UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato **Software Architecture Design**

Sistema Prenotazione vaccini COVID-19

Anno Accademico 2020/21

Gruppo:

Michele Maresca M63/1151

Vincenzo Riccardi M63/1146

Marco Carlo Feliciano M63/1136

Indice

1. Processo di sviluppo

Si è scelto di utilizzare un processo di sviluppo agile, seguendo le indicazioni fornite dal framework organizzativo Scrum.

1.1 Scrum

Come previsto dalla Scrum Guide, lo scrum team dedicato al progetto è composto dai seguenti ruoli:

- **Product owner**: figura responsabile di massimizzare il valore del prodotto da realizzare ed unico responsabile del product backlog. Egli definisce le caratteristiche del prodotto e le loro priorità.

- **Development team**: insieme di professionisti il cui compito è quello di realizzare incrementi del prodotto potenzialmente rilasciabili. Il team è autonomo, auto-organizzante ed è composto al massimo da 7 persone.

- **Scrum master**: figura responsabile di promuovere e sostenere Scrum, aiutando il team a comprendere appieno i principi su cui si fonda. Ha il compito di rimuovere gli impedimenti riscontrati dal team di sviluppo, gestendo le interazioni tra il team e l'esterno.

Sebbene consapevoli della separazione esistente tra i diversi ruoli, a scopo didattico, ciascun membro del team ha ricoperto all'occorrenza ognuno di essi.

Durante il processo di sviluppo si è cercato di rispettare quanto più fedelmente gli eventi previsti da Scrum:

- **Sprint** (4 settimane): evento contenente gli eventi descritti in seguito.

- **Sprint planning** (4 ore): evento in cui di pianifica il lavoro da svolgere durante uno sprint. In particolare, viene definito cosa deve essere realizzato e come deve essere svolto il lavoro. Vengono prelevati alcuni task dal product backlog e si compone lo sprint backlog per la successiva iterazione. Durante tale evento viene definito lo sprint goal, ovvero l'obiettivo da raggiungere al termine dello sprint.

- **Daily Scrum** (15 minuti): evento quotidiano in cui il team sincronizza le proprie attività e pianifica il lavoro delle successive 24 ore.

- **Sprint review** (3 ore): evento in cui viene revisionato l'incremento rilasciato al termine di uno sprint. Lo scrum team illustra agli stakeholders il lavoro completato, al fine di ricevere feedback sull'incremento del prodotto e su cosa realizzare nell'incremento successivo.

- **Sprint retrospective** (1 ora e 30 minuti): evento in cui lo scrum team analizza sé stesso, al fine di migliorare il proprio rendimento nello sprint successivo. Si analizza cosa è andato bene e cosa, invece, si potrebbe migliorare.

In particolare, è stato realizzato uno sprint della durata di quattro settimane.

Durante lo sviluppo sono stati prodotti gli artefatti previsti da Scrum:

- **Product Backlog**: elenco ordinato di tutto ciò che è necessario al prodotto. È l'unica fonte a cui bisogna fare riferimento in caso di cambiamenti. È bene osservare che il product backlog non è mai completo ma evolve contemporaneamente allo sviluppo.

- **Sprint Backlog**: insieme degli elementi del product backlog selezionati per lo sprint corrente, a cui si aggiungono tutte le informazioni necessarie per consegnare l'incremento del prodotto e realizzare lo sprint goal.

- **Incremento**: somma di tutti gli elementi del product backlog completati durante lo sprint corrente e di tutti quelli realizzati negli sprint precedenti. È bene osservare che un incremento rilasciato al termine di uno sprint deve essere sempre potenzialmente rilasciabile.

1.2 Pratiche Agili

Sono state applicate diverse pratiche agili, tra le quali vale la pena menzionare:

- TDD (Test Driven Development);

- sviluppo iterativo;

- pair programming;

- time-boxing;

- refactoring del codice.

1.3 Strumenti software di supporto

1.3.1 Jira

Come supporto all'intero processo di sviluppo è stata adottata la piattaforma Jira Software. In particolare, tra le funzionalità offerte sono state utilizzate le seguenti:

- definizione del product backlog;
- definizione di un epic;
- definizione di una storia utente;
- definizione di un task;
- assegnazione di priorità ad una storia utente;
- assegnazione di sottotask ad un task o ad una storia utente;
- assegnazione di una storia utente ad un epic;
- avvio di uno sprint.

1.3.2 Github

Per supportare la gestione della configurazione e la collaborazione tra i diversi membri del team è stato utilizzato il version control system GitHub. In questo modo è stato possibile lavorare in parallelo sul codice senza avere particolari conflitti. Al seguente link è possibile accedere al repository del progetto:

https://github.com/Shotokhan/sad_project_microservices

1.3.3 Visual Paradigm

Per la rappresentazione dei diagrammi relativi alla documentazione è stato utilizzato Visual Paradigm. Esso consente di rappresentare i principali diagrammi UML, dunque si è fatto uso di stereotipi per esprimere concetti specifici del dominio applicativo.

1.3.4 Flask

Per la fase di sviluppo del back-end è stato utilizzato un microframework in python ovvero Flask, basato sullo strumento Werkzeug WSGI (Web Service Gateway Interface) e con il motore di template Jinja2. Il primo strumento è un protocollo e specifica come i server si facciano carico delle richieste provenienti dai browser/client ed inoltrino le informazioni richieste alle relative applicazioni. Il secondo strumento è un motore di template e permette la creazione di file HTML, XML o in altri formati di markup, che vengono restituiti all'utente mediante una risposta HTTP.
Flask è chiamato "microframework" perché ha un nucleo semplice ma estendibile. Non c'è uno strato di astrazione per la base di dati, validazione dei formulari, o qualsiasi altra componente per fornire funzionalità comuni per le quali esistono già librerie di terze parti. A ogni modo, Flask supporta estensioni che possono aggiungere funzionalità a un'applicazione come se fossero implementate dallo stesso Flask. Ad esempio, sono presenti estensioni per la validazione dei formulari, la gestione del caricamento dei file, varie tecnologie di autenticazione e altro.

1.3.5 SQLAlchemy

SQLAlchemy è un toolkit SQL open source e un mappatore relazionale a oggetti (ORM) per il linguaggio di programmazione Python rilasciato sotto la licenza MIT.
La filosofia di SQLAlchemy è che i database relazionali si comportano meno come raccolte di oggetti man mano che la scala aumenta e le prestazioni iniziano a essere un problema, mentre le raccolte di oggetti si comportano meno come tabelle e righe poiché in esse viene progettata una maggiore astrazione. Per questo motivo ha adottato il modello data mapper (simile a Hibernate per Java) piuttosto che il modello di record attivo utilizzato da un certo numero di altri mappatori relazionali a oggetti. Tuttavia, i plug-in opzionali consentono agli utenti di sviluppare utilizzando la sintassi dichiarativa.

1.3.6 Docker, docker-compose & Makefile

Docker è un progetto open-source che automatizza il processo di deployment di applicazioni all'interno di contenitori software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux.
Il tool docker-compose serve per definire dei parametri per l’esecuzione di Docker, ad esempio relativi alle interfacce di rete, al port forwarding ed a variabili d’ambiente. Esso semplifica ulteriormente l’API per la gestione dei container, definendo dei metodi “build”, “up”, “down”, “start”, “stop”, “logs” e così via. Se si fa in modo che non sia eseguita direttamente l’applicazione, ma che essa sia eseguita mediante un wrapper script, è possibile eseguire i test di unità dell’applicazione all’avvio dell’applicazione stessa mediante Docker, coniugando l’automazione di build & deployment con l’automazione dei test. Quest’osservazione è molto utile se si adotta un approccio di sviluppo di tipo TDD.
In generale Docker si colloca bene nell’ambito di pratiche DevOps.
Il Makefile è un formato di file usato dall’utility make, che permette di gestire le dipendenze delle applicazioni e di definire rapidamente dei comandi. Ad esempio, se si hanno 3 microservizi, ciascuno con il suo docker-compose.yml, un approccio ingenuo è di fare un solo docker-compose.yml per avviarli tutti e 3; tuttavia, così facendo, essi non sono più separati, e quindi non possono essere deployed in modo indipendente. Quindi si può piuttosto creare un Makefile che fornisce delle primitive del tipo “make build”, “make up” e così via per chiamare i metodi rispettivamente “build”

ed “up” di ciascun docker-compose.yml, fornendo la possibilità di fare build & test rapido in locale, e lasciando la possibilità di effettuare deployment di ciascun microservizio su nodi differenti.

1.3.7 Swagger - OpenAPI

Swagger è un insieme di specifiche e di strumenti che mirano a semplificare e standardizzare i processi di documentazione di API per servizi web RESTful. Il cuore di Swagger consiste in un file testuale (in formato sia YAML che JSON) dove sono descritte tutte le funzionalità di un'applicazione web e i dettagli di input e output in un formato studiato per essere interpretabile correttamente sia dagli umani che dalle macchine. I vantaggi di questa standardizzazione sono molti, come una migliore e più condivisibile esplorazione delle funzionalità di un'applicazione, oltre che alla possibilità di generare codice client/server sfruttando direttamente i vincoli definiti nello schema. Infatti, i metadati presenti nel file forniscono informazioni sufficienti sia per generare il componente backend con le rotte http e la validazione degli input, sia la parte client che in modo automatico può adattarsi all’evoluzione delle API del backend. La generazione della parte server è sicuramente vantaggiosa in quanto è possibile concentrarsi direttamente alla programmazione della business logic.

2. Avvio del progetto

Si vuole realizzare un sistema software per le prenotazioni dei vaccini COVID-19.

2.1 Parti interessate

Il sistema è organizzato in modo tale da essere utilizzato da tre attori: **Paziente**, **Operatore** e **Amministratore**.

2.2 Obiettivi Utente

2.2.1 Paziente

Il sistema deve consentire al paziente di effettuare una prenotazione per il vaccino COVID-19. La prenotazione può essere effettuata solo appena è schedulata la fascia d'età alla quale appartiene il paziente, fatta eccezione per i pazienti affetti da patologie, i quali possono effettuare una prenotazione anche quando non è stata ancora schedulata la propria fascia d'età. (es. se la fascia d'età schedulata è over-60, un paziente che ha età 30 anni, e non è affetto da alcuna patologia, non può ancora effettuare una prenotazione, mentre un paziente che ha 65 anni può effettuare una prenotazione, così come un paziente di 30 anni che è affetto da una patologia).
Nel caso in cui il vaccino richieda una doppia dose la singola prenotazione effettuata varrà per entrambe le dosi.
Inoltre, al paziente è fornita un'area personale alla quale accede successivamente ad una fase di registrazione prima e login poi. In particolare, all'interno dell'area personale può visualizzare la propria prenotazione e gli è consentito anche di disdire la propria prenotazione.
Il paziente può visualizzare un report giornaliero riguardante il numero di somministrazioni totali effettuate e il numero totale di vaccinati.

2.2.2 Operatore

Il sistema permette agli operatori, una volta autenticati, di visualizzare la lista giornaliera di prenotati per poter gestire i turni di vaccinazione. Inoltre, l'operatore è responsabile di confermare l'avvenuta vaccinazione del paziente, in modo tale da potergli fornire la "*certificazione vaccino"*.

2.2.3 Amministratore

Il sistema permette all'amministratore di aggiungere o modificare le finestre di prenotazione.
Inoltre, se un paziente non riesce ad effettuare la prenotazione mediante il sistema (es. pazienti anziani) può chiamare un numero verde e l'amministratore si occuperà di aggiungere la prenotazione al sistema.

3. Specifica ed analisi dei requisiti

In linea con la metodologia agile adottata, si è scelto di documentare i requisiti del sistema attraverso il meccanismo delle **storie utente**.

3.1 Storie utente

Le storie utente sono state riportate all'interno del product backlog stesso. Ogni storia utente è caratterizzata da:

- **titolo**;
- **descrizione**: breve frase che descrive una funzionalità del prodotto dal punto di vista di un attore specifico, scritta usando il seguente formato:

  Come [attore]
  Io voglio [funzionalità]
  In modo da [motivo]

- **priorità**: indica il grado di importanza della storia utente all'interno del sistema. Storie utente con maggiore priorità rappresentano funzionalità da implementare prima nel processo di sviluppo;
- **story points**: unità di misura per esprimere una stima dello sforzo richiesto per implementare completamente la storia utente;
- **epic link** (opzionale): epic a cui la storia utente appartiene;
- **assegnatario** (opzionale): membro del team incaricato di implementare la storia utente;
- **etichette** (opzionali): parole chiave associate alla storia utente;

Di seguito si riportano le storie utente realizzate per la specifica dei requisiti.

