ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

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

PROGETTAZIONE E SVILUPPO DI UN SISTEMA A MICROSERVIZI INTEROPERABILE E MODULARE PER TELEVISITE IN AMBITO SANITARIO

Tesi in
PERVASIVE COMPUTING

Relatore
Prof. ALESSANDRO RICCI

Corelatore
Dott. Ing. ANGELO CROATTI

Presentata da
MARCIN TOMASZ PABICH

Anno Accademico 2019 – 2020

"I choose a lazy person to do a hard job. Because a lazy person will find an easy way to do it."
- Bill Gates

Indice

Introduzione

La pandemia che ha coinvolto il mondo durante l'anno 2020 ha fatto emergere diverse criticità in ambiti dove l'informatica non era stata vista come necessità primaria. Uno dei contesti coinvolti da questa crisi è risultato essere quello sanitario [17][12]. Le varie difficoltà non sono dovute solo all'esponenziale incremento dei ricoveri ospedalieri, ma anche all'impossibilità di assistere pazienti che solitamente si recavano di persona negli ambulatori specifici. Tale mancanza risulta essere particolarmente pericolosa per soggetti che necessitano una continua osservazione da parte del personale medico, nonchè delle categorie più deboli della società, quali ad esempio gli anziani. L'impossibilità di procedere ad una classica visita medica ha fatto emergere diverse esigenze in questo ambito, nel quale i medici hanno cercato di affrontare la situazione adottando soluzioni utilizzate nel campo della comunicazione comunemente impiegate dalle persone [15].

L'occasione di colmare tali lacune è stata colta da un'azienda, I-TEL, la quale aveva istanziato risorse per l'esplorazione e l'implementazione delle soluzioni nel campo della Telemedicina. Il progetto proposto include la collaborazione del gruppo di ricerca dell'Università di Bologna a realizzare prototipi nell'ambito di Telemedicina, coprendo molteplici aspetti esistenti intrinsecamente nel contesto sanitario. È di estremo interesse effettuare un'esplorazione approfondita in tale ambito: è un caso di studio reale e consiste in una serie di servizi interoperabili che devono garantire l'effettuazione di videochiamate tra pazienti, medici e operatori. La crescente necessità di dover offrire servizi a distanza ha fatto emergere diverse possibilità nei campi come la medicina, per le quali l'"*Industria 4.0*" soltanto recentemente è riuscita ad dimostrare i suoi punti di forza [9]. L'approccio con il quale si vuole affrontare questo percorso non è però incentrato ad una banale soluzione: infatti, è basato su una progettazione di sistemi moderni, modulari e interoperabili, capaci di poter offrire le proprie funzionalità non soltanto all'interno di uno stesso ambiente, ma inserendosi anche nei contesti più internazionali applicanti lo stesso standard del dominio analizzato.

Per questo motivo il percorso che si vuole affrontare risulta di particolare interesse: si tratta infatti di un caso di studio reale capace di porre diverse sfide durante la sua realizzazione. Inoltre, essendovi più persone lavoranti su uno stesso progetto, è lecito aspettarsi che alcuni degli aspetti vengano progettati e dichiarati come requisiti parallelamente allo sviluppo dei servizi previsti. Per questo motivo si decide di semplificare la struttura della tesi, in modo da coprire subito gli aspetti ritenuti essenziali per entrambi i progetti, anche se nella realtà dei fatti i requisiti, soprattutto per quanto riguarda la parte delle Televisite, non erano ben definiti sin dall'inizio. Di conseguenza, l'ordine cronologico di alcune tematiche realmente discusse non è sempre rispettato.

La seguente tesi avrà una composizione suddivisa in 6 macro-capitoli, analizzanti in dettaglio le varie fasi della progettazione:

1. Contestualizzazione del percorso e indicazione degli obiettivi, con introduzione a terminologie specifiche dell'ambito in analisi.

2. Analisi dei requisiti per il servizio di videochiamate e il sistema per Televisite.

3. Progettazione del servizio di videochiamate.

4. Progettazione del sistema per Televisite, comprendendo anche la dipendenza dal servizio di videochiamate ed eventuali spunti relativi alla sua architettura.

5. Implementazione prototipale del servizio di videochiamate, con cenni relativi agli sviluppi per il sistema per Televisite.

6. Conclusioni e ringraziamenti.

Capitolo 1

Il Progetto Televisita: Contesto di Riferimento

In questa sezione si affrontano gli aspetti terminologici e il background, focalizzandosi meglio sull'obiettivo di questo lavoro.

1.1 Introduzione al Progetto

Il progetto affrontato fa parte di un'esperienza di tirocinio, riguardante l'analisi delle possibilità che si hanno nella progettazione e sviluppo di servizi per effettuare (video)chiamate, utilizzando protocolli standard e gratuiti basati sul framework WebRTC. La soluzione deve provvedere ad una basilare interfaccia con la quale un utente può interagire per instaurare il collegamento ("*frontend*") e una parte di gestione della stessa ("*backend*"). Trattandosi perlopiù di un *proof-of-work*, è essenziale poter dimostrare le capacità del protocollo scelto ed eventuali criticità di esso, nonché analizzare le diverse alternative che sono a disposizione del programmatore. Essendo un tale servizio propenso ad essere utilizzato in diversi ambienti, è lecito aspettarsi un buon isolamento di una tale soluzione in modo da poterla integrare in diversi contesti. Tale servizio deve poi collegarsi strettamente all'ambito sanitario: si provvederà all'utilizzo di un servizio general-purpose, calato poi nell'ambito di Televisite. Anche questa parte però risulta essere non soltanto incentrata sulla creazione dei flussi audio/video, bensì sulla vera e propria trasformazione di una prestazione reale (una visita medica) in un servizio fruibile completamente in digitale sia dai pazienti che dai medici e/o operatori sanitari.

È inoltre importante sottolineare sin dall'inizio che gli aspetti di *modularità* e *interoperabilità* diventano i concetti chiave di tutto lo sviluppo: questi termini non stanno a indicare soltanto aspetti relativi all'interno di uno stesso

ambiente, come quello aziendale, ma si riferiscono alla volontà di rendere la soluzione capace di essere utilizzata da client diversi e progetti che adottano gli stessi standard definiti a livello italiano, europeo e mondiale. In questo modo, soluzioni diverse potranno essere compatibili tra di loro, senza preoccuparsi maggiormente di come vengano implementare al loro interno: come già definito, questa soluzione non mira ad essere un ulteriore progetto nel suddetto ambito, ma piuttosto un modo più moderno per affrontare una tematica che risulta essere molto interessante nel contesto di una pandemia. Considerando come l'Industria 4.0 abbia cambiato il modo di pensare e sviluppare soluzioni informatiche [7][9], con il sempre più imminente avvento del 5G [16], è necessario poter affrontare le moderne sfide, affrontandole in un'ottica sempre più *pervasive-oriented* [1][5].

Siccome risulta essere necessario introdurre alcuni termini utilizzati nelle fasi successive per evitare problemi di ambiguità, le sezioni introduttive chiariranno alcuni concetti ritenuti fondamentali per la realizzazione di questo progetto.

Per quanto riguarda lo standard, è necessario sottolineare che prima dell'effettiva esplorazione dei requisiti, nonchè della progettazione architetturale, si è dovuto affrontare il problema della rappresentazione del modello di dominio analizzato, senza incorrere in problemi di ambiguità e in errori di concettualizzazione. Per questo motivo, un'importante parte del background posseduto risulta essere quello relativo alla scelta del *working group* di appoggiarsi sullo standard FHIR come fonte ufficiale di concetti da rappresentare. Essendo tale aspetto di cruciale importanza, verrà analizzato in dettaglio nella sezione 1.3.

1.2 Terminologia

Prima di partire con l'analisi dei requisiti, è necessario porre una particolare attenzione alla terminologia: si vogliono infatti evitare le possibilità di errore introdotte da eventuali fraintendimenti o ambiguità. In questa sezione verranno trattati in dettaglio soltanto i termini relativi al campo della medicina che riguardano strettamente il contesto generale analizzato, facendo solo piccoli riferimenti generici a quello che risulta essere il quadro completo della sanità. Inoltre, verranno accennati alcuni termini strettamente utilizzati nella parte dei requisiti.

1.2.1 Telemedicina

La **Telemedicina** è l'insieme di tecniche mediche ed informatiche che permettono la cura di un paziente a distanza o, più in generale, di fornire servizi

sanitari a distanza. A causa della pandemia sono state rilevate diverse criticità in questo ambito [17][12], che si stanno cercando man mano di coprire: è infatti possibile notare una rapida crescita in questo determinato contesto [18], anche se non tutte le soluzioni proposte dalle aziende sono di particolare gradimento ai pazienti [14]. Sono però numerosi gli sforzi per rendere anche la "*sanità digitale*" più vicina all'Industria 4.0 [13][10][11] e all'adozione di un modello che possa essere rispecchiato non solo a livello nazionale, ma anche europeo [6].

Alcuni tentativi di portare la medicina nel campo online sono mossi direttamente dallo stato: il Fascicolo Sanitario Elettronico (FSE) [21] è un esempio di integrazione di un servizio già esistente, digitalizzando naturalmente quello che è la sua controparte reale (anche se può essere ritenuto soltanto un misero "*blocco*" componente il mondo della sanità digitale). Un ulteriore esempio di un blocco importante mosso dallo stato è l'adozione da parte di alcune regioni del servizio della ricetta telemedica [29], la quale passa online al paziente (o in alcuni casi, direttamente in farmacia).

La telemedicina è composta da diversi ambiti, dei quali esistono le controparti reali portate sul "*virtuale*". Essendo però un contesto attualmente esplorato da molti enti, è possibile prevedere che in un prossimo futuro ci possano essere componenti della telemedicina strettamente legate alla fruizione dei servizi online, senza una vera e propria controparte reale. Per contestualizzare meglio questo aspetto, si può pensare per esempio ad un consulto medico, che come sua controparte virtuale ha una televisita: nulla vieta che alcuni aspetti di esso siano caratteristici soltanto della visita online, come del resto è possibile immaginarsi un servizio di storico e raccolta dati direttamente da parte dell'utente (ed eventualmente da strumentazione di misura in utilizzo), che attualmente risulta riservata soltanto ai medici.

È necessario sin da subito sottolineare che non vi è una nomenclatura definita in via ufficiale a livello europeo e/o mondiale per quanto riguarda ambiti specifici della telemedicina, oppure non vi sono diretti corrispettivi con la terminologia italiana. Nonostante ciò, è possibile reperire diverse risorse che cercano di sopperire a tali lacune. In questa tesi si farà maggiormente riferimento al testo "*Informatica medica*" di Alberto Rosotti e soprattutto alle "*linee di indirizzo nazionali*" [4] per quanto riguarda la Telemedicina.

Televisita

Citando direttamente le "*linee di indirizzo nazionali*" [4]:

> La Televisita è un atto sanitario in cui il medico interagisce a distanza con il paziente. L'atto sanitario di diagnosi che scaturisce dalla visita può dar luogo alla prescrizione di farmaci o di cure. Durante la Televisita un operatore sanitario che si trovi vicino al paziente, può assistere il medico. Il collegamento deve consentire di vedere e interagire con il paziente e deve avvenire in tempo reale o differito.

Bisogna sottolineare che il concetto di **Televisita** non è da confondere con la definizione di **Teleconsulto**, trattato nella successiva sezione.

Teleconsulto

Citando direttamente le "*linee di indirizzo nazionali*" [4]:

> Il Teleconsulto è un'indicazione di diagnosi e/o di scelta di una terapia senza la presenza fisica del paziente. Si tratta di un'attività di consulenza a distanza fra medici che permette a un medico di chiedere il consiglio di uno o più medici, in ragione di specifica formazione e competenza, sulla base di informazioni mediche legate alla presa in carico del paziente.

Appuntamento

Si tratta di un incontro pianificato tra due (o più) soggetti in un determinato giorno ad una certa ora. Fuori dal contesto analizzato, un appuntamento può essere inteso sia come quello reale, in presenza, sia come quello online; di conseguenza, per mantenere l'universalità di questo termine, nell'ambito preso in considerazione dev'essere opportunamente identificata la modalità di svolgimento di esso.

Incontro

Nell'ambito analizzato, esso fa parte di una Televisita (o di un Teleconsulto), ed è essenzialmente il momento nel quale i soggetti partecipanti ad una (video)chiamata riescono a stabilire un contatto e a comunicare fluidamente.

Referto

È il risultato della Televisita prodotto dal medico verso un paziente, nel quale il dottore attesta di aver prestato la propria assistenza al soggetto partecipante alla Televisita, includendo eventualmente i risultati degli esami effettuati.

1.2.2 Linguaggio Tecnico

In questa sezione verranno spiegati termini e definizioni, presenti negli stessi requisiti, che potrebbero portare a eventuali ambiguità. Non si vogliono però includere in alcun modo riferimenti a particolari strutturazioni del progetto, viste soltanto nelle successive fasi. Eventuali riferimenti a librerie e/o framework sono dovuti al background già posseduto e sono necessari per capire meglio gli obiettivi da raggiungere, in quanto fanno parte imprescindibile dei requisiti richiesti trattati successivamente nel capitolo 2.

Frontend

Parte dell'applicazione visibile agli utenti finali. Possiede una logica di controllo e l'interfaccia grafica. Essendo ancora in una fase di contestualizzazione, è importante sottolineare che l'effettiva tecnologia con la quale verrà sviluppato il *frontend* ancora non è ben nota, ma si suppone che possa essere una soluzione basata sulle tecnologie Web e/o mobili.

Backend

In termini molto poveri esso può essere considerato ciò che nel gergo viene chiamato "*server*", anche se questa affermazione è estremamente inadeguata, visto che suppone già una delle possibili organizzazioni del progetto. Essendo ancora generici e non considerando alcuna specifica tecnologia, è più utile utilizzare questo termine per indicare tutta la parte che gestisce i processi che sono meno visibili all'utente, quali aspetti generici e funzionali senza cui il sistema non potrebbe funzionare. Il *backend* offrirà delle API ben definite per essere richiamato da un qualsiasi *frontend*, indipendentemente dalla tecnologia utilizzata per svilupparlo.

Videochiamata

Si intende per videochiamata una parte dell'interazione tra utenti che permette l'instaurazione di un canale di comunicazione audio (ed eventualmente anche video) tra i partecipanti a tale evento. La parte del flusso video è, nel

dominio analizzato, opzionale. Il motivo di questa scelta ricade soprattutto nel rendere non necessaria la parte video alla creazione di un collegamento e nel rendere gli utenti sempre liberi di disattivare i dispositivi di acquisizione in utilizzo. Si vuole inoltre sottolineare che la parte essenziale richiesta per la comunicazione sia quella relativa alla chiamata e quindi all'interazione verbale tra i partecipanti, ad esempio attraverso l'uso di un microfono.

È possibile, inoltre, far riferimento ad una funzionalità più avanzata, ovvero quella basata sullo scambio di messaggi diretti tra utenti (comunemente chiamata "*chat*"). Questo aspetto è volutamente omesso in questa definizione, in quanto si presuppone che una (video)chiamata sia composta soltanto dalle due parti caratterizzanti descritte prima (audio ed eventualmente video).

WebRTC

Nonostante faccia parte di una scelta specifica di una libreria, è necessario introdurre il concetto di WebRTC, in quanto risulta essere la tecnologia scelta dal dominio del progetto: infatti, sin dalle prime discussioni era emersa la volontà di utilizzare questo tipo di approccio per un futuro sviluppo di una soluzione.

WebRTC [32] è una tecnologia Open-Source che consente di stabilire una comunicazione Real-Time tra due o più endpoint in modalità P2P (Peer to Peer). Essa è supportata da aziende come Google, Mozilla o Microsoft ed è basata su HTML5 e JavaScript, ma può essere sfruttata in diversi ambiti grazie alle numerose librerie che si possono reperire online al giorno di oggi.

Il progetto originale ha una documentazione completa disponibile online [32] per sfruttare tutte le potenzialità di WebRTC, integrate direttamente nel linguaggio JavaScript. Il suo utilizzo, infatti, predilige perlopiù un ampio supporto per i browser moderni, ma la compatibilità del Framework è ben estesa grazie all'uso di ulteriori librerie specifiche per le piattaforme considerate. L'utilizzo di WebRTC nella sua maniera classica è dunque possibile senza includere particolari librerie, ma è altamente consigliato dalla community utilizzare soluzioni più mantenibili, che nascondono parte del boilerplate dovuto alla manuale creazione di tutto il background necessario al funzionamento di WebRTC.

1.3 Standard FHIR

Una sezione a parte va dedicata soprattutto allo standard FHIR (acronimo di Fast Healthcare Interoperability Resources), scelto come riferimento per l'analisi del dominio. In un ambito difficilmente già conosciuto da parte di un programmatore, è facile scontrarsi con terminologia specifica, che potrebbe indurre diversi errori anche durante la sola fase dell'analisi dei requisiti. Per questo motivo, si è scelto di approfondire l'argomento sullo standard selezionato: nel corso del progetto risulterà essere il riferimento per la rappresentazione di dati e processi nell'ambito della sanità digitale. Esso definisce in modo dettagliato quale sia il modello da seguire per garantire l'interoperabilità tra le soluzioni che adottano questo standard, non specificando però nulla riguardo l'effettiva implementazione in un linguaggio di programmazione. Non risulta essere un riferimento puramente per l'ambito "*virtuale*", ma consiste nella standardizzazione di tutte le fasi e informazioni coinvolte nel mondo reale relative al contesto medico.

FHIR [20] ritrova la sua utilità anche nell'ambito della sanità digitale, in quanto *permette di definire in modo preciso uno schema da seguire per costruire applicazioni che possono essere interoperabili tra di loro.* Vi sono infatti diversi studi riguardanti la considerazione di FHIR come riferimento principale alla progettazione di dati e processi coinvolti [8] [3] [2].

Una nota negativa riguardante questo standard è il fatto di lasciare molta libertà di interpretazione: non vi è, ad esempio, un modello definito per differenziare un appuntamento "*reale*" da quello "*virtuale*", in quanto risulta essere uno standard nato per regolare processi non solo strettamente digitali. Inoltre, nonostante una vasta copertura di diversi aspetti, è impossibile poter essere completamente esaustivi nell'elencare tutte le eventuali possibilità che potrebbero capitare durante il susseguirsi delle fasi caratterizzanti un percorso medico. Di conseguenza, è necessario saper interpretare in maniera corretta tutta la documentazione a disposizione, in modo da non rendere il proprio lavoro di *standardizzazione* del tutto inutile, vanificando completamente l'obiettivo dello standard di rendere diverse applicazioni interoperabili tra di loro.

La successiva analisi della terminologia è frutto di un attento studio svolto in concomitanza con tutto il *working group.* La terminologia scelta da affrontare, di conseguenza, non risulta essere relativa a tutto lo standard né a tutti i concetti possibilmente modellabili, ma riguarda soltanto una parte congrua a quanto affrontato durante il progetto. Questo permette di raggiungere l'obiettivo della interoperabilità, senza necessariamente considerare l'intera struttura

del sistema FHIR per quello che risulta esserne una sottoparte.

Appointment

`https://www.hl7.org/fhir/appointment.html`

Risorsa che non è completamente corrispondente alla definizione di "*Appuntamento*" della sezione 1.2.1, in quanto la sua prenotazione prevede una pre-pianificazione attraverso il concetto di "*Schedule*" (1.3). Per tutto il resto degli aspetti può essere ritenuto un valido concetto rappresentante un appuntamento. È ulteriormente da notare che un "*Appointment*" è formato da diversi "*Encounter*", trattati successivamente nella sezione 1.3. Gli unici concetti non opzionali di tale entità risultano essere lo stato[1] in cui si trova e i partecipanti ad esso. Per ulteriori dettagli, si rimanda alle successive fasi di analisi dei requisiti e dell'architettura.

Schedule

`https://www.hl7.org/fhir/schedule.html`

Si tratta di una risorsa che separa concettualmente l'appuntamento dall'effettiva disponibilità per la sua prenotazione. Le "*Schedule*" risultano essere comode per definire delle finestre temporali nelle quali medici e/o utilizzatori possono vedere le disponibilità altrui e prenotare appuntamenti in tali finestre. Il disaccoppiamento di questo concetto può rivelarsi utile per differenziare il servizio di prenotazione dall'appuntamento. L'unico riferimento non opzionale di tale entità risulta essere colui che effettivamente "*prenota*" la finestra temporale. Per ulteriori dettagli, si rimanda alle successive fasi di analisi dei requisiti e dell'architettura.

Encounter

`https://www.hl7.org/fhir/encounter.html`

Parte cruciale di un appuntamento: vi possono essere più "*Encounter*" in un singolo "*Appointment*". In esso vengono svolte tutte le azioni relative a quello che nel dominio specifico risulta essere la Televisita: eventuali misurazioni, stesura del referto e la videochiamata fanno tutte parte di un unico incontro. Tale aspetto è fondamentale e richiede alcuni parametri non opzionali come

[1] `https://www.hl7.org/fhir/valueset-appointmentstatus.html`

la classe[2] dell'incontro (nell'ambito specifico, sarà semplicemente "*virtuale*") e lo stato[3]. Inoltre, come specificato, si può comporre di diverse "*Observation*" (sezione 1.3) e un "*Diagnostic Report*" (sezione 1.3). Per ulteriori dettagli, si rimanda alle successive fasi di analisi dei requisiti e dell'architettura.

Observation

`https://www.hl7.org/fhir/observation.html`

Concetto molto generico che nel contesto analizzato permette di rappresentare una **misurazione** effettuata durante una visita: che si tratti di pressione sanguigna o temperatura, essa verrà codificata attraverso un "*Observation*". I parametri richiesti da tale entità sono essenzialmente lo stato in cui si trova[4] e il codice relativo al tipo di misurazione effettuata[5]. Per ulteriori dettagli, si rimanda alle successive fasi di analisi dei requisiti e dell'architettura.

Diagnostic Report

`https://www.hl7.org/fhir/diagnosticreport.html`

Risulta essere il concetto di referto. I parametri richiesti di questa entità sono relativi al suo stato[6] e al codice specificante il tipo di referto[7]. Per ulteriori dettagli, si rimanda alle successive fasi di analisi dei requisiti e dell'architettura.

Patient

`http://www.hl7.org/fhir/patient.html`

Il concetto di paziente espresso da FHIR. Esso non richiede obbligatoriamente particolari campi, per cui può essere utilizzato come concetto generale da cui prendere spunto. Per questo motivo è più utile rimandare direttamente alle successive fasi di analisi dei requisiti e dell'architettura per i dettagli relativi a tale entità, valutando quali attributi possono rivelarsi necessari.

[2] `https://www.hl7.org/fhir/v3/ActEncounterCode/vs.html`
[3] `https://www.hl7.org/fhir/valueset-encounter-status.html`
[4] `https://www.hl7.org/fhir/valueset-observation-status.html`
[5] `https://www.hl7.org/fhir/valueset-observation-codes.html`
[6] `https://www.hl7.org/fhir/valueset-diagnostic-report-status.html`
[7] `https://www.hl7.org/fhir/valueset-report-codes.html`

Practitioner

`https://www.hl7.org/fhir/practitioner.html`

In FHIR non esiste il concetto di "*medico*" o "*operatore*", ma piuttosto vi è un'unica categoria che raccoglie tutto il personale sanitario, differenziandolo con l'indicazione della professione attribuita a quel determinato soggetto. Di conseguenza, il medico è un "*Practitioner*" con uno specifico "*Role Code*"[8] dichiarante che egli è un medico. Oltre a questo concetto non vi sono particolari requisiti per l'indicazione di ulteriori attributi, per cui è più utile rimandare direttamente alle successive fasi di analisi dei requisiti e dell'architettura per i dettagli relativi a tale entità, valutando quali attributi possono rivelarsi necessari.

1.4 Obiettivo del Progetto Televisita

Noto il contesto, l'obiettivo di questa tesi si focalizza sulla progettazione di un'architettura a microservizi che permetta l'effettuazione delle (video)chiamate tra gli utenti. Essendo un modulo potenzialmente molto generico, l'organizzazione di tale struttura deve prevedere di essere adattabile anche in contesti diversi, ad esempio, calandosi sul mondo delle Televisite. La soluzione proposta deve considerare le varie sfide che possono esistere nell'affrontare un caso di studio reale, prendendo in esame i diversi aspetti critici che possono emergere durante le future analisi. Inoltre, gli aspetti di **modularità** e **interoperabilità** sono ritenuti concetti chiave per tutto lo sviluppo dell'applicazione, ed è necessario adattare di conseguenza l'architettura progettata in modo da soddisfare questi due primi requisiti. Una tale progettazione deve considerare la realizzazione di un prototipo capace di coprire i casi d'uso individuati e poter essere poi utilizzata in modo trasparente dal servizio di Televisite.

Successivamente, sin dall'inizio della pianificazione del servizio per le videochiamate, si vuole contrapporre il caso generico a quello calato in un contesto definito dall'instaurazione di una Televisita. Il paragone tra i due servizi in progettazione servirà per sottolineare le capacità di un sistema correttamente strutturato di adattarsi in maniera agevole ai requisiti calati su un dominio specifico. Per poter affrontare il tema della Televisita verranno quindi svolti diversi approfondimenti su tale ambito, progettando un'architettura capace di coprire casi d'uso e scenari previsti, utilizzando direttamente FHIR per la modellazione del dominio. Questi ultimi aspetti sono frutto di reali requisiti di

[8] `https://www.hl7.org/fhir/valueset-practitioner-role.html`

un'applicazione che, sulla base di una buona progettazione di un'architettura coprente i requisiti, vedrà la sua successiva ed effettiva realizzazione.

Come obiettivi secondari, si vogliono affrontare principalmente tre macrosfide:

- Sviluppare una soluzione capace di influenzare il mondo della telemedicina, calata perlopiù sull'ambito di Televisita.
- Mettere alla prova le capacità personali acquisite all'Università, al tirocinio e autodidatte relative all'uso di diverse piattaforme di sviluppo, sia per quanto riguarda la parte dell'utilizzatore finale (generalmente *frontend*), sia per quanto concerne la parte più specifica al funzionamento dell'intera infrastruttura (generalmente *backend*).
- Riuscire a coordinare il tempo a disposizione con il lavoro da svolgere in maniera indipendente, partendo dai requisiti di un problema reale e interfacciandosi contemporaneamente con un ente esterno.

Capitolo 2

Analisi dei Requisiti

Partendo dalla descrizione molto generica della sezione 1.1, è necessario definire in maniera più precisa i requisiti da raggiungere. Questi permetteranno di concentrare meglio il lavoro svolto, in modo da non cadere nella tentazione di over-engineering e nell'implementazione di feature per le quali non era richiesta né la necessità né l'urgenza. Infine, la suddivisione in categorie e sottopunti permetterà una valutazione più immediata, alla conclusione del progetto, della qualità del lavoro svolto, nonché della copertura dei requisiti stabiliti.

È necessario sottolineare che, essendovi due maggiori step da affrontare, si preferisce suddividere i relativi requisiti in due sotto-categorie, in modo da non confondere quelli generali con quelli specifici del dominio di telemedicina. Questo passaggio è utile a sottolineare l'importanza di progettare sistemi modulari che possano avere riscontro in ambiti diversi da quelli strettamente in analisi e che con pochi passaggi possano essere adattati in progetti differenti.

La suddivisione in due sotto-progetti è giustificata dal voler porre una corretta attenzione all'isolamento dei servizi e quindi della loro modularità. Inoltre, per riportare in maniera efficace i requisiti, nati da una continua evoluzione del lavoro del *working group* coinvolto, è stato deciso di riportare immediatamente tutti quelli da soddisfare. Nella realtà dei fatti, però, i requisiti per il secondo modulo sono stati definiti parallelamente allo sviluppo del primo, con un attivo coinvolgimento di tutti i partecipanti al progetto complessivo, sottolineando ancora una volta come il progetto in questione sia un problema reale e non affrontato per scopi puramente didattici.

Oltretutto, ponendo i requisiti in un unico capitolo, è possibile apprezzare la richiesta di modularità e interoperabilità tra i sotto-progetti, nonché capire meglio le scelte intraprese nei successivi paragrafi. Si rimarca però che la

stesura di questo schema non è stata il frutto di un singolo incontro, bensì di diverse discussioni attive sulle caratteristiche e sulle possibilità che i sistemi dovevano garantire. Questi incontri hanno avuto come soggetto diverse tematiche riguardanti sia aspetti puramente tecnologici, sia più architetturali. Tra le diverse problematiche analizzate si possono citare:

- la necessità di creare un sistema basato su WebRTC,
- la necessità di utilizzare FHIR,
- come avviene l'instaurazione di una chiamata,
- quali sono gli aspetti cruciali sui quali bisogna concentrarsi,
- quali sono i limiti e le libertà che si possono assumere durante lo sviluppo,
- quanta modularità si vuole garantire,
- quali API devono garantire le soluzione sviluppate.

2.1 Call Service

In primo luogo vengono analizzati gli obiettivi da raggiungere per sviluppare un servizio capace di instaurare una (video)chiamata tra utenti. Essendo utile anche al di fuori del contesto della sanità digitale, la sua progettazione parte dalla specifica di obiettivi generici e adattabili in diversi ambiti.

2.1.1 Requisiti

1. **User Requirements**

   1.1. L'utente è abilitato a:
   - 1.1.1 creare una (video)chiamata con altri utenti;
   - 1.1.2 ricevere un invito ad una (video)chiamata con altri utenti;

   1.2. Durante una chiamata l'utente può:
   - 1.2.1 mutare il proprio audio o quello degli altri partecipanti;
   - 1.2.2 abilitare/disabilitare il proprio flusso video;
   - 1.2.3 lasciare la (video)chiamata;
   - 1.2.4 essere nominato moderatore della (video)chiamata.

   1.3. Durante una chiamata il moderatore può:
   - 1.3.1 eseguire azioni abilitate per un utente semplice,
   - 1.3.2 invitare/rimuovere i partecipanti dalla chiamata,
   - 1.3.3 chiudere definitivamente la chiamata.

2. **Functional Requirements**

   2.1. Dev'essere possibile effettuare una (video)chiamata:

      2.1.1 Alla chiamata possono partecipare più di 2 persone.

      2.1.2 Si pone come *upper-bound* (opzionale) la realizzazione di (video)chiamate che possano sostenere i flussi audio/video di massimo 5 persone.

      2.1.3 L'interfaccia deve offrire un modo per poter vedere e comunicare oralmente con tutti i partecipanti all'incontro.

   2.2. Durante una (video)chiamata dev'essere possibile:

      2.2.1 regolare il volume in ingresso (ovvero quello degli altri partecipanti);

      2.2.2 regolare il volume in uscita (ovvero il proprio);

      2.2.3 abilitare/disabilitare il proprio flusso video.

   2.3. Una (video)chiamata possiede almeno un moderatore, che ha la capacità di governare sulla chiamata.

      2.3.1 Può nominare ulteriori moderatori.

      2.3.2 Può mutare e/o rimuovere i membri dalla (video)chiamata.

      2.3.3 Può chiudere definitivamente la (video)chiamata.

      2.3.4 Il creatore effettivo della chiamata diventa automaticamente il (primo) moderatore di quest'ultima.

3. **Non-functional Requirements**

   3.1. **Indipendenza** - la soluzione deve poter essere completamente isolabile e utilizzabile in qualsiasi contesto.

   3.2. **Generalità** - il modulo deve permettere di essere il più generico e il più riutilizzabile possibile, in modo da coprire in maniera esaustiva la maggior parte degli scenari che possono capitare se calato in contesti più specifici.

   3.3. **UX** - il progetto non deve porre particolare cura all'aspetto grafico, puntando più sulle funzionalità richieste che sulla loro rappresentazione.

4. **Implementation Requirements**

   4.1. **Budget** - il progetto dev'essere completato senza dispendio economico, puntando su piattaforme completamente gratuite.

