2020-12` ma, in futuro, come proposto nelle conclusioni, potrà essere esteso il package di validazione in modo da poter eseguire controlli più raffinati eventualmente inserendo degli script di validazione all'interno del database.

Infine, il package "*it.polito.links.ccinfo.security*" implementa i `Bean` che possono essere iniettati nei microservizi che richiedono delle restrizioni di accesso. L'accesso ai microservizi è ammesso solo per gli utenti autenticati, ovviamente fatta eccezione per il microservizio che espone l'endpoint per il `Single Sign On`. Inoltre, tramite le annotazioni messe a disposizione da `Spring Security`, i microservizi possono abilitare gli accessi agli endpoint in base al tipo di utenza.

Indice

Capitolo 1

Introduzione

Il centro ipertensione arteriosa del reparto di Medicina Interna 4 dell'ospedale Molinette di Torino gestisce i pazienti utilizzando il software proprietario `HyperMacondo versione 2.0`, sviluppato con lo strumento `FileMaker PRO versione 12`. Il software presenta alcune limitazioni che ne richiedono una revisione consistente. Lo scopo di questo progetto è quello di fornire uno strumento innovativo che permetta di gestire in modo flessibile e facilmente aggiornabile i pazienti del reparto e che potrà costituire una solida base per lo sviluppo futuro di una cartella clinica elettronica e per essere integrato con il fascicolo sanitario elettronico.

1.1 Cartella Clinica Informatizzata

Definiamo per questo progetto la cartella clinica informatizzata (CCINFO) come un sistema elettronico utilizzato per la registrazione, la gestione e la condivisione delle informazioni cliniche dei pazienti. Sostituisce la tradizionale cartella clinica in formato cartaceo e consente un accesso rapido, sicuro ed efficiente ai dati medici.

Una CCINFO contiene una vasta gamma di informazioni relative al paziente, come anamnesi, risultati di test di laboratorio, immagini diagnostiche, prescrizioni di farmaci, note degli operatori sanitari, referti e comunicazioni per altri medici, registrazioni degli appuntamenti, informazioni di assicurazione e altro ancora. Queste informazioni sono organizzate in modo strutturato e possono essere facilmente recuperate e visualizzate dai professionisti sanitari autorizzati in diversi reparti e sedi.

Le CCINFO offrono numerosi vantaggi rispetto alle cartelle cliniche tradizionali. Innanzitutto, migliorano l'accessibilità e la disponibilità delle informazioni. I medici possono accedere alle cartelle cliniche dei pazienti in modo remoto, consentendo una migliore continuità delle cure e una consultazione più rapida delle informazioni necessarie. Inoltre, le CCINFO riducono il rischio di errori di lettura o scrittura illeggibile, migliorando la precisione e la chiarezza delle informazioni registrate.

Le CCINFO facilitano anche la condivisione sicura dei dati tra i membri del team sanitario. I professionisti possono collaborare e comunicare più facilmente, consentendo un coordinamento più efficace delle cure. Inoltre, i sistemi di CCINFO possono generare avvisi e promemoria per i professionisti sanitari, ad esempio suggerendo interazioni farmacologiche potenzialmente pericolose, avvertendo di allergie note del paziente o suggerendo percorsi diagnostici sulla base di esperienze consolidate.

La sicurezza e la privacy dei dati sono elementi fondamentali nella gestione delle CCINFO. Le informazioni dei pazienti sono protette da rigorose misure di sicurezza informatica, adottando metodi di crittografia e controlli di accesso per garantire che solo il personale autorizzato possa visualizzare o modificare i dati. Inoltre, i sistemi di CCINFO devono essere conformi alle normative sulla privacy, come ad esempio il Regolamento generale sulla protezione dei dati (GDPR) nell'Unione europea.

In sintesi, una CCINFO è un sistema elettronico che consente la gestione, l'accesso e la condivisione efficiente delle informazioni cliniche dei pazienti da parte del personale sanitario ma che non prevede la completa dematerializzazione dei documenti sanitari. Infatti, si differenzia dalla *Cartella Clinica elettronica* dalla necessità di produrre i referti da consegnare ai pazienti in formato cartaceo. I documenti prodotti devono essere siglati dal personale sanitario affinché abbiano valore legale. Inoltre, non prevede l'accesso diretto ai dati da parte dei pazienti.

1.2 Esigenze reparto, elenco dei requisiti

L'analisi effettuata con il personale medico del reparto interessato nell'utilizzo del software ha evidenziato quali sono le caratteristiche necessarie affinché il servizio ambulatoriale possa essere erogato. In questo paragrafo viene data una descrizione dettagliata di queste esigenze.

Alcune di queste esigenze sono legate in tutto o in parte allo sviluppo del frontend, che non è oggetto di questa tesi. Nel presente paragrafo vengono ugualmente presentate in quanto hanno fatto parte dell'analisi effettuata con il personale medico e perché impattano comunque sulle soluzioni adottate lato backend[1]. Il cap. 3, che presenta l'implementazione del progetto, descrive se, come e perché vengono soddisfatti i requisiti qui descritti.

1.2.1 Sicurezza e gestione ruoli utenza

Il sistema deve essere protetto e tutti gli utenti che accedono devono farlo tramite un sistema di autenticazione che assegni loro il ruolo ricoperto in modo da poter differenziare le azioni che gli stessi possono compiere. Tutte le funzionalità messe a disposizione e le tecnologie utilizzate devono soddisfare i requisiti previsti dalle norme sulla protezione dei dati e fornire per questi ultimi sufficienti livelli di sicurezza sia per il loro trasferimento sia per la gestione cosiddetta a riposo.

Sono state individuate tre tipologie di utenti:

Amministratori i responsabili del reparto che possono censire gli utenti, intervenire sulla definizione dei dati, modificare le interfacce utente in relazione alla modifica dei dati sottostanti, oltre, ovviamente, ad effettuare tutte le operazioni previste per gli altri ruoli;

Medici che devono poter seguire la manutenzione dei dati con o senza il supporto degli utenti generici e devono completare il processo diagnostico con l'emissione del referto da siglare e consegnare ai pazienti. Devono inoltre poter estrarre i dati dall'applicativo al fine di produrre l'input per programmi di analisi statistica;

Utenti gli utenti generici, spesso identificabili con studenti e specializzandi, destinati a fornire supporto ai medici per l'inserimento degli esami di laboratorio o dei dati relativi ai pazienti.

1.2.2 Sistema distribuito e accessibilità offline

L'applicativo deve funzionare in un ambiente distribuito permettendo agli utenti di connettersi ad unico database, senza la necessità di effettuare delle attività di sincronizzazione tra i diversi dispositivi.

Inoltre, per favorire lo spostamento tra gli ambulatori e l'utilizzo di dispositivi portatili non sempre connessi alla rete, il sistema deve permettere di eseguire le operazioni di lettura ed aggiornamento dei dati quando il dispositivo è offline. In una fase successiva, quando la linea torna disponibile ed una volta che il dispositivo è nuovamente connesso, le operazioni effettuate dall'utente devono venire sincronizzate con il database centrale senza che sia necessario l'intervento manuale dell'utente.

[1] La scelta del DB e l'architettura della soluzione deve tener conto di tutte le esigenze espresse dall'utenza

1.2.3 Produzione referti

Il requisito indica la necessità di produrre dei referti che, partendo da un modello standard, possano essere modificabili dall'utente prima di essere stampati. Le modifiche effettuate dovranno essere salvate sul database in modo da poter essere ricaricate dal frontend per ulteriori modifiche ed implementazioni. Il modello standard del referto dovrà essere modificabile dagli amministratori del sistema ed essere generato in modo dinamico recuperando le informazioni variabili dalle visite e dagli esami registrati per i pazienti. Inoltre, dovrà essere garantita la facilità di utilizzo mediante le funzionalità di copia e incolla, ormai di uso comune nelle interfacce utente.

I referti dovranno essere stampabili per essere consegnati ai pazienti e disponibili sia per le singole visite sia per gruppi di informazioni legate al paziente in caso di monitoraggio della pressione arteriosa (ABPM).

1.2.4 Importazione dei dati

Le informazioni relative ai pazienti caricate fino all'adozione del nuovo strumento di gestione devono essere importate correttamente nel sistema. Il software utilizzato attualmente rimarrà in produzione fino a quando non verrà reso disponibile il nuovo software. Tutta l'attività svolta nel periodo di sviluppo del progetto continua ad essere registrata con la vecchia metodologia. Quindi, benché sia stata fornita una prima immagine dei dati quando è stata avviata l'analisi del nuovo prodotto, il progetto deve prevedere una funzionalità di importazione dei dati che permetta di effettuare l'avvio dell'applicativo senza soluzione di continuità rispetto alla vecchia implementazione.

Le operazioni di importazione dei dati dovranno tener in conto che, specialmente per i dati più vecchi, le informazioni potrebbero non essere coerenti con la definizione dei dati nelle tabelle, anche perché il software utilizzato attualmente non effettua controlli di congruità sui dati. La funzionalità di importazione deve quindi prevedere la possibilità di effettuare dei controlli, di correggere ove possibile i vecchi dati e di produrre dei report per facilitare la ridefinizione e la pulizia dei dati da importare.

1.2.5 Esportazione dei dati

Tutte le informazioni inserite nel database devono poter essere estratte per permetterne l'elaborazione a fini statistici e di ricerca. Per la selezione dei dati da estrarre deve essere disponibile un sistema che permetta di specificare se si vuole estrarre l'intero database o quale sottoinsieme utilizzare per quanto riguarda sia i pazienti da elaborare (le righe) sia il tipo di dato da estrarre (le colonne). Il formato dei dati deve prevedere diverse tipologie in modo da poter essere utilizzato da diversi software e deve essere il più possibile personalizzabile.

1.2.6 Formato e amministrazione dei dati

I dati devono tenere conto delle continue evoluzioni che la scienza medica e la ricerca ad essa associata comportano. Tutti i parametri devono poter essere gestiti dagli amministratori del sistema in modo da poter modificare sia i controlli da effettuare all'atto dell'inserimento delle informazioni nel sistema sia l'elenco dei valori che l'interfaccia utente presenta all'operatore. I controlli possono riguardare sia i valori limite, eventualmente forzabili, sia delle enumerazioni. Nel caso durante l'importazione dei dati presenti nella vecchia procedura questi non siano congruenti con i nuovi valori, l'importazione deve ugualmente essere effettuata segnalando la casistica nel report di conversione.

Lista farmaci

La lista dei farmaci da assegnare per le terapie deve essere aggiornata periodicamente in modo da tenere conto delle evoluzioni del sistema farmaceutico. Un primo aggiornamento deve essere

effettuato con la sostituzione dell'attuale software (che presenta una lista obsoleta) e poi dovrà prevedere la possibilità di aggiornamento da parte degli amministratori del sistema o dovrà appoggiarsi ad un servizio online.

Flussi diagnostici

Un tipo di dato particolare utile per il supporto alla diagnosi è la definizione dei percorsi diagnostici. Il sistema dovrà contenere un insieme di alberi decisionali che permettano all'utente di seguire un filo logico per determinare se sono necessari ulteriori passaggi nel processo diagnostico e se è stato svolto tutto ciò che occorre per effettuare la giusta diagnosi. Anche nel caso degli alberi decisionali deve essere possibile per gli utenti amministratori effettuare integrazioni e modifiche anche prevedendo l'aggiunta di nuovi alberi.

1.2.7 Utilizzo su qualsiasi dispositivo

La strumentazione in uso nel reparto è eterogenea e, di conseguenza, è stata richiesta la possibilità di utilizzare dispositivi con sistemi operativi differenti per accedere al sistema.

1.2.8 Funzionalità non necessarie in fase iniziale

Durante l'analisi è emerso che alcune funzionalità normalmente previste in software analoghi e descritte nel paragrafo 1.1 non sono indispensabili, per lo meno nella fase iniziale del progetto. Seguendo una filosofia "agile", queste funzionalità non sono state sviluppate rimandando a futuri sviluppi [2] la loro eventuale implementazione.

In particolare sono state escluse la gestione dell'agenda degli appuntamenti, in quanto il reparto utilizza il sistema di prenotazione della sanità piemontese comune a tutti i reparti dell'ospedale, e l'inserimento di immagini collegate ai referti (e.g. ecocardiogrammi, ecografie, ecc.). In quest'ultimo caso, la scelta di non implementare tale funzionalità, oltre al fatto di non essere indispensabile, è legata al peso in termini di spazio occupato e di prestazioni nel trasferimento dei dati tra client e server che avrebbe comportato. In attesa di conoscere su quale hardware potrà essere installato il nuovo software[3], appare giustificato il rinvio ad una prossima evoluzione del progetto.

1.3 Descrizione del software attualmente utilizzato

L'attuale implementazione utilizzata dal reparto implementa solo alcune delle funzionalità e delle caratteristiche definite nel paragrafo 1.1. Si basa sul software proprietario `Filemaker PRO 12` che utilizza un database relazionale.

1.3.1 Funzionalità obsolete

Alcune funzionalità presenti nell'applicativo non vengono utilizzate e quindi saranno ignorate nel nuovo progetto. In particolare, lo strumento per la gestione degli appuntamenti (Agenda) è stato completamente sostituito dai sistemi di prenotazione della sanità piemontese e non necessita di essere sviluppato nella nuova applicazione. Ugualmente anche alcuni tipi di esami, come ad esempio la sezione relativa agli ecocardiogrammi visibile nella figura 1.19, non sono più attuali e non necessitano di essere implementati. In questo caso, però, per conservare le informazioni caricate in passato e permettere ai client di recuperare le informazioni ed eventualmente di visualizzarle, gli schemi relativi agli esami obsoleti verranno mantenute nel nuovo progetto.

[2] vedi il capitolo 4

[3] vedi 3.4.3

1.3.2 Funzionalità mancanti

Il software attualmente utilizzato non implementa alcune funzionalità indispensabili per una corretta gestione dell'ambulatorio. Infatti, l'applicazione non prevede la possibilità di modificare la struttura dei dati e di inserire i controlli sui dati inseriti. Qualsiasi intervento sulla struttura dei dati o dei valori tabellari, come ad esempio l'elenco dei farmaci da utilizzare per indicare la terapia assegnata ai pazienti, sono modificabili solo richiedendo l'intervento degli sviluppatori della piattaforma. Inoltre, non permette di gestire gli utenti in base al ruolo ricoperto e non differenzia l'operatività ammessa. L'esportazione dei dati permette di filtrare le informazioni da estrarre selezionando solo alcuni pazienti o in base alla data di inserimento, in modo da facilitare la sincronizzazione tra i vari dispositivi, ma non di creare una proiezione dei dati basata solo su alcune colonne del database.

Il programma è stato generato per il sistema operativo Windows™, benché la piattaforma utilizzata (`Filemaker PRO versione 12`) preveda la possibilità di pubblicare il software per la piattaforma Apple™. Inoltre non prevede la gestione del database in condivisione tra i vari dispositivi. Infatti, l'applicativo viene installato sui singoli dispositivi del personale medico e accede ai dati direttamente sul disco in cui viene eseguito. È necessaria una fase di sincronizzazione periodica tra tutti i dispositivi per mantenere il database allineato.

1.3.3 Interfaccia utente

Il programma attualmente utilizzato si presenta con una interfaccia obsoleta. L'analisi effettuata congiuntamente ai medici del reparto si è largamente basata sulle maschere di inserimento dati attualmente in vigore. I pazienti sono rappresentati da alcune videate riassuntive a cui vengono aggiunti i dettagli delle visite, degli esami, delle terapie, ecc. Non avendo la possibilità di confronto con lo sviluppatore della precedente versione del programma, per poter comprendere dove risiedono le informazioni presenti nelle maschere di input, è stato creato un paziente fittizio al quale sono state attribuite tutte le informazioni disponibili[4].

Schede riepilogative

La pagina `Anagrafica` (fig. 1.1) presenta, oltre ai dati relativi alla residenza, l'elenco delle visite, degli ecocardiogrammi e i monitoraggi pressori effettuati, presentando per primi i dati più recenti.

La `Scheda riepilogativa` (fig. 1.2) fornisce una visione d'insieme di quanto registrato per il paziente. I dati anagrafici (residenza, età, ecc.) sono contenute nella tabella `Anagrafica`. Le altre informazioni presenti in queste due schede sono provenienti da varie fonti.

[4] I dati inseriti non hanno nessun fondamento scientifico, sono stati creati fittiziamente al solo fine di poter individuare dove risiede l'informazione.

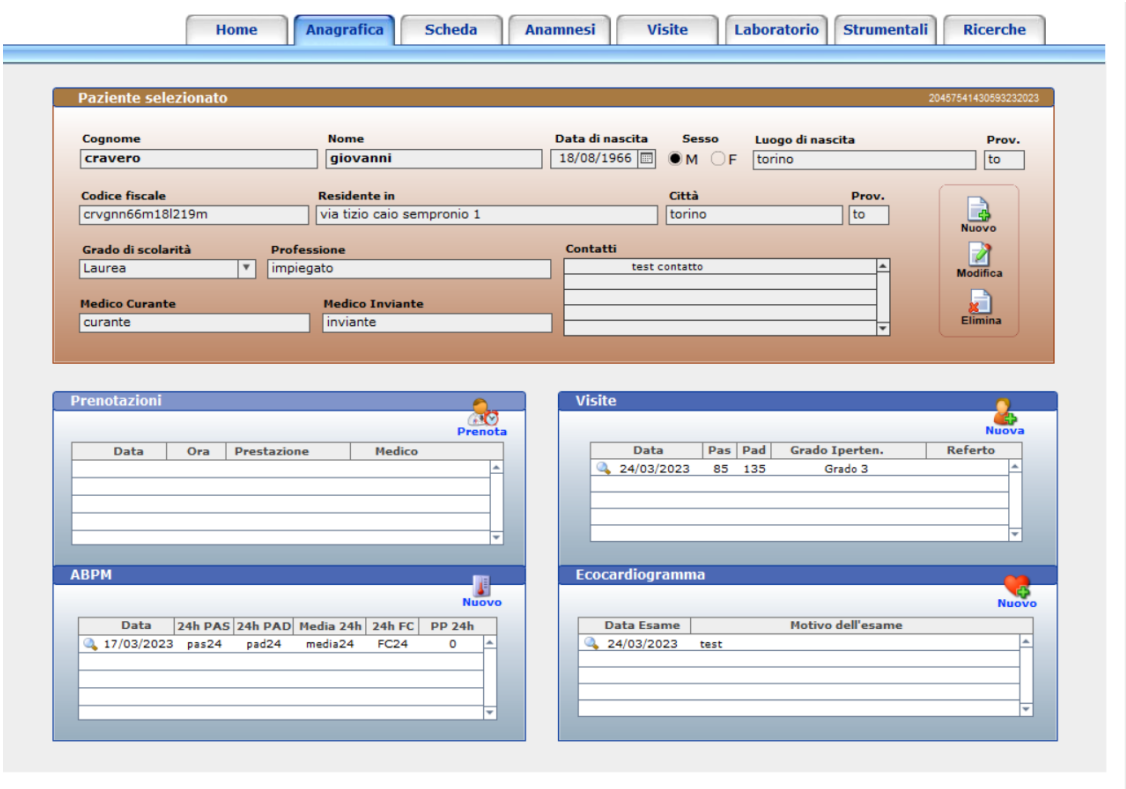


Figura 1.1: Scheda anagrafica

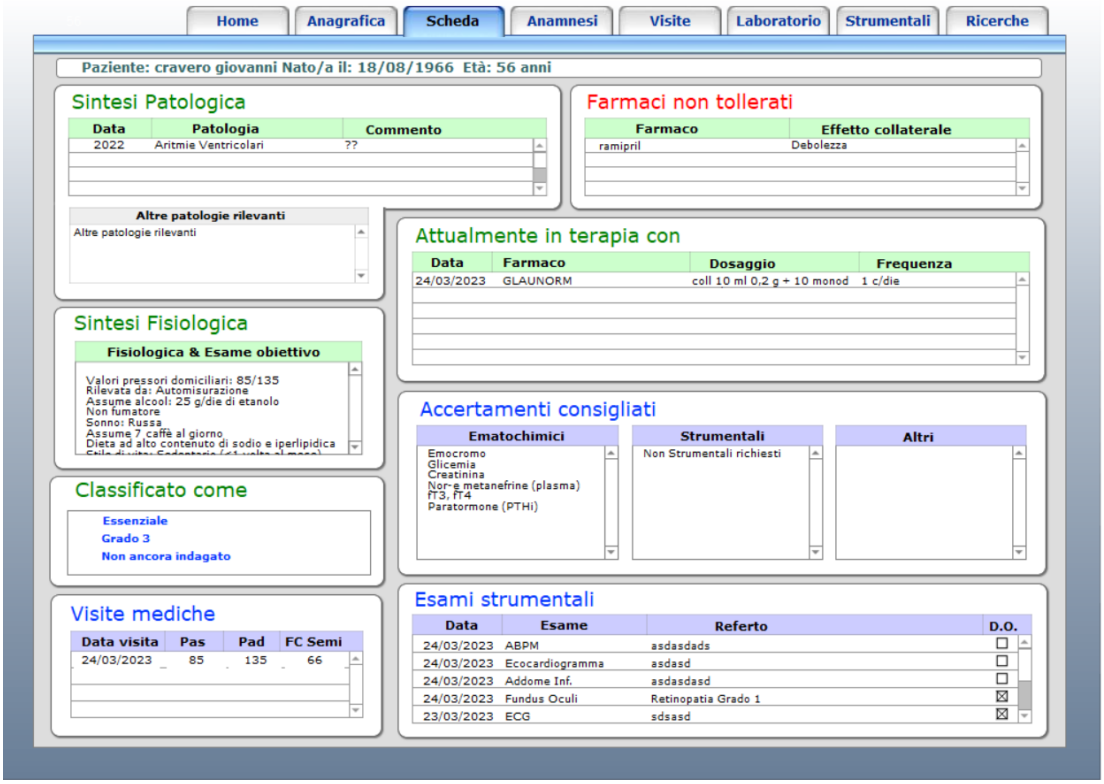


Figura 1.2: Scheda riepilogativa

Monitoraggio pressorio

La figura 1.3 presenta le informazioni caricate per il monitoraggio pressorio, che risiedono interamente nella tabella `ABPM`.

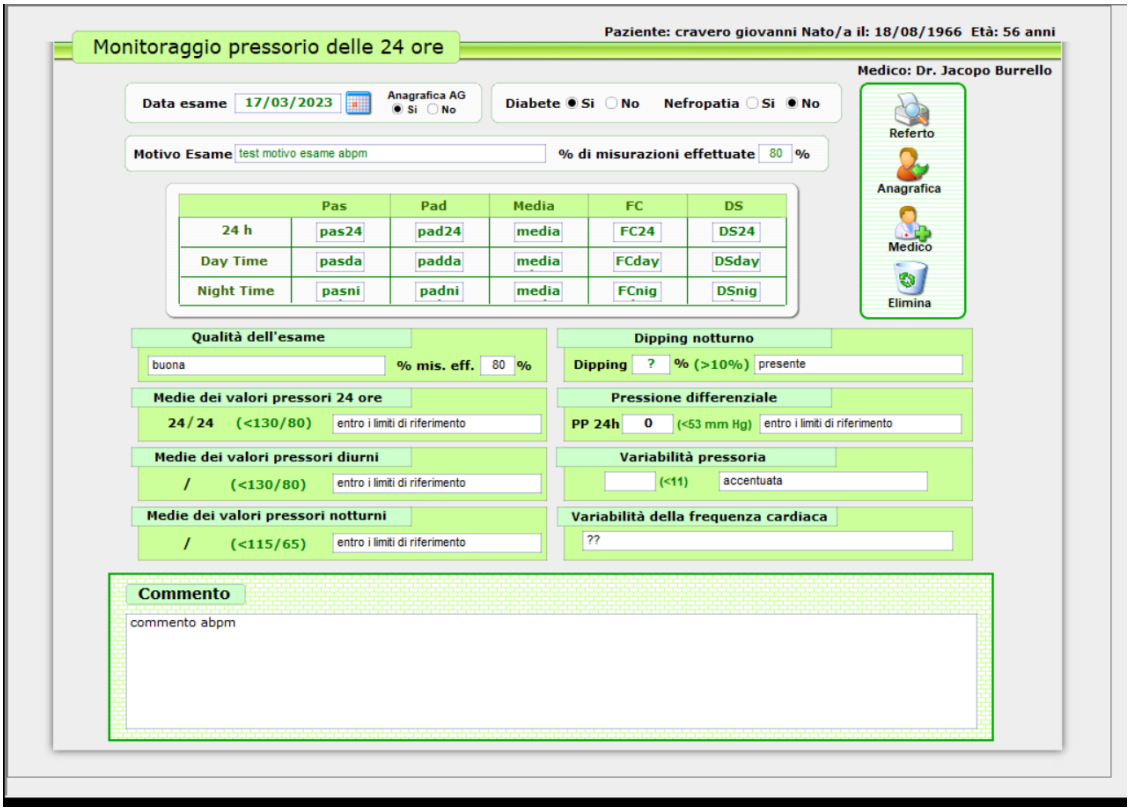


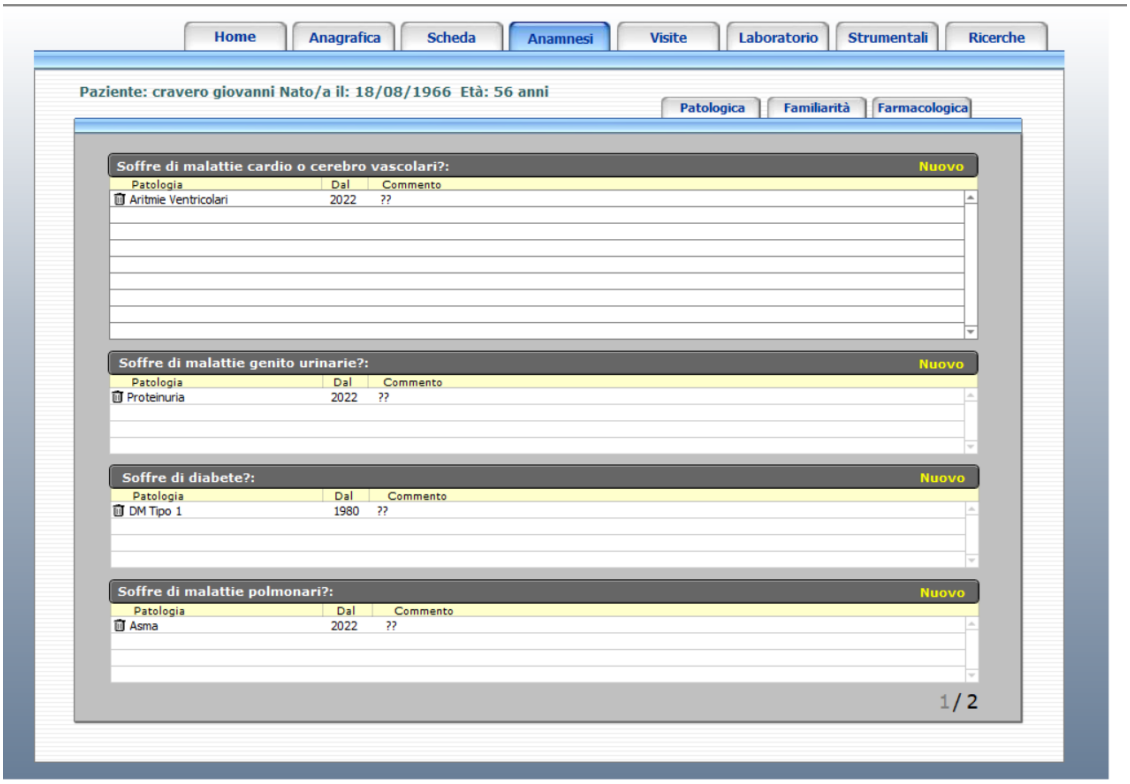
Figura 1.3: Monitoraggio pressorio

Anamnesi

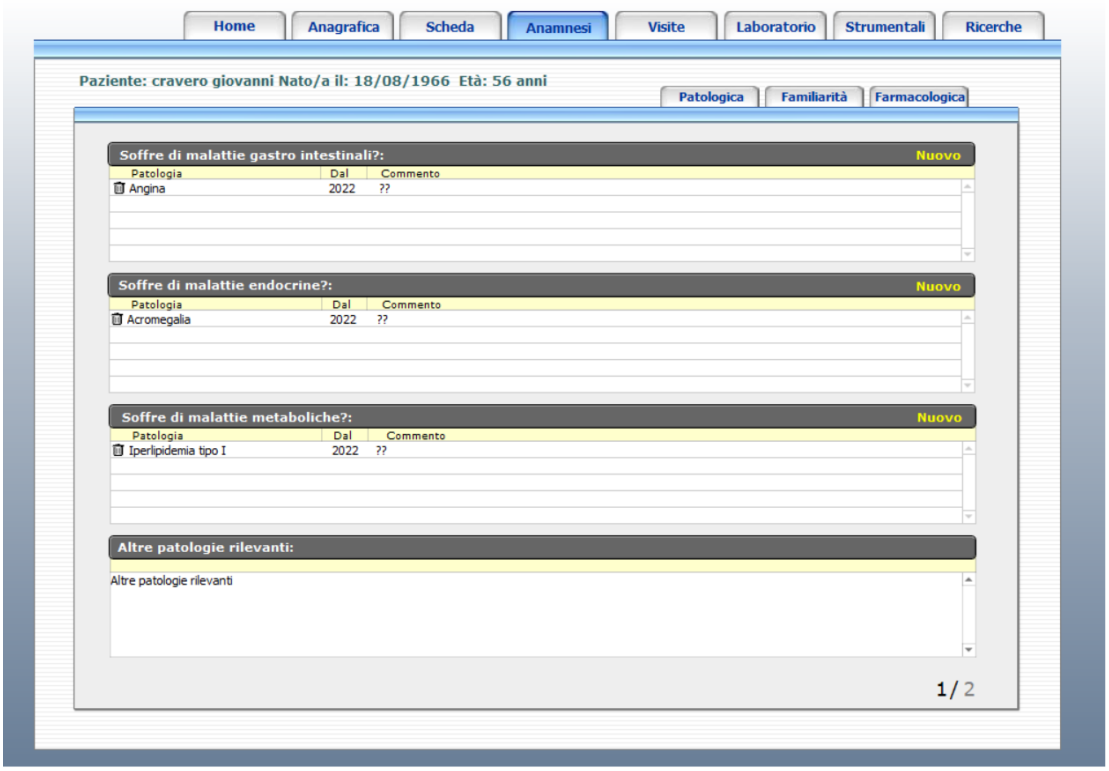
Le informazioni legate all'anamnesi (vedi figure 1.4 e 1.5, 1.6 e 1.7) sono registrate in diverse tabelle. Le patologie riassunte nella figura 1.4 sono caricate nella tabella `Patologie` in cui è presente un campo tipo che identifica a quale gruppo di patologie appartengono (cardiovascolare, genito urinarie, diabete, ecc.). I dati presenti nelle schede riassuntive di figura 1.5 riassumono l'elenco delle patologie già descritte e i sintomi che il paziente segnala. Detti sintomi, la cui pagina di dettaglio è visibile in figura 1.5b, sono archiviati nella tabella `Anagrafica` nei campi `Sintomi` e `Sintomi in occasione`.