Storia Utente	
Titolo	Registrazione
Descrizione	*Come* Paziente *Voglio* registrarmi *In modo da* poter successivamente accedere ai servizi
Priorità	High
Story point estimate	9
Epic link	Gestione Utente
Etichette	Requisito Paziente

Tabella 3.1: Storia Utente: Registrazione

Storia Utente	
Titolo	Login
Descrizione	*Come* Paziente/Operatore *Io voglio* autenticarmi *In modo da* accedere alle funzionalità del sistema
Priorità	High
Story point estimate	5
Epic link	Gestione Utente
Etichette	Requisito Paziente Requisito Operatore

Tabella 3.2: Storia Utente: Login

Storia Utente	
Titolo	Effettua prenotazione
Descrizione	*Come* paziente *Io voglio* effettuare una prenotazione *In modo da* poter essere vaccinato
Priorità	High
Story point estimate	7
Epic link	Gestione Prenotazione
Etichette	Requisito Paziente

Tabella 3.3: Storia Utente: Effettua prenotazione

Storia Utente	
Titolo	Visualizza prenotati
Descrizione	*Come* operatore *Io voglio* vedere la lista dei prenotati *In modo da* poter visualizzare i pazienti da vaccinare
Priorità	High
Story point estimate	4
Epic link	Gestione Prenotazione
Etichette	Requisito Operatore

Tabella 3.4: Storia Utente: Visualizza prenotati

Storia Utente	
Titolo	Visualizza propria prenotazione
Descrizione	*Come* paziente *Io voglio* visualizzare la mia prenotazione *In modo da* poter vedere la data, l'orario e il luogo dell'impegno
Priorità	High
Story point estimate	4
Epic link	Gestione Prenotazione
Etichette	Requisito Paziente

Tabella 3.5: Storia Utente: Visualizza propria prenotazione

Storia Utente	
Titolo	Effettua prenotazione patologia
Descrizione	*Come* paziente *Io voglio* poter effettuare la prenotazione a maggiore priorità *In modo da* poter essere inserito nel turno di una categoria schedulata prima rispetto alla mia
Priorità	Medium
Story point estimate	3
Epic link	Gestione Prenotazione
Etichette	Requisito Paziente

Tabella 3.6: Storia Utente: Effettua prenotazione patologia

Storia Utente	
Titolo	Disdici prenotazione
Descrizione	*Come* paziente *Io voglio* disdire la mia prenotazione *In modo da* poter cambiare data della vaccinazione
Priorità	Medium
Story point estimate	4
Epic link	Gestione Prenotazione
Etichette	Requisito Paziente

Tabella 3.7: Storia Utente: Disdici prenotazione

Storia Utente	
Titolo	Visualizza report
Descrizione	*Come* paziente *Io voglio* poter visualizzare un report *In modo da* poter visualizzare i dati sulle vaccinazioni
Priorità	Low
Story point estimate	3
Epic link	
Etichette	Requisito Paziente

Tabella 3.8: Storia Utente: Visualizza report

Storia Utente	
Titolo	Notifica prenotazione presa in carico
Descrizione	*Come* paziente *Io voglio* essere avvisato quando una prenotazione è stata presa in carico *In modo da* sapere che il sistema sta elaborando la mia richiesta di prenotazione
Priorità	Medium
Story point estimate	5
Epic link	Gestione Prenotazione
Etichette	Requisito Paziente

Tabella 3.9: Storia Utente: Notifica prenotazione effettuata

Storia Utente	
Titolo	Notifica promemoria
Descrizione	*Come* paziente *Io voglio* ricevere un promemoria *In modo da* ricordare la data del vaccino
Priorità	Low
Story point estimate	5
Epic link	Gestione Prenotazione
Etichette	Requisito Paziente

Tabella 3.10: Storia Utente: Notifica promemoria

Storia Utente	
Titolo	Notifica prenotazione vaccino
Descrizione	*Come* paziente *Io voglio* essere avvisato quando una prenotazione ha esito positivo *In modo da* sapere le informazioni per la vaccinazione
Priorità	Medium
Story point estimate	5
Epic link	Gestione Prenotazione
Etichette	Requisito Paziente

Tabella 3.11: Storia Utente: Notifica prenotazione vaccino

Storia Utente	
Titolo	Notifica disponibilità vaccino
Descrizione	*Come* paziente *Io voglio* ricevere una notifica *In modo da* sapere quando la mia fascia d'età può vaccinarsi
Priorità	Medium
Story point estimate	5
Epic link	Gestione Prenotazione
Etichette	Requisito Paziente

Tabella 3.12: Storia Utente: Notifica disponibilità vaccino

Storia Utente	
Titolo	Gestione account utente
Descrizione	*Come* Paziente *Io voglio* avere accesso ai dati del mio account *In modo da* cambiare le mie informazioni di contatto
Priorità	Low
Story point estimate	4
Epic link	Gestione Utente
Etichette	Requisito Paziente

Tabella 3.13: Storia Utente: Gestione utente

Storia Utente	
Titolo	Certificazione vaccino
Descrizione	*Come* Operatore *Io voglio* certificare l'avvenuta vaccinazione *In modo da* confermare che il paziente ha effettuato il vaccino
Priorità	Medium
Story point estimate	5
Epic link	
Etichette	Requisito Operatore

Tabella 3.14: Storia Utente: Certificazione vaccino

Storia Utente	
Titolo	Aggiungi prenotazione
Descrizione	*Come* Amministratore *Io voglio* poter aggiungere una prenotazione *In modo da* poter prenotare anche pazienti che non interagiscono direttamente col sistema
Priorità	Low
Story point estimate	5
Epic link	Gestione Prenotazione
Etichette	Requisito Amministratore

Tabella 3.15: Storia Utente: Aggiungi prenotazione

Storia Utente	
Titolo	Modifica finestre prenotazione
Descrizione	*Come* Amministratore *Io voglio* modificare le finestre di prenotazione *In modo da* consentire a nuovi pazienti di prenotarsi
Priorità	Low
Story point estimate	5
Epic link	Gestione Prenotazione
Etichette	Requisito Amministratore

Tabella 3.16: Storia Utente: Modifica finestre prenotazione

3.2 Diagramma dei casi d’uso

Si riporta per una maggiore completezza il diagramma di alto livello dei casi d’uso in figura 3.1.

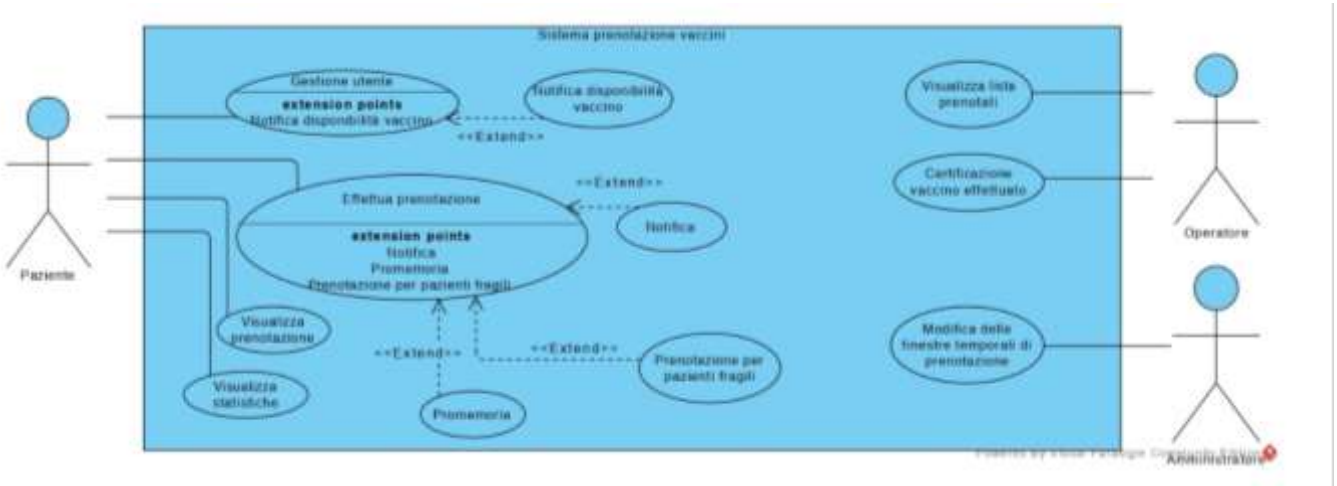


Figura 3.1: Diagramma dei casi d’uso

3.2 Altri requisiti

Di seguito sono riportati ulteriori requisiti dell’applicazione.

3.2.1 Requisiti del prodotto

Nel seguente paragrafo sono elencati i requisiti rispettati dall’applicativo:

- **Requisiti di usabilità**: il sistema deve essere intuitivo e facile da comprendere, in modo da poter essere utilizzato dai diversi utenti senza la necessità di corsi di formazione.
- **Requisiti di portabilità**: il sistema deve essere in grado di eseguire su sistemi operativi ed hardware diversi.
- **Requisiti di scalabilità**: il sistema deve essere adattato a diversi contesti con forti differenze di complessità (per esempio database molto piccoli o molto grandi) senza che questo richieda la riprogettazione dello stesso sistema.
- **Requisiti di verificabilità**: il sistema deve essere testabile in qualsiasi momento sulle proprietà di correttezza ed affidabilità.
- **Requisiti di manutenibilità**: il sistema deve essere modificabile con una spesa minima in termini di costi e tempo, prevedendo un massimo riutilizzo delle componenti esistenti.
- **Requisiti di robustezza**: il sistema deve essere in grado di rispondere in modo ragionevole ad errori o eccezioni non previste dalle specifiche.

- **Requisiti di riservatezza**: ciascun attore del sistema può accedere solo alle aree del sistema ad esso riservate. Un paziente può accedere solo alla propria prenotazione.

- **Requisiti di riusabilità**: il sistema deve essere affine all'evolvibilità.

3.3 System Domain Model

Il primo diagramma realizzato in fase di analisi è il **System Domain Model**, un class diagram di alto livello con il quale si indica il dominio applicativo del sistema. Mediante tale diagramma si specificano le entità coinvolte nel sistema software e le relazioni che vi sono tra di esse.

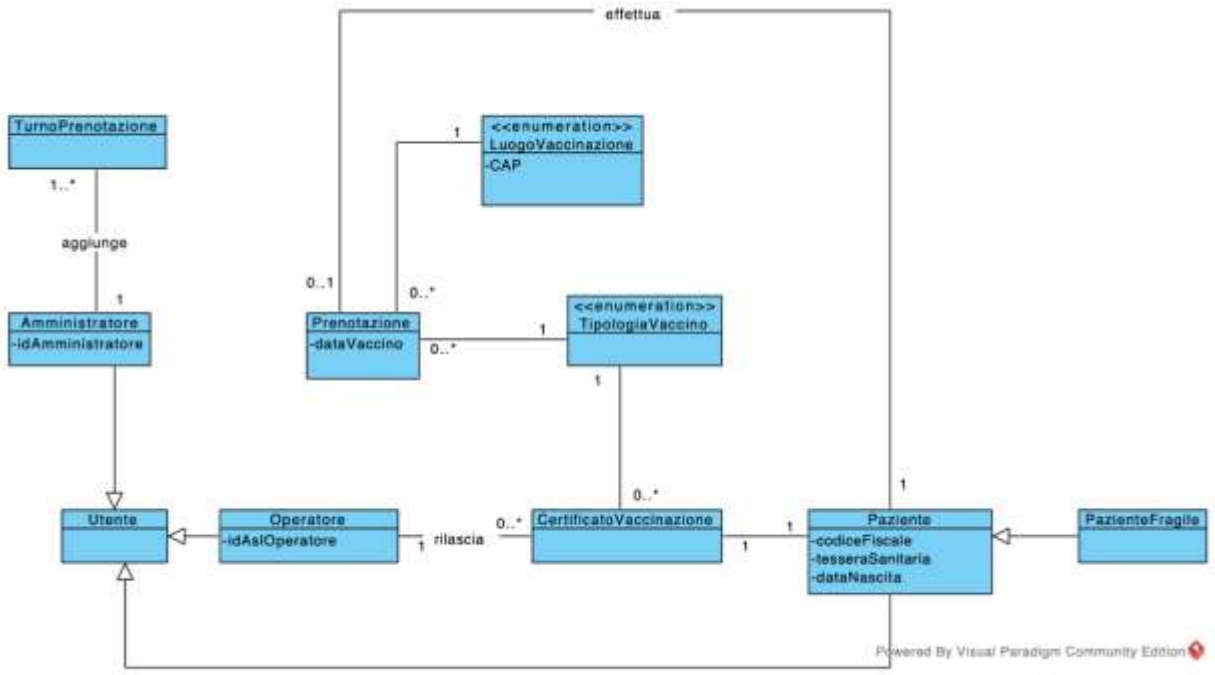


Figura 3.2: System domain Model

3.4 System Context Diagram

Allo scopo di descrivere il contesto nel quale è posto il sistema è stato rappresentato il diagramma di contesto in figura 3.3. È possibile comprendere gli attori che interagiscono con il sistema di prenotazione vaccini. In particolare, tramite browser possono accedere al sistema i pazienti, gli operatori e gli amministratori. Questi ultimi accedono al sistema per poterlo utilizzare per completare le proprie operazioni. Inoltre, il sistema interagisce anche con quattro sistemi esterni: “Sistema di notifiche”, “Database Anagrafica”, “Database Patologie” e “Database Luoghi”.
Sono state fatte le seguenti assunzioni:

- Il “Database Anagrafica” è un database esterno al sistema che memorizza tutti i cittadini, ognuno con le informazioni che lo riguardano, come nome, cognome, data di nascita, tessera sanitaria e codice fiscale. Viene utilizzato dal sistema di prenotazione vaccini al momento della registrazione dei pazienti, al fine di verificare l’effettiva esistenza di un paziente e quindi la correttezza delle informazioni da esso fornite.
- Il “Database Patologie” è un database esterno al sistema che memorizza tutti i pazienti affetti da una patologia, la quale risulta rilevante in fase di scheduling della prenotazione per il vaccino, poiché fornisce al paziente maggiore priorità. Viene utilizzato dal sistema di prenotazione vaccini al momento della prenotazione, nel caso in cui un paziente indica che è affetto da una determinata patologia, al fine di verificare la veridicità di tale informazione.
- Il “Database Luoghi” è un database esterno che memorizza i luoghi. Esso viene utilizzato dal sistema di prenotazione vaccini per verificare l’esistenza dei luoghi indicati dai pazienti in fase di registrazione.
- Il “Sistema di notifiche” è un sistema esterno utilizzato per inviare le notifiche ai pazienti.