      4.1.1 L'eccezione alla regola può essere considerata soltanto relativamente alle piattaforme nelle quali vi è la coesistenza di due piani ben distinti: "*gratuito*" e "*a pagamento*". Il secondo deve introdurre esclusivamente funzionalità non espressamente richieste in questi obiettivi e deve quindi essere completamente opzionale.

   4.2. **WebRTC** - la tecnologia utilizzata per instaurare le (video)chiamate deve appoggiarsi direttamente o indirettamente al protocollo utilizzato da WebRTC.

      4.2.1 Ciò non vuol dire utilizzare per forza la libreria di default, ma anche la possibilità di integrare librerie che includano al loro interno la soluzione basata su WebRTC, amplificando notevolmente il ventaglio di scelta possibile.

   4.3. **Suddivisione frontend-backend** - la soluzione deve possedere sia una parte dedicata alla grafica e all'interazione con l'utente (*frontend*) sia un'architettura che permetta la gestione della logica interna (*backend*).

      4.3.1 È preferibile spostare il carico di lavoro perlopiù sulla parte *backend*.

      4.3.2 Il *backend* deve poter affrontare aspetti architetturali non banali.

         4.3.2.1 Deve poter essere agevolmente manutenibile e modificabile.

2.1.2 Casi d'uso

Per poter apprezzare meglio i requisiti richiesti, vengono progettati alcuni schemi che riassumono ad alto livello le funzionalità dell'applicazione, rispecchiando precisamente le richieste definite. Questa sezione è da considerarsi parte integrante dell'analisi dei requisiti, nei quali si vogliono trattare in modo approfondito le specifiche per ottenere, sin dalle fasi iniziali, un progetto ben strutturato. Si vuole sottolineare che la creazione dei casi d'uso è stata svolta parallelamente a quella della formazione del modello del dominio: per un'agevole lettura le tue fasi risultano essere separate ma essenzialmente sono una dipendente dall'altra. Per dettagli relativi alla terminologia utilizzata nel seguente diagramma si rimanda alla sezione 2.1.3 trattante dei relativi dettagli.

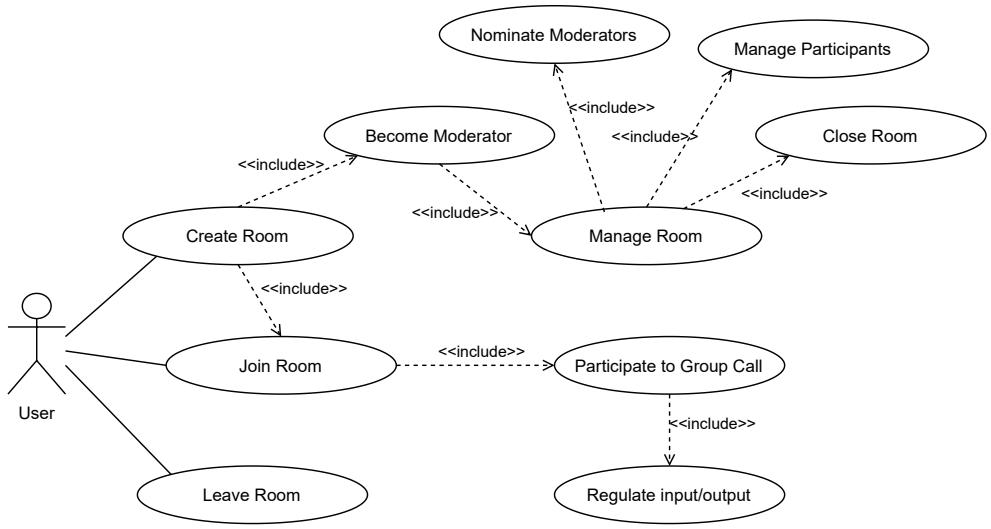


Figura 2.1: Schema caratterizzante un utente generico e le interazioni che possono avvenire durante la gestione di una chiamata. Si sottolineano le dipendenze tra le varie interazioni.

Non vi sono ulteriori particolari scenari di alto livello che potrebbero essere presi in considerazione durante questa fase. Lo schema in figura 2.1 esprime tutti i concetti essenziali di una videochiamata, senza considerare l'eventuale architettura.

2.1.3 Modello del Dominio

Prima di poter procedere con l'architettura del sistema, occorre comporre un modello di dominio che possa rispecchiare in modo univoco e non ambiguo i concetti espressi nei requisiti. Di conseguenza, si compone un modello rappresentativo di quelli che sono i principali termini utilizzati in questo contesto, per poi poter affrontare in maniera più agevole la sua effettiva progettazione.

In primo luogo è necessario chiarire il concetto di videochiamata, soprattutto per quanto riguarda l'aspetto multi-utente di quest'ultima. Nonostante sia molto chiaro come termine astratto, quello di voler partecipare in tanti ad una conferenza, si preferisce introdurre una terminologia meno ambigua proponente un modello organizzativo migliore. Essendo una chiamata di gruppo essenzialmente un raduno virtuale di utenti, si vuole esprimere il concetto di "***stanza***" per definire in maniera molto precisa il ruolo di essa e dei suoi relativi "***partecipanti***".

La stanza viene vista come un luogo virtuale dove uno o più utenti possono entrare per poter partecipare a quella che risulterà essere a tutti gli effetti una (video)chiamata di gruppo. Il concetto sottolinea che ogni collegamento eseguito sarà fine a se stesso: esso verrà creato, gestito e poi distrutto. Questo permette di isolare ulteriormente gli utenti tra di loro, in modo tale da non permettere ad una qualsiasi persona di connettersi ad un'altra senza che essa abbia creato prima una stanza. Quando quest'ultima risulta essere creata, accetta ingressi soltanto da persone esplicitamente indicate dalle **policy** della stanza stessa: se "*ad invito*", potranno entrarvi soltanto persone invitate; se "*pubblica*", tutti gli utenti aventi l'accesso al suo identificativo potranno unirsi alla conferenza. Gli **inviti** sono spediti da moderatori della stanza ad utenti che ancora non vi partecipano e valgono soltanto per quella determinata stanza.

Si definisce anche il concetto di "***partecipante***" come l'utente che attualmente è presente in una stanza e che quindi *partecipa* a quella conferenza: lo status di partecipante decade se l'utente esce da quella stanza. Ogni partecipante può diventare un moderatore, se nominato da altri moderatori oppure se risulta essere il creatore di quella stanza.

Inoltre, si precisa che il ruolo di *moderatore* viene definito grazie ai **permessi** esplicitamente definiti: il moderatore è colui che ha maggiori permessi in una stanza rispetto ad un partecipante qualsiasi. Una stanza ha dei permessi definiti di default che ognuno dei partecipanti eredita quando vi entra. I permessi possono essere modificati soltanto da moderatori, tramite un invito esplicito.

Considerando la terminologia esposta è possibile definire molto più precisamente le caratteristiche di queste entità, facilitando una futura pianificazione dell'architettura basata sul dominio. Essendo inoltre il concetto di "*videochiamata di gruppo*" estremamente vago, l'introduzione di entità che caratterizzano tale evento chiarisce notevolmente i requisiti analizzati. In figura 2.2 è possibile apprezzare una schematizzazione del modello.

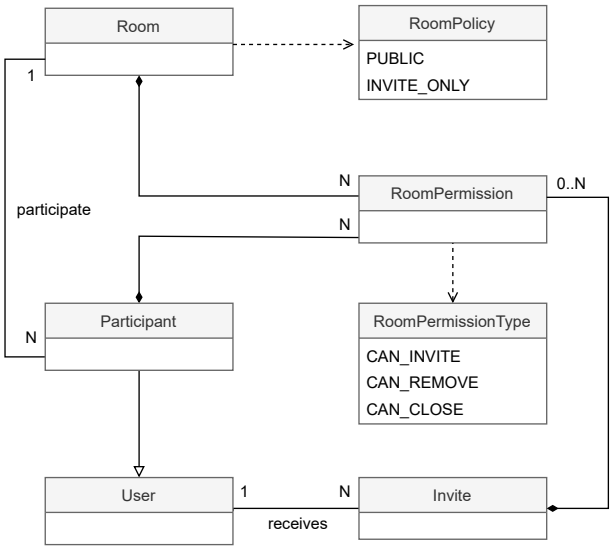


Figura 2.2: Schematizzazione del modello di dominio per quanto riguarda la gestione delle stanze per una videochiamta.

2.2 Servizio Televisita

Per contrapporre un generico servizio di chiamate e poterlo calare in un dominio più specifico, vengono analizzati ora i requisiti di un servizio per la creazione di Televisite, calato nell'ambito generico di Telemedicina. In particolare, il seguente studio deve porre l'attenzione all'integrazione del servizio per le videochiamate creato e utilizzarlo in maniera completamente trasparente. Il precedente progetto, dati i requisiti di *indipendenza* e *generalità*, deve poter essere facilmente trasportato all'interno del nuovo contesto, rispondendo ad ulteriori requisiti: alcuni semplicemente trasformati in richieste del dominio meno generiche ma più specifiche, altri richiedenti l'aggiunta di nuove funzionalità. Inoltre, grazie all'analisi effettuata prima di affrontare l'analisi dei requisiti (sezione 1.3), è possibile utilizzare la terminologia corrispondente direttamente allo standard FHIR per definire meglio quello che risulterà essere il modello del dominio analizzato.

2.2.1 Requisiti

1. **User Requirements**

   1.1. Il paziente è abilitato a:

      1.1.1 Richiedere l'assistenza al sistema e attendere l'accettazione di tale richiesta da parte di un operatore, instaurando immediatamente un collegamento audio/video (Televisita).

      1.1.2 Richiedere la pianificazione di un appuntamento.

      1.1.3 Ricevere le notifiche riguardanti i propri appuntamenti.

   1.2. Il medico è abilitato a:

      1.2.1 Richiedere un incontro con un paziente, instaurando immediatamente un collegamento audio/video (Televisita).

      1.2.2 Richiedere la pianificazione di un appuntamento con il paziente.

      1.2.3 Ricevere le notifiche riguardanti i propri appuntamenti.

      1.2.4 Rilasciare un referto alla fine della Televisita.

      1.2.5 Richiedere misurazioni al paziente durante la Televisita.

   1.3. Gli operatori sono abilitati a:

      1.3.1 Richiedere un incontro con un paziente, instaurando immediatamente un collegamento audio/video (Televisita).

      1.3.2 Ricevere le notifiche riguardanti i propri appuntamenti.

      1.3.3 Richiedere misurazioni al paziente durante la Televisita.

2. **Functional Requirements**

   2.1. La modellazione dei concetti dev'essere eseguita utilizzando FHIR.

   2.2. Dev'essere possibile effettuare una Televisita.

      2.2.1 La Televisita può essere instaurata attraverso:

         2.2.1.1 una richiesta di un paziente verso un operatore;

         2.2.1.2 una richiesta di un medico/operatore verso il paziente;

         2.2.1.3 un appuntamento pianificato in precedenza.

      2.2.2 L'interfaccia utente deve offrire un modo per poter vedere e comunicare oralmente con tutti i partecipanti.

2.2.3 Si impone un *upper-bound* (opzionale) di 5 persone in Televisita da gestire, che comprendono un paziente, uno o più medici oppure uno o più operatori.

2.2.4 Il sistema deve prevedere la possibilità di aggiunta "dinamica" dei partecipanti ad una Televisita, anche se non inizialmente pianificati.

2.2.5 Soltanto il medico e l'operatore possono effettivamente creare l'istanza di una Televisita. Il paziente può soltanto richiederlo.

2.3. Durante una Televisita dev'essere possibile:

2.3.1 regolare il volume in ingresso (ovvero quello dei partecipanti);

2.3.2 regolare il volume in uscita (ovvero il proprio);

2.3.3 abilitare/disabilitare il proprio flusso video.

2.4. Una Televisita possiede almeno un moderatore, che ha la capacità di governare su di essa.

2.4.1 Può nominare ulteriori moderatori.

2.4.2 Può mutare e/o rimuovere i membri dalla Televisita.

2.4.3 Può chiudere definitivamente la Televisita.

2.4.4 Istanziando una Televisita, il medico e/o l'operatore ne diventano automaticamente i moderatori.

2.5. Dev'essere possibile creare un appuntamento.

2.5.1 L'appuntamento può essere richiesto dal paziente o stabilito dal dottore.

2.5.2 L'appuntamento può essere fissato in un determinato giorno oppure a cadenza regolare (ad esempio, giornaliera, settimanale, mensile o annuale)

2.5.3 L'appuntamento è formato da uno o più incontri.

2.6. L'incontro è composto da una videochiamata, svolgente il ruolo di Televisita.

2.6.1 L'incontro, se pianificato con il medico, deve avere anche un referto finale.

2.6.2 L'incontro può avere una, più di una o nessuna misurazione effettuata.

2.6.3 La fine di un incontro richiede al paziente il pagamento di una somma di denaro per il servizio richiesto.

3. **Non-Functional Requirements**

   3.1. **Interoperabilità** - il sistema deve garantire una massima interoperabilità non soltanto considerando moduli della soluzione sviluppata in-house, ma anche tutti gli applicativi esistenti attualmente nel mercato che applichino gli stessi standard utilizzati dalla soluzione.

   3.2. **Modularità** - la soluzione deve rimanere suddivisa in componenti che svolgono determinati ruoli all'interno dell'intera soluzione.

   3.3. **Standard** - la modularità e interoperabilità devono essere garantite grazie all'uso di standard confermati.

   3.4. **Future-proof** - dev'essere possibile adattare il sistema per eventuali futuri scenari che possono essere previsti dagli standard utilizzati.

   3.5. **UX** - il progetto non deve porre particolare cura all'aspetto grafico, puntando più sulle funzionalità richieste che sulla loro rappresentazione.

4. **Implementation Requirements**

   4.1. **Call Service** - il progetto deve utilizzare il modulo realizzato.

2.2.2 Casi d'uso

I requisiti relativi all'ambito sanitario risultano essere più complessi e articolati. Per semplicità, anche in questo caso i casi d'uso e il dominio risultano essere trattati in due sezioni diverse, ma nella realtà dei fatti entrambi i processi sono stati svoli in parallelo. La dipendenza relativa all'uso di "*Call Service*" è evidenziata in tutti gli schemi proposti, in cui è inoltre possibile notare l'utilizzo della terminologia di FHIR per definire alcuni dei concetti.

Le figure 2.3 e 2.5 specificano in dettaglio gli scenari che un servizio di telemedicina deve coprire per quanto riguarda i due principali soggetti. Entrambi si differenziano in realtà di poco nell'utilizzo, ma è necessario sottolineare sin da subito gli essenziali cambiamenti riguardanti soprattutto i ruoli di entrambi.

Per poter contrapporre ancor meglio le differenze tra gli utenti, si esegue inoltre un collegamento diretto con il "*Call Service*" che modella già la figura

dell'utente. Nel caso del servizio previsto per la telemedicina si specifica che esistono utenti più specializzati, aventi diversi ruoli e permessi nella gestione del processo di creazione e partecipazione alla chiamata. Nella figura 2.6 è possibile apprezzare infatti che i pazienti e medici non siano nient'altro che utenti specializzati.

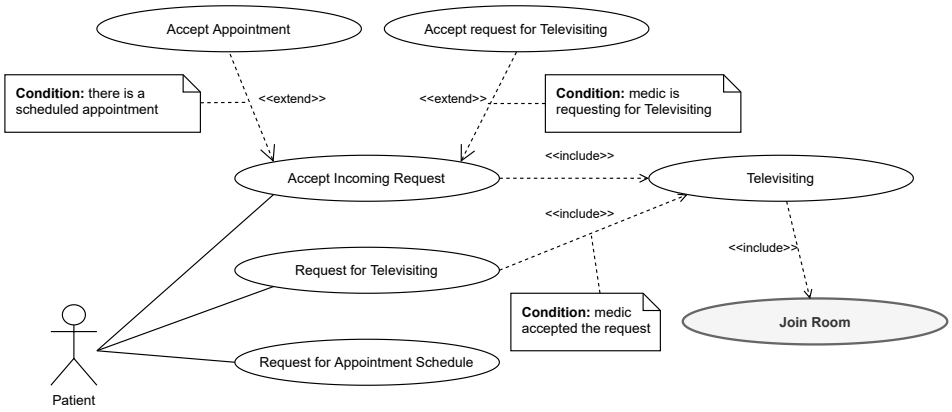


Figura 2.3: Schema caratterizzante **il paziente** e le interazioni che possono avvenire durante la gestione del suo ciclo di vita. Si sottolineano le dipendenze tra le varie interazioni, nonchè una maggiore complessità rispetto al semplice modulo delle chiamate. Il punto di unione tra il "*Call Service*" e il paziente risulta essere identificato nel nodo "*Join Room*".

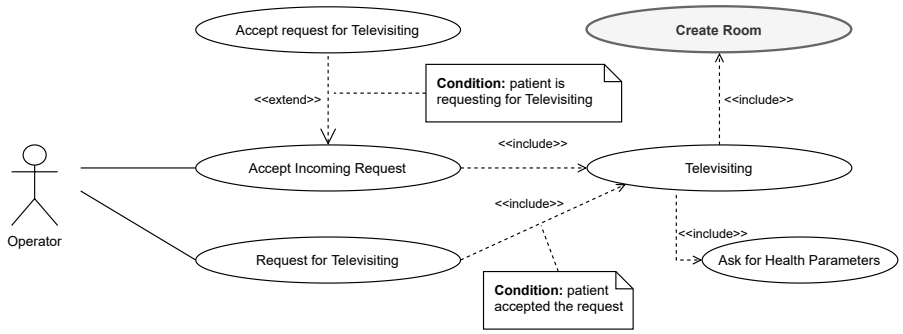


Figura 2.4: Schema caratterizzante **l'operatore** e le interazioni che possono avvenire durante la gestione del suo ciclo di vita. Si differenzia dal medico per la minor possibilità di gestione. Il punto di unione tra il "*Call Service*" e l'operatore risulta essere identificato nel nodo "*Create Room*".

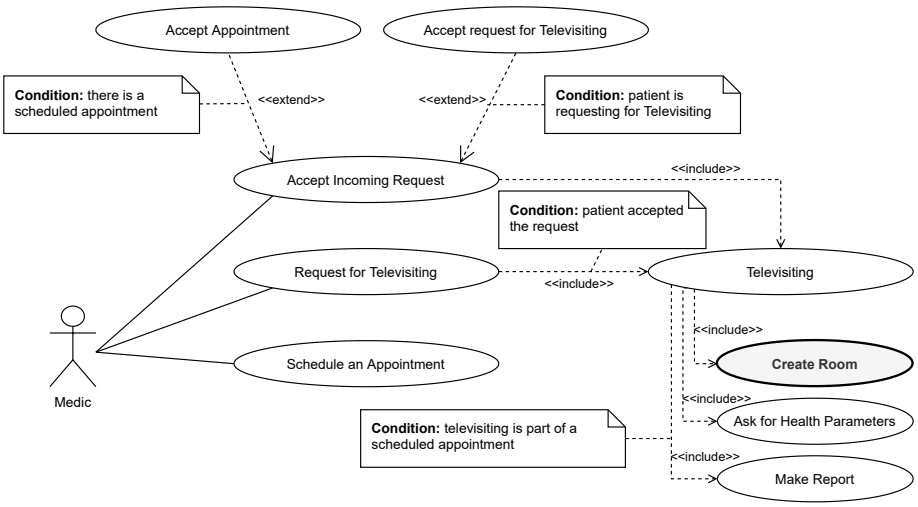


Figura 2.5: Schema caratterizzante **il medico** e le interazioni che possono avvenire durante la gestione del suo ciclo di vita. Si sottolineano le dipendenze tra le varie interazioni, nonchè una maggiore complessità rispetto al semplice modulo delle chiamate. Risulta essere del tutto simile a quello del paziente, ma presenta un punto di vista diverso dell'applicazione che è necessario sottolineare sin da subito. Il punto di unione tra il "*Call Service*" e il medico risulta essere identificato nel nodo "*Create Room*".

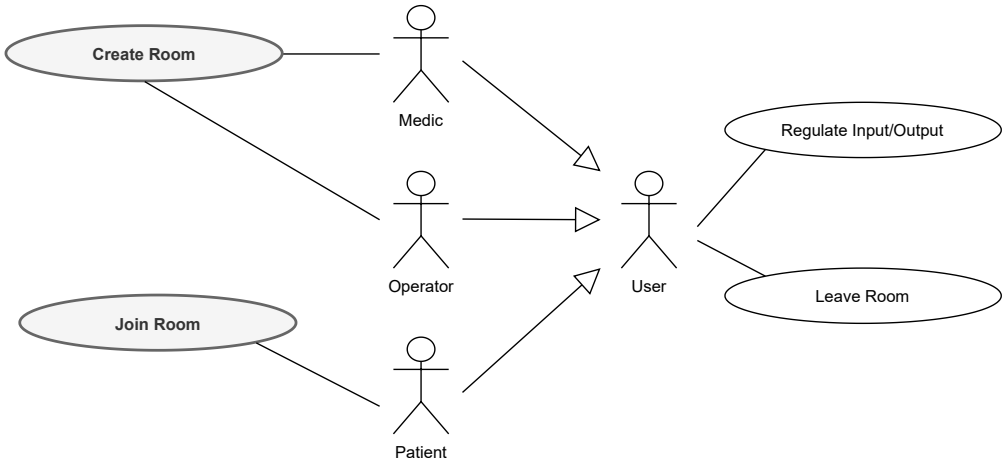


Figura 2.6: Schema generale rappresentante una visione globale sul sistema. Ogni attore è un'istanza di utente, che ha determinati permessi in una chiamata. Le parti relative all'effettuazione di quest'ultima a o all'unione ad essa sono ereditate dallo schema 2.1 e non sono state riprodotte per semplicità.

2.2.3 Scenari

In un contesto più complesso, oltre ai casi d'uso, si vogliono esplicitare degli scenari che dovranno essere coperti dalla soluzione proposta. Gli schemi proposti sono frutto del lavoro di tutto il *working group* per riuscire ad esplicitare in maniera più chiara i diversi requisiti. Per l'analisi degli scenari viene introdotta terminologia più specifica relativa allo standard FHIR.

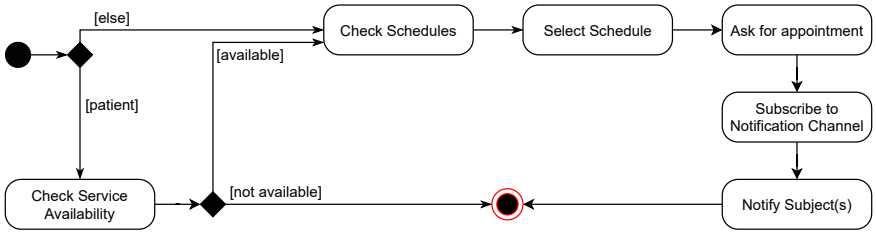


Figura 2.7: Processo per la richiesta di un appuntamento. È da notare che la richiesta coincide con la creazione di un appuntamento, che cambierà di stato in base alla sua effettiva accettazione o rifiuto. Il paziente dovrà attendere la disponibilità di un tale servizio, mentre il dottore e l'operatore potranno creare appuntamenti senza averne necessità.

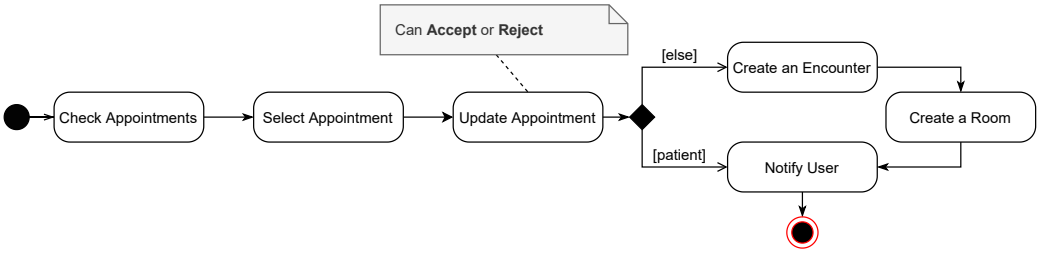


Figura 2.8: Processo per il cambiamento di stato di un appuntamento. Il paziente verrà notificato se quest'ultimo è stato accettato o meno, mentre il dottore/operatore avranno anche la facoltà di creare l'incontro e la stanza virtuale dove si svolgera un tale appuntamento.

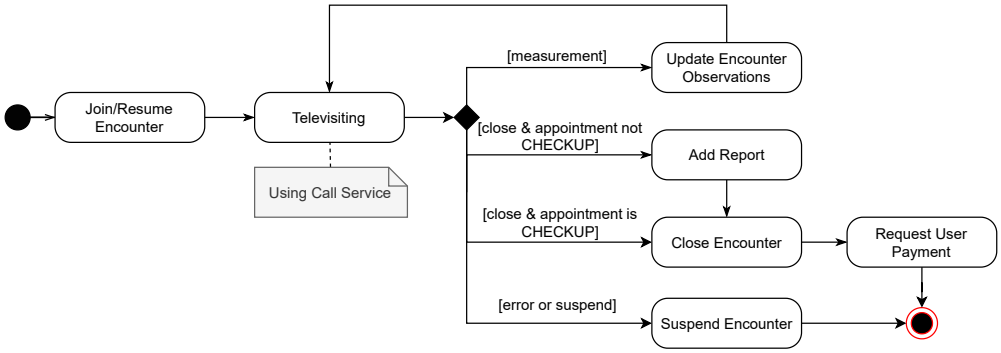


Figura 2.9: Processo che stabilisce un esempio di esecuzione di una Televisita.

2.2.4 Modello del dominio

Prima di poter procedere con l'architettura del sistema, occorre comporre un modello di dominio che possa rispecchiare in modo univoco e non ambiguo i concetti espressi nei requisiti. Di conseguenza, si compone un modello rappresentativo di quelli che sono i principali termini utilizzati in questo contesto, per poi poter affrontare in maniera più agevole la sua effettiva progettazione. Grazie all'analisi effettuata prima di affrontare l'analisi dei requisiti, è possibile sfruttare direttamente il modello proposto dallo standard FHIR per contribuire alla creazione di uno schema con semantica non ambigua coprente in maniera esaustiva i concetti che si vogliono rappresentare.

In contrapposizione con un servizio generico per le videochiamate, in ambito delle Televisite non esiste soltanto il concetto di *utente*, ma anche quello di **medico**, **operatore**, e **paziente**.

Il **medico** risulta essere la figura che effettua e gestisce una Televisita. Esso può richiederne una direttamente al paziente oppure accettarne una richiesta in precedenza (sia pianificata che in quel determinato istante). Tali interazioni inviano **notifiche** ai partecipanti a tali eventi, che si vedono informati del progressivo cambiamento di stato. Il medico è la figura che crea le stanze nelle quali si svolgono le chiamate; di conseguenza, anche la moderazione di queste ultime risulta essere a carico del soggetto analizzato. Fa parte della categoria "*Practictioner*" (sezione 1.3), per quello che riguarda lo standard utilizzato.

L'**operatore** è un soggetto specializzato nella risposta alle Televisite richieste dal paziente. Esso è abilitato a creare le stanze in cui si svolgono le videochiamate nel caso in cui il medico non fosse disponibile e/o fosse assente. Inoltre, può richiedere una chiamata diretta verso il paziente. Insieme al medico, risulta essere rappresentato dall'entità "*Practictioner*" (sezione 1.3).

Il **paziente** è la figura che richiede o partecipa ad una Televisita. Durante quest'ultima, esso è visitato virtualmente da un medico, il quale può chiedergli di svolgere determinate azioni (quali, ad esempio, misurazione dei parametri vitali come la pressione). Il paziente è notificato degli eventi che capitano all'interno del sistema, soprattutto per quel che riguarda le Televisite richieste. Esso è direttamente proposto come "*Patient*" (sezione 1.3) nello standard FHIR.

Oltre ai soggetti è necessario modellare le interazioni non strettamente relative alla creazione delle stanze nelle quali avviene l'effettuazione della Televisita. È infatti di notevole importanza introdurre concetti di **appuntamento**, **incontro** e **prenotazione** che caratterizzano la pianificazione e l'esecuzione di prestazioni mediche. Queste, inoltre, includono al loro interno le operazioni di **misurazione** e stesura di un eventuale **referto**.

L'**appuntamento** ("*Appointment*", nella sezione 1.3) è un concetto per il quale esiste un determinato giorno e una determinata ora alla quale verrà svolta una Televisita tra due o più soggetti (si preferisce stabilire che anche richieste "*al volo*" debbano avere tali dati). Per prenotare l'appuntamento vi è la necessità di trovare uno slot libero e disponibile nella data/ora specifica ("*Schedule*", nella sezione 1.3). L'appuntamento può essere annullato (di conseguenza, anche rimandato) o accettato (avrà uno stato che seguirà precisamente la sua evoluzione) e include almeno un medico e/o operatore e un paziente partecipanti a tale evento. Un appuntamento accolto produrrà l'effettivo **incontro** ("*Encounter*", nella sezione 1.3) tra i partecipanti alla Televisita. Tale meeting è caratterizzato dalla partecipazione dei membri ad una videoconferenza, nella quale possono essere richieste al paziente alcune **misurazioni** ("*Observation*", nella sezione 1.3) dei suoi parametri vitali e alla fine del quale verrà steso un **referto** ("*Diagnostic Report*", nella sezione 1.3). Quest'ultimo potrà essere creato soltanto se l'appuntamento è stato pianificato in precedenza tra i membri, e non nel caso in cui vi sa una diretta richiesta di chiamata, differenziando così due tipologie di *appuntamenti* possibili. Le *misurazioni*, al contrario, potranno essere sempre richieste. Queste ultime sono caratterizzate dai dettagli relativi alla misurazione, quali ad esempio la tipologia e il valore. Vi possono essere più misurazioni in un incontro e un appuntamento può essere caratterizzato da più incontri.

La figura 2.10 mostra uno schema del dominio. È necessario specificare che si tratta soltanto di una semplificazione, in quanto i vari aspetti gestionali di un appuntamento non vengono considerati in tale schema. Questo è dato dalla volontà di agevolare la lettura della figura per comprendere il generale quadro

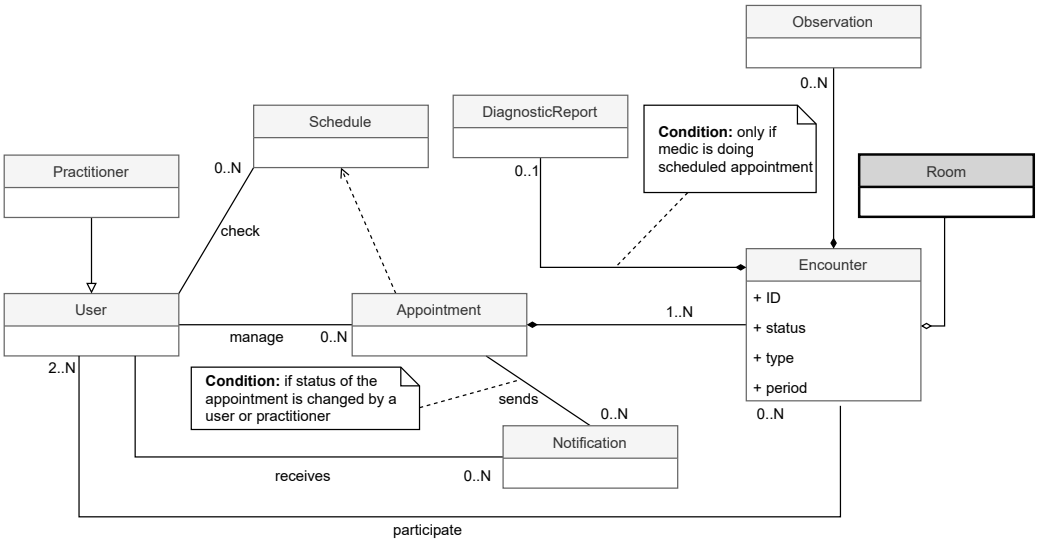


Figura 2.10: Schematizzazione semplificata del modello di dominio per quanto riguarda la gestione di una Televisita. Le specifiche delle relazioni tra appuntamento e un soggetto della Televisita sono visibili in figure 2.11 e 2.12.

del modello analizzato. Come però definito dai casi d'uso e dalle relative analisi del modello, nella realtà dei fatti ogni utente svolge un ben determinato ruolo per quanto riguarda gestione e creazione degli appuntamenti. Tali concetti sono rappresentati nei diagrammi 2.11 (per quanto riguarda la contrapposizione medico-operatore) e 2.12 (per il paziente). Inoltre la relazione "*manages*" non si riferisce alla possibilità di creare l'appuntamento (già esplicitato come concetto), ma di stabilire ulteriori dettagli, discussi durante l'introduzione di tali soggetti come entità di dominio (quali ad esempio l'accettazione o il rifiuto di un Appuntamento).