La pagina illustrata in figura 1.6 contiene l'elenco delle patologie legate ai parenti dei pazienti che sono registrate nella tabella `Familiarita`

Per quanto riguarda la sezione farmacologica (figura 1.7) i dati sono in parte registrati nella tabella `Terapia` ed in parte nella tabella `Anagrafica`.

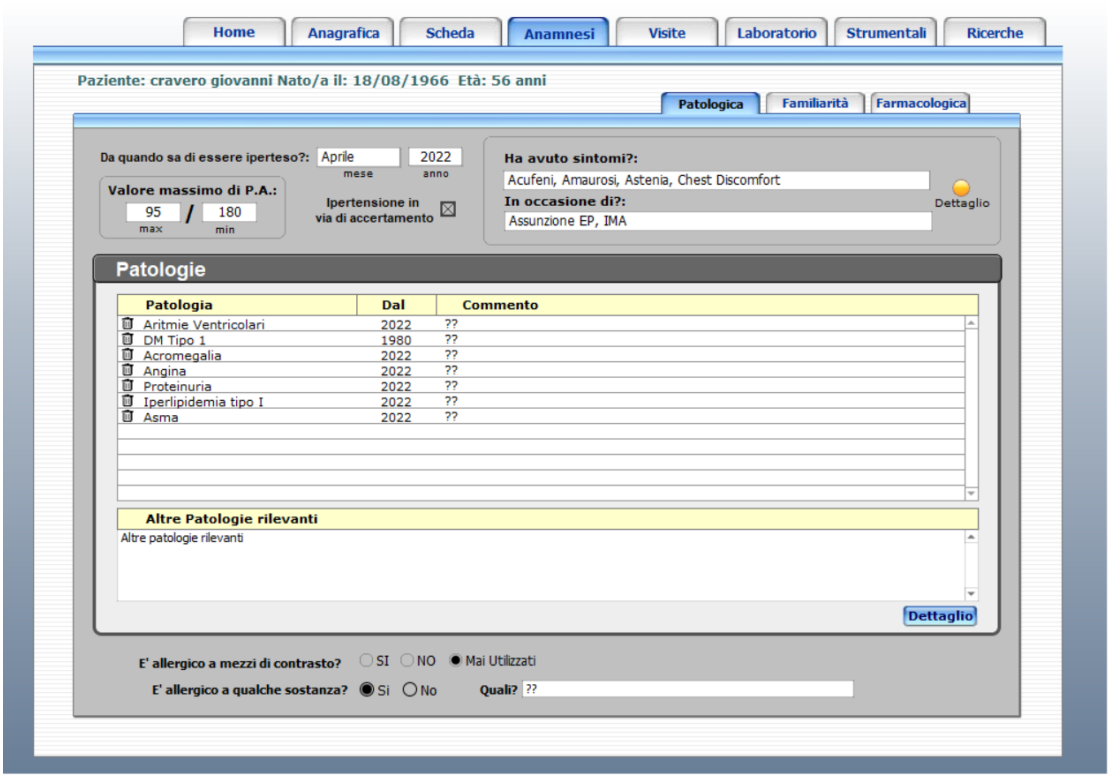


(a) Patologie riepilogo pagina 1

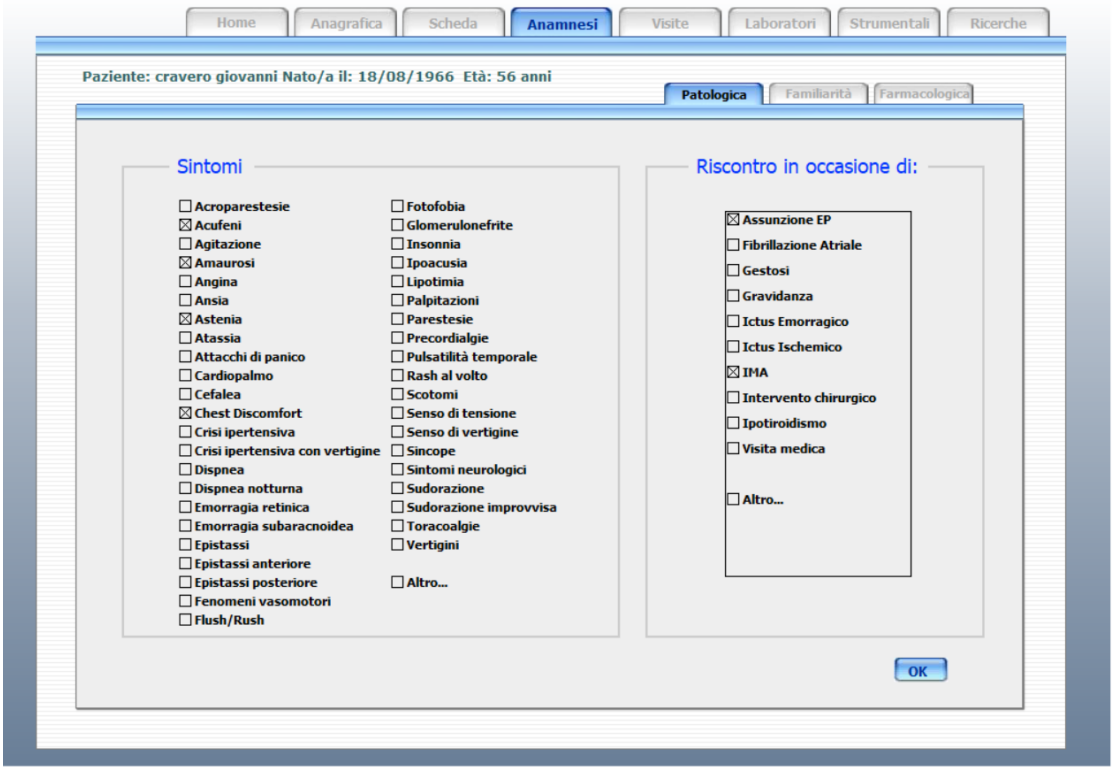


(b) Patologie riepilogo pagina 2

Figura 1.4: Anamnesi - riepilogo patologie



(a) Sintesi patologica



(b) Dettaglio sintomi

Figura 1.5: Anamnesi - scheda patologica

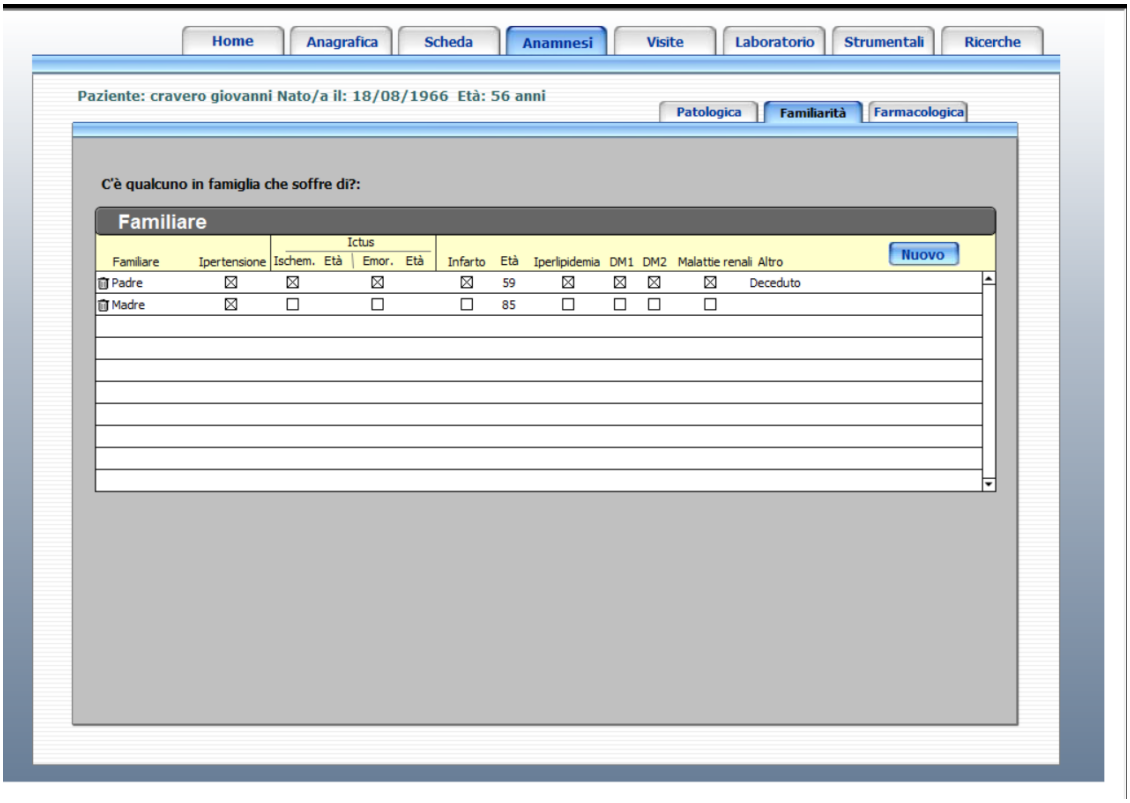


Figura 1.6: Anamnesi - sezione familiarità

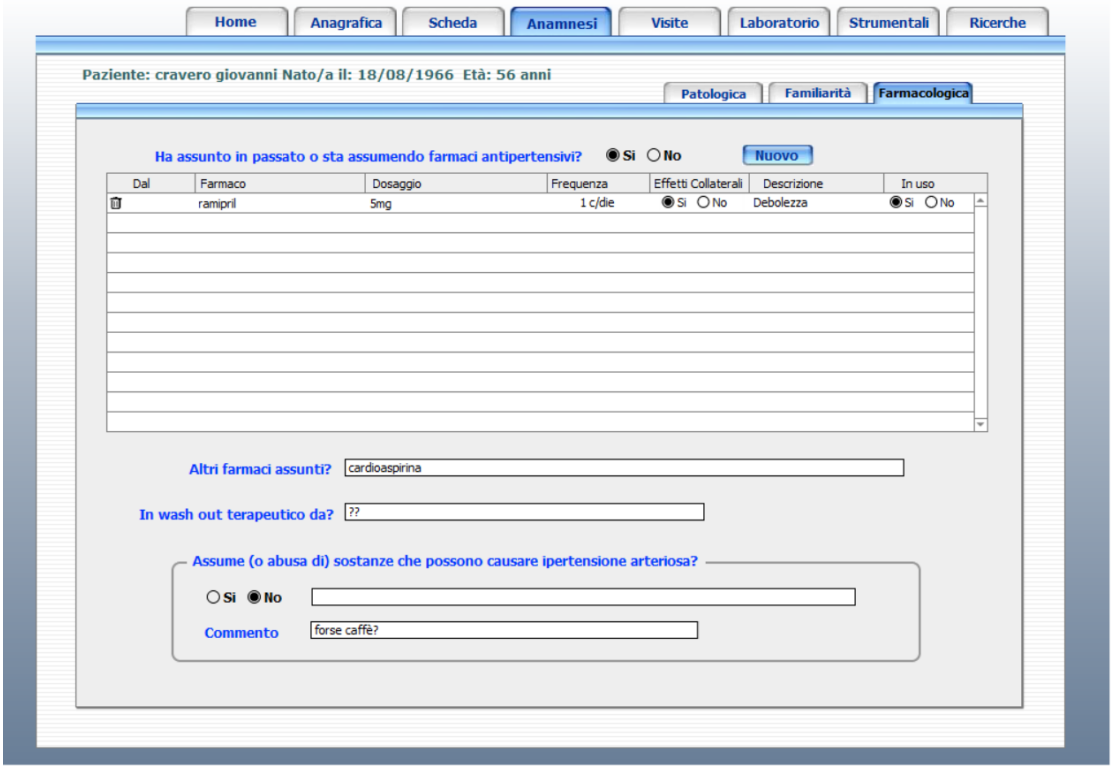


Figura 1.7: Anamnesi - sezione farmacologica

Visite

Tutte le informazioni gestite con le maschere relative alle visite sono registrate nella tabella `Visite`. La dimensione di questa tabella, in termini di numero di proprietà, è considerevole (circa 400). Le figure 1.8, 1.9, 1.10, 1.11 e 1.12 mostrano alcune delle immagini relative a questa sezione del programma. Le immagini presenti nelle sezioni destinate al calcolo del rischio non sono disponibili nell'attuale database.

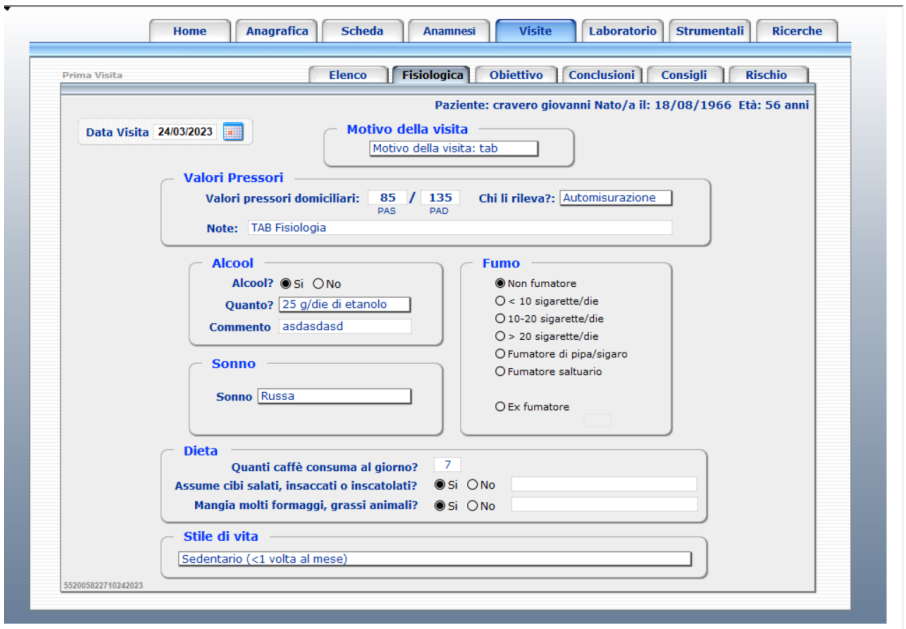


Figura 1.8: Visite - Sezione fisiologica

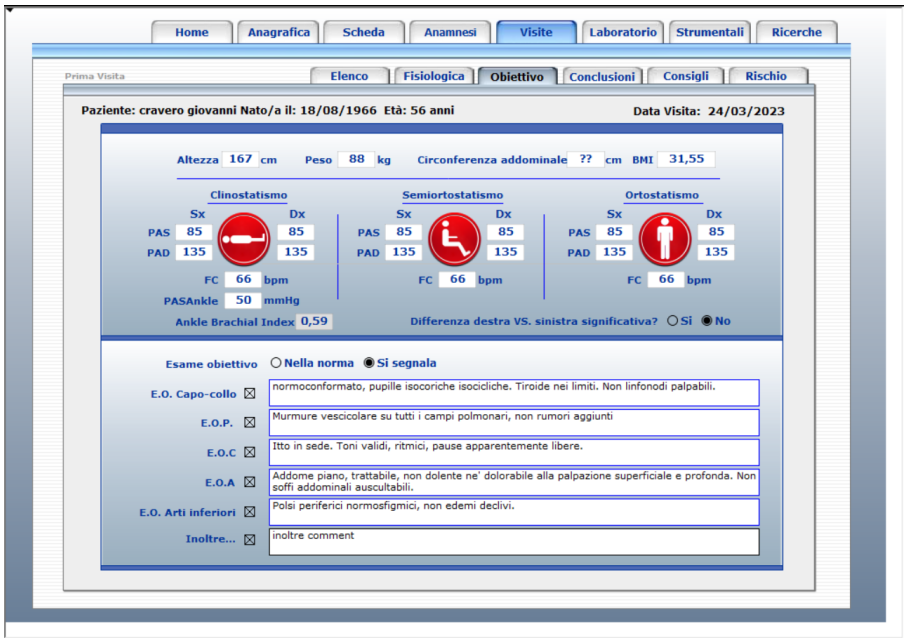


Figura 1.9: Visite - Sezione obiettivo

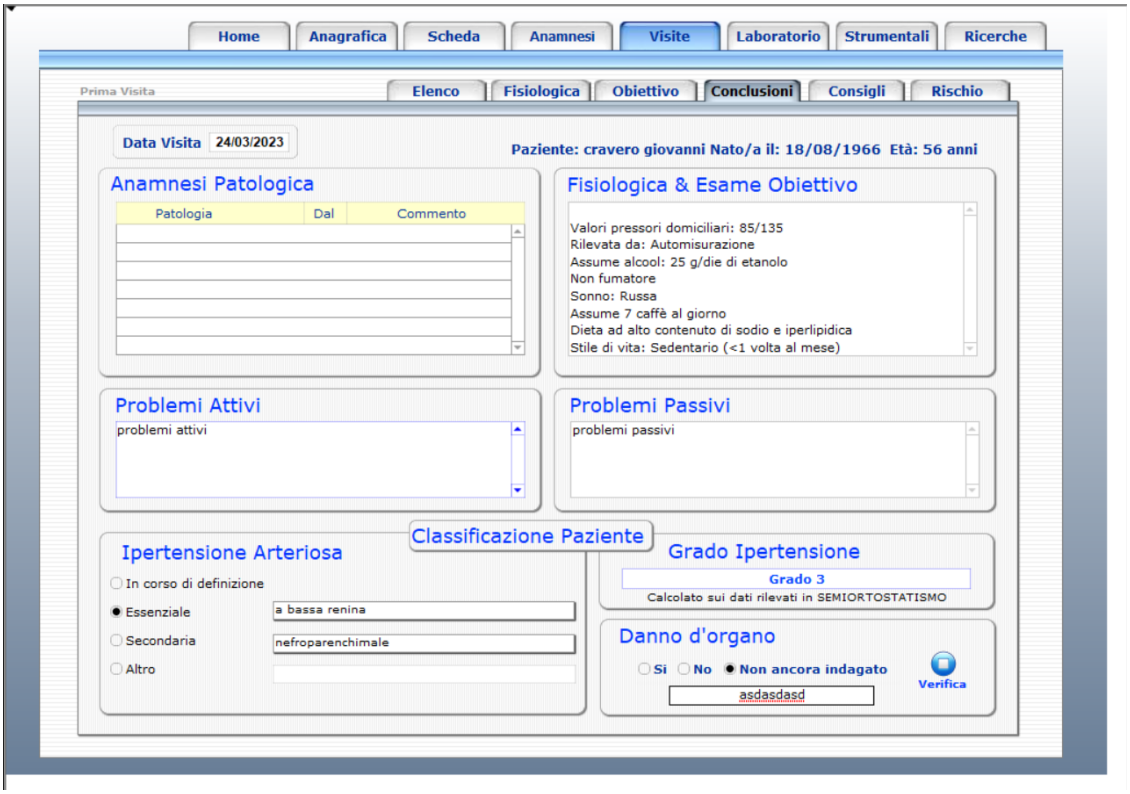


Figura 1.10: Visite - Sezione conclusioni

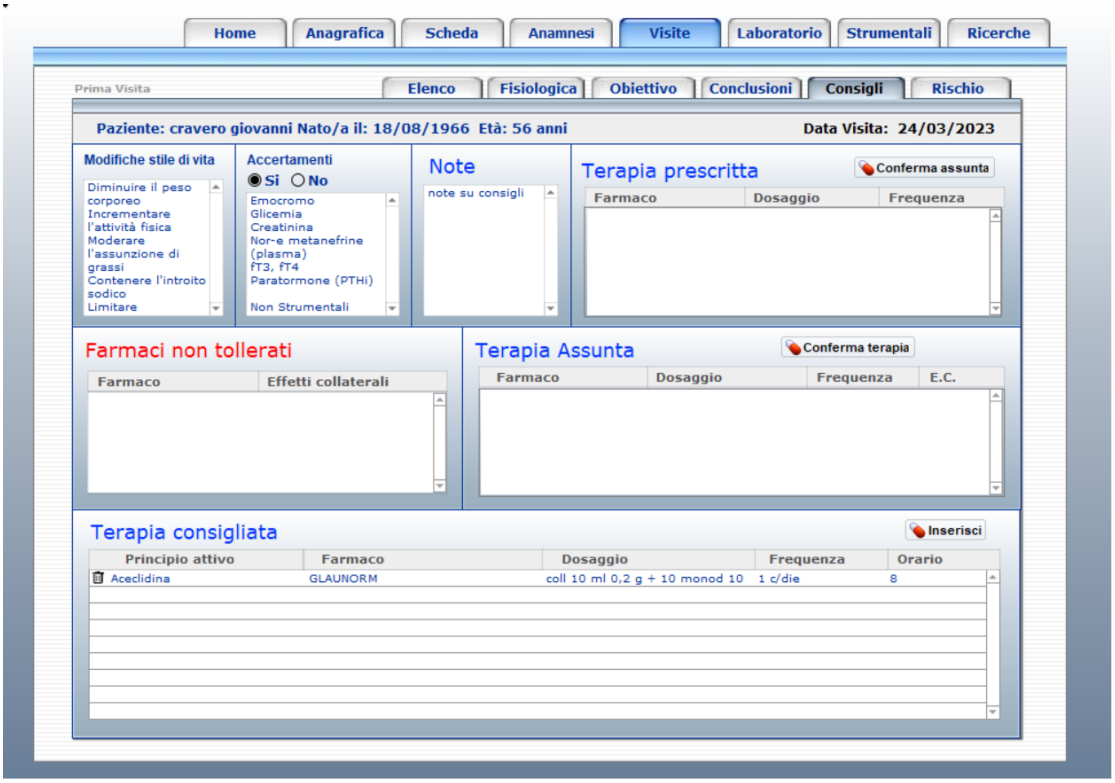
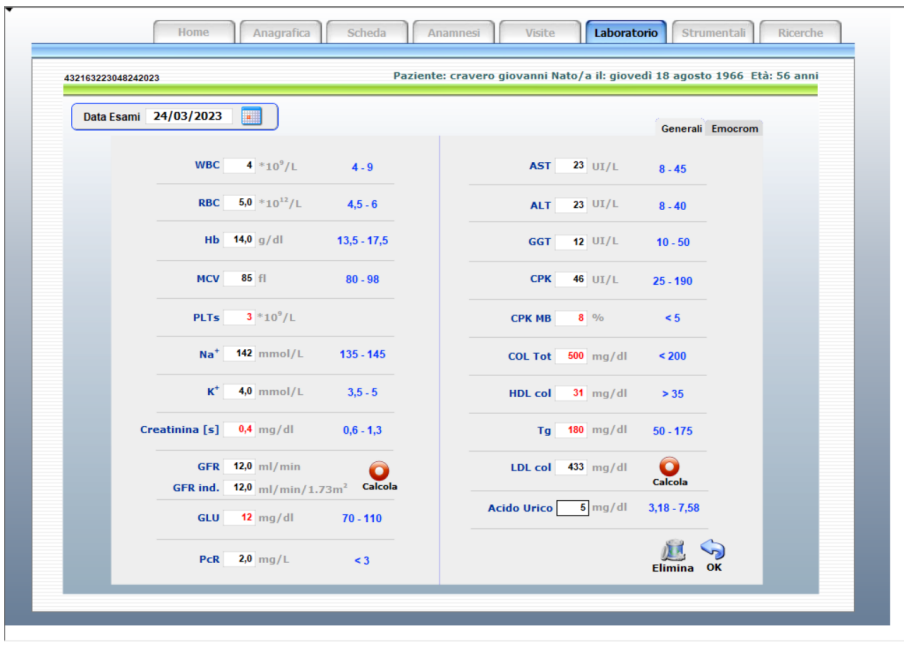


Figura 1.11: Visite - Sezione consigli

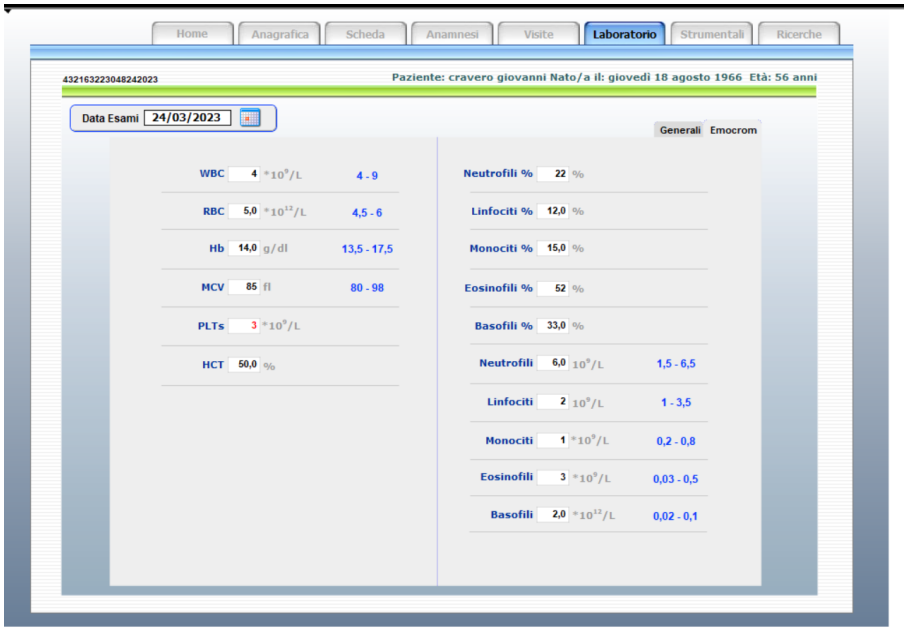
Figura 1.12: Visite - Sezione rischio

Esami di laboratorio

Gli esami di laboratorio risiedono su diverse tabelle. Nella tabella `Ematochimica Generali 1` si trovano i dati delle sezioni `Generali` e `Emocromo` illustrate in figura 1.13 mentre i dati presentati in figura 1.14 sono caricati nella tabella `Ematochimica Generali 2`. Le tabelle `Ematochimica Ormonale` e `Ematochimica Altri esami` contengono, rispettivamente, i dati delle figure 1.15 e 1.16. Infine, in figura 1.17 è illustrata la maschera di inserimento dei dati per la tabella `Esami estemporanei`. Questa tabella è strutturata per l'inserimento di un solo valore per ogni record con riportato nel campo `Test` il tipo di esame da registrare (Test Clonidina, PSA, REUMA TEST, ecc.)



