

Osteoarthritis Portal
Applicazioni e Servizi Web

Alessandro Magnani^{*1}, Andrea Matteucci^{†1}, and Simone Montanari^{‡1}

¹Università di Bologna

15 Gennaio 2025

^{*}alessandro.magnani18@unibo.it
[†]andrea.matteucci5@unibo.it
[‡]simone.montanari14@unibo.it

Indice

1	Introduzione	7
2	Requisiti	8
2.1	Obiettivo del sistema	8
2.2	Panoramica generale	8
2.3	Requisiti funzionali	8
3	Design	10
3.1	Obiettivi di design	10
3.2	Target User Analysis	10
3.2.1	Medici	10
3.2.2	Pazienti	11
3.2.3	Sfide comuni e soluzioni progettuali	12
3.3	Mockup	12
3.4	Analisi e modello del dominio	15
3.5	Scelte progettuali chiave	16
3.6	Esperienza utente (UI/UX)	18
3.6.1	Struttura delle pagine e dei flussi di navigazione	18
3.6.2	Scelte di stile e palette di colori	18
3.6.3	Adesione ai principi di accessibilità e responsive design	20
3.6.4	Feedback e interazioni in tempo reale	20
3.7	Scalabilità e modularità	22
3.7.1	Scalabilità del sistema	22
3.7.2	Modularità del sistema	23
3.7.3	Flessibilità nell'integrazione di nuove tecnologie	23
3.7.4	Gestione delle risorse e ottimizzazione dei costi	24
3.8	Diagrammi tecnici	24
4	Tecnologie	27
4.1	Linguaggi utilizzati	27
4.2	Tecnologie lato frontend	27
4.2.1	Node.js	27
4.2.2	Vue.js	27
4.2.3	Axios	28
4.3	Backend	28
4.3.1	Flask per la gestione delle API REST	28
4.4	Autenticazione	28
4.4.1	Firebase Authentication	29
4.4.2	Firestore Database	29
4.5	Archiviazione	29
4.6	Containerizzazione e Deployment	29
4.6.1	Docker, Containerizzazione e Orchestrazione	30
4.6.2	Deployment su Google Compute Engine	30
4.7	Integrazione del Modello di Rete Neurale Pre-addestrato	30
4.8	Github	30
4.9	Addestramento periodico del modello	30

5 Codice	32
5.1 Struttura	32
5.1.1 Struttura del Database	32
5.1.2 Struttura del sistema: moduli principali	34
5.1.2.1 Modulo Core	36
5.1.2.2 Modulo Utility	37
5.1.3 Modulo Controller	39
5.1.3.1 Modulo Factory	42
5.1.3.2 Modulo Routing	44
5.1.3.3 Modulo Config	45
5.1.3.4 Modulo UI	46
5.1.3.5 Modulo Service	49
5.2 Docker	50
5.2.1 Dockerfile per il frontend	50
5.2.2 Dockerfile per il backend	51
5.2.3 Orchestrazione dei container: docker-compose.yml	52
6 Test	53
6.1 Testing delle funzionalità	53
6.1.1 Testing automatizzato	54
6.2 Testing delle prestazioni e della sicurezza	56
6.3 Testing dell'Esperienza Utente (UX)	57
6.4 Esiti e analisi critica	58
7 Deployment	59
7.1 Creazione delle immagini Docker	59
7.2 Tagging delle immagini Docker	59
7.3 Pubblicazione su Docker Hub	59
7.4 Deploy del sistema con Docker Compose	60
8 Esempi di utilizzo	61
8.1 Registrazione e Login	61
8.2 Schermata di benvenuto e Barra di navigazione	66
8.3 Dashboard	67
8.4 Caricamento e Predizione	67
8.5 Visualizzazione Radiografie	69
8.6 Pianificazione di Attività e Notifiche	69
9 Conclusioni	72

Elenco delle figure

1	Mockup della piattaforma, progettata per una navigazione intuitiva e una chiara visualizzazione dei dati clinici.	12
2	Mockup della dashboard del medico	13
3	Mockup per il caricamento e la predizione delle radiografie.	14
4	Mockup del calendario delle attività comprensivo al suo interno di tutte le informazioni riguardanti radiografie e operazioni per ogni giorno.	14
5	Mockup della sezione notifiche.	15
6	Entità del dominio e le loro interazioni.	16
7	Diagramma del flusso del sistema. La WGAN-GP attualmente non è integrata. .	18
8	Palette di colori selezionata per il progetto: i colori bianco e grigio sono stati scelti per lo sfondo, mentre i colori più vivaci offrono una buona contrapposizione, facilmente visibile per chiunque.	19
9	Differenze tra la versione desktop e la versione mobile.	20
10	Differenze tra la versione desktop e la versione mobile.	21
11	Struttura di Firestore. Permette di espandersi in modo dinamico a supporto di una crescita illimitata di dati e utenti.	22
12	Struttura di autenticazione di Firestore. Permette di aggiungere nuovi utenti senza compromettere le performance.	23
13	UML - Registrazione di un nuovo utente.	24
14	UML - Login.	25
15	UML - Predizione.	26
16	UML - Visualizzazione radiografie.	26
17	Struttura del sistema	32
18	Schema Entity/Relationship del Database	33
19	Diagramma delle dipendenze	35
20	Diagramma di sequenza: modulo Core.	36
21	Diagramma delle classi: modulo Controller, ogni classe è indipendente e si occupa di un dominio differente, totale separazione delle responsabilità.	41
22	Diagramma delle attività: modulo Controller.	42
23	Diagramma di flusso: modulo factory	44
24	Diagramma di flusso: modulo Routing	45
25	Diagramma di sequenza: modulo UI.	47
26	Diagramma di sequenza: modulo UI.	49
27	Costruzione dell'immagine Docker per il frontend	59
28	Costruzione dell'immagine Docker per il backend	59
29	Pubblicazione dell'immagine Docker per il frontend	60
30	Pubblicazione dell'immagine Docker per il backend	60
31	Deploy del sistema.	60
32	Schermata iniziale.	61
33	Schermate della registrazione.	61
34	Processo di verifica dell'email.	62
35	Processo di login.	63
36	Email non verificata.	63
37	Errori nell'inserimento della password.	64
38	Reset della password.	65
39	Schermata di benvenuto.	66
40	Differenze nelle barre di navigazione.	66

41	Dashboard per medici e pazienti.	67
42	Caricamento di una radiografia.	67
43	Predizione dell'osteoartrite.	68
44	Visualizzazione radiografie medici.	69
45	Visualizzazione radiografie pazienti.	69
46	Vista del calendario lato pazienti.	70
47	Vista del calendario lato medici.	70
48	Processo di pianificazione di un'attività e ricezione notifica.	71

Listings

1	File <code>app.py</code>	37
2	File <code>firestore_utils.py</code>	38
3	File <code>gcs_utils.py</code>	38
4	File <code>email_utils.py</code>	39
5	File <code>model_utils.py</code>	39
6	File <code>auth_controller.py</code>	39
7	File <code>user_controller.py</code>	40
8	File <code>radiograph_controller.py</code>	40
9	File <code>operation_controller.py</code>	40
10	File <code>notification_controller.py</code>	40
11	File <code>manager_factory.py</code>	43
12	Rotte definite nel file <code>api_routes.py</code>	44
13	File <code>app_config.py</code>	46
14	Esempio di componente Vue (1/2)	47
15	Esempio di componente Vue (2/2)	48
16	Esempi di operazioni del file <code>api_service.js</code>	49
17	Corpo della funzione <code>getPatientsFromDoctor()</code>	50
18	Dockerfile per il frontend	51
19	Dockerfile per il backend	51
20	<code>docker-compose.yml</code>	52
21	Contenuto del file <code>conftest.py</code>	54
22	Esempio di test	55

1 Introduzione

La salute è un pilastro della società e ogni innovazione in questo ambito rappresenta un'opportunità per migliorare la qualità della vita. Questo progetto nasce con l'ambizione di trasformare il modo in cui medici e pazienti affrontano l'osteoartrite, una patologia diffusa e spesso debilitante che colpisce le ginocchia, proponendo una piattaforma digitale capace di integrare tecnologia e scienza medica per offrire strumenti di diagnosi e supporto decisionale.

Il sistema consente a medici e pazienti di accedere a una gestione innovativa delle radiografie. I pazienti possono caricare e visualizzare le proprie radiografie mentre i medici hanno la possibilità di analizzare le immagini dei loro assistiti e ottenere, tramite intelligenza artificiale, predizioni approfondite sullo stato della malattia. Non solo: la piattaforma è in grado di determinare con precisione il grado di severità dell'osteoartrite e di evidenziare i punti più critici, suggerendo quando potrebbe essere necessario un intervento chirurgico.

Un aspetto distintivo è l'integrazione di un sistema di feedback: gli utenti possono valutare le analisi ricevute, contribuendo a perfezionare ulteriormente le capacità del sistema. Questo crea un ciclo virtuoso di miglioramento continuo, in cui la tecnologia non si limita a fornire risposte, ma impara e si adatta alle esigenze del mondo reale.

Questa piattaforma vuole essere una nuova modalità di interazione tra tecnologia e medicina, promuovendo diagnosi più precise e favorendo la personalizzazione delle cure. Il progetto si propone di rendere accessibile a tutti un sistema avanzato, in grado di accelerare la ricerca clinica e di aprire nuove strade nella medicina predittiva. È un passo verso una cura che non è solo tecnologicamente avanzata, ma anche più umana e vicina alle esigenze dei pazienti.

2 Requisiti

2.1 Obiettivo del sistema

L'obiettivo consiste nel creare un sistema distribuito che sfrutti un'applicazione web per consentire a pazienti e medici di caricare e analizzare radiografie di ginocchia, diagnosticando l'osteoartrite attraverso una rete neurale pre-addestrata. Il sistema mira a migliorare la precisione, offrire un'interfaccia intuitiva e promuovere la collaborazione tra utenti e tecnologia. Inoltre, il sistema integra un meccanismo di miglioramento continuo, riaddestrando periodicamente la rete neurale con nuove radiografie valutate positivamente dai medici, per garantire una progressiva ottimizzazione delle diagnosi.

2.2 Panoramica generale

1. Front-end

- Interfaccia per la registrazione e l'autenticazione degli utenti.
- Dashboard per il caricamento delle radiografie e la visualizzazione delle diagnosi.
- Interfaccia calendario per mostrare tutte le attività svolte da medici e pazienti.

2. Back-end

- Sistema di gestione delle richieste (autenticazione, caricamento, visualizzazione).
- Integrazione con la rete neurale per l'analisi delle immagini.
- Gestione della coda di richieste per garantire scalabilità.
- Sistema di notifiche per informare i pazienti sulle operazioni pianificate.
- Sottosistema dedicato al riaddestramento della rete neurale, basato sulle radiografie valutate positivamente dai medici.

3. Archiviazione

- Database per le credenziali e i profili utente.
- Archiviazione sicura delle radiografie e delle diagnosi associate.
- Gestione delle notifiche e degli eventi pianificati.
- Archiviazione delle valutazioni degli utenti, utilizzate per il miglioramento continuo del modello.

2.3 Requisiti funzionali

1. Utenti e autenticazione

- Il sistema supporta due tipologie di utenti:
 - Pazienti, che possono visualizzare le proprie radiografie.
 - Medici, che possono caricare, analizzare e visualizzare le radiografie dei loro pazienti e pianificare operazioni.
- Gli utenti devono potersi registrare, autenticare (login) e disconnettere (logout).

2. Caricamento e gestione delle radiografie

- Le radiografie caricate vengono archiviate e associate all'utente che le ha caricate.
- Ogni immagine caricata viene processata per l'analisi automatica attraverso una rete neurale.

3. Diagnosi automatizzata

- Il sistema utilizza una rete neurale pre-addestrata per analizzare le radiografie e classificare il grado di gravità dell'osteoartrite (5 livelli).
- Vengono evidenziate, tramite heat map (Grad-CAM), le aree critiche identificate dalla rete.
- Il sistema fornisce indicazioni sulla necessità di intervento chirurgico quando necessario.

4. Visualizzazione e gestione delle diagnosi

- Gli utenti possono accedere a una dashboard per consultare lo storico delle diagnosi.
- I medici possono visualizzare le diagnosi relative ai pazienti associati al loro profilo.
- Le diagnosi includono:
 - Grado di gravità dell'osteoartrite.
 - Heat map sovrapposte alle radiografie.
 - Indicatori di confidenza delle predizioni.

5. Interfaccia calendario e gestione degli eventi

- I medici possono pianificare operazioni o attività, come il caricamento delle radiografie.
- I pazienti possono visualizzare esclusivamente gli eventi che li riguardano.
- Il calendario è sincronizzato con il sistema di notifiche.

6. Sistema di notifiche

- I pazienti ricevono notifiche quando un medico pianifica un'operazione.
- Gli utenti possono segnare le notifiche come lette.
- Le notifiche sono archiviate per consultazioni future.

7. Sistema di feedback e miglioramento continuo

- I medici possono valutare le diagnosi fornite dal sistema.
- Le valutazioni dei medici influenzano, in misura controllata, i successivi processi di predizione.
- Le radiografie che ricevono una valutazione di 5 stelle da parte dei medici vengono utilizzate per il riaddestramento periodico della rete neurale.
- Il riaddestramento viene effettuato in batch, migliorando la qualità delle predizioni future.

3 Design

3.1 Obiettivi di design

Il design del sistema si pone come obiettivo quello di tradurre i requisiti funzionali e non funzionali in modo da rispondere alle esigenze di pazienti e medici e di adattarsi a scenari d'uso molteplici.

Un aspetto fondamentale è la **centralità dell'utente**: l'interfaccia deve essere **intuitiva** e **accessibile**, anche per utenti *con competenze tecnologiche limitate*. La semplicità e la chiarezza sono perseguiti attraverso un'organizzazione efficace dei flussi di lavoro, dalla registrazione alla gestione delle diagnosi e mediante un **design responsivo** che garantisce un'esperienza ottimale su *dispositivi mobili e desktop*. In questo processo si è avuto il supporto di un chirurgo primario - **esperto del dominio** e **target user** - che ci ha fornito di tanto in tanto consigli pratici su aspetti clinici, assicurando che il sistema fosse allineato con le reali necessità dei professionisti sanitari. Oltre a questo si è tenuto un vero e proprio **Focus group** nel quale abbiamo discusso personalmente con lui e abbiamo potuto raccogliere *opinioni, suggerimenti e feedback*.

Il sistema è progettato per presentare informazioni complesse, come le diagnosi generate dalla rete neurale, in modo chiaro e interpretabile. Le heat map, utilizzate per evidenziare le aree critiche nelle radiografie, devono integrarsi visivamente con il resto dell'interfaccia, rendendo immediatamente comprensibili i risultati anche a utenti inesperti.

Un ulteriore obiettivo è garantire la **sicurezza** e la **privacy** dei dati, attraverso soluzioni progettuali che minimizzano i rischi di accesso non autorizzato e rispettano le normative vigenti. L'archiviazione sicura delle immagini e delle credenziali, così come la progettazione di interfacce che **riducano al minimo il rischio di errori umani**, sono stati principi cardine che hanno guidato lo sviluppo.

Infine il design deve tener conto della necessità di **scalabilità** e **modularità**. Ogni componente del sistema deve essere *indipendente e flessibile*, in modo da poter supportare sia un numero crescente di utenti sia future estensioni del sistema.

L'intero processo di design, quindi, non si limita a rispondere ai requisiti ma punta a creare **un'esperienza utente che sia funzionale, gradevole e orientata alla collaborazione tra tecnologia e persone (Human-Computer Interaction)**.

3.2 Target User Analysis

Definiti gli obiettivi principali, è stata condotta un'analisi dettagliata del target di riferimento per il sistema sviluppato. I principali gruppi di utenti sono i **medici** e i **pazienti**, ciascuno con esigenze e aspettative specifiche che il sistema deve soddisfare.

3.2.1 Medici

I medici, in particolare i chirurghi, sono il pubblico di riferimento primario per la piattaforma. La loro necessità principale è quella di *poder consultare facilmente e velocemente le diagnosi e le radiografie dei pazienti*, con una chiara visualizzazione delle aree critiche, che possano essere evidenziate tramite le heat map fornite dalla rete neurale. Il sistema deve permettere ai medici di *pianificare e gestire gli interventi chirurgici o le attività legate ai pazienti in modo semplice ed efficace*, senza dover utilizzare strumenti separati per la gestione delle radiografie e per la

programmazione degli eventi.

Inoltre i medici necessitano di un sistema che gestisca in modo automatico le diagnosi, riducendo la complessità nella valutazione delle immagini. La piattaforma deve offrire anche funzionalità di notifica per avvisarli quando ci sono aggiornamenti relativi agli appuntamenti o alle diagnosi.

Avendo bene a mente questo, sono state stilate delle **Personas** interessanti definendo bene la loro storia. Si è cercato di **evitare** di rendere gli utenti fittizi troppo *elastici, autoreferenziali o appartenenti a classi troppo piccole*.

Personas: Dr. Marco

Dr. Marco è un chirurgo che cerca un sistema che lo colleghi direttamente ai suoi pazienti, permettendogli di visualizzare rapidamente le radiografie e le diagnosi associate. La sua priorità è ridurre il tempo speso nella gestione delle immagini e delle diagnosi, così da concentrarsi maggiormente sulla pianificazione delle operazioni. Poiché determinare se un ginocchio ha necessità di un intervento chirurgico è un processo complesso e spesso difficile da intuire, Dr. Marco desidera avere una rete neurale esperta e affidabile come supporto, per aiutarlo nelle decisioni e aumentare la precisione delle sue valutazioni.

Scenario d'uso: Dr. Marco ha bisogno di un sistema che gli consenta di accedere rapidamente alle radiografie dei suoi pazienti e di ottenere diagnosi supportate da intelligenza artificiale.

1. Dr. Marco accede alla piattaforma e si autentica.
2. Visualizza le radiografie dei pazienti e le diagnosi associate.
3. Esamina la gravità dell'osteoartrite tramite le heat map generate dalla rete neurale.
4. Prende decisioni informate sui trattamenti, con l'affidabilità dell'IA a supporto della sua esperienza clinica.

3.2.2 Pazienti

I pazienti sono il secondo gruppo di utenti del sistema. Le loro esigenze sono incentrate sulla possibilità di visualizzare le proprie radiografie e comprendere la gravità della propria condizione.

I pazienti devono poter *consultare le diagnosi in modo chiaro*, con l'utilizzo di heat map che aiutano a comprendere meglio il proprio stato di salute. Inoltre, il sistema deve inviare notifiche per aggiornamenti sui propri appuntamenti, come quando un medico pianifica un intervento chirurgico o quando ci sono cambiamenti nel piano di cura.

Personas: Laura

Laura è una paziente che ha recentemente ricevuto una diagnosi di osteoartrite e ha bisogno di monitorare l'evoluzione della sua condizione. Non ha molta esperienza con la tecnologia, quindi la piattaforma deve essere facile da usare.

Scenario d'uso:

1. Laura accede alla piattaforma utilizzando le proprie credenziali.
2. Visualizza le radiografie caricate dal suo medico.
3. Seleziona una radiografia e visualizza la diagnosi automatica, inclusa la heat map con le aree critiche evidenziate.
4. Riceve una notifica per un appuntamento pianificato dal medico, con tutti i dettagli necessari.

3.2.3 Sfide comuni e soluzioni progettuali

Le principali sfide comuni tra medici e pazienti sono la gestione di informazioni complesse e la necessità di una piattaforma che sia allo stesso tempo potente e facile da usare. I medici richiedono uno strumento avanzato per analizzare le radiografie, ma *senza perdite di tempo o complicazioni*, mentre i pazienti necessitano di un'interfaccia chiara e accessibile che non richieda competenze tecniche.

Il sistema deve integrare tutte le funzionalità necessarie, come la visualizzazione delle immagini, la pianificazione degli interventi e la gestione delle notifiche, in un'unica piattaforma centralizzata. Ciò ridurrà la necessità di coordinarsi tra più sistemi, come avviene attualmente per la gestione di radiografie e appuntamenti separati.

3.3 Mockup

In fase di ideazione, una priorità assoluta è stata quella di garantire una **navigazione intuitiva** e una rapida comprensione delle funzionalità. *Ad esempio*, la schermata iniziale è stata concepita per offrire una panoramica chiara con un design che favorisse la priorità visiva verso i dati clinici più rilevanti (Figura 1).

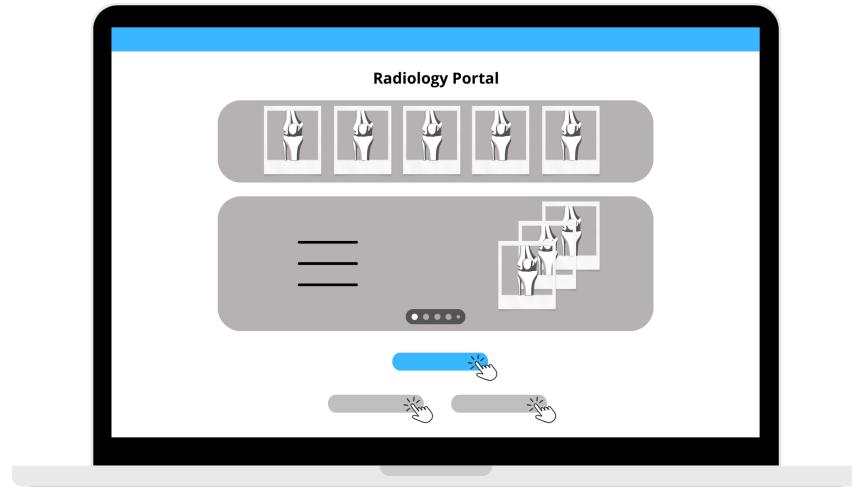


Figura 1: Mockup della piattaforma, progettata per una navigazione intuitiva e una chiara visualizzazione dei dati clinici.