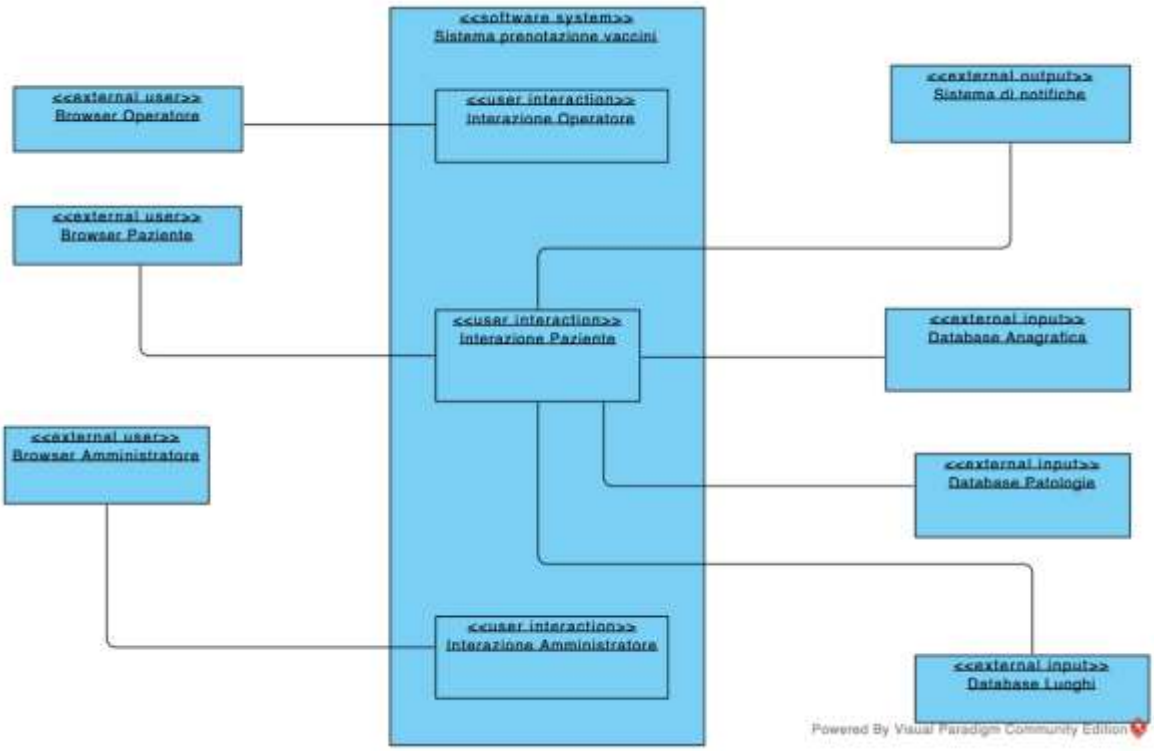


Figura 3.3: System Context Diagram

3.5 Sequence Diagram

Al fine di comprendere le funzionalità principali del sistema sono stati realizzati due Sequence Diagram di alto livello. In particolare, si è scelto di realizzare i diagrammi relativi alla registrazione e alla prenotazione di un vaccino, queste sono le due user stories aventi story points più elevati oltre che priorità massima.
Il primo diagramma descrive il flusso di messaggi per effettuare una registrazione.

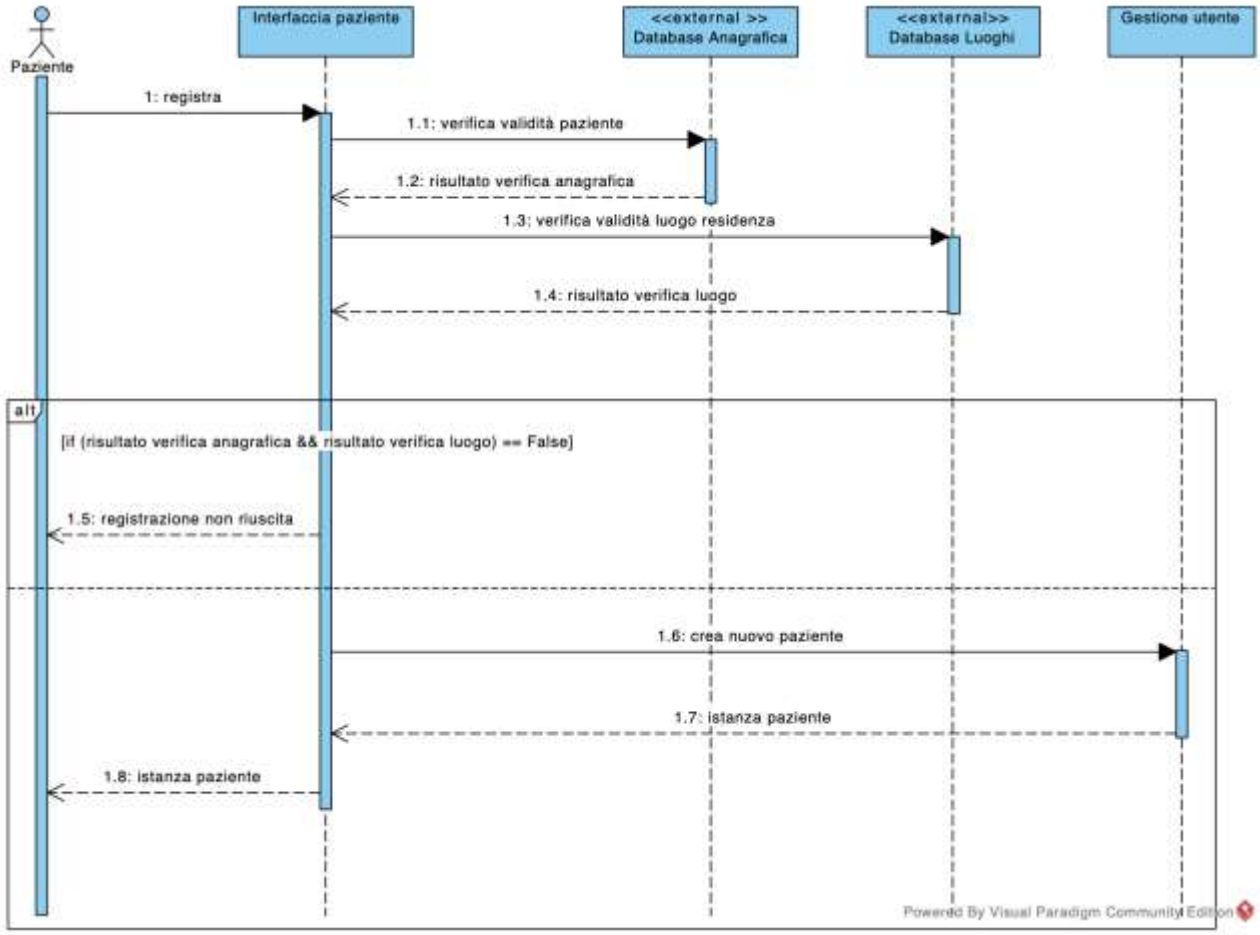


Figura 3.4: Sequence Diagram: Registrazione

Il Paziente ha un'interfaccia sulla quale può effettuare la registrazione. Quindi una volta fornite le informazioni necessarie e confermata la volontà di registrarsi, viene verificata la validità dei dati forniti dal Paziente mediante l'interazione con due sistemi esterni: "Database Anagrafica" e "Database Luoghi". Nel caso in cui la verifica si conclude senza successo (informazioni fornite scorrette), viene indicato al paziente che la registrazione non è riuscita; nel caso in cui la verifica ha successo, viene registrato il Paziente e gli viene indicato che la registrazione è avvenuta con successo.

Il secondo diagramma presenta il flusso di messaggi per effettuare una prenotazione.

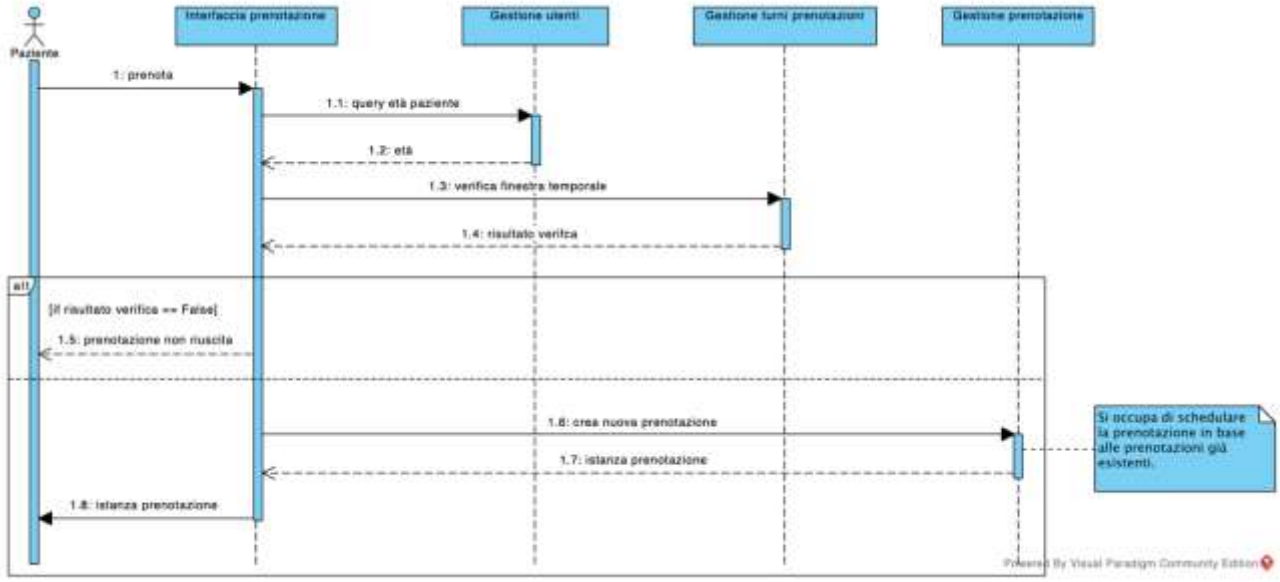


Figura 3.5: Sequence Diagram: Effettua prenotazione

Il Paziente ha un'interfaccia sulla quale può effettuare la prenotazione. Una volta confermata la volontà di prenotarsi, viene effettuato un controllo sull'età del paziente al fine di verificare la sua appartenenza ad un turno di prenotazione attivo. Nel caso in cui il risultato della verifica non ha successo, viene indicato al Paziente che la prenotazione non è stata effettuata; nel caso in cui la verifica ha successo viene registrata la prenotazione, viene indicato al Paziente che la prenotazione è avvenuta con successo e sono fornite le informazioni su data e luogo della vaccinazione.

4. Progettazione

In questo capitolo sono descritte le scelte progettuali relative al primo sprint.

4.1 Rationale

L'architettura scelta per lo sviluppo dell'applicativo è microservizi software. Si è scelto il modello architetturale a microservizi dati i seguenti vantaggi:

.

- Time-to-market rapido**,** permette di gestire i cicli di sviluppo in un tempo ridotto rispetto ai metodi tradizionali grazie alla loro particolare struttura e all'agilità degli aggiornamenti;

- Indipendenza, in quanto componenti separati e autonomi, possono essere rimpiazzati, sviluppati e ridimensionati singolarmente;

- Maggiore flessibilità e resilienza, un difetto isolato non inficia le stabilità del sistema, inoltre la sostituzione di un microservizio non ne modifica l'assetto e la funzionalità;

- Gestione dati decentralizzata, l'architettura microservizi prevede la suddivisione in database distribuiti garantendo minori costi di connessione mettendo i dati più vicini al sito che li utilizza maggiormente;

- Easy to deploy, le app basate su microservizi sono più piccole e modulari delle tradizionali applicazioni monolitiche, tutti i problemi associati a tali deployment vengono automaticamente eliminati. Benché questo approccio richieda un coordinamento superiore, i vantaggi che ne derivano sono determinanti;

- Scalabilità, al crescere della domanda per determinati servizi, i microservizi possono essere distribuiti su più server e infrastrutture aumentando in modo dinamico la disponibilità;

- Accessibilità, data la divisione dell'app in parti più piccole, per gli sviluppatori è molto più semplice comprendere, aggiornare e migliorare tali componenti, e questo permette di accelerare i cicli di sviluppo, soprattutto in combinazione con le metodologie di sviluppo Agile;

- Maggiore apertura, grazie alle API indipendenti dal linguaggio, gli sviluppatori sono liberi di scegliere il linguaggio e la tecnologia ottimali per la funzione da creare.