(a) Generali



(b) Emocromo

Figura 1.13: Ematochimica Generali 1

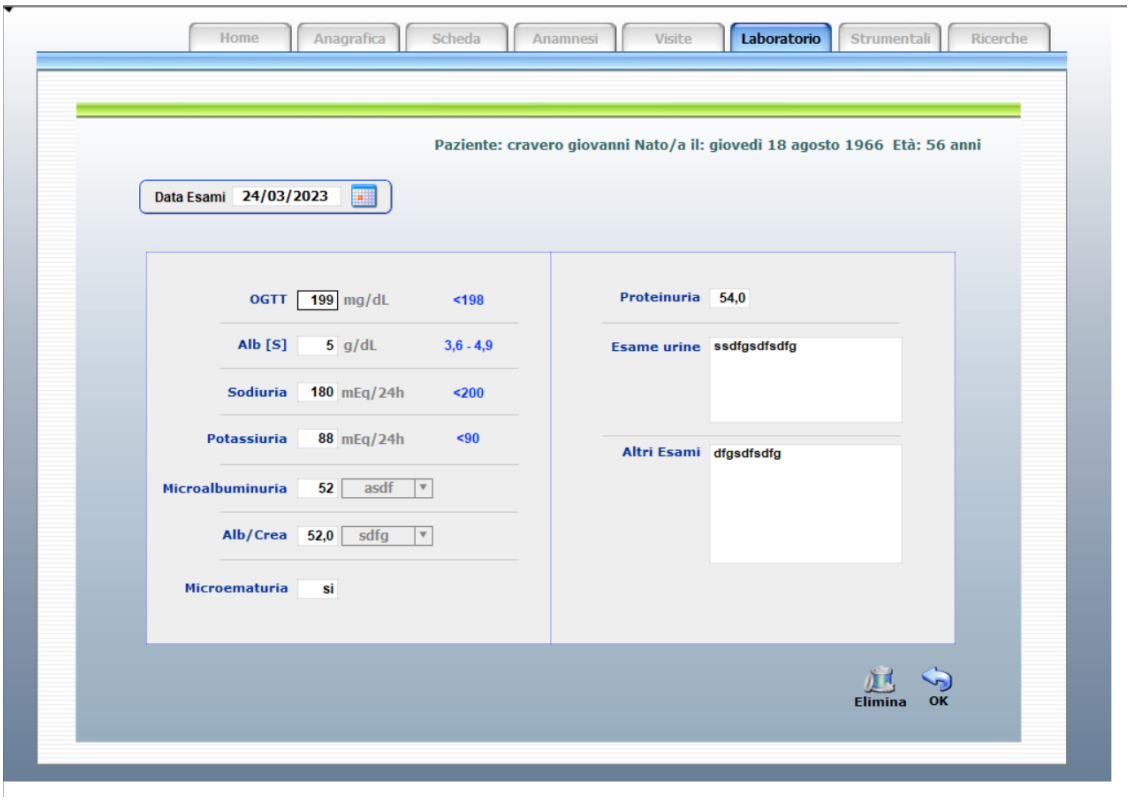


Figura 1.14: Ematochimica Generali 2

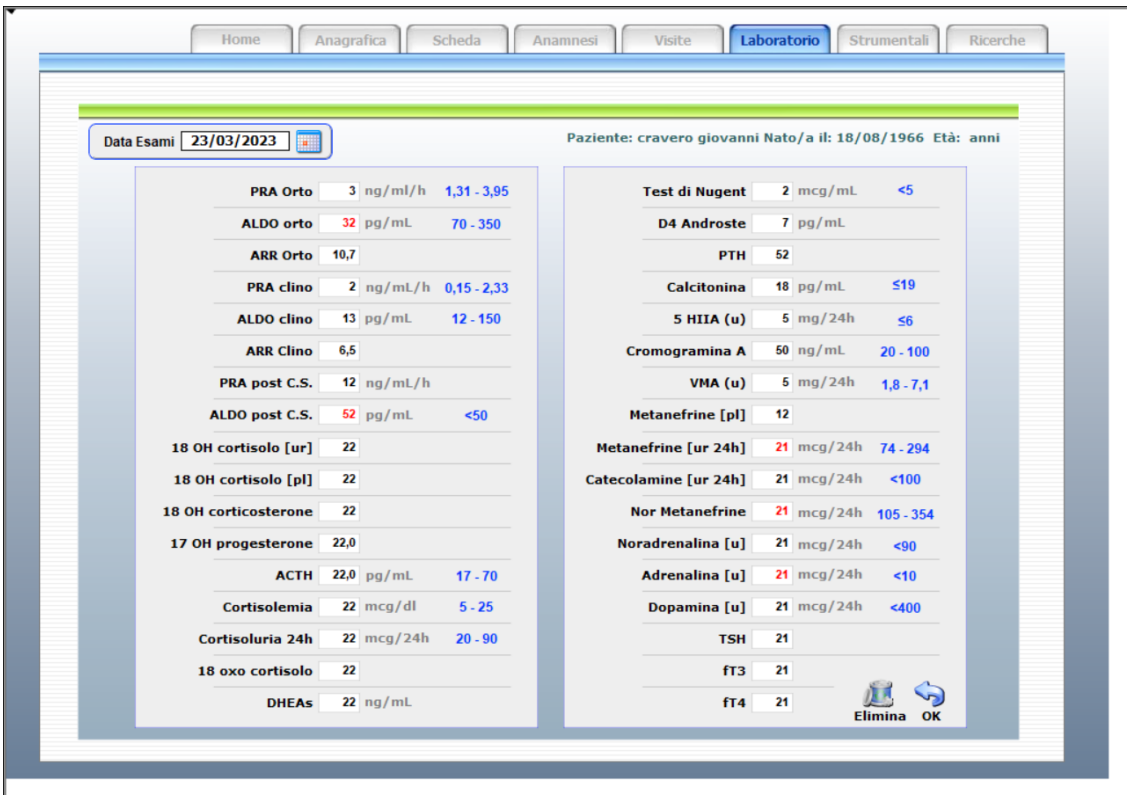


Figura 1.15: Ematochimica Ormonale

Figura 1.16: Ematochimica Altri Esami

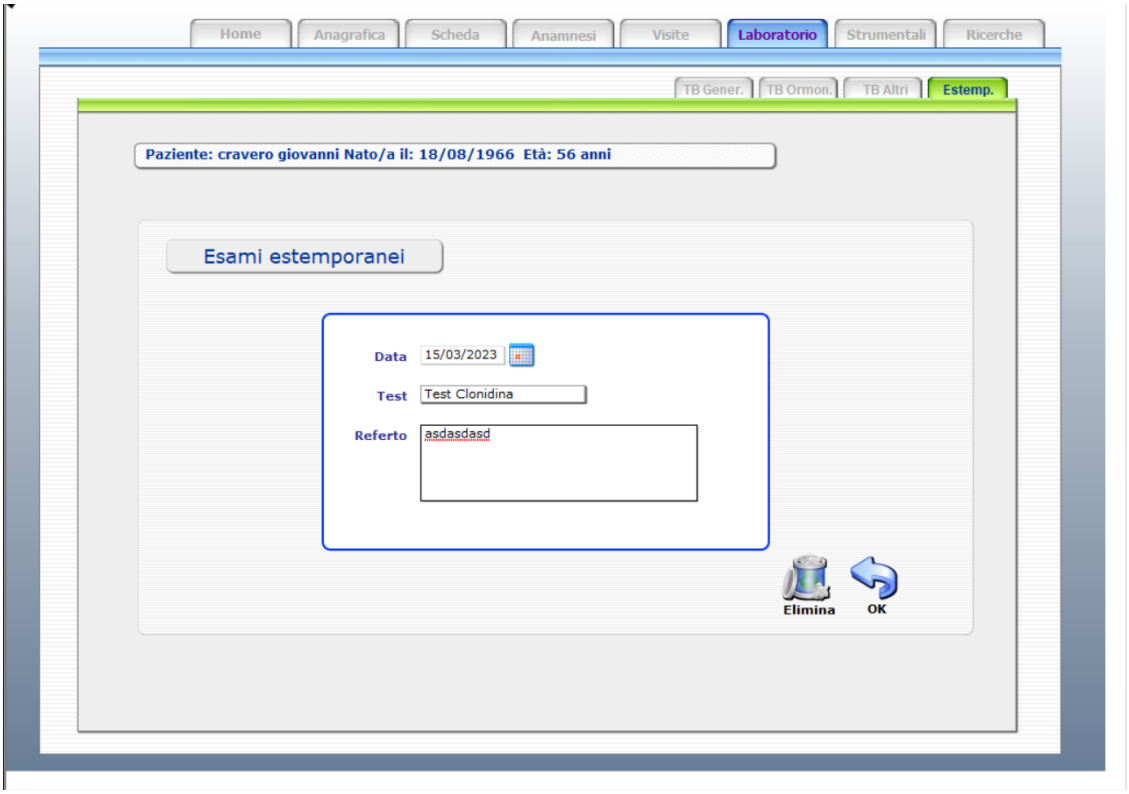
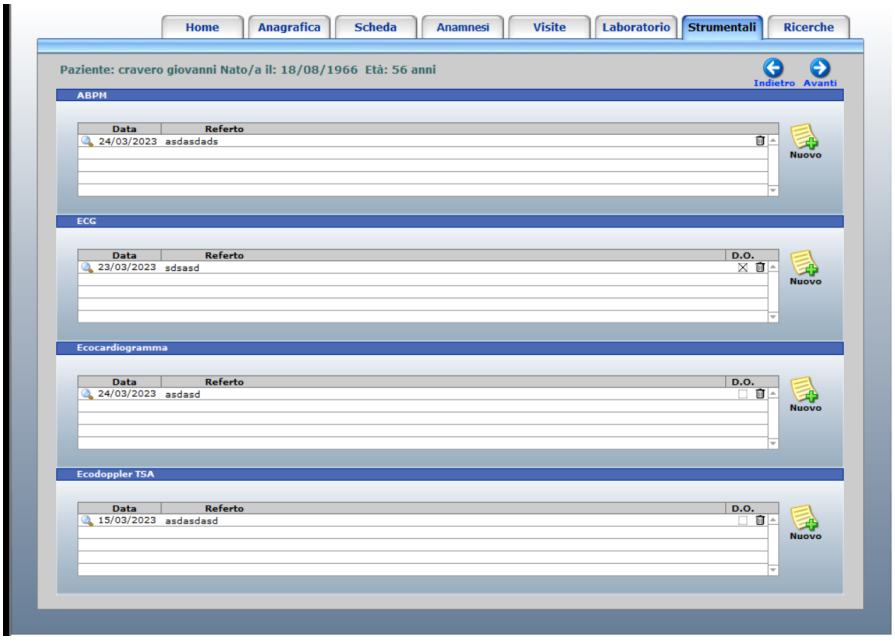


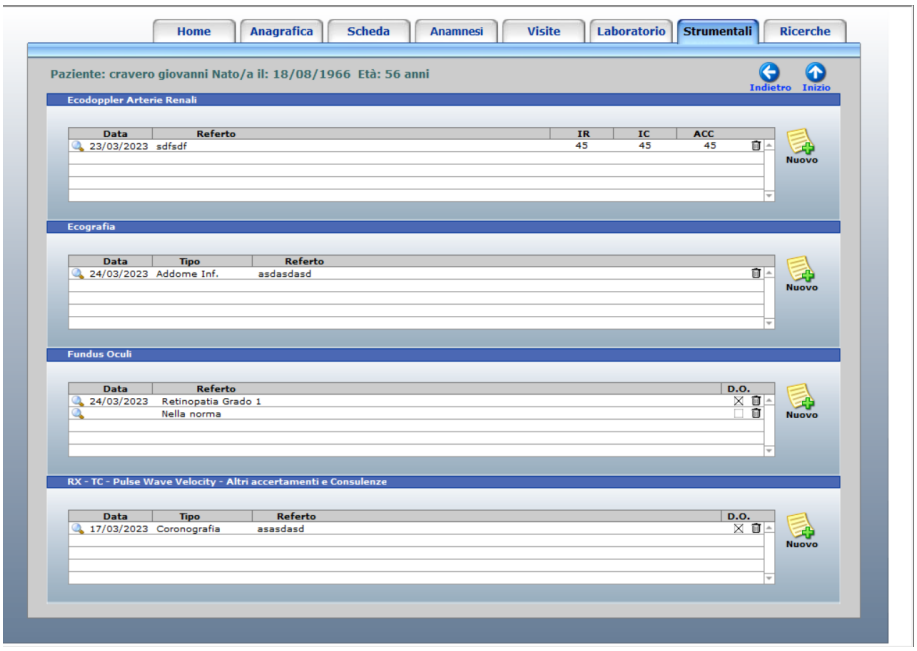
Figura 1.17: Esami di laboratorio estemporanei

Esami strumentali

La tabella `Esami Strumentali` ha una struttura simile a quella degli `Esami estemporanei` con la proprietà `TipoESAME STRUMENTALE` contenente la tipologia dell'esame caricato (ABPM, ECG, Ecocardiogramma, ecc.). Questo tipo di esami è riassunto in figura 1.18.



(a) Esami strumentali - riepilogo pagina 1



(b) Esami strumentali - riepilogo pagina 2

Figura 1.18: Esami strumentali

Ecocardiogramma

Infine, la figura 1.19 presenta le informazioni presenti nella tabella `Ecocardio`.

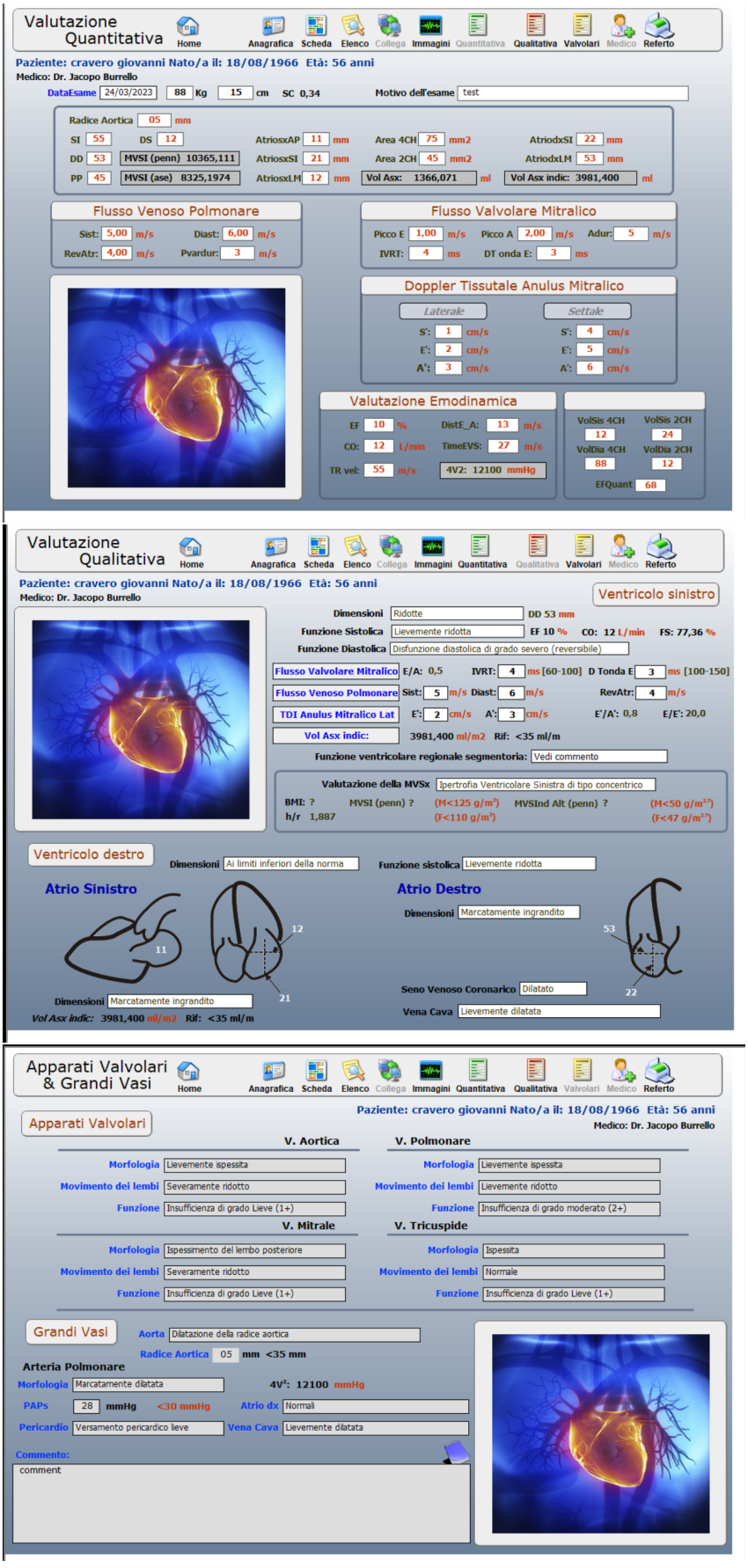


Figura 1.19: Ecocardiogramma

Capitolo 2

Lavoro svolto e scelte progettuali

Il capitolo presenta le attività svolte per progettare la nuova soluzione del backend distribuito per la gestione dell'ambulatorio.

Inizialmente, in base alla analisi effettuata con il reparto dell'ospedale interessato all'adozione del nuovo software, sono stati individuati gli strumenti più idonei per soddisfare le esigenze espresse. Successivamente si è proceduto ad analizzare la base dati esistente per verificare come fosse possibile automatizzare l'importazione delle informazioni nel nuovo database. Data la dimensione ingente delle informazioni da mantenere nella nuova versione del software, questa operazione è fondamentale per poter adottare il nuovo progetto. In seguito, estratti e convertiti i dati e dovendo condividere con gli sviluppatori del frontend i dati dei pazienti dei quali va protetto il diritto alla privacy, si sono trasformati gli stessi oscurando tutte le informazioni sensibili in modo da poter utilizzare un gestore di repository per la collaborazione durante lo sviluppo[1].

2.1 Architettura, linguaggi e framework

2.1.1 Architettura a microservizi

La prima scelta da effettuare riguarda l'architettura e gli strumenti per implementarla. In relazione alla natura complessa, con svariati endpoint da implementare, e l'elasticità richiesta per poter inserire nuove caratteristiche al prodotto in futuro, l'architettura scelta è quella a microservizi che offre, rispetto all'adozione di un unico modulo monolitico, migliori caratteristiche riguardo alla manutenibilità, alla flessibilità, all'affidabilità e alla scalabilità, benché richieda una progettazione molto più articolata.

> A microservice is a small, loosely coupled, distributed service. Microservices let you take an extensive application and decompose it into easy-to-manage components with narrowly defined responsibilities. Microservices help combat the traditional problems of complexity in a large codebase by decomposing it down into small, well-defined pieces.[7]

La decomposizione e il disaccoppiamento dei vari elementi che compongono il progetto renderanno più semplice in futuro estendere il progetto o modificarne una parte senza incorrere nei problemi che generalmente si incontrano in applicazioni monolitiche, come nel caso del software in uso attualmente nel reparto. Inoltre, essendo ogni microservizio slegato dagli altri, diventa possibile testare i componenti separatamente, anche da gruppi distinti di lavoro, e, mantenendo

[1] Tutto il software è stato sviluppato utilizzando *GitHub* come repository

le competenze di ogni singolo servizio confinato ad un compito specifico, gli interventi di manutenzione risultano semplificati. In definitiva, diventa possibile estendere il progetto in tempi più rapidi rispetto ad una architettura monolitica.

La separazione delle competenze tra i vari microservizi permette anche di estendere più facilmente il progetto aggiungendo all'occorrenza i servizi in relazione alle mutate esigenze dell'utenza o di integrare più facilmente questo progetto con altri già esistenti. Ad esempio, la scelta per il sistema di autenticazione prevede la creazione di un servizio che soddisfa le specifiche OAuth2 ma che può essere sostituito dal sistema di autenticazione dell'ospedale.

2.1.2 Framework Spring

Il framework ritenuto migliore per la progettazione di un sistema a microservizi è `Spring` (in particolare `Spring Boot`). La ricchezza dei progetti presenti nel framework unita alla maturità raggiunta ne fanno uno strumento affidabile e potente per sviluppare il backend.

> Any nontrivial application comprises many components, each responsible for its own piece of the overall application functionality, coordinating with the other application elements to get the job done. When the application is run, those components somehow need to be created and introduced to each other.
>
> At its core, Spring offers a ***container***, often referred to as the ***Spring application context***, that creates and manages application components. These components, or ***beans***, are wired together inside the Spring application context to make a complete application, much like bricks, mortar, timber, nails, plumbing, and wiring are bound together to make a house.
>
> The act of wiring beans together is based on a pattern known as ***dependency injection*** (DI). Rather than have components create and maintain the life cycle of other beans that they depend on, a dependency-injected application relies on a separate entity (the container) to create and maintain all components and inject those into the beans that need them. This is done typically through constructor arguments or property accessor methods.[9]

2.1.3 Linguaggio Kotlin

Il linguaggio maggiormente adottato in `Spring` è `Java`. Malgrado esista maggior documentazione con questa combinazione, si è ritenuto opportuno sviluppare il backend con il linguaggio `Kotlin` che, rispetto a `Java`, offre alcuni vantaggi che permettono una maggiore produttività e semplificano la manutenzione[13][14]:

- è più sintetico in quanto ha eliminato la necessità di scrivere molto codice ridondante, come la definizione dei costruttori, dei metodi getter e setter, le implementazioni standard dei metodi per la clonazione e le operazioni di confronto degli oggetti[2];
- ha una maggiore capacità espressiva, riuscendo a esprimere in poche righe di codice ciò che in Java richiede molto codice ripetitivo;
- offre una migliorata gestione delle stringhe;
- gestisce in modo più affidabile e consapevole il valore nullo degli oggetti riducendo i rischi di errori a runtime permettendo, di conseguenza, di scrivere codice più robusto.

`Kotlin` può essere eseguito su svariate virtual machine, compresa la `JVM` (Java Virtual Machine), potendo quindi interagire con i package Java.

[2]Occorre precisare che esistono librerie per il linguaggio Java, ad esempio https://projectlombok.org, che, attraverso il meccanismo delle annotazioni sollevano lo sviluppatore dal dover inserire e, soprattutto, manutenere i metodi citati. In questo progetto si è preferito comunque un linguaggio che offre direttamente queste funzionalità

2.1.4 JSON Schema e validazione dati

Per il formato dei dati, ovviamente anche in relazione a quanto vedremo nel paragrafo 2.1.7 riguardo la scelta del database, si è scelto di adottare il formato JSON che offre una semantica elastica e sintetica (soprattutto rispetto a XML). Per definire i tipi di informazione da registrare ci si è appoggiati alle specifiche `JSON Schema, versione 2020-12` che offre la possibilità di definire il formato dei dati e i vincoli per la validazione degli stessi. In questo modo le proprietà relative alle informazioni che riguardano le visite e gli esami dei pazienti non dipendono da un prestabilito formato legato a delle classi definite all'interno dei package, ma possono variare con la semplice ridefinizione degli schemi che ne rappresentano la forma.

Anche la validazione dei dati non viene effettuata in modo statico direttamente nei programmi ma attraverso uno strumento, `jsonsKema`[3], che utilizza lo standard `JSON Schema` per effettuare i controlli. La scelta di questo software è legata:

- alla frequenza con la quale viene aggiornato;
- all'aderenza ai requisiti richiesti dalle specifiche di `JSON Schema`;
- al linguaggio di programmazione utilizzato (`Kotlin`);
- alla licenza adottata (`MIT`), che permette piena libertà nell'adozione e nell'eventuale ulteriore sviluppo della libreria.

Ulteriori necessità per la validazione dei dati quando questa non fosse possibile utilizzando le specifiche di `JSON Schema`, potrà essere implementata estendendo il progetto.

2.1.5 API REST

Si è scelto di implementare le API con le specifiche REST. In confronto a GraphQL, le altre specifiche prese in considerazione, la complessità è minore e gli svantaggi appaiono non vincolanti. Infatti, benché `GraphQL` offra una flessibilità maggiore e la possibilità di non definire a priori il formato dei dati per la comunicazione tra client e server, permettendo di costruire delle richieste dinamiche, la caratteristica principale del progetto oggetto di questa tesi consiste nel guidare la formazione della user interface in base agli schemi forniti dal server e di poterli modificare a piacere. Risulta più importante fare in modo che il server sia in grado di fornire al client le informazioni necessarie per costruire l'interfaccia piuttosto che permettere al client e al server di essere completamente disaccoppiati. Si ritiene, quindi, che una API semplificata, come nel caso di REST, sia preferibile ad un'infrastruttura complessa almeno nella prima fase di sviluppo del progetto. La rigidità imposta dalla specifica viene superata da una ridotta esposizione di risorse che comprendono però delle specifiche API per restituire l'elenco degli schemi che il client può trattare. Quando ci sarà la necessità di intervenire sulla struttura degli schemi per adeguare l'applicazione a nuove esigenze sarà sufficiente modificare gli schemi presenti nel database o aggiungerne di nuovi.

2.1.6 Servizio di autenticazione

Data la natura molto riservata dei dati trattati, il sistema di autenticazione ricopre un ruolo essenziale. Il progetto è strutturato per potersi affidare ad un servizio esterno, che potrebbe essere il sistema di autenticazione dell'ospedale. La struttura modulare del progetto permette comunque di implementare i filtri di sicurezza in un unico package che può essere adattato al sistema di autenticazione prescelto. Per garantire il funzionamento del prototipo è comunque stato creato un servizio di autenticazione basato su un progetto open source chiamato `Keycloak` che soddisfa tutti i requisiti richiesti per le autenticazioni forti.

[3] Vedi https://github.com/erosb/json-sKema

spostare in implementazione progetto

Nella definizione del prototipo si è scelto di non attivare l'autenticazione a due fattori in quanto non è ancora definito quale sarà l'effettivo servizio di autenticazione (quello definito nel progetto, quello del progetto appoggiato a un server LDAP dell'ospedale, uno esterno dell'ospedale, ecc.) e, inoltre, in questo caso, il servizio dovrebbe essere esposto per l'accesso dall'esterno dell'applicativo direttamente o per tramite del gateway, mentre adesso è configurato per essere accessibile solo dai microservizi interni al sistema oggetto del progetto [4].

2.1.7 Database e storage

MongoDB

Per l'archiviazione dei dati, non volendo vincolare a strutture predefinite le informazioni, scelta che ha di fatto comportato una delle maggiori criticità nell'utilizzo della precedente piattaforma, è stata individuata una soluzione con un DB cosiddetto `NoSQL`. Tra le tipologie di database `NoSQL` si è scelto un DB documentale in quanto ha una struttura molto vicina al tipo di informazioni che vogliamo salvare: ogni paziente è rappresentato da un documento che contiene tutta la sua storia, dati anagrafici, visite, esami, dati clinici e via dicendo.

Una limitazione importante di questo DB, i 16MB per ogni documento, diventano un problema solo se si volesse archiviare nel documento anche parti di grandi dimensioni, come ad esempio le immagini degli esami diagnostici. Nel caso il progetto evolvesse in quella direzione, sarà sufficiente utilizzare l'estensione `GridFS`[5] di `MongoDB` che permette di salvare file di grandi dimensioni spezzandoli in chunk non più grandi di 16MB e gestendo il salvataggio e il recupero atomico di questi spezzoni[6].

Un ulteriore vantaggio è dato dalla struttura interna alle collezioni di MongoDB che, utilizzando il formato BSON[6] per l'archiviazione dei dati permette di essere utilizzato insieme a `JSON Schema` e al validatore `jsonsKema` senza dover effettuare conversioni durante l'elaborazione delle richieste dei client.

Vault

`Spring` mette a disposizione un progetto per gestire i parametri di configurazione dei microservizi in modo dinamico e, soprattutto, non codificato a programma. Questa modalità di gestione è importante soprattutto in un progetto in cui si vogliono mantenere i servizi il più possibile indipendenti gli uni dagli altri e per i quali si desidera poter modificare le configurazioni senza effettuare interventi diretti sul software. Questo progetto, `Spring Cloud Config` è diviso in due parti, il server e i client, e può prelevare da molteplici fonti i parametri di configurazione (file system, repository git, vault server, AWS server, ...). Alcune di queste fonti permettono di gestire in modo dinamico i servizi potendo variare i parametri di configurazione "a caldo" aggiornando automaticamente tutti i servizi senza richiedere il loro riavvio. In una architettura che potrebbe essere affidata a dei servizi in cloud e gestita da orchestratori[7] i vantaggi sono evidenti. Tra le fonti disponibili, data la natura protetta dei dati che vengono trattati, si è preferito adottare `Vault`, uno strumento consolidato di gestione dei segreti, accessibile dagli amministratori dell'hardware sul quale viene installato l'applicativo e semplice da utilizzare nelle macchine di sviluppo tramite

[4] vedi la definizione del client ccinfo

[5] *GridFS* costituisce una specie di file system all'interno del motore del database. Abbina ad ogni documento un insieme di metadati che ne descrivono le caratteristiche, si occupa di archiviare i dati suddividendoli in blocchi di dimensioni opportune che rimangano nei vincoli imposti dal motore del database e li restituisce nuovamente assemblati quando viene richiesta la lettura degli stessi.

[6] MongoDB utilizza BSON, una forma binaria di JSON, che fornisce una maggiore efficienza per il trattamento dei dati oltre ad estenderne la semantica, permettendo di supportare una maggiore quantità di tipi di dato[5].

[7] vedi *Kubernetes*

l'avvio di un container docker. Il servizio viene utilizzato da Spring Cloud prelevando i valori di configurazione in base al contenitore in cui vengono salvati ed è automatizzabile con la sola definizione delle librerie da importare e dei parametri di configurazione.

2.1.8 PostgreSQL

Un ulteriore servizio che è necessario prevedere è il db per il servizio di autenticazione `Keycloak`. Infatti, come detto, il prototipo è stato implementato in modo da fornire un sistema di autenticazione per poter gestire l'accesso degli utenti e l'assegnazione dei ruoli. Il servizio richiede un db SQL per archiviare gli utenti e gli oggetti utili alla gestione del protocollo di autenticazione OAuth2. La scelta di `PostgreSQL` è dettata da due principi: la natura open source che non vincola il progetto in base a futuri sviluppi nel caso si volesse mantenere questo sistema di autenticazione e alla aderenza che questo motore garantisce rispetto allo standard SQL.

2.2 Importazione e inizializzazione dei dati

I dati sono stati esportati tramite il tool previsto dal programma `HyperMacondo` nel formato `FileMaker Pro 12`. Il tool estrae i dati dal database dell'applicazione e li salva nel formato con estensione `*.fmp12` nella sottocartella `Export - Import` della cartella principale in cui risiede l'attuale implementazione. L'esportazione è avvenuta sulla base dati estratta a novembre 2022 e tutti i test per ora effettuati si riferiscono a quella elaborazione (24.334 pazienti). Quando verrà installata la versione definitiva, sostituendo l'applicazione utilizzata attualmente, dovrà essere effettuato un nuovo processo di verifica delle informazioni convertite.

Il tool non esporta la lista dei medici presenti in procedura. Di conseguenza, si è proceduto manualmente al recupero dell'elenco dalla tabella anagrafica dei medici utilizzando il vecchio programma. Nel nuovo database è stato inserito come riferimento il codice che identifica il medico su `HyperMacondo`, recuperando la chiave presente sulla maschera di aggiornamento dell'anagrafica (visibile in fig. 2.1 sotto l'immagine del corpo umano)[8]. Il codice è stato mantenuto come riferimento sul nuovo database per permettere di effettuare i controlli in fase di conversione ma potrà essere eliminato in futuro.