Grazie alla definizione preliminare di **Personas** interessanti, il team è riuscito a *mantenere costantemente al centro del processo di design le esigenze degli utenti*. In questo modo è stato possibile simulare flussi di lavoro realistici, immedesimandosi negli utenti. Così facendo si è concesso ai medici di accedere rapidamente a tutte le informazioni necessarie senza perdere tempo (Figura 2).

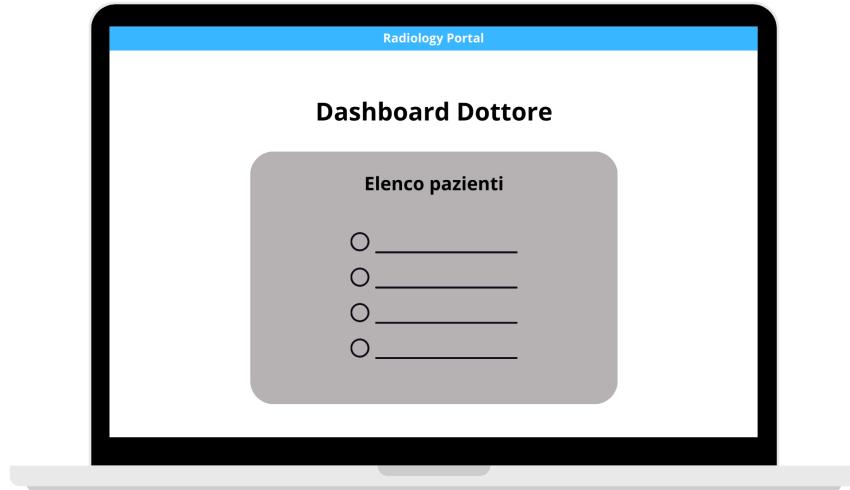


Figura 2: Mockup della dashboard del medico

Grande attenzione è stata riservata alle schermate che avrebbero integrato strumenti di intelligenza artificiale. L'idea di fondo era quella di creare un sistema che non sostituisse il giudizio del chirurgo ma lo affiancasse, fornendo suggerimenti e analisi basati su algoritmi avanzati (Figura 3).

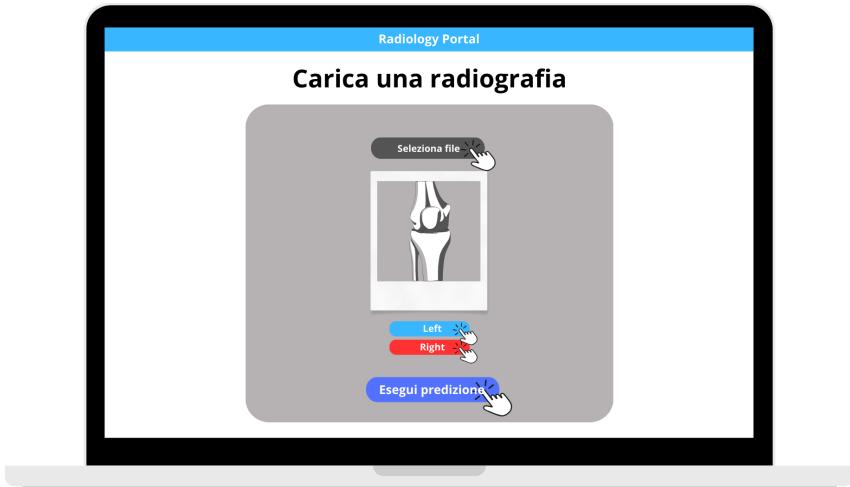




Figura 3: Mockup per il caricamento e la predizione delle radiografie.

Uno degli obiettivi principali dei mockup era immaginare un sistema che potesse semplificare la comunicazione tra i chirurghi e i loro pazienti. La figura 4 illustra un calendario, dove per ogni giorno vengono mostrate in dettaglio le informazioni relative alle radiografie caricate e alle operazioni pianificate. Questo approccio è stato pensato per favorire una consultazione più fluida e immediata.

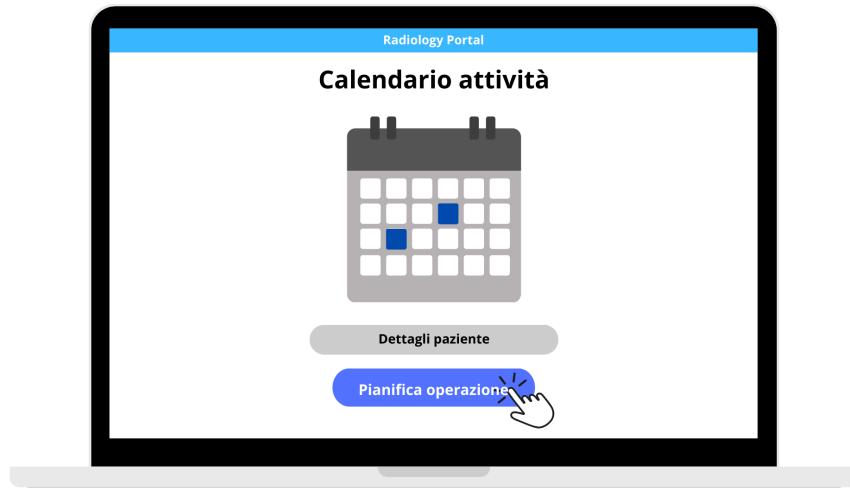


Figura 4: Mockup del calendario delle attività comprensivo al suo interno di tutte le informazioni riguardanti radiografie e operazioni per ogni giorno.

Infine, un altro aspetto emerso durante questa fase creativa è la gestione delle notifiche. La figura 5 illustra il sistema di notifiche integrato, progettato per garantire che chirurghi e pazienti siano sempre aggiornati sugli eventi importanti in tempo reale.

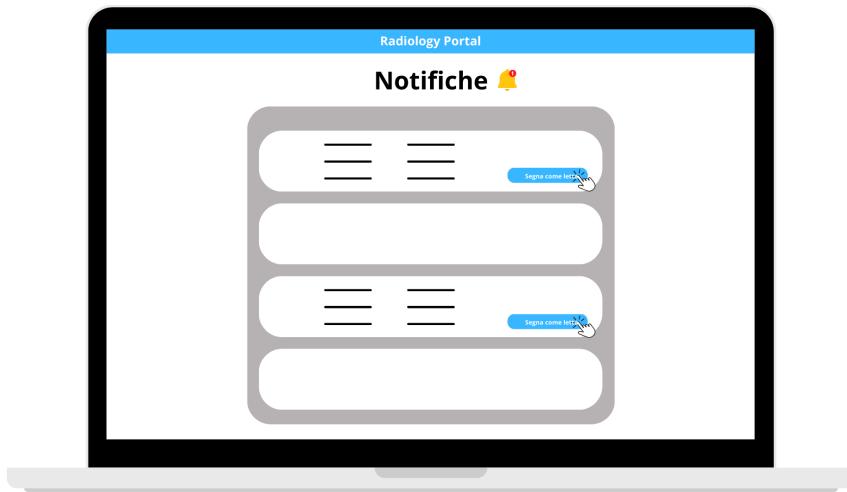


Figura 5: Mockup della sezione notifiche.

3.4 Analisi e modello del dominio

Dato che l'applicazione consiste in un sistema Web progettato per gestire in modo integrato diagnosi mediche e attività cliniche, si è deciso di realizzare il programma aderendo al modello dei servizi *RESTful*. Questo approccio ha consentito di implementare un'interfaccia uniforme, riutilizzabile e facilmente estendibile per l'accesso alle funzionalità del sistema.

A fronte di tutte le analisi svolte precedentemente, si è scelto di distinguere in questo modo le entità principali del dominio:

- **Client web:** costituisce i punti di accesso al sistema per medici e pazienti. Specificando opportune rotte API, i client potranno eseguire operazioni come il caricamento e la visualizzazione delle radiografie.
- **Server web:** si occupa di esporre il servizio ai client, definendo le rotte API necessarie e implementando la logica applicativa. È responsabile di interagire con un database non relazionale.
- **Rete neurale:** viene integrata nel sistema una rete neurale convoluzionale pre-addestrata (*ResNet50*), in grado di classificare il grado di osteoartrite presente nelle radiografie. La rete è progettata per essere periodicamente riaddestrata utilizzando dati di alta qualità, selezionati in base al feedback fornito dai medici.
- **Database non relazionale:** si occupa di mantenere le informazioni relative agli utenti (profili e privilegi), alle radiografie, alle diagnosi e alle operazioni pianificate. Inoltre, fornisce tali informazioni al server su richiesta, consentendo l'esecuzione delle operazioni necessarie. Il database è ottimizzato per garantire sicurezza, efficienza e conformità con le normative *GDPR*. La gestione dei dati è flessibile grazie alla struttura del database non relazionale e, così, consente di scalare facilmente con l'aumentare della complessità e delle dimensioni del sistema.

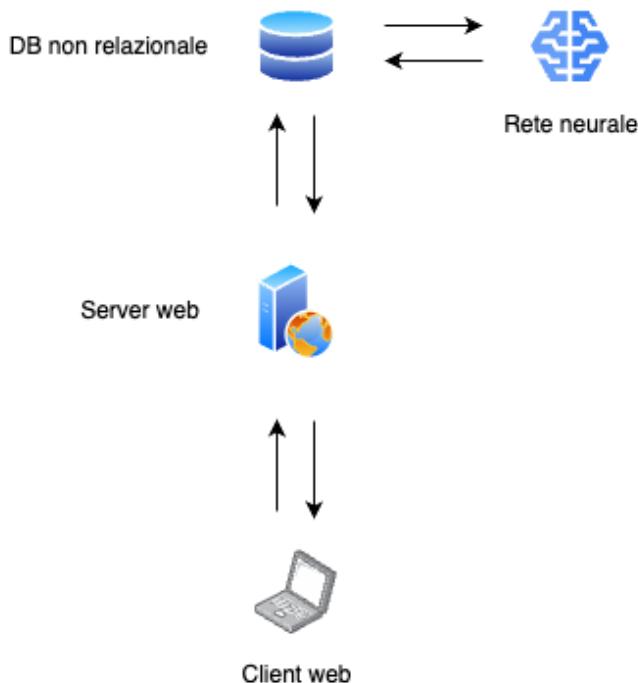


Figura 6: Entità del dominio e le loro interazioni.

3.5 Scelte progettuali chiave

Come base di partenza, a fronte di quanto spiegato in precedenza, viene di seguito riportato il quadro generale comprensivo delle scelte chiave di design.

Per la parte *frontend*, volendo noi implementare una **Single-Page Application (SPA)**, si è scelto il framework **Vue.js**, noto per la sua leggerezza e semplicità nell’implementazione di interfacce utente dinamiche e reattive. Vue inoltre segue il modello strutturato **MVVM**.

Il *backend* è stato sviluppato in **Python** utilizzando il framework **Flask**, una soluzione versatile e performante per la gestione delle richieste e delle operazioni lato server. Flask è stato integrato con **Node.js** per garantire la comunicazione e il coordinamento tra le diverse parti dell’applicazione.

La persistenza dei dati è stata affidata a **Google Firestore**, un database NoSQL offerto dalla piattaforma Google, che consente una gestione rapida e scalabile dei dati oltre a garantire sincronizzazione in tempo reale tra utenti. Firestore è stato usato principalmente per gestire dati strutturati (con attributi), come le informazioni sui pazienti e i dettagli delle operazioni pianificate. Per l’archiviazione dei file, come le radiografie caricate dagli utenti e le diagnosi associate, è stato utilizzato **Google Cloud Storage**, ideale per gestire file di grandi dimensioni in modo scalabile e sicuro. Questa separazione consente di usare Firestore per interrogazioni rapide sui dati strutturati e Cloud Storage per conservare immagini e altri file, ottimizzando così le prestazioni e i costi complessivi del sistema.

L'intera infrastruttura è stata orchestrata utilizzando **Docker**, con i due container - uno per il backend e uno per il frontend - eseguibili tramite file `docker-compose`. Questa configurazione assicura un ambiente isolato e riproducibile per ogni componente del sistema, riducendo i problemi di incompatibilità e semplificando il processo di deployment.

I container Docker sono eseguiti su una macchina virtuale ospitata su **Google Compute Engine**. Questo approccio permette al sistema di essere accessibile attraverso l'indirizzo IP pubblico della VM, rendendolo disponibile su qualsiasi dispositivo connesso a Internet. La scelta di utilizzare Docker su Compute Engine consente di combinare la flessibilità dei container con le prestazioni e l'affidabilità offerte dalla piattaforma Google Cloud.

Il modello di classificazione è costruito con **Keras**. Ogni settimana, un **Job Scheduler** avvia automaticamente l'addestramento del modello, usando parte delle immagini iniziali con l'aggiunta di quelle che sono state valutate con 5 stelle dai medici. Questo aggiornamento è gestito tramite **Vertex AI**, che facilita la gestione dei processi di addestramento del modello e ottimizza le performance, e **Dataproc**, che offre un cluster scalabile per eseguire l'addestramento in modo veloce, anche con grandi volumi di dati.

Inoltre è stata sviluppata una **WGAN-GP**, una rete neurale generativa in grado di creare immagini sintetiche di radiografie. Sebbene attualmente *non sia integrata nel sistema*, **potrebbe essere implementata in futuro** qualora fosse necessario arricchire il dataset per migliorare l'efficacia dell'addestramento del modello di classificazione.

Nel design del sistema, sono stati seguiti principi come **DRY** (Don't Repeat Yourself) e **KISS** (Keep It Simple, Stupid) per garantire un codice pulito e mantenibile, seguendo il concetto **Less is More**. L'architettura è stata progettata per essere modulare e scalabile, assicurando che le nuove funzionalità possano essere integrate senza compromettere la stabilità del sistema esistente. L'usabilità è stata una priorità nella progettazione delle interfacce, adottando i principi del **Responsive Design** per garantire una buona accessibilità su dispositivi di ogni dimensione. A tal proposito, una menzione va fatta anche per l'uso di elementi grafici: le icone rispettano il loro utilizzo convenzionale e i colori sono stati scelti accuratamente in base al tipo di progetto.

Infine, l'intero ciclo di sviluppo è stato gestito seguendo un approccio iterativo e ispirato al framework **Agile**. Il team ha organizzato il lavoro in sprint definiti, identificando le funzionalità da implementare e testare in ogni fase.

Questo è il quadro generale delle tecnologie adottate all'interno del progetto che verranno sviluppate e spiegate meglio successivamente.

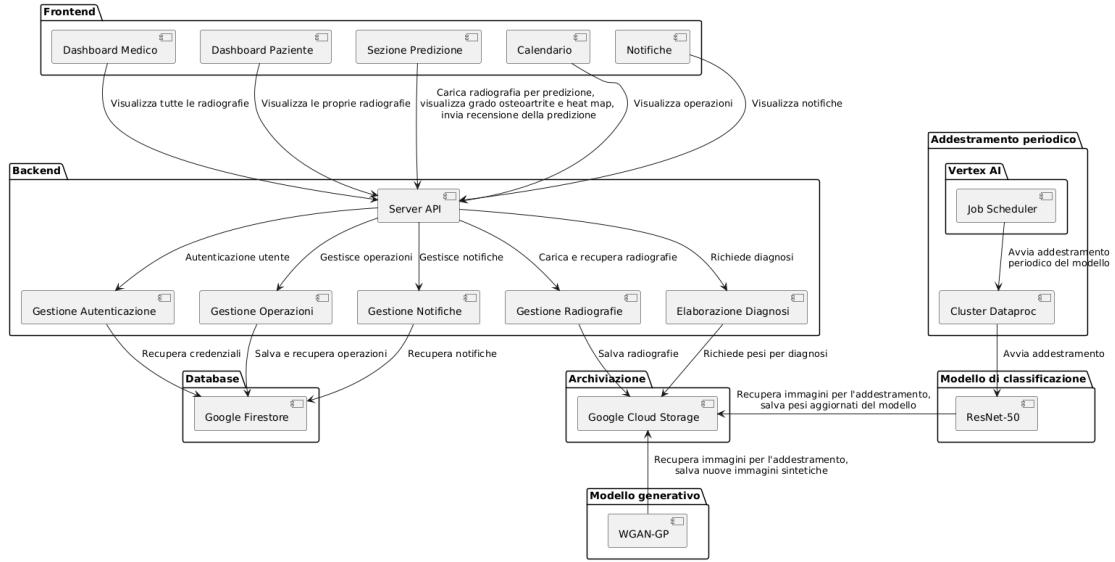


Figura 7: Diagramma del flusso del sistema. La WGAN-GP attualmente non è integrata.

3.6 Esperienza utente (UI/UX)

L'esperienza utente (UI/UX) è stata una delle priorità durante la progettazione della piattaforma. Ogni aspetto del design è stato pensato per rispondere a esigenze specifiche, semplificando il flusso di lavoro e ottimizzando l'interazione con il sistema.

3.6.1 Struttura delle pagine e dei flussi di navigazione

La struttura delle pagine è stata progettata per ridurre al minimo la complessità e migliorare l'accesso rapido alle funzionalità principali. La piattaforma è suddivisa in sezioni facilmente navigabili, come la pagina di caricamento delle radiografie, la visualizzazione delle diagnosi e la gestione degli appuntamenti. Ogni sezione è stata progettata per guidare l'utente in modo chiaro attraverso il flusso di lavoro, riducendo al minimo il numero di passaggi necessari per completare ogni attività. In particolare, il flusso di navigazione tra il caricamento delle immagini, la visualizzazione dei risultati e la gestione delle comunicazioni con il medico è stato reso il più lineare possibile, utilizzando menu a tendina e notifiche chiare per facilitare l'orientamento.

3.6.2 Scelte di stile e palette di colori

Il design visivo è stato sviluppato tenendo conto di un'estetica professionale e facilmente comprensibile. È stata adottata una paletta di colori che utilizza toni neutri per lo sfondo, combinati con colori più vivaci per evidenziare le aree di interesse, come i risultati delle analisi e le informazioni cruciali per il paziente.

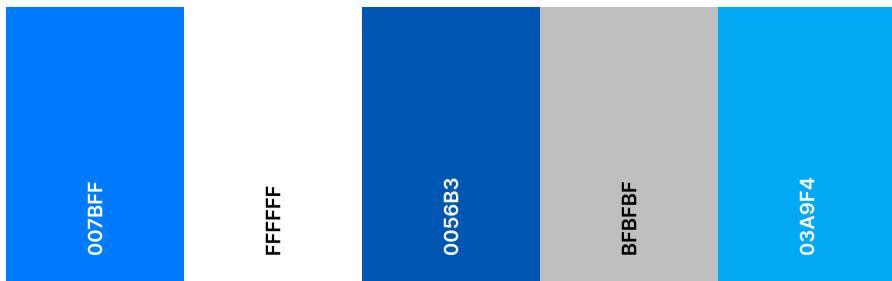


Figura 8: Palette di colori selezionata per il progetto: i colori bianco e grigio sono stati scelti per lo sfondo, mentre i colori più vivaci offrono una buona contrapposizione, facilmente visibile per chiunque.

L’uso di colori distintivi permette di facilitare la lettura e l’interpretazione delle informazioni senza sovraccaricare l’utente. L’interfaccia è progettata per essere semplice e chiara, evitando elementi visivi che potrebbero distrarre o confondere l’utente. I pulsanti, le etichette e le icone sono ben distinti e facilmente identificabili, contribuendo a un’esperienza coerente e senza frizioni. Un’ulteriore attenzione è stata dedicata all’accessibilità per gli utenti daltonici, utilizzando combinazioni di colori che garantiscono un buon contrasto anche per chi ha difficoltà a distinguere certi colori. Sono stati evitati colori che potrebbero risultare indistinguibili per persone con forme comuni di daltonismo, assicurando che ogni elemento visivo sia percepibile da una vasta gamma di utenti.

Per i toni vivaci si è scelto il blu, colore che evoca fiducia, serenità e sicurezza, qualità fondamentali in un contesto medico. Il blu, inoltre, è rilassante e aiuta a ridurre l’ansia dei pazienti, garantendo al contempo un buon contrasto per migliorare la leggibilità delle informazioni cruciali. Questa scelta cromatica contribuisce sia all’estetica che alla percezione di competenza e affidabilità del sito.

3.6.3 Adesione ai principi di accessibilità e responsive design

Per garantire che la piattaforma sia utilizzabile da un'ampia gamma di utenti, sono stati implementati principi di accessibilità che includono un design reattivo, la compatibilità con lo screen reader e la gestione di contrasto cromatico per utenti con difficoltà visive.

La piattaforma è progettata per adattarsi in modo fluido a vari dispositivi, assicurando un'esperienza ottimale indipendentemente dalla dimensione dello schermo. Le funzionalità principali sono accessibili tramite comandi facili da usare anche su dispositivi mobili, dove la disposizione degli elementi cambia per migliorare la leggibilità e la facilità di interazione.

Inoltre, è stata prestata particolare attenzione all'inclusività, garantendo che la piattaforma possa essere utilizzata anche da pazienti con limitate conoscenze tecnologiche, come anziani o persone con disabilità cognitive.

Figura 9: Differenze tra la versione desktop e la versione mobile.

3.6.4 Feedback e interazioni in tempo reale

La piattaforma offre anche un sistema di feedback in tempo reale che consente agli utenti di ricevere aggiornamenti istantanei sullo stato delle loro richieste. Ad esempio, quando un medico carica una radiografia, riceve immediatamente una notifica che conferma l'avvenuto caricamento.