Una nota importante da sottolineare si ha per quanto riguarda l'apparente ridondanza di concetti: infatti, l'appuntamento e l'incontro possono sembrare inizialmente una stessa risorsa. Per quanto riguarda una Televisita potrebbe effettivamente trattarsi di un'unica chiamata svolta in un unico appuntamento, rendendo tale esplicitazione superflua. L'obiettivo di questo progetto però non è soltanto la modellazione di una struttura capace di offrire le funzionalità di una Televisita, ma di essere il più possibilmente modulare. Per questo motivo si prevede che la Televisita possa essere soltanto una delle possibilità offerte da questo sistema e che in futuro vi potranno essere più incontri relativi ad un appuntamento, come succede nella realtà dei fatti: ad esempio, un appuntamento "*oncologico*" potrebbe essere composto da un'incontro relativo ad un'ecografica da svolgere e da un prelievo del sangue, risultando a tutti gli effetti formato da due incontri distinti.

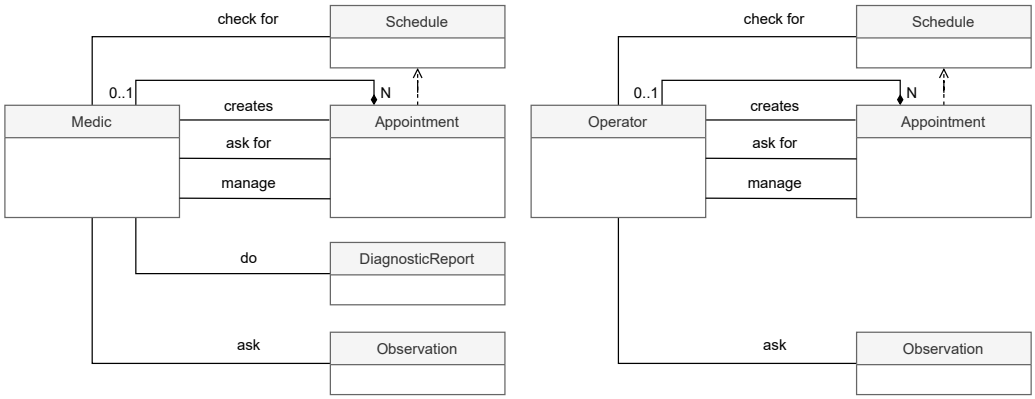


Figura 2.11: Schematizzazione semplificata dei concetti validi per il dottore e per l'operatore, due casi particolari di "*Practitioner*".

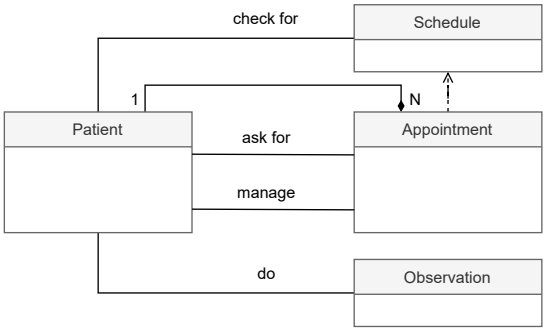


Figura 2.12: Schematizzazione semplificata dei concetti validi per il paziente.

Capitolo 3

Progettazione Call Service

In questo capitolo, prima di procedere all'effettiva progettazione di aspetti più architetturali, è necessario analizzare in quale maniera i requisiti posti possano influenzare una futura realizzazione del Call Service. Questo si rende necessario a fronte del requisito di modularità, provvedendo ad una strutturazione dei vari componenti del servizio considerando sin da subito tutti gli aspetti che potrebbero caratterizzarlo in maniera non banale. Successivamente allo svolgimento di approfondimenti sui requisiti implementativi, si svolgerà l'effettiva progettazione del servizio, della sua architettura e API offerte, analizzando in dettaglio le sue possibilità e criticità.

3.1 Vincoli implementativi

Oltre alla definizione del modello del dominio, durante la stesura dei requisiti sono stati posti alcuni vincoli per la futura implementazione del progetto, in quanto è necessario:

- utilizzare WebRTC per instaurare (video)chiamate tra utenti;
- suddividere il progetto in una parte *frontend* e *backend*;
- studiare un'architettura capace di offrire modularità dei componenti.

3.1.1 WebRTC

Come definito nella sezione 1.2.2, WebRTC offre la possibilità di instaurare connessioni P2P tra i partecipanti di una (video)chiamata. Data questa specifica è facile intendere che non vi sarà a disposizione un'entità centrale regolante i flussi, bensì i singoli utenti dovranno predisporre dei metodi che permetteranno un'agevole creazione della comunicazione. Influenzando non banalmente un'eventuale architettura in progetto, questo requisito pone un'ulteriore necessità

di approfondire le modalità con le quali questa libreria consente di connettere i partecipanti.

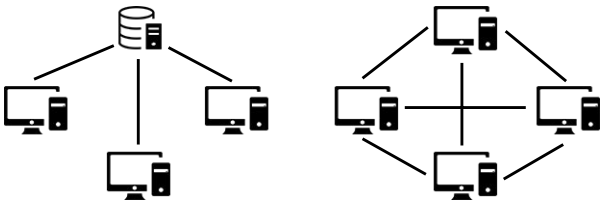


Figura 3.1: Differenze tra un'eventuale architettura con un'entità centrale e una soluzione P2P. Nella figura viene considerato soltanto un canale di comunicazione: in realtà vi è necessità di un collegamento *downstream* e *upstream* per instaurare una comunicazione bidirezionale, con il numero dei collegamenti crescenti all'aumentare dei partecipanti.

WebRTC risulta essere una libreria completa di tutti i dettagli, ma l'adozione di un approccio così scrupoloso potrebbe portare ad una divergenza dall'obiettivo di avere un prototipo funzionante, concentrandosi soltanto sull'esplorazione di questa tecnologia e non su tutti gli altri aspetti che riguardando il resto del progetto. Dato che i requisiti non impongono l'uso della libreria nativa ma permettono l'utilizzo di pacchetti più semplici da capire e usare, si ritiene opportuno poter individuare alcune delle possibilità offerte in questo contesto.

Anche se la scelta di librerie non dovrebbe essere fatta a livello di progettazione architetturale, essendo WebRTC un vincolo posto in principio si ritiene necessario dover individuare ulteriori possibilità basate su questo standard. Questa scelta è inevitabile per poter progettare meglio l'architettura del rimanente sistema, in modo da inglobare perfettamente l'interfaccia offerta da una libreria scelta, senza andare a compromettere l'integrità del servizio stesso. Avendo individuato anche le diverse difficoltà con le quali è possibile scontrarsi utilizzando una soluzione basata su un'implementazione puramente WebRTC, si considera questa valutazione necessaria affinché i diversi moduli possano efficacemente collaborare tra di loro. In particolare, durante le discussioni relative all'implementazione di WebRTC, sono stati citati due approcci differenti: **PeerJS** e **OpenVidu**.

PeerJS

PeerJS [26] è una libreria costruita sopra WebRTC che permette di semplificare notevolmente l'instaurazione di una (video)chiamata tra due parteci-

panti: come risulta essere indicato anche dalla homepage del progetto[1], sono necessarie all'incrica 40 righe di codice JavaScript per permettere una tale connessione. Il suo funzionamento è basato su un server centrale, al quale però è assegnato soltanto il compito di *ID Dispensing* e coordinazione, e non della comunicazione vera e propria.

Infatti, per poter garantire una comunicazione tra due partecipanti, occorre che entrambi siano in possesso di un *PeerID*, che risulterà essere il loro identificativo nel sistema. È compito del programmatore prevedere un modo sia per mantenerli che per condividerli tra utenti nella piattaforma progettata: nasce sin da subito la necessità di un componente che possa gestire tale aspetto nel progetto considerato. Ottenuto l'identificativo, uno degli utenti inizializza la connessione chiedendo esplicitamente di connettersi ad un *PeerID* da lui indicato. Se il soggetto richiesto è attualmente in ascolto (ed è quindi connesso al sistema), il server centrale provvederà a consegnare tutte le informazioni necessarie all'instauramento della connessione tra i due partecipanti e la (video)chiamata potrà avere inizio. Quest'ultima verrà eseguita esclusivamente su un canale appositamente creato soltanto tra i due utenti, rispettando la natura P2P del protocollo WebRTC sottostante. Tramite l'uso di PeerJS è possibile stabilire anche connessioni dati, per eventuale invio di flussi diversi da audio e video.

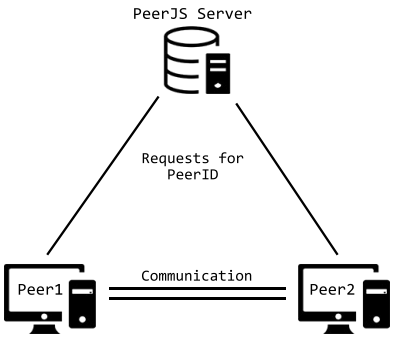


Figura 3.2: PeerJS Server svolge il ruolo di coordinatore, ma non di media-center. Il flusso di comunicazione passa soltanto da un canale bidirezionale appositamente creato tra tutti i partecipanti alla (video)chiamata.

Lo svantaggio principale di tale soluzione è l'esplicita richiesta di un servizio che dev'essere contattato dai singoli utenti per ottenere l'identificativo, attraverso il quale sarà possibile instaurare successivamente una (video)chiamata. È anche richiesto un modo con cui gli utenti possano scambiarsi di identificativi, e il mantenimento di quest'ultimi. Inoltre, al crescere dei partecipanti, il

[1] `https://peerjs.com/`

numero delle connessioni sale esponenzialmente, per cui è possibile che una tale scelta non sia indicata per videoconferenze con un numero elevato di utenti, penalizzando soprattutto partecipanti non aventi dispositivi moderni (con una connessione internet performante) capaci di operare con un'elevata quantità di flussi in entrata/uscita.

OpenVidu

OpenVidu [25] è una soluzione che pone un approccio completamente differente da PeerJS: anziché essere una semplice libreria per collegamenti P2P offre una serie di plugin, componenti e librerie integrabili in diverse soluzioni. Essi permettono l'instaurazione delle (video)chiamate organizzate in *sessioni*, nelle quali possono comunicare diversi utenti. La differenza architetturale sta nel fatto di utilizzare stavolta un'entità centrale, alla quale i partecipanti devono interfacciarsi. Questa fa essenzialmente da media-center, nel quale è possibile svolgere anche del monitoraggio sulla chiamata stessa.

La procedura per instaurare una chiamata è simile a PeerJS: una sessione può essere creata da un utente e nel momento in cui comincia ad esistere, ogni partecipante, prima di entrare, deve richiedere alla sessione stessa un token con il quale verrà identificato. Ottenuto il token, l'instaurazione della chiamata procede in modo differente, in quanto un utente possiederà soltanto un flusso di uscita e un flusso di entrata, connesso direttamente al server che gestisce la sessione. Di conseguenza, anche la visualizzazione dei diversi partecipanti non è più delegata all'implementazione manuale, ma utilizza un formato definito dal server: ciò permette di avere soltanto un singolo punto d'ingresso per i tutti i flussi dei partecipanti alla conferenza.

Le API sono disponibili in soluzioni NodeJS e Java, nonchè eventualmente ottenibili grazie al protocollo REST, permettendo un facile inglobamento della libreria all'interno di quello che dovrà essere il *backend* della soluzione. A differenza di PeerJS, OpenVidu non richiede ai singoli utenti di gestire gli aspetti legati della chiamata, ma sposta la logica di essa sull'architettura interna. Inoltre, mette a disposizione diversi componenti per l'integrazione di questo servizio direttamente nel *frontend*, riducendo anche il carico di lavoro del programmatore per quanto riguarda la progettazione e gestione dell'interfaccia grafica.

Il principale svantaggio di questa soluzione sono i limiti posti dalla versione gratuita del piano. Anche se le funzionalità incluse nel piano *free* possono sembrare sufficienti, non è del tutto improbabile che una futura release del

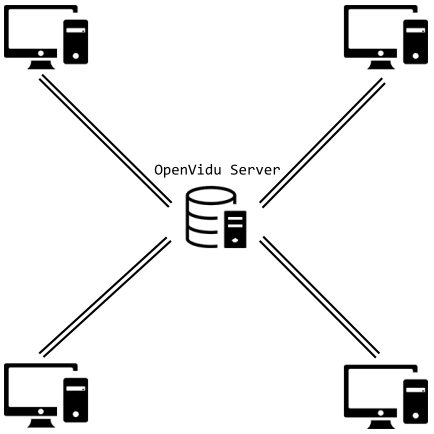


Figura 3.3: OpenVidu ha un server centrale che gestisce le sessioni; grazie a queste gli utenti hanno solo due flussi (uno in entrata e uno in uscita), spostando la maggior parte del carico di lavoro sull'ente esterno.

protocollo possa cambiare strategia e l'intero modulo diventi completamente a pagamento. Essendo un requisito essenziale quello di utilizzare librerie gratuite, una soluzione non pienamente conforme a tale richiesta potrebbe rivelarsi a lungo andare un limite, che costringerà lo sviluppatore a doverla sostenere economicamente. Incentrare inoltre tutta la difficoltà computazionale in un singolo punto potrebbe potenzialmente portare a dover scalare il sistema nei momenti di maggior utilizzo, e non essendovi alcun riferimento sulla scalabilità orizzontale della libreria analizzata, sarebbe opportuno effettuare alcuni test di performance prima di decidere di puntare su questo approccio. Inoltre, a differenza di WebRTC, sul quale è basato OpenVidu, esso non è puramente P2P, in quanto introduce un'entità centrale nel sistema. Dipendentemente dal punto di vista (e dai requisiti dell'applicazione in analisi) questo può ovviamente essere sia un vantaggio (minore gestione del codice lato client) che uno svantaggio (minore privacy e introduzione di un "*Single Point of Failure*").

3.1.2 Frontend e Backend

La maggior parte delle soluzioni che prevedono l'interazione di uno o più utenti con degli applicativi esposti da un'azienda è strutturata in modo tale da esporre soltanto la minima parte essenziale per la fruizione del servizio. Una tale suddivisione permette di separare concettualmente ciò che possono vedere gli utilizzatori da quella che risulta essere la logica di funzionamento. Questa pratica è comunemente riconosciuta nell'ambito della progettazione di soluzioni doventi utilizzare un qualche protocollo di comunicazione, per cui

non vi sono particolari criticità ad essa legata. L'analisi relativa a tali concetti è stata già effettuata nella sezione 1.2.2.

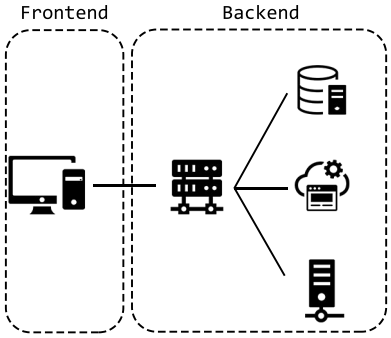


Figura 3.4: Schema generale di un'architettura a *frontend* + *backend*. Essa risulta essere una semplificazione della realtà, in quanto dipendentemente dall'architettura scelta vi possono essere più punti d'ingresso e, ovviamente, più client eterogenei connessi.

3.1.3 Architettura Modulare

Considerando i requisiti e le caratteristiche analizzate nelle sezioni precedenti, è necessario individuare una possibile progettazione di un'architettura capace di offrire tutte le *feature* richieste, scendendo a compromessi soltanto quando strettamente necessario. L'architettura richiesta deve infatti poter garantire:

- **interoperabilità** tra componenti sia interni che esterni alla soluzione,
- **modularità**, in quanto composta da diverse parti che espongono interfacce,
- **efficienza** sia in termini di costi che di pura potenza computazionale a disposizione.

In ambito di sviluppo soluzioni che comunicano via Web esistono diversi approcci con i quali è possibile affrontare questi problemi. Il più classico riguarda la strutturazione interna di diverse classi e moduli all'interno di un unica grande soluzione *backend*, il quale attraverso delle API ben note espone i suoi servizi. Questa può sembrare una valida considerazione, pensando ad una comune pratica del *Single Responsibility Principle* che riguarderà i vari sorgenti di cui è composta il progetto. L'architettura così strutturata non si

differenzierebbe moltissimo dallo schema presentato in figura 3.4.

Il problema più grande riguardante questo classico approccio di sviluppo è una dubbia facoltà di garantire una modularità tale da renderlo una scelta ufficiale: questo è causato dal fatto di possedere un'unica grande soluzione, offrente un unico punto di ingresso. Anche se la progettazione strutturerà in maniera corretta le classi corrispondenti ai singoli compiti da svolgere, la reale interfaccia esposta sarà sempre e comunque una sola. Essa potrà ovviamente suddividere il proprio lavoro in task, beneficiando soprattutto di un sistema con diversi *core* computazionali utilizzabili, ma una volta finite le risorse a disposizione comincerà il degrado significativo di performance. Questo è inaccettabile, soprattutto in un sistema che tratta con la salute dei suoi utilizzatori.

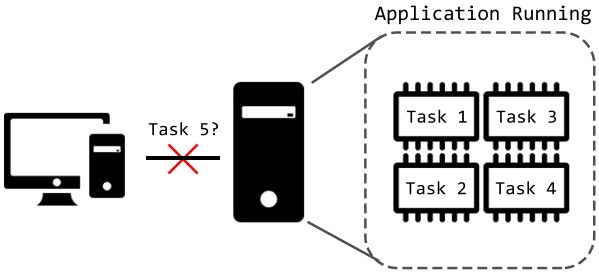


Figura 3.5: Schematizzazione semplificata delle problematiche. Il server è completamente impegnato a svolgere dei task e non può occuparsi di svolgerne altri. Inoltre, un'interruzione del collegamento o il guasto all'interno del server rende completamente inutilizzabile l'applicazione.

Si rende quindi necessario analizzare possibili soluzioni ai problemi evidenziati, mettendo soprattutto in primo luogo la necessità di poter garantire una maggiore efficienza a fronte di molteplici richieste. La prima idea che potrebbe essere applicata è quella di incrementare il numero di macchine sulle quali è in esecuzione la soluzione finale, ma ciò significherebbe includere un modulo aggiuntivo per bilanciare il carico in ingresso. Inoltre, si introdurrebbe un collo di bottiglia per quanto riguarda l'accesso all'eventuale database (se non si decidesse di diversificare le istanze anche di quello, con ancor maggiori problematiche del caso). Non essendo tale soluzione applicabile in campo reale, si analizza un'ulteriore metodo per poter organizzare le risorse a disposizione.

Piuttosto che pensare all'architettura legata all'hardware su cui viene eseguita, si potrebbe più generalmente pensare a diversi moduli "*virtuali*" eseguiti su una o più macchine, abbracciando completamente il principio di *modularità*. Non essendovi più il vincolo di avere un'unica soluzione, è possibile suddividere

un modulo in diversi componenti più piccoli, ognuno dei quali offre una propria interfaccia con la quale può essere utilizzato. Ognuna di queste semplici parti lavora indipendentemente una dall'altra, consentendo l'eventuale bilanciamento di carico tra di esse, creando più istanze di uno stesso servizio.

È lecito pensare che il problema sia stato già affrontato in letteratura: infatti, un'organizzazione del genere è chiamata "***a Microservizi***". Si tratta di un approccio in cui ogni modulo risulta essere in fin dei conti un (ridotto) servizio. Si chiamano "*micro*" in quanto le funzionalità da loro offerte si concentrano perlopiù su poche semplici cose, in modo da seguire il *Single Responsibility Principle*, coperto nella soluzione precedente con la sola suddivisione tra le classi. Ovviamente, il suddetto non è l'unico motivo per cui si decide di organizzare un'applicazione in questo modo: vi sono ragioni più pratiche per quanto riguarda sia la manutenibilità ed estensibilità della soluzione, sia la sua scalabilità e *fault tolerance*. Ognuno dei (micro)servizi è quindi responsabile di una parte piccola ma significativa dell'applicazione, e sono comunicanti tra di loro tramite API ben definite (che, in alcuni casi, vengono completamente esposte; in altri casi, agglomerate in un gateway per poi essere distribuite internamente).

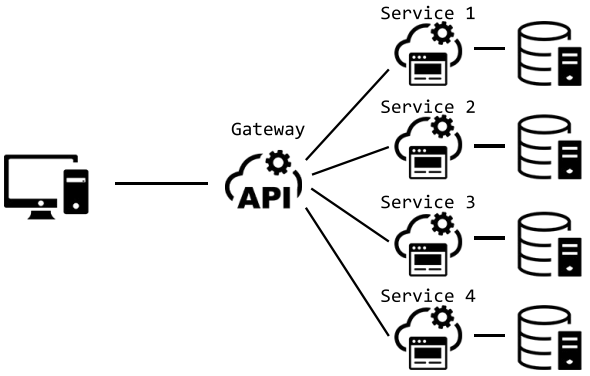


Figura 3.6: Schematizzazione semplificata dell'architettura a microservizi. Vi è un punto di ingresso (generalmente chiamato "*Gateway*"), il quale nasconde l'architettura a microservizi. Le API richiamate finiscono poi nei rispettivi moduli che gestiscono quella determinata richiesta. Inoltre, gli eventuali database sono univoci per ognuno dei microservizi, permettendo così una facile scalabilità.

I microservizi hanno il vantaggio di essere indipendenti: l'API garantisce l'interoperabilità, ma la loro implementazione può essere eterogenea. È possibile infatti incrociare microservizi di linguaggi e architetture diverse senza

avere nessuna preoccupazione sulla compatibilità, purché si definisca un'interfaccia comune per poter comunicare. Inoltre, grazie alla loro indipendenza, in caso di problemi è sufficiente rilanciare e/o sospendere soltanto il servizio interessato, senza intaccare l'integrità dell'applicazione stessa.

L'architettura a microservizi offre il vantaggio ulteriore di poter rendere dinamicamente scopribili i servizi all'interno di un'organizzazione. Per offrire questa tipologia di *feature*, è necessario disporre di un modulo che si occupi della "*registrazione*" di questi servizi. Successivamente, vi sarà possibilità di indicare quale servizio svolga quale compito, in maniera tale da selezionare con un criterio il candidato per portare a termine un determinato compito.

Maggiori dettagli relativi all'architettura a microservizi possono essere trovati ad esempio su `https://microservices.io/`.

3.1.4 Selezione finale

Considerando le informazioni raccolte nelle sezioni precedenti emergono diversi dettagli non trascurabili per la valutazione dell'architettura finale. Come analizzato nella sezione 3.1.3, l'architettura a **Microservizi** offre notevoli vantaggi, tra cui la granulosa modularità dei componenti in progetto. Di conseguenza, essa risulterà essere sicuramente il punto di riferimento per i prossimi capitoli.

Per quanto riguarda l'analisi dei vincoli:

- La **scelta della libreria** per il supporto di WebRTC ricade su **PeerJS**. Essendo completamente gratuita e facile da integrare nel codice, risulta essere una scelta migliore per soddisfare i requisiti. L'aggiunta di un servizio per il *PeerJSServer* si rende notevolmente agevolata grazie alla scelta dell'architettura a microservizi, in quanto è possibile ottenere un tale server direttamente dai repository della libreria[2] e includerlo senza difficoltà come un modulo completamente separato dal resto del progetto. Si sottolinea comunque che la modularità del sistema dovrebbe garantire l'eventuale cambio di libreria, se questo dovesse mai essere eseguito.

- La **suddivisione tra *backend* e *frontend*** è possibile, e risulta essere comodamente gestibile, sempre grazie all'adozione dell'architettura a microservizi. Infatti, è possibile valutare di avere un punto di ingresso al sistema con delle API comuni, il quale risolve automaticamente il problema di *bilanciamento del carico*, nascondendo la struttura sottostante

[2] `https://github.com/peers/peerjs-server`

del *backend*. Il *frontend*, dall'altro lato, verrà gestito con un approccio successivamente valutato, che non dipenderà in alcun modo dalle scelte effettuate per la parte del *backend*.

3.2 Strutturazione dati

Prima di trattare in via generica le API fornite dal servizio, è necessario fornire un modello di dati valido affinché le informazioni presenti all'interno del sistema siano univoche e corrispondenti all'interfaccia esposta. Come base si utilizza il modello del dominio esplicitato in figura 2.2, composto durante l'analisi dei requisiti.

La figura 3.7 mostra la composizione di tali entità, esplicitando maggiormente le relazioni che intercorrono tra di esse. È doveroso sottolineare che tale struttura verrà utilizzata sia lato *backend* che *frontend*, in quanto entrambi necessitano di utilizzare gli stessi dati. Nello schema non sono presenti dettagli relativi al server di PeerJS, in quanto il suo utilizzo è completamente trasparente dal punto di vista del *backend*: gli unici dati di cui quest'ultimo è a conoscenza risultano essere quelli relativi ai "*PeerID*" dei partecipanti (discussi nella sezione 3.5).

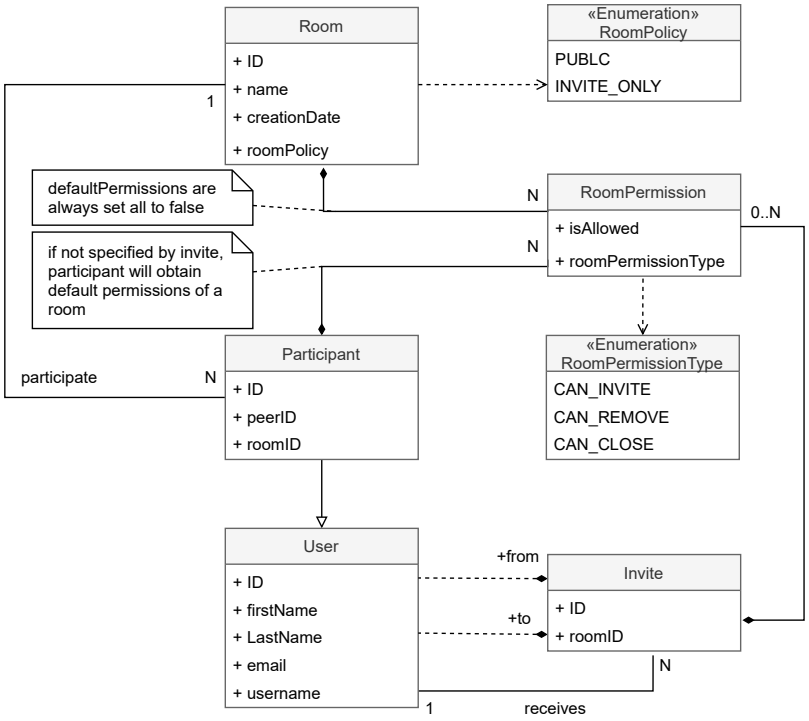


Figura 3.7: Classi e attributi del Call Service.

3.3 API

3.3.1 API Chiamata

Le API vengono progettate nel pieno rispetto dei requisiti posti. Lo schema in figura 3.8 mostra in maniera sintetica le API previste per la gestione del servizio di (video)chiamate. Per semplicità non vengono mostrati gli attributi in ingresso e in uscita, che verranno trattati in maniera approfondita più avanti in questa sezione, e vengono omessi i nomi delle sotto-risorse, ove questi non sono presenti. Inoltre, si intendono utilizzare le risorse del diagramma 2.2 e per ora si escludono le API dovute alla libreria PeerJS.

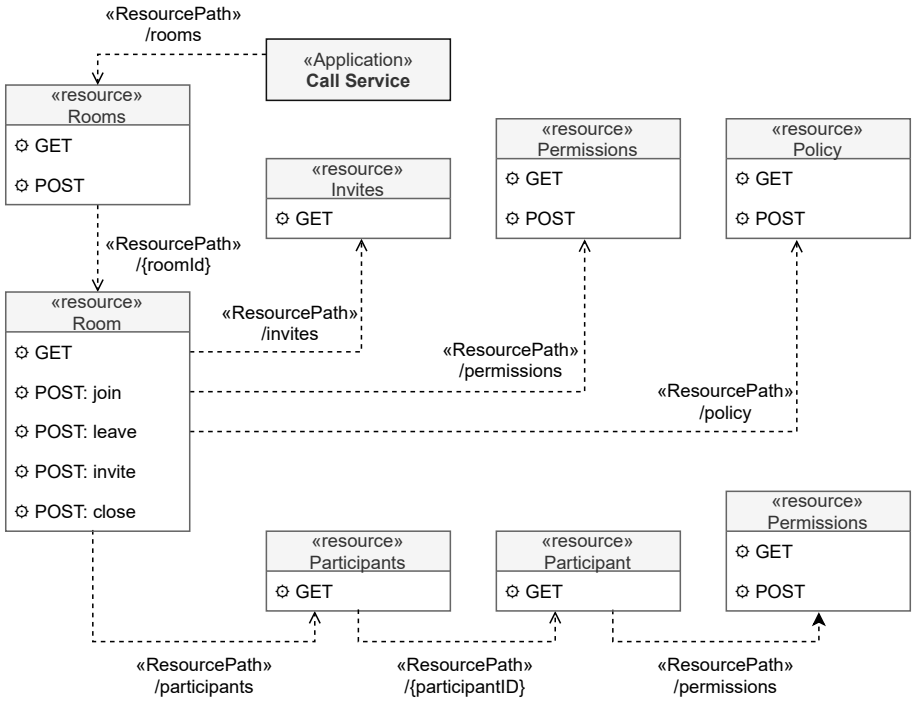


Figura 3.8: Schematizzazione delle API offerte dal *backend*.

Le chiamate di `GET` restituiscono tutte la risorsa (o le risorse) corrispondenti a quella determinata categoria: a scopo dimostrativo si elencano alcuni esempi possibili:

- `GET` sulla risorsa `/rooms` ottiene tutte le stanze (pubbliche) attualmente attive nel sistema;
- `GET` su `/rooms/{roomId}/invites` permette di avere l'accesso a tutti gli inviti di quella determinata stanza;

- `GET` su `/rooms/{roomId}/participants/{participantID}` ottiene i dettagli di quel determinato partecipante in quella determinata stanza.

I permessi relativi a chi può accedere a determinate risorse sono definiti in base al ruolo:

- un **moderatore** può accedere a qualsiasi risorsa relativa alla chiamata stessa;
- un **partecipante** può accedere soltanto ai parametri necessari allo stabilimento della connessione, ovvero alla lettura dei dettagli della stanza (nonchè dei suoi permessi e policy) e dei partecipanti, una volta che vi è entrato. Non può né gestire la chiamata né ottenere informazioni non necessarie ai fini della connessione (ad esempio, chiedendo i permessi di un altro partecipante);
- un **utente** può soltanto leggere le stanze (pubbliche) attualmente attive e crearne una, oppure tentare di unirsi ad una stanza esistente.