Successivamente all'esportazione, i dati sono stati convertiti nel formato JSON utilizzando il package `fmptools versione 0.2.0` abbinato al package `yajl-devel versione 2.1.0-18.fc36` su un sistema Linux Fedora release 36. I dati così trasformati presentano un unico documento per ogni tabella che riporta tre proprietà: il nome della tabella stessa, una lista con il nome delle colonne della tabella e una lista di documenti contenenti i dati effettivi. Tutti i dati sono di tipo stringa per cui è stato necessario inferire il tipo di dato effettivo. L'analisi dei dati contenuti nelle tabelle e la loro conversione è effettuata dal modulo `data_obfuscation` descritto nel seguente paragrafo.

2.2.1 Mascheramento dati sensibili e generazione schemi

Come indicato in premessa al capitolo, la necessità di condividere i dati con gli sviluppatori del frontend ha richiesto un'operazione di modifica sui dati per garantire la privacy dei pazienti. L'analisi dei dati convertiti dal vecchio formato ha evidenziato quali informazioni possono identificare i pazienti. Il package `it.polito.links.ccinfo.data_obfuscation` è stato sviluppato utilizzando il framework `Spring`, progetto `Spring Batch`, per elaborare i dati estratti da `HyperMacondo` e mascherare i dati individuati nella fase di analisi. Per ogni singola tabella viene eseguito su tutti i dati sensibili l'algoritmo presentato nel listato 2.1 che sostituisce i caratteri del testo originale in modo casuale, conservando unicamente la lunghezza del testo originario e gli spazi tra i caratteri.

[8] L'immagine fa riferimento ad un medico fittizio creato in fase di test.

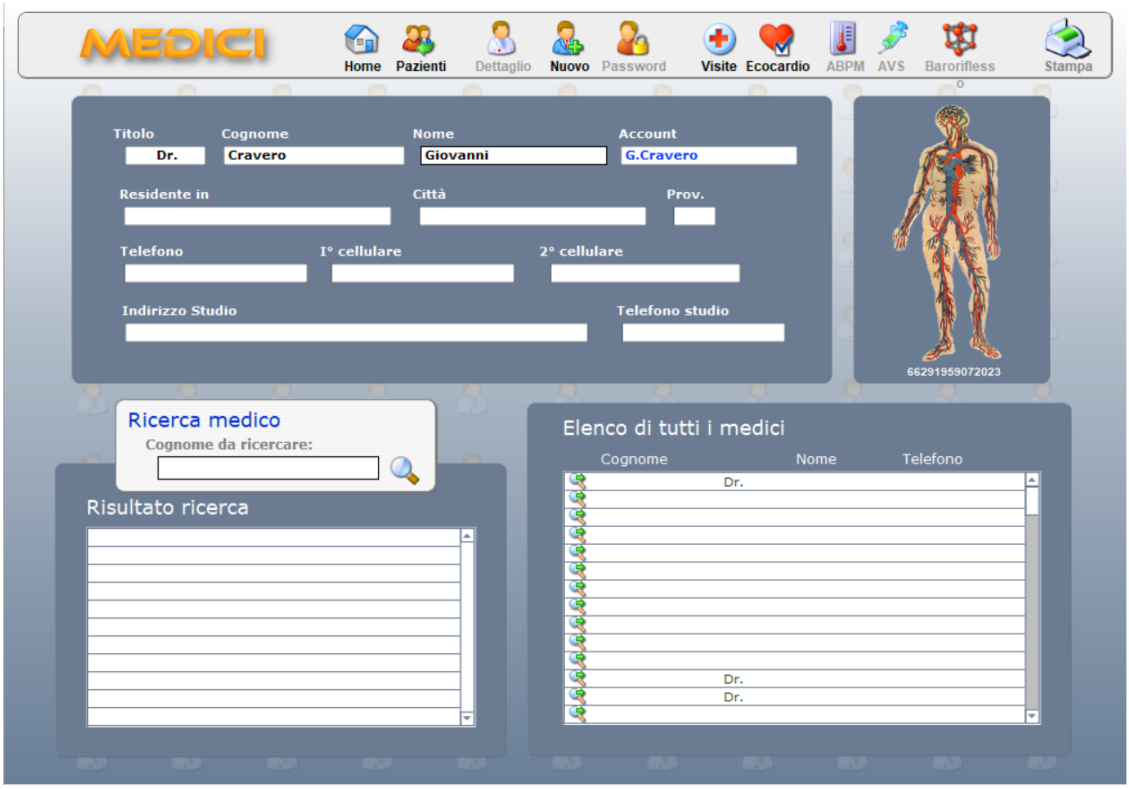


Figura 2.1: Maschera manutenzione anagrafica medici

Listato 2.1: Funzione mascheramento dati

```
fun string_obfuscation(inputstring: String): String {

        val alphabet: List<Char> = ('a'..'z') + ('A'..'Z') + ('0'..'9')
        val outputstring = inputstring
                .toCharArray()
                .map {if (it.equals('␣')) it else alphabet.random() }
                .joinToString("")

        return outputstring
}
```

Durante questa fase sono stati anche analizzati i vincoli referenziali presenti nelle tabelle, con particolare riferimento all'anagrafica dei medici, ed è stato verificato che il nome della proprietà che identifica il medico non è sempre la stessa.

Nel processo batch che esegue il mascheramento dei dati, vengono inizialmente create due variabili dallo step illustrato nel listato 2.2.

Listato 2.2: Inizializzazione variabili condivise

```
        @Configuration
        class InitTasklet : Tasklet {
                override fun execute(contribution: StepContribution, chunkContext: ChunkContext):
                    RepeatStatus? {
                    val jobExecution: JobExecution =
                         chunkContext.stepContext.stepExecution.jobExecution
                    val schema = Schema(
                    type = "object",
                    properties = mutableMapOf<String, Schema>().apply {
                            this["anagrafica"] = Schema(
                                    schema = null,
                                    id = null,
                                    ref = "#/\$defs/anagrafica"
                            )
                            this["abpm"] = Schema(
```

```
14                                      schema = null,
15                                      id = null,
16                                      ref = "#/\$defs/abpm"
17                                  )
18                                  this["ecocardio"] = Schema(
19                                      schema = null,
20                                      id = null,
21                                      ref = "#/\$defs/ecocardio"
22                                  )
23                                  this["ematochimicaAltri"] = Schema(
24                                      schema = null,
25                                      id = null,
26                                      ref = "#/\$defs/ematochimicaaltri"
27                                  )
28                                  this["ematochimicaGenerali1"] = Schema(
29                                      schema = null,
30                                      id = null,
31                                      ref = "#/\$defs/ematochimicagenerali1"
32                                  )
33                                  this["ematochimicaGenerali2"] = Schema(
34                                      schema = null,
35                                      id = null,
36                                      ref = "#/\$defs/ematochimicagenerali2"
37                                  )
38                                  this["ematochimicaOrmonale"] = Schema(
39                                      schema = null,
40                                      id = null,
41                                      ref = "#/\$defs/ematochimicaormonale"
42                                  )
43                                  this["esamiEstemporanei"] = Schema(
44                                      schema = null,
45                                      id = null,
46                                      ref = "#/\$defs/esamiestemporanei"
47                                  )
48                                  this["esamiStrumentali"] = Schema(
49                                      schema = null,
50                                      id = null,
51                                      ref = "#/\$defs/esamistrumentali"
52                                  )
53                                  this["familiarita"] = Schema(
54                                      schema = null,
55                                      id = null,
56                                      ref = "#/\$defs/familiarita"
57                                  )
58                                  this["patologie"] = Schema(
59                                      schema = null,
60                                      id = null,
61                                      ref = "#/\$defs/patologie"
62                                  )
63                                  this["terapia"] = Schema(
64                                      schema = null,
65                                      id = null,
66                                      ref = "#/\$defs/terapia"
67                                  )
68                                  this["visite"] = Schema(
69                                      schema = null,
70                                      id = null,
71                                      ref = "#/\$defs/visite"
72                                  )
73                                  this["agenda"] = Schema(
74                                      schema = null,
75                                      id = null,
76                                      ref = "#/\$defs/agenda"
77                                  )
78                              },
79                                  defs = mutableMapOf()
80                              )
81                              jobExecution.executionContext.put("SchemasDef", schema)
82                              val physiciansSet: MutableMap<String, MutableSet<String>> = mutableMapOf()
83                              jobExecution.executionContext.put("PhysiciansSet", physiciansSet)
84                              return RepeatStatus.FINISHED
85                      }
```

```
86          }
```

La prima, `schema`, contiene gli schemi che vengono generati analizzando il contenuto dei dati esportati dal vecchio sistema. Lo schema comprende la definizione di tutti i tipi di informazione dei quali andrà effettuata la validazione e costituirà l'ossatura del sistema di controllo e della formazione delle interfacce utente per la manipolazione dei dati. Lo schema di validazione contiene tutti gli schemi delle tabelle utilizzate. Per ogni tabella è definito un riferimento interno allo schema stesso (le proprietà `ref`) che permettono di utilizzare lo stesso documento per la validazione di qualsiasi tipo di tabella. La seconda, `pysiciansSet`, è una mappa in cui vengono registrate le chiavi anagrafiche dei medici abbinate alle collezioni nelle quali vengono utilizzate. L'insieme viene utilizzato per verificare quali sono i medici effettivamente presenti come riferimento nei vecchi archivi per guidare l'importazione dell'anagrafica che, come detto, deve essere svolta manualmente.

Per ogni tabella esiste un `Job` dedicato che legge il file `JSON` convertito da `FMP12`, genera il `JSON Schema` per quella tabella e salva i dati in un file `JSON` con lo stesso nome del file di input con i valori delle proprietà eventualmente mascherate. Se lo schema elaborato contiene un riferimento alla chiave di un medico viene salvata per permettere di comprendere quali medici dovranno essere riportati nella nuova anagrafica. Il formato delle tabelle importate è stato mantenuto generico per mantenere più flessibile la sua elaborazione. Come si vede nei listati 2.3 e 2.4, le tabelle esportate nel formato `JSON` da `Hypermacondo` hanno tutte la stessa struttura, un solo documento con tre proprietà: il nome della tabella, la lista dei nomi delle proprietà e la lista dei valori che, come detto, vengono caricati dal `Job` come documenti `Bson`.

Listato 2.3: Estratto tabella ABPM convertita da Filemaker a JSON

```
1  [
2    {
3          "name": "ABPM",
4          "columns": [
5                  {
6                          "name": "ID_ABPM"
7                  },
8                  {
9                          "name": "IDPaziente"
10                 },
11                 {
12                         "name": "ID_Medico"
13                 },
14                 {
15                         "name": "G_IDPaziente"
16                 },
17                 {
18                         "name": "Eta"
19                 },
20                 {
21                         "name": "-----------------"
22                 },
23                 { ... },
24         ],
25         "values": [
26                 {
27                         "ID_ABPM": "94 G 21533 D 892005",
28                         "IDPaziente": "17502",
29                         "Eta": "27",
30                         "ABPM Data": "09/08/2005",
31                         "ABPM Data Odierna": "09/08/2005",
32                         "ABPM Diabete": "No",
33                         "ABPM Nefropatia": "No",
34                         "ABPM 24h PAS": "131",
```

```
                    "ABPM 24h PAD": "77",
                    "ABPM 24h Media": "94",
                    "ABPM 24h FC": "78",
                    "ABPM 24h PA DS": "9,57",
                    "ABPM DayTime Pas": "134",
                    "ABPM DayTime Pad": "80",
                    "ABPM DayTime Media": "96",
                    "ABPM DayTime FC": "79",
                    "ABPM DayTime PA DS": "9,22",
                    "ABPM NightTime Pas": "125",
                    "ABPM NightTime Pad": "70",
                    "ABPM NightTime Media": "88",
                    "ABPM NightTime FC": "74",
                    "ABPM NightTime PA DS": "8,7",
                    "ABPM PP 24h": "54",
                    "ABPM Dipping": "6,7164179104477612",
                    "ABPM Valore riferimento Pres Diurna": "(<135/85)",
                    "ABPM Valore riferimento Pres Notturna": "(<120/70)",
                    "ABPM Commento": "nn",
                    "ABPM MotivoEsame": "valutazione in WO"
                },
                { ... }
        ]
    }
]
```

Listato 2.4: Tabella di importazione di esempio: elaborazione dati ABPM

```
@JsonInclude(JsonInclude.Include.NON_NULL)
data class Abpm(
        @JsonProperty("name") val name: String,
        @JsonProperty("columns") val columns: List<AbpmColumnName>,
        @JsonProperty("values") val values: List<Document>
)

@JsonInclude(JsonInclude.Include.NON_NULL)
data class AbpmColumnName(
        @JsonProperty("name") val name: String
)
```

Il listato 2.5 presenta l'esempio di job utilizzato per convertire la tabella `ABPM` e la creazione dello schema corrispondente.

Listato 2.5: Job di esempio: elaborazione tabella ABPM

```
// Imports and package definition omitted
@Configuration
class AbpmJob(
val jobRepository: JobRepository,
val transactionManager: PlatformTransactionManager,
val appConfiguration: AppConfiguration
) {

        @Bean
        fun abpmStep(): Step =
        StepBuilder("abpmStep", jobRepository)
        .chunk<Abpm, Abpm>(1, transactionManager)
        .reader(abpmJsonReader())
        .processor(abpmItemProcessor())
        .writer(abpmJsonWriter())
        .build()

        @Bean
        fun abpmJsonReader(): JsonItemReader<Abpm> =
        JsonItemReaderBuilder<Abpm>()
```

```
21              .jsonObjectReader(JacksonJsonObjectReader<Abpm>(Abpm::class.java))
22              .resource(appConfiguration.resourcesConfig().abpmInp)
23              .name("abpmJsonReader")
24              .build()
25
26              @Bean
27              fun abpmJsonWriter(): JsonFileItemWriter<Abpm> =
28              JsonFileItemWriterBuilder<Abpm>()
29              .jsonObjectMarshaller(JacksonJsonObjectMarshaller<Abpm>())
30              .resource(FileSystemResource(appConfiguration.resourcesConfig().abpmOut))
31              .name("abpmJsonWriter")
32              .build()
33
34      }
35
36      class abpmItemProcessor: ItemProcessor<Abpm, Abpm>, StepExecutionListener {
37
38              lateinit var schemas: Schema
39              lateinit var physiciansSet: MutableMap<String, MutableSet<String>>
40              override fun beforeStep(stepExecution: StepExecution) {
41                      super.beforeStep(stepExecution)
42                      this.schemas =
                            stepExecution.jobExecution.executionContext.get("SchemasDef") as
                            Schema
43                      this.physiciansSet =
44                      stepExecution.jobExecution.executionContext.get("PhysiciansSet") as
                            MutableMap<String, MutableSet<String>>
45              }
46
47              override fun process(item: Abpm): Abpm {
48                      var schema = Schema(
49                      id = "/abpm",
50                      type = "object",
51                      title = "ABPM",
52                      description = "ABPM␣-␣Monitoraggio␣pressione",
53                      comment = "type:ESAME;cluster:esami",
54                      properties = mutableMapOf())
55                      item.columns
56                      .filter { !it.name.startsWith('-') }
57                      .forEach {
58                              schema.properties?.put(
59                              it.name,
60                              Schema(schema = null, id = null, type = null)
61                              )
62                      }
63                      item.values.listIterator()
64                      .forEach {
65                              it.entries.forEach {
66                                      schema.setPropertyTypeAndFormat(it.key, it.value as? String)
67                              }
68                              if (it.containsKey("ABPM␣MotivoEsame")) {
69                                      it["ABPM␣MotivoEsame"] =
70                                      string_obfuscation(it.getString("ABPM␣MotivoEsame"))
71                              }
72                              if (it.containsKey("Medico␣refertante")) {
73                                      it["Medico␣refertante"] =
74                                      string_obfuscation(it.getString("Medico␣refertante"))
75                              }
76                              if (it.containsKey("g_Medico␣refertante")) {
77                                      it["g_Medico␣refertante"] =
78                                      string_obfuscation(it.getString("g_Medico␣refertante"))
79                              }
80                              if (it.containsKey("ID_Medico")) {
81                                      val id_medico = it.getString("ID_Medico")
82                                      if (this.physiciansSet.containsKey(id_medico)) {
83                                              this.physiciansSet.get(id_medico)!!.add("ABPM")
84                                      } else {
85                                              this.physiciansSet.put(id_medico,
                                                    mutableSetOf("ABPM"))
86                                      }
87                              }
88                      }
```

```
89                      schemas.defs?.put("abpm", schema)
90                      return item
91                  }
92          }
```

Come si evince dal listato, il job legge una risorsa dal file `JSON` e la restituisce con lo stesso formato ma con i dati oscurati e dopo aver estratto l'eventuale codice del medico di riferimento. Il formato è stato mantenuto identico in quanto il batch di inizializzazione (vedi par. 3.3.2) deve essere in grado di elaborare sia i dati offuscati, output di questa elaborazione, sia i dati effettivi, input di questa elaborazione, quando verrà eseguita la messa in produzione di questo progetto. Il `Job` processa un solo documento con, al suo interno, la lista dei nomi delle proprietà e la lista dei valori, per cui l'elaborazione dei chunk non può essere ottimizzata in parallelo. Benché l'intero processo di conversione sia leggermente lento, non si è ritenuto necessario provvedere alla sua ottimizzazione in quanto viene svolto una tantum e non incide sulle performance generali dell'applicazione. Viceversa, per la fase di inizializzazione del sistema che ha il compito di importare i dati nel database nella fase di installazione, il processo è stato ottimizzato, come si vedrà nel paragrafo 3.3.2.

Nella fase di elaborazione del documento caricato vengono svolte due iterazioni. La prima sui nomi delle proprietà dalle quali vengono eliminate quelle non significative prima del loro inserimento nello schema corrispondente. Gli schemi di ogni singola tabella sono generati con una parte fissa in cui vengono definite le proprietà descrittive dello schema e, sfruttando il campo `$comment` dello standard di `JSON Schema`, viene impostato in quale proprietà dei documenti dei pazienti i dati recuperati devono essere inseriti. Infatti, i dati dei pazienti sono inseriti in un unico documento che rappresenta la cartella clinica del paziente, ma divisi in quattro cluster, ognuno dei quali può contenere informazioni di tipo diverso:

- `personalData` in cui vengono salvati i dati anagrafici del paziente (schema anagrafica);
- `esami` con le informazioni degli schemi abpm, ecocardio, ematochimicaaltri, ematochimicagenerali1, ematochimicagenerali2, ematochimicaormonale, esamiestemporanei e esamiestemporanei;
- `dati` che contiene i dati degli schemi familiarita, patologie e terapia;
- `visite` con i dati che fanno riferimento allo schema omonimo.

Ad esempio, nel listato in esame, il cluster impostato è `esami`. Vengono scartate le proprietà segnaposto (contenenti solo trattini) e analizzati i contenuti delle proprietà delle tabelle. Inizialmente tutte le proprietà presenti nel campo "name" dello schema originale vengono impostate con il tipo `null`. Se la proprietà non compare nei contenuti della tabella vuol dire che non è mai stata utilizzata nel precedente programma e, previa una valutazione effettuata con il committente, viene scartata nella definizione dei nuovi schemi[9]. Per inferire il tipo di dato, viene eseguito l'algoritmo del listato 2.6 sull'informazione presente nella proprietà.

Listato 2.6: Inferenza del tipo dato

```
1         fun data_type_and_format(data: String): Pair<String, String?> {
2
3                 var datatype = when {
4                         data.matches("^[0-9]{1,2}/[0-9]{1,2}/[0-9]{2,4}$".toRegex()) ->
                              Pair("string", "date")
5                         data.matches("^-?[0-9]+$".toRegex()) -> Pair("integer", null)
6                         data.matches("^-?[0-9]*[.,][0-9]+$".toRegex()) -> Pair("number", null)
7                         else -> Pair("string", null)
8                 }
9                 if (datatype.first == "integer" && data.toBigInteger() >
                      BigInteger.valueOf(Long.MAX_VALUE)) {
10                        datatype = Pair("string", null)
```

[9] In un ottica di semplificazione e pulizia si è deciso di mantenere solo le informazioni utili alle quali verranno aggiunte quelle necessarie per far progredire la CCINFO.

```
11                }
12                return datatype
13
14        }
```

Lo schema viene aggiornato e genera i `JSON Schema` da utilizzare nelle collezioni di `MongoDB` con la definizione delle proprietà dedotta dal contenuto dei campi presenti nei record. Durante questa fase, negli schemi, vengono anche registrati dei commenti per evidenziare le anomalie riscontrate, in particolare quelle legate alla presenza nella stessa proprietà di dati appartenenti a domini diversi (alfanumerici e numerici, date e campi non data, ecc.). Se la proprietà viene incontrata per la prima volta (il tipo di dato è ancora NA) viene impostato il valore restituito dall'algoritmo. Se, invece, la proprietà è già stata impostata allora viene controllato che il dominio sia compatibile (vedi listato 2.7) ed eventualmente viene generato un commento di errore per segnalare l'anomalia. L'output prodotto sarà utilizzato per determinare con il committente la versione definitiva degli schemi da adottare ed i controlli da inserire per la validazione dei documenti.

Listato 2.7: Controllo dominio tipo di dato

```
1  fun check_domain(a: String, b: String): Pair<Boolean, String> {
2
3         return when {
4                 a.equals(b) -> Pair(true, a)
5                 a.equals("integer") && b.equals("number") -> Pair(true, "number")
6                 a.equals("number") && b.equals("integer") -> Pair(true, "number")
7                 else -> Pair(false, "string") // Include one type equals to string
8         }
9
10 }
```

La funzione che esegue questi algoritmi è visibile nella classe `Schema` visibile nel listato 2.8.

Listato 2.8: Definizione JSON Schema

```
1         package it.polito.links.ccinfo.data_obfuscation.model
2
3         import com.fasterxml.jackson.annotation.JsonInclude
4         import com.fasterxml.jackson.annotation.JsonProperty
5         import it.polito.links.ccinfo.data_obfuscation.check_domain
6         import it.polito.links.ccinfo.data_obfuscation.data_type_and_format
7         import java.io.Serializable
8
9         @JsonInclude(JsonInclude.Include.NON_NULL)
10        data class Schema(
11        // Common to all types
12        @get:JsonProperty("\$schema", index = 0) val schema: String? =
              "https://json-schema.org/draft/2020-12/schema",
13        @get:JsonProperty("\$id", index = 1) val id: String? =
              "http://links.polito.it/schemas/ccinfo",
14        @get:JsonProperty(index = 10) var type: String? = null, // object, string, number,
              integer, array, boolean, null
15        @get:JsonProperty("enum") var enumProp: MutableList<Any>? = null, // valid also without
              type
16        var title: String? = null,
17        var description: String? = null,
18        var default: Any? = null,
19        var examples: MutableList<Any>? = null,
20        var deprecated: Boolean? = null,
21        var readOnly: Boolean? = null,
22        var writeOnly: Boolean? = null,
23        @get:JsonProperty("\$comment") var comment: String? = null,
24        @get:JsonProperty("const") var constValue: Any? = null,
25        @get:JsonProperty("\$defs", index = 999) var defs: MutableMap<String, Schema>? = null,
26        @get:JsonProperty("\$ref", index = 20) var ref: String? = null,
27        var definitions: MutableMap<String, Schema>? = null,
28        @get:JsonProperty("\$anchor") var anchor: String? = null,
29
30        // Type object
31        @get:JsonProperty(index = 30) var properties: MutableMap<String, Schema>? = null,
32        var minProperties: Int? = null,
33        var maxProperties: Int? = null,
```

```
34          var additionalProperty: Boolean? = null,
35          @get:JsonProperty(index = 25) var required: MutableList<String>? = null,
36          var unevaluatedProperties: Boolean? = null,
37
38          // Type string
39          var minLength: Int? = null,
40          var maxLength: Int? = null,
41          var pattern: String? = null,
42          var format: String? = null,
43
44          // Type number and integer
45          var multipleOf: Int? = null,
46          var minimum: Int? = null,
47          var exclusiveMinimum: Int? = null,
48          var maximum: Int? = null,
49          var exclusiveMaximum: Int? = null,
50
51          // Type array
52          var items: MutableMap<String, String>? = null,
53          var prefixItems: MutableList<Schema>? = null,
54          var contains: Schema? = null,
55          var minContains: Int? = null,
56          var maxContains: Int? = null,
57          var minItems: Int? = null,
58          var maxItems: Int? = null,
59          var uniqueItems: Boolean? = null,
60
61          // Schema composition
62          var allOf: MutableList<Schema>? = null,
63          var anyOf: MutableList<Schema>? = null,
64          var oneOf: MutableList<Schema>? = null,
65          @get:JsonProperty("not") var notOf: Schema? = null,
66
67          // Conditionally
68          var dependentRequired: MutableMap<String, List<String>>? = null,
69          var dependentSchemas: MutableMap<String, Schema>? = null,
70          @get:JsonProperty("if") var ifClause: Schema? = null,
71          @get:JsonProperty("then") var thenClause: Schema? = null,
72          @get:JsonProperty("else") var elseClause: Schema? = null
73          ): Serializable {
74                  fun setPropertyTypeAndFormat(key: String, value: String?) {
75                          // TODO("Check validator for format keyword: it needs pattern addendum?")
76                          val newtype: Pair<String, String?> = data_type_and_format(value ?: "")
77                          if (this.properties?.containsKey(key) == true) {
78                                  this.properties?.get(key).apply {
79                                          if (this?.type?.equals("NA") == true) {
80                                                  this.type = newtype.first
81                                                  this.format = newtype.second
82                                          } else {
83                                                  val check = check_domain(newtype.first, this?.type
                                                        ?: "")
84                                                  if (!check.first) {
85                                                          this?.comment = "More␣types␣for␣properties.␣
                                                                From␣${this?.type}␣to␣${newtype.first}.␣
                                                                Set␣to␣${check.second}."
86                                                          this?.type = check.second
87                                                          this?.format = null
88                                                  } else {
89                                                          // TODO("Check and set minval, maxval and
                                                                other constraints")
90                                                          // The type must be set for promoting domain
                                                                in case of included set (e.g. integer to
                                                                number)
91                                                          this?.type = check.second
92                                                  }
93                                          }
94                                  }
95                          } else {
96                                  this.properties?.put(
97                                  key, Schema(schema = null, id = null, type = newtype.first, format
                                        = newtype.second)
98                                  )
```

```
 99                          }
100                     }
101                }
```

Capitolo 3

Implementazione progetto

Il capitolo tratta l'implementazione del progetto.

Inizialmente vengono descritte le scelte fatte in funzione dei requisisti elencati nel par.1.2. Successivamente viene data una panoramica complessiva dell'architettura per proseguire con il commento sulla struttura dei microservizi. Infine, vengono illustrati i metodi per installare ed eseguire il progetto.

3.1 Aderenza ai requisiti

L'implementazione dei requisiti definiti nel par.1.2 ha tenuto conto di due aspetti[1]. Il primo riguarda la necessità di fornire nel più breve tempo possibile un prototipo utile per effettuare delle valutazioni su come proseguire lo sviluppo tenendo conto del limitato numero di risorse disponibili per lo sviluppo del progetto. e delle difficoltà ad interfacciarsi con la struttura dell'ospedale che si occupa delle infrastrutture informatiche. Questo bisogno è anche legato all'esigenza di collaborare con gli sviluppatori del frontend evitando di dover rivedere le specifiche di comunicazione con l'applicazione web in un secondo momento. Il secondo che, almeno per alcuni aspetti, non tutte le specifiche potevano essere definite in fase iniziale. Alcune problematiche non potevano essere approfondite nella prima fase di analisi in quanto non è stato possibile confrontarsi con i responsabili dell'ospedale (ad esempio riguardo l'hardware da utilizzare, il sistema di deploy e il sistema di autenticazione) o si è ritenuto prematuro affrontare alcune questioni che potrebbero richiedere diversi approcci a seconda di come si deciderà di intervenire sul formato dei dati (in particolare per la produzione dei report e per l'esportazione dei dati). Infatti, si è deciso, come primo step, di importare le informazioni provenienti dal vecchio database mantenendo la struttura dei dati stessi, corretti laddove possibile in fase di conversione. L'ottimizzazione delle strutture dati verrà fatta una volta che il reparto eseguirà un'analisi del funzionamento della versione beta del progetto. Questa ottimizzazione potrà prevedere di rivedere non solo i contenuti degli schemi ma anche una eventuale redistribuzione per rendere più agevole la gestione dell'informazione[2].

3.1.1 Sicurezza e gestione ruoli utenza

Questo requisito richiede di sviluppare due funzionalità: la gestione degli utenti per ruoli e le funzioni di auditing che permettano di verificare chi ha effettuato le modifiche e quando. Per la gestione dei ruoli, in attesa di verificare se sarà possibile o necessario integrare il progetto con i sistemi di autenticazione dell'ospedale, è stato utilizzato un servizio che implementa le specifiche OAuth2 (`Keycloak`) nel quale sono stati inseriti i ruoli indicati nei requisiti: `ADMIN` per

[1] È stato utilizzato un approccio "agile".