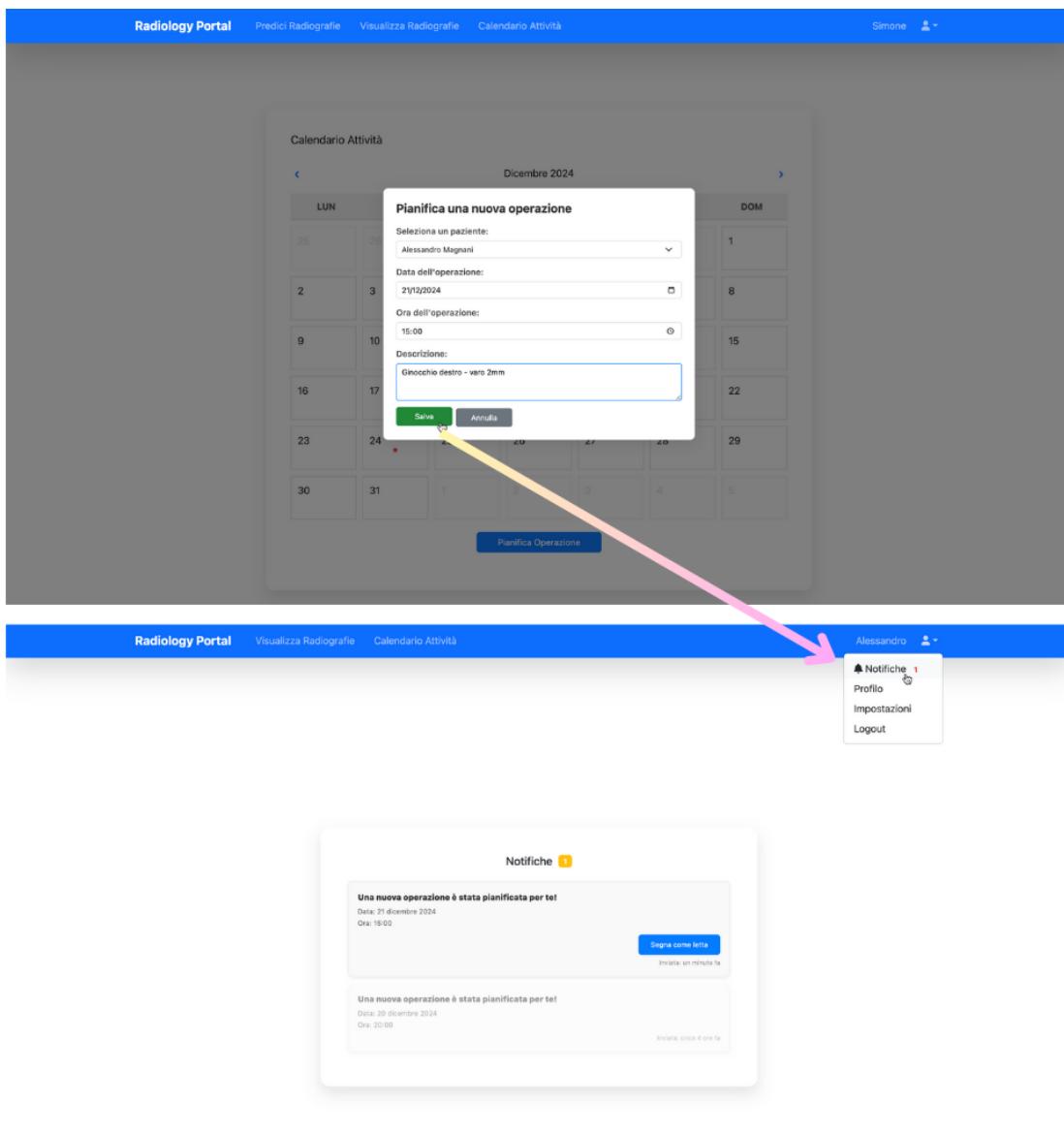


Figura 10: Differenze tra la versione desktop e la versione mobile.

Queste scelte progettuali mirano a rendere l'esperienza utente il più semplice, efficiente e soddisfacente possibile per garantire che ogni interazione con il sistema sia fluida e senza frizioni, contribuendo a migliorare sia l'esperienza del medico che quella del paziente.

3.7 Scalabilità e modularità

La scalabilità e la modularità garantiscono non solo l'adattabilità alle esigenze future, ma anche la facilità di aggiornamento e manutenzione senza compromettere la performance o l'affidabilità del sistema esistente.

3.7.1 Scalabilità del sistema

Il sistema è stato progettato per gestire un aumento significativo del numero di utenti senza influire negativamente sulle prestazioni. La scalabilità orizzontale è stata presa in considerazione per tutti i componenti principali del sistema, in particolare per la parte legata al backend e il database. Grazie all'adozione di tecnologie cloud come **Google Firestore** e **Google Cloud Storage**, il sistema può facilmente espandersi in modo dinamico per supportare una crescita illimitata dei dati e degli utenti. Firestore, essendo un database NoSQL, consente una gestione scalabile dei dati, potendo aggiungere nuovi nodi senza compromettere la performance in caso di picchi di traffico.

The screenshot shows the Google Cloud Firestore interface. At the top, there are buttons for 'Visualizzazione riquadro' and 'Query Builder'. Below the header, the path 'operations > WuZxX4lqueme...' is displayed. On the left, a sidebar shows a tree structure with 'operations' selected. In the main area, a table lists documents under the 'operations' collection. One document is expanded, showing its fields: 'createdAt' (2024-11-29T13:10:49.546242), 'description' ('Varo'), 'doctorId' ('12345'), 'notificationStatus' ('pending'), 'operationDate' ('2024-12-08T00:00:00'), and 'patientId' ('cFcgfvxbP8XMFX5cVSqNZkI3n8S2').

Figura 11: Struttura di Firestore. Permette di espandersi in modo dinamico a supporto di una crescita illimitata di dati e utenti.

Authentication				
Utenti	Metodo di accesso	Modelli	Utilizzo	Impostazioni
<input type="text"/> Cerca per indirizzo email, numero di telefono o UID utente <button>Aggiungi utente</button> ⋮				
Identificatore	Provider	Data di creazione	Accesso eseguito	UID utente
mattboss84@libero.it	✉️	17 dic 2024		zPnKINrvGfNqooFbT7FMQX...
montanarisimone99@g...	✉️	17 dic 2024	17 dic 2024	Lh2zNrs1jJWrYTtJyJTxXf0DA...
andyalemonta@gmail.c...	✉️	17 dic 2024	17 dic 2024	lZsV5wRqzRNvukZfaYZMadD...
alessandro.magnani.00...	✉️	11 dic 2024		07fgk7WPgHV6kASEvj6I95xh...
alessandro.magnani.00...	✉️	1 dic 2024	1 dic 2024	c8P25qf4WvSPOtnF1Vq0HCR...
alessandro.magnani.00...	✉️	22 nov 2024	22 nov 2024	XISTYsYNITaJxu52IvLiLnT138...
alessandro.magnani.00...	✉️	21 ott 2024	17 dic 2024	cFcgfvxbP8XMF5cVSqNZKi3...
alessandromagnani17...	✉️	21 ott 2024	19 dic 2024	EAGjm4TedzU2vWMI3dl8uUt...

Figura 12: Struttura di autenticazione di Firestore. Permette di aggiungere nuovi utenti senza compromettere le performance.

Inoltre la piattaforma è progettata per sfruttare la capacità di **Google Compute Engine** di adattarsi alle necessità di carico, permettendo la gestione automatica delle risorse in base alla domanda. Ciò assicura che l'applicazione mantenga un'alta disponibilità anche sotto carico pesante e durante periodi di traffico elevato.

3.7.2 Modularità del sistema

Il design modulare consente di sviluppare, testare e implementare nuove funzionalità in modo indipendente senza la necessità di modificare l'intero sistema. Ogni componente del sistema è stato progettato come un modulo separato e autonomo. Questo concetto verrà ripreso ulteriormente nella sezione dedicata al codice.

Inoltre, l'uso di container **Docker** ha migliorato ulteriormente la modularità. Ogni servizio è eseguito in un proprio container, il che consente di gestire in modo indipendente le versioni di ciascun componente e di implementare aggiornamenti o modifiche senza impattare sull'intero sistema. Questa struttura modulare rende anche il sistema facilmente manutenibile.

3.7.3 Flessibilità nell'integrazione di nuove tecnologie

La modularità consente una facile integrazione di nuove tecnologie in futuro, ad esempio l'integrazione di tecniche di intelligenza artificiale più avanzate o nuovi metodi di visualizzazione delle immagini. Grazie all'architettura basata su **microservizi**, è possibile aggiungere moduli aggiuntivi che interagiscono con il sistema esistente, mantenendo il tutto coerente e facilmente gestibile. Ciò permette di **rispondere rapidamente alle nuove esigenze e innovazioni del settore medico**, garantendo che la piattaforma rimanga al passo con i tempi.

3.7.4 Gestione delle risorse e ottimizzazione dei costi

In un sistema scalabile l'ottimizzazione delle risorse è fondamentale per mantenere i costi sotto controllo. La piattaforma sfrutta l'infrastruttura **cloud** per ridurre al minimo l'uso di risorse inutilizzate, attivando e disattivando le risorse in base alla domanda. Questo modello consente di *evitare sprechi di risorse e ottimizzare i costi operativi*, mantenendo al contempo una performance ottimale.

Oltre a questo va segnalato che, essendo questa applicazione una **Single-Page Application (SPA)**, a seguito del rendering iniziale, ogni interazione dell'utente viene intercettata come evento da Javascript e viene aggiornata dinamicamente solamente la parte necessaria del **DOM tree**. A differenza delle server-based applications in questo modo *si hanno vantaggi sull'uso efficiente delle risorse, oltre chiaramente a tempi di attesa inferiori e carico sul server inferiore*.

3.8 Diagrammi tecnici

In questo capitolo, vengono presentati i principali diagrammi che descrivono l'architettura, le interazioni e le funzionalità del sistema. Ogni diagramma è stato pensato per guidare il lettore attraverso le dinamiche del progetto, evidenziando le scelte progettuali dei processi chiave.

Ad esempio, il diagramma dei casi d'uso allegato (Figura 13) illustra il processo di registrazione degli utenti, suddiviso in specifiche attività per pazienti e medici. Questo diagramma offre una visione d'insieme delle interazioni tra utenti e sistema, delineando i passi essenziali come l'inserimento dei dati personali, la scelta del medico e la verifica dell'account tramite email.

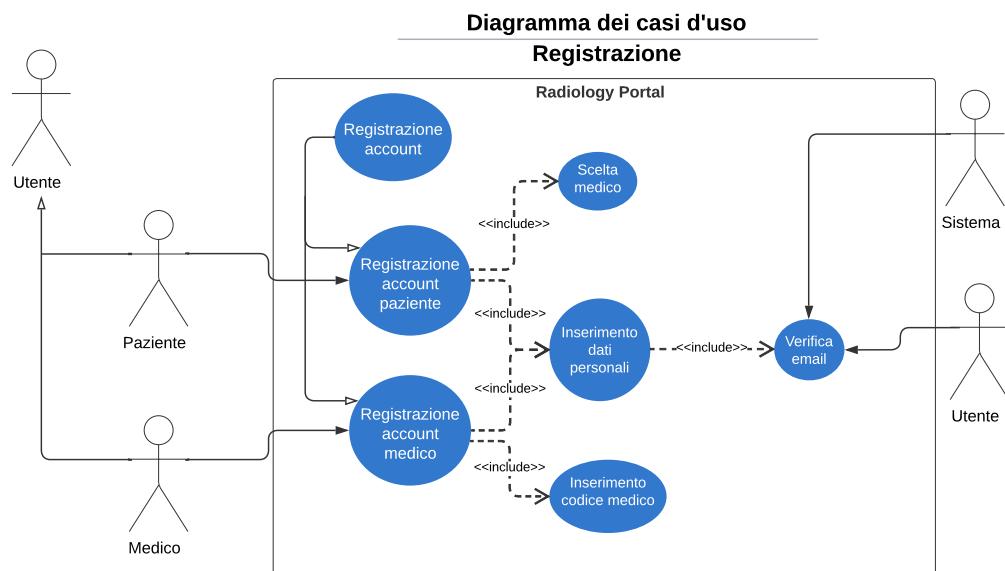


Figura 13: UML - Registrazione di un nuovo utente.

Il processo di login, mostrato in Figura 14, descrive il flusso essenziale per l'accesso al sistema. Partendo dall'inserimento delle credenziali, il sistema verifica la correttezza delle informazioni e, se necessario, gestisce il processo di reset della password tramite invio email. Questo rappresenta un esempio di semplicità ed efficienza nella gestione dell'autenticazione.

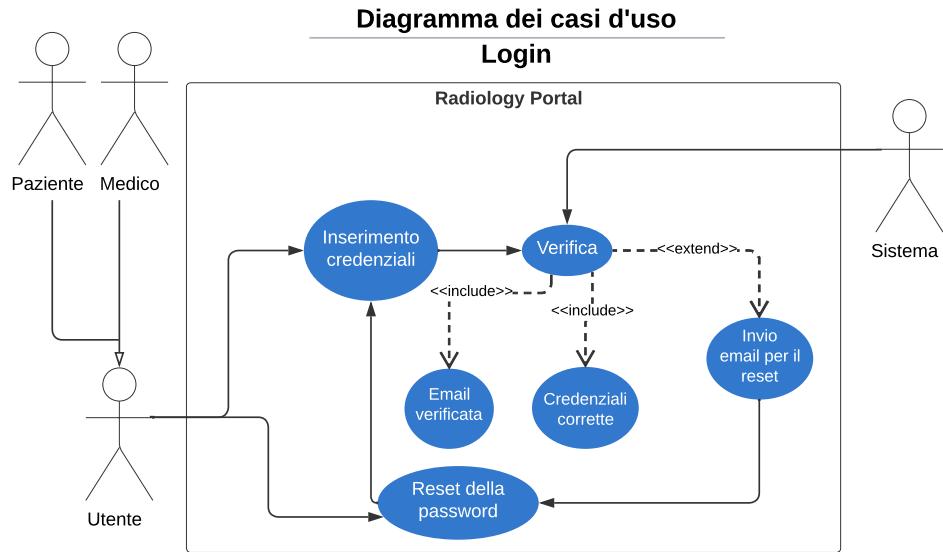


Figura 14: UML - Login.

In Figura 15 viene mostrato il processo di predizione dell'osteoartrite, uno dei flussi centrali e più articolati del sistema. Il medico inizia selezionando un paziente dal proprio profilo, operazione subordinata alla presenza di almeno un paziente associato. Una volta individuato il paziente, il medico procede scegliendo la radiografia desiderata e specificando il lato interessato. A questo punto, attivando il pulsante "Predici Osteoartrite", il sistema elabora automaticamente la predizione e restituisce i risultati. Nella fase finale, il medico ha la possibilità di fornire una recensione del risultato, contribuendo al miglioramento continuo del modello di classificazione.

Diagramma dei casi d'uso

Predizione dell'osteoartrite

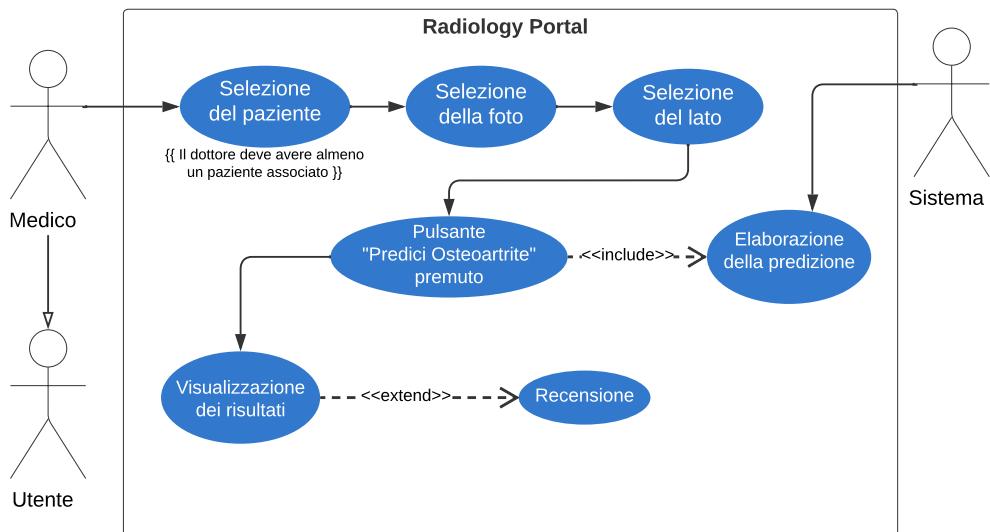


Figura 15: UML - Predizione.

Infine, la visualizzazione delle radiografie descrive come un medico accede alle immagini dei pazienti. Dopo aver selezionato un paziente associato, vengono recuperate le informazioni necessarie e visualizzate in formato miniatura. La selezione di una radiografia specifica consente di accedere ai dettagli, inclusi i risultati precedenti.

Diagramma dei casi d'uso

Visualizzazione delle radiografie

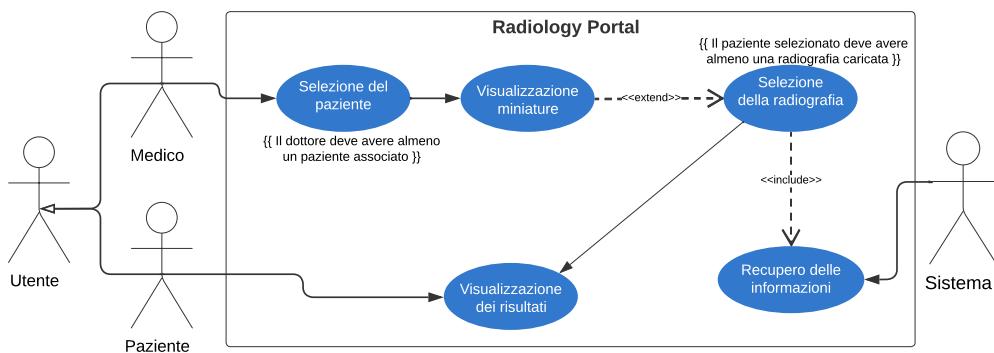


Figura 16: UML - Visualizzazione radiografie.

4 Tecnologie

Per il progetto si è deciso di usare un **signature stack** dato che ci permetteva di essere *flessibili* (nessun limite dettato dagli stack standard) e *innovativi* (prendendo d'esempio aziende importanti). Oltre a questo va segnalato anche l'integrazione con l'addestramento periodico che richiedeva obbligatoriamente di appoggiarci su una piattaforma che ci consentisse di farlo come quella di Google.

4.1 Linguaggi utilizzati

Nel progetto sono stati utilizzati diversi linguaggi di programmazione, ciascuno scelto per soddisfare specifiche esigenze tecnologiche:

- **JavaScript**: per gestire l'interattività dell'interfaccia utente e le comunicazioni asincrone con il backend tramite chiamate API. Grazie alla sua versatilità, JavaScript si integra perfettamente con librerie e framework come `Vue.js` e `Axios`.
- **Python**: per la gestione del backend, è stato utilizzato insieme al framework `Flask` per implementare le API REST. Inoltre, grazie alla vasta disponibilità di librerie come `TensorFlow` e `Keras`, è stato possibile integrare un modello di machine learning preaddestrato.
- **HTML**: per creare la struttura delle pagine web.
- **CSS**: per la definizione dello stile e del layout delle pagine web, sfruttando librerie come `Bootstrap` per accelerare lo sviluppo e assicurare una buona esperienza utente.

4.2 Tecnologie lato frontend

Le tecnologie principali utilizzate per il frontend sono `Node.js`, `Vue.js` e `Axios`, ciascuna scelta per soddisfare esigenze specifiche.

4.2.1 Node.js

`Node.js` è stato utilizzato come runtime JavaScript lato server per lo sviluppo del frontend. Grazie alla sua architettura asincrona e basata su eventi, ha permesso di gestire in modo efficiente le richieste degli utenti e la comunicazione con il backend tramite API REST. Inoltre permette di implementare connessioni **real-time** e **bidirezionali**. `Node.js` è stato usato per la gestione del server locale durante lo sviluppo dell'interfaccia, grazie alla vasta disponibilità di pacchetti tramite `npm` (Node Package Manager).

4.2.2 Vue.js

Per la creazione dell'interfaccia utente, è stato scelto `Vue.js`. Ogni elemento dell'interfaccia è stato suddiviso in componenti autonomi e modulari, come il caricamento delle radiografie o la visualizzazione delle diagnosi.

Inoltre, la presenza di un `Vue Router` integrato ha facilitato la navigazione tra le diverse pagine del sistema, migliorando l'organizzazione e la fluidità dell'esperienza utente. La scelta di Vue rispetto ad altre tecnologie come Angular o React è stata dettata dalla sua sintassi intuitiva, dalla flessibilità nella gestione dei componenti e dalla crescente popolarità tra gli sviluppatori, che lo rende particolarmente adatto a progetti moderni e scalabili.

4.2.3 Axios

Axios è una libreria di JavaScript utilizzata per semplificare la gestione delle comunicazioni HTTP tra il frontend e il backend.

Le principali operazioni realizzate tramite Axios includono:

- L'invio di dati al backend, come le radiografie caricate dai medici.
- La ricezione di risposte dalle API REST, come le diagnosi e le informazioni di autenticazione.
- La gestione degli errori, garantendo notifiche adeguate agli utenti in caso di problemi di rete o del server.

4.3 Backend

Il backend del sistema è stato progettato per gestire le API REST, orchestrare la comunicazione con i servizi esterni (**Firebase** e **Google Cloud Storage**) e integrare il modello di rete neurale per la diagnosi dell'osteoartrite. La tecnologia principale utilizzata è **Flask**, un framework Python leggero e flessibile, ideale per la creazione di applicazioni web e microservizi.