Vi sono però anche delle problematiche nell'uso dell'architettura a microservizi, come ad esempio la gestione della coerenza dei dati nei database distribuiti. Sono possibili tecniche di interazione tra microservizi gestendo la coerenza dei dati con protocolli di snooping o sfruttando tool come HazelCast. L'idea alla base di questo tool è gestire la coerenza dei dati tra microservizi, sistemi distribuiti o cloud sfruttando alla base una memoria condivisa, prevedendo periodici backup per aggiornare lo stato dei dati in memoria centrale. Il tool in particolare fornisce un'interfaccia REST rendendolo compatibile con qualunque applicativo che dispone di API RESTful. Si può pensare che esso funzioni, con riferimento all'applicativo prodotto, in un modo simile al seguente.

Se un utente fa una richiesta ad un microservizio e tale microservizio trova l'utente, allora NON è un falso positivo. Tuttavia, se l'utente non viene trovato, può essere un falso negativo, perché si può trovare in un altro microservizio: allora ci deve essere un microservizio BaseDiDati che contiene l'unione delle basi di dati dei diversi microservizi, con una opportuna gestione delle collisioni degli id ed un opportuno adattamento dello schema relazionale per contenere sia gli utenti sia le prenotazioni. Quindi, nel caso in cui si verifichi un "cache miss" (es. Paziente non trovato), il microservizio contattato fa una richiesta al microservizio BaseDiDati portando la riga nel suo DB. Affinché questo approccio sia efficiente, i microservizi devono fare "edge computing", ovvero ci dev'essere un reindirizzamento degli utenti agli opportuni microservizi su base geografica.
In generale, l'uso di database separati per ogni microservizio fornisce il vantaggio che in caso di corruzione dei dati in un database le stesse o alcune informazioni perse possono essere state memorizzate in database di altri microservizi, grazie alla ridondanza introdotta dalla scomposizione strutturale della base di dati.
Il pattern strutturale di riferimento è il Proxy Pattern, implementato nel microservizio di gateway, dati i seguenti vantaggi:

- Single access point, essendo l'applicativo a microservizi e web potrebbe essere distribuito localizzato su una macchina remota e contattare direttamente il servizio, soprattutto se complesso, determina latenze ed overhead;

- Consente di definire delle policy di sicurezza, non implementate nella business logic, evitando accessi indiscriminati alle risorse;

- Funge da dispatcher delle richieste client.

La struttura del pattern proxy prevede quattro partecipanti:

1. Client, colui che effettua l'invocazione all'operazione di interesse;

2. Subject Interface, ovvero l'interfaccia utilizzata dal Client che viene implementata dal Proxy e dal RealSubject;

3. Real Subject, l'oggetto reale di cui il Proxy avrà il compito di surrogare, nel caso del nostro applicativo identifichiamo come real subject i servizi di gestione utenti e gestione prenotazioni;

4. Proxy, ovvero la classe chiave che avrà il compito di surrogare l'oggetto reale mantenendo una reference a tale oggetto, creando e distruggendo l'oggetto ed esponendo gli stessi metodi pubblici del real subject definiti dall'interfaccia. Nel nostro applicativo già accennato il microservizio di gateway è il proxy.

Per la scelta del design è stato adottato un approccio RDD (Responsibility Driven Design), ovvero un modello di progettazione legato alla programmazione object-oriented, in cui l'attenzione è posta sulle responsabilità dei diversi oggetti e sulle informazioni che essi condividono.
In particolare, sono stati seguiti tre dei nove pattern GRASP ovvero:

- Controller, in quanto è stata disaccoppiata la logica applicativa dagli oggetti della user interface. Il microservizio di gateway si comporta come controller facade o controller di sistema in quanto ha il compito di rappresentare l'intero sistema agli occhi della UI. I task app nei microservizi di gestione utenti e gestione prenotazioni fungono da controller di caso d'uso essendo il loro lavoro delegato ad uno dei possibili compiti del sistema;

- Creator, in quanto le classi Paziente ed Operatore creano le rispettive classi DAO ed Utente DAO mediante la superclasse, anche la classe Prenotazione crea l'omonima classe DAO;

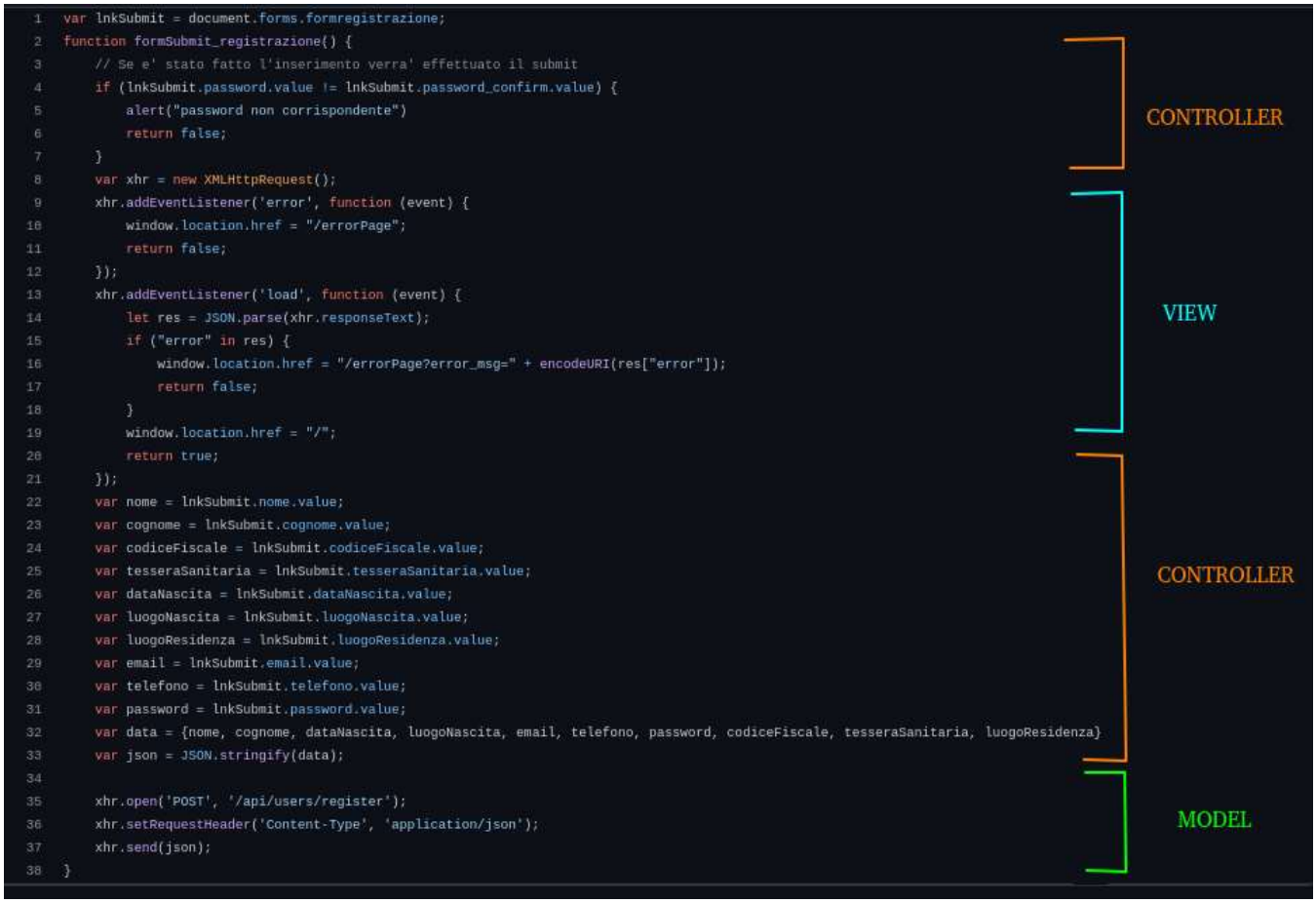
- Information expert, in quanto le classi Paziente, Operatore, Prenotazione ed User sono in grado di conoscere un determinato oggetto a partire dal suo identificatore, in particolare ogni classe è esperta di sé stessa.

Legati ai pattern GRASP sono rispettati i principi di low coupling ed high cohesion, derivanti dal pattern architetturale scelto, in quanto i microservizi sono pensati per essere indipendenti tra loro (basso accoppiamento) e ad avere la logica operativa completamente integrata (alta coesione).
Inoltre, sono stati sfruttati altri due design pattern nello specifico:

- Singleton, per gestire mediante un'unica istanza la configurazione dell'applicazione e l'oggetto mediante il quale si accede al database, ovvero che realizza l'API per la persistenza;

- DAO, in quanto sono state create classi ad hoc per la comunicazione col database, le quali espongono metodi per eseguire le operazioni di creazione e cancellazione di tuple dal database, il vantaggio nell'uso del pattern DAO è nel mantenere una rigida separazione tra le componenti dell'applicazione.

Nell'applicativo prodotto non è stato pensato in modo esplicito l'uso del pattern MVC nel front-end. Tuttavia, nelle funzioni javascript che effettuano richieste al server, possiamo riconoscere tale pattern. Di seguito è proposto un esempio.
Con riferimento alla funzione di registrazione, della quale riportiamo il codice *Figura 4.1*, notiamo che le linee di codice da quattro a sette e da ventidue a trentatré sono identificabili come Controller in quanto interpretano i dati in input dall'utente e formulano la richiesta al Model.
Come Model riconosciamo le linee di codice da trentacinque a trentasette, in quanto esponiamo i metodi per accedere ai dati utili dell'applicazione. Infine, la View, resa dalle linee di codice tra otto e ventuno, interpreta i dati e rende due output distinti. Di seguito il codice.




```
var lnkSubmit = document.forms.formregistrazione;
function formSubmit_registrazione() {
    // Se e' stato fatto l'inserimento verra' effettuato il submit
    if (lnkSubmit.password.value != lnkSubmit.password_confirm.value) {
        alert("password non corrispondente")
        return false;
    }
    var xhr = new XMLHttpRequest();
    xhr.addEventListener('error', function (event) {
        window.location.href = "/errorPage";
        return false;
    });
    xhr.addEventListener('load', function (event) {
        let res = JSON.parse(xhr.responseText);
        if ("error" in res) {
            window.location.href = "/errorPage?error_msg=" + encodeURI(res["error"]);
            return false;
        }
        window.location.href = "/";
        return true;
    });
    var nome = lnkSubmit.nome.value;
    var cognome = lnkSubmit.cognome.value;
    var codiceFiscale = lnkSubmit.codiceFiscale.value;
    var tesseraSanitaria = lnkSubmit.tesseraSanitaria.value;
    var dataNascita = lnkSubmit.dataNascita.value;
    var luogoNascita = lnkSubmit.luogoNascita.value;
    var luogoResidenza = lnkSubmit.luogoResidenza.value;
    var email = lnkSubmit.email.value;
    var telefono = lnkSubmit.telefono.value;
    var password = lnkSubmit.password.value;
    var data = {nome, cognome, dataNascita, luogoNascita, email, telefono, password, codiceFiscale, tesseraSanitaria, luogoResidenza}
    var json = JSON.stringify(data);

    xhr.open('POST', '/api/users/register');
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.send(json);
}
```

Figura 4.1 codice form_submitRregistrazione

Tale schema implicito è riconoscibile in ogni funzione javascript che ha il compito di effettuare richieste verso il back-end, dove il Controller risulterà sempre la porzione di codice che genera la richiesta in base agli input del client sulla UI, il Model sarà la porzione di codice che genera la HTTP request verso il server (o meglio, mediante richiesta viene invocato il Model, presente sul back-end) e la View sarà sempre la porzione di codice responsabile della resa grafica dei dati.

4.2 Progettazione dell'architettura

Di seguito saranno presentati i diagrammi e le tabelle utili nel progetto e, successivamente, nello sviluppo del primo sprint.

4.2.1 Component Diagram

Si è scelto di proporre una vista architetturale d'insieme usando un Component Diagram.