Per trattare dei parametri richiesti relativi alle ulteriori risorse, si propone lo schema 3.9, che utilizza le classi definite nella figura 2.2. Per semplicità, non si includono collegamenti direttamente rilevabili dal diagramma 3.8 e la struttura delle API viene riassunta in tabelle leggibili. Si vuole sottolineare che la rappresentazione degli oggetti è puramente dimostrativa, in quanto tutto il contenuto dei messaggi inviati sarà esclusivamente in formato JSON, come definito nella sezione 3.7.3. Vengono inoltre supposte alcune considerazioni:

- per ogni `GET` senza ritorno - l'oggetto ritornato è la risorsa (o le risorse) che corrispondono a quel determinato path. Ad esempio, una `GET` su `rooms/{roomID}/participants` restituisce tutti i partecipanti di quella determinata stanza.
- per ogni `POST` senza ritorno - l'oggetto ritornato è un "*HTTP Response Status Code*"[3], nel caso di operazione avvenuta con successo. Nel caso di insuccesso, è ritornato un "*RestError*", come spiegato nella sezione 3.4. Ad esempio, una `POST` su `rooms/{roomID}/leave` restituisce 200 in caso di successo.
- per ogni risorsa non trattata - i soprastanti punti valgono anche per le risorse non trattate; esse non vengono riportate soltanto per semplicità di visualizzazione.

[3] `https://developer.mozilla.org/it/docs/Web/HTTP/Status`

/rooms		
GET	*Retrive all (public) rooms*	
	Input	-
	Output	*JSON object representing a collection of rooms.* `rooms: Room[]`
POST	*Creates a new room*	
	Input	*JSON object representing a Create Room Request.* `username: string, roomName: string, roomPolicy: RoomPolicy, defaultPermissions: RoomPermission[], invites: Invite[]`
	Output	*JSON object represeting a Create Room Response.* `roomID: string, inviteResults: InviteResults[]`

(a) API sulla risorsa `/rooms`.

/rooms/{roomID}/permissions /rooms/{roomID}/participants/{participantID}/permissions		
GET	*Retrive permissions of that Room (or of that participant)*	
	Input	-
	Output	*JSON object representing a collection of permissions.* `permissions: Permission[]`
POST	*Set permissions of that room (or of that participant)*	
	Input	*JSON object representing a Permission Set Request.* `permissions: Permission[]`
	Output	`HTTP Response Status`

(b) API sulla risorsa `/rooms/{roomID}/permissions`

.

/rooms/{roomID}/policy		
GET	*Retrive policy of that Room*	
	Input	-
	Output	*JSON object representing the policy of the room.* `roomPolicy: RoomPolicy`
POST	*Set policy of that room*	
	Input	*JSON object representing a Policy Set Request.* `roomPolicy: RoomPolicy`
	Output	`HTTP Response Status`

(c) API sulla risorsa `/rooms/{roomID}/policy`.

Figura 3.9: API proposte dal *backend* (parte 1)

/rooms/{roomID}		
GET	*Retrive a specific room details*	
	Input	-
	Output	*JSON object representing a room.* `room: Room`
POST	`/join` – *join into a room*	
	Input	*JSON object representing a Join Room Request.* `peerId: string, username: string`
	Output	*JSON object represeting a Join Room Response.* `room: Room`
POST	`/leave` – *leave a room*	
	Input	*JSON object representing a Leave Room Request.* `Username: string`
	Output	`HTTP Response Status`
POST	`/invite` – *invite a user into a Room*	
	Input	*JSON object representing a Invite Room Request.* `usernameFrom: string, usernameTo: string, permissions: RoomPermission[]`
	Output	*JSON object represeting a Invite Room Response.* `inviteID: string`
POST	`/close` – close *a room*	
	Input	*JSON object representing a Close Room Request.* `username: string`
	Output	`HTTP Response Status`

Figura 3.10: API sulla risorsa `/rooms/{roomID}` (parte 2).

3.3.2 API Utenti

Oltre agli aspetti gestiti puramente dal sistema delle stanze, esistono ulteriori due aspetti fondamentali da coprire in un sistema di questo genere. Infatti, nei requisiti è specificato che dev'essere necessario invitare utenti e di conseguenza avere la possibilità di gestire tali inviti. Inoltre, relativamente alla sezione 3.6.1, è emerso l'aspetto di autenticazione, il quale dev'essere opportunamente coperto. Le API offerte possono essere schematizzate secondo le seguenti figure.

Le premesse relative allo schema degli utenti riguardano perlopiù l'aspetto prototipale di tale servizio: non essendo una priorità quella di gestire in modo approfondito l'aspetto riguardante la gestione, registrazione e autenticazione degli utilizzatori, si prevedono soltanto API basilari. Nonostante ciò, si ritene opportuno progettare sin dall'inizio un servizio capace di offrire tale capacità, in modo da essere facilmente estensibile e integrabile in futuro con una soluzione capace di cogliere tutti gli aspetti riguardanti questo dettaglio.

Un'altra considerazione riguarda la strutturazione delle risorse: infatti, nonostante gli inviti facciano riferimento a quelli trattati nelle stanze, gli utenti hanno modo di interagire coi propri anche senza essere presenti in una stanza. Per questo motivo è necessario esplicitare che gli inviti personali di un utente, pur essendo lo stesso oggetto, hanno API maggiori che permettono, ad esempio, la loro accettazione/rifiuto.

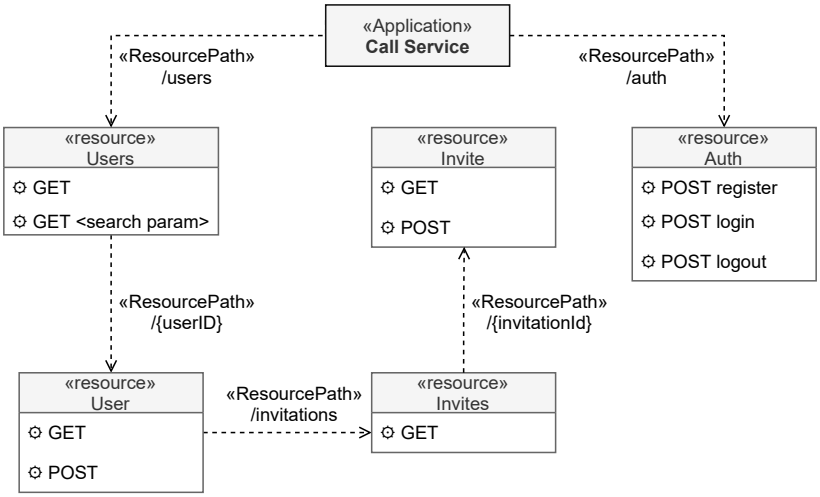


Figura 3.11: API sugli utenti.

/user		
GET	*Retrive all users*	
	Input	-
	Output	*JSON object representing a collection of users.* `users: User[]`
GET	`<filter>` – *retrive all users that correspond to a given filter*	
	Input	-
	Output	*JSON object representing a collection of users.* `users: User[]`

(a) API sulla risorsa `/users`.

/user/{userID}		
GET	*Retrive data of a specific user*	
	Input	-
	Output	*JSON object representing an users.* `user: User`
POST	*Set details about a user*	
	Input	*JSON object representing an users.* `users: User[]`
	Output	`HTTP Response Status`

(b) API sulla risorsa `/users/{userID}`

/user/{userID}/invites		
GET	*Retrive all invites for that specific user*	
	Input	-
	Output	*JSON object representing a collection of invites.* `invites: Invite[]`

(c) API sulla risorsa `/users/{userID}/invites`.

/user/{userID}/invites/{inviteID}		
GET	*Retrive that specific invite*	
	Input	-
	Output	*JSON object representing aaninvites.* `invite: Invite`
POST	*Set details about that invite*	
	Input	*JSON object representing aaninvites.* `invite: Invite`
	Output	`HTTP Response Status`

(a) API sulla risorsa `/users/{userID}/invites/{inviteID}`.

/auth		
POST	*/register – register a new user into the system*	
	Input	*JSON object representing a User and a hashed password.* `user: User, password: string`
	Output	`HTTP Response Status`
POST	*/login – authorize a user to use the system*	
	Input	*JSON object representing username and hashed password.* `username: string, password: string`
	Output	*JSON object representing JWT token to use.* `token: string`
POST	*/logout – perform the loggin out of the user from the system*	
	Input	-
	Output	`HTTP Response Status`

(b) API sulla risorsa `/auth`.

Figura 3.13: API proposte dal *backend* sugli utenti (parte 2)

3.4 Standardizzazione

Una delle clausole poste durante la definizione dei requisiti riguardava una piena compatibilità del sistema, per poter essere in grado, al bisogno, di includerlo in ulteriori progetti. Per far fronte a tale richiesta non è sufficiente trattare in maniera approfondita soltanto le API, ma anche il modo in cui dati vengono poi restituiti agli utilizzatori.

Per quanto riguarda i **risultati** delle chiamate, come definito durante il design architetturale nella sezione 3.7.3, verrà utilizzato un mapping in formato JSON per quanto riguarda tutti i dati inviati dal *backend* e provenienti dal *frontend*. Questo garantisce che vi sia un unico standard ben definito per ciò che concerne la lettura dei dati da parte delle applicazioni che utilizzeranno questa soluzione, e non vi è quindi necessità di analizzarne ulteriori, onde evitare di complicare notevolmente il progetto.

Un'analisi più approfondita va fatta invece per quanto riguarda gli **errori**, costituenti una parte imprescindibile del modello analizzato. Le cause possono essere molteplici, tra cui quelle maggiori riguardano malfunzionamenti interni oppure un'errata formazione dei parametri in ingresso. Anche se esiste già un modo standard per la restituzione dei cosiddetti "*HTTP Response Status Codes*", esso viene ritenuto non sufficiente per la segnalazione di dettagli relativi a *cosa* e *dove* sia sbagliato. Grazie ad una continua evoluzione del Web è possibile però assistere alla nascita di diversi standard, dei quali uno in particolare offre la possibilità di estendere i classici codici di errore con dei campi standard.

Si tratta dello standard denominato **RFC 7807** [28]: un modello applicabile opzionalmente per la segnalazione degli errori in formato del tutto compatibile con REST. Definisce un modo sia machine-readable, sia human-readable per specificare il luogo, l'entità e i dettagli di un problema che può capitare durante l'elaborazione di una richiesta. Non sostituisce il codice HTTP standard, ma ne risulta essere un'opzionale estensione.

Lo schema prevede 5 parti aggiuntive al body dell'errore:

- `type` - tipologia dell'errore, meglio se definita tramite un URI,
- `title` - un messaggio breve human-readable specificante l'errore,
- `status` - HTTP Response Status Code,
- `detail` - dettagli human-readable a proposito dell'errore,
- `instance` - location dove l'errore è avvenuto (possibilmente URI).

Seguendo le indicazioni di tale direttiva è possibile di conseguenza ampliare la struttura dell'architettura, introducendo un microservizio extra: quest'ultimo avrà lo scopo di provvedere all'identificazione di errori, trattandoli come se fossero delle risorse intrinseche all'interno del sistema. Offrirà una sua API ben strutturata, la quale restituirà, se richiesto, i dettagli dei problemi capitati durante l'esecuzione della soluzione. Avendo ben definito la struttura del servizio delle chiamata, è possibile progettare l'architettura di un **RestError**, permettendo così di coprire anche questa funzionalità.

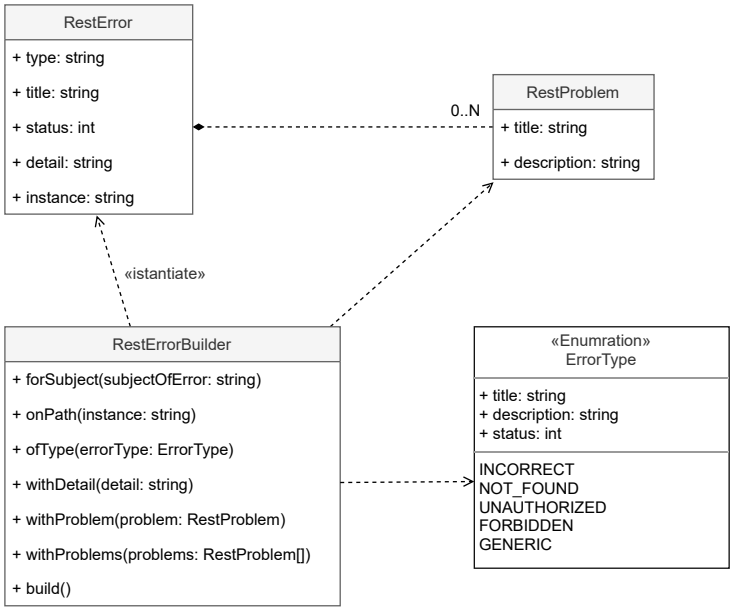


Figura 3.14: Classi che definiscono un RestError.

Il servizio, di conseguenza, è strutturato in modo tale da avere un percorso standard per ogni tipo di errore. Può essere richiamato sia a livello di utente (ad esempio, se vuole avere maggiori dettagli relativi ad un errore o ad una classe di errori), sia utilizzato in maniera automatica per ottenere dati relativi a quell'errore quando esso capita durante un normale utilizzo da parte di un client. Si vuole precisare che nello schema 3.15 sono omessi per semplicità gli attributi "*ResourcePath*", in quanto coincidono esattamente col nome della risorsa. Il diagramma contiene anche un'ulteriore semplificazione, in quanto tutte le API di secondo livello sono connesse a quelle del terzo: per non complicare la sua lettura, è stato evidenziato soltanto il collegamento con il primo di essi.

Si vuole sottolineare come tale servizio possa essere esteso: è possibile, infatti, offrire direttamente un'API per includere nuovi errori a run-time, in modo

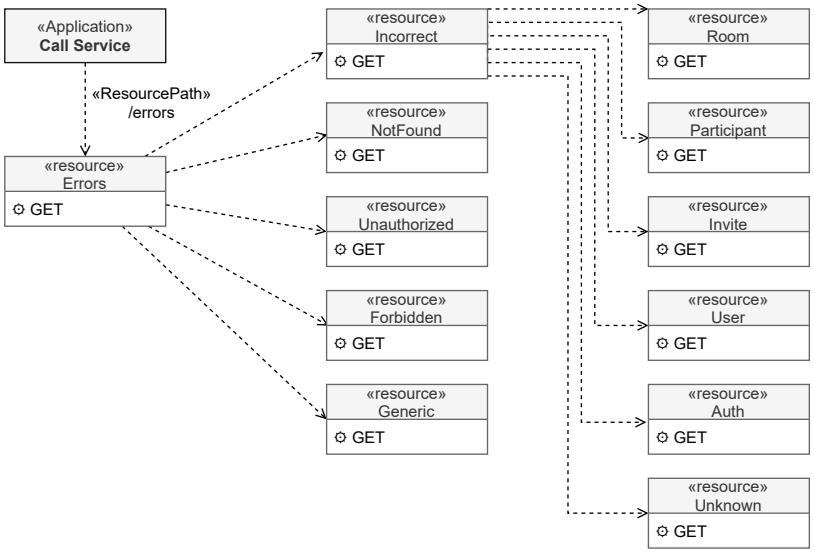


Figura 3.15: API che definiscono un RestError. I nomi di "*ResourcePath*" sono omessi in quanto sempre coincidenti con la risorsa. Inoltre, tutte le risorse del secondo livello sono connesse a tutte quelle del terzo: per semplicità di lettura ne è stata evidenziata soltanto una.

da essere ancora più dinamici. Ovviamente, l'aggiunta di ulteriori categorie di errori spetta ad eventuali manutentori del sistema: gli utenti potrebbero avere soltanto la facoltà di segnalarli ad un eventuale team di analisi, il quale provvederà a scartare le richieste irrilevanti. Inoltre, il servizio così strutturato è estendibile verso ulteriori ambiti: è soltanto necessario aggiungere percorsi, che si vogliono coprire con gli errori definiti, e l'API sarà automaticamente sfruttabile anche per quell'ambito.

Il risultato di tale progettazione è di indubbio valore; si vogliono considerare a scopo dimostrativo alcuni esempi:

- l'utente passa i parametri errati nel metodo per creare una stanza: in questo caso l'errore identificato dall'interfaccia è quello di "*INCORRECT*", mappato direttamente nel concetto di "*Bad Request*" (codice 400). Di conseguenza, la risposta sarà verosimilmente strutturata come segue:
  - `type` - `errors/incorrect/room`,
  - `title` - "*The given parameter was not correct.*",
  - `status` - "*400*",
  - `detail` - "*The parameter is invalid: <invalid_param_name>*",
  - `instance` - `/rooms`.

- l'utente senza aver effettuato il login sta cercando di effettuare una "*join*" all'interno di una stanza: in questo caso l'errore identificato dall'interfaccia è quello di "*UNAUTHORIZED*", mappato nell'omonimo codice HTTP 401. La risposta sarà verosimilmente strutturata come segue:
  - `type` - `errors/unauthorized/room`,
  - `title` - "*User is not logged in.*",
  - `status` - "*401*",
  - `detail` - "*User is not logged in.*",
  - `instance` - `/rooms/join`.

Si vuole precisare che tale servizio non verrà incluso nella soluzione finale: i requisiti non prevedono una tale gestione e l'effort per crearlo potrebbe portare ad una sovra-ingegnerizzazione della soluzione. Si prevede tuttavia la strutturazione di `RestError` come entità interna (presumibilmente come classe), in modo da provvedere ad una sua facile estrazione dal contesto ed isolamento in un microservizio extra.

3.5 Creazione di una chiamata (con PeerJS)

È necessario a questo punto progettare meglio l'interazione che si vuole avere per completare non solo la strutturazione della chiamata ma anche la sua effettiva realizzazione. Come accennato nella sezione 3.1.1, l'architettura puramente P2P, sia di WebRTC che della sua implementazione usata in questo ambito (PeerJS), non permette di centralizzare tutto il flusso delle informazioni e vi è necessità di appoggio sulle API definite dalla libreria stessa.

È infatti previsto **PeerJS Server**, direttamente ottenibile dal repository di tale libreria, il quale garantisce la piena operatività. Le API risultano essere nascoste dall'implementazione di PeerJS, ma grazie alla natura open-source del codice è possibile consultarle e/o modificarle a piacimento. L'obiettivo di questa analisi, però, non è quello di esplorare in dettaglio il suo funzionamento ma di utilizzarlo direttamente e prevedere un modo per unire la struttura della chiamata all'effettiva chiamata tra utenti.

A questo punto, piuttosto che trattare di un generico caso di PeerJS applicato ad un sistema, si preferisce progettare flussi di controllo capaci di ottenere il risultato voluto. Si pianificano, di conseguenza, le azioni necessarie affinché una chiamata possa essere ritenuta istanziata. Nel diagramma 3.16:

- si suppone che l'utente abbia già effettuato il login e ad ogni richiesta venga allegato il token JWT;

- non si riportano i singoli servizi responsabili alle azioni ma un generico schema di tale flusso;
- si suppone che ci sia già una stanza esistente con dentro un partecipante;
- si omette l'architettura più interna di PeerJS, nascosta dalle API di libreria.
- per chiarezza, sono indicati i tipi di parametri che vengono ritornati dal *backend*. Nel caso di flussi audio/video (e quelli di dati) non viene specificato nulla, in quanto gestito più a livello di libreria PeerJS.

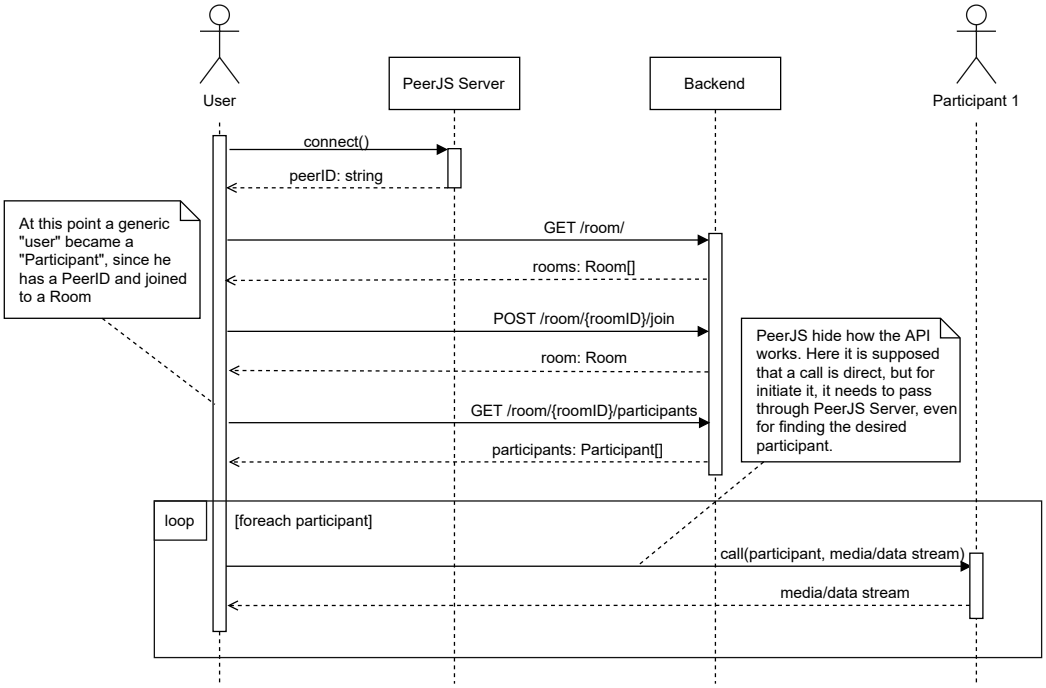


Figura 3.16: Flusso per poter partecipare ad una chiamata. Non vengono esplicitate le API nascoste di PeerJS Server e si suppone che la richiesta di una (video)chiamata arrivi direttamente al destinatario.

Una nota importante per quanto riguarda lo schema 3.16 è il fatto di sottolineare che un utente diventi un partecipante quando si unisce ufficialmente ad una stanza, ottenendo prima un PeerID: questo garantisce che egli possa effettivamente *partecipare* ad una conferenza e che possa essere chiamato da altri utenti che si aggiungeranno in un successivo momento. Inoltre, lo stesso schema chiarisce come essenzialmente il *backend* sia completamente all'oscuro

dell'interazione svolta tra i diversi partecipanti: la comunicazione avviene soltanto tra i membri della stanza. Ovviamente, è sempre possibile richiamare tutte le API del *backend* anche nello stato di *partecipante*.

3.5.1 Analisi criticità

Grazie ad un'attenta progettazione è possibile prevedere comportamenti anomali e/o indesiderati all'interno del proprio sistema prima che esso diventi effettivamente funzionante. La soprastante analisi offre un quadro generale per quel che riguarda un tale servizio, che funziona in tutti i casi "*ottimali*". Nella realtà dei fatti, spesso si verificano problematiche di diverso tipo, alle quali l'intera struttura deve far fronte e reagire in maniera tale da annullare o minimizzare i danni dovuti a questo tipo di errori. Si analizzano in dettaglio degli scenari che potrebbero capitare durante l'utilizzo, per vedere se il sistema possa rispondere in modo adeguato.

In primo luogo, l'utente potrebbe richiedere l'accesso a risorse a lui non consentite (come ad esempio la chiusura di una stanza senza essere un moderatore). Per affrontare questo problema è necessario verificare che la richiesta provenga da un partecipante che abbia i permessi necessari per eseguire tale operazione. Essendo il token JWT unico per l'utente che ha effettuato il login, è possibile incrociare il dato "*token*" al "*nome utente*" ed ottenere la validazione (o il rifiuto) di tale richiesta. Per questo motivo, in alcune delle richieste è esplicitamente indicato di dover inserire lo username come parametro non opzionale.

Un'altra problematica che si può analizzare è dovuta ai generici errori di ottenimento dati. Se l'utente richiede una stanza che è stata chiusa nel momento in cui lo richiedeva, potrebbe ottenere un indirizzo verso una risorsa non più presente. Sarà compito del *frontend* assicurarsi della validità dei dati, ma anche il *backend* offrirà una protezione verso tale scenario: facendo ad esempio una `GET` su un ID di una stanza non più presente, verrà restituito uno dei già progettati *RestError* con il codice "*NOT_FOUND*". Essendo asincrona la generale natura del *backend*, occorre pianificare con estrema precisione le azioni svolte. I controlli sul codice dovranno essere presenti in ogni operazione che possa richiedere una tale interazione, sia interna al servizio che esterna (ad esempio verso un database). Soltanto isolando ogni possibile errore si può garantire che l'asincronia diventi un vantaggio, e non un peso, sia per lo sviluppo che per la successiva manutenzione.

Analizzando successivamente la struttura della chiamata, è possibile che si verifichino diversi eventi non controllati: caduta della connessione, uscita non "*graceful*" da una stanza o alcuni partecipanti impossibilitati a connettersi sono soltanto alcune delle problematiche che potrebbero capitare. Si ricorda però che la natura del sistema delle chiamate, tralasciando una minima interazione col *backend*, è interamente P2P: sono i diversi partecipanti a dover regolare la chiamata stessa, per cui, in questo caso, non è compito del *backend* assicurarsi che la comunicazione avvenga senza intoppi. Per affrontare però un caso particolare, ovvero quello in cui nessuno dei partecipanti riesce a connettersi ad un utente specifico, si può predisporre un'ulteriore API sul servizio delle chiamate. Quest'ultima avrà il compito di segnalare che vi è un partecipante impossibile da raggiungere e che dev'essere rimosso per poter ritentare di collegarsi. Un'eventuale API di questo genere potrebbe servirsi del "*PeerID*" del partecipante e la sua segnalazione proverrebbe da uno degli utenti in chiamata.

I problemi di questa soluzione risultano essere molteplici, ma non sono facilmente affrontabili:

- i partecipanti alla chiamata dovrebbero mettersi d'accordo su quali sono gli utenti attualmente non raggiungibili. Uno di essi avrà il ruolo di "*master*" di questa decisione, inizializzandola e inviando ad ognuno dei partecipanti una richiesta, chiedendo quale sia lo stato di quell'utente per lui non raggiungibile. Al ricevere di tutte le risposte, attraverso un *criterio di scelta* dovrebbe decidere se mandare o no la richiesta al server per rimuovere l'utente non raggiungibile.

- nel precedente punto è stato citato un *criterio di scelta* che bisognerebbe definire in modo tale da non disconnettere eventuali utenti attualmente presenti e attivi in chiamata, penalizzando soltanto quelli che effettivamente hanno difficoltà di connessione. Tale criterio potrebbe essere intelligente, in modo da capire che in realtà esiste un partecipante che non vede un membro della chiamata ma che quest'ultimo risulta essere pienamente operativo per gli altri. Data questa situazione, il "*master*" della votazione potrebbe indicare al partecipante con difficoltà di riaggiornare la propria connessione e di riunirsi in chiamata.

- nel primo e nel secondo punto è stato citato un ruolo di "*master*", il quale dovrebbe essere scelto tra gli utenti. Anche in questo caso vi è bisogno di un *criterio di scelta* per la selezione di questo utente, che a tutti gli effetti *può* avere un indiretto permesso di rimuovere i membri dalla stanza, pur non essendo un moderatore.

- potrebbe darsi che non sia il partecipante irraggiungibile ad avere problemi, ma tutti i membri della chiamata che non riescono raggiungere

quel determinato membro. Di conseguenza, tutti i membri, tranne quello "*problematico*", dovrebbero riaggiornare la propria connessione nella stanza.

Tali complessità introducono un effort non banale nella ricerca delle soluzioni possibili. Il progetto ha però una natura prototipale: si ritiene necessario sottolineare in modo molto evidente questi aspetti, senza però andare nel dettaglio di come si potrebbe risolverli, in quanto non costituiscono esattamente il compito di questa analisi. Si rimarca però che l'API del partecipante non raggiungibile verrà presa in considerazione durante lo sviluppo e, nel caso, verrà identificata nella risorsa "*Partecipante*" grazie all'indicazione del suo ID e PeerID.

3.6 Architettura

Ricostruendo tutto il percorso fin'ora effettuato, si può concludere con la seguente suddivisione dei diversi moduli:

- **Frontend** - modulo che si occupa della parte comunicante con il *backend*. Esposto all'utente finale, sarà l'interfaccia grafica che permetterà di fruire del servizio (ad esempio, instaurando di collegamenti audio/-video). Integrerà al suo interno parti della libreria PeerJS per il suo corretto funzionamento.
- **Backend** - modulo che si occupa della logica dell'applicazione. Si comporrà di diversi servizi:
  - **Gateway** - è un microservizio esposto all'esterno; il punto d'ingresso dell'applicazione con il quale diversi utenti si interfacciano.
  - **Call Service** - microservizio che si occupa della gestione delle risorse relative alle (video)chiamate.
  - **PeerJS Server** - il server richiesto dalla libreria di PeerJS per l'instaurazione delle (video)chiamate.
  - **Registry Service** - microservizio abilitato a raccogliere le sottoscrizioni di altri microservizi, in modo da renderli dinamicamente scopribili. Sarà compito del gateway selezionare il servizio opportuno: il registry fornirà soltanto i riferimenti ad esso.
  - **User Service** - microservizio che organizza gli utilizzatori del sistema, in modo da preservare le loro informazioni.

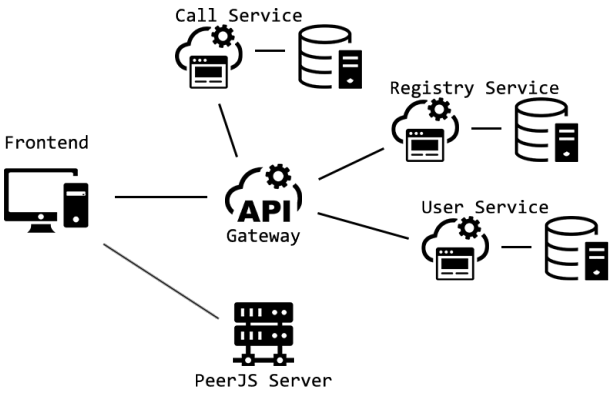


Figura 3.17: Strutturazione del progetto: si schematizzano i vari microservizi coinvolti e la comunicazione tra di essi. Ad ognuno dei servizi che lo richiede, è previsto anche un collegamento con un database.

3.6.1 Verifica casi d'uso

Per verificare una tale progettazione architetturale, si esegue una possibile percorrenza ad alto livello delle richieste che possono provenire dai client, in modo da verificare i diversi caso d'uso precedentemente esposti.

1. **L'utente entra nel sistema**: criticità rilevata. In che modo l'utente "*entra*" nel sistema? Vi è necessità di introdurre un modulo capace di autorizzare gli utenti per fruire dei servizi proposti: i dati sono salvati nello User Service, ma non esiste alcun modo per autorizzare e/o autenticare operazioni nel sistema.

2. **L'utente richiede la creazione di una chiamata**: nessuna criticità. Il Call Service fornirà API necessarie affinchè la chiamata possa avvenire e la diretta comunicazione con PeerJS Server permette un corretto ottenimento e gestione dei Peer.

3. **L'utente partecipa alla chiamata**: nessuna criticità. Sia che la richiesta venga dal creatore che da un utente esterno, il Call Service e PeerJS Server offriranno le API corrette per effettuare tale operazione.

4. **L'utente modera e/o gestisce la chiamata**: nessuna criticità. Il Call Service e PeerJS Server offriranno le API corrette per effettuare tale operazione.

3.6.2 Correzione criticità

Considerando le criticità rilevate, si riprogetta l'architettura, cercando di porre maggiore attenzione sui requisiti non completamente coperti. La figura 3.18 ne rappresenta l'ufficiale evoluzione, separando concretamente i concetti. Essa sarà ritenuta l'architettura ufficiale per il progetto a seguire.