4.3.1 Flask per la gestione delle API REST

La scelta di **Flask** rispetto ad altri framework, come **Django** o **FastAPI**, è stata influenzata dalla sua leggerezza e dalla flessibilità nel configurare solo i moduli necessari. Inoltre, la semplicità di integrazione con librerie come **TensorFlow** e **Firebase** lo ha reso ideale per soddisfare le esigenze del progetto.

Le principali funzionalità includono:

- **gestione delle API REST**: implementazione di endpoint per la gestione delle radiografie, delle richieste di predizione, delle notifiche e del calendario.
- **integrazione con Firebase Authentication**: gestione delle credenziali degli utenti durante la registrazione e il login, utilizzando token sicuri.
- **interazione con Google Cloud Storage**: salvataggio e recupero di radiografie e risultati delle diagnosi in modo sicuro ed efficiente.
- **elaborazione delle diagnosi**: invio delle radiografie al modello di rete neurale (**ResNet50**) e restituzione dei risultati al frontend.

4.4 Autenticazione

Per la gestione della registrazione, dell'accesso e dell'autenticazione degli utenti, il sistema utilizza due servizi principali di **Google Firebase**: **Firebase Authentication** e **Firestore Database**.

4.4.1 Firebase Authentication

Firebase Authentication è stato utilizzato per gestire la registrazione, il login e l'autenticazione degli utenti. Questo servizio offre un sistema di autenticazione sicuro basato su token, garantendo che solo utenti autorizzati possano accedere al sistema.

Le principali funzionalità di Firebase Authentication implementate nel progetto includono:

- **Autenticazione tramite email e password:** gli utenti possono registrarsi con credenziali univoche e accedere in modo sicuro.
- **Verifica dell'email:** durante la registrazione, viene inviato un link di conferma per verificare l'indirizzo email dell'utente.
- **Gestione dei tentativi di login falliti e reset della password:** se un utente supera il limite massimo di tentativi di accesso (6), l'account viene temporaneamente bloccato e viene inviata un'email automatica con un link per il reset della password.

4.4.2 Firestore Database

Firestore Database è un database NoSQL utilizzato per memorizzare dati strutturati legati agli utenti, alle operazioni pianificate e alle notifiche.

Firestore è stato scelto per la sua semplicità di configurazione, l'integrazione nativa con **Firebase Authentication** e la capacità di garantire un'elevata disponibilità e affidabilità anche in applicazioni distribuite.

4.5 Archiviazione

Per l'archiviazione di file e dati non strutturati, il sistema utilizza **Google Cloud Storage**, un servizio di storage scalabile e altamente affidabile offerto da Google Cloud Platform. I tipi di file salvati nel bucket includono immagini, come radiografie e heat map, e file di testo con i risultati delle predizioni. Inoltre, sono salvati anche i pesi del modello di ResNet50 e le strutture dati necessarie per l'addestramento periodico.

Il bucket è configurato per garantire:

- **archiviazione sicura:** i dati sensibili, come le radiografie, sono archiviati in modo sicuro grazie alla crittografia end-to-end di Google Cloud.
- **accesso controllato e recupero rapido:** solo gli utenti autorizzati (medici), tramite le API del backend, possono caricare radiografie. Inoltre, le radiografie devono poter essere recuperate rapidamente per essere analizzate dal modello di rete neurale.
- **scalabilità e disponibilità:** Google Cloud Storage offre una soluzione altamente scalabile che consente di gestire un grande volume di file senza compromettere le performance del sistema.

4.6 Containerizzazione e Deployment

Il sistema, come spiegato anche in precedenza, è stato containerizzato con **Docker** e distribuito su una macchina virtuale ospitata su **Google Compute Engine**. Questa configurazione permette di separare i diversi componenti del sistema in **container**, semplificando il deployment e la gestione.

4.6.1 Docker, Containerizzazione e Orchestrazione

Docker è stato utilizzato per creare container che isolano le dipendenze e i componenti del sistema. Ogni container rappresenta un'unità indipendente, che può essere eseguita in modo uniforme su qualsiasi piattaforma compatibile con Docker.

Sono stati configurati due container principali: uno per il frontend e uno per il backend. Per semplificare e orchestrare il processo di deployment, è stato creato un file `docker-compose.yml`, che definisce i servizi e i volumi necessari per eseguire il sistema in modo coordinato ed efficiente.

4.6.2 Deployment su Google Compute Engine

Il sito è ospitato su una macchina virtuale configurata su **Google Compute Engine**, una piattaforma scalabile e affidabile offerta da Google Cloud. La VM è stata configurata per eseguire i container creati con Docker, rendendo il sistema accessibile tramite il suo indirizzo IP pubblico. L'utilizzo di Google Compute Engine ha inoltre permesso di integrare agevolmente il sistema con altri servizi della piattaforma, come Google Cloud Storage e Firebase Authentication, ottimizzando la gestione dei dati e l'autenticazione degli utenti.

4.7 Integrazione del Modello di Rete Neurale Pre-addestrato

Per integrare il modello pre-addestrato ResNet50 nel sistema, sono state utilizzate diverse librerie Python che hanno facilitato sia il pre-processing delle immagini sia l'inferenza del modello.

Nello specifico, sono state utilizzate le seguenti librerie:

- OpenCV.
- NumPy.
- Tensorflow.
- Keras.
- h5py.

4.8 Github

GitHub è stato utilizzato come piattaforma di versionamento del codice per facilitare la collaborazione tra i membri del team.

4.9 Addestramento periodico del modello

L'idea di implementare un sistema di addestramento periodico nasce dall'obiettivo di rendere il modello sempre più accurato nel tempo, sfruttando il feedback diretto dei chirurghi che utilizzano quotidianamente il software. Nello specifico, per ogni predizione effettuata, il chirurgo ha la possibilità di valutare l'accuratezza del risultato. Se la predizione viene considerata accurata, la radiografia corrispondente viene aggiunta al dataset esistente. Periodicamente, ad esempio ogni due settimane, si è previsto di avviare automaticamente un nuovo addestramento del modello, includendo queste nuove immagini. Una volta completato l'addestramento, si confrontano le metriche del nuovo modello con quello attualmente utilizzato. Se il nuovo modello risulta migliore, il sistema aggiorna la versione in uso.

Per implementare la schedulazione di questo processo, si sono sperimentate diverse soluzioni, tra cui *Cloud Function* e *Job Scheduler*, ma si è scelto di utilizzare *Vertex AI*, che ha mostrato i migliori risultati in termini di stabilità e integrazione. Tuttavia, le **risorse limitate** del piano gratuito di Google Platform hanno rappresentato un *ostacolo significativo*. Pur essendo la schedulazione funzionante, si è osservato che la parte di pre-processing veniva completata senza difficoltà ma il processo di training veniva spesso interrotto a causa dell'insufficienza di risorse. Questo non dipendeva da problemi nel codice, in quanto l'esecuzione manuale non causava complicazioni.

Nonostante queste difficoltà, si è riusciti a creare prototipi funzionanti per la schedulazione. Con risorse più potenti si ritiene che il sistema possa diventare completamente operativo, consentendo di automatizzare ulteriormente non solo il confronto delle metriche ma anche operazioni di *data cleaning* e ottimizzazione dei dati per gli addestramenti successivi. Questo approccio, una volta implementato con risorse adeguate, garantirebbe un miglioramento incrementale del modello e del sistema complessivo.

5 Codice

5.1 Struttura

Come spiegato all'inizio il nostro dominio è costituito da quattro entità base: database non relazionale, Server, Client Web e Rete neurale. Come ampiamente discusso in precedenza il sistema è stato suddiviso in due container distinti, uno per il backend e uno per il frontend, orchestrati attraverso Docker. Di conseguenza, la macro-struttura del nostro sistema si presenta in questo modo.

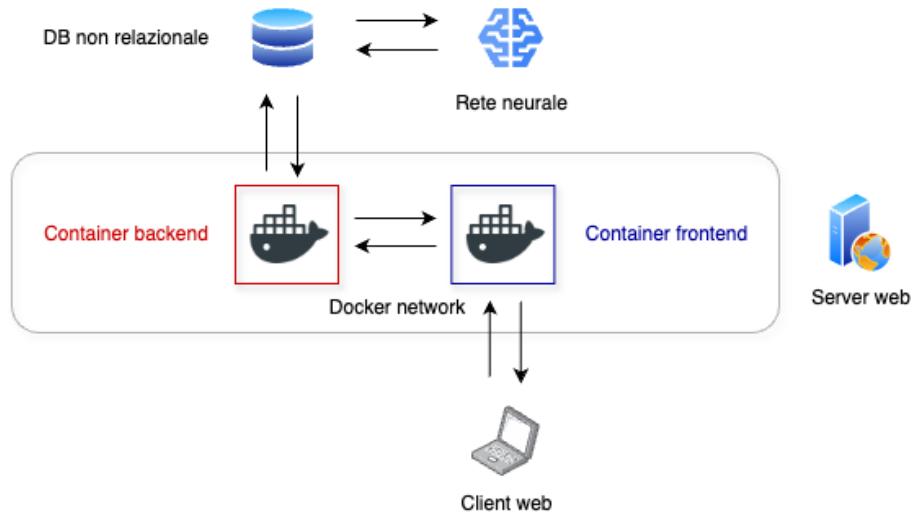


Figura 17: Struttura del sistema

5.1.1 Struttura del Database

Per prima cosa è stato definito lo schema Entity/Relationship del database, mettendo in luce quali sono le diverse entità che dovranno essere modellate e le relazioni che vi sono fra queste.

Come si può vedere dalla figura 18, lo schema è costituito dalle seguenti entità:

- **Users**, rappresenta gli utenti iscritti all'applicazione. Un utente può essere un medico (*Doctor*) o un paziente (*Patient*) ed è identificato in modo univoco tramite l'attributo `id`.
- **Doctors**, rappresenta i medici registrati nel sistema, associati agli utenti tramite l'attributo `doctorId`.
- **Patients**, rappresenta i pazienti registrati nel sistema. Un paziente è associato a un medico attraverso l'attributo `doctorRef`.
- **Radiographs**, rappresenta le radiografie dei pazienti. Poiché le radiografie sono immagini non strutturate, sono memorizzate in Google Cloud Storage e non hanno attributi specifici, tranne un identificativo `id`.
- **Predictions**, rappresenta le predizioni fatte dai medici sulle radiografie. Anche le predizioni sono memorizzate in Google Cloud Storage e sono collegate alle radiografie tramite l'attributo `id`.

- **Models**, rappresenta l'architettura del modello di rete neurale preaddestrata. Anche questi sono dati non strutturati e vengono utilizzati dai medici per predire le radiografie.
- **Operations**, rappresenta le operazioni pianificate per i pazienti. Ogni operazione è associata a un paziente e un medico specifico tramite gli attributi `patientId` e `doctorId`.
- **Notifications**, rappresenta le notifiche inviate ai pazienti riguardanti le operazioni pianificate. Ogni notifica è associata a un paziente tramite l'attributo `patientId`.

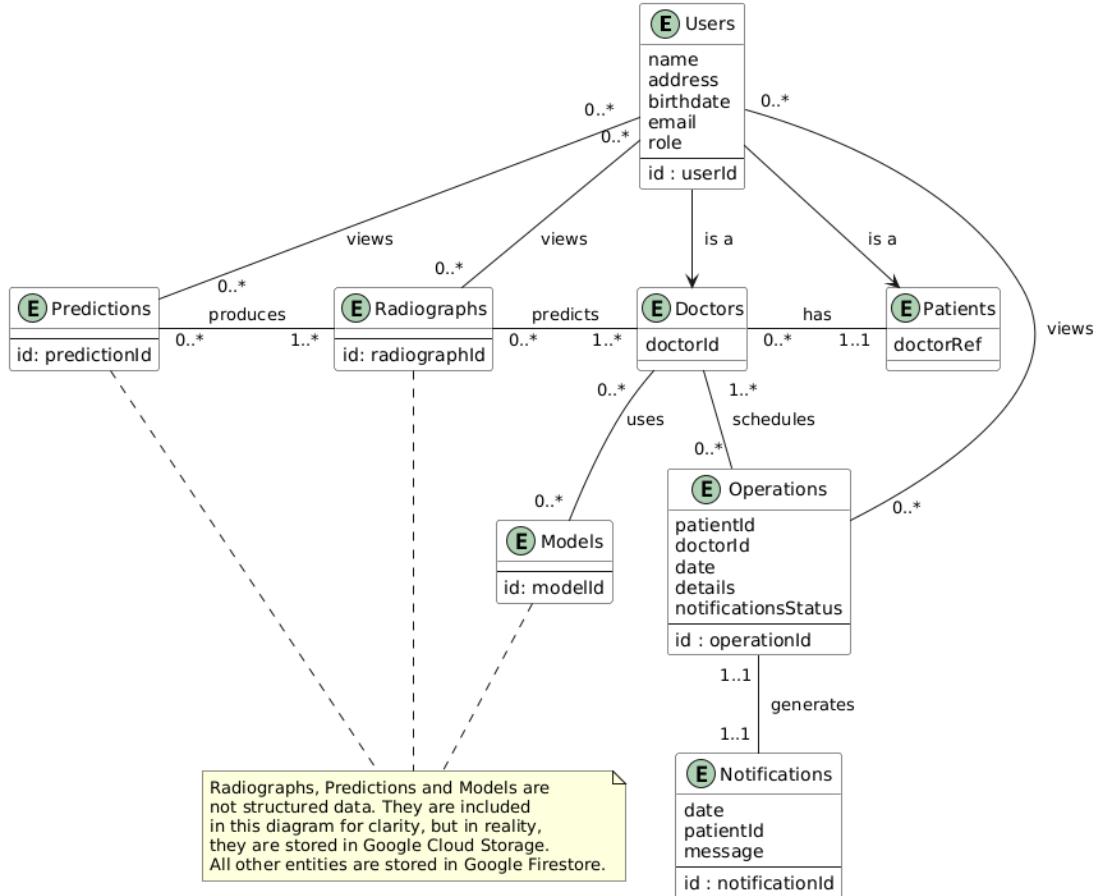


Figura 18: Schema Entity/Relationship del Database

Le radiografie (**Radiographs**), le predizioni (**Predictions**) e i pesi (**Weights**) sono dati non strutturati e vengono memorizzati separatamente in Google Cloud Storage. Tali entità non hanno attributi complessi ma sono solo immagini o file di predizione. Questi dati sono stati inclusi nel diagramma ER per chiarezza, ma in realtà non sono parte della struttura del database Firestore.

Le operazioni (**Operations**) sono pianificate dai medici per i pazienti. Ogni operazione è legata a una notifica (**Notifications**), che viene inviata al paziente una volta che l'operazione è stata programmata. Ogni operazione genera una notifica, e ogni notifica è associata a un singolo

paziente. Le notifiche sono memorizzate come dati strutturati nel database, contrariamente alle radiografie e alle predizioni.

La modellazione segue il principio di separare chiaramente i dati strutturati (salvati in Firestore) dai dati non strutturati (salvati in Google Cloud Storage), per semplificare la gestione del database e migliorare le performance.

5.1.2 Struttura del sistema: moduli principali

Il progetto è stato sviluppato suddividendo il sistema in diversi moduli funzionali, garantendo una chiara separazione delle responsabilità e una gestione efficiente delle funzionalità. I moduli principali del sistema sono i seguenti:

Lato Backend

- **Core:** rappresenta il cuore del sistema, coordina tutte le operazioni e i servizi. È responsabile di inizializzare e configurare il sistema. Coordina l'interazione tra i diversi moduli per assicurare il corretto svolgimento delle operazioni
- **Utility:** fornisce servizi di supporto per i diversi moduli del sistema, gestendo operazioni specifiche come l'invio di notifiche, la gestione dei dati, l'archiviazione e le operazioni del modello. Questo modulo garantisce un insieme di strumenti affidabili per supportare le funzionalità principali del sistema
- **Controller:** è il modulo responsabile della logica di business. Coordina le operazioni richieste dagli utenti, interfacciandosi con i servizi e le funzionalità del sistema per garantire una gestione coerente e strutturata dei processi
- **Factory:** si occupa della creazione e configurazione dei vari elementi del sistema, assicura che i servizi e i moduli siano inizializzati e configurati correttamente. Coordina la gestione delle dipendenze tra i componenti, favorendo un approccio centralizzato alla configurazione
- **Routing:** gestisce l'instradamento delle richieste all'interno del sistema, assicurando che ogni richiesta venga elaborata dal modulo appropriato. Mappa le interazioni degli utenti verso le operazioni corrispondenti, facilitando una comunicazione strutturata all'interno del sistema
- **Config:** si occupa della gestione della configurazione del sistema come le credenziali per accedere a Firestore e GCS o le variabili di configurazione di email e model

Lato Frontend

- **UI:** si occupa della presentazione e dell'interazione con l'utente, gestendo le diverse viste e funzionalità dell'applicazione. Consente agli utenti di accedere alle informazioni e ai servizi offerti dal sistema, garantendo un'esperienza fluida e intuitiva
- **Service:** è il modulo che gestisce la comunicazione tra l'interfaccia utente e il sistema centrale. Si occupa della gestione delle richieste e delle risposte e garantisce un flusso di dati strutturato ed efficiente. Questo modulo centralizza le interazioni con il sistema backend, semplificando la gestione delle chiamate e degli errori

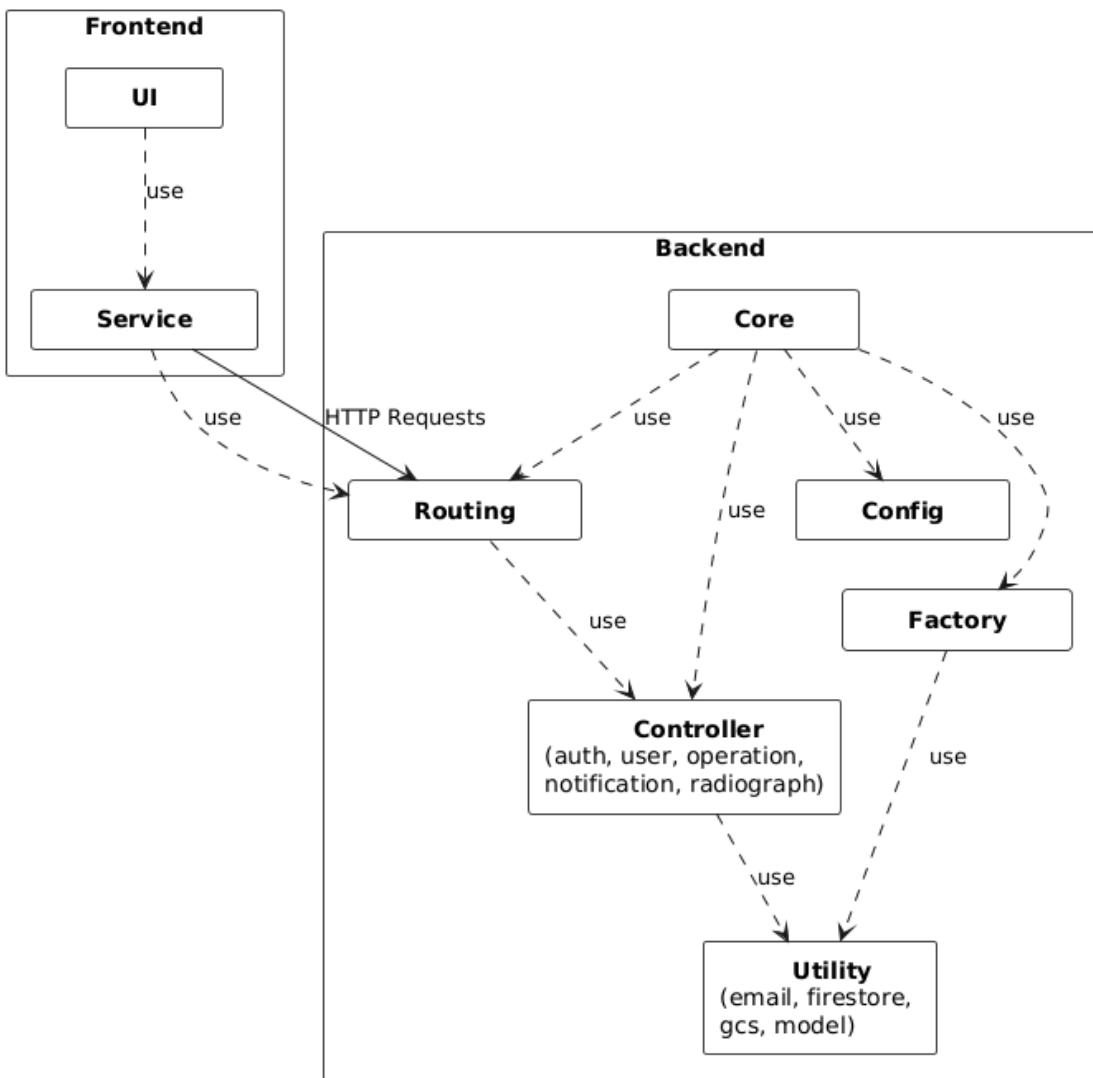


Figura 19: Diagramma delle dipendenze

Come si può vedere in Figura 26 il flusso di interazioni tra i moduli ha origine dal frontend, dove i componenti dell’interfaccia utente interagiscono con il modulo dei **Servizi**. Questo servizio gestisce le chiamate HTTP verso il backend, comunicando principalmente con il modulo di **Routing**, che espone le API attraverso delle rotte specifiche.

Nel backend, il cuore del sistema è rappresentato dal **Core**, che coordina tutte le interazioni tra i vari moduli. Il **Core** si occupa di orchestrare la creazione delle istanze delle **Utility** tramite il modulo **Factory**. Le **Utility** gestiscono funzioni di supporto, come l’invio di email, l’interazione con il database, la gestione dello storage su Google Cloud e l’esecuzione delle operazioni del modello. Queste **Utility** sono poi iniettate nei **Controllers**, i quali costituiscono il punto di contatto tra il **Routing** e la logica di business del sistema.