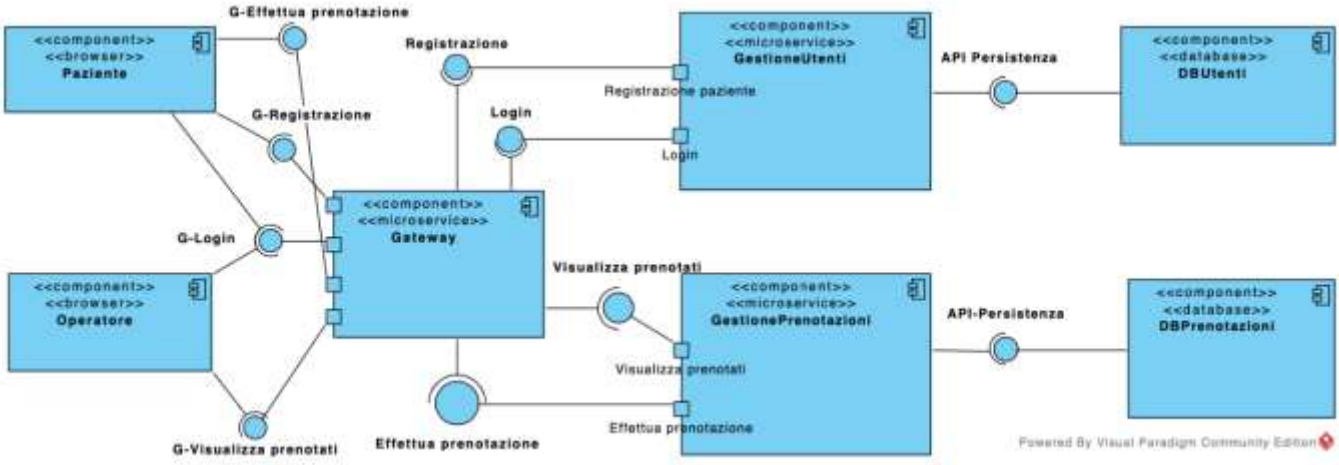


Figura 4.2: Component Diagram

Come evidenziato dal diagramma, i componenti del sistema sono:

1. Browser, web client in grado di inoltrare richieste HTTP al server. I diversi client possono essere visti come *thin-client*, in quanto la logica applicativa è concentrata quasi interamente sul server. Nel lato client si hanno le interfacce di operatore e paziente.

2. Microservice, web server (container) che espone risorse ai client. Il server prevede un singolo punto d'accesso ovvero il gateway e due servizi di elaborazione ovvero gestione utenti e prenotazioni.

3. Database, permette al server di gestire la persistenza dei dati. Nell'applicativo prodotto i database sono due, uno per microservizio.

4.2.2 Class Diagram

Di seguito è riportato il diagramma delle classi di alto livello della prima iterazione.

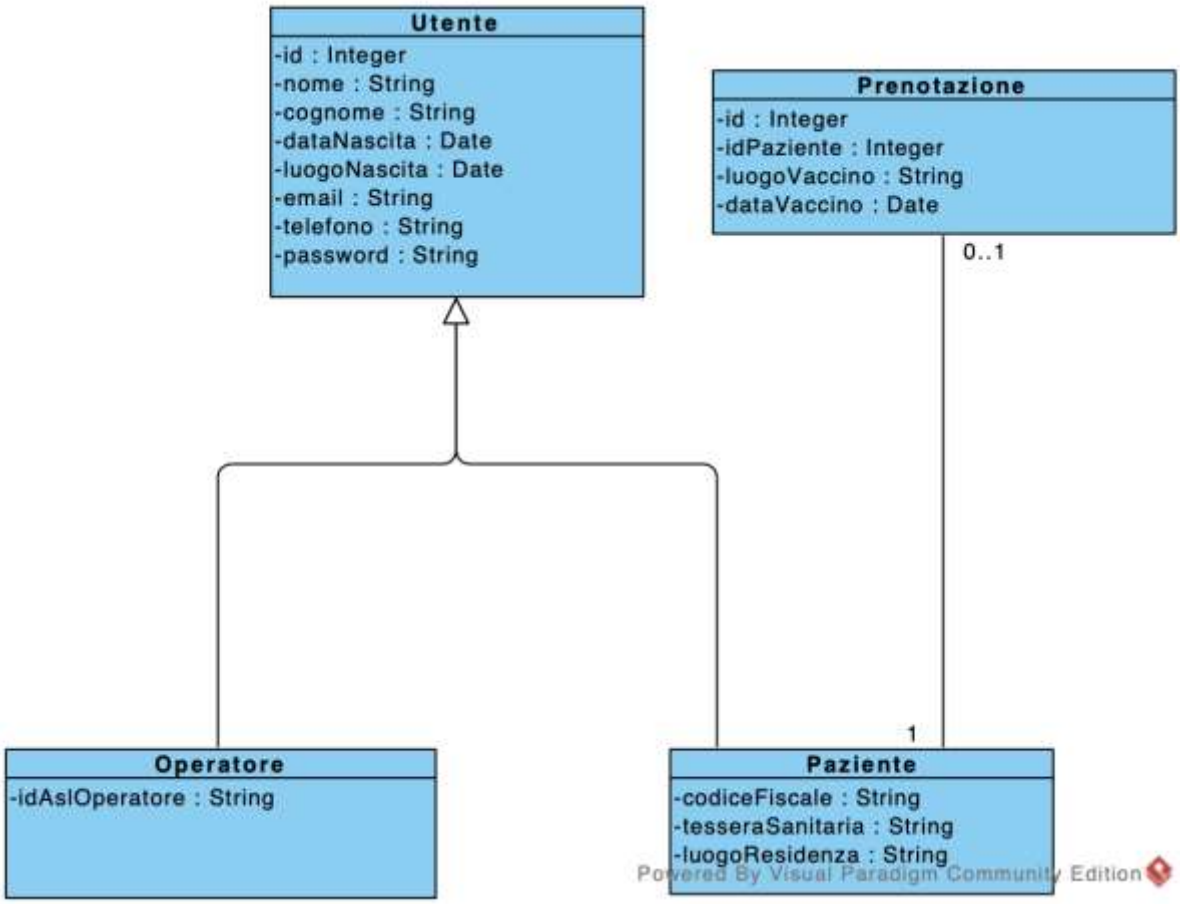


Figura 4.3: Diagramma delle classi prima iterazione

4.2.3 User Stories Primo Sprint

Durante la prima iterazione sono stati scelti dal product backlog le seguenti user stories che hanno costituito lo sprint backlog, queste rappresentano le funzionalità del sistema implementate:

- Registrazione
- Login
- Effettua prenotazione
- Visualizza prenotati
- Visualizza propria prenotazione

Si riportano per comodità le storie utente ed alcuni test di accettazione.

Storia Utente	
Titolo	Registrazione
Descrizione	*Come* Paziente *Voglio* registrarmi *In modo da* poter successivamente accedere ai servizi
Priorità	High
Story point estimate	9
Epic link	Gestione Utente
Etichette	Requisito Paziente

Tabella 4.1: Storia Utente: Registrazione

TEST ACCETTAZIONE 1:

- Registrazione con successo

  Dato un paziente non registrato
  Quando il paziente effettua la registrazione
  Allora viene creato l'account del paziente

Tabella 4.2: Registrazione: Test di accettazione 1

TEST ACCETTAZIONE 2:

- Registrazione fallita (Codice Fiscale già esistente)

  Dato un paziente già registrato
  Quando il paziente effettua la registrazione
  Allora viene restituito un messaggio di errore

Tabella 4.3: Registrazione: Test di accettazione 2

Storia Utente	
Titolo	Login
Descrizione	*Come* Paziente/Operatore *Io voglio* autenticarmi *In modo da* accedere alle funzionalità del sistema
Priorità	High
Story point estimate	5
Epic link	Gestione Utente
Etichette	Requisito Paziente Requisito Operatore

Tabella 4.4: Storia Utente: Login

TEST ACCETTAZIONE 1:

- Login con successo (Codice Fiscale e password corretti)

  Dato un paziente già registrato al sistema o un operatore
  Quando il paziente/operatore effettua il login correttamente
  Allora il paziente risulta autenticato e viene diretto alla pagina iniziale

Tabella 4.5: Login: Test di accettazione 1

TEST ACCETTAZIONE 2:

- Login fallito (Codice Fiscale non corretto)

  Dato un paziente non registrato al sistema
  Quando il paziente effettua il login
  Allora il paziente risulta NON autenticato e si riceve un messaggio di errore

Tabella 4.6: Login: Test di accettazione 2

TEST ACCETTAZIONE 3:

- Login fallito (Password non corretta)

  Dato un paziente registrato al sistema o un operatore
  Quando il paziente/operatore effettua il login inserendo una password scorretta
  Allora il paziente/operatore risulta NON autenticato e si riceve un messaggio di errore

Tabella 4.7: Login: Test di accettazione 3

Storia Utente	
Titolo	Effettua prenotazione
Descrizione	*Come* paziente *Io voglio* effettuare una prenotazione *In modo da* poter essere vaccinato
Priorità	High
Story point estimate	7
Epic link	Gestione Prenotazione
Etichette	Requisito Paziente

Tabella 4.8: Storia Utente: Effettua prenotazione

TEST ACCETTAZIONE:

- Prenotazione con successo

  Dato un paziente già registrato al sistema
  E che non ha ancora effettuato la prenotazione
  Quando il paziente effettua la prenotazione
  Allora il paziente risulta prenotato per la prima data disponibile

Tabella 4.9: Effettua prenotazione: Test di accettazione

Storia Utente	
Titolo	Visualizza prenotati
Descrizione	*Come* operatore *Io voglio* vedere la lista dei prenotati *In modo da* poter visualizzare i pazienti da vaccinare
Priorità	High
Story point estimate	4
Epic link	Gestione Prenotazione
Etichette	Requisito Operatore

Tabella 4.10: Storia Utente: Visualizza prenotati

TEST ACCETTAZIONE:

- Test con MOLTI prenotati presenti in lista

Dato un operatore autenticato
E un elenco di prenotazione già esistenti
Quando l'operatore visualizza i prenotati
Allora vengono restituite la lista dei pazienti prenotati

Tabella 4.11: Visualizza prenotati: Test di accettazione

Storia Utente	
Titolo	Visualizza propria prenotazione
Descrizione	*Come* paziente *Io voglio* visualizzare la mia prenotazione *In modo da* poter vedere la data, l'orario e il luogo dell'impegno
Priorità	High
Story point estimate	4
Epic link	Gestione Prenotazione
Etichette	Requisito Paziente

Tabella 4.12: Storia Utente: Visualizza propria prenotazione

TEST ACCETTAZIONE 1:

- Prenotazione visualizzata con successo

Dato un paziente già registrato al sistema
E che ha effettuato già la prenotazione
Quando il paziente visualizza la propria prenotazione
Allora vengono restituite le informazioni sull'appuntamento prenotato

Tabella 4.13: Visualizza propria prenotazione: Test di accettazione 1

TEST ACCETTAZIONE 2:

- Prenotazione non visualizzata (Perché NON effettuata)

Dato un paziente già registrato al sistema
E che NON ha effettuato già la prenotazione
Quando il paziente visualizza la propria prenotazione
Allora viene restituito un messaggio che indica che la prenotazione non è stata ancora effettuata

Tabella 4.14: Visualizza propria prenotazione: Test di accettazione 2

4.2.4 Sequence diagram

Di seguito sono riportati i sequence diagram delle funzionalità “Registrazione” ed “Effettua prenotazione”.
In particolare, il diagramma di sequenza della funzionalità “Registrazione” segue quanto indicato nella versione di analisi [3.5 Sequence Diagram], e sono stati gestiti i sistemi esterni mediante due stub.

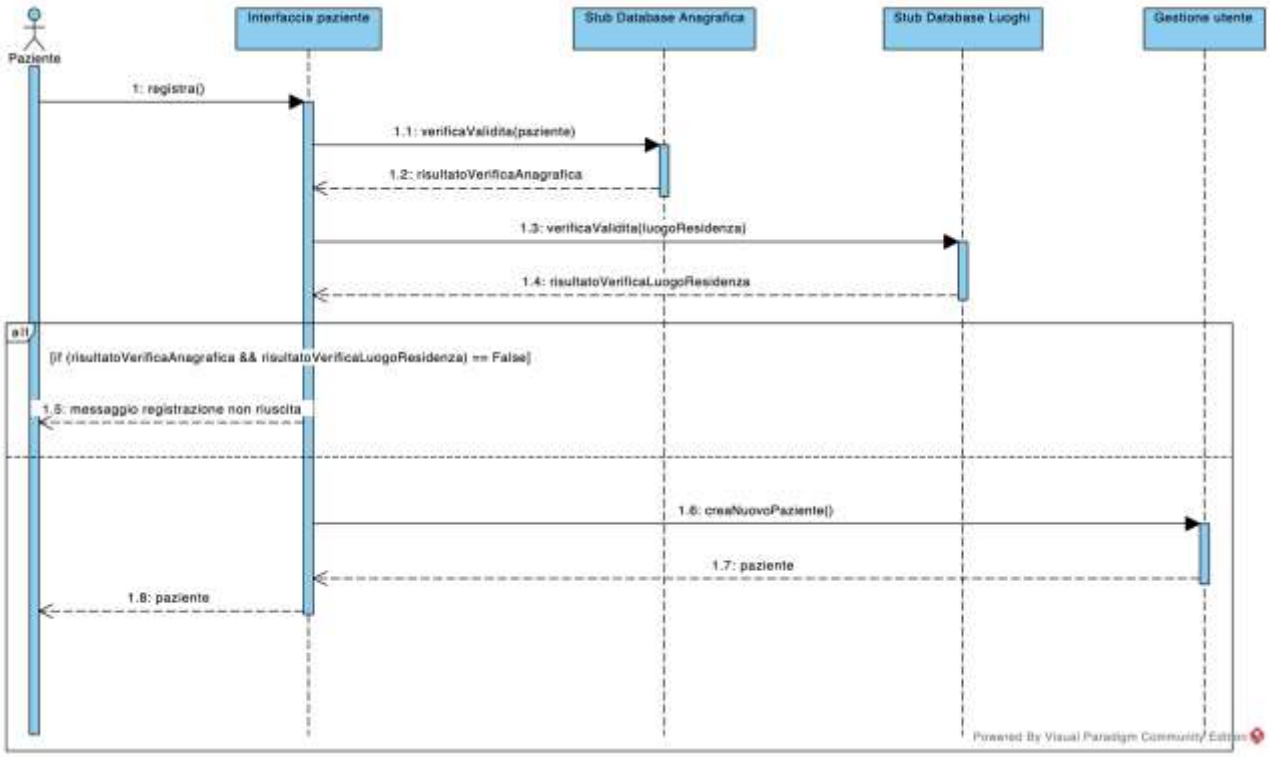


Figura 4.4: Sequence diagram: Registrazione prima iterazione

Anche il diagramma di sequenza della funzionalità “Effettua prenotazione” segue quanto indicato nella versione di analisi [3.5 Sequence Diagram], ed è stato gestito con un Mock la logica per la schedulazione dei turni.