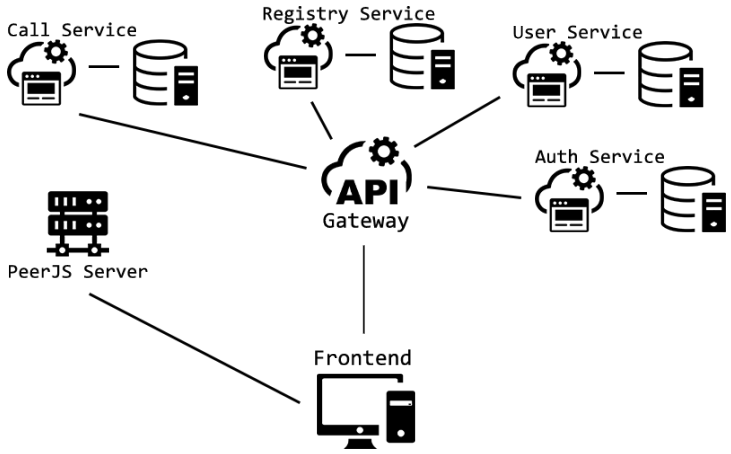


Figura 3.18: Strutturazione definitiva del progetto.

È necessario approfondire il servizio per l'autenticazione, in quanto si deve prevedere un meccanismo per una efficiente fruizione del servizio da parte degli utenti. Non avendo alcun requisito per quanto riguarda tale servizio, ma riconoscendo la sua effettiva necessità, si provvede alla selezione di uno standard comunemente usato in molte piattaforme, chiamato `JWT` [22]. Esso prevede la generazione di un token a fronte di un corretto *pairing* tra nome utente e password dell'utilizzatore. Questo token verrà poi fornito all'utente che provvederà ad allegarlo in ogni sua richiesta. Di conseguenza, le API che necessiteranno di un'autenticazione, verificheranno la presenza e l'autenticità di tale token. Quest'ultimo dev'essere mantenuto in una struttura dati capace di offrire elevate performance: si consiglia, ad esempio, l'utilizzo di Redis (un database in-RAM) per minimizzare i ritardi dovuti alle suddette verifiche.

3.7 Tecnologie Utilizzate

Componendosi di diversi moduli, l'applicazione utilizzerà diverse tecnologie e linguaggi al suo interno. Anche la scelta di un linguaggio di programmazione rispetto ad un altro può influenzare non banalmente il progresso compiuto durante la realizzazione del progetto, per cui è necessario porre particolare attenzione alle competenze che si hanno nell'ambito dell'architettura prevista. In

figura 3.19 è possibile apprezzare lo schema generale delle tecnologie utilizzate, trattate in dettaglio nelle seguenti sezioni.

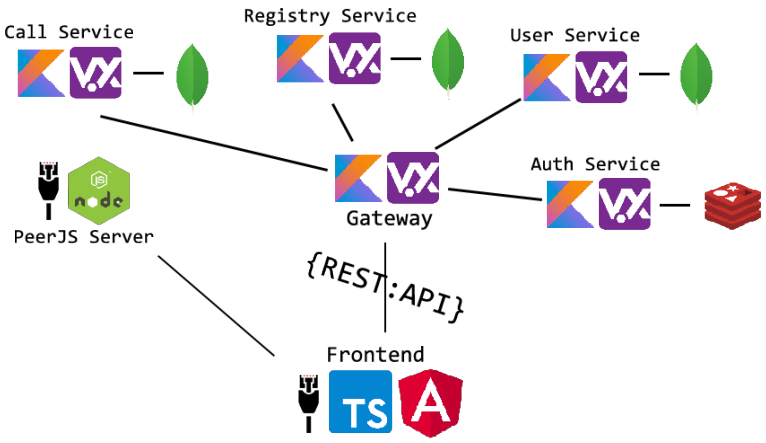


Figura 3.19: Rivisitazione della figura 3.18 con le tecnologie utilizzate nel progetto. Angular, TypeScript e PeerJS (*frontend*), Kotlin, Vert.X e PeerJS (*backend*), MongoDB e Redis (database). La comunicazione è garantita grazie alle REST API e i dati sono in formato JSON.

3.7.1 Frontend

Il *frontend*, occupandosi più della parte relativa all'utilizzatore finale, dev'essere consultato attraverso device muniti di un display. Esistono diversi framework che offrono lo sviluppo su specifiche piattaforme (quali ad esempio Android, iOS, Windows o Mac) e quelli che permettono di unire alcuni (o tutti) gli aspetti di una soluzione, definendola come un'applicazione "*Cross Platform*".

L'ambito più universale tra tutti quelli che possono esistere al giorni di oggi rimane comunque il **web**: essendo accessibile da essenzialmente tutti i dispositivi, si rende la fruizione del servizio automaticamente possibile per una vasta gamma di utenti. Grazie a questa universalità è superfluo analizzare ulteriori modalità, in quanto, partendo dai requisiti, è richiesta un'interoperabilità generale tra tutti i moduli realizzati: eliminando quasi del tutto i vincoli di piattaforma, si ottiene esattamente tale obiettivo.

Una nota negativa riguardante lo sviluppo web è una frammentazione del supporto per quanto riguarda diverse *feature*, introdotte come standard ma praticamente implementate in modo parziale o proprietario dalle software-

house di quel determinato browser[4]. L'utilizzo di librerie che accomunano soluzioni cross-browser sicuramente può agevolare lo sviluppo, ma si vuole rimarcare che, nonostante la presenza di diversi enti partecipanti alla definizione del *living standard* di HTML e CSS, il problema della diversità tra essi permane da anni (tendendo soltanto in maniera molto timida a decrescere nel tempo).

Considerando che il progetto sarà un prototipo dimostrante le possibilità nell'ambito delle videochiamate, è necessario selezionare un framework capace di rispettare soltanto il requisito della modularità. Non essendoci vincoli per quanto riguarda l'effettiva realizzazione della user experience, non sono necessarie approfondite ricerche in tale ambito, indirizzando la scelta del framework verso una preferenza personale. Guardando al background posseduto, la più ovvia scelta cade nell'utilizzo di **Angular** [19] con un diretto supporto a **TypeScript** [30] come libreria di implementazione per il *frontend*.

Il vantaggio offerto da questa soluzione è di indubbio valore: oltre a poter organizzare l'intera struttura del sito web in moduli, è possibile definire e riutilizzare componenti in tutte le pagine. Il principio sfruttato da questa tecnologia si basa su SPA (Single Page Application), dove una singola pagina è composta da diversi componenti, aggiornati eventualmente in tempo reale. Il cambio di percorso, in realtà, non ricarica l'intera pagina, ma sostituisce semplicemente alcuni componenti di cui si compone: quelli non richiesti vengono distrutti e i nuovi sono caricati. Per far funzionare questa soluzione è però necessario introdurre un ulteriore servizio il quale riconosce queste richieste degli utenti e le traduce nei corrispettivi comandi.

L'utilizzo di **TypeScript** semplifica notevolmente lo sviluppo grazie all'introduzione dei concetti maggiormente conosciuti nei linguaggi di programmazione ad oggetti (come ad esempio Java), quali classi, enum e una "*leggera forzatura*" sulla tipizzazione delle variabili (in quanto è possibile comunque utilizzare il tipo "*qualsiasi*", ma è fortemente sconsigliato). L'intera struttura del linguaggio gli permette di essere perfettamente compatibile con JavaScript, estendendo le sue possibilità.

Per quanto riguarda lo stile, è possibile utilizzare i classici fogli CSS con i meccanismi di classi e `ID`. La differenza sta nel poter indicare per ogni componente un **suo** foglio di stile, che non verrà visto da nessun altro componente:

[4]Esiste un comodo tool online per la verifica di tale aspetto. Per quanto riguarda, ad esempio, WebRTC si può considerare il seguente url `https://caniuse.com/?search=webrtc`

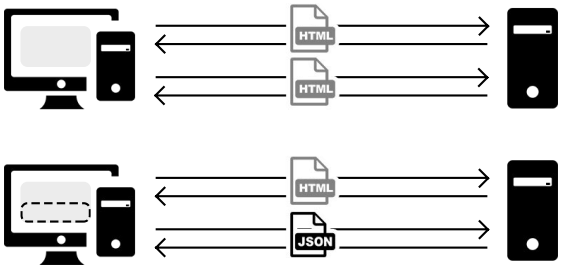


Figura 3.20: Differenze tra approccio "*classico*" e quello SPA: è evidente che nel primo caso, ogni nuova richiesta ricarichi completamente l'intera struttura del sito web; nel secondo caso, invece, vengono sostituite soltanto le parti necessarie.

questo permette di utilizzare stessi nomi di classe in diversi file `.css`, senza che essi interferiscano tra di loro.

3.7.2 Backend

Il discorso si rende molto più complesso parlando delle soluzioni relative alle possibilità che si hanno lato *backend*. In questo ambito esistono diverse soluzioni possibili ai problemi posti, soprattutto per quanto riguarda la parte dell'organizzazione a microservizi. Anche in questo caso però, essendovi la completa libertà di scelta per quanto riguarda il framework di sviluppo, è stata effettuata una selezione più concentrata a considerare soluzioni già conosciute, sulle quali si è potuto sperimentare nel corso degli anni. Per questo motivo è stato subito individuato un framework comodo, versatile e robusto: **Vert.X** [31].

Esso risulta essere adatto per lo sviluppo di applicazioni reattive sotto la Java Virtual Machine. Il principale vantaggio di questa tecnologia è offrire la possibilità di creare soluzioni scalabili, permettendo un'organizzazione a microservizi e abilitando quasi automaticamente la discovery dinamica di essi. Vert.X offre agevoli interfacce attraverso le quali si può comunicare con diverse architetture (anche esterne dall'ambiete stesso, come ad esempio un database MongoDB). Un altro motivo per cui è indicato puntare su questo framework è la rapidità con la quale si possono creare architetture RESTful [27], in quanto è nativamente supportato il *routing* delle *path*, identificate tramite URL: essenzialmente, ad ogni istanza (chiamata verticle) corrisponde anche un server HTTP il quale può rispondere alle chiamate esterne. Possiede inoltre un supporto nativo per il linguaggio Kotlin [23], nonchè Java, JavaScript, Groovy, Ruby, Python, Scala, Clojure e Ceylon. Se si volesse paragonare il suo funzio-

namento a qualcosa di già esistente, sarebbe molto simile a NodeJS in campo JavaScript.

I dettagli al relativo framework possono essere trovati direttamente sulla documentazione ufficiale [31].

Come **linguaggi**, Java risulta essere quello più utilizzato e supportato dalla piattaforma scelta. Essendo però un linguaggio con diverse criticità (come uno scarso supporto ai tipi opzionali o una debole implementazione dell'immutabilità delle variabili), si preferisce puntare a quella che risulta essere una sua naturale evoluzione: **Kotlin** [23]. Questa scelta stavolta non dipende dal solo background delle conoscenze, ma anche da motivi molto più pratici. Essendo Kotlin un linguaggio in continua evoluzione e con diverse feature mancanti a Java, risulta essere un buon candidato per prototipare applicazioni in modo molto agevole.

3.7.3 Comunicazione e API

In questo ambito, i due modelli più utilizzati e conosciuti fanno riferimento alla serializzazione dei dati a disposizione del client e del server. Infatti, per poterli trasmettere, generalmente vi è necessità di convergere su un unico standard comprensibile da entrambe le parti: di conseguenza, risulta essere obbligatoria la scelta di un formato per codificare le informazioni. I principali due candidati per questa scelta sono **XML** e **JSON**, con una forte preferenza sul secondo per motivi pratici. Infatti, sia lato *frontend* (utilizzante TypeScript), sia lato *backend* (utilizzante Kotlin e Vert.X) il supporto a quest'ultimo è nativo. È indubbiamente un vantaggio enorme che indirizza automaticamente sul supporto di tale formato.

Dopo che si ha a disposizione un formato standard, vi è necessità anche di decidere il modo in cui richiedere e inviare dati. Il più semplice, considerando JSON come partenza, è quello di usare API REST.

Representational State Transfer (REST) rappresenta un sistema di trasmissione stateless sul protocollo HTTP, dove ogni concetto è considerato una risorse identificata tramite un URI. Le risorse possono a loro volta puntare ad ulteriori risorse, formando URI complessi che svolgono ben determinati ruoli in base all'operazione richiesta. Le operazioni permesse sono generalmente 9 (`GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `CONNECT`, `OPTIONS`, `TRACE`, `PATCH`), ma in generale vengono utilizzate soltanto le prime 5. Le suddette risorse possono essere trasmesse ai richiedenti in uno dei formati standard, con una grande maggio-

ranza delle soluzioni disponibili utilizzanti JSON come standard ufficiale[5].

Per ulteriori dettagli si rimanda alla sua documentazione ufficiale [27].

3.7.4 Tecnologia (video)chiamate

Come già ben analizzato nella sezione 3.1.1 e 3.1.4, la tecnologia utilizzata per instaurare delle (video)chiamate sarà WebRTC [32], con la relativa implementazione in PeerJS [26].

3.7.5 DataBase

Considerando il formato JSON come standard di formattazione dati e tenendo presente che Vert.X ha un supporto nativo per il framework di MongoDB, risulta essere molto valida come scelta considerare tale approccio.

MongoDB [24] è un database distribuito basato su *documenti* piuttosto che su tabelle. Memorizza i dati salvati utilizzando esattamente JSON, risultando essere una perfetta soluzione per persistere in modo agevole i dati provenienti direttamente dal *frontend* o *backend*. Avendo poi la facoltà di non dover per forza strutturare i dati, dichiarando tipologie o il loro formato, risulta essere molto comodo per realizzare rapidamente dei prototipi funzionanti, senza preoccuparsi maggiormente dell'organizzazione delle informazioni.
Per maggiori dettagli, si rimanda alla sua documentazione ufficiale [24].

Oltre a MongoDB viene selezionato anche Redis (soltanto per il servizio di autenticazione) per un agevole salvataggio dei token JWT, citato precedentemente nella sezione 3.6.2.

3.8 Design di dettaglio

3.8.1 Strutturazione dei microservizi

Nelle sezioni precedenti è stato analizzato l'insieme delle funzionalità che verranno offerte dal *backend* progettato. In questa sezione, considerando Vert.X come piattaforma di sviluppo, verranno analizzati più in dettaglio i microservizi previsti. La loro organizzazione è stata già schematizzata nel diagramma 3.18.

[5] `https://www.toptal.com/web/json-vs-xml-part-1`

Aspetti comuni

Ogni microservizio sarà caratterizzato da una sua API, corrispondente ad una parte specifica relativa alla progettazione svolta nelle sezioni 3.3.1 e 3.3.2. Per affrontare tale necessità, il microservizio offrirà un punto d'ingresso per tutte le richieste, che verranno gestite ognuna in modo asincrono nel proprio handler. Le API, come gli identificativi, sono stati già ampiamente trattati, per cui non vi è necessità di specificare ogni dettaglio relativo ai singoli design. Si preferisce progettare un'interfaccia comune a tutti i microservizi, in modo da poter essere riutilizzata in ogni aspetto.

Questa interfaccia si compone essenzialmente di due metodi: uno per l'eventuale inizializzazione del servizio, l'altro per impostare gli URL ai quali si dovrà rispondere. Ognuno di questi *path* avrà un proprio handler asincrono, il quale provvederà alla deserializzazione dei parametri in ingresso, all'elaborazione della richiesta e alla relativa formazione della risposta in formato standard, secondo le API definite. Essendo ogni microservizio formato da alcuni aspetti comuni, viene creato un raccoglitore di tali funzionalità: successivamente ognuno di essi "*estenderà*" da questo servizio base per dichiarare le proprie specifiche. Inoltre, come ulteriore approfondimento di questo tema, vi sono servizi interfacciati con i database. Viene prevista la generalizzazione di tale servizio a livello più alto, in modo da far valere lo stesso principio di riuso di una base comune.

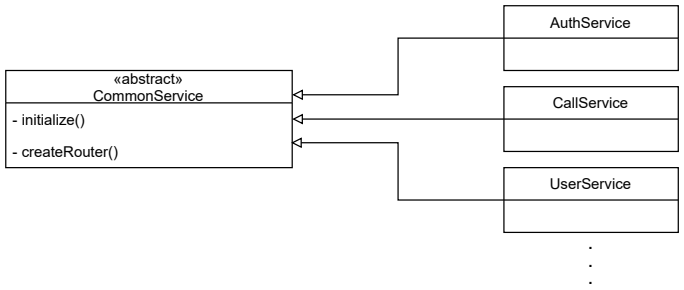


Figura 3.21: Funzionalità comuni verranno raggruppate. Non si riportano tutti i microservizi per semplicità rappresentativa.

Gateway

Il punto d'ingresso per l'applicazione deve prevedere alcuni aspetti fondamentali per il suo intero funzionamento:

- essendo l'unica parte visibile al client, esso deve esporre l'API pubblica in maniera chiara e permettere di ottenere le risorse richieste;

- essendo un sistema a microservizi dinamicamente scopribili, deve poter individuare i servizi che offrono un certo tipo di funzionalità;
- avendo il compito di accogliere le richieste, dev'essere in grado di inoltrarle ai microservizi corretti ed attendere la loro risposta;
- essendo il primo luogo dove le richieste vengono lette, deve garantire che l'utente sia almeno autenticato per determinate operazioni interne, scartando *di default* le richieste non autorizzate;
- il tutto dev'essere garantito senza compromettere l'architettura del sistema e non esponendo alcun servizio al di fuori del *boundary* del *backend*.

Consideranto le API progettate, tutte quante passeranno prima da questo microservizio. Questo aspetto sarà del tutto irrilevante per l'utilizzatore ma è di fondamentale importanza per quanto riguarda l'architettura del *backend*. Nel caso di malfunzionamenti o di un carico elevato, si possono predisporre più nodi di smistamento, in modo da garantire un'agevole fruizione del servizio.

Registry Service

Microservizio che garantisce il riferimento ad ulteriori microservizi per permettere una loro individuazione all'interno del sistema. È necessario che ogni servizio, alla propria creazione, si sottoscriva a quest'ultimo, perché soltanto in questo modo si renderà disponibile a ricevere eventuali chiamate. Oltre all'accumulo delle varie risorse, il "*registro*" si occupa di monitorare lo stato dei servizi e garantire che all'interno del proprio archivio contenga soltanto riferimenti a microservizi operativi e funzionanti. Per quest'ultima fase è necessario stabilire un criterio con il quale selezionare servizi non attivi:

- si definisce un tempo di 30 secondi, ogni quali verrà effettuata un'analisi sui servizi attualmente registrati;
- si definisce un tempo di timeout di 2 secondi, dopo i quali verrà incrementato un conteggio di fallimenti per quel servizio;
- si definisce un massimo di 2 tentativi falliti per il servizio, prima di essere completamente rimosso.

User Service

Microservizio che mantiene il riferimento agli utenti attualmente presenti nel sistema. Permette, generalmente, di gestire tutti gli aspetti riguardanti un utente, come la modifica dei suoi dati: possiede un'API propria per poter

gestire questi aspetti. Inoltre, trattandosi di un servizio collegato al mondo delle chiamate, esso provvede utilità per poter consultare e ricevere inviti ai collegamenti creati da altri utenti.

Una precisazione riguardante questo servizio è la sua progettazione a priori: esso non era uno degli obiettivi posti dai requisiti, ma la gestione degli utenti è intrinsecamente collegata al sistema progettato: per questo motivo è opportuno provvedere sin da subito ad una tale gestione. L'architettura a microservizi garantisce che questo possa essere modificato ed esteso, senza intaccare le altre parti già funzionanti: basterà eventualmente aggiornare opportunamente le API.

Si sottolinea che per lo sviluppo del prototipo verranno predisposti alcuni utenti *default* all'intero del sistema, per poterlo testare senza necessità di progettare una completa architettura per la loro registrazione.

Auth Service

Microservizio responsabile all'autenticazione e registrazione degli utenti che voglio interagire con il sistema. È considerato un modulo a sé stante in quanto esistono diverse metodologie di autenticazione: nel progetto trattato, per semplicità di utilizzo, si è preferito puntare ad un'autenticazione basata sul token JWT, mentre in un caso reale potrebbe essere necessario adeguarsi ad una metodologia già usata in altri servizi offerti dallo stesso ente. Isolando completamente un servizio di tale importanza si rende la soluzione più propensa ad essere modificata soltanto nelle parti in cui è essenziale farlo, senza intaccare il codice già funzionante di altri microservizi. Si sottolinea quindi ancora una volta l'importanza di strutturare in maniera indipendente ogni modulo a disposizione.

Call Service

Modulo *core* della soluzione, è responsabile di tutta la gestione delle stanze, dei relativi partecipanti ed inviti. Si occupa di garantire un unico punto di riferimento per quanto riguarda la vera gestione delle chiamate, sfruttando la dipendenza imposta dall'uso di PeerJS come architettura sulla quale vengono costruiti i collegamenti tra gli utenti.

Una nota positiva per quanto riguarda la strutturazione di tale servizio è la possibilità di renderlo completamente isolabile, aggiornando soltanto le parti

nelle quali vi sia qualche dipendenza da ulteriori servizi (come quello di utenti). In questo modo si offre la possibilità di riutilizzare completamente le API previste per il solo modulo delle chiamate, isolando completamente la gestione degli utenti e/o autenticazione in altri microservizi già trattati in precedenza.

L'unico punto su cui bisogna porre maggiore attenzione è l'aspetto riguardante proprio la dipendenza della libreria scelta. PeerJS impone un modo di funzionare che risulta essere compatibile con le dichiarazioni originali del WebRTC: il passaggio verso framework offerenti diversi approcci potrebbe di conseguenza risultare non del tutto banale. Avendo però studiato a fondo il funzionamento delle stanze, le quali permettono l'instaurazione di una chiamata tra utenti, dev'essere possibile ottenere lo stesso approccio anche utilizzando diversi framework: questo grazie ad una buona progettazione dell'architettura di tale servizio, che di conseguenza copre in maniera esaustiva gli scenari previsti dai requisiti.

3.8.2 Frontend

La strutturazione del *frontend* rispecchia il principio di modularità affrontato durante la progettazione del *backend*, in quanto vengono individuati aspetti fondamentali per coprire tutti i requisiti richiesti. Vi sono infatti tre principali moduli responsabili al corretto funzionamento di questa parte dell'architettura: comunicazione col *backend*, gestione degli aspetti della chiamata (utilizzando PeerJS) e interfaccia grafica.

Comunicazione col Backend

La strutturazione a microservizi potra automaticamente ad una suddivisione dei ruoli che devono offrire le varie classi progettate, seguendo naturalmente gli aspetti che caratterizzano tali funzionalità. Per ognuno dei microservizi, infatti, viene prevista una sua classe di gestione, in modo da isolare le API e renderle più mantenibili nel caso di modifiche. Avendo a che fare con API intrinsecamente asincrone, è previsto l'utilizzo di librerie che permettano tale comunicazione. I metodi e i parametri da loro richiesti rispecchieranno pienamente le specifiche delle API progettate, in modo da facilitare la loro futura integrazione nel codice.

Avendo trattato in maniera dettagliata tutte le API nelle sezioni precedenti, risulta essere superfluo trattare in dettaglio la progettazione di ogni singola classe: si preferisce definire un modello che verrà seguito da tutte. In questo

modo è possibile definire un pattern da seguire ogni qual volta che si richiede la progettazione di tale funzionalità:

- Il nome della classe rispecchierà quello del servizio richiesto: ad esempio, AuthService (microservizio del *backend*) avrà il suo corrispettivo `auth.service` (classe nel *frontend*).
- La classe così formata punterà a tutte le risorse di quel determinato servizio: ad esempio, `auth.service` (classe nel *frontend*) avrà modi per accedere a tutte le API fornite dalla sua controparte del *backend*, mappando la chiamata REST (es. `/login`) in una funzione con lo stesso nome (es. `login()`).
- I metodi definiti dalla classe per accedere al servizio saranno uguali a quelli utilizzati dalle API ufficiali: ad esempio, `auth.service` (classe nel *frontend*) si comporrà di tre metodi, quali `login`, `logout` e `register`.
- I metodi avranno gli stessi parametri in input e output delle API definite: ad esempio, il metodo `login` richiederà lo username dell'utente e la sua password e ritornerà un *HTTP Response Status Code* in caso di risposta positiva, oppure un "*RestError*" nel caso di insuccesso.
- Il modo nel quale le risorse vengono richieste è delegato a tale classe ed è completamente trasparente. L'unica osservazione è relativa al fatto della natura asincrona degli eventi, per cui il tipo di ritorno consisterà in un oggetto osservabile, del quale si gestiranno gli aspetti una volta ottenuta una risposta.

Avendo deciso di utilizzare Angular come framework di sviluppo, si può già progettare di utilizzare il client HTTP direttamente incluso nel pacchetto per le chiamate remote. Inoltre, per alcune funzionalità avanzate, è possibile far riferimento alla libreria "*RxJS*", offrente funzionalità nella programmazione asincrona e basata sugli eventi.Grazie alla scelta del framework è anche possibile prevedere che i "*Servizi Angular*" progettati saranno tutti disposti in un unica cartella. Ad ognuno dei servizi così progettati corrisponderanno anche eventuali classi di appoggio, rappresentanti gli oggetti delle API trattate nelle sezioni 3.3.1 e 3.3.2.

Gestione della chiamata

Come anticipato più volte, la gestione della chiamata relativa all'uso di PeerJS risulta essere P2P: ogni utente è responsabile di avere due flussi (uno in entrata e uno in uscita) per ognuno dei membri partecipanti al collegamento. È necessario quindi predisporre una struttura dati, la quale permetterà

un'agevole implementazione e astrazione dei concetti, in modo da nascondere il più possibile la scelta di tale libreria.

Anche se i rimanenti moduli trattano in maniera triviale gli aspetti comunicativi *frontend-backend*, il componente relativo alle chiamate risulta essere più complesso, abbracciando la natura P2P. Nonostante la maggior parte dei metodi rispecchi effettivamente le chiamate da eseguire sul *backend* definite nella sezione 3.8.2 (come ad esempio `getRooms()` per la `GET` su `/rooms`), vi sono alcuni metodi più particolari che richiedono uno studio più approfondito. È infatti il caso di `joinRoom`, nel quale vi è una diretta interazione con il servizio di PeerJS, completamente gestito nella parte del *frontend*.

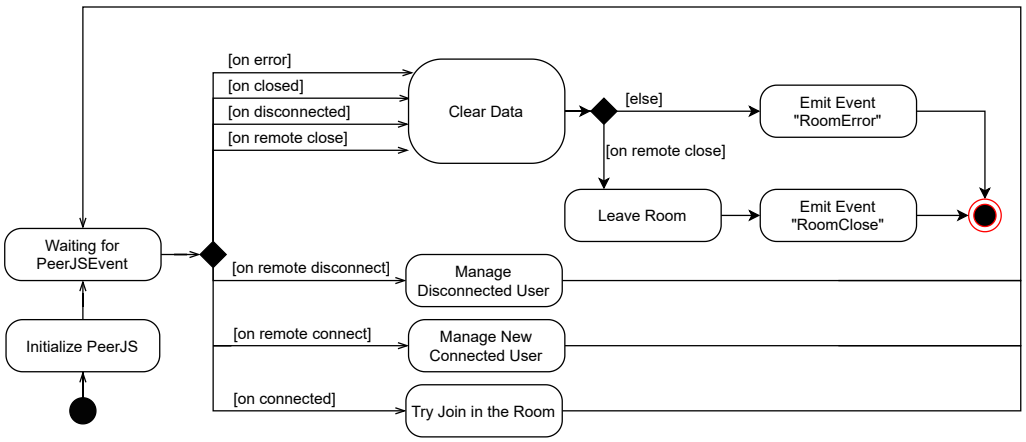


Figura 3.22: Eventi ad alto livello che possono capitare durante il ciclo di vita del servizio per le "*room*".

La figura 3.22 rappresenta in linea generale gli stadi che si vogliono gestire durante la chiamata: un qualsiasi errore di PeerJS ne provoca l'immediata chiusura, gli eventi degli utenti remoti vengono propriamente gestiti e quando il servizio di PeerJS risulta essere pronto viene effettuata la procedura per l'effettiva connessione alla stanza. Il diagramma 3.23 dimostra il flusso per eseguire tale richiesta, sottolineando quali azioni sfruttano le API del *backend* e quali le funzionalità della libreria.

A questo punto è necessario analizzare il servizio che permette di interfacciarsi con PeerJS. Come specificato dalla documentazione ufficiale [26], sono necessarie soltanto alcune righe di codice per instaurare la chiamata tra due partecipanti. Il problema principale della soluzione in analisi è che sono previste chiamate a più membri, complicando notevolmente la strutturazione di tale architettura. Per far fronte a tutti i requisiti si pone una particolare atten-

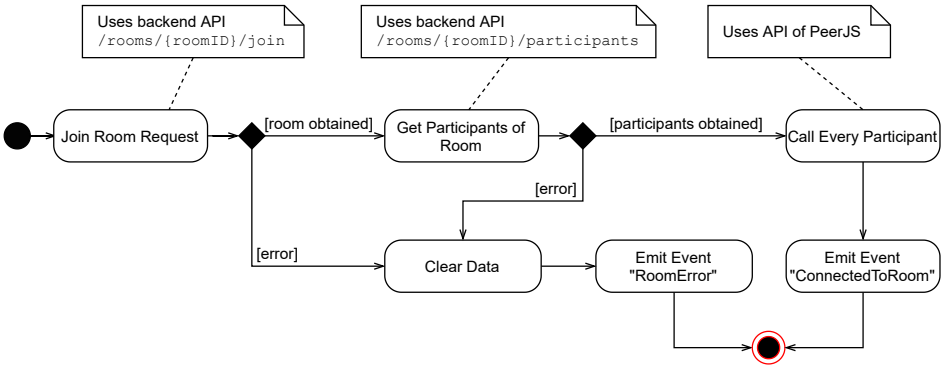


Figura 3.23: Flusso necessario per effettuare una "*join*" dentro una stanza.

zione all'intrinseca asincronia di tali servizi, gestendo il modello ad eventi. Per sottostare a tale scopo, si progetta un diagramma di attività rappresentante il ciclo di vita del servizio di PeerJS implementato.

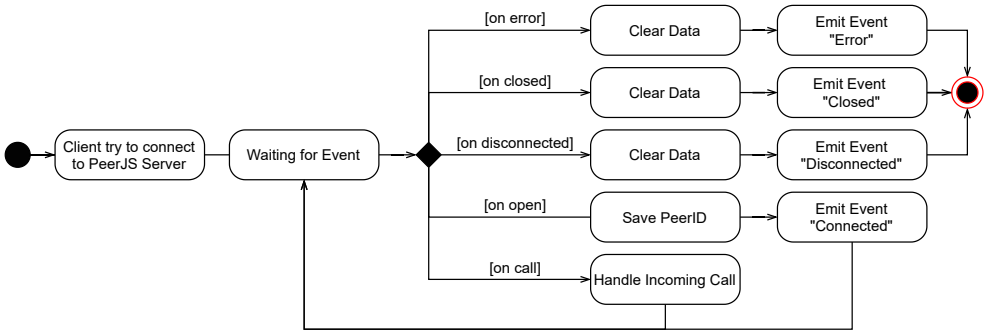


Figura 3.24: Eventi ad alto livello che possono capitare all'interno del ciclo di vita di PeerJS.

Come si può vedere dalla figura 3.24, il ciclo di vita risulta essere concluso alla disconnessione, chiusura o se vi sono errori gravi nella comunicazione. Per gli altri due casi rimanenti, il handler degli eventi di PeerJS si rimette in ascolto. Per poter permettere un'agevole interazione con tale servizio si predispone un emettitore di eventi, il quale rispecchierà in maniera fedele ciò che succede a basso livello, in modo da poter astrarre dall'implementazione effettiva e riuscire comunque a reagire.