[2] Ad esempio lo schema delle visite contiene 328 proprietà e potrebbe essere oggetto di suddivisione.

gli amministratori, `PHYSICIAN` per i medici e `USER` per gli altri utenti. Il progetto `Keycloak` potrà comunque essere utilizzato anche in produzione in quanto rispetta i requisiti richiesti dal GDPR per l'autenticazione forte. Per la gestione dell'auditing e la sicurezza del db è stato utilizzato `MongoDB` che garantisce di poter crittografare i dati e di utilizzare il journaling per garantire l'integrità e la rigenerazione delle operazioni, quando necessario. In ogni collezione è presente una proprietà con le informazioni relative all'utente che ha effettuato le variazioni sul db. Ogni singolo microservizio che necessita di proteggere gli endpoint in relazione al ruolo ricoperto può includere il package `it.polito.links.ccinfo.security` che fornisce l'infrastruttura necessaria per filtrare le richieste e utilizzare le annotazioni per gestire l'autorizzazione ad eseguire determinate operazioni[3]

3.1.2 Sistema distribuito e accessibilità offline

Il progetto è stato sviluppato con una architettura a microservizi accessibile attraverso delle chiamate REST, garantendo la possibilità di essere acceduto da qualsiasi dispositivo collegato alla rete. Ovviamente questo requisito richiede che il frontend sia adeguato[4] e che il server sia in grado di gestire un controllo di versione sui record per evitare sovrascritture e gestire la concorrenza[5]. Inoltre, il server deve essere in grado di fornire uno strumento per scaricare le informazioni utili per gestire la CCINFO prima che il client si scolleghi dalla rete. Il progetto mette a disposizione una serie di endpoint per permettere al client di creare un DB temporaneo che permetta all'utente di lavorare offline e di essere aggiornato a posteriori con delle chiamate asincrone. Attraverso il controllo di versione sui record è in grado di comunicare al client l'eventuale conflitto durante la fase di sincronizzazione.

3.1.3 Report ed esportazione dati

Nel caso della produzione dei report e dell'esportazione dei dati si è deciso di rimandare l'approfondimento dell'analisi e di conseguenza lo sviluppo dopo la prima fase di test sulla versione beta. In particolare, in questa fase verrà analizzato quali informazioni, e di che tipo, verranno mantenute nel db e come eventualmente verranno riorganizzate. La definizione di questi servizi dipende fortemente dalla conferma della modalità con cui i dati sono storicizzati e anche la modalità di lettura ed elaborazione richiede che la versione del formato dei dati sia sufficientemente stabile[6].

3.1.4 Importazione dei dati

Per l'importazione dei dati è stato sviluppato un package che esegue l'elaborazione correggendo le anomalie. In particolare, il package permette di effettuare delle importazioni parziali per poter rieseguire l'elaborazione in fasi successive intercettando solo gli inserimenti e le modifiche effettuate partendo da una data fissata[7].

3.1.5 Formato e amministrazione dati

Il requisito verrà soddisfatto con l'adozione di un database schema less (`MongoDB`) per poter archiviare i dati di qualsiasi natura abbinato alla definizione di schemi aderenti alle specifiche

[3] Attraverso le configurazioni e le annotazioni di *Spring* possono essere posti dei vincoli su chi può accedere agli endpoint o a un sottoinsieme di dati[11].

[4] Ad esempio può essere sviluppato come PWA (Progressive Web Application) che, attraverso l'utilizzo dei service workers, può gestire gli aggiornamenti in modo asincrono.

[5] Questa necessità esiste comunque, soprattutto per applicazioni distribuite, in particolare per quelle in cui la comunicazione è senza stato, come nel caso del protocollo HTTP.

[6] Eventuali revisioni del progetto che possano intervenire sulla struttura dei dati impatterebbero negativamente su quanto progettato per l'esportazione o per la produzione dei report.

[7] Il vecchio programma permette di estrarre dal db solo le modifiche effettuata a partire da una certa data. In questo modo sarà anche possibile estrarre i dati dai diversi dispositivi utilizzati nel reparto effettuando la sincronizzazione dei dati direttamente nella nuova procedura.

`JSON Schema, versione 2020-12`. Per la modifica e l'inserimento dei nuovi schemi si utilizzerà lo stesso sistema di validazione (ovviamente rispetto alle specifiche di JSON Schema stesso). L'aggiornamento degli schemi sarà ammesso solo agli utenti con il ruolo di amministratore.

La lista dei farmaci è stata reperita dal sito dell'AIFA[8] scaricando il file contenente i farmaci di classe A, su indicazione dei medici del reparto. L'aggiornamento dell'elenco sul sito dell'AIFA risale al 15 dicembre 2022. Nella versione prototipo del progetto non è previsto di aggiornare in modo automatico la lista. L'utilizzo di una funzione online potrà essere sostituita da un batch periodico o a richiesta che, scaricata la lista aggiornata dal sito, provvede ad aggiornare il database.

Per i flussi diagnostici è stato generato un primo esempio di grafo che, una volta sottoposto ai medici del reparto nella fase di test dell'applicazione, verrà integrato e accompagnato da altri flussi su richiesta dei medici stessi.

3.1.6 Utilizzo su qualsiasi dispositivo

Ovviamente l'esigenza si manifesta nel tipo di frontend utilizzato che, come già affermato, non è oggetto di questa tesi. La separazione del backend dal frontend e la natura distribuita dell'architettura scelta per implementare questo progetto garantisce la possibilità di creare degli strumenti indipendenti dal dispositivo utilizzato.

3.2 Architettura implementazione

Il progetto è costituito da 16 moduli, elencati nella tab. 3.1.

I tre moduli di tipo "package"[9] costituiscono le librerie utilizzabili per l'accesso ai dati, la gestione delle eccezioni con la generazione in forma standard delle risposte alle chiamate che terminano con errore e l'infrastruttura per gestire i filtri di autenticazione e abilitare la sicurezza all'interno dei microservizi.

Il processo batch di mascheramento delle informazioni e caricamento degli schemi provenienti dal software `Hypermacondo` è già stato ampiamente descritto nel paragrafo 2.2.1.

Il modulo di inizializzazione è un processo batch eseguibile all'atto dell'inizializzazione del sistema e per importare nelle fasi successive le modifiche intervenute dall'avvio in fase di test del progetto alla messa in produzione dell'applicativo con la contestuale dismissione del vecchio programma.

I microservizi, descritti in dettaglio nel paragrafo 3.3, costituiscono i moduli che effettivamente vengono eseguiti sulla macchina ospite e la cui architettura è visibile in figura 3.1.

Il gateway (Spring Cloud Gateway) è l'unico servizio effettivamente esposto verso l'esterno. Il suo compito è di smistare verso gli altri microservizi le richieste provenienti dalla rete in base al path della richiesta. Dalla figura si può notare che le chiamate sul path principale[10] vengono smistate al frontend (React Frontend) che, non essendo oggetto di questa tesi, non verrà descritto. Tutte le altre chiamate sono indirizzate verso il microservizio competente.

Esistono alcuni microservizi per i quali non vengono mappate le API in modo che non siano richiamabili direttamente dall'esterno[11] e che hanno una funzionalità trasversale rispetto a tutti gli altri:

---

[8] Vedi (https://www.aifa.gov.it/web/guest/liste-dei-farmaci)

[9] Benché anche altri moduli siano dei package nella terminologia Kotlin, qui con il termine package si è voluto identificare quei moduli che non generano direttamente dei microservizi ma che costituiscono delle librerie che possono (o devono) essere utilizzate dagli altri moduli per aggiungere quelle funzionalità che sono trasversali a tutti i moduli del progetto.

[10] Il path principale è identificato dal simbolo / ed è disegnata in colore rosso

[11] Solo in fase di test o di manutenzione del server e conoscendo gli indirizzi dei container (docker) o dei pod (kubernetes) che li eseguono o attivando delle funzionalità di proxy è possibile accedere direttamente a questi processi.

Nome modulo	Tipo	Descrizione
data	package	Accesso ai database e validazione dei dati
error	package	Definizione eccezioni e gestione comunicazione errori ai client
security	package	Modulo di configurazione dei componenti iniettabili per l'inserimento dei filtri di sicurezza
data_obfuscation	processo batch	Processo batch utilizzato in una fase iniziale per il mascheramento dei dati e per l'estrazione degli schemi dei dati da utilizzare nell'applicativo
init	processo batch	Processo batch utilizzato per la conversione dei dati dalla vecchia versione al nuovo progetto e per l'importazione/impostazione degli altri contenuti dei database (utenti, farmaci, flussi diagnostici)
gateway	microservizio	Servizio esposto all'esterno che esegue il routing delle richieste
config	microservizio	Servizio di supporto che fornisce i parametri di configurazione agli altri servizi
oauth	microservizio	Servizio di autenticazione `Keycloak`
sso	microservizio	Servizio REST per il single sign on
users	microservizio	Servizio REST per l'amministrazione degli utenti
schema	microservizio	Servizio REST per il reperimento e l'amministrazione degli schemi dei dati e dello schema di validazione
patients	microservizio	Servizio REST per la gestione dei dati relativi ai pazienti
drugstore	microservizio	Servizio REST per il reperimento della lista dei farmaci
reports	microservizio	Servizio REST per la generazione e la gestione dei referti
exporter	microservizio	Servizio REST per l'esportazione dei dati
flows	microservizio	Servizio REST per la gestione dei flussi diagnostici

Tabella 3.1: Elenco moduli che compongono il progetto

- Il server di configurazione (sviluppato utilizzando il progetto `Spring Cloud Config`), si appoggia ad un contenitore di proprietà e segreti (`Vault`) e fornisce i parametri di configurazione per avviare e aggiornare gli altri microservizi;
- Il database delle cartelle informatizzate (CCInfo DB) che viene eseguito in più repliche per permettere la gestione delle transazioni, acceduto tramite le funzionalità messe a disposizione dal package `data`;
- Il servizio di autenticazione (`Keycloak`) che offre al servizio di single sign on (SSO service) l'accesso per autenticare gli utenti generando i JWT (JSON Web Token) di accesso e di refresh, mentre per gli altri servizi espone un endpoint per verificare se il token utilizzato dai client per l'autenticazione è valido e quali ruoli autorizzativi sono attribuiti al proprietario del token.

I microservizi che si occupano dell'esportazione dei dati e della gestione dei report non sono ancora stati sviluppati benché siano previsti nel disegno generale del progetto.

Si è scelto di non implementare un servizio di service discovery, comune nelle architetture a microservizi, perché si ritiene che la soluzione migliore per l'installazione del progetto sia l'utilizzo di `Kubernetes`, per il quale vengono forniti i file di configurazione, che gestisce in autonomia l'orchestrazione dei servizi. Per contro, non ci si è voluti affidare al servizio di `Ingress` della stessa specifica perché è stato riscontrato che diverse implementazioni di `Kubernetes`[12] hanno

[12]Il test è stato effettuato con `microk8s` e `k3s` che implementano la specifica con diversi componenti, `Ingress Nginx` e `Traefik` rispettivamente, che basano alcune modalità di esecuzione (ad esempio per la riscrittura dei path o la gestione delle richieste `CORS`) su annotazioni di tipo diverso rendendo complicato produrre file di configurazione stabili nel tempo.

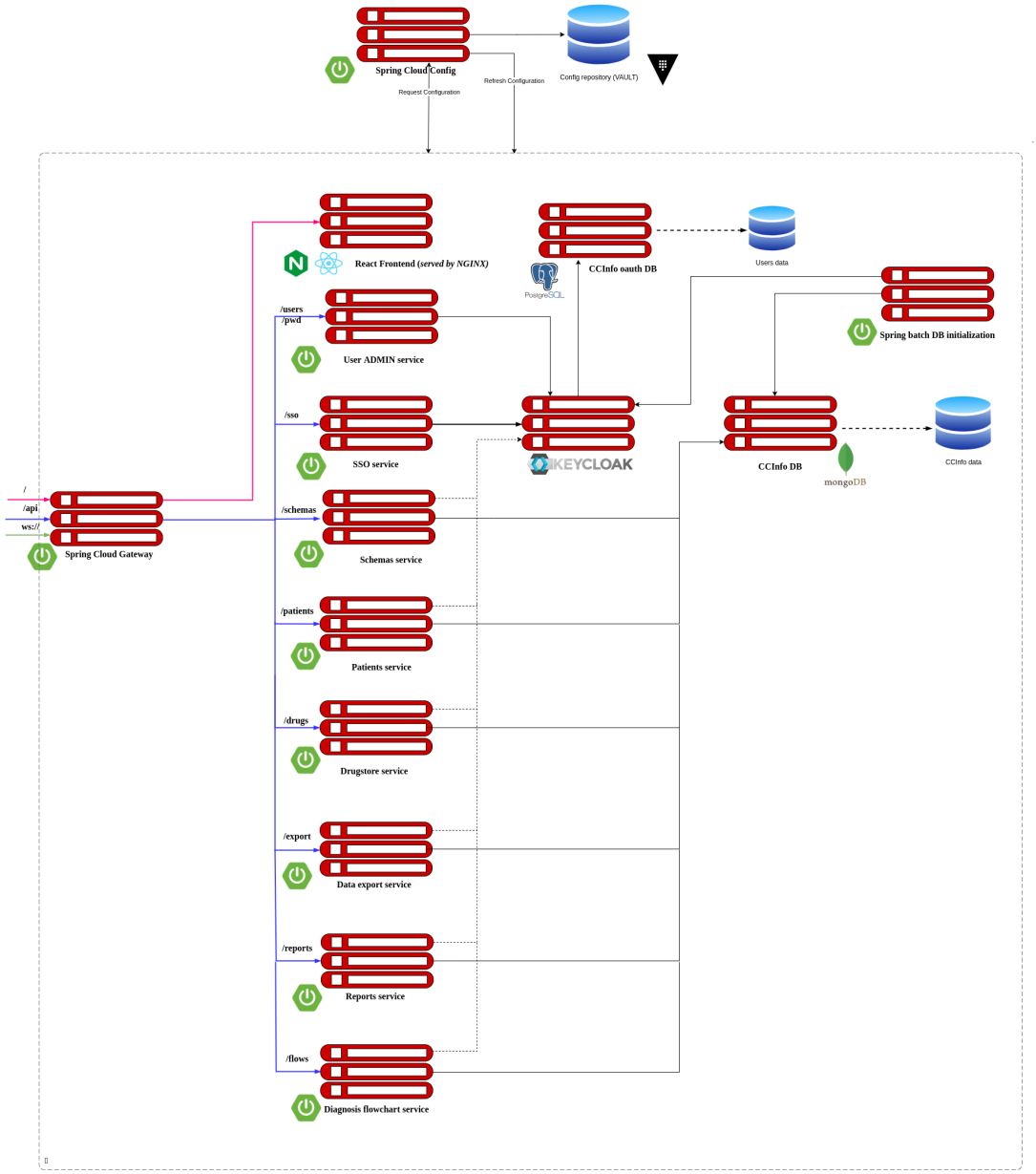


Figura 3.1: Architettura implementazione

comportamenti differenti. Di conseguenza il progetto contiene il modulo `Gateway` descritto nel paragrafo 3.3.3.

3.2.1 Database

I dati sono registrati su due database. Entrambi sono interfacciati dal package descritto nel paragrafo 3.3.1 in modo da garantire la separazione tra i microservizi e l'effettiva implementazione del DB[13]. Gli utenti sono gestiti tramite il servizio `Keycloak` che, a sua volta, utilizza un database relazionale (`PostgreSQL`) per archiviare i dati. Le altre informazioni risiedono sul database `MongoDB`.

[13]Come già accennato non è ancora stato possibile verificare con il committente quali scelte operare riguardo l'infrastruttura hardware e se dovranno esistere delle interazioni tra il sistema informativo dell'ospedale e il progetto oggetto di questa tesi.

Le collezioni presenti nel prototipo sono cinque: `Drugs` (lista dei farmaci), `Flows` (flussi diagnostici), `Patients` (dati dei pazienti), `Schemas` (schemi dei dati della CCINFO), e `logs` (anomalie riscontrate) Ogni record contiene delle informazioni utili alla gestione della concorrenza e alle funzioni di auditing. Infatti, nella proprietà `auditData` sono presenti il numero di versione del record e una collezione di informazioni che riportano, per ogni singolo elemento, lo userid di chi ha effettuato la modifica sul database, il tipo di operazione effettuata (inserimento o aggiornamento) e il timestamp di quando la variazione è avvenuta. Per conoscere i dettagli della modifica potrà essere sfruttato il tool `wt` della libreria `WiredTiger`, il motore di archiviazione utilizzato dalla versione di `MongoDB` utilizzata in questo progetto[14].

La lista dei farmaci è stata importata dal sito dell'AIFA e ribaltata con gli stessi dati nella collezione `Drugs` per la quale è stato creato un indice testuale sull'insieme di proprietà che si sono ritenute utili per facilitare la ricerca agli operatori: il nome del principio attivo, il nome del farmaco e la descrizione del gruppo presenti nell'archivio prodotto da AIFA.

I flussi diagnostici sono gestiti tramite una struttura a nodi con le informazioni utili al frontend per disegnare a video il grafico che guida il medico nella decisione dei passi da intraprendere per proseguire la diagnosi e definire le cure. In ogni nodo sono definite le caratteristiche che ne identificano la tipologia e che ne definiscono la modalità di visualizzazione a video. La connessione dei nodi determina lo sviluppo del flusso in relazione al successo o al fallimento dello step rappresentato dal nodo stesso.

Ogni singolo paziente è rappresentato da un documento unico all'interno della collezione `Patients`. La struttura dell'informazione, di cui un estratto è visibile nel listato 3.1, è suddivisa in due gruppi, i dati personali e i dati clinici.

I dati per la gestione anagrafica del paziente contengono le informazioni personali, la chiave presente nel software `HyperMacondo` e il campo per la ricerca testuale formato dal cognome e nome in minuscolo[15]. La chiave di riferimento proveniente dal vecchio software duplicata con e senza gli zeri significativi è motivata dalle necessità descritte nel paragrafo 3.3.2 relativo all'inizializzazione del database. Il campo per la ricerca testuale è utilizzato da un indice aggiuntivo creato sulla collezione.

I dati clinici sono suddivisi in cluster. Nello schema di riferimento del tipo di dato che si vuole gestire viene indicato il nome del contenitore all'interno della proprietà `clinicRecords`. La scelta della quantità e del nome dei contenitori è lasciata agli utenti abilitati a gestire gli schemi. Nella definizione del prototipo si è deciso di creare tre proprietà:

1. esami, che contiene gli schemi ABPM, ecocardio, ematochimicaaltri, ematochimicagenerali1, ematochimicagenerali2, ematochimicaormonale, esamiestemporanei, ed esamistrumentali;
2. dati, che contiene gli schemi familiarita, patologie e terapie;
3. visite, che contiene lo schema visite.

Listato 3.1: Estratto anagrafica paziente

```
1       {
2               "_id" : "64aa622780a709674932ad4d",
3               "auditData" : {
4                       "recordVersion" : 25,
5                       "items" : [
6                       {
7                               "user_id" : "user_init",
8                               "type" : "INSERT",
```

[14]Le funzioni avanzate di auditing non sono state sviluppate all'interno di questo progetto per ragioni di tempo rimandando l'implementazione a futuri sviluppi. Si veda il capitolo 4.2.4.

[15]Il nome e cognome presenti nel listato fanno riferimento al valore mascherato secondo quanto illustrato nel paragrafo 2.2.1.

```
                        "date" : "2023-07-09T07:30:47.891142813"
                    },
                    {...}
                ]
            },
            "hypermacondoid" : "00791",
            "hypermacondoidzerodropped" : "791",
            "searchstring" : "a6ez oerkuqb",
            "personalData" : {...},
            "clinicRecords" : {
                "esami" : [
                    {
                        "_id" : "64aa64f780a709674933a368",
                        "auditData" : {...},
                        "schemaRef" : "64aa621580a7096749328560",
                        "schemaName" : "ematochimicaaltri",
                        "schemaType" : "ESAME",
                        "content" : {...}
                    },
                    {
                        "_id" : "64aa67d080a70967493491d9",
                        "auditData" : {...},
                        "schemaRef" : "64aa621580a7096749328561",
                        "schemaName" : "ematochimicagenerali1",
                        "schemaType" : "ESAME",
                        "content" : {...}
                    },
                    {...}
                ],
                "dati" : [
                    {
                        "_id" : "64aa827580a70967493cd4a9",
                        "auditData" : {...},
                        "schemaRef" : "64aa621580a7096749328568",
                        "schemaName" : "terapia",
                        "schemaType" : "TERAPIA",r
                        "content" : {...}
                    },
                    {...}
                ],
                "visite" : [
                    {
                        "_id" : "64aa8a7680a7096749403969",
                        "auditData" : {...},
                        "schemaRef" : "64aa621580a7096749328569",
                        "schemaName" : "visite",
                        "schemaType" : "VISITA",
                        "content" : {...}
                    ]
            },
            "_class" : "it.polito.links.ccinfo.data.model.Patient"
        }
```


La struttura degli schemi è visibile nel listato 3.2. Il campo `patientDataCluster` identifica la sezione all'interno dei documenti della collezione `Patients` in cui i dati vanno inseriti. Lo schema,

aderente alle specifiche di `JSON Schema, versione 2020-12`[16], è definito nel campo schema ed è stato impostato dal modulo `data_obfuscation`.

Listato 3.2: Definizione schemi

```
{
        "_id" : "64aa621580a7096749328569",
        "auditData" : {
                "recordVersion" : 1,
                "items" : [
                        {
                                "user_id" : "user_init",
                                "type" : "INSERT",
                                "date" : "2023-07-09T07:30:29.225799651"
                        }
                ]
        },
        "schemaName" : "visite",
        "description" : "Visite",
        "type" : "VISITA",
        "patientDataCluster" : "visite",
        "schema" : {
                "_id" : "/visite",
                "schema" :
                    "https://json-schema.org/draft/2020-12/schema",
                "type" : "object",
                "title" : "Visite",
                "description" : "Visite",
                "comment" : "type:VISITA;cluster:visite",
                "properties" : {
                        "IDVisita" : {
                                "type" : "string"
                        },
                        "IDPaziente" : {
                                "type" : "string"
                        },
                        "IDMedico" : {
                                "type" : "string"
                        },
                        "DataVisita" : {
                                "type" : "string",
                                "format" : "date"
                        },
                        "eta" : {
                                "type" : "integer"
                        },
                        {...}
                },
        "_class" : "it.polito.links.ccinfo.data.model.SchemaStore"
}
```

[16] I campi che nello standard sono preceduti dal simbolo $ sono definiti nel database senza questo carattere iniziale in quanto lo stesso carattere è utilizzato da `MongoDB` con un significato particolare nella definizione delle query oltre a non essere ammesso come carattere per la definizione delle variabili nel linguaggio `Kotlin`. La validazione dello schema è garantita dalla ridefinizione dei metodi `getter` per la classe che modella questa collezione. Si veda il paragrafo 3.3.1.

3.3 Descrizione moduli

3.3.1 Package utilizzati dagli altri moduli

Tre moduli costituiscono i package utilizzati dai microservizi e dal processo batch.

Package it.polito.links.ccinfo.data

Questo modulo costituisce l'interfaccia per l'accesso ai database. Malgrado il pattern normalmente utilizzato in una architettura a microservizi sia quello di gestire in modo indipendente per ogni singolo modulo l'archiviazione dei dati[10], per il rilascio del prototipo si è deciso di mantenere l'accesso e la manutenzione del DB sotto forma di un unico package utilizzato da tutti i microservizi in attesa di definire meglio con il committente gli eventuali collegamenti con il sistema informativo esistente.

La funzione del package è quella di disaccoppiare i moduli che devono interagire con il database dall'implementazione dello stesso. La modifica del database sottostante, come nel caso della sostituzione del sistema di autenticazione e quindi dell'anagrafica dei medici e degli operatori, impatterà esclusivamente sulla ridefinizione di questo package senza dover intervenire con modifiche sugli altri moduli.

Il sotto package `services` espone i servizi per manipolare i dati presenti sul sistema utilizzando la semantica di `Spring` e possono essere iniettati nei `controller` con la annotazione `@Autowired`. Il servizio dedicato alla manutenzione degli utenti interagisce con le API di `Keycloak` per recuperare e variare le informazioni degli utenti abilitati a operare nel sistema. Invece, i servizi che accedono ai dati dei pazienti, agli schemi, ai flussi diagnostici e ai farmaci, utilizzano delle interfacce che estendono `MongoRepository` per accedere direttamente al database descritto nel paragrafo 3.2.1. La validazione dei documenti è effettuata tramite la funzione di utilità del file `Utils.kt` del sottopackage `utils` illustrata nel listato 3.3

Listato 3.3: Funzioni di utilità del package data

```
1  import org.bson.Document as Bson
2
3  fun buildPatientSearchString(document: Bson): String {
4          return ((document?.getString("Cognome") ?: "") + "␣" +
5          (document?.getString("Nome") ?: ""))
6          .lowercase()
7  }
8
9  /**
10 * Dcoument Validation
11 * @param document The org.bson.Document to be validated
12 * @param schema The org.bson.Document reference schema
13 * @return an optional com.github.erosb.jsonsKema.ValidationFailure
14 */
15 fun schemaValidation(document: Bson, schemaStore: SchemaStore): ValidationFailure? {
16         // Parsing schema and load for validator
17         val schema = BsonUtils.asBson(schemaStore.schema).toBsonDocument()
18         val schemaJson = JsonParser(schema.toJson()).parse()
19         val jsonsKema = SchemaLoader(schemaJson).load()
20         val validator = Validator.forSchema(jsonsKema)
21         // Validate document content
22         // Return null for valid documents
23         return validator.validate(JsonParser(document.toJson()).parse())
24 }
```

Per comunicare con gli altri moduli il package espone una serie di API nel sotto package `DTO` (Data Transfer Object) che costituiscono le strutture dati utilizzate per comunicare e astrarre la comunicazione rispetto all'implementazione del database sottostante. Gli oggetti del modello (le classi che mappano i dati del database) utilizzano una forma standard per trasformare un DTO nel modello per il DB e viceversa, come illustrato nel listato 3.4 che riporta la definizione della collezione `Schemas`. Il costruttore può essere richiamato passando il `DTO` come parametro e

dall'oggetto si può ricavare il `DTO` tramite la funzione `buildDTO`[17]. Dal listato si può anche notare che per le proprietà che nello standard `JSON Schema, versione 2020-12` sono precedute dal simbolo $ o che utilizzano dei vocaboli che rappresentano parole chiave per il linguaggio `Kotlin` sia stato ridefinito il metodo `getter` per automatizzare la valorizzazione del record da inserire sul db.