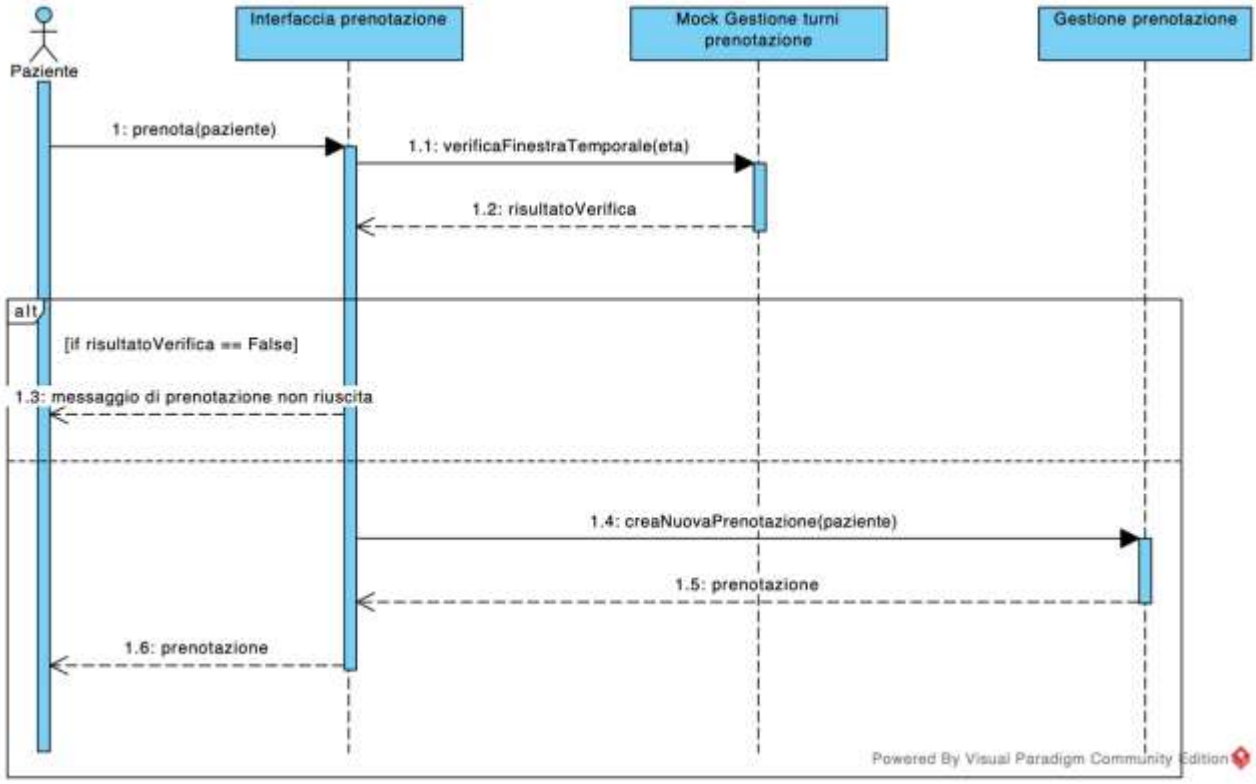


Figura 4.5: Sequence diagram: Effettua prenotazione prima iterazione

5. Implementazione

In questa sezione sono riportati tutti gli aspetti relativi all'implementazione del sistema.

5.1 Class Diagram

Di seguito traviamo il diagramma delle classi dettagliato del microservizio di prenotazione, dove in sostanza possediamo una sola classe "Prenotazione" che controlla il servizio gestendo per mezzo della classe "PrenotazioneDAO" le interazioni col Database.

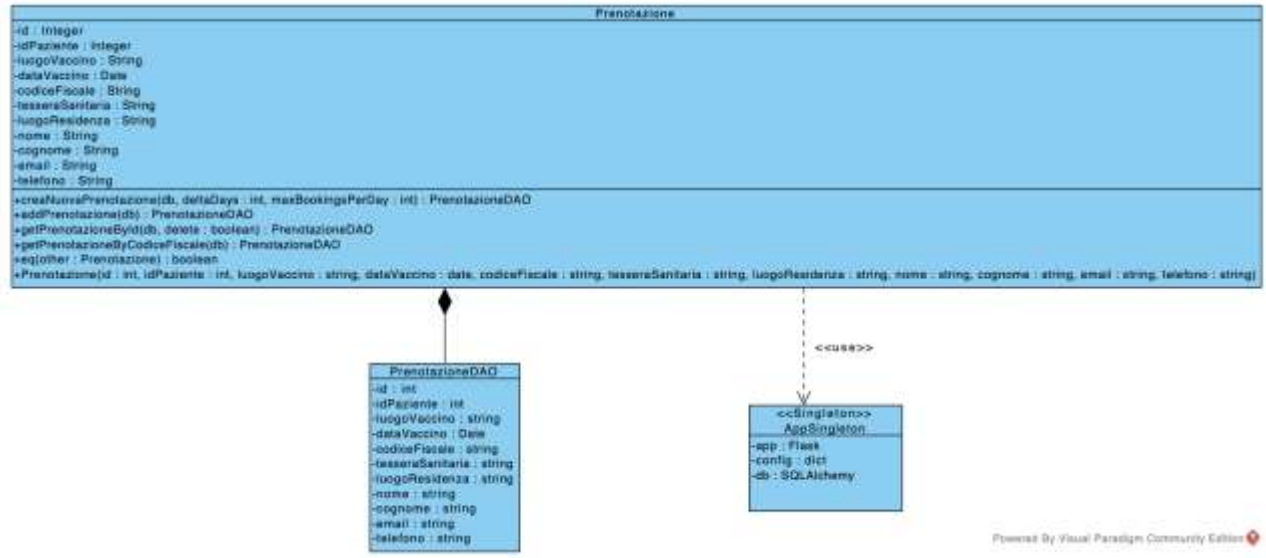


Figura5.1: Diagramma delle classi di implementazione della prima iterazione del microservizio gestione prenotazioni

Di seguito è riportato il diagramma delle classi del servizio di gestione utenti, più complesso del precedente prevede tre classi chiave Paziente, Operatore ed Utente.
Ogni oggetto gestisce un'interfaccia col database per messo dell'omonima classe DAO e le classi Paziente ed Operatore ereditano dalla classe Utente.

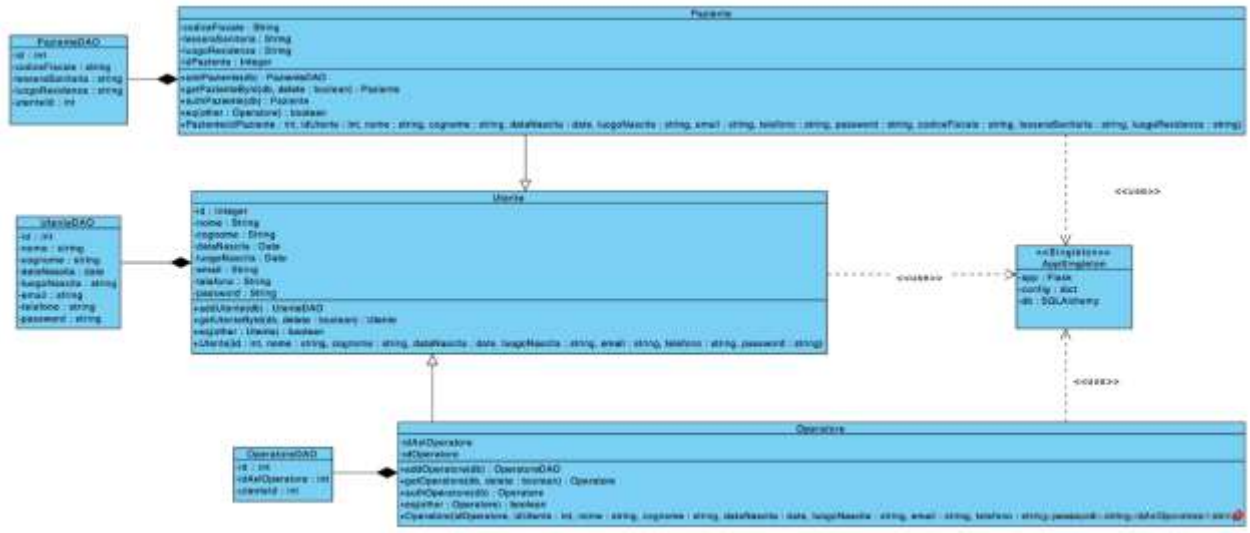


Figura 5.2: Diagramma delle classi di implementazione della prima iterazione del microservizio gestione utenti

5.2 Regole di business

Sono state fatte le seguenti assunzioni:

- Lo scheduling delle finestre temporali dei turni di prenotazione è gestito mediante un mock, il quale vede l'anno corrente e prende in ingresso l'età del paziente. I pazienti che hanno età 80+ li fa prenotare a partire dal primo gennaio dell'anno corrente (calcolato all'interno della funzione). I pazienti che hanno età 60+ li fa prenotare a partire dal primo marzo dell'anno corrente. I pazienti che hanno età 40+ li fa prenotare a partire dal primo maggio dell'anno corrente. I pazienti che hanno età 20+ li fa prenotare a partire dal primo luglio dell'anno corrente. I pazienti che hanno età 0+ li fa prenotare a partire dal primo settembre dell'anno corrente. Restituisce un boolean che indica che si può effettuare la prenotazione o no e, inoltre, restituisce la data a partire dalla quale il paziente può effettuare la prenotazione in entrambi i casi;
- Il luogo della vaccinazione è gestito mediante uno stub, che restituisce come luogo di vaccinazione il luogo di residenza del paziente che effettua la prenotazione;
- Per quanto riguarda il massimo numero di prenotati per un dato giorno e il deltaDays sono entrambi parametri configurabili.
  In particolare, il deltaDays è il parametro che impatta sulla data del vaccino: se D0 è il giorno in cui un paziente effettua una richiesta di prenotazione e DV è la data di vaccino che gli è assegnata dal sistema, allora risulterà che DV >= D0 + deltaDays.

5.3 Diagramma Relazionale

Di seguito si riporta il diagramma relazionale realizzato per descrivere la struttura del database.
In fase di implementazione è stato suddiviso il database in parti differenti, in modo tale che ogni microservizio presenti una propria base di dati alla quale afferisce.

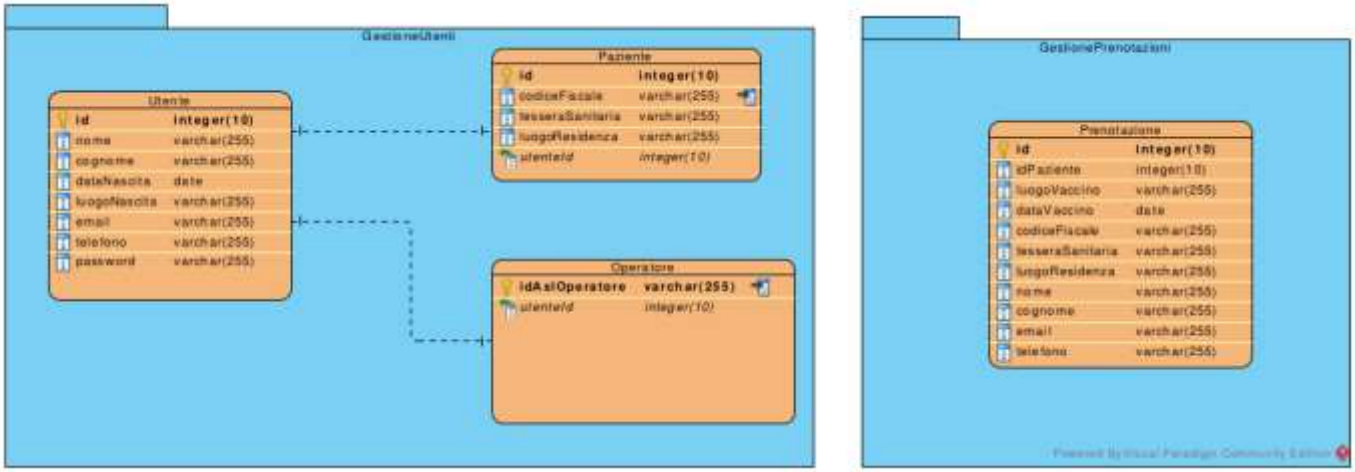


Figura 5.3: Diagramma relazionale

Si è scelto dunque di avere un database che gestisce gli utenti e dunque pazienti ed operatori che ereditano dalla classe utente ed un database che memorizza le prenotazioni.
Si noti che il database delle prenotazioni possiede attributi appartenenti alla classe utente essendo che per la logica architetturale il database del microservizio di prenotazione non può avere un riferimento ad un database di un altro microservizio.