La parte relativa all'inizio del ciclo di vita è rappresentata dal ramo condizionale `[on open]`. Infatti nel diagramma 3.22 è possibile notare che la procedura per la connessione alla stanza viene inizializzata soltanto dopo aver ricevuto l'evento `Connected` da parte di PeerJS. Successivamente, oltre agli eventi che potrebbero capitare durante tutto il ciclo di vita, vi sono metodi

di utilità che permettono di interagire con il sistema sottostante, in maniera tale da garantire la corretta instaurazione delle connessioni P2P. In particolare, l'azione relativa al chiamare tutti i partecipanti è svolta attraverso l'uso di una funzione che raggruppa le minuzie di PeerJS sotto un'unica chiamata.

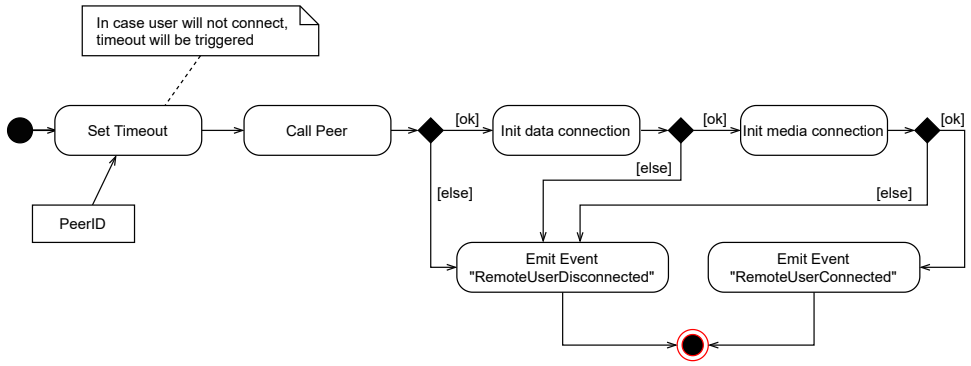


Figura 3.25: Flusso di eventi per chiamare un determinato partecipante.

Per alcuni aspetti riguardanti questi schemi si possono approfondire alcune questioni in merito alla gestione delle risorse:

- Il servizio di PeerJS si occupa di mantenere i riferimenti a tutti i partecipanti, associando ad ogni PeerID in ingresso una struttura dati comprendente:
  - il flusso audio/video (media) del chiamato,
  - il flusso dati del chiamato,
  - gli eventi che capitano nella connessione tra il chiamante e il chiamato,
  - gli eventi che capitano nella connessione audio/video tra il chiamante e il chiamato,
  - gli eventi che capitano nella connessione dati tra il chiamante e il chiamato.
- Ad ogni nuovo utente connesso viene istanziata la struttura dati del punto soprastante, popolando correttamente tutti i campi. Soltanto nel momento in cui tutti quanti risultano essere corretti l'utente è ritenuto "*connesso*" alla stanza.
- Per ognuno degli utenti connessi sono di conseguenza aperti molteplici listener: si sottolinea ancora una volta che tale soluzione potrebbe portare ad un utilizzo eccessivo di risorse e risultare insostenibile per chiamate con diversi partecipanti.

- I messaggi vengono scambiati grazie alla connessione dati: non si tratta soltanto dei comuni messaggi "*chat*", ma anche informazioni di servizio. Essendo presente soltanto un canale di comunicazione, vi è necessità di differenziare tali messaggi, predisponendo alcune categorie fondamentali:
  - `chat` per i messaggi testuali semplici,
  - `close` per l'evento di chiusura stanza,
  - `error` per un qualsiasi evento di errore che può capitare.

UX

I requisiti specificano chiaramente che la parte della User Experience non debba essere curata nei minimi dettagli: questo è dovuto al fatto di concentrare maggiori risorse sull'analisi delle possibilità del modulo delle chiamate e della sua futura adattabilità piuttosto che alla cura dell'interfaccia grafica. Ciononostante, si preferisce analizzare tale aspetto in modo da proporre un'architettura *frontend* capace di seguire l'esempio del *backend*, rispecchiando i principi di modularità e manutenibilità.

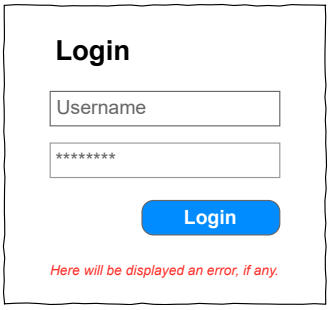
Vi sono alcune interazioni che possono intercorrere durante il ciclo di vita di una sessione dell'utente, che definiscono le schermate da progettare:

- **Login**: verrà predisposto un "*LoginComponent*" per isolare la parte relativa all'autenticazione.
- **Home**: verrà predisposto un "*HomeComponent*" per poter:
  - visualizzare stanze pubbliche attive,
  - creare una stanza (e gestirne gli inviti/permessi),
  - unirsi ad una stanza (ad inviti o pubblica).
- **Room**: verrà predisposto un "*RoomComponent*" per poter:
  - interagire coi flussi audio/video utenti
  - gestire la stanza (considerando i relativi permessi),

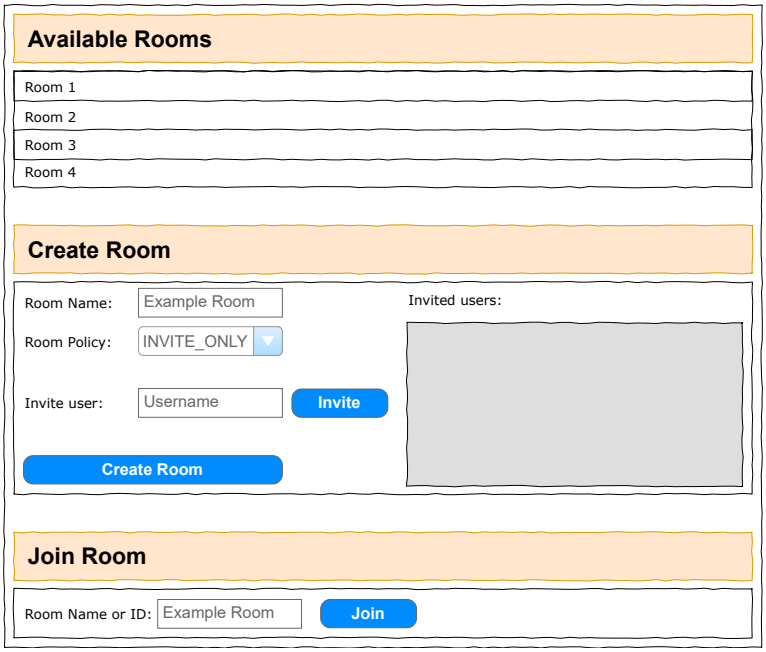
Grazie a questa analisi è possibile individuare ulteriori moduli, incapsulanti alcune delle macro-funzionalità:

- **RoomManagementComponent**: sarà il componente che permetterà la specifica dei parametri di una stanza durante la sua creazione e mentre questa risulta essere già attiva.
- **VideoContainerComponent**: si occuperà della visualizzazione dei flussi audio/video, in modo da non caricare il "*componente padre*" (*RoomComponent*) di tutta la logica relativa a tale gestione.

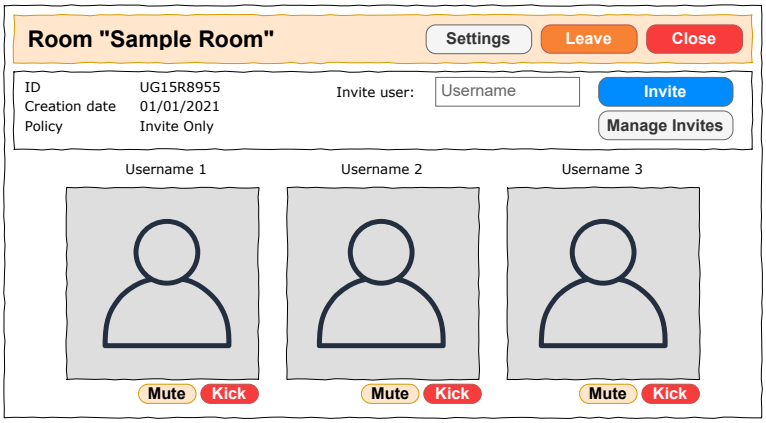
Per quanto riguarda i relativi mockup, si possono consultare le immagini disponibili in figura 3.26.



(a) Schermata Login.



(b) Schermata Home.



(c) Schermata Room.

Figura 3.26: Mockup del *frontend*.

Capitolo 4

Progettazione Sistema Televisita

In questo capitolo, prima di procedere all'effettiva progettazione di aspetti più architetturali, è necessario analizzare in quale maniera i requisiti posti possano influenzare una futura realizzazione del Sistema di Televisita. Questo si rende necessario a fronte del requisito di modularità, provvedendo ad una strutturazione dei vari componenti del servizio, considerando sin da subito tutti gli aspetti che potrebbero caratterizzarlo in maniera non banale. Successivamente allo svolgimento di approfondimenti sui requisiti implementativi, si svolgerà l'effettiva progettazione del servizio, della sua architettura e delle API offerte, analizzando in dettaglio le sue possibilità e criticità.

4.1 Vincoli implementativi

Oltre alla definizione del modello del dominio, durante la stesura dei requisiti è stato posto un vincolo per la futura implementazione del progetto, in quanto è necessario utilizzare il servizio "*Call Service*" per l'effettuazione di chiamate. Tale requisito non pone però particolari problematiche. Essendo progettato sin dall'inizio con l'idea di dover garantire una trasparenza nel suo utilizzo, non è necessario approfondire alcun dettaglio implementativo: questa parte, infatti, sarà isolata dal resto della soluzione in modo tale da poter essere riusata anche in contesti diversi. Non vi è dunque motivo per considerare architetturalmente questa problematica, in quanto il "*Call Service*" sarà intrinsecamente presente come servizio staccabile e separato dal resto: infatti, offrirà un'interfaccia universale permettente un'agevole comunicazione con esso.

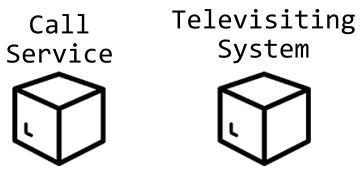


Figura 4.1: Le parti sono viste come delle "*blackbox*": la comunicazione avverrà tramite interfacce ben definite che non influenzano i rispettivi sistemi in alcun modo.

4.2 Strutturazione dati

Prima di trattare in via generica delle API fornite dal servizio, è necessario definire un modello di dati valido affinché le informazioni presenti all'interno del sistema siano univoche e corrispondenti all'interfaccia esposta. Come base si utilizza il modello del dominio esplicitato in figura 2.10, composto durante l'analisi dei requisiti.

La figura 4.2 mostra la composizione di tali classi, esplicitando maggiormente le relazioni che intercorrono tra di esse. È doveroso sottolineare che tale struttura verrà utilizzata sia lato *backend* che *frontend*, in quanto entrambi necessitano di utilizzare gli stessi dati. Come specificato dall'analisi dello standard FHIR (sezione 1.3), il seguente modello non mira a coprire in maniera esaustiva tutti gli scenari previsti dallo standard, ma, pur rimanendo completamente compatibile con future estensioni, si concentra perlopiù sul soddisfacimento dei requisiti posti. A differenza del "*Call Service*", in questo caso la conoscenza specifica di un dominio (basato essenzialmente sullo standard FHIR) risulta essere necessaria affinché si possa garantire l'interoperabilità richiesta. Inoltre, partendo dal presupposto che "*User Service*" offre un'unica interfaccia per entrambi i servizi, è necessario assicurarsi che l'utente venga riconosciuto da ambedue i progetti.

Nonostante lo schema 4.2 cerchi di raffigurare in maniera esaustiva tutti i concetti relativi ai dati e alle associazioni tra essi, vi sono alcune cruciali osservazioni da affrontare:

1. Il concetto che lega l'utente all'appuntamento (*manage*) è molto generico. Questo è dovuto alla semplicità di rappresentazione, in quanto l'analisi effettuata per differenziare i ruoli degli utenti fa parte dell'analisi del dominio già svolta. Infatti, si può far riferimento alle figure 2.11 e 2.12 per apprezzare la completezza delle relazioni che intercorrono tra le entità. Lo stesso principio vale per il discorso di *check* tra l'utente e l'entità "*schedule*".

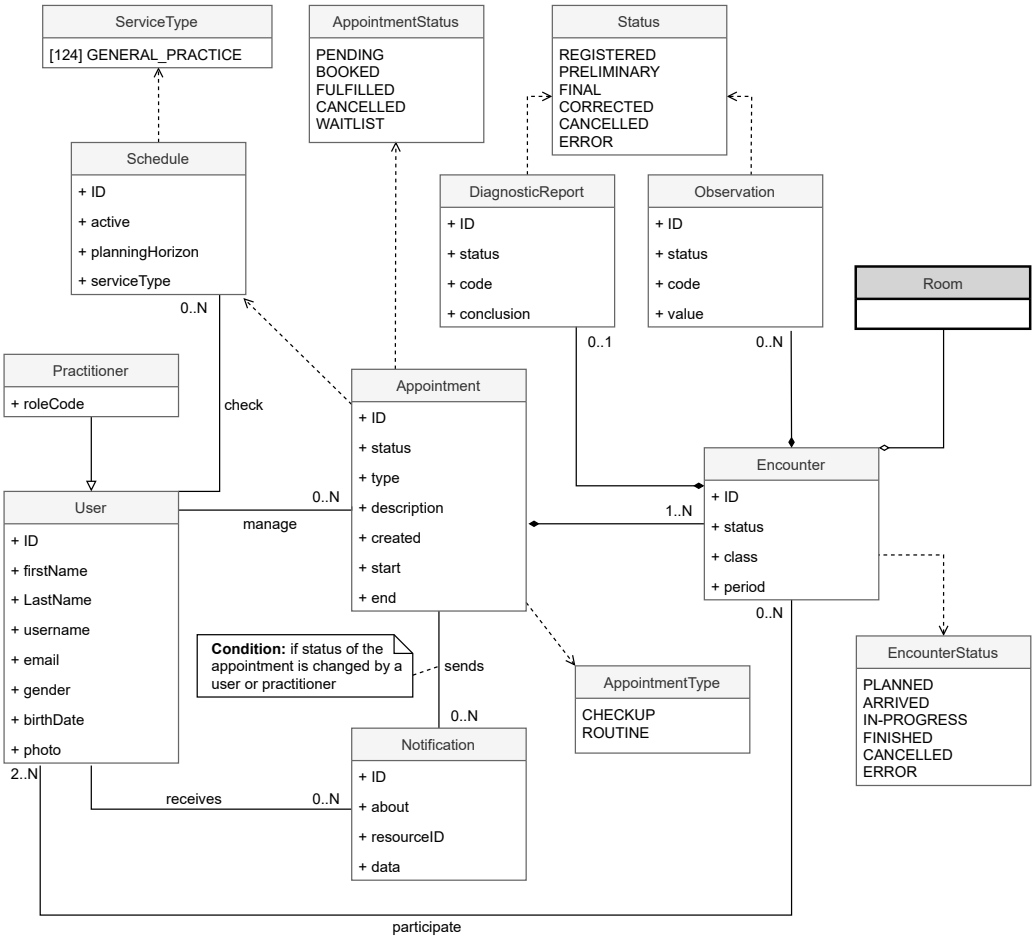


Figura 4.2: Classi, attributi e relazioni del Sistema di Televisita.

2. Guardando il modello del dominio vi sono alcune differenze con la figura proposta: questo non è dovuto al cambiamento del domino stesso, ma all'esplorazione più approfondita dei concetti e degli attributi necessari a realizzare il quadro completo della situazione. Alcune classi sono semplicemente richieste dal modello, altre utilizzate per poter coprire completamente i requisiti. I riferimenti più dettagliati vanno alle seguenti entità:

    - `ServiceType` ha soltanto un'istanza, in quanto l'unica tipologia di prenotazioni tratterà di generici incontri con dottori/pazienti/operatori. Essendo però una struttura flessibile, sarà possibile definire in futuro diversi tipi di prenotazioni per differenti servizi.

- `EncounterStatus`, `AppointmentStatus` e `Status` sono tutte figure che non riportano tutti i relativi dettagli riguardanti il modello FHIR. È infatti obbligatorio specificare uno stato, ma non è necessario definire quali degli stati verranno utilizzati.
- `AppointmentType` ha soltanto due tipi, in quanto l'appuntamento sarà differenziato da questi due attributi. Nel caso di "*CHECKUP*" è l'operatore ad instaurare la chiamata, mentre nel caso di "*ROUTINE*" è il medico. La conseguenza è relativa al fatto di poter concludere gli incontri con dei referti nel caso di appuntamenti di tipo "*ROUTINE*".
- Il valore `code` all'interno di "*DiagnosticReport*" e "*Observation*" si riferisce ad una specifica indicazione medica per l'identificazione del concetto affrontato. Tale aspetto risulta essere prettamente appartenente al dominio medico ed è difficile prevedere l'insieme di valori che potrebbe assumere. Di conseguenza, viene fatto riferimento a tutto l'elenco possibile.
- Il valore `class` all'interno di "*Encounter*" può essere soltanto di un tipo nel contesto analizzato ("*VR*"), in quanto specifica che tale incontro è stato svolto interamente online.
- Il valore `about` e `resourceID` all'interno di "*Notifications*" assumono i valori di classe di appartenenza di tale notifica (ad esempio "*Appointment*" o "*Encounter*") e l'indirizzo che punta verso tale risorsa. Eventuale campo opzionale `data` permette di specificare ulteriori dettagli relativi a quella particolare notifica.

3. Nella maggior parte delle entità mancano attributi considerati da FHIR: essendo un modulo trattante esclusivamente di Televisite, sarebbe stato del tutto inutile coprire interamente il quadro offerto dallo standard. Si preferisce infatti minimizzare l'effort dovuto all'adozione di tale struttura, considerando soltanto aspetti strettamente necessari alla rappresentazione del dominio. Stessa logica vale anche per eventuali *enum* incontrati, in quanto non è prevista la modellazione completa di tutti i casi.

4. L'entità "*Room*" non possiede alcun attributo perchè fa parte della modellazione del "*Call Service*": serve soltanto per esprimere la connessione tra il contesto delle Televisite e la parte delle chiamate. Questa contrapposizione vuole sottolineare quanto questo modulo sia importante per la realizzazione del "*Televisiting System*", grazie anche alla sua modularità e quanto la realtà del "*Call Service*" diventi minima di fronte ad un contesto ben più specifico.

5. Essendo i requisiti non trattanti specificatamente di ogni singolo particolare relativo all'ambito di una visita medica, è difficile supporre quali attributi siano necessari: si suppone di conseguenza che il modello debba trattare soltanto degli aspetti previsti dagli scenari, con eventuali future possibilità di espansione.

6. Per i dettagli sullo standard FHIR e sugli attributi previsti si rimanda alle singole definizioni già trattate nella sezione 1.3. Per semplicità di lettura si riportano di seguito in ordine alfabetico tutti i concetti utilizzati in tale schema come elenco consultabile di link alle risorse.

    - Appointment:
      `https://www.hl7.org/fhir/appointment.html`
    - Appontment Status:
      `https://www.hl7.org/fhir/valueset-appointmentstatus.html`
    - Appoitment Type:
      `https://www.hl7.org/fhir/v2/0276/index.html`
    - Diagnostic Report:
      `https://www.hl7.org/fhir/diagnosticreport.html`
    - Diagnostic Report Code:
      `https://www.hl7.org/fhir/valueset-report-codes.html`
    - Diagnostic Report Status:
      `https://www.hl7.org/fhir/valueset-diagnostic-report-status.html`
    - Encounter:
      `https://www.hl7.org/fhir/encounter.html`
    - Encounter Class:
      `https://www.hl7.org/fhir/v3/ActEncounterCode/vs.html`
    - Encounter Status:
      `https://www.hl7.org/fhir/valueset-encounter-status.html`
    - Observation:
      `https://www.hl7.org/fhir/observation.html`

- Observation Code:
  `https://www.hl7.org/fhir/valueset-observation-codes.html`
- Observation Status:
  `https://www.hl7.org/fhir/valueset-observation-status.html`
- Patient:
  `http://www.hl7.org/fhir/patient.html`
- Practitioner:
  `https://www.hl7.org/fhir/practitioner.html`
- Practitioner Role Code:
  `https://www.hl7.org/fhir/valueset-practitioner-role.html`
- Schedule:
  `https://www.hl7.org/fhir/schedule.html`

4.3 API

Le API vengono progettate nel pieno rispetto dei requisiti posti. Lo schema in figura 4.3 mostra in maniera sintetica le API previste per la gestione del servizio di Televisita. Per semplicità non vengono mostrati gli attributi in ingresso e in uscita, che verranno trattati in maniera approfondita successivamente in questa sezione. Inoltre, si intendono utilizzate le risorse del diagramma 4.2, con riferimenti alle API dell'utente dello schema 3.11.

Dopo la dichiarazione del modello generale in figura 4.3, analogamente al "*Call Service*" si riportano in figure 4.4, 4.5, 4.6 le API in termini di parametri richiesti in input e quelli restituiti in output.

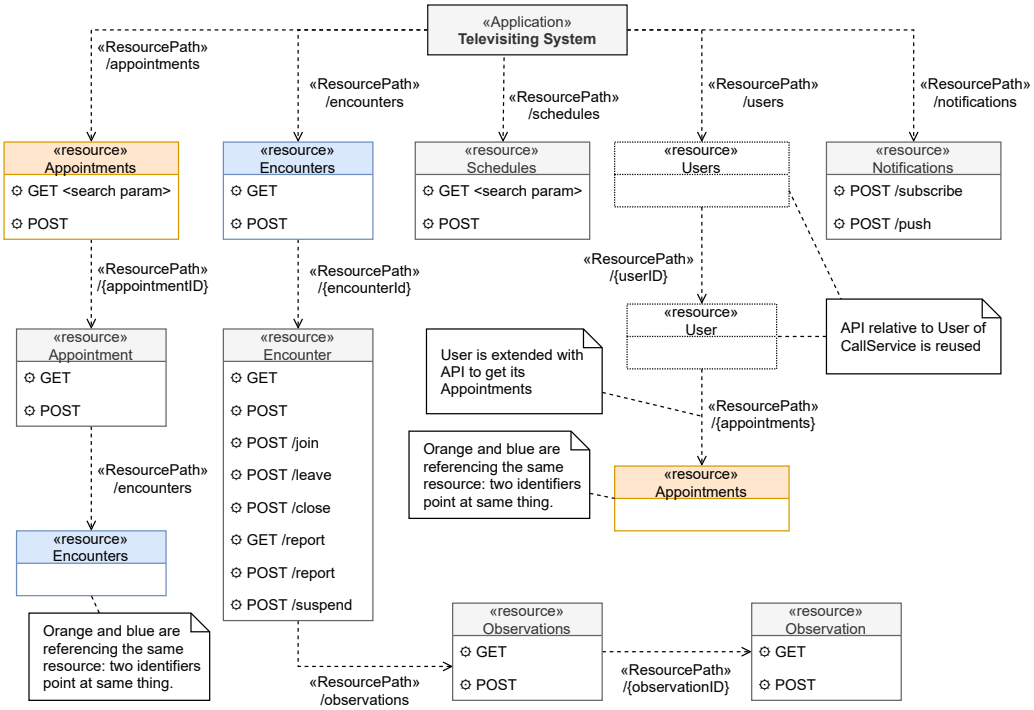


Figura 4.3: API del Sistema di Televisita.

/schedules?<search param>		
GET	*Retrive schedules, given a filter. Filter can specify dates, daily, weekly or monthly*	
	Input	-
	Output	*JSON object representing a collection of free Schedules* `schedules: Schedules[]`
POST	*Change status of that Schedule. Operation Type could be "occupy" or "free".*	
	Input	*JSON object representing a Occupy Schedule Request.* `startDateTime: DateTime, endDateTime: DateTime, operation: OperationType`
	Output	`HTTP Response Status`

/appointments?<search param>		
GET	*Retrive all appointments for that specific user. Can specify a filter (date, subject etc...)*	
	Input	-
	Output	*JSON object representing an Appintment.* `appointment: Appointment`
POST	*Creates a new appointment. Depending of the role of user, may produce different results. Default ServiceType is "General Practice". It takes automatically one of reserved schedules.*	
	Input	*JSON object representing a Create Appointment Request.* `type: ServiceType`
	Output	*JSON object represeting a Join Room Response.* `appointmentID: string`

/appointments/{appointmentID} /users/{userID}/appointments/{appointmentID}}		
GET	*Retrive details about that appointment. Eventually, apply the given search param.*	
	Input	-
	Output	*JSON object representing an Appintment.* `appointment: Appointment`
POST	*Update that specific appointment (delete doesn't mean it's deleted)*	
	Input	*JSON object representing an updated Appointment* `appointment: Appointment`
	Output	`HTTP Response Status`

Figura 4.4: API Appointment(s) e Schedules.

/appointments/{appointmentID}/encounters /users/{userID}/appointments/{appointmentID}/encounters		
GET	*Retrive all encounters for that specific appointment*	
	Input	-
	Output	*JSON object representing a collection of Encounters* `encounters: Encounter[]`
POST	*Create a new Encounter on that specific appointment. Default EncounterType is "VR"*	
	Input	-
	Output	*JSON object representing a Create Encounter Response.* `encounterID: string`

/appointments/{appointmentID}/encounters/{encounterID} /users/{userID}/appointments/{appointmentID}/encounters/{encounterID}		
GET	*Retrive a specific encounter for that specific appointment.*	
	Input	-
	Output	*JSON object representing an Encounter* `encounter: Encounter`
POST	*Update a specific encounter*	
	Input	*JSON object representing an Encounter* `encounter: Encounter`
	Output	`HTTP Response Status`
POST	*/join – join on that specific encounter. Joining of a medic/operator resumes an Encounter.*	
	Input	-
	Output	*JSON object represeting a Join Room Response.* `room: Room`
POST	*/leave – leave the encounter*	
	Input	-
	Output	`HTTP Response Status`
POST	*/close – close and finish the encounter*	
	Input	-
	Output	`HTTP Response Status`
GET	*/report – Get the Diagnostic Report of that Encounter*	
	Input	-
	Output	*JSON object represeting a DiagnosticReport.* `diagnosticReport: DiagnosticReport`
POST	*/report – Create or update a DiagnosticReport*	
	Input	*JSON object represeting a DiagnosticReport.* `diagnosticReport: DiagnosticReport`
	Output	`HTTP Response Status`
POST	*/suspend – suspend current Encounter*	
	Input	-
	Output	`HTTP Response Status`

Figura 4.5: API Encounter(s).

…/{encounterID}/observations	
GET	*Retrive all observations of that ecounter*
	Input: -
	Output: *JSON object representing a collection of Observations.* `observations: Observation[]`
POST	*Add a new observation for that specific encounter*
	Input: *JSON object representing an Observation.* `observation: Observation`
	Output: `HTTP Response Status`

…/{encounterID}/observations/{observationID}	
GET	*Retrive that specific observation of that encounter*
	Input: -
	Output: *JSON object representing an Observation* `observation: Observation`
POST	*Update that specific observation*
	Input: *JSON object representing an Observation.* `observation: Observation`
	Output: `HTTP Response Status`

/notifications	
POST	*/push – Push a notification. Internal use only.*
	Input: *JSON object representing a Notification Push Request.* `fromService: string, about: ResourceType, title: string, message: string, toUser: string`
	Output: `HTTP Response Status`
POST	*/subscribe – Enable sending of notifications about a specific ResourceType.*
	Input: *JSON object representing a Notification Request.* `about: ResourceType`
	Output: *JSON object representing a resource where to point at in order to receive notifications.* `resource: Resource`

Figura 4.6: API Observation(s) e Notifications.

4.4 Standardizzazione

Oltre a FHIR, che garantisce l'interoperabilità del sistema, vi è possibilità di referenziare un ulteriore standard. Infatti, nella sezione 3.4 relativo alla standardizzazione del "*Call Service*" si è parlato di un modo comodo per gestire errori che potrebbero capitare sul *backend*. La stessa struttura può essere replicata ed estesa anche al "*Televisiting System*", introducendo semplicemente nuovi termini attraverso i quali gli errori potrebbero venire identificati. Per i relativi dettagli si rimanda alla sezione 3.4, mentre di seguito vengono soltanto elencate le risorse che estenderanno la classe di `RestError`, in termini di concetti dove vi possono capitare degli errori:

- Appointment;
- Encounter;
- Schedule;
- Notification.

4.5 Architettura

Il servizio legato alle Televisite risulta essere una parte distinta del progetto ma che al suo interno utilizza il "*Call Service*". Volendo contrapporre entrambe le parti in ogni capitolo, la progettazione della parte telemedica prenderà spunto da quella originalmente proposta per il primo servizio. Avendo a disposizione un'architettura a microservizi sul *backend* e una composizione a componenti e moduli sul lato *frontend*, è possibile immaginarsi una strutturazione del tutto analoga alla figura 3.18. Vi sono però da considerare alcuni aspetti fondamentali di tale architettura, come la presenza concetti che estendono il significato di "*utente*", aspetti come "*incontri*" e "*prenotazioni*" e tutta la questione relativa all'utilizzo dello standard FHIR.

Attraverso la già scelta strutturazione a microservizi nella progettazione del "*Call Service*", si è potuta apprezzare la facilità con la quale i vari servizi possono essere aggiunti e modellati. Essendo tale architettura predisposta ad avere più microservizi al suo interno, è possibile riutilizzare tutta la struttura prevista per il primo servizio progettato e implementare soltanto le parti necessarie per coprire le funzionalità relative alle Televisite. Non mancano però alcune criticità da discutere:

- **Autenticazione e Utenti**: nel "*Call Service*" vi sono due microservizi riguardanti tali aspetti, che necessariamente sono rispecchiati anche nel

"*Televisiting System*". Essendo tali, è possibile prevedere una semplice estensione di "*User Service*" in modo da aggiungere API atte ad ottenere i dettagli più relativi al dominio specifico. Tale "*estensione*" non intacca in nessun modo il servizio base, che continuerà a funzionare con il "*Call Service*", e permette di riusare un modulo già progettato in modo da adottare soluzioni il più possibilmente coincise con i requisiti.

- **Registrazione microservizi**: nel "*Call Service*" è già presente un microservizio capace di permettere la discovery dinamica dei sottosistemi. Non è dunque necessario provvedere alla replicazione di tale servizio, in quanto tutti i moduli componenti la soluzione completa verranno inglobati dietro un'unica interfaccia API.

Per quanto riguarda ulteriori microservizi previsti, si vuole sottolineare come gli appuntamenti, incontri e tutte le fasi relative alla misurazione e stesura referti sono strettamente collegati tra di loro: non vi sono, infatti, "*Appoitments*" senza un "*Encounter*", ed è raro avere un "*Encounter*" senza nessuna "*Observation*" o "*DiagnosticReport*". In vista dei molteplici aspetti coperti da FHIR e prevedendo una possibile espansione di tale architettura, si preferisce adottare ancora una volta un approccio completamente modulare, separando i macro-concetti in più microservizi. Questi ultimi, considerando tutte le precedenti osservazioni, verranno disposti come segue:

- **Registry e Auth Service**: servizi riutilizzati dal modulo di "*Call Service*" così come sono stati originalmente progettati.
- **User Service**: viene espanso con la parte relativa agli aspetti strettamente necessari al contesto di Televisita nel quale viene calato il "*Televisiting System*".
- **Appointment Service**: servizio che possiede le capacità di gestire appuntamenti.
- **Encounter Service**: servizio che possiede le capacità di gestire incontri. Si rende opportuno prevedere questo microservizio, in quanto la complessità del singolo appuntamento e del singolo incontro potrebbero man mano crescere con lo sviluppo di ulteriori soluzioni basate su questa architettura, considerando che FHIR offre numerose possibilità di estensione dei concetti.
- **Schedule Service**: servizio responsabile della prenotazione degli appuntamenti. Isola al suo interno la gestione degli slot disponibili nei quali è possibile riservare una visita, nonchè rende più semplice la sua gestione completamente indipendente dagli appuntamenti stessi.