Il modulo **Routing** riceve le richieste HTTP dal frontend e le indirizza ai **Controllers** appropriati. I **Controllers**, a loro volta, utilizzano le **Utility** per eseguire le operazioni necessarie. Il sistema è configurato attraverso il modulo **Config**, che fornisce i parametri essenziali per il corretto funzionamento dell'applicazione.

5.1.2.1 Modulo Core

Il modulo Core vuole essere il punto di orchestrazione del sistema, implementando un **server stateless basato su Flask**. La scelta di un'architettura stateless è stata dettata dalla necessità di garantire scalabilità e manutenibilità del sistema: ogni richiesta contiene tutte le informazioni necessarie per la sua elaborazione, eliminando la necessità di mantenere uno stato server-side e facilitando così la gestione delle risorse.

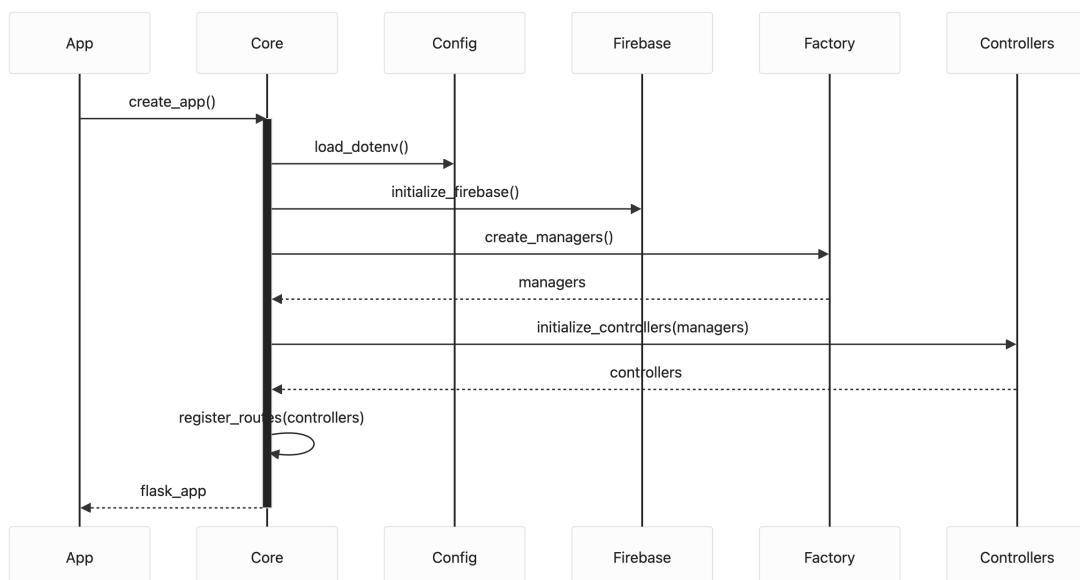


Figura 20: Diagramma di sequenza: modulo Core.

Il core sfrutta principalmente il file `app.py`, che definisce l'app Flask e ne configura l'esecuzione in modalità stateless, inizializzando i vari managers e registrando le rotte.

Il Core adotta il pattern **Factory** per la creazione e la gestione delle dipendenze, un approccio che favorisce la modularità e semplifica l'aggiunta di nuovi componenti. Questa scelta progettuale permette di centralizzare la logica di inizializzazione.

L'architettura del Core si basa su una chiara separazione delle responsabilità: mentre il Core si occupa dell'orchestrazione generale, delega le specifiche funzionalità ai rispettivi moduli specializzati. Questa separazione è implementata attraverso un sistema di dependency injection, dove il Core si occupa di inizializzare e iniettare le dipendenze necessarie nei vari controllers.

Un aspetto fondamentale del design è la configurabilità del sistema. Il Core sfrutta un approccio basato su variabili d'ambiente, permettendo una facile configurazione in diversi ambienti.

di deployment senza necessità di modifiche al codice.

La natura stateless del server, combinata con l'utilizzo di Flask, permette di ottenere un sistema altamente scalabile e performante, capace di gestire multiple richieste in modo efficiente liberando rapidamente le risorse.

Listing 1: File app.py.

```
1 def create_app():
2     app = Flask(__name__)
3
4     # Set Google Cloud credentials environment variable
5     os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = AppConfig.GCS_CRED_PATH
6
7     # Configure CORS using AppConfig
8     CORS(app, resources={r"/+": {"origins": AppConfig.CORS_ORIGIN}})
9
10    # Initialize Firebase Admin
11    cred = firebase_admin.credentials.Certificate(AppConfig.FIREBASE_CRED_PATH)
12    firebase_admin.initialize_app(cred)
13
14    # Inizializza managers
15    managers = ManagerFactory.create_managers(app.config)
16
17    # Inizializza controllers
18    controllers = {
19        'auth': AuthController(managers),
20        'user': UserController(managers),
21        'operation': OperationController(managers),
22        'notification': NotificationController(managers),
23        'radiograph': RadiographController(managers)
24    }
25
26    # Registra routes
27    register_routes(app, controllers)
28
29    return app
30
31 app = create_app()
```

5.1.2.2 Modulo Utility

Il modulo Utility è stato progettato per fornire un insieme di funzionalità di supporto essenziali per l'interazione con vari servizi esterni e per l'elaborazione dei dati. Il modulo è stato suddiviso in quattro componenti principali, ognuno specializzato in un aspetto specifico del sistema:

- **FirestoreManager:** gestisce tutte le interazioni con il database Firestore, fornendo un'interfaccia uniforme per le operazioni CRUD (Create, Read, Update, Delete) e implementando funzionalità specifiche per il dominio applicativo.

Listing 2: File `firestore_utils.py`.

```
1  class FirestoreManager:
2      def __init__(...):
3
4          # Funzioni generiche CRUD
5          def create_document(...):
6              def get_document(...):
7                  def update_document(...):
8                      def query_documents(...):
9
10         # Funzioni specifiche per gli utenti
11         def create_user(...):
12             def get_users_by_role(...):
13                 def get_doctor_patients(...):
14
15         # Funzioni specifiche per le operazioni
16         def create_operation(...):
17
18         # Funzioni specifiche per le notifiche
19         def create_notification(...):
20             def get_user_notifications(...):
21                 def mark_notification_read(...):
22
23         # Funzioni per la gestione dei tentativi di login
24         def update_login_attempts(...):
25             def get_patient_information(...)
```

- **GCSManager:** si occupa dell’interazione con Google Cloud Storage. Le funzionalità principali sono le operazioni CRUD sui file nel bucket di storage, il caricamento e la gestione dei modelli di machine learning e delle radiografie e la generazione di URL pubblici per l’accesso ai file

Listing 3: File `gcs_utils.py`.

```
1  class GCSManager:
2      def __init__(...):
3
4          # Operazioni CRUD di base
5          def upload_file(...):
6              def download_file(...):
7                  def delete_file(...):
8
9          # Operazioni specifiche per il dominio
10         def list_patient_radiographs(...):
11             def get_radiograph_info(...):
12                 def save_radiograph(...):
13                     def count_patient_radiographs(...):
14                     def load_model(...):
15                     def get_public_url(...):
16                     def save_gradcam_image(...):
17                     def process_radiograph_folder(...)
```

- **EmailManager**: fornisce un’interfaccia semplificata per l’invio di email attraverso il servizio SMTP di Gmail.

Listing 4: File `email_utils.py`.

```

1 class EmailManager:
2     def __init__(...)
3     def send_email(...)

```

- **ModelManager**: gestisce le operazioni relative al modello di machine learning per l’analisi delle radiografie. In particolare si occupa del preprocessing delle immagini per l’input al modello, l’esecuzione delle predizioni, la generazione di heatmap e l’elaborazione delle immagini per la visualizzazione dei risultati.

Listing 5: File `model_utils.py`.

```

1 class ModelManager:
2     def __init__(...)
3     def preprocess_image(...)
4     def predict_class(...)
5     def make_gradcam_heatmap(...)
6     def generate_gradcam(...)

```

5.1.3 Modulo Controller

Il modulo Controller gestisce la logica di business e fa da intermediario tra le richieste degli utenti e i servizi del sistema. Il modulo è organizzato secondo il principio della separazione delle responsabilità, con ogni controller dedicato a uno specifico dominio dell’applicazione. La struttura del modulo si basa su un’architettura orientata agli oggetti, dove ogni controller è implementato come una classe che riceve le sue dipendenze attraverso il costruttore (dependency injection). Si è pensato a questo approccio dal momento che rende più semplice testare e manutenere il codice.

Il modulo si articola in cinque controller specializzati:

- **AuthController**: gestisce tutti gli aspetti legati all’autenticazione e all’autorizzazione degli utenti, inclusa la registrazione, il login, la verifica dell’email e il reset della password. Integra i servizi di Firebase Authentication per garantire una gestione sicura delle credenziali.

Listing 6: File `auth_controller.py`.

```

1 class AuthController:
2     def __init__(self, managers)
3     def register(...)
4     def login(...)
5     def check_email_verification(...)
6     def verify_email(...)
7     def reset_password(...)
8     def send_reset_email(...)
9     def decrement_attempts(...)
10    def get_attempts_left(...)

```

- **UserController:** si occupa della gestione degli utenti del sistema, fornisce funzionalità per la gestione dei profili, la distinzione tra pazienti e dottori e le relazioni tra questi. Implementa logiche specifiche per il recupero e l'aggiornamento dei dati utente.

Listing 7: File `user_controller.py`.

```

1  class UserController:
2      def __init__(self, managers)
3      def get_user(...)
4      def update_user(...)
5      def get_doctors(...)
6      def get_patients(...)
7      def get_patients_from_doctor(...

```

- **RadiographController:** responsabile della gestione delle radiografie, incluso il caricamento, l'elaborazione e l'analisi delle immagini. Integra funzionalità di machine learning per la predizione diagnostica e la generazione di heatmap (Grad-CAM).

Listing 8: File `radiograph_controller.py`.

```

1  class RadiographController:
2      def __init__(self, managers)
3      def get_patient_radiographs(...)
4      def download_radiograph(...)
5      def upload_to_dataset(...)
6      def get_radiographs(...)
7      def get_radiographs_info(...)
8      def predict(...

```

- **OperationController:** gestisce la pianificazione e il tracciamento delle operazioni mediche, mantiene le relazioni tra pazienti e interventi programmati.

Listing 9: File `operation_controller.py`.

```

1  class OperationController:
2      def __init__(self, managers)
3      def add_operation(...)
4      def get_patient_operations(...

```

- **NotificationController:** implementa il sistema di notifiche, gestisce l'invio e il tracciamento delle comunicazioni tra il sistema e gli utenti.

Listing 10: File `notification_controller.py`.

```

1  class NotificationController:
2      def __init__(self, managers)
3      def send_notification(...)
4      def get_notifications(...)
5      def mark_notification_as_read(...

```

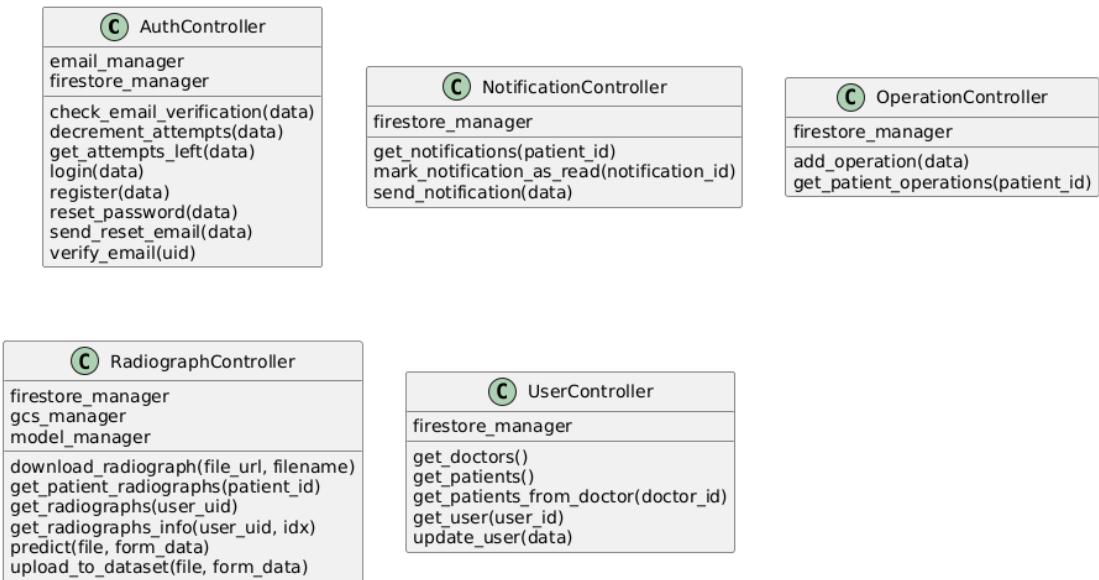


Figura 21: Diagramma delle classi: modulo Controller, ogni classe è indipendente e si occupa di un dominio differente, totale separazione delle responsabilità.

Ogni controller implementa una REST API che espone le proprie funzionalità attraverso endpoint HTTP, seguendo le best practice REST. La gestione degli errori è centralizzata, con ogni controller che implementa appropriate strategie di error handling e reporting.

L’architettura adottata permette una facile estensione del sistema: nuove funzionalità possono essere aggiunte implementando nuovi controller o estendendo quelli esistenti, senza impattare sul resto del sistema.

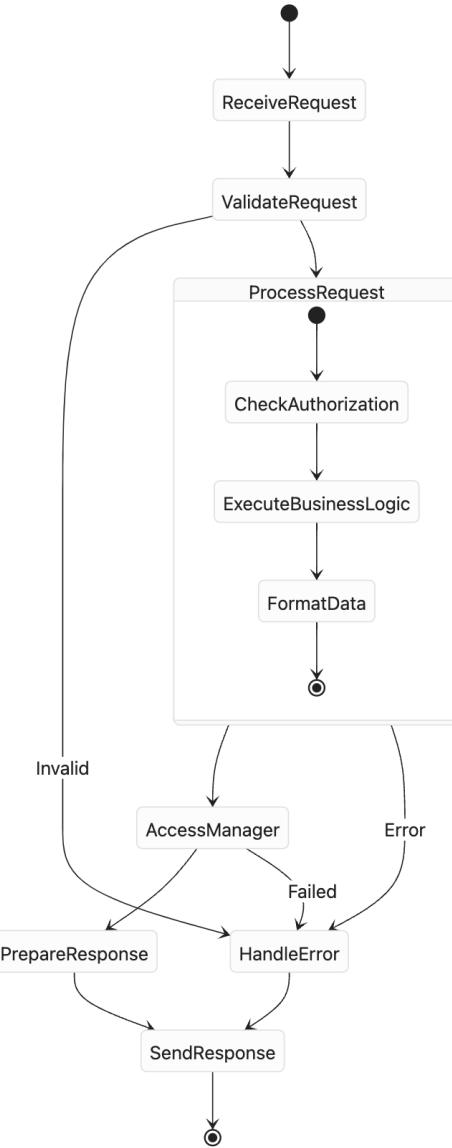


Figura 22: Diagramma delle attività: modulo Controller.

5.1.3.1 Modulo Factory

Il modulo Factory implementa il pattern Factory, centralizzando l'inizializzazione dei manager del sistema. Questa architettura è stata progettata per gestire in modo efficiente le dipendenze dell'applicazione e garantire una corretta inizializzazione delle risorse.

Il modulo si basa sulla classe `ManagerFactory` che, seguendo il principio di Single Responsibility, crea e configura i diversi manager del sistema. L'utilizzo del pattern Factory permette di encapsulare la logica di creazione degli oggetti, nascondendo i dettagli implementativi e fornendo

un’interfaccia pulita per istanziare i manager.

Listing 11: File `manager_factory.py`.

```
1 # Classe per inizializzare tutti i manager (sfruttando le chiavi di AppConfig)
2 class ManagerFactory:
3     @staticmethod
4     def create_managers(app_config):
5         # Inizializza Firestore
6         db = firestore.client()
7         firestore_manager = FirestoreManager(db)
8
9         # Inizializza GCS
10        gcs_manager = GCSManager(AppConfig.GCS_BUCKET_NAME)
11
12        # Inizializza Model
13        model = gcs_manager.load_model(AppConfig.MODEL_PATH)
14        model_manager = ModelManager(model)
15
16        # Inizializza Email
17        email_manager = EmailManager(
18            sender_email=AppConfig.SMTP_USERNAME,
19            sender_password=AppConfig.SMTP_PASSWORD
20        )
21
22        return {
23            'firestore': firestore_manager,
24            'gcs': gcs_manager,
25            'model': model_manager,
26            'email': email_manager
27        }
```

L’architettura sfrutta `AppConfig` (modulo `Config`), che mantiene tutti i parametri di configurazione necessari per l’inizializzazione dei manager. L’uso di metodi statici nella factory riflette la natura utility-oriented del modulo, non è infatti necessario istanziare la factory stessa.

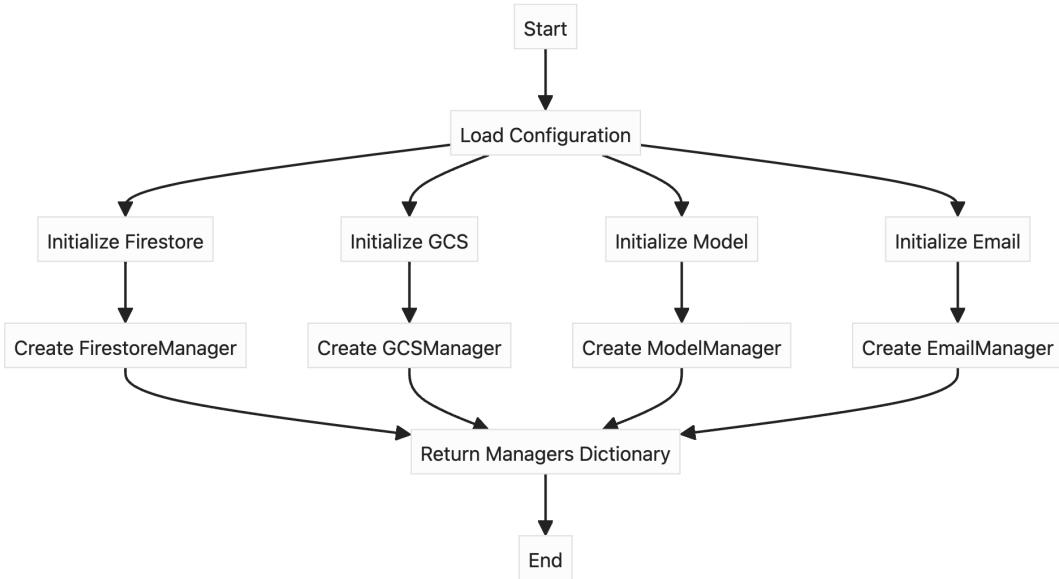


Figura 23: Diagramma di flusso: modulo factory

5.1.3.2 Modulo Routing

Il modulo Routing implementa il pattern **Front Controller**: centralizza la gestione di tutte le richieste HTTP dell'applicazione. Infatti questo modulo funge da punto di ingresso per tutte le richieste client, orchestrando il loro instradamento verso i controller appropriati. L'architettura del routing è organizzata secondo una struttura REST, dove le routes sono raggruppate logicamente in base al dominio di competenza (Auth, User, Operation, Notification, Radiograph).

Listing 12: Rotte definite nel file `api_routes.py`

```

1 def register_routes(app, controllers):
2     # Auth Routes
3     @app.route('/register', methods=['POST'])
4     @app.route('/login', methods=['POST'])
5     @app.route('/check-email-verification', methods=['POST'])
6     @app.route('/verify-email/<string:uid>', methods=['GET'])
7     @app.route('/reset-password', methods=['POST'])
8     @app.route('/send-reset-email', methods=['POST'])
9     @app.route('/decrement-attempts', methods=['POST'])
10    @app.route('/get-attempts-left', methods=['POST'])
11
12    # User Routes
13    @app.route('/api/get_user/<user_id>', methods=['GET'])
14    @app.route('/update_user', methods=['PATCH'])
15    @app.route('/api/doctors', methods=['GET'])
16    @app.route('/api/patients', methods=['GET'])
17    @app.route('/api/<doctor_id>/patients', methods=['GET'])
18
19    # Operation Routes
20    @app.route('/api/operations', methods=['POST'])
21    @app.route('/api/patients/<patient_id>/operations', methods=['GET'])

```

```

23     # Notification Routes
24     @app.route('/api/notifications', methods=['POST'])
25     @app.route('/api/notifications', methods=['GET'])
26     @app.route('/api/notifications/<notification_id>', methods=['PATCH'])
27
28     # Radiograph Routes
29     @app.route('/api/patients/<patient_id>/radiographs', methods=['GET'])
30     @app.route('/api/download-radiograph', methods=['GET'])
31     @app.route('/upload-to-dataset', methods=['POST'])
32     @app.route('/get_radiographs/<user_uid>', methods=['GET'])
33     @app.route('/get_radiographs_info/<user_uid>/<idx>', methods=['GET'])
34     @app.route('/predict', methods=['POST'])

```

Ogni route è configurata con il proprio metodo HTTP appropriato (GET, POST, PATCH) seguendo le convenzioni REST, e include la gestione dei parametri necessari, sia che provengano dal body della richiesta (`request.json`), dai parametri URL, o dai form data (`request.files`, `request.form`). L'architettura adottata permette una facile estensione del sistema: nuove routes possono essere aggiunte in modo strutturato, mantenendo la coerenza con il pattern esistente e la separazione delle responsabilità.