5.4 Deployment Diagram

Per illustrare come l'applicazione potrebbe essere deployata su un sistema cloud è stato realizzato il seguente deployment diagram.

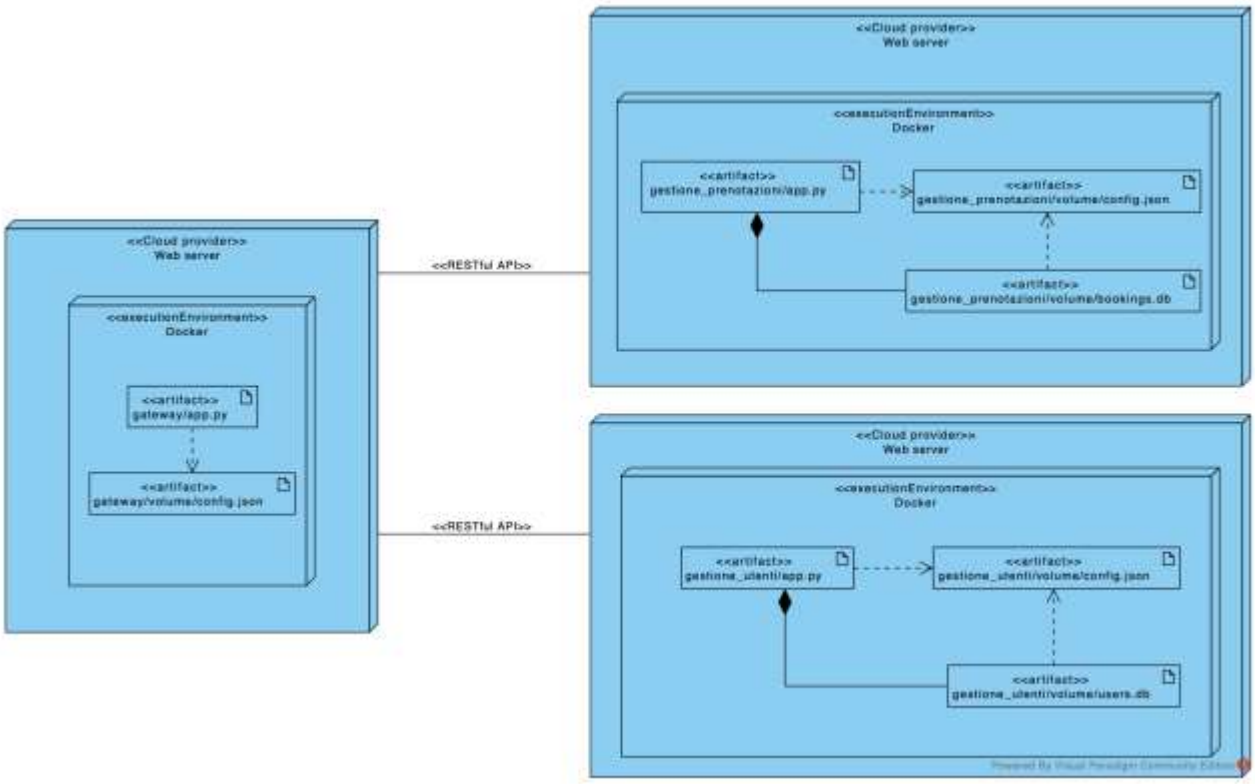


Figura 5.4: Diagramma di deployment

Nel diagramma si legge che un generico microservizio viene distribuito su un web server fornito da un cloud provider ed il suo execution environment è realizzato mediante un container docker.
All'interno di tale ambiente il top module (controller del microservizio) è il modulo app.py, che dipende da un file di configurazione scritto in json.
Per i microservizi che prevedono un database il top module contiene strettamente la propria base dati.
I microservizi dialogano tra loro mediante RESTful API.

5.5 Build, Esecuzione & Test

I comandi presenti nei paragrafi [5.5 Build, Esecuzione & Test] e [5.6 Configurazione] assumono che l'utente usi un sistema Linux.

È possibile effettuare la build del progetto usando il Makefile fornito, oppure è possibile fare la build di ciascun microservizio singolarmente con il relativo docker-compose.yml file.
Nel primo caso, da terminale, nella root directory del progetto:

- Build: make build
- Esecuzione: make up
- Visualizzazione dei log: make logs
- Stop del sistema: make stop

e così via.

Per assicurare la corretta comunicazione tra i microservizi, è necessario configurare il gateway. Vedere la sezione "Configurazione".
Se si desidera effettuare la build e l'esecuzione di un singolo servizio, ad esempio gestione_utenti, allora da terminale, a partire dalla root directory del progetto:

cd gestione_utenti
docker-compose up --build -d
docker-compose logs

Ciascun Dockerfile esegue un wrapper script, che esegue tutti i test d'unità per i relativi microservizi, e se tutti i test passano, allora l'app è eseguita.

Per eseguire i test d'integrazione, bisogna assicurarsi che i 3 microservizi siano up e configurati correttamente, ovvero il gateway deve correttamente fare da proxy verso gli altri servizi (se questo non è vero, allora i primi test d'integrazione falliscono, perché essi effettuano dei "ping" verso il sistema).
Dopo essersi assicurati di ciò, per eseguire i test è necessario aprire di nuovo il terminale nella root directory del progetto ed eseguire i seguenti comandi:

cd integration_test
python run_all_tests.py

Se non c'è nessun errore, allora tutti i test sono passati, e viene fornito a terminale il numero di test eseguiti ed il tempo che è stato necessario ad eseguirli.

5.6 Configurazione

Ci sono tre file di configurazione:

- gateway/volume/config.json
- gestione_prenotazioni/volume/config.json
- gestione_utenti/volume/config.json

Cominciamo dall'ultimo: gestione_utenti.

```
{
	"port": 5000,
	"debug": true,
	"db": "sqlite:////usr/src/app/volume/users.db",
	"secret": "dont_use_this_in_production",
	"session_validity_days": 7
}
```

Il parametro "port" è la porta sulla quale l'applicazione Flask sarà in ascolto: se la si cambia, allora è necessario cambiare anche la porta nel relativo file docker-compose.yml (quella sulla destra, perché è del tipo <porta dell'host> : <porta del docker>), altrimenti non sarà possibile raggiungere l'applicazione.
Il parametro "debug" indica a Flask se deve eseguire in modalità debug oppure no.
Il parametro "db" è l'URL del database: non si è forzati ad usare un DB SQLite perché è stato utilizzato il componente SQLAlchemy per realizzare l'Object Relational Mapping; tuttavia, c'è del codice back-end che assume di avere un DB serverless, ovvero per controllare se il DB è già stato inizializzato, vede se è presente nel path specificato nel filesystem.
Il parametro "secret" è utilizzato per firmare i cookie di session usando un HMAC.
Il parametro "session_validity_days" impatta sulla data di scadenza dei cookie.

Vediamo ora la configurazione di gestione_prenotazioni.

```
{
	"port": 5000,
	"debug": true,
	"db": "sqlite:////usr/src/app/volume/bookings.db",
	"secret": "dont_use_this_in_production",
	"session_validity_days": 7,
	"deltaDays": 3,
	"maxBookingsPerDay": 5
}
```

Il parametro "secret" dev'essere lo stesso tra gestione_utenti e gestione_prenotazioni, ovvero è un segreto condiviso (shared secret).

I parametri "deltaDays" e "maxBookingsPerDay" sono legati alla logica applicativa.
Il parametro "deltaDays" impatta lo scheduling della dataVaccino: se D_0 è il giorno in cui un Paziente effettua una richiesta di Prenotazione e D_V è l'effettiva dataVaccino restituita dal sistema, allora risulterà che:

D_V >= D_0 + deltaDays

Il parametro "maxBookingsPerDay" è il massimo numero di prenotazioni che possono essere schedulate in ciascun giorno.

Vediamo infine la configurazione del gateway.

```
{
        "port": 5000,
        "debug": true,
        "gestione_utenti_url": "http://172.21.0.1:5001",
        "gestione_prenotazioni_url": "http://172.21.0.1:5002",
        "requests_timeout": 2
}
```

Poiché il gateway lavora da proxy, ricevendo le richieste dai client ed inoltrandole agli altri microservizi, esso deve sapere dove si trovano gli altri microservizi nella rete.
Esso funziona sia in reti locali sia su Internet: bisogna semplicemente specificare i giusti URL.
Se si effettua la build in locale di tutti i microservizi usando il Makefile fornito, per scrivere i giusti URL bisogna sapere su quali IP locali i microservizi si trovano e su quali porte ascoltano, ponendosi il problema relativo al fatto che ciascun Docker ha la sua interfaccia di rete.
Eseguire i seguenti comandi:

```
make build
make up
make logs
ifconfig
```

Nei logs, è possibile osservare, tra le varie linee, qualcosa di questo tipo:

```
...
gateway_1  |  * Running on http://172.21.0.2:5000/ (Press CTRL+C to quit)
...
```

Dopo aver effettuato il comando "ifconfig", si vedono le varie interfacce di rete, tra cui bisogna cercare quella in comune con il gateway (stesso indirizzo di broadcast), che in questo caso sarà la seguente:

...

br-ef26eaab39a7: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 172.21.0.1 netmask 255.255.0.0 broadcast 172.21.255.255
inet6 fe80::42:17ff:fe2b:23c8 prefixlen 64 scopeid 0x20<link>
ether 02:42:17:2b:23:c8 txqueuelen 0 (Ethernet)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

...

Questo vuol dire che il gateway può accedere alla rete dell'host usando l'indirizzo IP 172.21.0.1; gli altri due microservizi saranno accessibili DAL GATEWAY sulla rete dell'host mediante tale IP, e le porte da specificare nell'URL sono quelle specificate nei file docker-compose.yml dei rispettivi microservizi (la parte sinistra delle ports, ricordando che è del tipo <porta dell'host>:<porta del docker>).
Dunque, ancora, quando si usa il Makefile, gli URL da scrivere nel file di configurazione si ottengono usando come indirizzo IP per entrambi l'indirizzo IP della rete host visto dall'interfaccia di rete del gateway, ed usando come porte quelle sulla parte sinistra dei rispettivi docker-compose.yml files.
Adesso si è pronti a configurare i parametri "gestione_utenti_url" e "gestione_prenotazioni_url".
Il parametro "requests_timeout" è il timeout, espresso in secondi, relativo alle richieste che il gateway effettua verso i microservizi per i quali opera da proxy, in seguito a richieste ricevute dal client.

6. Test

In questo capitolo viene descritta la fase di testing effettuata.
Sono stati progettati dei test di unità per ciascun microservizio e dei test di integrazione dell'intero sistema.

6.1 Test di unità

I test di unità sono eseguiti test per verificare il corretto funzionamento dei tre microservizi.

6.1.1 Gestione Utenti

Di seguito saranno spiegati i test per l'unità di gestione utenti:

1) Primo test, sono state testate le operazioni di inserimento ed estrazione di una istanza di utente dal database del microservizio di gestione utente;

2) Secondo test, è stato replicato il test numero 1 inserendo ed estraendo istanze multiple di utente verificando che fossero diversi (non possono esistere due utenti con gli stessi campi della tupla);

3) Terzo test, è stato testato il caso di prelievo di un utente non presente nel database;

4) Quarto test, è stato verificato che la richiesta di un utente estratto dal database risultasse di esito negativo;

5) Sono stati replicati i primi quattro test sia per il database del paziente e dell'operatore;

6) Tredicesimo test, è stato verificato che il database delle password hashate non risultasse corrotto.

6.1.2 Gestione Prenotazioni

Di seguito saranno spiegati i test per l'unità di gestione pazienti:

1) Primo test, è stato testato l'inserimento ed il prelievo di una prenotazione usando come chiave l'id della prenotazione;

2) Secondo test, è stato testato l'inserimento ed il prelievo di una prenotazione usando come chiave il codice fiscale del paziente;

3) Terzo test, è stato replicato il test numero 1 inserendo ed estraendo istanze multiple di prenotazione verificando che fossero diverse (non possono esistere due prenotazioni con gli stessi campi della tupla);

4) Quarto test, è stato verificato che il prelievo di una prenotazione non presente nel database risultasse di esito negativo;

5) Quinto test, è stato verificato che il prelievo di una prenotazione estratta dal database risultasse di esito negativo;

6) Sesto test, è stato verificato che non è possibile inserire due prenotazioni uguali nel database;

7) Settimo test, è stato verificato che il prelievo di una prenotazione dal database vuoto risultasse di esito negativo;

8) Ottavo test, è stato verificato il prelievo di istanze multiple di prenotazione verificando che fossero ordinate in ordine ascendente secondo la data;

9) Nono test, è stato verificato che una prenotazione appena creata abbia una data lecita (la data della prenotazione deve essere successiva alla data odierna);

10) Decimo test, è stato verificato che se in un certo giorno sono presenti il numero massimo di pazienti da vaccinare, la data della prenotazione appena conseguita slitta al giorno successivo.