Listato 3.4: Model e DTO

```kotlin
/**
* Constants
*/
val SCHEMA_REF = "https://json-schema.org/draft/2020-12/schema"
val SCHEMA_ID = "http://links.polito.it/schemas/ccinfo"

@JsonInclude(JsonInclude.Include.NON_NULL)
@JsonIgnoreProperties
data class Schema(
        // Common to all types
        @get:JsonProperty("\$schema", index = 0) val schema: String? = null,
        @get:JsonProperty("\$id", index = 1) val id: String? = null,
        @get:JsonProperty(index = 10) var type: String? = null, // object, string, number,
            integer, array, boolean, null
        @get:JsonProperty("enum") var enumProp: MutableList<Any>? = null, // valid also without
            type
        var title: String? = null,
        var description: String? = null,
        var default: Any? = null,
        var examples: MutableList<Any>? = null,
        var deprecated: Boolean? = null,
        var readOnly: Boolean? = null,
        var writeOnly: Boolean? = null,
        @get:JsonProperty("\$comment") var comment: String? = null,
        @get:JsonProperty("const") var constValue: Any? = null,
        @get:JsonProperty("\$defs", index = 999) var defs: MutableMap<String, Schema>? = null,
        @get:JsonProperty("\$ref", index = 20) var ref: String? = null,
        var definitions: MutableMap<String, Schema>? = null,
        @get:JsonProperty("\$anchor") var anchor: String? = null,

        // Type object
        @get:JsonProperty(index = 30) var properties: MutableMap<String, Schema>? = null,
        var minProperties: Int? = null,
        var maxProperties: Int? = null,
        var additionalProperty: Boolean? = null,
        @get:JsonProperty(index = 25) var required: MutableList<String>? = null,
        var unevaluatedProperties: Boolean? = null,

        // Type string
        var minLength: Int? = null,
        var maxLength: Int? = null,
        var pattern: String? = null,
        var format: String? = null,

        // Type number and integer
        var multipleOf: Int? = null,
        var minimum: Int? = null,
        var exclusiveMinimum: Int? = null,
        var maximum: Int? = null,
        var exclusiveMaximum: Int? = null,

        // Type array
        var items: MutableMap<String, String>? = null,
        var prefixItems: MutableList<Schema>? = null,
        var contains: Schema? = null,
        var minContains: Int? = null,
        var maxContains: Int? = null,
        var minItems: Int? = null,
```

[17]La funzione per reperire il `DTO` non puòessere denominata `getDTO` in quanto `Kotlin` la riconoscerebbe come funzione `getter` e creerebbe una proprietà aggiuntiva `dto` nel di tipo `schemaDTO` nella lettura dei documenti da disco.

```
57          var maxItems: Int? = null,
58          var uniqueItems: Boolean? = null,
59
60          // Schema composition
61          var allOf: MutableList<Schema>? = null,
62          var anyOf: MutableList<Schema>? = null,
63          var oneOf: MutableList<Schema>? = null,
64          @get:JsonProperty("not") var notOf: Schema? = null,
65
66          // Conditionally
67          var dependentRequired: MutableMap<String, List<String>>? = null,
68          var dependentSchemas: MutableMap<String, Schema>? = null,
69          @get:JsonProperty("if") var ifClause: Schema? = null,
70          @get:JsonProperty("then") var thenClause: Schema? = null,
71          @get:JsonProperty("else") var elseClause: Schema? = null
72          ) {
73                  constructor(schemaDTO: SchemaDTO) : this(
74                          schema = schemaDTO.schema,
75                          id = schemaDTO.id,
76                          type = schemaDTO.type,
77                          enumProp = schemaDTO.enumProp,
78                          title = schemaDTO.title,
79                          description = schemaDTO.description,
80                          default = schemaDTO.default,
81                          examples = schemaDTO.examples,
82                          deprecated = schemaDTO.deprecated,
83                          readOnly = schemaDTO.readOnly,
84                          writeOnly = schemaDTO.writeOnly,
85                          comment = schemaDTO.comment,
86                          constValue = schemaDTO.constValue,
87                          defs = schemaDTO.defs?.mapValues { Schema(it.value) } as
                                MutableMap<String, Schema>?,
88                          ref = schemaDTO.ref,
89                          definitions = schemaDTO.definitions?.mapValues { Schema(it.value) } as
                                MutableMap<String, Schema>?,
90                          anchor = schemaDTO.anchor,
91
92                          // Type object
93                          properties = schemaDTO.properties?.mapValues { Schema(it.value) } as
                                MutableMap<String, Schema>?,
94                          minProperties = schemaDTO.minProperties,
95                          maxProperties = schemaDTO.maxProperties,
96                          additionalProperty = schemaDTO.additionalProperty,
97                          required = schemaDTO.required,
98                          unevaluatedProperties = schemaDTO.unevaluatedProperties,
99
100                         // Type string
101                         minLength = schemaDTO.minLength,
102                         maxLength = schemaDTO.maxLength,
103                         pattern = schemaDTO.pattern,
104                         format = schemaDTO.format,
105
106                         // Type number and integer
107                         multipleOf = schemaDTO.multipleOf,
108                         minimum = schemaDTO.minimum,
109                         exclusiveMinimum = schemaDTO.exclusiveMinimum,
110                         maximum = schemaDTO.maximum,
111                         exclusiveMaximum = schemaDTO.exclusiveMaximum,
112
113                         // Type array
114                         items = schemaDTO.items,
115                         prefixItems = schemaDTO.prefixItems?.map { Schema(it) }?.toMutableList(),
116                         contains = schemaDTO.contains?.let { Schema(it) },
117                         minContains = schemaDTO.minContains,
118                         maxContains = schemaDTO.maxContains,
119                         minItems = schemaDTO.minItems,
120                         maxItems = schemaDTO.maxItems,
121                         uniqueItems = schemaDTO.uniqueItems,
122
123                         // Schema composition
124                         allOf = schemaDTO.allOf?.map { Schema(it) }?.toMutableList(),
125                         anyOf = schemaDTO.anyOf?.map { Schema(it) }?.toMutableList(),
```

```
126                     oneOf = schemaDTO.oneOf?.map { Schema(it) }?.toMutableList(),
127                     notOf = schemaDTO.notOf?.let { Schema(it) },
128
129                     // Conditionally
130                     dependentRequired = schemaDTO.dependentRequired,
131                     dependentSchemas = schemaDTO.dependentSchemas?.mapValues {
                            Schema(it.value) } as MutableMap<String, Schema>?,
132                     ifClause = schemaDTO.ifClause?.let { Schema(it) },
133                     thenClause = schemaDTO.thenClause?.let { Schema(it) },
134                     elseClause = schemaDTO.elseClause?.let { Schema(it) }
135             )
136
137             fun buildDTO(): SchemaDTO = SchemaDTO(
138                     schema = this.schema,
139                     id = this.id,
140                     type = this.type,
141                     enumProp = this.enumProp,
142                     title = this.title,
143                     description = this.description,
144                     default = this.default,
145                     examples = this.examples,
146                     deprecated = this.deprecated,
147                     readOnly = this.readOnly,
148                     writeOnly = this.writeOnly,
149                     comment = this.comment,
150                     constValue = this.constValue,
151                     defs = this.defs?.mapValues { it.value.buildDTO() } as MutableMap<String,
                            SchemaDTO>?,
152                     ref = this.ref,
153                     definitions = this.definitions?.mapValues { it.value.buildDTO() } as
                            MutableMap<String, SchemaDTO>?,
154                     anchor = this.anchor,
155
156                     // Type object
157                     properties = this.properties?.mapValues { it.value.buildDTO() } as
                            MutableMap<String, SchemaDTO>?,
158                     minProperties = this.minProperties,
159                     maxProperties = this.maxProperties,
160                     additionalProperty = this.additionalProperty,
161                     required = this.required,
162                     unevaluatedProperties = this.unevaluatedProperties,
163
164                     // Type string
165                     minLength = this.minLength,
166                     maxLength = this.maxLength,
167                     pattern = this.pattern,
168                     format = this.format,
169
170                     // Type number and integer
171                     multipleOf = this.multipleOf,
172                     minimum = this.minimum,
173                     exclusiveMinimum = this.exclusiveMinimum,
174                     maximum = this.maximum,
175                     exclusiveMaximum = this.exclusiveMaximum,
176
177                     // Type array
178                     items = this.items,
179                     prefixItems = this.prefixItems?.map { it.buildDTO() }?.toMutableList(),
180                     contains = this.contains?.let { it.buildDTO() },
181                     minContains = this.minContains,
182                     maxContains = this.maxContains,
183                     minItems = this.minItems,
184                     maxItems = this.maxItems,
185                     uniqueItems = this.uniqueItems,
186
187                     // Schema composition
188                     allOf = this.allOf?.map { it.buildDTO() }?.toMutableList(),
189                     anyOf = this.anyOf?.map { it.buildDTO() }?.toMutableList(),
190                     oneOf = this.oneOf?.map { it.buildDTO() }?.toMutableList(),
191                     notOf = this.notOf?.let { it.buildDTO() },
192
193                     // Conditionally
```

```
194                     dependentRequired = this.dependentRequired,
195                     dependentSchemas = this.dependentSchemas?.mapValues { it.value.buildDTO()
                            } as MutableMap<String, SchemaDTO>?,
196                     ifClause = this.ifClause?.let { it.buildDTO() },
197                     thenClause = this.thenClause?.let { it.buildDTO() },
198                     elseClause = this.elseClause?.let { it.buildDTO() }
199             )
200     }
```

Package it.polito.links.ccinfo.error

In questo modulo viene gestita la comunicazione degli errori con i client che accedono all'applicazione. L'inserimento della dipendenza da questo package automaticamente attiva la generazione di una API predefinita per la comunicazione degli errori al client. Infatti, il modulo espone una API, eventualmente modificabile in accordo con gli sviluppatori dei client, che non richiede ai moduli che lo utilizzano di conoscerne i dettagli. L'insieme del modulo per la gestione degli errori e dell'accesso ai dati fornisce automaticamente la gestione dell'accesso al database e un comportamento uniforme per gestire le eccezioni[18].

Il listato 3.5 presenta un estratto del codice che gestisce la generazione del valore di ritorno in caso di eccezioni.

Listato 3.5: Estratto moduli gestione errori

```
1       data class ApiError(
2               val httpStatus: HttpStatus,
3               @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd/MM/yyyy␣hh:mm:ss")
4               val timestamp: LocalDateTime,
5               val message: String
6       )
7
8       @ControllerAdvice
9       class Advices {
10
11              @ResponseBody
12              @ExceptionHandler(Unauthorized::class, BadCredentialsException::class)
13              @ResponseStatus(HttpStatus.UNAUTHORIZED)
14              fun unauthorizedHandler(ex: Exception): ApiError =
15                      ApiError(HttpStatus.UNAUTHORIZED, LocalDateTime.now(), ex.message?:
                            "Unable␣to␣authenticate" )
16
17              @ResponseBody
18              @ExceptionHandler(InvalidRefreshTokenException::class, ValidatorException::class)
19              @ResponseStatus(HttpStatus.BAD_REQUEST)
20              fun badrequestHandler(ex: Exception): ApiError =
21                      ApiError(HttpStatus.BAD_REQUEST, LocalDateTime.now(), ex.message?:
                            "Malformed␣request" )
22
23              @ResponseBody
24              @ExceptionHandler(UserNotFoundException::class, PatientNotFoundException::class,
                    FlowNotFoundException::class,
25                                      SchemaNotFoundException::class,
                                            ClinicDataNotFoundException::class,
                                            DrugNotFoundException::class)
26              @ResponseStatus(HttpStatus.NOT_FOUND)
27              fun notFoundHandler(ex: Exception): ApiError =
28                      ApiError(HttpStatus.NOT_FOUND, LocalDateTime.now(), ex.message?: "Resource␣
                            not␣found")
29
30              @ResponseBody
31              @ExceptionHandler(ForbiddenResourceException::class)
32              @ResponseStatus(HttpStatus.FORBIDDEN)
33              fun forbiddenHandler(ex: Exception): ApiError =
```

[18]Le eccezioni vengono sollevate dai servizi messi a disposizione dal package `it.polito.links.ccinfo.data` di cui questo modulo può essere considerato un complemento.

```
34                      ApiError(HttpStatus.FORBIDDEN, LocalDateTime.now(), ex.message?: "Access␣
                            to␣this␣resource␣is␣not␣allowed")
35
36              @ResponseBody
37              @ExceptionHandler(InvalidRecordVersionNumberException::class,
                    UserAlreadyExistsException::class,
38                                          DuplicatedSchemaNameException::class)
39              @ResponseStatus(HttpStatus.CONFLICT)
40              fun conflictHandler(ex: Exception): ApiError =
41                      ApiError(HttpStatus.CONFLICT, LocalDateTime.now(), ex.message?: "Your␣
                            environment␣is␣not␣up␣to␣date")
42
43      }
44
45      data class ApiError(
46              val httpStatus: HttpStatus,
47              @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd/MM/yyyy␣hh:mm:ss")
48              val timestamp: LocalDateTime,
49              val message: String
50      )
```

Package it.polito.links.ccinfo.security

Il modulo mette a disposizione dei servizi `REST` una serie di componenti utili a effettuare il controllo degli accessi e delle autorizzazioni. L'inserimento di questa dipendenza unita alla eventuale annotazione per il tipo di protezione che si vuole abilitare sugli endpoint del servizio REST abilita il processo di autenticazione e autorizzazione. Il package definisce il componente `AuthFilter` (vedi listato 3.6) che estende `GenericFilterBean` e che estrae da ogni richiesta effettuata il token presente nell'intestazione `Authorization` della richiesta, ne verifica la validità e imposta nel contesto della richiesta le informazioni legate all'utente che ha richiesto l'operazione.

Listato 3.6: Filtro autenticazione

```
1   @Component
2   class AuthFilter : GenericFilterBean() {
3
4           companion object : KLogging()
5
6           lateinit var userService: UserService
7
8           override fun doFilter(
9                   request: ServletRequest?,
10                  response: ServletResponse?,
11                  chain: FilterChain?
12          ) {
13                  // Retrieve config beans
14                  servletContext = request!!.servletContext
15                  userService =
16                          WebApplicationContextUtils.getWebApplicationContext(servletContext)!!
17                                  .getBean(UserService::class.java)
18
19                  val bearerToken = (request as HttpServletRequest).getHeader("Authorization")
20                  logger.info("Start␣auth␣filter␣for␣${bearerToken}")
21                  if (bearerToken == null || !bearerToken.startsWith("Bearer")) {
22                          chain?.doFilter(request, response)
23                          return
24                  }
25
26                  val tokenInfo = userService.checkToken(bearerToken.substring(7))
27
28                  if (tokenInfo.active) {
29                          val authorities = tokenInfo.realm_access!!.roles!!.map {
30                                  if (it.startsWith("ROLE_")) {
31                                          SimpleGrantedAuthority(it)
32                                  } else {
33                                          SimpleGrantedAuthority("ROLE_${it}")
34                                  }
35                          }
36                          val userDetails = this.userDetails(tokenInfo, authorities)
```

```
37                      SecurityContextHolder.getContext().setAuthentication(
38                              UsernamePasswordAuthenticationToken(userDetails, "",
                                    userDetails.authorities)
39                      )
40              } else {
41                      logger.error("Unable to validate token ${bearerToken.substring(7)}")
42              }
43              chain?.doFilter(request, response)
44      }
45
46      fun userDetails(token: TokenInfoDTO, authorities: List<SimpleGrantedAuthority>):
            UserDetails = User(
47              token.username,
48              "",
49              true,
50              true,
51              true,
52              true,
53              authorities
54      )
55  }
```

Il package mette anche a disposizione per i servizi REST la configurazione standard del listato 3.7 che viene automaticamente iniettata nel microservizio. Questa definizione può essere disabilitata con la definizione della proprietà `security.rest.config.useprovided` impostata a `false` nel caso sia necessaria una personalizzazione della stessa.

Listato 3.7: Configurazione standard per gli accessi ai servizi REST

```
1   // custom entry point for a REST service, returns 401 UNAUTHORIZED
2   @Configuration
3   class RestAuthenticationEntryPoint: AuthenticationEntryPoint {
4           override fun commence(
5           request: HttpServletRequest?,
6           response: HttpServletResponse?,
7           authException: AuthenticationException?
8           ) {
9                   response?.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorized")
10          }
11  }
12
13  @Configuration
14  @ConditionalOnProperty(
15          name = arrayOf("security.rest.config.useprovided"),
16          havingValue = "true",
17          matchIfMissing = true
18  )
19  @EnableWebSecurity
20  @EnableMethodSecurity(prePostEnabled = true)
21  class SecurityConfig {
22          @Autowired
23          lateinit var restAuthenticationEntryPoint: RestAuthenticationEntryPoint
24
25          @Bean
26          @ConditionalOnMissingBean(CorsConfigurationSource::class)
27          fun corsConfigurationSource(): CorsConfigurationSource {
28                  val configuration = CorsConfiguration()
29                  configuration.allowedOrigins = mutableListOf("*")
30                  configuration.setAllowedMethods(mutableListOf("GET", "POST", "PUT", "PATCH",
                        "DELETE", "OPTIONS"))
31                  configuration.allowedHeaders =
32                          mutableListOf("authorization", "content-type", "x-auth-token")
33                  configuration.exposedHeaders = mutableListOf("x-auth-token")
34                  val source = UrlBasedCorsConfigurationSource()
35                  source.registerCorsConfiguration("/**", configuration)
36                  return source
37          }
38
39          @Bean
40          @ConditionalOnMissingBean(SecurityFilterChain::class)
41          @Throws(Exception::class)
42          fun securityFilterChain(http: HttpSecurity) : SecurityFilterChain {
```

```
43                http
44                        .cors() // by default uses corsConfigurationSource Bean
45                        .and()
46                        .httpBasic().disable()
47                        .csrf().disable()
48                        .exceptionHandling()
49                        .authenticationEntryPoint(restAuthenticationEntryPoint)
50                        .and()
51                        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
52                        .and()
53                        .authorizeHttpRequests()
54                        .anyRequest().authenticated()
55                        .and()
56                        .addFilterBefore(AuthFilter(), BasicAuthenticationFilter::class.java)
57                return http.build()
58        }
59
60  }
```

3.3.2 Processo batch di inizializzazione

Il batch di inizializzazione legge i dati del vecchio sistema (mascherati o in chiaro) e li converte nel database del progetto. In aggiunta a questi dati, sono state create delle risorse aggiuntive da importare nel nuovo database. In particolare, sono state create l'anagrafica dei medici, prelevata dalle maschere di inserimento del vecchio programma (vedi 2.1), la lista dei farmaci prelevati dal sito dell'AIFA, la definizione di un flusso diagnostico di prova (in attesa di verificare con i medici del reparto la corretta definizione di questa collezione) e gli schemi derivati dal processo di mascheramento dei dati descritto nel paragrafo 2.2.1.

È stato implementato con `Spring Batch`. Per ogni tabella definisce lo `Step` da eseguire con la lettura della risorsa salvata, la conversione nel `DTO` da utilizzare per la comunicazione con il package che gestisce l'accesso al database e la successiva scrittura sullo stesso. Il batch può essere eseguito sia effettuando la completa inizializzazione del database sia aggiornando i dati esistenti per permettere l'esecuzione del processo di aggiornamento e gestire il passaggio dal vecchio applicativo al nuovo senza soluzione di continuità.

La funzione di utilità `document_parser`, visibile nel listato 3.8, analizza il contenuto del documento letto e determina se è in grado di effettuare la registrazione del dato in automatico o se deve rimandare al chiamante un errore per decidere come e se convertire il dato.

Listato 3.8: Analisi contenuto documento

```
1   fun document_parser(schemaStoreDTO: SchemaStoreDTO, document: Document): Document {
2           var newdoc = Document()
3           val dateFormatter6chr = DateTimeFormatter.ofPattern("dd/MM/yy")
4           val dateFormatter8chr = DateTimeFormatter.ofPattern("dd/MM/yyyy")
5           val errors: MutableMap<String, Any> = mutableMapOf()
6           document.forEach { t, u ->
7                   val key = t.replace(".", "_")
8                   val ustring = u as String
9                   // Skip obsolete fields
10                  if (schemaStoreDTO.schema.properties!!.get(key) != null) {
11                          val type = schemaStoreDTO.schema.properties!!.get(key)!!.type
12                          val format = schemaStoreDTO.schema.properties!!.get(key)!!.format
13                          val value: Any? = when {
14                                  type.equals("number") -> try {
15                                          ustring.replace(",", ".").replace("[^0-9-.]".toRegex(),
                                                "").toDouble()
16                                  } catch (ex: Exception) {
17                                          errors.put(key, ustring)
18                                          null
19                                  }
20
21                                  type.equals("integer") -> try {
22                                          ustring.replace("^0-9-]".toRegex(), "").toLong()
23                                  } catch (ex: Exception) {
24                                          errors.put(key, ustring)
```

```
25                                      null
26                                  }
27
28                                  type.equals("array") -> u
29                                  type.equals("boolean") -> try {
30                                      ustring.toBoolean()
31                                  } catch (ex: Exception) {
32                                      errors.put(key, ustring)
33                                      null
34                                  }
35
36                                  type.equals("string") &&
37                                  format?.equals("date") == true &&
38                                  ustring.length == 8 -> try {
39                                      LocalDate.parse(ustring, dateFormatter6chr)
40                                  } catch (ex: Exception) {
41                                      errors.put(key, ustring)
42                                      null
43                                  }
44
45                                  type.equals("string") &&
46                                  format?.equals("date") == true &&
47                                  ustring.length == 10 -> try {
48                                      LocalDate.parse(ustring, dateFormatter8chr)
49                                  } catch (ex: Exception) {
50                                      errors.put(key, ustring)
51                                      null
52                                  }
53
54                                  else -> u
55                          }
56                          value?.let { newdoc.append(key, it) }
57                  }
58          }
59          if (errors.size > 0) {
60                  newdoc.append("init_errors", errors)
61          }
62          return newdoc
63  }
```

L'elaborazione registra sulla collezione `logs` le anomalie riscontrate in fase di acquisizione dati. Nel log viene descritto se i dati sono stati scartati e per quale motivo o se il dato viene forzato (ad esempio se in un campo per il quale ci si aspetta un dato numerico intero viene trovato un valore con i decimali, questi ultimi vengono troncati). In alcuni casi sono scartati dei singoli valori all'interno del documento perché non compatibili con il tipo di dato atteso (alfanumerici contro numerici, ecc.) ma viene conservato il documento complessivo. In altri, possono venire scartati i record interamente in quanto l'anomalia non è risolvibile.

Ad esempio può capitare che l'anagrafica pazienti contenga due chiavi uguali o che la chiave anagrafica presente nelle tabelle relative agli esami, alle visite, ecc., non sia stata trovata. Un primo tentativo di estrazione e offuscamento dei dati e successiva inizializzazione del db (effettuato dal processo batch descritto nel paragrafo 3.3.2) ha evidenziato una anomalia importante: circa il 35% dei record analizzati contengono una chiave non presente nell'anagrafica rendendo impossibile abbinare correttamente l'informazione al paziente a cui si riferisce. Infatti, il vecchio sistema registra il codice della chiave con formati diversi, troncando in alcuni casi gli zeri non significativi del codice numerico, in altri lasciandone un numero arbitrario. Verificata l'anomalia, è stata creata la variabile `hypermacondoidzerodropped` di tipo stringa, visibile nel listato 3.1, che contiene il valore numerico della chiave senza zeri non significativi utilizzata per la ricerca dell'anagrafica paziente nei casi descritti. In questo modo le anomalie sono state quasi interamente risolte. Permangono solamente undici record riferiti a dati clinici non abbinabili alle anagrafiche dei pazienti, errore che si può ritenere trascurabile.

3.3.3 Gateway

Il servizio implementato con il progetto `Spring Cloud Gateway` è l'unico esposto per le chiamate provenienti dall'esterno ed ha il compito di smistare le richieste. La scelta di implementare un

servizio di gateway è dettata dalla volontà di fornire una soluzione completa senza affidare la messa in produzione a implementazioni terze, come già illustrato nella descrizione dell'architettura. La definizione delle route, che segue le specifiche di `Spring Cloud Gateway` [7], avviene definendo nel database di `Vault` utilizzato dal microservizio di configurazione una chiave denominata `gateway` che contiene i percorsi definiti per il routing e che permette l'aggiornamento "a caldo" delle regole di smistamento.

Un secondo motivo che giustifica l'adozione di questo strumento è che attualmente il servizio ha il solo scopo di fornire il routing verso gli altri microservizi ma potrà essere implementato per intercettare tutte le richieste e tutte le risposte per fornire funzionalità di auditing o per avviare delle comunicazioni con i client attraverso `Websocket` in modo da implementare un sistema in grado di aggiornare le viste in tempo reale e garantire una migliore esperienza utente.

3.3.4 Servizio di configurazione

Il servizio di configurazione interfacciato da tutti i moduli si affida a `Vault` per reperire le proprietà utilizzate dai servizi per auto configurarsi. Il progetto contiene uno script della shell bash che elabora tutti i file di proprietà contenuti nella directory `vault/conf` denominati in modo da permettere di automatizzare il caricamento delle configurazioni. I nomi dei file hanno infatti la seguente struttura: `<nomeconfigurazione>.<profilo>.json`. Il servizio è comunque accessibile tramite interfaccia web permettendo anche la manutenzione dei file di configurazione direttamente (vedi figura 3.2).

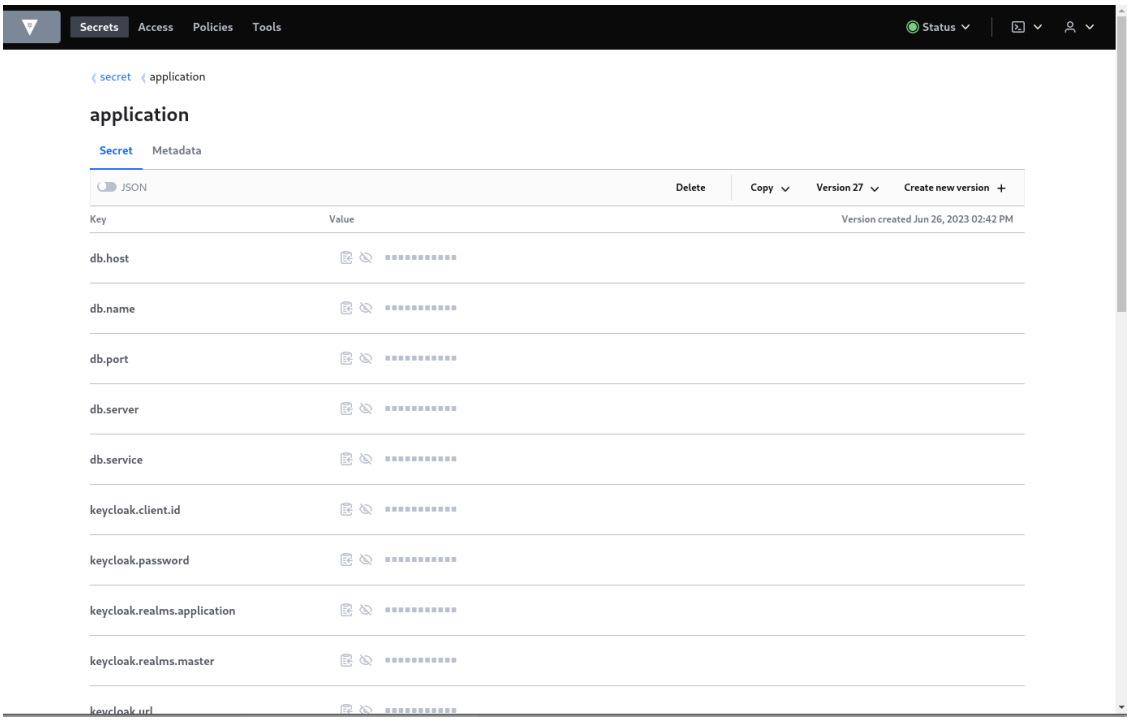


Figura 3.2: Lista proprietà della configurazione di default dell'applicazione

3.3.5 Servizio di autenticazione

Il prototipo viene rilasciato con un modulo per la gestione degli accessi basato su `Keycloak`.

> Keycloak is an open source identity and access management solution for our services and applications. The main objective of Keycloak is to facilitate the protection of the services and applications with little or no code. Some characteristics of Keycloak include the following:

- It centralizes authentication and enables single sign-on (SSO) authentication.
- It allows developers to focus on business functionality instead of worrying about security aspects like authorization and authentication.
- It allows two-factor authentication.
- It is LDAP compliant.
- It offers several adapters to secure applications and servers easily.
- It lets you customize password policies.[7]

L'utilizzo di `Keycloak` permette di integrare il progetto con uno strumento che garantisce i requisiti di sicurezza richiesti dalle norme vigenti. Inoltre, la sua sostituzione con un servizio che rispetti i protocolli previsti da `OAuth2` non richiede nessun intervento ai microservizi client. `OAuth2` è un framework di sicurezza al quale può essere richiesta una delega attraverso la generazione di un token autorizzativo, senza che il servizio che protegge la risorsa debba gestire l'autenticazione dell'utente.

La creazione del container per poter avviare il processo in produzione richiede che le richieste siano protette da una connessione sicura. Di conseguenza è necessario fornire un certificato valido accettato da tutti i servizi per permettere la comunicazione con il server di autenticazione. Per la fase di test è stato creato un comando che genera un certificato auto verificato con il comando `mkcert` per un server denominato `ccinfo.oauth`. Per agevolare tutti i processi di manutenzione del servizio è possibile risolvere il dominio modificando l'elenco degli host riconosciuti dalla tabella di routing della macchina ospite[19]. In questo modo il certificato creato corrisponde al common name utilizzato per crearlo. Se si intendesse utilizzare questo sistema di autenticazione in produzione potrà eventualmente essere utilizzato il certificato rilasciato per il dominio dell'ospedale. Il certificato creato:

- viene registrato sulla macchina ospite in modo da poter avviare l'interfaccia web descritta successivamente in questo paragrafo per la configurazione del servizio;
- viene copiato nella immagine del servizio di autenticazione e importato con il comando `keytool` all'atto della creazione dell'immagine stessa;
- viene importata la parte pubblica del certificato con il comando `keytool` in tutti i servizi che devono accettare come fidato il servizio di autenticazione (compreso il container con il processo batch di inizializzazione).

Il processo batch di inizializzazione interagisce con il server `Keycloak` tramite il package `it.polito.links.ccinfo.data` per inserire i ruoli `ADMIN`, `PHYSICIAN` e `USER`, gli utenti estratti dall'anagrafica di `HyperMacondo` e creare il realm `ccinfo`. Successivamente è necessario eseguire alcuni passi di configurazione per poter avviare il servizio e rendere operativo il sistema.

Se sono stati eseguiti i passi precedentemente descritti, si può accedere alla maschera di accesso illustrata in figura 3.3 digitando l'indirizzo `https://ccinfo.oauth:8443/admin/` nel browser. Lo username e la password devono corrispondere alle variabili d'ambiente `KEYCLOAK_ADMIN` e `KEYCLOAK_ADMIN_PASSWORD` che devono essere impostate all'avvio del container[20]. Una volta eseguita l'autenticazione occorre selezionare nel menu a tendina in alto a sinistra di figura 3.4 il realm `ccinfo` e la voce `Realm settings` nell'elenco posto a sinistra. Nella finestra visualizzata in figura 3.5 deve essere abilitato il realm `ccinfo` ovvero deve essere evidenziata in alto a destra la scelta `Enabled`[21]. Una volta abilitato il realm selezionare la voce `Clients` che presenta l'elenco dei client già visti in figura 3.4. Tramite il bottone `Create client` si deve avviare il processo di creazione

[19] Ad esempio sulla macchina utilizzata per il test, equipaggiata con il sistema operativo `Fedora 36`, è stato modificato il file `/etc/hosts` aggiungendo un elemento per la tabella di routing che risolve il nome di dominio `ccinfo.oauth` con l'indirizzo IP della macchina virtuale che esegue il container.

[20] Per il test sono state impostate semplicemente come `admin` entrambe.

[21] Nella figura il realm è disabilitato e appare la scritta `Disabled`.