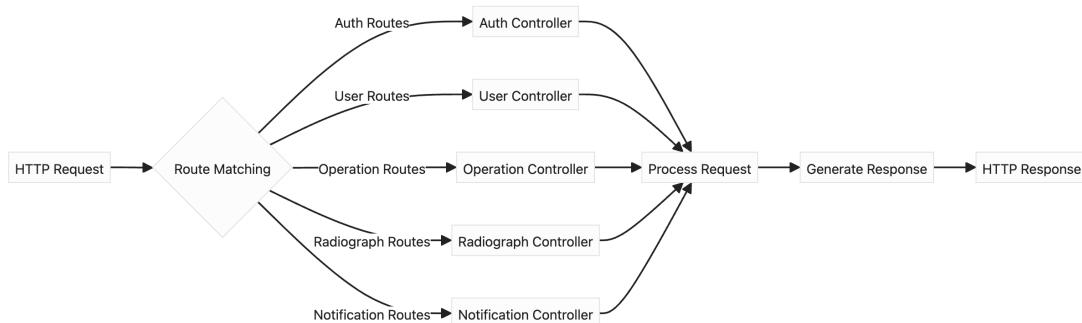


Figura 24: Diagramma di flusso: modulo Routing

5.1.3.3 Modulo Config

Il modulo Config è responsabile della gestione della configurazione del sistema attraverso la classe `AppConfig`. Questa classe segue il principio della configurazione esterna, consentendo di definire i parametri critici del sistema tramite variabili d'ambiente, pur garantendo la presenza di valori di default appropriati per una configurazione predefinita e robusta.

Anche in questo modulo, si è scelto di adottare una struttura organizzata in sezioni logiche, ciascuna dedicata a un ambito specifico. Le configurazioni includono:

- Configurazioni di ambiente: per abilitare e controllare la modalità di debug.
- Impostazioni CORS: per gestire le richieste cross-origin in modo sicuro.
- Percorsi delle credenziali: necessari per integrare i servizi di Firebase e Google Cloud Storage.
- Configurazioni SMTP: per il servizio di invio email.
- Percorsi dei modelli di machine learning: per il caricamento e l'uso degli algoritmi di apprendimento automatico.

Listing 13: File `app_config.py`.

```

1 class AppConfig:
2     # Environment
3     DEBUG = os.environ.get('DEBUG', 'True') == 'True'
4
5     # LOCALE
6     CORS_ORIGIN = os.environ.get('CORS_ORIGIN', 'http://localhost:8080')
7
8     # VM
9     # CORS_ORIGIN = os.environ.get('CORS_ORIGIN', 'http://34.122.99.160:8080')
10
11    # Firebase
12    FIREBASE_CRED_PATH = os.path.join(
13        os.path.abspath(os.path.dirname(__file__)),
14        '..',
15        'config',
16        'firebase-adminsdk.json',
17    )
18
19    # Google Cloud Storage
20    GCS_CRED_PATH = os.path.join(
21        os.path.abspath(os.path.dirname(__file__)),
22        '..',
23        'config',
24        'meta-geography-438711-r1-de4779cd8c73.json',
25    )
26    GCS_BUCKET_NAME = 'osteoarthritis-portal-archive'
27
28    # Email
29    SMTP_SERVER = os.environ.get('SMTP_SERVER', 'smtp.gmail.com')
30    SMTP_PORT = int(os.environ.get('SMTP_PORT', 587))
31    SMTP_USERNAME = os.environ.get('SMTP_USERNAME')
32    SMTP_PASSWORD = os.environ.get('SMTP_PASSWORD')
33
34    # Model
35    MODEL_PATH = 'MODELLO/pesi.h5'

```

5.1.3.4 Modulo UI

Il modulo UI rappresenta il livello di presentazione del sistema ed è implementato seguendo il pattern architettonale **MVVM (Model-View-ViewModel)**. Questa scelta permette una netta separazione tra la logica di presentazione e la logica di business, facilitando la manutenibilità e la scalabilità dell'applicazione, concetto più e più volte rimarcato in fase di design.

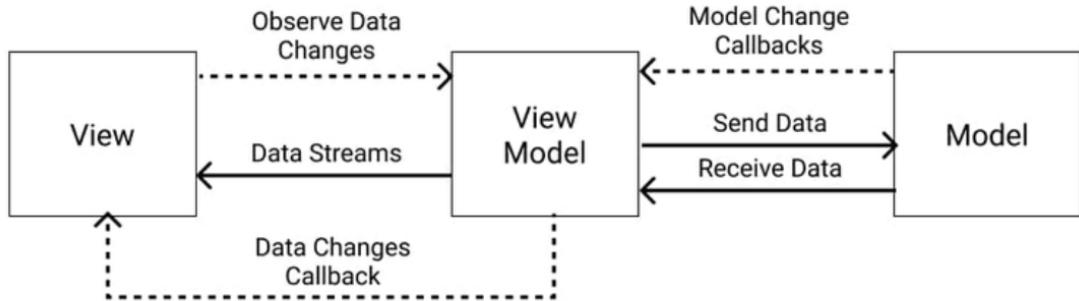


Figura 25: Diagramma di sequenza: modulo UI.

Come definito in fase di progettazione, l’interfaccia utente è strutturata come una **Single Page Application (SPA)**, consentendo una navigazione fluida senza ricaricamento delle pagine e il caricamento asincrono delle risorse. Inoltre, il sistema gestisce lo stato dell’applicazione lato client, comunicando in modo efficiente con il backend attraverso chiamate API.

L’architettura è basata su componenti Vue, dove ogni componente rappresenta un’unità funzionale indipendente e riutilizzabile. La reattività dei dati è garantita da meccanismi come le proprietà reattive interne (“data”) e quelle passate dai componenti padre ai figli (“props”). La comunicazione tra i componenti avviene tramite eventi personalizzati per le interazioni figlio-padre e mediante un event bus per quelle tra componenti non collegati direttamente. Inoltre, il binding bidirezionale dei dati è implementato usando la direttiva “v-model”.

Listing 14: Esempio di componente Vue (1/2).

```

1 <template>
2   <div class="verify-email">
3     <div class="container mt-5">
4       <h2 class="mb-4">{{ verificationMessage }}</h2>
5       <p v-if="errorMessage">{{ errorMessage }}</p>
6       <div class="btn-group mt-4" v-if="!isLoading && !errorMessage">
7         <button class="btn btn-primary btn-next" @click="goToLogin">
8           Esegui il login
9         </button>
10        </div>
11      </div>
12    </div>
13 </template>
  
```

L’interfaccia segue un design minimalistico, orientato alla chiarezza e all’usabilità. Il layout è responsive grazie all’uso di un sistema di griglie flessibile, mentre la paletta cromatica professionale basata su tonalità di blu trasmette affidabilità. La consistenza visiva è mantenuta attraverso l’uso di componenti riutilizzabili.

Listing 15: Esempio di componente Vue (2/2).

```
<script>
import { verifyEmail } from "@/services/api-service";

export default {
  name: "VerifyEmail",
  data() {
    return {
      verificationMessage: "Verificando la tua email...",
      errorMessage: null,
      isLoading: true,
    };
  },
  async mounted() {
    const uid = this.$route.params.uid;
    const response = await verifyEmail(uid);

    if (response.data.message === "Email gi verificata!") {
      this.verificationMessage = "La tua email gi stata verificata!";
    } else if (response.data.message === "Email verificata con successo!") {
      this.verificationMessage = "Email verificata con successo!";
    }
    this.isLoading = false;
  },
  methods: {
    goToLogin() { this.$router.push("/login"); },
  },
};
</script>
```

Per garantire prestazioni ottimali, il sistema sfrutta il **Virtual DOM** offerto da Vue, che consente di ottimizzare le renderizzazioni. La gestione delle dipendenze tra i componenti è stata pianificata per ridurre il carico computazionale e migliorare la reattività dell'applicazione.

I form sono validati in real-time con feedback immediato per l'utente, mentre gli errori sono gestiti centralmente e comunicati con messaggi chiari. Per le operazioni asincrone, vengono utilizzati stati di loading accompagnati da feedback visivi che rendono evidente lo stato delle operazioni all'utente.

In conclusione, il modulo UI è stato progettato e sviluppato per garantire una user experience di alto livello, la stessa che era stata pianificata all'inizio del progetto, con un'architettura scalabile e un design moderno, rispettando le **best practices** consolidate.

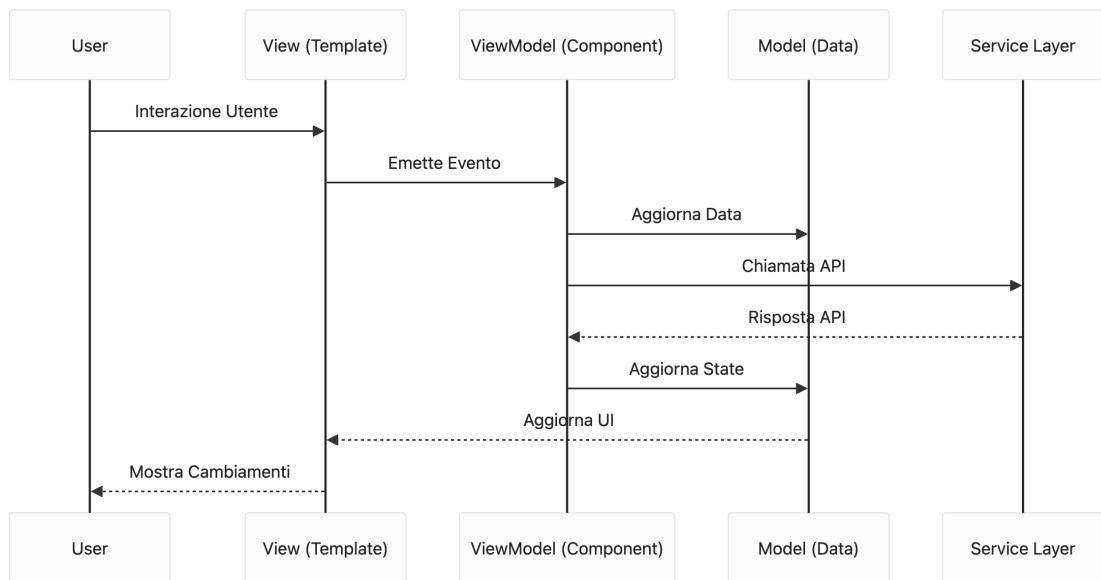


Figura 26: Diagramma di sequenza: modulo UI.

5.1.3.5 Modulo Service

Il modulo **Service** del Server è fondamentale, in quanto definisce l'API del Server stesso e gestisce l'esposizione del servizio ai client. L'intero funzionamento dell'applicazione si basa sul meccanismo di *Remote Procedure Call (RPC)*: i client inviano richieste al Server e attendono una risposta. Ogni richiesta è caratterizzata da tre elementi fondamentali:

- **URL**, che identifica la risorsa o operazione richiesta.
- **Metodo**, che specifica l'operazione da eseguire.
- **Risposta**, che include uno status code standard e, in caso di successo, il risultato dell'operazione.

Come specificato anche prima, viene usato il protocollo HTTP per le comunicazioni e consente di eseguire operazioni *CRUD (Create, Read, Update, Delete)* attraverso i seguenti metodi:

- **POST**: per creare una nuova risorsa.
- **GET**: per leggere o recuperare dati.
- **PUT**: per aggiornare o sostituire una risorsa esistente.
- **PATCH**: per modificare parzialmente una risorsa.
- **DELETE**: per rimuovere una risorsa.

Listing 16: Esempi di operazioni del file `api_service.js`.

```

export const getPatientsFromDoctor(...) // GET
export const checkEmailVerification(...) // POST
export const patchNotifications(...) // PATCH

```

Il modulo **Service** si basa su librerie come **Axios** e il pacchetto **Firebase**. Ogni funzione all'interno del modulo rappresenta un'interfaccia per un endpoint specifico del server, fornendo una forma semplificata per inviare richieste e gestire le risposte.

Ogni funzione costruisce richieste HTTP che puntano agli endpoint definiti nel backend.

Listing 17: Corpo della funzione `getPatientsFromDoctor()`.

```
// Funzione per ottenere i pazienti associati a un dottore specifico
export const getPatientsFromDoctor = async (doctorId) => {
  try {
    const response = await axios.get(`${API_URL}/api/${doctorId}/patients`, {
      headers: {
        "Content-Type": "application/json",
      },
    });

    return response.data;
  } catch (error) {
    console.error("Errore:", error);
    return [];
  }
};
```

5.2 Docker

5.2.1 Dockerfile per il frontend

Il Dockerfile per il frontend è basato su Node.js. Dopo aver impostato la directory di lavoro, i file `package.json` e `package-lock.json` vengono copiati per installare le dipendenze tramite `npm install`. Infine viene costruita l'applicazione Vue.js tramite `npm run build` esponendo la porta 8080 per permettere l'accesso all'applicazione.

Listing 18: Dockerfile per il frontend

```
# Usa un'immagine base di Node.js
FROM node:14

# Imposta la directory di lavoro
WORKDIR /osteoarthritis-project/frontend

# Copia il file package.json e package-lock.json
COPY package*.json .

# Installa le dipendenze
RUN npm install

# Copia il resto del codice sorgente
COPY .

# Costruisce il progetto Vue.js
RUN npm run build

# Espone la porta che l'applicazione utilizzerà
EXPOSE 8080

# Comando per avviare il server
CMD ["npm", "run", "serve"]
```

5.2.2 Dockerfile per il backend

Il Dockerfile per il backend è basato su un'immagine di Python 3.9. Dopo l'installazione dei pacchetti necessari per OpenCV, viene impostata la directory di lavoro e copiati i file del progetto. Successivamente, viene eseguita l'installazione delle dipendenze Python tramite il file `requirements-vm.txt`. Il Dockerfile espone la porta 5000, necessaria per l'esecuzione del server Flask, e avvia l'applicazione con il comando `python ./app.py`.

Listing 19: Dockerfile per il backend

```
FROM python:3.9

# Aggiunge i pacchetti necessari per OpenCV
RUN apt-get update && apt-get install -y \
    libgl1-mesa-glx \
    libglib2.0-0 \
    && rm -rf /var/lib/apt/lists/*

# Imposta la directory di lavoro
WORKDIR /osteoarthritis-project/backend/app

# Copia i file del progetto nella directory di lavoro
COPY .

# Installa le dipendenze dal file requirements.txt
RUN pip install --no-cache-dir -r requirements-vm.txt

# Espone la porta 5000 per il server Flask
EXPOSE 5000

# Comando per avviare l'applicazione
CMD ["python", "./app.py"]
```

5.2.3 Orchestrazione dei container: docker-compose.yml

Il file `docker-compose.yml` definisce due container separati, uno per il backend e uno per il frontend del progetto. Il container del backend espone la porta 5000, mentre quello del frontend espone la porta 8080. Entrambi sono connessi alla stessa rete (`app-network`), che consente la comunicazione tra essi.

Listing 20: docker-compose.yml

```
1 version: "3"
2 services:
3   backend:
4     image: andyalemonta/backend-docker-image:latest
5     container_name: backend_container
6     restart: always
7     ports:
8       - "5000:5000"
9     networks:
10      - app-network
11
12   frontend:
13     image: andyalemonta/frontend-docker-image:latest
14     container_name: frontend_container
15     restart: always
16     ports:
17       - "8080:8080"
18     networks:
19       - app-network
20
21 networks:
22   app-network:
23     driver: bridge
```

6 Test

Per la fase di testing sono state adottati **test automatizzati**, **test manuali** e il **coinvolgimento del chirurgo esperto del dominio**.

Il testing manuale ha permesso di valutare le funzionalità principali e identificare eventuali anomalie, simulando il flusso di lavoro di medici e pazienti. Particolare attenzione è stata posta sulla chiarezza dell'interfaccia e sulla sua intuitività.

Il sistema è stato anche testato dal chirurgo esperto del dominio, che ha fornito feedback sul suo utilizzo. Nonostante i contatti con lui siano stati sporadici, le sue osservazioni hanno suggerito miglioramenti nell'organizzazione dell'interfaccia. Questi spunti sono stati utili per rendere il sistema più adatto al lavoro quotidiano di un medico.

6.1 Testing delle funzionalità

Il testing si è concentrato sulla verifica delle funzionalità principali del sistema per garantire che tutte le componenti operassero correttamente secondo i requisiti definiti. Ogni modulo è stato testato per assicurarne l'affidabilità e l'integrazione all'interno del flusso complessivo del sistema. I principali test effettuati sono stati i seguenti:

- **autenticazione e gestione degli utenti:** è stato testato il flusso di registrazione, login e logout, assicurandosi che il sistema gestisse correttamente l'autenticazione degli utenti, la gestione dei permessi e la protezione dei dati sensibili. Inoltre sono stati verificati i meccanismi di gestione degli account, inclusi reset della password, gestione delle sessioni utente e invio della mail di verifica.
- **caricamento e visualizzazione delle radiografie:** sono stati effettuati test per garantire che le radiografie fossero caricate correttamente nel sistema e visualizzate in modo accurato. È stata testata anche la generazione delle heat map associate alle immagini, per assicurarsi che riflettessero correttamente le aree critiche identificate dal modello di intelligenza artificiale.
- **dashboard e calendario:** il sistema di dashboard e calendario è stato testato per verificare la sincronizzazione dei dati in tempo reale. È stato controllato che gli eventi pianificati, come le operazioni e le consultazioni, venissero visualizzati correttamente, e che le interazioni con l'interfaccia fossero fluide e intuitive.
- **front-end e back-end:** Sono stati testati i punti di integrazione tra il front-end e il back-end, concentrandosi sulle API utilizzate per la comunicazione tra le interfacce utente e i sistemi di backend. È stata verificata la corretta gestione delle richieste e delle risposte.
- **rete neurale e gestione delle richieste:** È stato testato il flusso di dati tra la rete neurale e la gestione delle richieste. In particolare, è stata verificata l'elaborazione delle radiografie, il passaggio delle informazioni al modello di intelligenza artificiale, e il ritorno delle diagnosi in modo accurato e tempestivo.
- **notifiche:** Sono stati testati i meccanismi di sincronizzazione tra le notifiche e il calendario per assicurarsi che gli utenti ricevevano le notifiche in tempo reale e con la giusta accuratezza. È stato controllato che gli eventi pianificati nel calendario fossero correttamente riflessi nelle notifiche inviate ai medici e ai pazienti.

6.1.1 Testing automatizzato

Per eseguire test automatizzati sulle principali rotte e funzioni del sistema è stato utilizzato **Pytest**, un framework per il testing in Python. In particolare, è stato creato un file chiamato `conftest.py`, che viene interpretato automaticamente da Pytest come configurazione generale.

Listing 21: Contenuto del file `conftest.py`

```
1 import pytest
2 import sys
3 import os
4 from unittest import mock
5 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '../
6     backend')))

7 # Fixture per il client Flask
8 @pytest.fixture
9 def client():
10     """Fixture per il client Flask."""
11     from app import app
12     #print("Registered routes:", app.url_map)
13     app.config['TESTING'] = True
14     with app.test_client() as client:
15         yield client
```

Nel codice riportato nel listing si nota come il file di configurazione è stato utilizzato per:

- **configurare l'ambiente di test**: aggiungendo ai path di sistema la cartella contenente il backend.
- **fornire la fixture client**: che crea un'istanza dell'applicazione Flask in modalità test.

Grazie a questa configurazione i vari test possono inviare richieste HTTP senza dover replicare ogni volta la creazione e configurazione del client.

Per garantire una verifica completa del comportamento del sistema, i test automatizzati sono stati raggruppati in base alle principali aree funzionali dell'applicazione. Ogni gruppo di test si concentra su un preciso ambito (ad esempio la gestione dell'autenticazione, delle notifiche o delle operazioni), in modo da coprire tutti i possibili casi rilevanti.

Per rendere i test più compatti, estensibili ed efficaci, è stato utilizzato il decoratore `pytest.mark.parametrize` di **Pytest**. Questo approccio ha permesso di eseguire più test con diverse combinazioni di parametri senza dover duplicare il codice, migliorando così la leggibilità e la manutenzione.