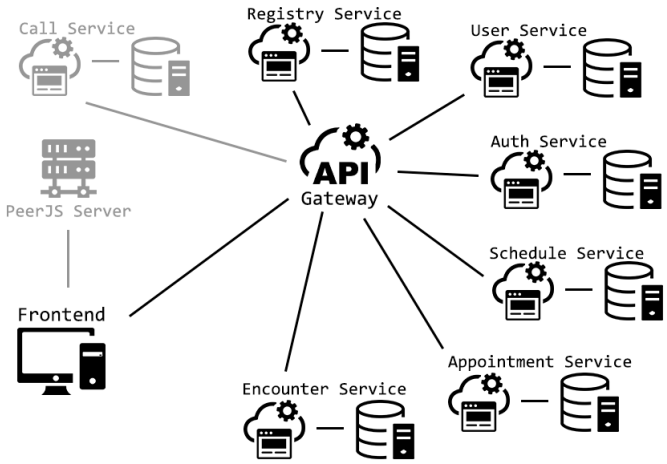


Figura 4.7: Composizione del servizio telemedico. Nella figura vengono riportate anche funzionalità relative al "*Call Service*", in quanto alcuni microservizi verranno condivisi dalle soluzioni per una maggiore integrità dei dati.

La figura 4.7 dimostra le capacità di un'architettura a microservizi ad essere adattata anche durante un cambio di contesto: la dovuta modifica relativa agli utenti è resa necessaria soltanto per mantenere un'integrità dei dati circolanti all'interno di sistema, ma non intacca in alcun modo il precedente funzionamento del "*Call Service*", che rimane perfettamente staccabile dal modulo previsto. Si vuole quindi apprezzare la corretta progettazione di tale servizio, in quanto ha permesso con alcuni semplici modifiche non invasive, di poter essere riusato e integrato all'interno di un ambito ben più specifico.

4.5.1 Verifica casi d'uso

Per verificare una tale progettazione architetturale, si esegue una possibile percorrenza ad alto livello delle richieste che possono provenire dai client, in modo da verificare i diversi caso d'uso precedentemente esposti.

1. **L'utente entra nel sistema**: nessuna criticità - il sistema prevede l'autenticazione attraverso "*Auth Service*" e gli utenti sono comunque identificati tramite i loro nickname e password, unici in tutta l'architettura progettata.

2. **L'utente richiede l'assistenza di un medico**: nessuna criticità - l'utente può utilizzare il servizio di prenotazione e fissare appuntamenti. Il medico e/o l'operatore provvederanno ad approvarli e a creare gli incontri.

3. **Il medico accetta/rifiuta la proposta**: nessuna criticità - "*Appointment Service*" fornirà tale funzionalità.

4. **Viene prenotato un appuntamento**: nessuna criticità - è a disposizione il servizio di prenotazione.

5. **Si effettua un incontro in una Room creata dal Call Service**: nessuna criticità - il servizio di incontri provvederà a comunicare tramite l'API la volontà di creare una stanza; una volta creata il suo riferimento sarà visibile ai partecipanti di quell'incontro.

6. **Viene richiesta la misurazione di un parametro**: nessuna criticità - il servizio relativo agli incontri provvederà all'API per l'aggiunta di tali parametri.

7. **Viene steso un referto**: nessuna criticità - il servizio relativo agli incontri provvederà all'API per l'aggiunta di tale documento.

8. **Invio di notifiche**: criticità rilevata - attualmente il diagramma presupporrebbe l'esistenza di un canale di notifiche per ogni microservizio presente. È una scelta della quale bisogna discutere ed approfondire le conseguenze.

4.5.2 Correzione criticità

Si considera critico soltanto il punto relativo all'invio di notifiche, in quanto sono possibili due strade percorribili in questo ambito:

- **Notifiche Integrate**: ogni microservizio che manda notifiche ha la sua API relativa al loro invio. La soluzione permette così ad ogni microservizio di isolare la propria parte di notifiche e automaticamente si ottiene una suddivisione in canali d'interesse. Lo svantaggio principale è la scarsa adattabilità di questo sistema e una completa frammentazione di un servizio: si può infatti immaginare, lato utente, un "*Notification Center*" dove potrebbero essere accumulate tutte le notifiche in arrivo. Una suddivisione di questo tipo non permetterebbe una facile implementazione di tale risorsa.

- **Notification As Service**: ogni microservizio fa riferimento ad un'ulteriore API con la quale l'utente si può interfacciare. Attraverso di essa vengono accumulate tutte le notifiche possibili mandate dall'applicazione, rendendo facile la loro fruizione. Lo svantaggio principale consiste nell'introduzione di un ulteriore microservizio che si occupi soltanto di

notifiche: questo aspetto permette però di isolare la tecnologia utilizzata per tale scopo, che altrimenti dovrebbe ricadere al livello di ogni microservizio. Un ulteriore vantaggio, indipendente dal contesto, è la possibilità di riuso di un fantomatico "*Notification Service*", in quanto ogni futura architettura mandante notifiche dovrà far riferimento ad un unico servizio tramite la sua API.

La seconda soluzione pare indubbiamente migliore, anche per aspetti puramente riguardanti l'architettura scelta: infatti, isolando il servizio delle notifiche si rendere l'applicazione più tollerante ad eventuali fallimenti. Essendo ogni architettura propensa ad avere criticità non rilevate durante la pianificazione, è utile poter permettere all'applicazione di funzionare nonostante qualche suo componente risulti essere non disponibile. Questo concetto è valido anche per gli altri microservizi: ad esempio, se per qualche motivo il servizio relativo agli incontri smettesse di funzionare, quello riguardante le prenotazioni degli appuntamenti sarebbe ancora pienamente operativo perchè indipendente dal precedente.

Questa precisazione permette di finalizzare la scelta definitiva dell'architettura, visibile in figura 4.8. Essendo ogni servizio indipendente, l'intera struttura risulta essere più tollerante ad errori, completamente modulare e perfettamente corrispondente ai requisiti posti. Inoltre, tale composizione sottolinea ancora una volta che la suddivisione dei ruoli tra i vari microservizi ha permesso una notevole semplificazione e il riuso dei componenti già progettati, predisponendoli ad essere modificati ed estesi a piacere. La contrapposizione tra un servizio "*base*" e quello più avanzato risulta essere quindi perfettamente in linea con questo concetto, evidenziando che un attento studio dell'architettura adottata porti a soluzioni compatibili, riusabili, modulari e ben strutturate.

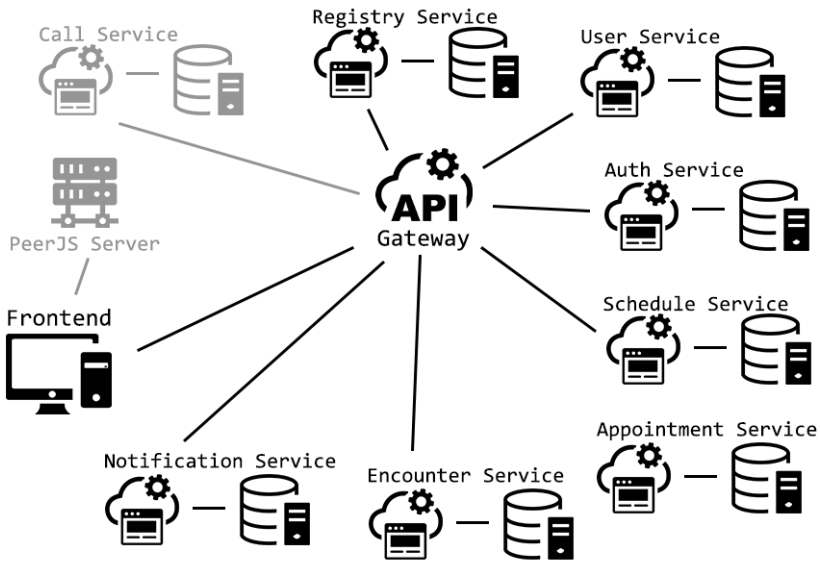


Figura 4.8: Composizione completa del Sistema di Televisita, comprendendo la parte del "*Call Service*".

4.6 Tecnologie Utilizzate

Oltre a FHIR, discusso nella sezione 1.3, non vi è alcuna tecnologia in più utilizzata rispetto al "*Call Service*". Volendo precisare un minuscolo dettaglio relativo al servizio di notifiche, è utile sottolineare che al momento è previsto un semplice utilizzo di un canale WebSocket per il loro invio. La modularità dell'applicazione permetterà in futuro un cambio di tale funzionalità, estendendola maggiormente anche in vista dei possibili utilizzatori lato *mobile app*. Inoltre, un dettaglio relativo al *frontend* è ovviamente il fatto di estenderlo con funzionalità relative al servizio di Televisita, riutilizzando anche questa volta il più possibile quello già strutturato per il "*Call Service*".

Per le rimanenti tecnologie utilizzate è possibile far riferimento alla sezione 3.7.

4.7 Design di dettaglio

4.7.1 Strutturazione dei microservizi

Tenendo presente le considerazioni effettuate sul "*Call Service*" nella sezione 3.8.1, è possibile apprezzare che gli stessi principi possono venir riutilizzati

anche nella piattaforma riguardante la Televisita: infatti, vi saranno sempre delle parti in comune tra tutti i servizi, per poi passare ai singoli dettagli di ognuno di essi. La schematizzazione generale dei microservizi per il "*Televisiting System*" è stata già analizzata ed è visibile nella figura 4.8. In questa sezione vengono trattate soltanto le parti rilevanti al servizio in progettazione, in quanto quelle del "*Call Service*" sono state già trattate nelle sezioni precedenti. L'unica eccezione si pone sul servizio dedicato agli utenti, il quale viene esteso di funzionalità relative a coprire il modello del dominio.

User Service

Rispecchia pienamente le API del servizio trattato nella sezione 3.3.2 e 3.8.1, ma in più, come da schema 4.2, viene esteso con i dettagli relativi a possibili ruoli nel contesto di telemedicina. Infatti, oltre a definire alcuni campi aggiuntivi, offre una sua specializzazione ("*Practitioner*") per definire il ruolo dell'operatore sanitario. La struttura di esso non cambia, in quanto si tratta soltanto di parametri aggiuntivi che possono essere ignorati dai sistemi che non ne hanno bisogno.

Schedule Service

Permette un disaccoppiamento tra gli slot disponibili per gli appuntamenti e gli appuntamenti stessi. Gestisce la parte relativa alla prenotazione degli slot, utilizzati successivamente per occupare effettivamente l'appuntamento. È utile tenere separato il *management* di tale struttura in quanto la sua organizzazione potrebbe dipendere in futuro da altri fattori, quali enti o piattaforme esterni. Un ulteriore motivo per cui questo servizio dev'essere isolato è relativo allo scenario in figura 2.7: in esso si richiede che il paziente possa richiedere uno slot soltanto se vi è disponibilità di utilizzo di tale piattaforma.

Appointment Service

Servizio capace di offrire le possibilità per fissare e aggiornare appuntamenti. Anche se risulta essere il luogo dove questi effettivamente vengono gestiti, gli appuntamenti possono essere visualizzati anche a partire dalla risorsa "*Utente*": anche in quel caso non vi è una duplicazione di risorse, bensì un puntamento di due identificatori verso una stessa risorsa (è sempre compito di "*Appointment Service*" procurarsi i dettagli dei relativi appuntamenti).

Il servizio fa riferimento diretto a "*Notification Service*", in quanto al cambiamento di stato di un appuntamento provvede ad informare i soggetti interessati di tale avvenimento. Inoltre, è dipendente da "*Schedule Service*" per

quanto riguarda le prenotazioni stesse e da "*Encounter Service*" per la programmazione degli incontri. Si può apprezzare quindi la modularità di tale soluzione, che, soprattutto nel caso di guasti di uno dei servizi elencati, continua comunque a funzionare indipendentemente da essi. Ovviamente, questo introduce sfide che dovranno essere gestite nel lato più implementativo, rendendo il futuro codice più difficile da strutturare, ma più facile da manutenere.

Encounter Service

Servizio che gestisce gli incontri relativi ad un appuntamento. Anche se risulta essere il luogo dove questi effettivamente vengono gestiti, gli incontri possono essere visualizzati anche a partire dalla risorsa "*Appuntamento*": in quel caso non vi è una duplicazione di risorse, bensì un puntamento di due identificatori verso una stessa risorsa: è sempre compito di "*Encounter Service*" procurarsi i dettagli dei relativi appuntamenti.

Il servizio fa riferimento diretto a "*Notification Service*", in quanto al cambiamento di stato di un appuntamento provvede ad informare i soggetti interessati di tale avvenimento. Inoltre, è dipendente da "*Appointment Service*" per quanto riguarda gli appuntamenti ai quali un incontro deve far riferimento. Al suo interno sono direttamente gestite questioni relative a misurazioni e referti: non vi era necessità di specificare ulteriori microservizi a riguardo, onde evitare il rischio di avere un'architettura eccessivamente granulosa e difficile da manutenere.

La diversificazione appuntamento-incontro è un puro concetto future-proof, in quanto il contesto attuale non prevede l'utilizzo di attributi al di fuori del puro modello relativo alla Televisita: non vi è infatti nessun requisito posto sotto tale punto di vista. Volendo giustificare questa scelta, si fa riferimento più alla volontà di mantenere l'applicazione interoperabile (adottando il modello FHIR) e modulare, facilitando la sua eventuale espansione. Non si tratta dunque di sovra-ingegnerizzazione dei concetti, bensì una corretta pianificazione con uno sguardo verso il futuro della soluzione progettata.

Notification Service

Servizio responsabile all'invio di notifiche ai *listener* sottoscritti su un determinato canale. Le notifiche sono strettamente personali: i vari servizi effettuano chiamate API su questo microservizio per poter delegare il compito della loro gestione, mentre il servizio stesso si preoccupa di farle recapitare ai corretti destinatari sottoscritti sui canali richiesti.

Anche in questo caso si tratta perlopiù di un servizio future-proof, che potrà accumulare al suo interno tutte le notifiche riguardanti gli eventuali moduli aggiunti in futuro. Esso semplifica notevolmente la gestione di tale risorsa, permettendo di isolare architetturalmente tale aspetto dai singoli servizi, che non devono più essere responsabili dell'invio delle notifiche. Inoltre, grazie alla strutturazione a microservizi, essendo poi "*Notification Service*" un servizio di secondaria importanza, è possibile che esso possa andare in errore senza intaccare in nessun modo il complessivo funzionamento della soluzione progettata. Tale scelta permette inoltre di cambiare a piacere il modo nel quale le notifiche vengono inviate, che attualmente utilizzano un canale WebSocket.

4.7.2 Frontend

Anche in questo caso la struttura del *frontend* relativa a "*Call Service*" può essere utilizzata come una buona base di estensione. L'interfaccia utente deve poter garantire sia ai pazienti che agli operatori sanitari di interagire con il sistema, in modo congruo ai loro ruoli e requisiti posti. I principi relativi alla comunicazione *backend-frontend* risultano dunque essere completamente compatibili con quelli relativi al servizio delle chiamate, trattate nella sezione 3.8.2: ovviamente, essendo la parte della videochiamata un modulo imprescindibile del *frontend*, anche esso sarà ereditato nella soluzione finale. Di conseguenza, rimane utile analizzare le parti restanti, strettamente relative al contesto della Televisita, e progettare i componenti in modo da soddisfare gli scenari richiesti:

- **Login**: rispecchia la struttura del "*Call Service*".
- **Home**: verrà predisposto un nuovo "*HomeComponent*" in grado di:
  - visualizzare recenti notifiche,
  - visualizzare recenti appuntamenti,
  - prenotare appuntamenti.
- **Schedule**: verrà predisposto un "*ScheduleComponent*" che visualizzerà gli slot prenotabili a disposizione, se il relativo servizio risulterà essere operativo. Nel caso del "*Practitioner*", esso funzionerà a prescindere.
- **Appoitments**: verrà predisposto un "*AppoitmentsComponent*" che gestirà tutta la parte relativa al *management* degli appuntamenti già prenotati, nonchè la loro visualizzazione. Dalla stessa schermata si potrà passare ad un appuntamento specifico, per visualizzare i relativi incontri.
- **Appoitment**: verrà predisposto un "*AppoitmentComponent*" che visualizzerà un particolare appuntamento con i relativi dettagli. Da esso si

potranno consultare i relativi incontri e/o gestire funzionalità aggiuntive non disponibili nella schermata generale di appuntamenti.

- **Encounters**: verrà predisposto un "*EncountersComponent*" che visualizzerà la lista degli incontri pianificati per quel determinato appuntamento. Da esso si potrà passare alla visualizzazione di un particolare incontro.

- **Encounter**: verrà predisposto un "*EncounterComponent*" che potrà gestire i dettagli relativi ad un determinato incontro in un determinato appuntamento. Da questa schermata l'utente potrà passare alla Televisita, attivando il relativo incontro ed entrando nella stanza creata da "*Call Service*".

- **Notification**: verrà predisposto un "*NotificationComponent*" che visualizzerà tutte le notifiche in ingresso per quel determinato utente. Ogni notifica punterà alla risorsa cui fa riferimento, in modo da potersi spostare comodamente tra di esse.

Considerando l'intera struttura, è utile prevedere sotto-componenti gestenti alcune delle parti minori, per delegare parte del lavoro della logica e della visualizzazione e isolare il comportamento dei singoli componenti:

- **AppointmentViewComponent**: risulterà essere la *preview* di un singolo appuntamento nell'elenco degli appuntamenti.

- **EncounterViewComponent**: risulterà essere la *preview* di un singolo incontro nell'elenco degli incontri.

- **NotificationViewComponent**: risulterà essere la *preview* della notifica nell'elenco delle notifiche.

- **ScheduleViewComponent**: rappresenterà uno singolo slot di prenotazione nel servizio di prenotazioni.

Si propongono di conseguenza dei mockup relativi alla parte della user experience: non vengono evidenziati componenti ovviamente derivabili, come il centro notifiche (estraibile da "*HomeComponent*") e non vi è la ripetizione del mockup del login, direttamente preso da "*Call Service*". Il menù laterale serve soltanto per spostarsi comodamente tra le sezioni già evidenziate e consiste in un elenco composto da pulsanti: *Home*, *Appointments*, *Notifications* e *Logout*.

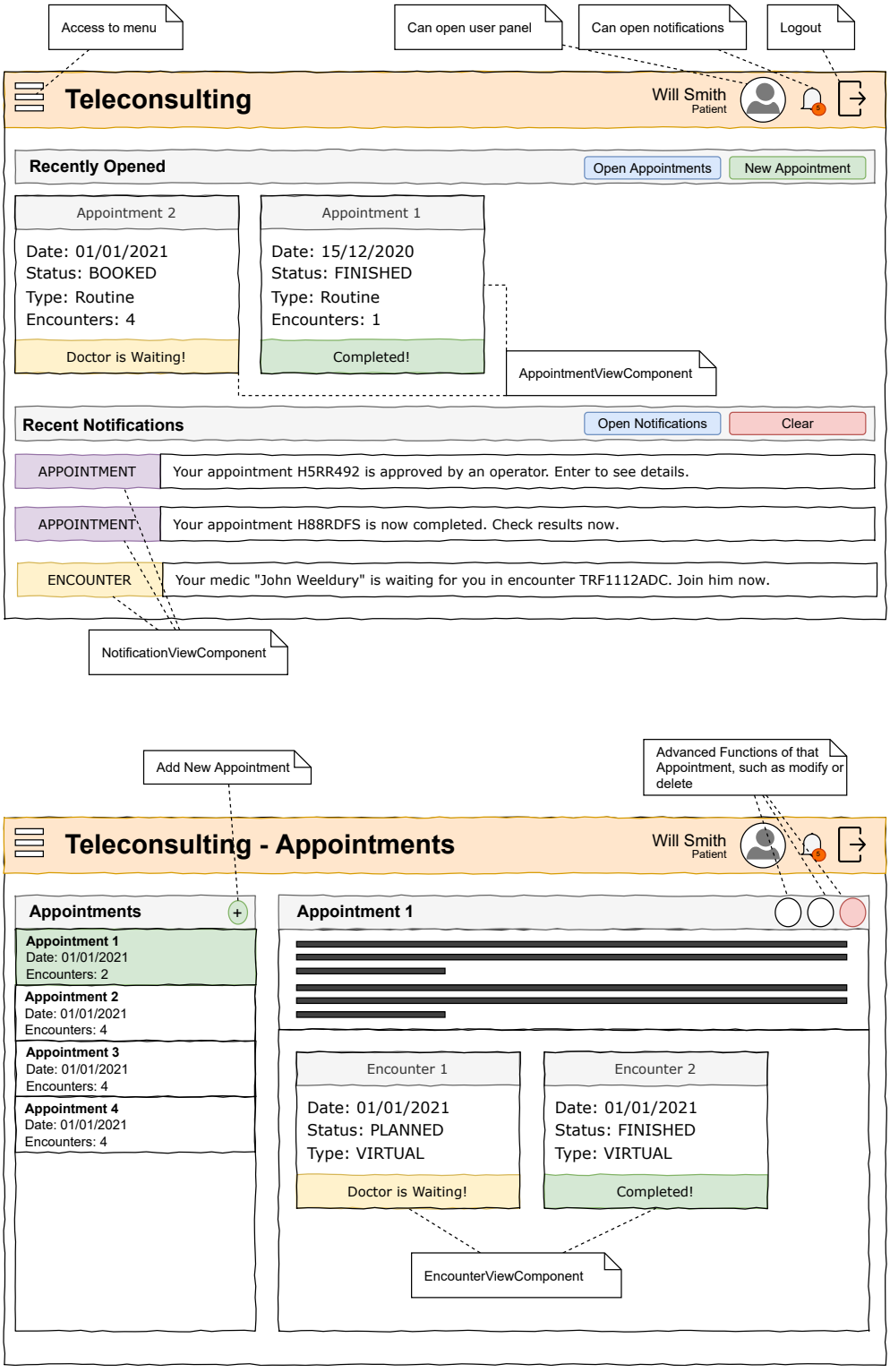


Figura 4.9: Mockup Televisita (parte 1).

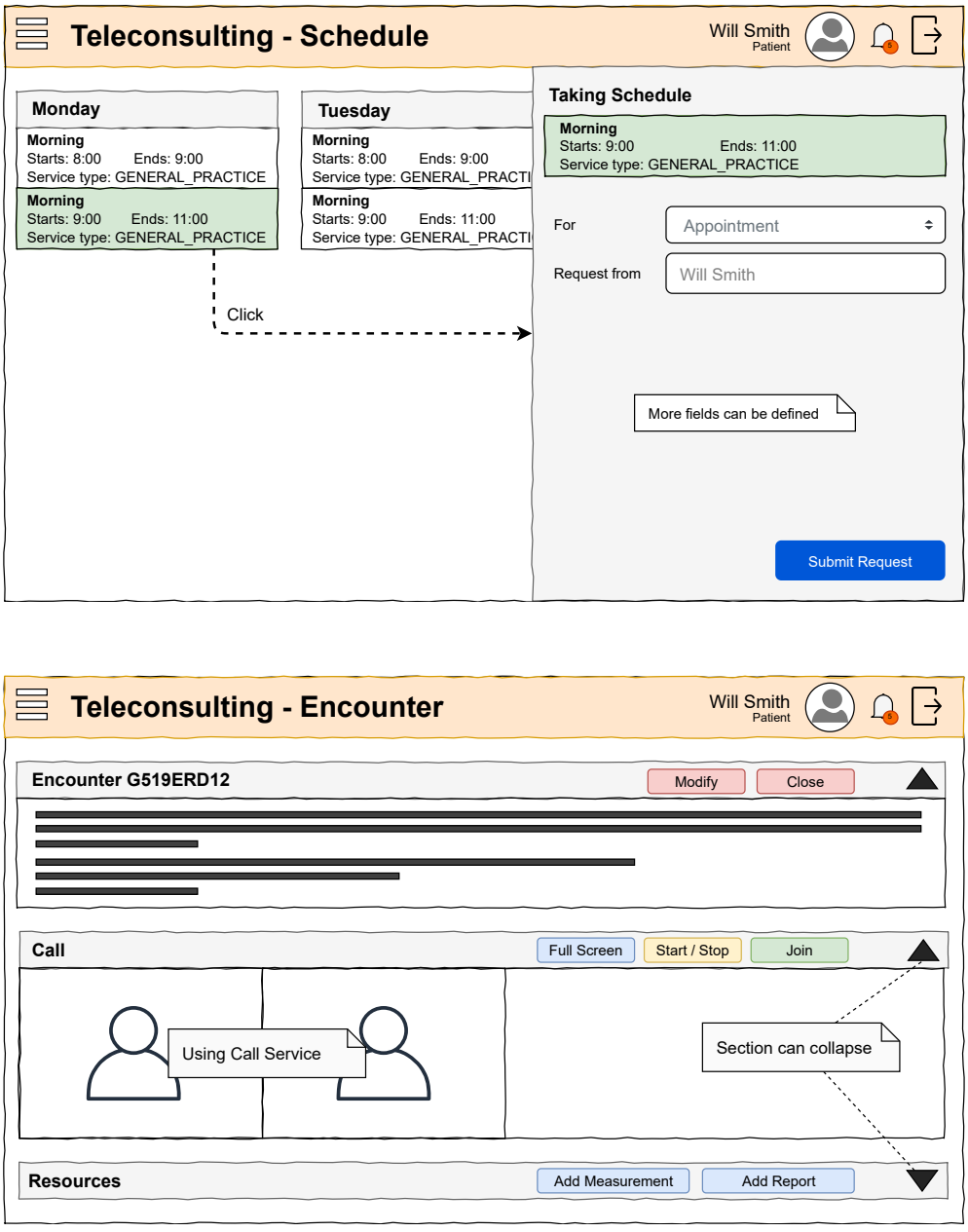


Figura 4.10: Mockup Televisita (parte 2).

4.8 Verifica scenari

Essendo il modulo della Televisita nettamente più complesso di quello relativo alle chiamate, si preferisce effettuare un ulteriore test di architettura riprendendo gli schemi rilevati durante le analisi dei requisiti. Questo passaggio permette di contrapporre le API progettate al modello del dominio, per verificarne eventualmente criticità e possibilità di miglioramento. Inoltre, grazie ad una verifica del flusso, sarà possibile passare più agevolmente all'effettiva implementazione di questa parte.

4.8.1 Richiesta di un appuntamento

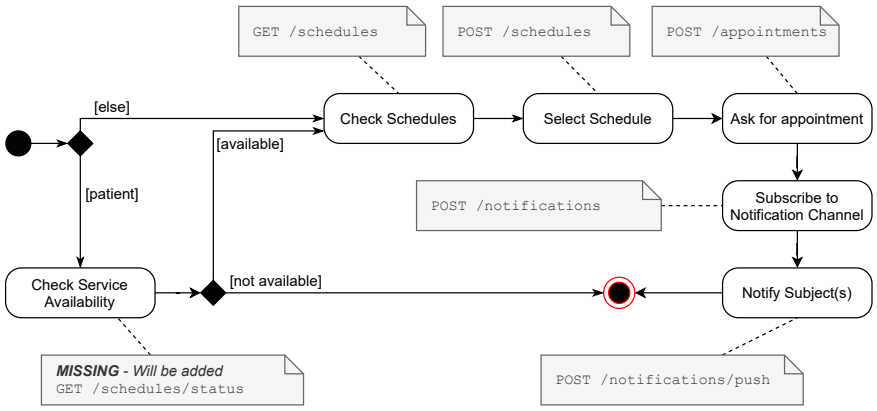


Figura 4.11: Rivisitazione della figura 2.7. Si può notare una minima criticità relativa al servizio di prenotazione.

Nella figura 4.11 è possibile notare una minima criticità: non vi è infatti modo per sapere se il servizio di prenotazioni sia attivo o meno: per questo motivo alle API finali verrà aggiunta anche una risorsa per ricavarne lo stato.

4.8.2 Ciclo di vita dell'appuntamento

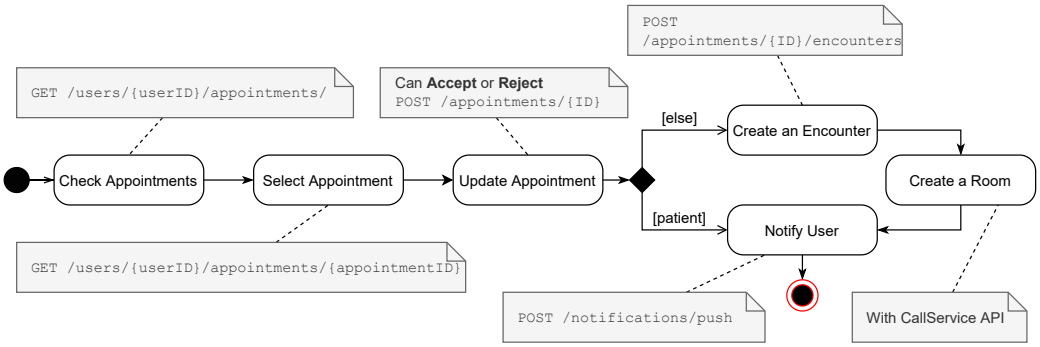


Figura 4.12: Rivisitazione della figura 2.8. Non vi sono criticità rilevate.

Nella figura 4.12 non risultano criticità rilevabili. Tutto il processo segue la logica, utilizzando tutte le API previste durante la progettazione.

4.8.3 Esecuzione Televisita

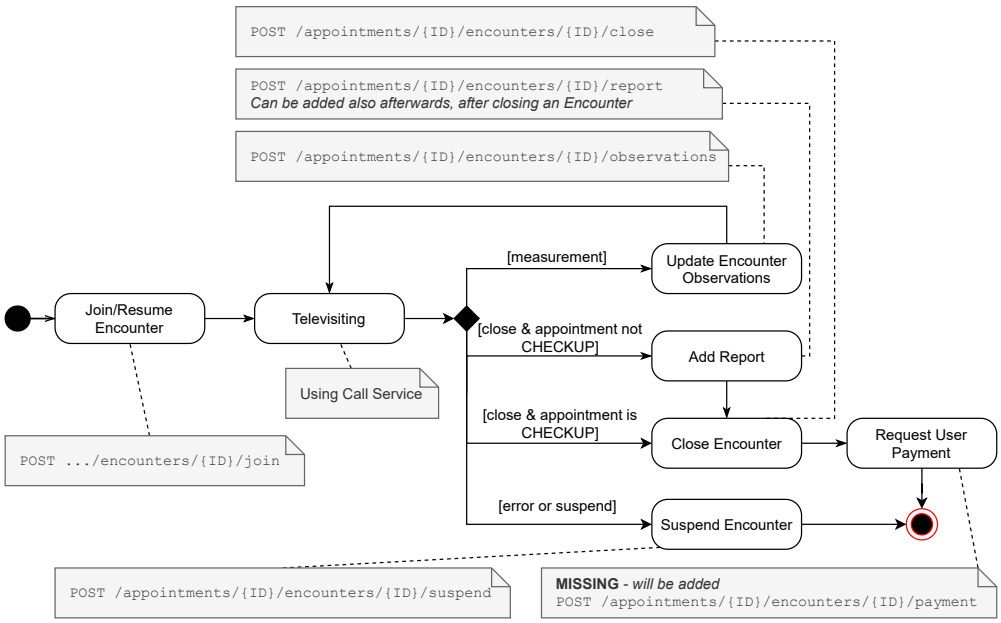


Figura 4.13: Rivisitazione della figura 2.9. Si può notare una criticità relativa al servizio di pagamento.