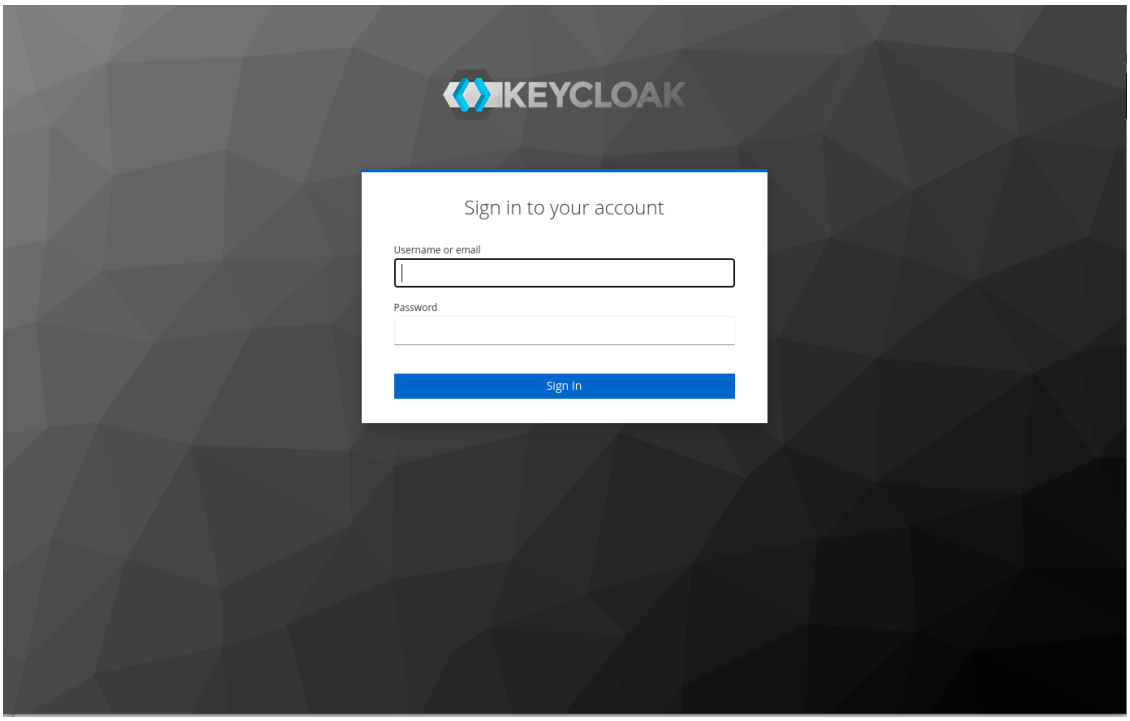


Figura 3.3: Accesso al server Keycloak

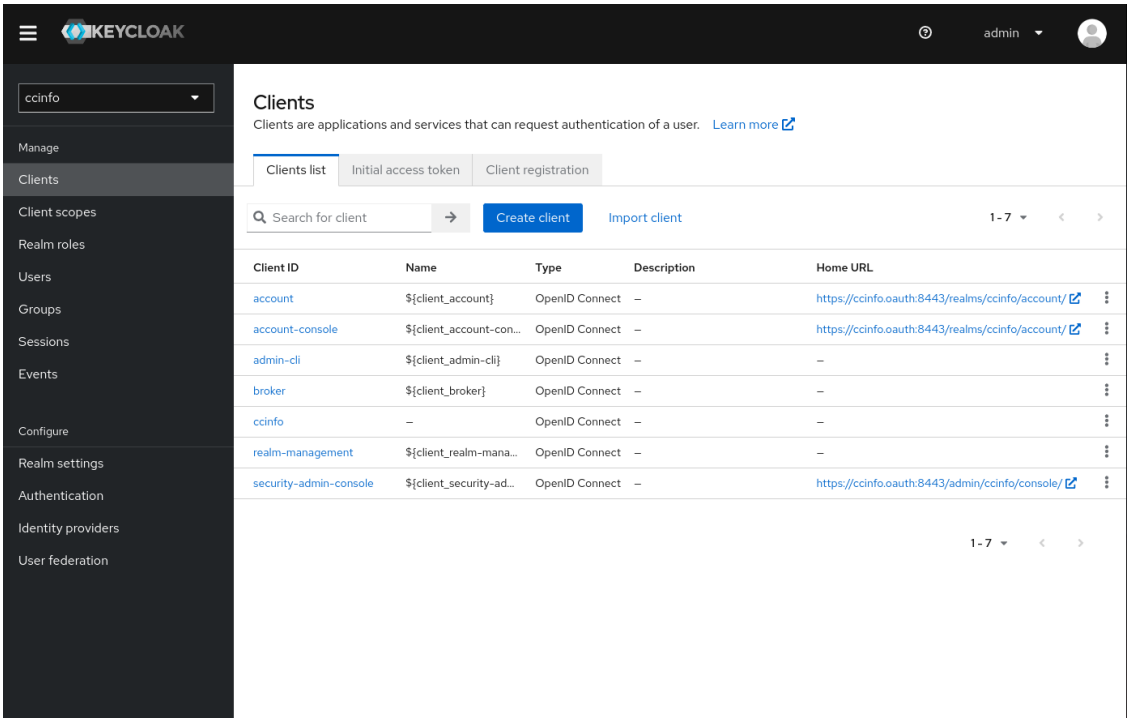


Figura 3.4: Lista dei client del realm ccinfo

del client che rappresenta l'utenza che verrà utilizzata dai microservizi che devono interagire con il sistema di autenticazione. Nella prima pagina di creazione del client, visibile in figura 3.6 deve essere indicato il `Client ID` `ccinfo` di tipo `OpenID Connect` per poi proseguire con il tasto `Next` alla pagina successiva (vedi figura 3.7). In questa seconda pagina vanno selezionate le opzioni `Client authentication` e `Authorization`. Infine, dopo aver premuto `Next` si accede alla terza e ultima pagina di configurazione del client, visibile in figura 3.8, nella quale vanno indicati i valori

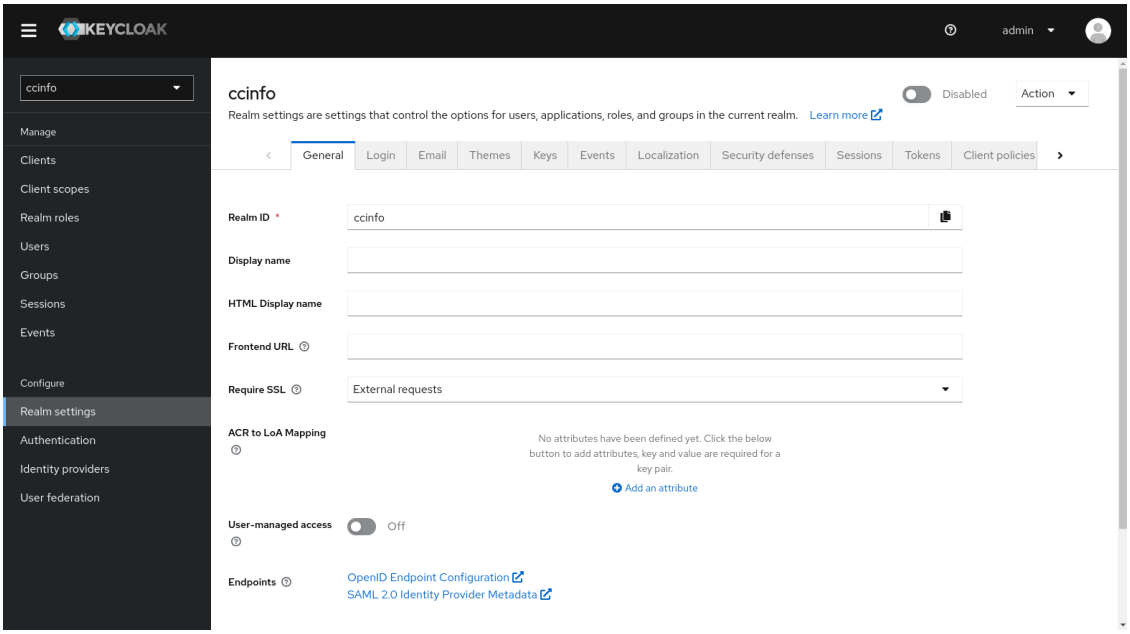


Figura 3.5: Abilitazione del realm ccinfo

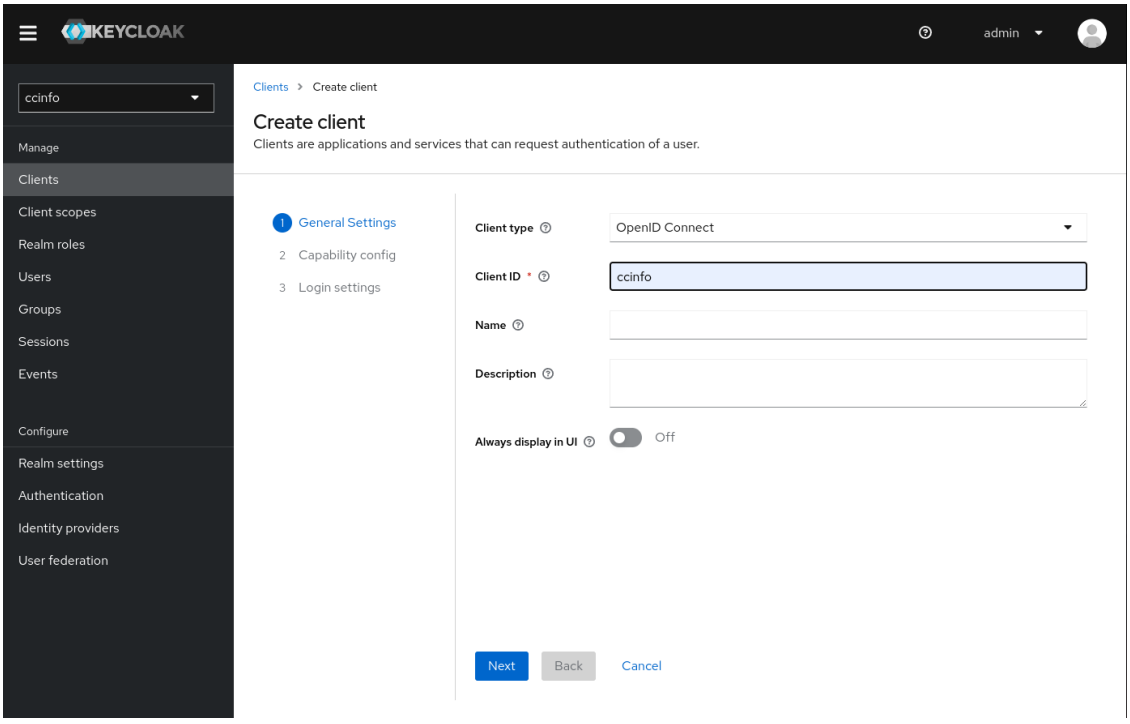


Figura 3.6: Creazione client ccinfo nel realm ccinfo - pag. 1

`https://ccinfo.oauth:8*` nel campo `Valid redirect URIs` e `*` nel campo `Web origins`.

Dopo il salvataggio del client `ccinfo` bisogna recuperare dallo stesso le credenziali che verranno utilizzate dai client per accedere al servizio. Il tab `Credentials` visibile nella figura 3.9 assolve a questo scopo. Le credenziali copiate vanno inserite nella proprietà `oauth.client.pwd` dello storage `Vault` nel path `application` in modo che tutti i servizi vengano aggiornati con le credenziali inizializzate dal server `Keycloak`.

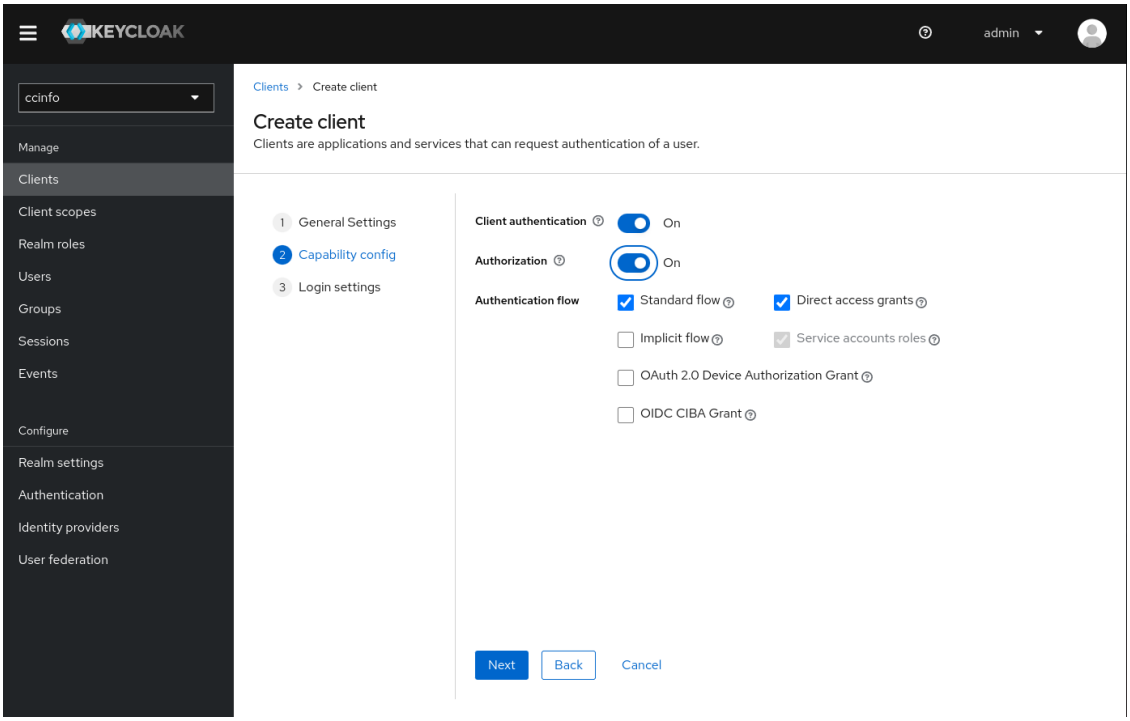


Figura 3.7: Creazione client ccinfo nel realm ccinfo - pag. 2

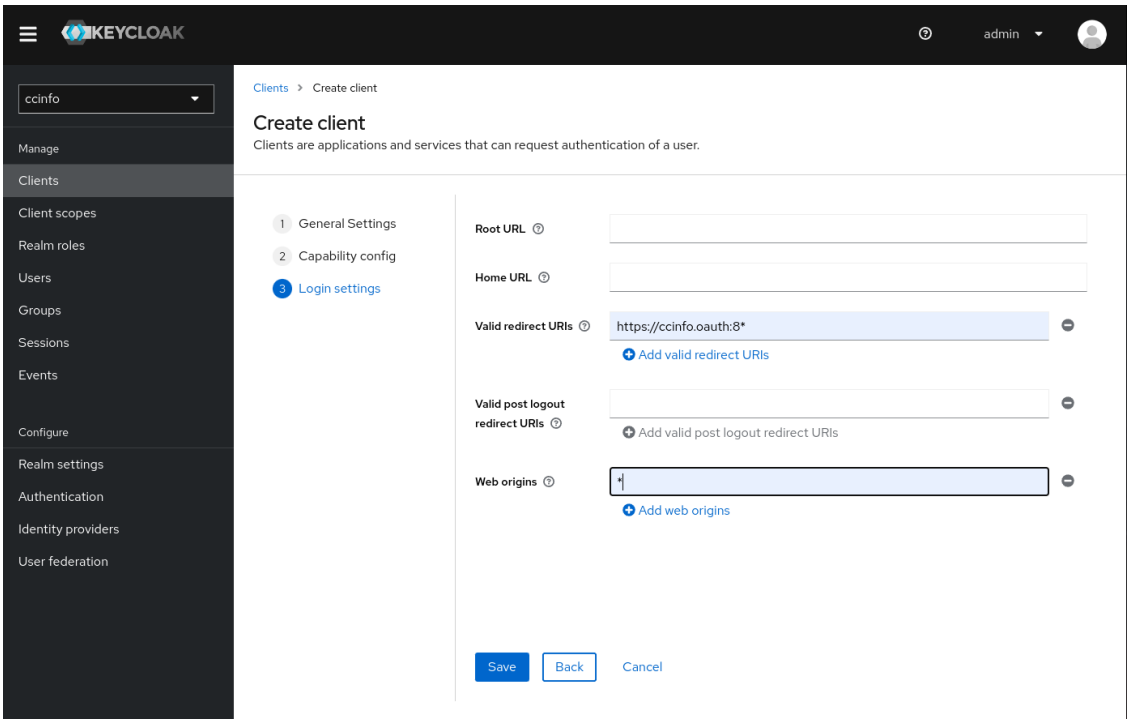


Figura 3.8: Creazione client ccinfo nel realm ccinfo - pag. 3

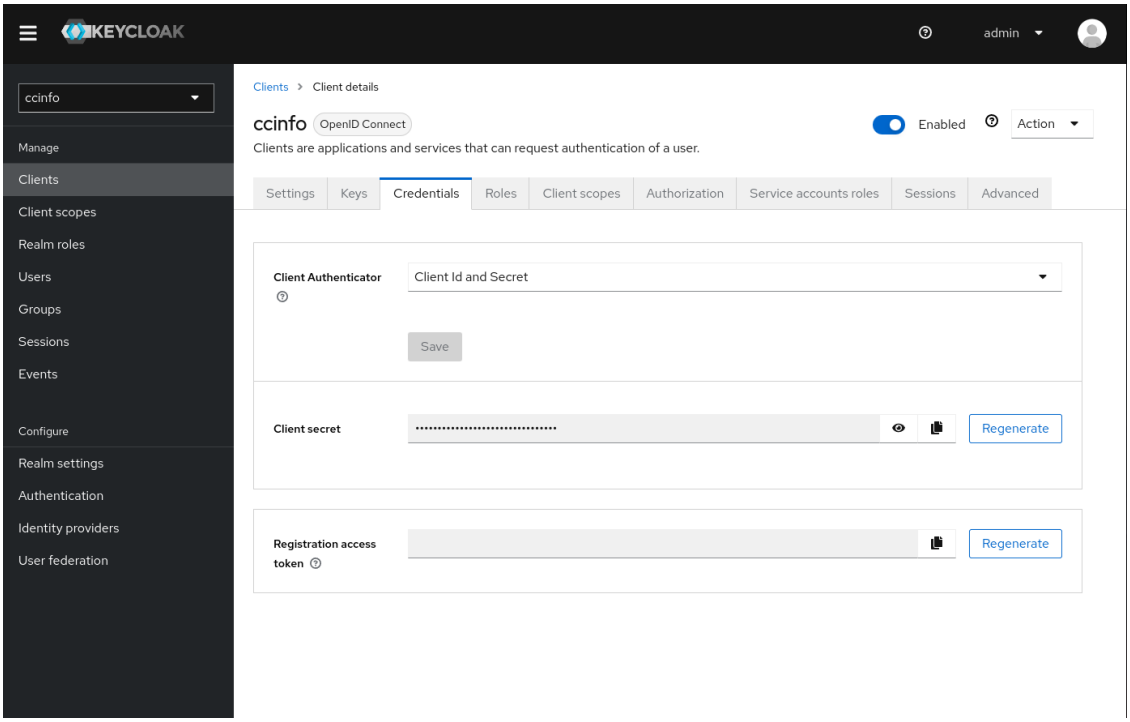


Figura 3.9: Recupero credenziali client ccinfo

3.3.6 Servizi REST

In questo paragrafo vengono presentati i servizi che implementano gli endpoint di accesso all'applicazione. Le sezioni relative ai singoli microservizi presentano una tabella con l'elenco delle API esposte con una breve descrizione per il loro utilizzo. Le colonne delle tabelle riportano:

1. l'url *relativo* che identifica la risorsa. Tutti gli endpoint dei servizi REST vengono esposti con il prefisso `/api` per evitare eventuali conflitti con i `path` utilizzati dal frontend[22];
2. il metodo HTTP da utilizzare per accedere alla risorsa e identificare il tipo di operazione che si intende effettuare sulla stessa;
3. una breve descrizione della funzione;
4. il payload, i parametri della richiesta o alcune note relative ai vincoli autorizzativi.

Le risorse relative ai microservizi di esportazione dati e gestione dei report sono indicativi e potrebbero subire modifiche in relazione all'analisi che verrà eseguita dopo la presentazione del prototipo.

Single Sign On

Endpoint	Metodo	Descrizione	Payload
/sso/login	POST	Richiesta login utente	Oggetto con username e password
/sso/refresh	POST	Richiesta aggiornamento token autorizzativo	Il refresh_token restituito dalla funzione di login
/sso/logout	GET	Richiesta disconnessione dal sistema	Oggetto con l'access_token ed il refresh_token

Tabella 3.2: Elenco delle API REST del microservizio Single Sign On

Il microservizio espone le API per la gestione dell'autenticazione. La richiesta di login richiede il passaggio di un oggetto contenente lo username e la password dell'utente e restituisce l'oggetto visibile nel listato 3.9. Il campo `access_token` deve essere inserito come intestazione `Authorization`[23] in tutte le chiamate seguenti al sistema e rappresenta l'identificazione dell'utente.

Listato 3.9: Login View

```
1  data class TokenView(
2          val access_token: String,
3          val expires_in: Int,
4          val refresh_expires_in: Int,
5          val refresh_token: String,
6          val token_type: String,
7          val not_before_policy: Int,
8          val session_state: String,
9          val scope: String,
10         val user_id: String? = null,
11         val username: String? = null,
12         val roles: List<String>? = mutableListOf()
13 )
```

Il campo `refresh_token` viene restituito dal sistema per poter garantire una maggiore durata delle sessioni e può essere utilizzato dal frontend per rinnovare il token autorizzativo. Infatti, richiamando l'endpoint `/sso/refresh` passando come payload questo token, verrà restituito un nuovo oggetto `LoginView` con il token di accesso rinnovato. È importante notare che il `refresh_token`

[22] Ovviamente il frontend è vincolato a non utilizzare percorsi che inizino con il prefisso `/api`.

[23] L'access_token inserito deve essere preceduto dalla stringa `Bearer`.

ha il solo scopo di favorire il rinnovo della sessione utente e non può essere utilizzato come `Bearer` token nel campo intestazione `Authorization`.

Infine, alla fine della sessione può essere richiamata la funzione di logout che comunica al server `Keycloak` la disconnessione dell'utente.

Gestione utenti

Endpoint	Metodo	Descrizione	Payload/Parametri richiesta
/users	GET	Lista utenti	Parametri (opzionali): searchstring e maxsize (default 25)
/users	POST	Aggiunge un nuovo utente. Accessibile solo ad utenti ADMIN.	Payload: UserView
/users/{id}	GET	Preleva l'utente con ID=id. Accessibile solo ad utenti ADMIN o alla propria anagrafica.	Parametro: Id utente
/users/{id}	PUT	Aggiorna l'utente con ID=id. Accessibile solo ad utenti ADMIN o alla propria anagrafica (in questo caso solo per un insieme limitato di campi).	Parametro: Id utente. Payload: UserView

Tabella 3.3: Elenco delle API REST esposte dal microservizio gestione utenti

Il servizio interagisce con il server `Keycloak` per la gestione degli utenti. L'API è illustrata nel listato 3.10. Tutti gli endpoint sono protetti e accessibili solo agli amministratori o ai legittimi proprietari dei dati, fatta eccezione per il reperimento della lista degli utenti che il frontend può utilizzare per reperire le informazioni sui medici. Attraverso questo endpoint anche gli utenti generici[24] adibiti all'inserimento per conto dei medici dei dati clinici dei pazienti possono recuperare i riferimenti per completare l'informazione da registrare sul database. I parametri di ricerca per la `GET` del path `/users` sono opzionali e servono per selezionare un sottoinsieme dei dati presenti nel DB. Quando viene impostato il parametro `searchstring`, la ricerca viene effettuata sul nome e cognome dell'utente ignorando eventuali caratteri maiuscoli[25]. L'ID utente da inserire nel path per il reperimento e la modifica del singolo utente corrisponde alla chiave generata dal servizio `Keycloak`.

Listato 3.10: User View

```
1  data class UserView(
2          var id: String? = null,
3          val hypermacondoid: String? = null,
4          var firstname: String,
5          var lastname: String,
6          var title: String? = null,
7          var username: String,
8          var email: String,
9          var roles: MutableList<String>,
10         var phones: MutableList<String>? = null
11 )
```

Gestione schemi

Il servizio offre l'interfaccia per la gestione degli schemi. Le funzionalità legate alla manutenzione degli schemi sono ammesse solo per gli utenti amministratori, che saranno i deputati a variare il tipo di informazioni che potranno essere gestiti dal sistema. Il parametro di ricerca nel reperimento della lista degli schemi disponibili permettono di effettuare una selezione in base al nome dello schema. L'endpoint che reperisce il *validatore* restituisce uno schema unico con l'insieme di tutti gli schemi definiti e può essere utilizzato come schema di riferimento per la validazione di qualsiasi

[24] Per utente generico si intende chi possiede solo il ruolo di `USER`.

[25] La ricerca è case insensitive.

Endpoint	Metodo	Descrizione	Payload/Parametri richiesta
/schemas	GET	Lista schemi	Parametri (opzionali): schemaname
/schemas/bounds	GET	Numero di schemi presenti	
/schemas/validator	GET	Restituisce lo schema con tutti i tracciati ai fini della validazione	
/schemas/{id}	GET	Preleva lo schema con ID=id	Parametro: Id schema
/schemas	POST	Aggiunge un nuovo schema. Accessibile solo a utenti ADMIN	Payload: SchemaStoreView
/schemas/{id}	PUT	Aggiorna lo schema con ID=id. Accessibile solo ad utenti ADMIN	Parametro: Id schema Payload: SchemaStoreView
/schemas/{id}	DELETE	Cancella lo schema con ID=id. Accessibile solo ad utenti ADMIN	Parametro: Id schema

Tabella 3.4: Elenco delle API REST esposte dal microservizio gestione schemi

documento che si desidera sia conforme alle specifiche definite nel database. Il listato 3.11 presenta le caratteristiche dell'API di comunicazione per questo servizio.

Listato 3.11: SchemaStoreView

```
data class SchemaStoreView(
    var _id: String? = null,
        var schemaName: String,
        var description: String,
        var type: SchemaTypeDTO,
        var subtype: SchemaSubTypeDTO? = null,
        var schema: SchemaView,
        var patientDataCluster: String
)

data class SchemaView(
        // Common to all types
        @get:JsonProperty("\$schema", index = 0) val schema: String? = null,
        @get:JsonProperty("\$id", index = 1) val id: String? = null,
        @get:JsonProperty(index = 10) var type: String? = null, // object, string, number,
            integer, array, boolean, null
        @get:JsonProperty("enum") var enumProp: MutableList<Any>? = null, // valid also without
            type
        var title: String? = null,
        var description: String? = null,
        var default: Any? = null,
        var examples: MutableList<Any>? = null,
        var deprecated: Boolean? = null,
        var readOnly: Boolean? = null,
        var writeOnly: Boolean? = null,
        @get:JsonProperty("\$comment") var comment: String? = null,
        @get:JsonProperty("const") var constValue: Any? = null,
        @get:JsonProperty("\$defs", index = 999) var defs: MutableMap<String, SchemaView>? =
            null,
        @get:JsonProperty("\$ref", index = 20) var ref: String? = null,
        var definitions: MutableMap<String, SchemaView>? = null,
        @get:JsonProperty("\$anchor") var anchor: String? = null,

        // Type object
        @get:JsonProperty(index = 30) var properties: MutableMap<String, SchemaView>? = null,
        var minProperties: Int? = null,
        var maxProperties: Int? = null,
        var additionalProperty: Boolean? = null,
        @get:JsonProperty(index = 25) var required: MutableList<String>? = null,
        var unevaluatedProperties: Boolean? = null,

        // Type string
        var minLength: Int? = null,
        var maxLength: Int? = null,
        var pattern: String? = null,
        var format: String? = null,

        // Type number and integer
        var multipleOf: Int? = null,
        var minimum: Int? = null,
```

```
48          var exclusiveMinimum: Int? = null,
49          var maximum: Int? = null,
50          var exclusiveMaximum: Int? = null,
51
52          // Type array
53          var items: MutableMap<String, String>? = null,
54          var prefixItems: MutableList<SchemaView>? = null,
55          var contains: SchemaView? = null,
56          var minContains: Int? = null,
57          var maxContains: Int? = null,
58          var minItems: Int? = null,
59          var maxItems: Int? = null,
60          var uniqueItems: Boolean? = null,
61
62          // Schema composition
63          var allOf: MutableList<SchemaView>? = null,
64          var anyOf: MutableList<SchemaView>? = null,
65          var oneOf: MutableList<SchemaView>? = null,
66          @get:JsonProperty("not") var notOf: SchemaView? = null,
67
68          // Conditionally
69          var dependentRequired: MutableMap<String, List<String>>? = null,
70          var dependentSchemas: MutableMap<String, SchemaView>? = null,
71          @get:JsonProperty("if") var ifClause: SchemaView? = null,
72          @get:JsonProperty("then") var thenClause: SchemaView? = null,
73          @get:JsonProperty("else") var elseClause: SchemaView? = null
74  )
```

Gestione pazienti

Endpoint	Metodo	Descrizione	Payload Parametri richiesta
/patients	GET	Lista pazienti	Parametri (opz.): pagesize, pagenumber, search
/patients	POST	Aggiunge un nuovo paziente	Payload: PatientView
/patients/{id}	GET	Recupera i dati personali del paziente con ID=id	Parametro: Id paziente
/patients/{id}	PATCH	Aggiorna i dati personali del paziente con ID=id	Parametro: Id paziente Payload: PatientView
/patients/{id}/clinicdata	GET	Recupera i dati clinici del paziente con ID=id	Parametro: Id paziente
/patients/{id}/clinicdata	POST	Aggiunge dati clinici al paziente con ID=id	Parametro: Id paziente Payload: CinicDataView
/patients/{id}/clinicdata/{cluster}	GET	Recupera i dati clinici dell'insieme cluster del paziente con ID=id	Parametri: Id paziente Nome cluster
/patients/{id}/clinicdata/{cluster}/{ccid}	GET	Recupera i dati clinici dell'insieme cluster con ID=ccid del paziente con ID=id	Parametri: Id paziente Nome cluster Id dati clinici
/patients/{id}/clinicdata/{cluster}/{ccid}	PATCH	Recupera i dati clinici dell'insieme cluster con ID=ccid del paziente con ID=id	Parametri: Id paziente Nome cluster Id dati clinici Payload: CinicDataView

Tabella 3.5: Elenco delle API REST esposte dal microservizio gestione pazienti

Questo modulo presenta gli endpoint per la gestione dei pazienti. Ogni paziente è rappresentato da un documento nel DB descritto nel listato 3.1. Le risorse rispecchiano la suddivisione dei dati in cluster e possono essere divise in due gruppi.