Di seguito è riportato un test che verifica il comportamento dell'endpoint /get-patient-operations:

Listing 22: Esempio di test.

```
1 @pytest.mark.parametrize(
2     "patient_id, mock_side_effect, mock_return_docs, expected_status_code,
3      expected_value_substr",
4     [
5         # Dati di test
6         ("pat123", None, [{"id": "op1", "description": "Operazione 1"}, {"id": "op2",
7             "description": "Operazione 2"}], 200, "op1"),
8         ("patError", Exception("Errore Firestore"), None, 500, "Errore Firestore")
9     ],
10 )
11
12 @patch("utils.firebaseio_utils.FirestoreManager.query_documents")
13 def test_get_patient_operations(
14     mock_query_documents,
15     patient_id,
16     mock_side_effect,
17     mock_return_docs,
18     expected_status_code,
19     expected_value_substr,
20     client
21 ):
22     # Configurazione del mock
23     if mock_side_effect:
24         mock_query_documents.side_effect = mock_side_effect
25     else:
26         mock_query_documents.return_value = mock_return_docs or []
27
28     # Costruzione URL e richiesta GET
29     url = f"/api/patients/{patient_id}/operations"
30     response = client.get(url)
31     json_data = response.get_json()
32
33     # Asserzioni
34     assert response.status_code == expected_status_code, json_data
35     if expected_status_code == 200:
36         # Caso di successo -> JSON con lista di operazioni
37         assert isinstance(json_data, list)
38         assert any(expected_value_substr in str(op) for op in json_data)
39     else:
40         # Caso di errore -> JSON con chiave "error"
41         assert "error" in json_data
42         assert expected_value_substr in json_data["error"]
```

Viene riportato di seguito l'elenco di tutti i test effettuati:

Controller	Test
Auth	<code>test_register_user</code> <code>test_login_success</code> <code>test_login_missing_id_token</code> <code>test_login_invalid_token</code> <code>test_login_parametrized</code> <code>test_check_email_verification</code> <code>test_verify_email</code> <code>test_decrement_attempts</code> <code>test_get_attempts_left</code> <code>test_send_reset_email</code>
Notification	<code>test_send_notification</code> <code>test_get_notifications</code> <code>test_mark_notification_as_read</code>
Operation	<code>test_add_operation</code> <code>test_get_patient_operations</code>
Radiograph	<code>test_get_patient_radiographs</code> <code>test_download_radiograph</code> <code>test_upload_to_dataset</code> <code>test_get_radiographs</code> <code>test_get_radiographs_info</code> <code>test_predict</code>
User	<code>test_get_user</code> <code>test_update_user</code> <code>test_get_doctors</code> <code>test_get_patients</code> <code>test_get_patients_from_doctor</code>

Tabella 1: Elenco dei test divisi per tipologia di controller

6.2 Testing delle prestazioni e della sicurezza

Sono state valutate le prestazioni del sistema esaminando due aspetti: i tempi di risposta e la scalabilità. I tempi di risposta sono stati misurati durante il caricamento delle radiografie e la generazione delle diagnosi, per garantire che il sistema fosse in grado di elaborare rapidamente le informazioni, anche in condizioni di utilizzo intensivo. In particolare è stata valutata la rapidità con cui le radiografie venivano visualizzate e le diagnosi venivano generate, per assicurarsi che il sistema fosse sufficientemente veloce da supportare un flusso di lavoro senza interruzioni.

Per quanto riguarda la scalabilità sono stati condotti stress test su carichi elevati per verificare la robustezza dell'architettura del sistema. Questi test hanno simulato scenari in cui venivano caricati grandi volumi di dati, per assicurarsi che il sistema potesse gestire un aumento del nume-

ro di utenti e delle dimensioni dei dataset senza compromettere le prestazioni. In questo modo, è stato confermato che l'architettura potesse supportare carichi di lavoro elevati senza degradare la velocità di risposta o la qualità delle operazioni.

La protezione dagli accessi non autorizzati è stata testata simulando tentativi di violazione, al fine di verificare la solidità delle policy di accesso. Sono stati testati diversi livelli di autorizzazione e le misure di protezione per garantire che solo gli utenti legittimi avessero accesso ai dati sensibili.

6.3 Testing dell'Esperienza Utente (UX)

Le funzionalità del sistema sono state testate su una vasta gamma di dispositivi e browser (tra cui Chrome, Opera e Firefox), verificando la correttezza e la portabilità dell'applicazione. È stata posta particolare attenzione nel garantire che l'accesso e il funzionamento del sistema fossero ottimali a partire dal link, indipendentemente dal dispositivo utilizzato. Inoltre, è stato verificato che l'interfaccia fosse ben visibile e utilizzabile anche su dispositivi mobili, adattandosi alle diverse risoluzioni e mantenendo la stessa qualità visiva e funzionale.

L'usabilità è stata ulteriormente validata utilizzando l'**euristica di Nielsen**, che ha portato alle seguenti osservazioni:

- **controllo e libertà:** Le operazioni necessarie per completare un task sono state ridotte al minimo per abbassare la complessità
- **aiuto utente:** Messaggi informativi e notifiche guidano l'utente in caso di errori, come durante l'inserimento di dati non validi
- **prevenzione errori:** Il sistema previene situazioni che potrebbero causare errori, fornendo indicazioni chiare e intuitive
- **riconoscimento più che ricordo:** L'interfaccia utilizza immagini e pulsanti autoesplicativi e uno stile uniforme per facilitare la navigazione
- **visibilità dello stato del sistema:** Latenze e feedback visivi sono stati ottimizzati per mantenere informato l'utente durante l'interazione
- **corrispondenza con il mondo reale:** Icone e termini utilizzati sono facilmente comprensibili e in linea con il dominio medico
- **consistenza e standard:** Elementi visivi e colori sono stati uniformati per enfatizzare gli aspetti più rilevanti
- **estetica e progettazione minimalista:** Il design segue il principio KISS, presentando solo le funzionalità essenziali per ogni schermata.
- **flessibilità ed efficienza:** I menù sono stati progettati per essere semplici e accessibili senza la necessità di scorciatoie complesse
- **documentazione**

Oltre a questo è stato anche condotto un **Cognitive Walkthrough**. Durante il processo, sono state analizzate diverse aree, tra cui la comprensione, la visibilità, l'adeguatezza delle azioni e il feedback fornito dal sistema. Il processo di Cognitive Walkthrough ha portato alla formulazione di domande come:

- **comprendere del sotto-task:**
esempio: L'utente comprende che deve cliccare sul pulsante “Accedi” per entrare nel sistema?
- **visibilità dell'azione necessaria:**
esempio: Il pulsante “Registrati” è chiaramente visibile nella schermata di benvenuto?
- **comprendere del risultato dell'azione:**
esempio: Dopo aver cliccato su “Registrati”, l'utente comprende che questa azione porterà alla creazione di un account?
- **adeguatezza del feedback:**
esempio: Dopo aver inviato un modulo, il sistema mostra un messaggio di conferma chiaro come “Registrazione completata con successo”?

6.4 Esiti e analisi critica

I risultati del processo di testing hanno evidenziato sia punti di forza significativi che aree di miglioramento che richiedono attenzione.

Tra i punti di forza, è emersa un'alta precisione nelle predizioni fornite dal modello di intelligenza artificiale, che ha dimostrato di essere altamente affidabile nel supportare le decisioni mediche. L'interfaccia utente si è rivelata semplice da usare, con un design intuitivo che ha facilitato l'interazione da parte dei medici e dei pazienti. Inoltre, la robustezza della gestione dei dati è stata confermata con una gestione sicura delle informazioni sensibili, senza perdite o errori nei trasferimenti.

D'altro canto, sono state identificate alcune aree di miglioramento. In particolare, è stato necessario ottimizzare le performance del sistema, soprattutto quando si trattano dataset molto grandi. Nonostante la buona capacità di gestione dei dati, l'elaborazione di volumi elevati di informazioni ha mostrato alcuni rallentamenti nelle performance. Oltre a questo, emerge chiaramente la possibilità di migliorare il progetto non solo ottimizzando la prova gratuita offerta dalla piattaforma di Google, che **consentirebbe di effettuare l'addestramento periodico senza limitazioni di risorse**, ma anche **integrando la rete neurale generativa (WGAN), già sviluppata** e pronta per essere inclusa. I dati generati dalla WGAN andrebbero poi trasferiti alla rete neurale predittiva in modo da migliorare ulteriormente la sua capacità di previsione.

7 Deployment

Il sistema è stato distribuito utilizzando **Docker** su una macchina virtuale configurata su **Google Compute Engine**.

7.1 Creazione delle immagini Docker

Le immagini Docker per il frontend e il backend sono state create direttamente sulla VM, utilizzando i Dockerfile descritti in precedenza, e contengono tutti gli elementi necessari per eseguire i rispettivi servizi, inclusi i file sorgenti, le dipendenze e le configurazioni.

Sono di seguito riportati gli output dei due comandi utilizzati per costruire le immagini:

```
andyalemonta@osteopathitis-vm:~/osteopathitis-project/frontend$ docker build -t frontend-docker-image .          docker:default
[+] Building 73.3s (12/12) FINISHED
   => [internal] load build definition from Dockerfile
   => [internal] load metadata for docker.io/library/node:14
   => [internal] load metadata for docker.io/library/node:14
   => [auth] library/node:pull token for registry-1.docker.io
   => [internal] load .dockerignore
   => => transferring context: 28
   => [1/6] FROM docker.io/library/node:14@sha256:a158d3b9b4a3fa813fa6c8c590b8f0a860e015ad4a59bbca5744d2f6fd8461aa
   => [internal] load build context
   => => transferring context: 4.54MB
   => [2/6] WORKDIR /osteopathitis-project/frontend
   => [3/6] COPY package*.json .
   => [4/6] RUN npm install
   => [5/6] COPY . .
   => [6/6] RUN npm run build
   => exporting to image
   => exporting layers
   => => Writing image sha256:b1b9336746fabc39d8e1431a37b6d4a8fc49567af473664727a401c016e3a69
   => => naming to docker.io/library/frontend-docker-image
0.0s
0.0s
0.3s
0.0s
0.0s
0.0s
0.0s
0.0s
0.0s
0.0s
2.5s
2.3s
0.0s
0.0s
0.0s
0.0s
35.4s
26.1s
8.9s
8.7s
0.0s
0.0s
```

Figura 27: Costruzione dell'immagine Docker per il frontend

```
andyalemonta@osteopathitis-vm:~/osteopathitis-project/backend$ docker build -t backend-docker-image .          docker:default
[+] Building 86.5s (10/10) FINISHED
   => [internal] load build definition from Dockerfile
   => [internal] load metadata for docker.io/library/python:3.9
   => [internal] load .dockerignore
   => => transferring context: 28
   => [1/5] FROM docker.io/library/python:3.9@sha256:dd8b65c39a729f946398d2e03a3e6defc8c0cfec409b9f536200634ad6408b54
   => [internal] load build context
   => => transferring context: 42.21kB
   => [2/5] RUN apt-get update && apt-get install -y libgl1-mesa-glx libgl1b2.0-0 && rm -rf /var/lib/apt/lists/*
   => [3/5] COPY .
   => [4/5] RUN pip install --no-cache-dir -r requirements-vm.txt
   => exporting to image
   => exporting layers
   => => writing image sha256:74a8ec7fe1ad8be17653e7b9f725cba131801513863e8a717372e6081e8d183
   => => naming to docker.io/library/backend-docker-image
0.0s
0.0s
0.2s
0.0s
65.7s
17.4s
0.0s
0.0s
0.0s
```

Figura 28: Costruzione dell'immagine Docker per il backend

7.2 Tagging delle immagini Docker

Dopo aver creato le immagini, è stato eseguito il comando di **tagging** per assegnare un'etichetta univoca alle immagini:

```
docker tag frontend-docker-image:latest andyalemonta/frontend-docker-image:latest
docker tag backend-docker-image:latest andyalemonta/backend-docker-image:latest
```

7.3 Pubblicazione su Docker Hub

Successivamente, le immagini sono state pubblicate su **Docker Hub** per consentirne l'accesso da qualsiasi ambiente. Questo passaggio garantisce che le immagini siano disponibili globalmente e possano essere utilizzate per distribuire i servizi su altri sistemi.

Sono di seguito riportati gli output della console durante il caricamento delle immagini.

```
andyalemonta@osteoathritis-portal-vm:~$ docker push andyalemonta/frontend-docker-image:latest
The push refers to repository [docker.io/andyalemonta/frontend-docker-image]
b9dd38c96411: Pushed
22ad928f4d5d: Pushed
5ad457866708: Mounted from andyalemonta/frontend-image
bdafe0359a149: Mounted from andyalemonta/frontend-image
a11ed1ab3f37: Mounted from andyalemonta/frontend-image
0d5f5a015e5d: Mounted from andyalemonta/frontend-image
3c777d951de2: Mounted from andyalemonta/frontend-image
f8a91dd5fc84: Mounted from andyalemonta/frontend-image
cb81227abde5: Mounted from andyalemonta/frontend-image
e01a454893a9: Mounted from andyalemonta/frontend-image
c45660adde37: Mounted from andyalemonta/frontend-image
fe0fb3ab4a0f: Mounted from andyalemonta/frontend-image
f1186e5061f2: Mounted from andyalemonta/frontend-image
b2dba7477754: Mounted from andyalemonta/frontend-image
latest: digest: sha256:7b337ab36d047a2cbd06c33ca08f76f84b62cf8bd4c23b87b8ef9b295facf003 size: 3268
```

Figura 29: Pubblicazione dell'immagine Docker per il frontend

```
andyalemonta@osteoathritis-portal-vm:~$ docker push andyalemonta/backend-docker-image:latest
The push refers to repository [docker.io/andyalemonta/backend-docker-image]
d5ba5741d6dd: Pushed
c845927df1f6: Pushed
80b997af8827: Pushed
2bf6136f8557: Pushed
fe5bbd4f8a42: Mounted from library/python
24f0c2413cd7: Mounted from library/python
8f9a13bfb118: Mounted from library/python
0aeee87c293d: Mounted from library/python
c81d4fdb67fc: Mounted from library/python
0e82d78b3ea1: Mounted from library/python
301c1bb42cc0: Mounted from library/python
latest: digest: sha256:5e8a3b20b87155bc987685b51893264d74a01825318dc9f961bfe1fe13ce07b3 size: 2637
```

Figura 30: Pubblicazione dell'immagine Docker per il backend

7.4 Deploy del sistema con Docker Compose

In Figura 31 è mostrato il processo di esecuzione del comando `docker-compose up --build -d` sulla VM, che avvia i container del sistema.

```
andyalemonta@osteoathritis-portal-vm:~$ docker compose up --build -d
[+] Running 3/3
✓ Network andyalemonta_app-network Created
✓ Container backend_container Started
✓ Container frontend_container Started
```

Figura 31: Deploy del sistema.

Una volta avviato il sistema, il sito è accessibile tramite il link <http://34.121.167.35:8080/>, dove l'IP corrisponde all'indirizzo pubblico della macchina virtuale che ospita l'applicazione.

8 Esempi di utilizzo

8.1 Registrazione e Login

Si suppone che un medico e un paziente vogliano utilizzare il sistema per caricare e analizzare radiografie. Inizialmente, entrambi visualizzano la schermata iniziale del sito, come mostrato in figura 32.

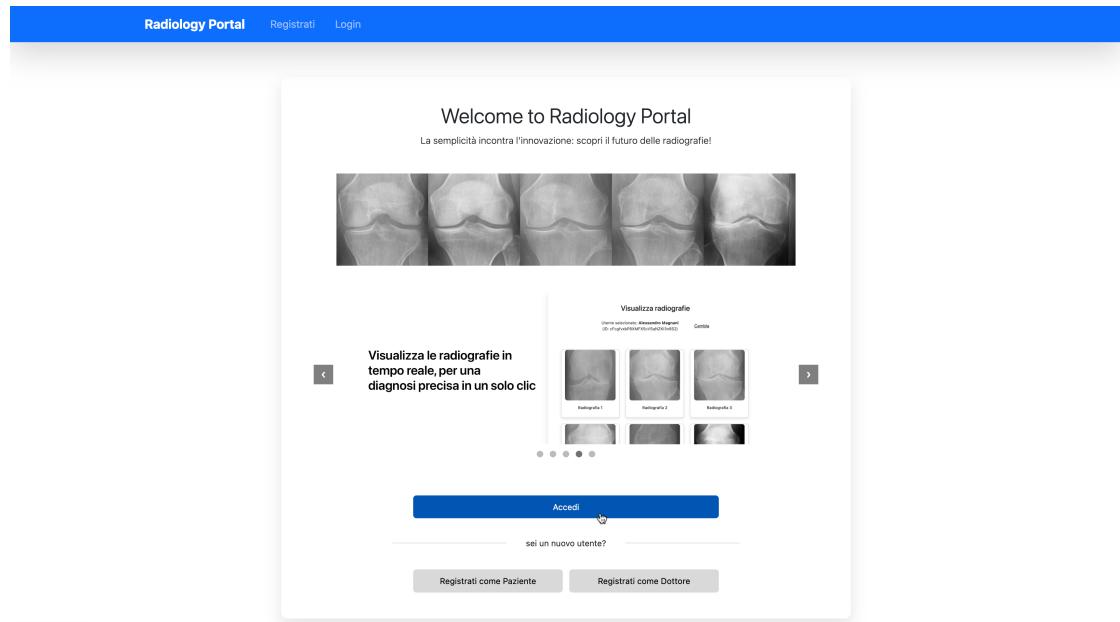


Figura 32: Schermata iniziale.

Entrambi devono registrarsi al sistema, seguendo un processo simile, con una piccola differenza nei campi richiesti durante la compilazione del modulo di registrazione.

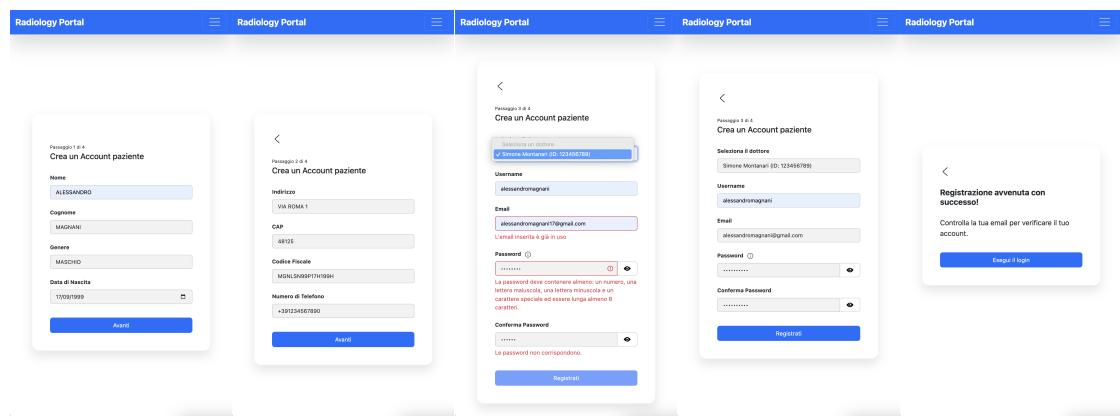


Figura 33: Schermate della registrazione.

Come si nota in figura 33, il processo di registrazione prevede i seguenti passaggi:

- **accesso alla pagina di registrazione.**
- **inserimento dei dati personali.**
 - medici: inseriscono il proprio codice identificativo univoco.
 - pazienti: selezionano il medico di riferimento.
- **verifica dell'indirizzo email.**

Completata la registrazione, gli utenti ricevono un'email di verifica all'indirizzo fornito. Per attivare l'account, è necessario cliccare sul link presente nella mail, come illustrato in Figura 34.

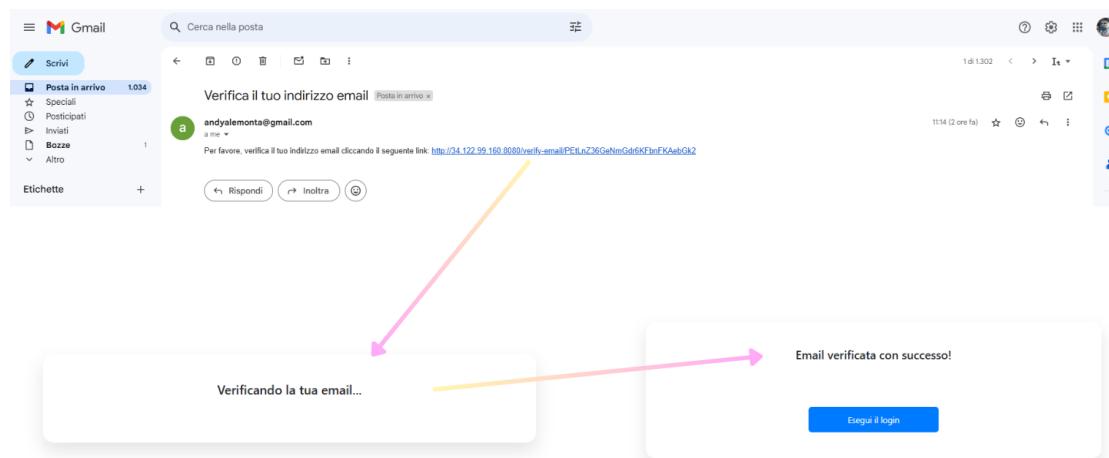


Figura 34: Processo di verifica dell'email.

Dopo aver verificato l'email tramite il link ricevuto, gli utenti possono accedere al sistema effettuando il login, come mostrato in figura 37.

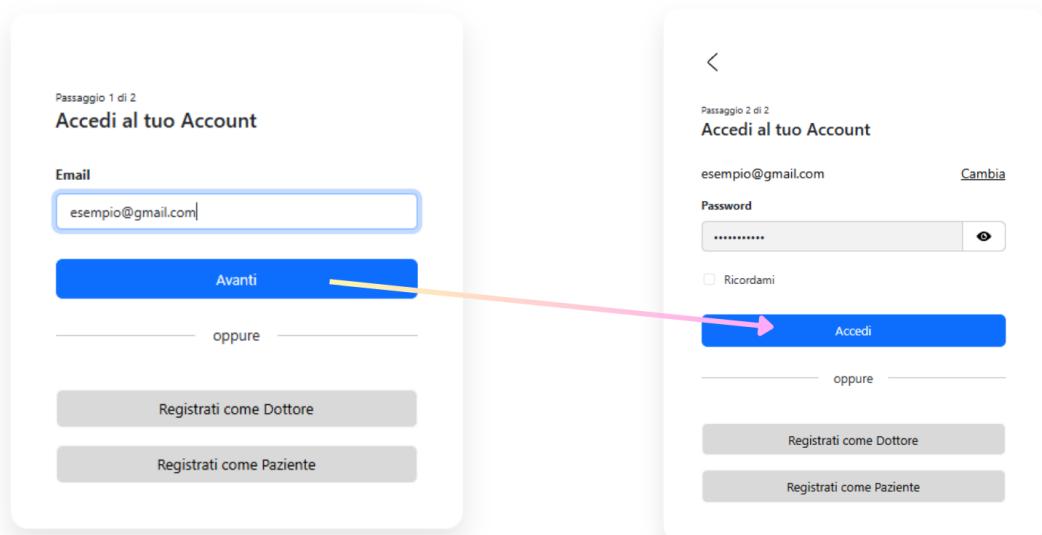


Figura 35: Processo di login.