6.2 Test di integrazione

Di seguito saranno riportati i test per integrare i tre microservizi:

1) Primo test, verifica microservizio gateway attivo;

2) Secondo test, verifica microservizio gestione utenti attivo;

3) Terzo test, verifica microservizio gestione prenotazioni attivo;

4) Quarto test, è stata testata la funzionalità di registrazione con successo di un paziente;

5) Quinto test, è stata testata la funzionalità di registrazione con fallimento di un paziente, causa uno o più campi necessari vuoti;

6) Sesto test, è stata testata la funzionalità di registrazione con fallimento di un paziente, causa formato data sbagliato;

7) Settimo test, è stata testata la funzionalità di registrazione con fallimento di un paziente, causa paziente già presente nel database;

8) Ottavo test, è stata testata la funzionalità di login di un paziente già loggato nel sito (verifica mantenimento cookie di login);

9) Nono test, è stata testata la funzionalità di login con successo di un paziente;

10) Decimo test, è stata testata la funzionalità di login con fallimento di un paziente, causa password sbagliata;

11) Undicesimo test, è stata testata la funzionalità di login con fallimento di un paziente, causa codice fiscale sbagliata;

12) Dodicesimo test, è stata testata la funzionalità di login con fallimento di un paziente, causa campo vuoto;

13) Tredicesimo test, è stata testata la funzionalità di login di un operatore già loggato nel sito (verifica mantenimento cookie di login);

14) Quattordicesimo test, è stata testata la funzionalità di login con successo di un operatore;

15) Quindicesimo test, è stata testata la funzionalità di logout con successo per un paziente;

16) Sedicesimo test, è stata testata la funzionalità di logout con successo per un operatore;

17) Quindicesimo test, è stata testata la funzionalità di logout con fallimento nel caso di nessuna sessione aperta;

18) Diciottesimo test, è stata testata la funzionalità di vedi prenotazione con fallimento, causa mancato login;

19) Diciannovesimo test, è stata testata la funzionalità di vedi prenotazione con notifica di prenotazione non effettuata;

20) Ventesimo test, è stata testata la funzionalità di effettua prenotazione con successo;

21) Ventunesimo test, è stata testata la funzionalità di effettua prenotazione con fallimento, causa prenotazione già effettuata;

22) Ventiduesimo test, è stata testata la funzionalità di visualizza prenotazione con successo;

23) Ventitreesimo test, è stata testata la funzionalità di visualizza lista prenotati con successo.

7. Documentazione RESTful API

Di seguito è riportata la documentazione della RESTful API complessiva fornita dai microservizi, ovvero quella esposta dal gateway. Tale documentazione è stata fatta utilizzando Swagger e dunque è stata esportata da lì, per essere riportata nel presente documento.

RESTful API del sistema

API del sistema di prenotazione di vaccini
Version: 1.0.0
GPL 3.0
https://www.gnu.org/licenses/gpl-3.0.html

Access

Methods

[Jump to Models]

Table of Contents

Operatore

GET /api/bookings

Up

Mostra lo stato del servizio (**apiBookings**)

Se non sei autenticato, ti dice se il servizio è disponibile; altrimenti ti fornisce informazioni sull'utente con il quale sei autenticato

Produces

- application/json

Responses

200
Messaggio che dipende dall'utente che fa la richiesta InfoMsg
400
Errore imprevisto ErrorMsg

GET /api/users

Up

Mostra lo stato del servizio (**apiUsers**)

Se non sei autenticato, ti dice se il servizio è disponibile; altrimenti ti fornisce informazioni sull'utente con il quale sei autenticato

Produces

- `application/json`

Responses
200
Messaggio che dipende dall'utente che fa la richiesta InfoMsg
400
Errore imprevisto ErrorMsg

GET /api/gateway/status

Up

Ritorna lo stato del gateway (**gatewayStatus**)

Il gateway è il microservizio che funge da dispatcher per gli altri microservizi, ovvero è quello con cui il frontend interagisce direttamente; questo endpoint serve per sapere se il gateway è up

Produces

- `application/json`

Responses
200
Il servizio è up InfoMsg
400
Errore imprevisto ErrorMsg

POST /api/users/login

Up

Autenticazione di un Paziente oppure di un Operatore (**login**)

Permette ad un Utente del sistema di ottenere una sessione, ritorna un messaggio di errore se il login non ha successo e ritorna un messaggio di info se l'utente è già autenticato; il messaggio di richiesta deve contenere il campo "idAslOperatore" se il campo "is_operatore" è true, altrimenti deve contenere il campo "codiceFiscale"; il messaggio di risposta riportato in questa documentazione è relativo al login per il Paziente, ma ovviamente esso cambia se il login è effettuato da un Operatore

Consumes

- `application/json`

Request body

body LoginMsg (optional)
Body Parameter —

Produces

- `application/json`

Responses
200
Utente già autenticato InfoMsg
201
Login effettuato con successo SuccessMsgUtente
400
Errore relativo all'input oppure errore nell'applicazione ErrorMsg

---

GET /api/users/logout

Up

Effettua il logout (**logout**)

Elimina il cookie di sessione, ritorna un messaggio di info se è stato effettuato il logout, oppure ritorna un messaggio di errore se l'utente che ha fatto la richiesta non era autenticato

Produces

- `application/json`

Responses

200
Messaggio di logout InfoMsg
400
Utente non autenticato oppure errore nell'applicazione ErrorMsg

---

GET /api/bookings/view

Up

Permette ad un Paziente di visualizzare la propria Prenotazione, oppure ad un Operatore di visualizzare la lista delle prenotazioni future (**viewBookings**)

Gestisce il caso in cui la Prenotazione non sia ancora stata effettuata con un messaggio di info, i vari casi di errore ad esempio dell'utente non autenticato con un messaggio di errore, e con messaggi di successo ritorna o la Prenotazione del Paziente o la lista di prenotazioni future che l'Operatore deve visualizzare; nella documentazione riportiamo il caso in cui il Paziente ottiene una Prenotazione, mentre nel caso dell'Operatore si avrà nel campo 'data' una lista di oggetti di tipo Prenotazione

Produces

- `application/json`

Responses

200
Prenotazione non ancora effettuata InfoMsg
201
Prenotazione presente SuccessMsgPrenotazione
400
Utente non autenticato oppure errori applicativi ErrorMsg

Paziente

GET /api/bookings

Up

Mostra lo stato del servizio (**apiBookings**)

Se non sei autenticato, ti dice se il servizio è disponibile; altrimenti ti fornisce informazioni sull'utente con il quale sei autenticato

Produces

- `application/json`

Responses

200
Messaggio che dipende dall'utente che fa la richiesta InfoMsg
400
Errore imprevisto ErrorMsg

GET /api/users

Up

Mostra lo stato del servizio (**apiUsers**)

Se non sei autenticato, ti dice se il servizio è disponibile; altrimenti ti fornisce informazioni sull'utente con il quale sei autenticato

Produces

- `application/json`

Responses

200
Messaggio che dipende dall'utente che fa la richiesta InfoMsg
400
Errore imprevisto ErrorMsg

GET /api/gateway/status

Up

Ritorna lo stato del gateway (**gatewayStatus**)

Il gateway è il microservizio che funge da dispatcher per gli altri microservizi, ovvero è quello con cui il frontend interagisce direttamente; questo endpoint serve per sapere se il gateway è up

Produces

- `application/json`

Responses

200
Il servizio è up InfoMsg
400
Errore imprevisto ErrorMsg

POST /api/users/login

Up

Autenticazione di un Paziente oppure di un Operatore (**login**)

Permette ad un Utente del sistema di ottenere una sessione, ritorna un messaggio di errore se il login non ha successo e ritorna un messaggio di info se l'utente è già autenticato; il messaggio di richiesta deve contenere il campo "idAslOperatore" se il campo "is_operatore" è true, altrimenti deve contenere il campo "codiceFiscale"; il messaggio di risposta riportato in questa documentazione è relativo al login per il Paziente, ma ovviamente esso cambia se il login è effettuato da un Operatore

Consumes

- `application/json`

Request body

body LoginMsg (optional)
Body Parameter —

Produces

- `application/json`

Responses

200
Utente già autenticato InfoMsg
201

Login effettuato con successo SuccessMsgUtente
400
Errore relativo all'input oppure errore nell'applicazione ErrorMsg

---

GET /api/users/logout

Up

Effettua il logout (**logout**)

Elimina il cookie di sessione, ritorna un messaggio di info se è stato effettuato il logout, oppure ritorna un messaggio di errore se l'utente che ha fatto la richiesta non era autenticato

Produces

- `application/json`

Responses
200
Messaggio di logout InfoMsg
400
Utente non autenticato oppure errore nell'applicazione ErrorMsg

---

POST /api/bookings/newBooking

Up

Permette di creare una nuova Prenotazione (**newBooking**)

Utilizza i dati del cookie di sessione per creare una nuova Prenotazione; si effettuano vari controlli, ad esempio sulla finestra temporale, e se essi passano tutti allora viene ritornato l'oggetto Prenotazione creato, altrimenti viene ritornato un messaggio di errore

Return type
SuccessMsgPrenotazione

Example data
Content-Type: application/json

```
{
  "data" : {
    "tesseraSanitaria" : "tesseraSanitaria",
    "cognome" : "cognome",
    "dataVaccino" : "dataVaccino",
    "nome" : "nome",
    "id" : 0,
    "luogoVaccino" : "luogoVaccino",
    "telefono" : "telefono",
    "codiceFiscale" : "codiceFiscale",
    "idPaziente" : 6,
    "email" : "email",
    "luogoResidenza" : "luogoResidenza"
  },
  "success" : "success"
}
```

Produces

- `application/json`

Responses
201
Prenotazione creata con successo SuccessMsgPrenotazione
400

Errore sull'input oppure utente non abilitato ad effettuare una Prenotazione, oppure errore nell'applicazione ErrorMsg

---

POST /api/users/register

Up

Registrazione di un nuovo Paziente (**registrazione**)

Registra un nuovo paziente, gestisce il caso in cui a fare la richiesta sia un utente già autenticato con un messaggio di info

Consumes

This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

body Paziente (optional)
Body Parameter — Dati del Paziente da registrare

Produces

- `application/json`

Responses

200
Utente già autenticato InfoMsg
201
Registrazione effettuata con successo SuccessMsgUtente
400
Errore relativo all'input oppure errore nell'applicazione ErrorMsg

---

GET /api/bookings/view

Up

Permette ad un Paziente di visualizzare la propria Prenotazione, oppure ad un Operatore di visualizzare la lista delle prenotazioni future (**viewBookings**)

Gestisce il caso in cui la Prenotazione non sia ancora stata effettuata con un messaggio di info, i vari casi di errore ad esempio dell'utente non autenticato con un messaggio di errore, e con messaggi di successo ritorna o la Prenotazione del Paziente o la lista di prenotazioni future che l'Operatore deve visualizzare; nella documentazione riportiamo il caso in cui il Paziente ottiene una Prenotazione, mentre nel caso dell'Operatore si avrà nel campo 'data' una lista di oggetti di tipo Prenotazione

Produces

- `application/json`

Responses

200
Prenotazione non ancora effettuata InfoMsg
201
Prenotazione presente SuccessMsgPrenotazione
400
Utente non autenticato oppure errori applicativi ErrorMsg

Models

[Jump to Methods]

Table of Contents

1. ErrorMsg
2. InfoMsg
3. LoginMsg
4. Operatore
5. Paziente
6. Prenotazione
7. SuccessMsg
8. SuccessMsgUtente
9. SuccessMsgPrenotazione

ErrorMsg

Up

error (optional)
String

InfoMsg

Up

info (optional)
String

LoginMsg

Up

codiceFiscale (optional)
String

idAslOperatore (optional)
String

password (optional)
String

is_operatore (optional)
Boolean

Operatore

Up

idOperatore (optional)
Integer

idUtente (optional)
Integer

idAslOperatore (optional)
String

nome (optional)
String

cognome (optional)
String

dataNascita (optional)
date format: date

luogoNascita (optional)
String

email (optional)
String

telefono (optional)
String

password (optional)
String

Paziente

[Up](#)

nome
String

cognome
String

dataNascita
date format: date

luogoNascita
String

email
String

telefono
String

password
String

codiceFiscale
String

tesseraSanitaria
String

luogoResidenza
String

Prenotazione

[Up](#)

id (optional)
Integer

idPaziente (optional)
Integer

luogoVaccino (optional)
String

dataVaccino (optional)
String

codiceFiscale (optional)
String

tesseraSanitaria (optional)
String

luogoResidenza (optional)
String

nome (optional)
String

cognome (optional)
String

email (optional)
String

telefono (optional)

String

SuccessMsg

[Up](#)

success (optional)
String

data (optional)
Object

SuccessMsgUtente

[Up](#)

success (optional)
String

data (optional)
Paziente

SuccessMsgPrenotazione

[Up](#)

success (optional)
String

data (optional)
Prenotazione