Il primo mette a disposizioni le funzionalità per gestire i dati anagrafici del paziente. L'API utilizzata è la `PatientView` che contiene un documento con i dati del paziente che deve essere conforme allo schema `anagrafica` e una lista opzionale con i dati clinici. Questa lista viene ignorata nelle operazioni di aggiornamento dei dati (e quindi non deve essere passata), ma è utile quando il frontend necessita di caricare la scheda del paziente permettendo, con una sola chiamata al backend, di ottenere tutti i dati da presentare all'utente. Il secondo, caratterizzato dalle risorse con path relativo `/clinicdata`, si occupa dei dati clinici del paziente e utilizza l'API `ClinicDataView` per il payload nelle funzioni di aggiornamento, mentre risponde sempre con lo stesso formato `PatientView`. Attraverso queste risorse è possibile reperire tutti i dati clinici del paziente o effettuare una selezione di un cluster di dati o anche di un singolo elemento. Come si può osservare nel listato 3.1, i dati clinici sono caratterizzati da un codice univoco (`_id`) che viene utilizzato per identificare il documento all'interno dello stesso cluster ed è denominato `ccid` nella tabella 3.5. Gli schemi delle due API sono visibili nel listato 3.12.

Listato 3.12: Patient View

```
data class PatientView(
        var _id: String?,
        var hypermacondoid: String? = null,
        var personalData: org.bson.Document,
        var clinicRecords: MutableMap<String, MutableList<ClinicDataView>>?
)

data class ClinicDataView(
        var _id: String? = null,
        var schemaRef: String,
        var schemaName: String,
        var schemaType: SchemaTypeDTO,
        var schemaSubType: SchemaSubTypeDTO?,
        var date: LocalDate,
        var content: Document
)
```

Gestione farmaci

Endpoint	Metodo	Descrizione	Payload/Parametri richiesta
/drugs	GET	Recupera la lista dei farmaci	Parametri (opz.): pagesize, search
/drugs	POST	Aggiunge un nuovo farmaco. Accessibile solo ad utenti ADMIN.	Payload: DrugView
/drugs/{id}	GET	Recupera farmaco con ID=id	Parametro: Id farmaco
/drugs/{id}	PUT	Aggiorna farmaco con ID=id. Accessibile solo ad utenti ADMIN.	Parametro: Id farmaco Payload: DrugView
/drugs/{id}	DELETE	Elimina il farmaco con ID=id. Accessibile solo ad utenti ADMIN.	Parametro: Id farmaco

Tabella 3.6: Elenco delle API REST esposte dal microservizio gestione farmaci

Come illustrato nel paragrafo 3.2.1, la lista dei farmaci è stata caricata dal sito dell'AIFA e periodicamente potrà essere aggiornata il medesimo processo di recupero dei dati. Il sistema mette comunque a disposizione gli endpoint per procedere con la manutenzione dell'archivio. Il processo di aggiornamento[26] potrà fare uso delle risorse messe a disposizione da questo microservizio per effettuare l'allineamento dei dati. Il parametro `search` a disposizione per il recupero della lista dei farmaci permette di effettuare una ricerca combinata su tre elementi legati al farmaco: il nome chimico (il principio attivo), la descrizione ed il nome. La ricerca avviene senza tenere conto della differenza tra maiuscole e minuscole. L'API da utilizzare per l'aggiornamento dei dati è presentata nel listato 3.13.

Listato 3.13: Drug View

[26] Il prototipo dell'applicazione non contiene ancora le funzionalità necessaria per allineare l'archivio dei farmaci a quanto pubblicato sul sito dell'AIFA.

```
data class DrugView(
        var _id: String? = null,
        var chemicalName: String,
        var groupDescription: String,
        var name: String,
        var price: String,
        var ownerAIC: String,
        var AICcode: String,
        var equivalentGroup: String,
        var inAIFATransparencyList: String? = null,
        var inRegionListOnly: String? = null,
        var oxygenM3: String? = null
)
```

Gestione flussi diagnostici

Endpoint	Metodo	Descrizione	Payload/Parametri richiesta
/flows	GET	Lista dei flussi diagnostici	
/flows	POST	Aggiunge un flusso diagnostico. Accessibile solo ad utenti ADMIN.	Payload: FlowView
/flows/{id}	GET	Recupera il flusso diagnostico con ID=id	Parametro: Id flusso
/flows/{id}	PUT	Aggiorna il flusso diagnostico con ID=id. Accessibile solo ad utenti ADMIN.	Parametro: Id flusso Payload: FlowView
/flows/{id}	DELETE	Cancella il flusso diagnostico con ID=id. Accessibile solo ad utenti ADMIN.	Parametro: Id flusso

Tabella 3.7: Elenco delle API REST esposte dal microservizio gestione flussi diagnostici

Il microservizio di gestione dei flussi diagnostici mette a disposizione le risorse indicate in tabella 3.7. Le funzionalità di aggiornamento sono a disposizione solo per gli utenti amministratori. Il formato dei flussi è una struttura a grafo e l'API è presentata nel listato 3.14.

Listato 3.14: Flow View

```
data class FlowView(
        var _id: String? = null,
        var nodeViews: MutableList<NodeView>,
        var name: String,
        var connectionViews: MutableList<ConnectionView>
)

data class NodeView(
        var title: String,
        var id: Int,
        var x: Int,
        var y: Int,
        var type: NodeTypeView,
        var state: NodeStateView,
        var schemaId: Int,
        var prevId: Int,
        var next: MutableList<NextNodeView>? = null
)

enum class NodeTypeView {
        START,
        OPERATION,
        END
}

enum class NodeStateView {
        PERFORMED,
        PERFORMING,
        IGNORED
}

data class NextNodeView(
        var transitionName: String,
        var nextId: Int
)
```

```
36
37  data class ConnectionView(
38          var source: EdgeView,
39          var destination: EdgeView,
40          var type: ConnectionTypeView,
41          var title: String?
42  )
43
44  enum class ConnectionTypeView {
45          SUCCESS,
46          FAIL
47  }
48
49  data class EdgeView(
50          var id: Int,
51          var position: String
52  )
```

Gestione report ed esportazione dati

Endpoint	Metodo	Descrizione
/export	GET	Esportazione dati
/reports	GET	Creazione report
/reports	POST	Genera il report nel formato definitivo o in bozza per un determinato paziente
/reports/{id}	PUT	Aggiorna il report con ID=id

Tabella 3.8: Elenco delle API REST esposte dai microservizi report ed esportazione dati

Come più volte indicato, le funzionalità per la produzione dei referti e l'esportazione dei dati non sono ancora stati analizzati. La creazione dei microservizi permette di avere una minima infrastruttura utilizzabile per future implementazioni. Gli endpoint per entrambi i servizi sollevano l'eccezione `kotlin.NotImplementedError` che, non essendo mappata in alcun `Advice`, restituisce un errore con lo stato HTTP 500.

3.4 Installazione applicazione

L'applicazione è composta da un insieme di microservizi sviluppati con il linguaggio `Kotlin` utilizzando il framework `Spring`. Ogni microservizio viene virtualizzato in un container docker. L'applicazione espone una serie di servizi `REST` accessibili tramite la rete internet (o una rete intranet) configurabile tramite l'impostazione di alcuni parametri all'avvio del sistema. Solo un servizio (il gateway) è effettivamente esposto nei confronti del mondo esterno ed esso fornisce le funzionalità di routing necessarie per raggiungere gli altri microservizi interni al sistema in base alla richiesta effettuata dall'utente.

In aggiunta ai servizi sviluppati nell'ambito di questo progetto, vengono utilizzati quattro ulteriori container, accessibili sul repository di `Docker`, che forniscono alcune funzionalità di supporto: una istanza di `Keycloak`[27] che implementa le funzionalità del servizio di autenticazione e che utilizza una istanza di `PostgreSQL`[28] come storage delle informazioni degli utenti, tre istanze (necessarie per la gestione delle transazioni) di `MongoDB`[29], il data base utilizzato per archiviare le

[27] vedi https://www.keycloak.org/. L'immagine utilizzata

[28] https://www.postgresql.org/

[29] L'installazione di MongoDB con una strategia di replica, oltre ad essere una configurazione raccomandata per l'installazione in produzione, è necessaria per gestire le transazioni. Nella versione beta del software sviluppato in questa tesi, tutte le repliche vengono avviate sulla stessa macchina. Nelle conclusioni viene discussa l'eventualità di implementare una versione che preveda il deploy su più macchine per garantire una maggiore sicurezza.

informazioni, ed un server `Vault`[30] utilizzato come storage dei parametri di configurazione esposti alle applicazioni tramite il microservizio di configurazione[31].

Per installare ed eseguire l'applicazione esistono due modalità:

1. l'esecuzione delle immagini virtualizzate con un motore compatibile con il formato dei container `Docker`;
2. l'utilizzo di `Kubernetes`.

In entrambi i casi è necessario utilizzare una macchina con sistema operativo `*NIX like` in grado di interpretare la shell `bash`[32] che sia accessibile agli utenti via rete.

3.4.1 Docker

Per utilizzare il programma in questa modalità è necessario installare i comandi `docker` e `docker-compose` tramite le distribuzioni di software relative al sistema operativo scelto o utilizzare una macchina virtuale (ad esempio sul cloud) che abbia questi programmi installati. L'applicazione viene fornita con uno script di lancio che esegue su richiesta la compilazione dei sorgenti utilizzando il tool `Gradle`, genera le immagini dei container e installa il sistema tramite l'utilizzo del comando `docker-compose` che deve essere installato sulla macchina ospite. In alternativa, lo script può essere eseguito senza effettuare la compilazione dei sorgenti ma accedendo direttamente al registro `docker.io` per prelevare le immagini già create e pubblicate in precedenza. Nella cartella principale del progetto sono presenti due file di configurazione ed una cartella con un elenco di proprietà:

1. il file `.env` che contiene la definizione di alcune variabili d'ambiente utili per avviare il contenitore dei file di configurazione;
2. il file `docker-compose.yaml` che contiene la definizione dell'architettura del sistema ed i contesti da utilizzare per la creazione delle immagini;
3. la cartella `vault/conf` che contiene un insieme di file di proprietà personalizzabili per avviare il sistema e per mettere in collegamento tutti i microservizi installati.

Dopo aver eventualmente personalizzato i file di configurazione, l'applicazione può essere lanciata tramite il comando `runtestenv.sh`. Passando l'opzione `--help` il comando non effettua alcuna elaborazione me presenta le opzioni disponibili per l'esecuzione del comando. Il sistema può essere utilizzato con tutti i sistemi compatibili con il formato dei container docker (ad esempio podman).

3.4.2 Kubernetes

In questo caso è necessario installare o utilizzare una macchina virtuale che abbia una implementazione di `Kubernetes` installata. `Kubernetes` non è una applicazione ma una specifica ed esistono diverse implementazioni, gratuite o a pagamento, che la implementano. Per effettuare i test ho utilizzato un implementazione leggera prodotta dalla `Rancher`, `K3S` (gratuita), che ha le caratteristiche idonee per essere utilizzata anche in produzione.

Nella cartella `kubernetes` sono presenti tutti i file di configurazione utili per l'utilizzo dell'applicazione con `Kubernetes`. Lo script `runk8s.sh` applica tutti i file che definiscono l'applicazione

[30] https://www.hashicorp.com/products/vault

[31] Il microservizio di configurazione fa parte del progetto ed è sviluppato come implementazione di uno Spring Config Server

[32] I comandi utilizzati per eseguire le funzioni di compilazione ed esecuzione dei microservizi sono sviluppati per la shell bash e testati su una macchina con sistema operativo Fedora, release 37.

nel corretto ordine: prima imposta i segreti, poi applica le configurazioni delle applicazioni ed infine installa i servizi. Per semplificare l'operatività, in particolare per sviluppi futuri, la politica di creazione ed aggiornamento dei pods utilizza le immagini presenti sul registro (`docker.io`). Pertanto, sarebbe preferibile avere accesso autenticato al registro `docker.io` per poter utilizzare le immagini nel caso si volessero mantenere queste ultime private e non accessibili alla comunità.

3.4.3 Requisiti per l'installazione

Per la scelta dell'hardware da utilizzare per eseguire l'applicazione sono state prese in considerazione due alternative: l'utilizzo di un servizio in cloud e l'installazione su hardware di proprietà dell'ospedale (`On-premise`). Oltre ad analizzare le caratteristiche per l'acquisto del servizio o dell'hardware e per la manutenzione dei sistemi, devono essere presi in considerazione alcuni requisiti necessari per la sicurezza e la tutela della privacy dei pazienti. Tra i requisiti di sicurezza e gli adempimenti richiesti dal GDPR per il trattamento dei dati sensibili[33] che interessano la Cartella Clinica Elettronica trovano particolare rilevanza al fine di determinare la corretta implementazione dell'applicativo e la scelta dell'hardware da utilizzare:

- protezione dei dati in transito;
- protezione dei dati a riposo;
- mantenimento dei dati nella Comunità Europea;
- protezione dei dati in tutte le fasi di vita, incluse quelle di progettazione ed implementazione. I dati devono essere protetti anche in caso di sottrazione dell'hardware;
- verifica continua dei principi di rispetto della privacy, il sistema deve permettere di verificare l'utenza che ha svolto determinate operazioni (auditing);
- confidenzialità e restrizioni nell'accesso ai dati e autenticazione a due fattori, solo gli utenti autorizzati devono poter accedere ai dati;
- il sistema deve esporre solo i servizi essenziali nei confronti dell'esterno;
- i dati devono essere sempre disponibili;
- i dati devono essere salvati e devono essere recuperabili in caso di necessità.

La tabella seguente riassume queste caratteristiche oltre a fornire alcune indicazioni sul tipo di installazione e manutenzione necessarie:

La Cartella Clinica Informatizzata deve garantire di essere accessibile all'interno della rete dell'ospedale Molinette ed eventualmente permettere l'accesso al sistema da reti esterne, in particolare per agevolare lo spostamento all'interno dell'ospedale del personale collegato con reti wireless e non direttamente con la rete ethernet e dare un giusto livello di disponibilità del servizio in modo da non interrompere la normale operatività del reparto. Inoltre, la comunicazione tra il client adoperato dal personale dell'ospedale e il server dove viene eseguita l'applicazione deve essere cifrata. Il servizio in Cloud garantisce queste funzionalità a patto di inserire un record DNS nei domini a disposizione dell'ospedale e di salvare un certificato SSL sull'hardware fornito dal Provider e fornisce una maggiore scalabilità dei servizi in quanto è semplice[34] aggiungere o rimodulare i servizi acquistati. Il salvataggio del certificato sui server del Provider potrebbe esporre ad un furto di identità per il dominio che si intende utilizzare. Nel caso, invece, On-premise il sistema risulta più sicuro se vengono adottate le opportune misure di protezione dell'hardware. Per poter accedere dall'esterno dell'ospedale è necessario fare in modo che il server sia raggiungibile o attraverso un port forwarding oppure con l'installazione di una Virtual Private Network

[33] tra i quali ovviamente troviamo i dati biometrici

[34] Ovviamente ogni aggiunta di risorse determina un aumento dei costi, mentre una diminuzione di risorse ne causa una diminuzione (nel caso si determinasse un minor utilizzo dell'applicativo)

	Cloud Provider	On-premise
Accessibilità dall'esterno	si	tramite port forwarding o VPN
Scalabilità	si	potrebbe richiedere ulteriori investimenti
Manutenzione sistema	si	a carico dell'ospedale
Protezione in transito	installazione certificato	si
Protezione a riposo	si, ma decifrabile da provider	cifratura disco
Localizzazione server EU	si	si
Protezione in tutte le fasi	da applicativo	da applicativo
Auditing	da applicativo	da applicativo
Confidenzialità	da applicativo	da applicativo
Backup	si	a carico dell'ospedale
Cifratura	si, ma chiave non privata	richiede un server con dischi cifrati
Esposizione minima	da applicativo	da applicativo
Garanzia raggiungibilità	si	richiede maggiori investimenti
Disaster recovery	si	a carico dell'ospedale

Tabella 3.9: Caratteristiche servizi *Cloud Provider* e *On-premise*

che permetta di comunicare con il server direttamente tramite la Intranet locale. L'applicativo eseguito tramite l'opzione `Kubernetes` permette di aumentare le istanze dei microservizi o i nodi del cluster a piacere ma potrebbe richiedere la necessità di acquistare nuovo hardware fornendo una minor scalabilità rispetto la soluzione in Cloud.

La protezione a riposo viene garantita dal Cloud Provider attraverso la cifratura dei dati tramite una chiave ad esso conosciuta mentre nel caso On-premise è necessario avere a disposizione una macchina con i dischi cifrati. Tuttavia, la soluzione attuale dell'applicativo non cifra i documenti che salva sul DB. Di conseguenza, l'amministratore del sistema che ha accesso ai dischi e che li può decifrare potrebbe estrarre i dati dal DB. Nel caso del servizio in Cloud potrebbe essere utile sviluppare una versione del software che preveda di salvare i dati cifrati, anche se, in questo caso, anche i client che accedono ai dati dovrebbero essere in grado di decifrare i dati che provengono dal servizio.

La protezione dei dati nelle fasi di sviluppo è stata garantita evitando di salvare in chiaro le informazioni sui repository software: prima di effettuare l'aggiornamento dei repository i dati sono stati camuffati per evitare di poter associare le informazioni a persone specifiche.

Il sistema deve impedire l'accesso e gli attacchi da parte di utenze non autorizzate. L'applicativo prevede sia funzioni di logging[35], sia l'implementazione di un servizio di autenticazione (`Keycloak`) che permette di effettuare l'autenticazione a due fattori. L'esposizione minima è garantita dall'applicativo che offre verso l'esterno l'accesso tramite un servizio di gateway e che controlla che l'accesso avvenga solo per gli utenti autenticati.

I dati devono essere protetti e regolarmente salvati. Il Cloud Provider fornisce questa protezione garantendo di poter recuperare i dati e salvando più copie degli stessi in diverse località[36], mentre la stessa funzionalità nel caso On-premise richiede costi aggiuntivi e l'utilizzo di risorse umane per la manutenzione. L'applicativo permette di eseguire dei lavori periodici che effettuano il dump del DB ma la gestione del salvataggio su supporti diversi e in località diverse[37] non è previsto.

[35] Il log viene registrato sul journal del DB e nei singoli record per tenere traccia di chi e che cosa è stato registrato per i pazienti

[36] I dati vengono salvati nelle zone geografiche richieste dagli utenti non facendo venir meno il requisito di mantenere i dati nella Comunità Europea o in paesi con accordi che rispettano il GDPR

[37] Le regole per garantire la possibilità di recuperare i dati in caso di disastro prevedono la dislocazione in aree non vicine per far fronte ad eventi naturali come alluvioni o terremoti che potrebbero riguardare zone limitrofe e causare una perdita irreversibile dei dati.

Anche per quanto riguarda la garanzia di raggiungibilità il servizio in Cloud offre infrastrutture in grado di soddisfare il requisito, mentre nel caso On-premise non appare motivato un investimento che potrebbe diventare ingente: nel caso non sia sopportabile una riduzione momentanea del servizio[38] il servizio On-premise non sarebbe in grado di soddisfare il requisito.

[38]Bisogna tener conto che il client può essere sviluppato in modo da eseguire i salvataggi in locale ed effettuare la sincronizzazione con il server in un secondo momento e quindi il servizio sarebbe ridotto in quanto le modifiche sarebbero effettuate solo in locale ma l'applicativo, nel suo complesso, continuerebbe a funzionare.

Capitolo 4

Conclusioni

4.1 Introduzione

Il progetto presenta un prototipo di applicazione che raggiunge l'obiettivo di fornire una strumentazione adattabile alle esigenze del reparto ospedaliero che lo dovrà utilizzare. Infatti, le scelte progettuali adottate garantiscono la possibilità di estendere l'applicativo e di adeguarlo a nuove necessità. In primo luogo, l'architettura a microservizi permette maggiori manutenibilità, scalabilità e flessibilità. La presenza di un unico punto di ingresso permetterà di aggiungere funzioni trasversali all'applicazione intervenendo esclusivamente sul `Gateway`. Inoltre, la sostituzione di una parte del progetto o una sua ulteriore implementazione sono meno onerose rispetto a soluzioni monolitiche. In secondo luogo, la scelta di gestire le informazioni sfruttando la semantica di `JSON Schema` senza legarle a strutture dati predefinite permetterà di intervenire sul formato dei dati senza dover intervenire sul software. Anche la validazione dei dati effettuata tramite l'impostazione dei vincoli sugli schemi assicura che il sistema possa evolvere senza dover intervenire direttamente sul software. In terzo luogo, la scelta degli strumenti di sviluppo offre delle prospettiva di tenuta sul medio o lungo periodo in quanto hanno dimostrato di essere maturi senza essere obsoleti. Anche la scelta della libreria utilizzata per la validazione dei dati va in questa direzione perché adotta come linguaggio `Kotlin`, rendendo omogenee le necessità di conoscenza da parte di chi si occuperà nel tempo della manutenzione e dello sviluppo dell'applicativo. Infine, il framework adottato, `Spring`, attraverso le annotazioni offre una semantica molto potente per condizionare il comportamento dei componenti senza richiedere la riscrittura di codice ridondante che, generalmente, causa molte difficoltà e dispendio di risorse in fase di manutenzione.

Alcune aree del progetto sono ancora da esplorare in quanto non è stato possibile definire con tutti gli attori coinvolti le linee da seguire. Inoltre, determinate scelte andranno fatte dopo che la versione beta del progetto sarà stata utilizzata e testata dal personale sanitario ed eventualmente modificata in relazione alle osservazioni o richieste che lo stesso personale farà.

4.2 Possibili sviluppi futuri

4.2.1 Referti ed esportazione dati

Le prime funzionalità che necessitano di essere sviluppate sono la creazione e la manutenzione dei referti e l'esportazione dei dati ai fini statistici. In sede di analisi il tema è stato affrontato superficialmente e rinviato alla fase successiva a quella di test della versione beta. Per quanto riguarda i referti, l'applicativo dovrà fornire dei template legati al tipo di visita o prescrizione oggetto del referto che l'utente potrà modificare e salvare per future interrogazioni. I template non dovranno avere una struttura fissa ma dipendere da parametri impostati sul database per formare il testo del referto. Una soluzione potrebbe consistere nell'utilizzo dei grafi per costruire il testo in modo dinamico e alternativo in base a valori presenti nel database o alle scelte effettuate

dal personale sanitario. Per l'esportazione dei dati il sistema dovrà prevedere l'estrazione delle informazioni in formati diversi, in relazione a quale uso i ricercatori ne dovranno fare, oltre a poter differenziare i dati da estrarre. Infatti, dovrà essere possibile effettuare una selezione sia del tipo di dato (tipo di esame, valori rilevati, ecc.) sia di quali pazienti (caratteristiche del paziente, range di date delle analisi, ecc.). In entrambi i casi le opzioni di estrazione/utilizzo dovranno essere modificabili dall'utente.

4.2.2 Database

In questa prima fase di sviluppo del progetto la struttura delle informazioni è stata mantenuta identica a quella presente nella vecchia implementazione del software. È necessario effettuare un'analisi accurata insieme al personale sanitario per verificare la coerenza delle strutture, la ridondanza dei dati e se per alcuni di essi è necessario effettuare uno spostamento in modo che rimanga la memoria storica di quanto inserito. Esistono, infatti, alcune informazioni che sono registrate sui dati personali (per i quali non esiste la possibilità di registrarne l'evoluzione nel tempo) ma che appaiono essere delle misurazioni per le quali potrebbe essere importante mantenere la cronologia delle modifiche.

Per migliorare la scalabilità e la suddivisione delle competenze, si può promuovere il package `data` a uno o più microservizi in modo da non richiedere la ricompilazione degli altri microservizi in caso di modifica dell'accesso al DB. Inoltre, questi servizi potranno essere sviluppati sfruttando il linguaggio `GraphQL` che permette di rompere la dipendenza tra i componenti fornendo un protocollo dichiarativo in cui il server comunica quali tipi di risorse dispone e i client compongono le richieste in base alle loro necessità ignorando il formato dei dati presenti nello storage sul server[18]. La scelta del linguaggio e delle API non è comunque vincolante in quanto le informazioni in questo progetto sono già state rese flessibili dalla scelta di renderle dipendenti dagli schemi salvati sul database. Di conseguenza, la tipologia di servizi da sviluppare andrà valutata in modo opportuno dopo attenta analisi del rapporto costi benefici.

Per quanto riguarda, invece, la gestione degli utenti dell'applicazione andrà verificato se mantenere l'attuale sistema di autenticazione (`Keycloak`) oppure se verrà adottata una soluzione diversa. In quest'ultimo caso andrà modificato anche il servizio di manutenzione dell'anagrafica degli utenti.

4.2.3 Autenticazione

Per quanto riguarda l'autenticazione, due sono gli aspetti che andranno affrontati. Il primo riguarda l'integrazione con i sistemi di autenticazione dell'ospedale. Il servizio del progetto può essere sostituito da qualsiasi componente che implementi le specifiche `OAuth2` ed inoltre `Keycloak` può essere configurato per utilizzare un repository esterno (ad esempio che implementi un server `LDAP`) con il quale interagire per l'autenticazione degli utenti. Il secondo riguarda l'autenticazione a due fattori prevista dal `GDPR` per il trattamento dei dati sensibili. Anche questa funzionalità andrà discussa con i responsabili dell'ospedale (in particolare con il `DPO`) per adottare una politica comune e coerente con le altre soluzioni utilizzate dal personale dell'ospedale. Per questa funzionalità sarà necessario intervenire anche sul servizio di Single Sign On che prevede solo l'autenticazione con username e password.

4.2.4 Implementazioni del Gateway

Il `gateway` riceve e smista tutte le richieste ed inoltra le relative risposte ai client. Attraverso il meccanismo dei filtri può elaborare le richieste ed effettuare delle azioni che interessano tutti i microservizi. La prima implementazione consiste nella registrazione di tutte le attività svolte sul database per permettere alle funzioni di auditing di effettuare le opportune verifiche. Infatti, i servizi registrano sul database le azioni che comportano una modifica del database, ma potrebbe essere necessario anche tenere conto di chi accede solo per leggere i dati o per esportarli. Il `gateway` potrebbe intercettare queste richieste ed effettuare le registrazioni nella collezione di log.

Un'altra implementazione riguarda l'intercettazione delle operazioni che hanno degli effetti sul database e che potrebbero essere veicolate ai client attraverso un servizio di `Websocket` in modo che le applicazioni di frontend possano reagire aggiornando l'interfaccia utente in funzione degli interventi avvenuti sul database.

Bibliografia

[1] Documentazione Keycloak, versione 21.1, https://www.keycloak.org/archive/documentation-21.1.html, visitato l'ultima volta il 30 giugno 2023

[2] Spring tutorial, https://spring.io/guides, visitato l'ultima volta il 2 giugno 2023

[3] Deploying a MongoDB Cluster with Docker https://www.mongodb.com/compatibility/deploying-a-mongodb-cluster-with-docker visitato l'ultima volta il 3 agosto 2023

[4] Spring Data MongoDB Transactions https://www.baeldung.com/spring-data-mongodb-transactions visitato l'ultima volta il 29 settembre 2023

[5] JSON and BSON https://www.mongodb.com/json-and-bson visitato l'ultima volta il 29 settembre 2023

[6] Kyle Banker, Peter Bakkum, Shaun Verch, Douglas Garrett, Tim Hawkins, *MongoDB in Action* Second Edition, Manning Publications Co., 2016, ISBN: 978-1-61729-160-9

[7] John Carnell, Illary Huaylupo Sánchez, *Spring Microservices in Action*, Second Edition, Manning Publications Co., 2021, ISBN: 978-1-61729-695-6

[8] Justin Richer, Antonio Sanso, *OAuth2 in Action*, First Edition, Manning Publications Co., 2017, ISBN: 978-1-61729-327-6

[9] Craig Walls, *Spring in Action*, Sixth Edition, Manning Publications Co., 2022, ISBN: 978-1-61729-757-1

[10] Chris Richardson, *Microservices Patterns*, First Edition, Manning Publications Co., 2019, ISBN: 978-1-61729-454-9

[11] Laurenţiu Spilcă, *Spring Security in Action*, First Edition, Manning Publications Co., 2020, ISBN: 978-1-61729-773-1

[12] Laurenţiu Spilcă, *Spring Security in Action*, MEAP?? Second Edition, Manning Publications Co., 2020, ISBN: 978-1-61729-773-1???

[13] Dmitry Jemerov, Svetlana Isakova, *Kotlin in Action*, First Edition, Manning Publications Co., 2017, ISBN: 978-1-61729-329-0

[14] Pierre-Yves Saumont *The Joy of Kotlin*, First Edition, Manning Publications Co., 2019, ISBN: 978-1-61729-536-2

[15] Marko Lukša, *Kubernetes in Action*, First Edition, Manning Publications Co., 2018, ISBN: 978-1-61729-372-6

[16] Gigi Sayfan, *Mastering Kubernetes*, Third Edition, Packt Publishing Ltd, 2020, ISBN: 978-1-83921-125-6

[17] Jeff Nickoloff, Stephen Kuenzli, *Docker in Action*, Second Edition, Manning Publications Co., 2019, ISBN: 978-1-61729-476-1

[18] Samer Buna, *GraphQL in Action*, First Edition, Manning Publications Co., 2021, ISBN: 978-1-61729-568-3