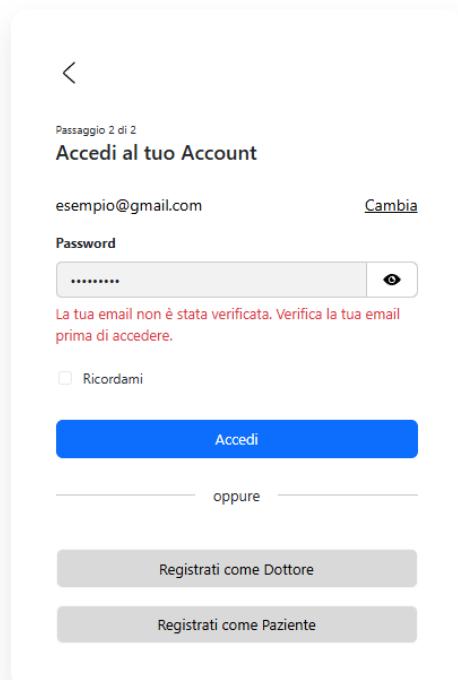


Figura 36: Email non verificata.

Nel caso in cui un utente inserisca una password errata ripetutamente, il sistema mostra un messaggio che indica i tentativi rimanenti. Quando i tentativi vengono esauriti, il sistema invia un'email per il reset della password.

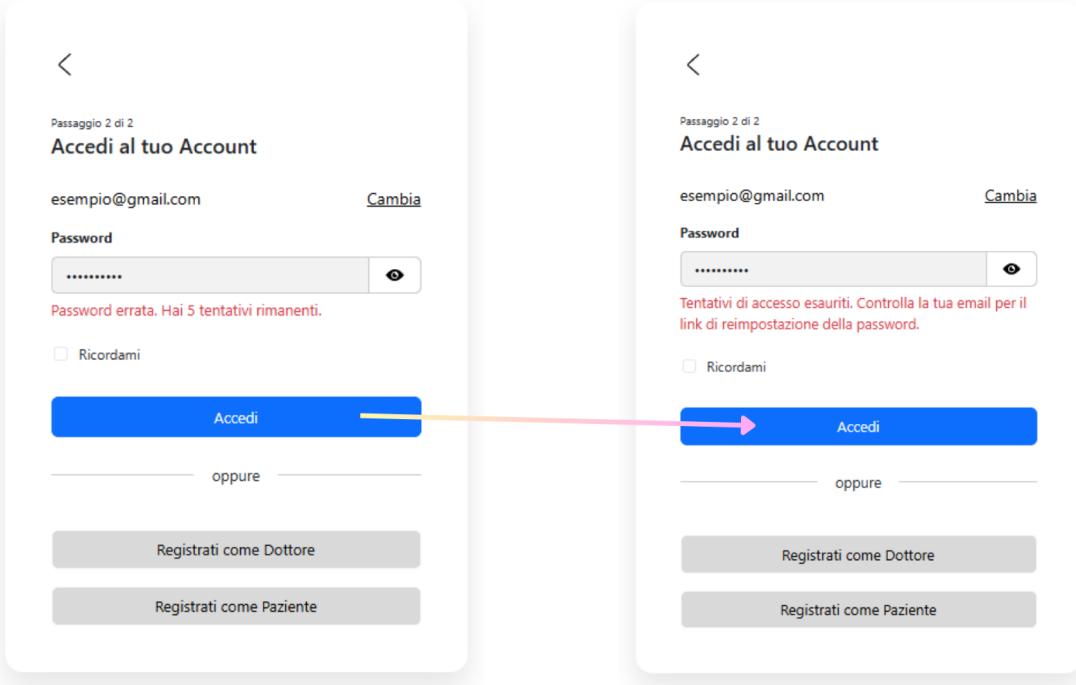


Figura 37: Errori nell'inserimento della password.

Una volta cliccato il link ricevuto via email, l'utente può reimpostare la propria password come mostrato in figura 41.

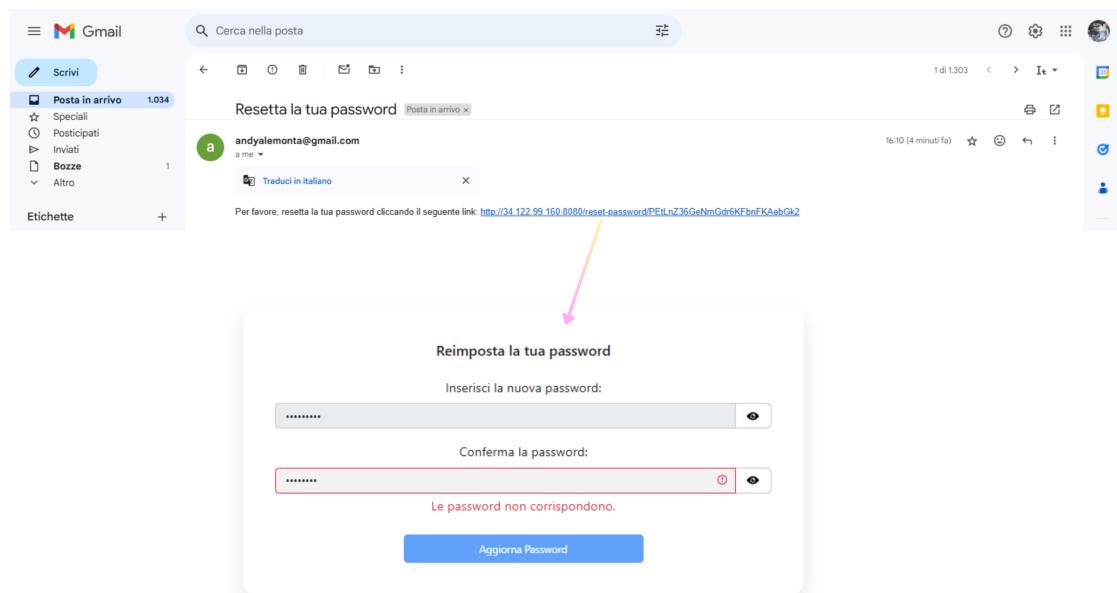


Figura 38: Reset della password.

8.2 Schermata di benvenuto e Barra di navigazione

Una volta autenticati, gli utenti vengono reindirizzati alla schermata di benvenuto, mostrata in Figura 39.

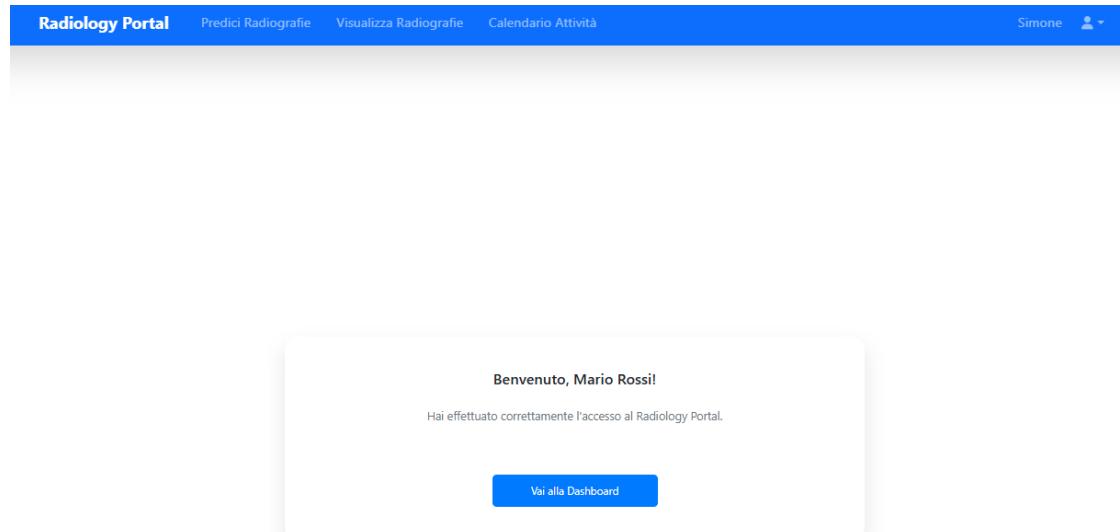


Figura 39: Schermata di benvenuto.

La figura 40 evidenzia le differenze nelle barre di navigazione di medici e pazienti.

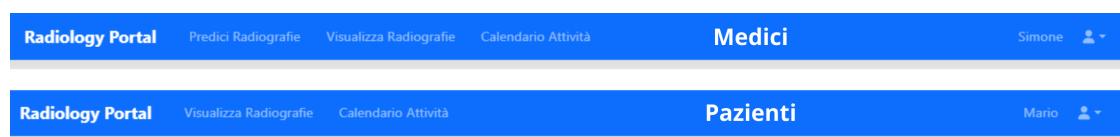


Figura 40: Differenze nelle barre di navigazione.

Questa schermata rappresenta il punto di accesso principale alle funzionalità del sistema:

- **dashboard.**
- **predizione dell'osteoartrite.**
- **visualizzazione radiografie.**
- **pianificazione di attività e notifiche.**

8.3 Dashboard

La dashboard rappresenta il punto di accesso centrale al sistema. I medici possono visualizzare un elenco completo delle informazioni anagrafiche e radiografie di tutti i pazienti a loro associati, mentre i pazienti possono consultare solo i propri dati.

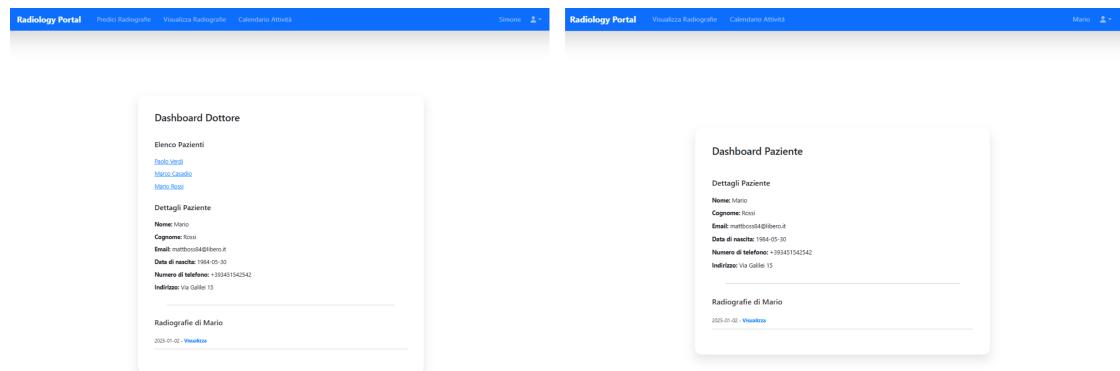


Figura 41: Dashboard per medici e pazienti.

8.4 Caricamento e Predizione

In questa sezione, i medici possono caricare le radiografie dei pazienti e utilizzare il modello pre-addestrato (figura 42).

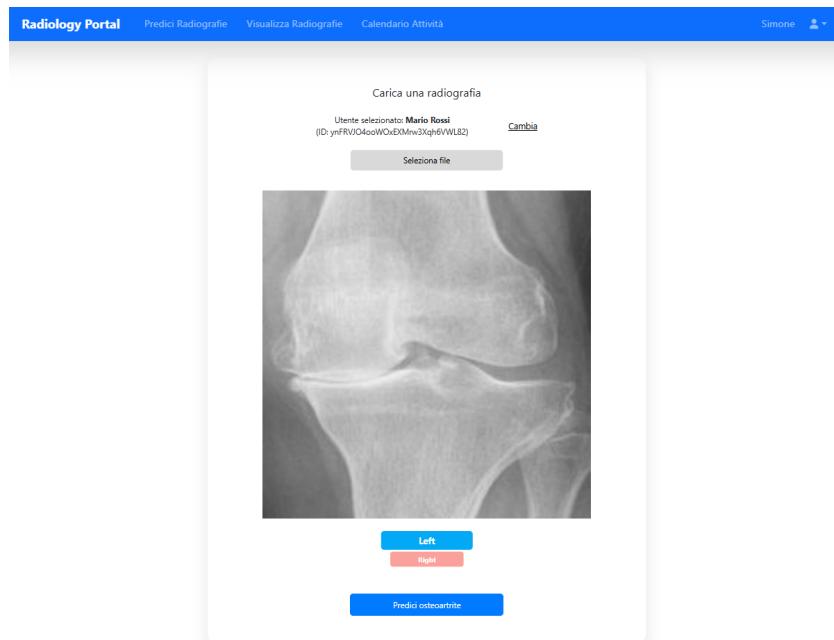


Figura 42: Caricamento di una radiografia.

Dopodichè, il sistema fornisce una predizione automatica, supportando il processo decisionale (figura 43).

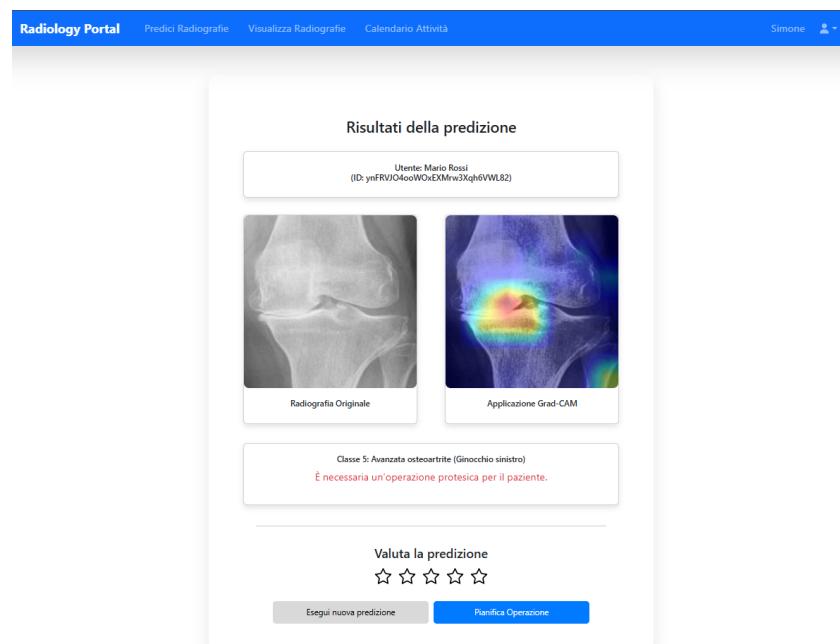


Figura 43: Predizione dell'osteoartrite.

8.5 Visualizzazione Radiografie

I medici hanno la possibilità di visualizzare le radiografie di tutti i pazienti a loro associati, con le relative predizioni (figura 44).

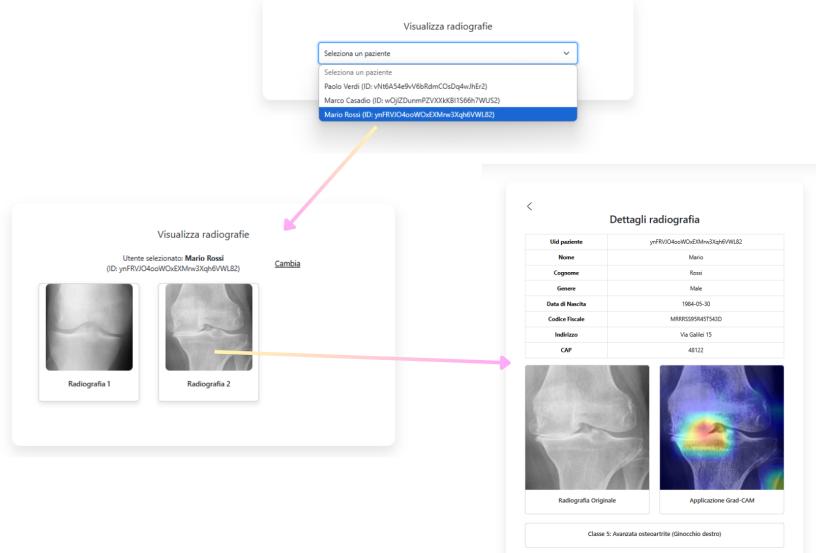


Figura 44: Visualizzazione radiografie medici.

I pazienti, invece, possono visualizzare solo le proprie informazioni (figura 45).

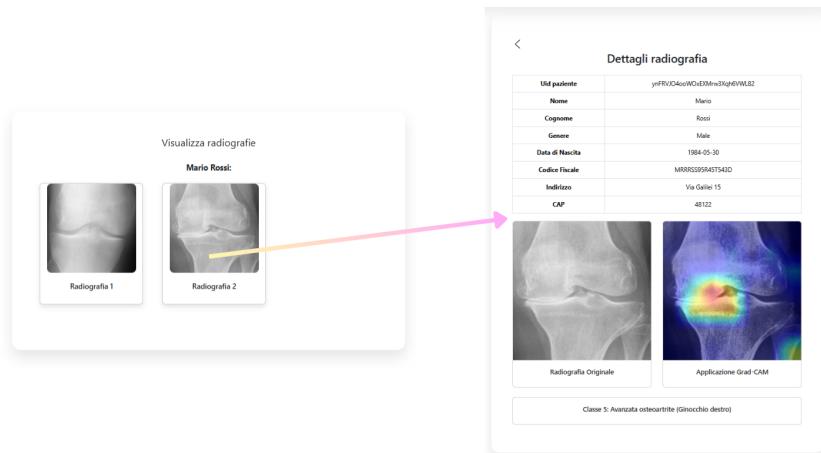


Figura 45: Visualizzazione radiografie pazienti.

8.6 Pianificazione di Attività e Notifiche

In questa sezione, medici e pazienti possono monitorare le attività, che includono le operazioni chirurgiche e il caricamento di radiografie. I pazienti possono visualizzare esclusivamente le

informazioni che li riguardano, come mostrato in figura 46.

Calendario Attività

Gennaio 2025

LUN	MAR	MER	GIO	VEN	SAB	DOM
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

Dettagli del Giorno: 3 Gennaio 2025
Non ci sono attività pianificate per questa data.

Figura 46: Vista del calendario lato pazienti.

I medici, invece, possono visualizzare le informazioni di tutti i pazienti associati (figura 47).

Calendario Attività

Gennaio 2025

LUN	MAR	MER	GIO	VEN	SAB	DOM
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

Pianifica Operazione

Figura 47: Vista del calendario lato medici.

Inoltre, possono pianificare un'operazione chirurgica inserendo tutti i dettagli relativi all'intervento. Una volta pianificata l'operazione, il sistema invia automaticamente una notifica al

paziente interessato (figura 48).

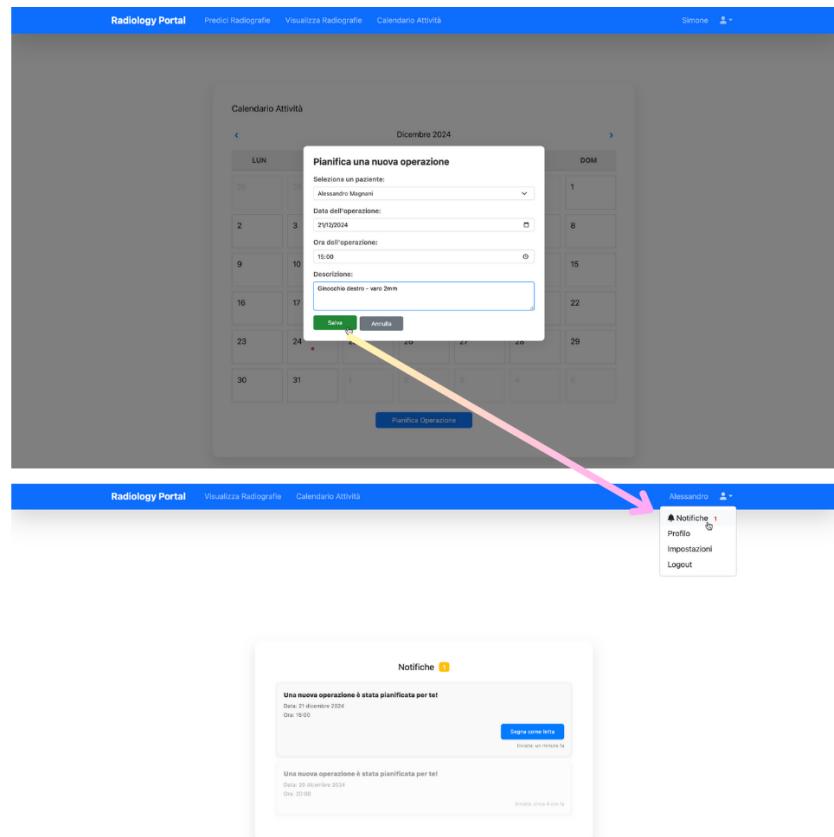


Figura 48: Processo di pianificazione di un'attività e ricezione notifica.

9 Conclusioni

Il progetto sviluppato si inserisce nel contesto della tesi che il team svolgerà in collaborazione con il primario di ortopedia dell’Ospedale Umberto I, con l’obiettivo di creare un sistema innovativo per la diagnosi dell’osteoartrite delle ginocchia attraverso l’analisi delle radiografie.

Durante lo sviluppo, i membri del team hanno acquisito e approfondito conoscenze significative nell’ambito delle applicazioni web, arricchendo la propria esperienza professionale.

La continua collaborazione tra i componenti del gruppo ha giocato un ruolo fondamentale nel raggiungimento degli obiettivi prefissati.

L’intero team ritiene che il progetto rappresenti un’importante opportunità di crescita e un’esperienza altamente formativa. Grazie a questo lavoro, il gruppo ha avuto l’opportunità di confrontarsi con aspetti complessi di progettazione e sviluppo, approfondendo tematiche relative all’architettura software, all’integrazione di sistemi complessi e alla gestione di dati sensibili in un contesto medico.