Nella figura 4.13 è possibile notare una criticità: non vi è infatti modo per segnalare e/o effettuare un pagamento per una visita effettuata. Essendo questa parte richiesta dai requisiti, ma non essendovi presenti specifiche

sulla modalità di avvenimento di tale procedura, si decide di provvedere semplicemente all'interfaccia per questa funzionalità, ma di non spendere risorse per progettarla. Questo è dovuto alla natura stessa del pagamento, cioè di poter dipendere da diverse questioni, legate perlopiù sul tipo di prestazione eseguita (pubblica o privata), su eventuali sconti, benefici, modalità di pagamento e convenzioni, nonchè rateizzazioni e pagamenti in anticipo/ritardo. Si prevede, di conseguenza, che tale aspetto debba essere progettato **in un servizio a parte**, che gestisce tutte le relative problematiche: un pagamento farà riferimento ad una (o più) risorse, ma sarà compito di un servizio esterno considerare tutte le problematiche del caso.

Come però già sottolineato, la progettazione di un servizio per i pagamenti non fa parte dei requisiti, di conseguenza, è preferibile chiarire l'aspetto di tale sistema e poi provvedere ad una sua corretta strutturazione. Progettando sin dall'inizio i casi relativi ad un generico servizio di pagamenti porterebbe ad una sovra-ingegnerizzazione dell'applicazione, introducendo funzionalità non espressamente richieste dai requisiti.

Capitolo 5

Implementazione Prototipale

Questo capitolo tratterà dello sviluppo della parte relativa al Call Service, sia dal lato *frontend* che dal lato *backend*. Il capitolo vuole, a scopo dimostrativo, realizzare i componenti progettati in via teorica e metterli alla prova di fronte ai requisiti. Si vuole sottolineare che non vi sono riferimenti alle parti trattanti di applicazioni mobile, né del Sistema di Televisita, in quanto la loro realizzazione fa parte di ulteriori progetti non trattati in questa analisi. Si vuole però notare sin dall'inizio che la strutturazione del *backend* è utilizzabile indipendentemente dalla tecnologia sfruttata per lo sviluppo di un qualsivoglia *client*.

5.1 Strumenti utilizzati

L'intero progetto è svolto sotto ambiente Windows 10. Per lo sviluppo del codice *backend* si utilizza la piattaforma IntelliJ IDEA Community Edition, insieme a Visual Studio Code. Viene utilizzato Docker per il contenimento delle immagini di MongoDB e Redis. Inoltre, per l'esplorazione del primo database si fa riferimento a Robo 3T. Per quanto riguarda lo sviluppo *frontend*, viene utilizzato esclusivamente Visual Studio Code. Le pagine web vengono poi testate su un browser basato su Chromium e Google Chrome stesso.

5.2 Backend

5.2.1 Microservizi

Come definito nel design, i microservizi hanno alcune caratteristiche comuni che possono essere configurate per la maggior parte di essi. Volendo rispettare il principio di modularità ed estensibilità, si strutturano due accomunamenti:

- **common** - funzionalità di inizializzazione di tali servizi, lettura di preferenze, registrazione presso il servizio di discovery e configurazione del *router*, permettente l'utilizzo di REST API. Essendo tali *feature* caratteristiche di ogni microservizio, vengono accumulate in un unico luogo.
- **mongo-common** - estensione del common che include al suo interno funzionalità aggiuntive per l'interfacciamento a MongoDB.

Di conseguenza, ogni microservizio, in base alle funzionalità richieste, estende da uno dei due servizi base per definirsi:

- **auth-service** - *mongo-common*. È essenziale che il servizio di autenticazione riesca a connettersi al database per conoscere i dettagli degli utenti attualmente nel sistema.
- **call-service** - *mongo-common*. Essendovi presente il concetto di stanze e partecipanti, è obbligatorio permettere a questo servizio la connessione al database.
- **gateway** - *common*. Si tratta del punto d'ingresso che non ha bisogno di accedere a particolari strutture dati: si interfaccia con i microservizi sottostanti.
- **registry-service** - *common*. Anche se ha bisogno di accesso ad un database, nel suo caso si tratta di Redis, per cui la dipendenza per MongoDB risulta essere superflua.
- **user-service** - *mongo-common*. Essendovi presente il concetto di utenti, è obbligatorio permettere a questo servizio la connessione al database.

Risulta opportuno notare che **PeerJS Server** non fa strettamente parte dell'architettura a microservizi, in quanto è una soluzione già esistente e predefinita dalla libreria. Come definito nelle sezioni 3.1.1 e 3.5, essa viene trattata *as-is*, con tutte le API di base offerte e sufficienti per la copertura dei requisiti.

5.2.2 Strutturazione Generale

Le specifiche di ogni singolo microservizio sono pressochè simili, per cui si preferisce approfondire la strutturazione generica che accomuna tutte le soluzioni. Ogni istanza di *Vert.X*, chiamata *verticle*, è lanciata su un processo a parte. Il lancio avviene grazie ad una dipendenza globale alla libreria del framework, la quale permette di fare tutto il setup iniziale ed eseguire la parte progettata. Questa parte diventa il `main` di ogni *verticle*, definito più in dettaglio attraverso la sua effettiva classe.

```
Vertx.vertx(options).deployVerticle(GatewayVerticle()) {
   if (it.succeeded()) {
      println("The verticle has been successfully deployed, with id:
          ${it.result()}")
   } else {
      println("Error during deployment: \n${it.cause()}")
   }
}
```

Listato 5.1: Lancio del *verticle* gateway. Si vuole notare come `GatewayVerticle()` sia l'inizializzazione della omonima classe, che controlla il successivo ciclo di vita del *verticle*.

Il lancio passa quindi all'effettiva classe che implementa il *verticle* interessato. Si tratta di una classe che estende da uno dei due *verticle* di base (*common* oppure *mongo-common*) e che al suo interno provvede alla configurazione di tutte le risorse necessarie al funzionamento di quella determinata parte del progetto. Si tratta generalmente nell'implementazione di tre metodi: `start`, `stop` e `initialize`, che gestiscono il lancio, e di eventuali variabili mutabili e immutabili per la fruizione del servizio.

Un ulteriore aspetto importante riguardante tutti i microservizi realizzati è la funzione che definisce le *routes* delle API riconosciute da essi: queste sono state già definite a livello architetturale nella sezione 3.3 e non verranno riprodotte. Ciò che preme sottolineare di tale aspetto è che, tranne il servizio *registry*, tutti i microservizi hanno al loro interno una *policy* di accettazione delle richieste in ingresso:

- Nel caso del *gateway* si tratta semplicemente di accettare tutte le richieste, analizzare se è possibile soddisfarle e rispondere adeguatamente. È necessario provvedere però all'autenticazione dell'utente.
- Nel caso di tutti gli altri servizi, si tratta di accettare le API definite nella sezione 3.3 relative a quel determinato servizio.

```
return getBaseRouter().apply {
   get("/$baseUrl/rooms").asyncHandler{ handleGetRooms(it) }
   post("/$baseUrl/rooms").asyncHandler{ handleCreateRoom(it) }
   get("/$baseUrl/rooms/:$URL_ROOM_ID_KEY").asyncHandler{ ... }
   ...
}
```

Listato 5.2: Esempio di gestione delle richieste in entrata al *call-service*.

5.2.3 Elaborazione richieste e risposte

Si passa successivamente ai relativi *handler* di funzionalità, presenti nel codice 5.2, specificate come parametro *lambda* all'espressione `asyncHandler`. Ognuna di esse svolge il compito relativo all'API richiesta, conforme con la sua definizione, analizzando i parametri d'ingresso e restituendo quelli in uscita, entrambi se presenti. Nel caso di malformazione di parametri, della loro incongruenza o in caso di interni malfunzionamenti, si fa riferimento al *RestError* trattato nella sezione 3.4.

Nel codice 5.3 è possibile apprezzare le potenzialità del linguaggio Kotlin. In primo luogo vi è l'utilizzo della parola `suspend`, che permette di dichiarare una funzione asincrona. In questo modo, l'utilizzo di tale metodo è vincolato ad essere richiamato in un ambiente non bloccante, affrontando in maniera molto agevole la natura non-sequenziale delle azioni che potrebbero capitare in tutto il ciclo di vita dell'applicazione. È però da notare che questa scelta "*maschera*" la vera natura del programma, introducendo ad eventuali assunzioni errate se si dovesse leggere il codice senza la conoscenza di questo dettaglio.

```
private suspend fun handleGetRoom(context: RoutingContext) {
   try {
      // Take params from request
      val roomId = context.getPathParamOrFail(
         URL_ROOM_ID_KEY,  // = "room"
         RestErrors.ROOM,  // type of error, if thrown
         "roomId"          // used to specify what param is wrong
      )

      // Take the room
      val room = dbHelp.onRooms.getRoom(roomId)

      // Send the response for client
```

```
      sendResponseOrError(
         room,
         context,
         RestErrors.ROOM
            .onPath(context.normalisedPath())
            .buildNotFound("roomId", roomId)
      )

   } catch (exception: Exception) {
      manageException(exception, context)
   }
}
```

Listato 5.3: Codice che permette l'ottenimento di una stanza.

Vi è poi un ampio uso delle cosiddette *extensions*: un costrutto del linguaggio Kotlin che permette di aggiungere funzionalità ad una determinata classe senza modificarne la struttura. Si tratta semplicemente di agganciare metodi che potranno essere eseguiti sullo stesso contesto dell'istanza della classe, come se ne facessero parte sin dall'inizio. Questo facilita la gestione del codice, in quanto non vi è necessità di estendere appositamente una classe per ottenere una funzionalità in più. Nel codice 5.3 la *extension* utilizzata fa riferimento a `getPathParamOrFail`, che rende automatico il *parsing* dell'input e il lancio di errori in caso di problemi. L'esecuzione di tale handler si può interrompere in qualsiasi momento a causa di un'eccezione, gestita poi successivamente nel metodo `manageException`: in caso di eccezioni pianificate, come quelle relative a *RestError*, le risposte sono generate in modo da far comprendere all'utilizzatore la natura dell'errore (senza svelare nulla dell'architettura sottostante); per i rari casi di errori imprevisti, viene invece generato un generico *RestError*.

```
fun <T>RoutingContext.getBodyOrFail(clazz: Class<T>, restError:
    RestErrors.RestError): T {
   try {
      return this.bodyAsJson.mapTo(clazz)
   } catch (exception: Exception) {
      throw BackendException(restError)
   }
}
```

Listato 5.4: Esempio di *extension* che aggiunge al `RoutingContext` la possibilità di convertire automaticamente il *body* in una classe specifica rappresentante quell'oggetto di input.

Per quanto riguarda `dbHelp` e `sendResponseOrError`, seguono gli stessi principi già descritti nei due paragrafi soprastanti.

- Il primo risulta essere una classe di appoggio disponibile per i *verticle* che estendono da *mongo-common*, e offre un'interfaccia agevole che nasconde l'implementazione delle query al database sottostanti. Anch'essa, in caso di problemi, lancia *RestError* che sono gestiti dal costrutto `try-catch`. Si vuole sottolineare che essendo richiamata da una `suspend function` l'operazione può risultare bloccante: il codice non ne risentirà, in quanto verrà eseguito al di fuori del flusso principale.
- Il secondo è un metodo di utility che permette di inviare l'oggetto (se valido) o il *RestError* (se l'oggetto non fosse valido) al richiedente.

```
private fun <T>sendResponseOrError(obj: T?, routingContext:
    RoutingContext, restError: RestErrors.RestError) {
   if (obj != null) {
      routingContext.response().json(JsonObject.mapFrom(obj))
   } else {
      routingContext.response().failWith(restError)
   }
}
```

Listato 5.5: Implementazione di `sendResponseOrError`: essa vede l'uso di ben due *extension*.

Per l'ottenimento dei dati essenziali per lo svolgimento di una determinata operazione, si utilizza lo stesso principio descritto nel codice 5.3: si richiede il parametro con una funzione *lambda* e si prosegue soltanto nel caso di risposta affermativa. Non essendo tale *feature* richiesta in ogni ambito, si preferisce utilizzare una funzione capace di accoglierne il significato.

```
...
val user = requireNotNull(
   RestErrors.USER
      .onPath(context.normalisedPath())
      .buildNotFound("username", request.username)) {
   dbHelp.onUsers.getUserByUsername(request.username)
}
...
```

Listato 5.6: La variabile "*utente*" è cruciale per l'esecuzione di tale codice: se la chiamata al database avrà successo, l'esecuzione del codice proseguirà; in alternativa verrà lanciata un'eccezione.

5.2.4 Organizzazione API

Si vuole notare che `baseUrl`, di cui riferimento nel codice 5.2, è essenzialmente la parte iniziale con cui ogni servizio si identifica. Per una corretta categorizzazione delle API, esse seguono un ben determinato schema che permette al *gateway* di differenziare le richieste in modo agevole. L'esempio di una tale richiesta può essere banalmente `/api/call/rooms`, dove si possono trovare elementi come:

- `/api` - prefisso comune a tutti i microservizi. Permette di raggruppare tutte le API pianificate.
- `/call` - nome del servizio, comune a tutti i microservizi che espongono un API. Permette al *gateway* di individuare il microservizio richiesto e di direzionare le richieste in arrivo verso tale destinazione. Per identificare il nome di un servizio si elimina il suffisso "-*service*" .
- `/rooms` - parte relativa alle API effettivamente progettate.

Tale strutturazione è resa da due principali motivi: in primo luogo si vogliono accumulare tutte le API offerte dal *backend* in un unico luogo, disponibile sotto indirizzo `/api`; in secondo luogo, dato che il *gateway* non è stato progettato per offrire funzionalità avanzate, si preferisce astrarre dal metodo scelto per l'individuazione dei microservizi sottostanti e, per scopi dimostrativi, passare il servizio richiesto come parte del *path*.

Con tale affermazione si ricorda che qualsiasi chiamata, prima di essere indirizzata verso il microservizio corrispondente, passa per il *gateway*: per questo motivo non vi è presente un'API come `/api/gateway/`, in quanto completamente fuorviante.

5.2.5 Database

In un'architettura a microservizi ogni servizio che necessita un collegamento con il database dovrebbe avere un suo riferimento a tale servizio. Per motivi di semplicità e di prototipazione si utilizzano in realtà soltanto due istanze: una per MongoDB e l'altra per Redis; in questo modo è possibile apprezzare più velocemente i risultati offerti dalla soluzione realizzata.

Per quanto riguarda MongoDB è stata predisposta una classe di utility che agevola notevolmente il suo utilizzo. Sempre per motivi di semplicità, si accomunano tutti i riferimenti dei singoli database specifici in un'unica classe. I dettagli relativi all'implementazione delle specifiche interrogazioni a più basso

livello rimangono incapsulate al suo interno, cosicché i servizi che ne fanno uso possano vedere soltanto una generica chiamata al metodo, piuttosto che una reale *query* MongoDB. La classe `DBHelp` si compone di diversi elementi che permettono tale strutturazione:

- `CommonFunctions` - classe che raggruppa tutte le funzionalità comuni specifiche di MongoDB, come le letture, scritture e metodi di utility per la creazione delle query.

- `RoomHelp`, `ParticipantHelp`, `InviteHelp`, `UserHelp` - classi contenenti al loro interno le possibili interrogazioni al database richieste dalla soluzione sviluppata.

Tale classe potrà essere opportunamente modificata in vista dei futuri sviluppi per poter isolare meglio la tecnologia del database sottostante, in quanto è strutturata in modo da essere completamente indipendente da quest'ultima. Gli utilizzatori finali, ovvero i servizi, eseguono semplici chiamate, come quelle nel codice 5.3.

5.3 Frontend

5.3.1 Moduli

Angular permette un'agevole strutturazione dell'intera soluzione discussa già nella sezione 3.8.2. Dipendentemente dallo scopo di un determinato componente, esso avrà un ruolo diverso nella gestione dell'applicazione. L'organizzazione interna dei file sorgenti segue una suddivisione in sottocartelle:

- `Components` - elementi della view con annessa la logica per gestirla. Ne fanno parte tutti i componenti già dichiarati in fase di progettazione, come `HomeComponent` o `RoomComponent`

- `Models` - elementi relativi al modello, classi che definiscono oggetti come "*Room*" o "*Participant*".

- `Services` - elementi responsabili alla comunicazione *frontend-backend*. Non sono da ritenere come entità costantemente in esecuzione, bensì elementi di utility che permettono una comoda organizzazione e isolamento del codice.

Components

I componenti non si differenziano moltissimo per le funzionalità che offrono: hanno una parte che dichiara l'interfaccia grafica (HTML + CSS) e una parte che controlla tale aspetto (TypeScript). È del tutto superfluo descrivere ogni componente nei minimi dettagli, per cui si preferisce puntare su alcuni esempi significativi del codice sviluppato.

In primo luogo, vi è un ampio uso di "*direttive Angular*", che permettono di inglobare parte della logica dell'applicazione direttamente nel codice HTML. Attraverso questa *feature* è possibile accedere alle variabili definite a livello del *controller* ed elaborarle a piacimento: un classico esempio è quello relativo alla visualizzazione di una lista. Come si può notare dal codice 5.7, la formattazione segue le linee di HTML, ma vi sono alcune differenze sostanziali: esse fanno parte delle specifiche di Angular, alle quali si può far riferimento direttamente sulla documentazione ufficiale [19]. Nello stesso codice è possibile apprezzare una completa connessione con il *controller*, grazie al fatto di poter richiamare direttamente le sue funzioni (come ad esempio `onRoomClick(...)`), passando addirittura l'argomento utilizzato per generare quella determinata lista con una direttiva Angular.

```
<div *ngFor="let room of activeRooms;">
   <mat-list-item (click)="onRoomClick(room)">
      <p matLine>{{room.name}} <i>[{{room.id}}]</i></p>
      <mat-icon>call</mat-icon>
   </mat-list-item>
   <mat-divider [inset]="true" class="base-divider"></mat-divider>
</div>
```

Listato 5.7: Parte del codice responsabile a visualizzare tutte le stanze disponibili. Lato *controller* vi è una variabile chiamata `activeRooms` contenente gli oggetti di tipo `Room` e un metodo `onRoomClick(...)`.

Il riuso e l'isolamento dei componenti è un'altro aspetto fondamentale dell'intera struttura. Si prenda come esempio il componente "*RoomComponent*": esso al suo interno deve gestire sia la parte generica riguardante una stanza che la visualizzazione dei flussi audio/video in ingresso. Coordinare tutti questi aspetti in un unico componente porterebbe ad un sovradimensionamento delle responsabilità di quest'ultimo, per cui si preferisce partizionare il lavoro creando uno specifico componente per la sola parte di visualizzazione video. Tramite un semplice riferimento in HTML e la sua successiva indicazione lato controller, è possibile infatti non solo isolare ma anche riutilizzare componenti

in diverse pagine. Le funzioni pubbliche dichiarate nel componente utilizzato saranno visibili e richiamabili dal componente padre.

```
<div>
   ...
   <app-video-container #videoContainer></app-video-container>
   ...
</div>
```

Listato 5.8: Parte dell'HTML relativo a "*RoomComponent*".

```
export class RoomComponent implements OnInit {
   @ViewChild('videoContainer', { static: false, read:
       VideoContainerComponent })
   videoContainer: VideoContainerComponent;
   ...
}
```

Listato 5.9: Parte del *controller* relativo a "*RoomComponent*".

Una parte importante della user experience riguarda la comunicazione: essendo le finestre di dialogo un aspetto imprescindibile di una qualsiasi applicazione, anche nella versione web trovano la loro implementazione. Relativamente al progetto, si può prendere in esempio "*error-dialog component*", il quale ha il solo compito di visualizzare a video un messaggio. Grazie ad una semplice struttura HTML e un codice estremamente ridotto nel *controller* è possibile riutilizzare a piacimento tale struttura, senza complicare il codice. Per poter utilizzare tale componente nella struttura del progetto, occorre anche dichiararlo in `entryComponents` dentro la dichiarazione del modulo dell'applicazione.

```
<h2 mat-dialog-title>{{title}}</h2>
<mat-dialog-content>
   <div *ngIf="message != null" style="white-space: pre-line">{{
       message }}</div>
   <div *ngIf="htmlMessage != null" [innerHTML]="htmlMessage"></div>
</mat-dialog-content>

<mat-dialog-actions>
   <button mat-raised-button color="primary"
       (click)="onOkClick()">Ok</button>
</mat-dialog-actions>
```

Listato 5.10: Codice HTML di "*error-dialog*".

```
export class ErrorDialogComponent implements OnInit {
   title: string
   message: string = null
   htmlMessage: string = null

   constructor(
      private dialogRef: MatDialogRef<ErrorDialogComponent>,
      @Inject(MAT_DIALOG_DATA) data: any) {
      // Logic here...
   }

   onOkClick() {
      this.dialogRef.close(true);
   }
   ...
}
```

Listato 5.11: Codice TypeScript di "*error-dialog*".

Models e Services

Importanti componenti della soluzione proposta, permettono una corretta gestione interna del codice. All'interno del modello vi sono definizioni di concetti trattati nella sezione 3.2, rispecchianti effettivamente la progettazione. Oltre ad essi, vi sono alcuni tipi di dati definiti per facilitare la comunicazione con il *backend*, come quelli relativi ai parametri di input e output delle API definite nella sezione 3.3. L'utilizzo di TypeScript ha permesso una notevole semplificazione dei concetti, in quanto le classi di tale linguaggio possono essere utilizzate quasi direttamente per l'invio delle richieste, senza dover indicare nessuna conversione in formato JSON. Similmente, i dati in ingresso possono essere facilmente trasformabili in oggetti e/o utilizzati anche senza particolare indicazione dell'appartenenza ad una classe. Eventuali tipi relativi più strettamente alle API utilizzate possono essere ritrovati in una sottocategoria (*messages*), nella quale si specificano le *request* e le *response*, nel caso in cui vi sia necessità di specificare tali parametri.

Per quanto riguarda i servizi, essi permettono il richiamo delle funzionalità del *backend*, nascondendo la natura della comunicazione. Possono essere utilizzati all'interno dei *controller* dei *components*, ma anche da altri servizi, favorendo una modularità di tali componenti. Per soddisfare completamente

le possibili richieste effettuabili sul *backend*, vengono predisposti alcuni servizi, con i loro relativi dati:

- `auth.service` - rispecchia le API per l'autenticazione.
- `call.service` - rispecchia le API per l'effettuazione di chiamata. Al suo interno vi sono alcune classi relative ad eventi che potrebbero capitare all'interno di una stanza e chiamata.
- `peerjs.service` - rispecchia le API per l'utilizzo di PeerJS come framework di chiamata. Al suo interno vi sono alcune classi relative ad eventi che potrebbero capitare durante la connessione al server di PeerJS. Rispetto a `call.service`, gli eventi di questo servizio sono più a basso livello, relativi ai singoli `peer` e dipendenti dall'architettura del framework scelto.

Essendo gli ultimi due servizi di rilevante importanza, si preferisce entrare in dettaglio del loro funzionamento collegandolo direttamente all'effettuazione e gestione di una chiamata. La parte relativa a tale approfondimento è disponibile nella sezione 5.3.2.

5.3.2 PeerJS e gestione chiamate

Una sezione di notevole importanza riguarda l'aspetto della creazione di una chiamata. Essendo tale parte il nucleo di tutto lo sviluppo, risulta essere necessario dedicare una particolare analisi alle scelte implementative, riguardanti soprattutto la gestione dei collegamenti multipli tra utenti. PeerJS, infatti, nasce come una libreria P2P che, anche nel tutorial disponibile nella guida ufficiale [26], è predisposta per instaurare un solo collegamento bidirezionale tra due partecipanti. Non trattando in alcun modo la possibilità di instaurare collegamenti multipli, lato *frontend* è stato necessario predisporre una struttura capace di instaurare tali connessioni, ampliando notevolmente il codice visto nelle linee guida.

Per far fronte ad una sfida non banale, è stato predisposto un *service* apposta, `peerjs.service`, il quale raccoglie tutte le funzionalità richieste per far funzionare il sistema, nascondendo la loro vera implementazione al suo interno. Esso provvede all'instaurazione del collegamento con il server di PeerJS, alla ricezione di eventi che possono capitare durante la connessione e alla gestione a basso livello della chiamata di gruppo. È importante sottolineare che la strutturazione del *backend* **non vede in alcun modo** quello che succede

all'interno di una stanza, in quanto sono i singoli `peer` a scambiarsi informazioni, senza passare da un intermediario.

Il servizio predisposto dichiara il riferimento per l'utente "*locale*" (ovvero quello connesso dal dispositivo in utilizzo) e il riferimento per tutti gli utenti connessi a lui. Ogni utente connesso possiede diversi riferimenti a ciò che può succedere durante la connessione tra lui e l'utente locale:

- `mediaConnection` - connessione audio/video (stream);
- `dataConnection` - connessione dati (scambio di messaggi);
- `textconnectionEvents` - eventi relativi alla connessione tra i due `peer`;
- `streamEvents` - eventi relativi allo stream tra i due `peer`;
- `dataEvents` - eventi relativi al canale dati tra i due `peer`

L'inizializzazione del servizio parte con la connessione dell'utente locale al PeerJS Server, grazie alla semplice instaurazione dell'oggetto `Peer`, il quale prende come parametro la configurazione del server (quali l'indirizzo, la sicurezza, la porta e altri aspetti fondamentali per l'instauramento della connessione). Una volta connesso, un *listener* si mette in attesa degli eventi in arrivo dal server stesso: quando ne capita uno, oltre ad aggiornare lo stato interno del servizio, viene emesso un ulteriore evento recepito da tutti i *listener* relativi a quel canale di ascolto. Quest'ultimo è istanziato da `call.service` che provvede al richiamo dell'inizializzazione e al passaggio dello stream audio/video dell'utente, come da figura 3.22. La connessione al server ritorna all'utente un suo identificativo all'interno del sistema, che viene salvato.

```
// Reference to "me" as user in this particular call
private localUser: LocalUser = null

// Reference to a particual user inside the call, having media/data
    connection and connection/stream events
private remoteUsers = new Map<string, PeerConnectionData>()

public initialize(localMediaStream: MediaStream):
    Observable<PeerJsEvent> {
   this.localUser = new LocalUser(
      localMediaStream,
      new EventEmitter(),
      new Peer(this.configService.getConfig().peerjs)
   )

   this.localUser.peerConnection.on('open', id => {
```

```
      this.connected = true;
      this.localUser.peerId = id
      this.localUser.eventEmitter.emit(new
          PeerJsEvent(PeerJsEventType.Connected))
   })
}
```

Listato 5.12: Inizio del codice di `peerjs.service`.

Seguendo l'esecuzione del diagramma 3.16, l'evento `Connected` viene recepito da `call.service`, che provvede a richiedere la stanza e i partecipanti di quella stanza. L'esecuzione può proseguire soltanto se tutti i parametri sono correttamente ottenuti, altrimenti viene catturata l'eccezione che ha causato il problema: potrebbe trattarsi di un errore remoto (*RestError*) oppure locale; in entrambi i casi non manca la gestione di questi aspetti, soprattutto per quanto riguarda il primo.

```
joinRoom(room: Room) {
   // Room cannot be null: exit immediatly
   if (room == null) {
      this.roomEventEmitter.emit(new RoomEvent(
         RoomEventType.Error,
         new RoomError("Invalid Parameter", "The room has no value.
             Please validate the room data and try again.")))
      return
   }

   this.room = room

   this.peerjsService
   .initialize(this.localMediaStream.forRemote)
   .subscribe(peerJsEvent => {
      switch(peerJsEvent.peerJsEventType) {
         case PeerJsEventType.Connected:
            // My connection to PeerJs is stable, proceed
            this.manageLocalUserConnected();
            break

         case PeerJsEventType.Disconnected:
         ...
}
```

Listato 5.13: Inizio del codice relativo all'unione di una stanza di `call.service`.

```
private manageLocalUserConnected() {
   this.localParticipant.peerId = this.peerjsService.getPeerId()

   this.joinRoomRequest(
      this.room.id,
      this.localParticipant.peerId,
      this.localParticipant.user.username).toPromise()

   .then(room => {
      this.room = room
      return this.getParticipantsOfRoom(this.room.id).toPromise()
   })

   .then(participants => {
      // If I'm present inside the list, add preview of my video and
          call participants
      if (participants.find(participant => participant.user.username
          == this.localParticipant.user.username)) {
         this.participants = participants
         // My video
         this.addVideoStream(this.localParticipant.peerId,
             this.localMediaStream.forPreview, true)
         this.callParticipants(participants)
         this.roomEventEmitter.emit(new
             RoomEvent(RoomEventType.ConnectedToRoom))
      }
      else {
         // Leave this room: I'm not inside the participant list
         this.leaveRoom(RoomLeaveType.Error)
            this.roomEventEmitter.emit(new RoomEvent(
            RoomEventType.Error,
            new RoomError("Invite Requested", "Unable to join the
                room: user is not invited in this room.")))
      }
   })
}
```

Listato 5.14: La connessione al server di PeerJS fa scattare ulteriori richieste al *backend*.

Strettamente parlando, quindi, non si tratta di una semplice stanza che aggrega gli utenti: infatti, ognuno dei partecipanti apre un canale bidirezionale con ogni componente del collegamento. Come dimostrato in figura 3.2, questo

può portare presto ad un utilizzo eccessivo di risorse e degradare le performance dell'applicazione in maniera non banale. La connessione con ogni membro passa nuovamente da `peerjs.service`, il quale si occupa di istanziare tutte le risorse necessarie per ogni `peer` chiamato.

```
private callParticipants(participants: Participant[]) {
   participants.forEach(participant => {
      if (participant.user.id != this.localParticipant.user.id &&
          participant.peerId != this.localParticipant.peerId) {
         this.peerjsService.openCall(participant.peerId)
      }
   })
}
```

Listato 5.15: La chiamata verso ogni utente lato `room.service`.

```
public openCall(peerId: string) {
   if (peerId != this.localUser.peerId) {
      this.remoteUsers.set(peerId, new PeerConnectionData(15000, ()
          => {
         // If after 15s still there is no answer, this user is dead
         let remoteUser = this.remoteUsers.get(peerId)
         if (remoteUser != null) {
            if (remoteUser.isPending) {
               this.handlePeerStatus(peerId,
                   PeerConnectionStatus.Unreachable)
            }
         }
      }))

      // Call user and save reference to media connection
      let mediaConnection =
          this.localUser.peerConnection.call(peerId,
          this.localUser.mediaStream);
      this.remoteUsers.get(peerId).mediaConnection = mediaConnection

      // Init media and data connection
      this.initializeMediaAndDataConnection(peerId)
   }
}
```

Listato 5.16: La chiamata verso ogni utente lato `peerjs.service`.

La corretta instaurazione della chiamata prevede che ognuno dei `peer` abbia correttamente effettuato una connessione al canale audio/video e al canale dati di ognuno dei partecipanti chiamato. Come già definito all'inizio di questa sezione, questo è gestito internamente al `peerjs.service`, il quale (per ogni collegamento eseguito con successo) notifica gli osservatori dell'avvenuto, specificando anche l'identificativo dell'utente appenna connesso.

```
private initializeMediaAndDataConnection(peerId: string) {
   this.initializeMediaConnection(peerId).pipe(take(1)).toPromise()
   .then(mediaResult => {
      if (!mediaResult) {
         // Error
      } else {
         this.remoteUsers.get(peerId).dataConnection =
             this.localUser.peerConnection.connect(peerId,
             <Peer.PeerConnectOption>{serialization: 'json'})
         return this.initializeDataConnection(peerId).pipe(take(1))
         .toPromise()
      }
   })
   .then(dataResult => {
      if (!dataResult) {
         // Error
      } else {
         this.connectionOpened = true;
         this.remoteUsers.get(peerId).isPending = false
         this.handlePeerStatus(peerId, PeerConnectionStatus.Opened);
      }
   })
   ...
}

private handlePeerStatus(peerId: string, peerConnectionStatus:
    PeerConnectionStatus) {
   if (peerConnectionStatus == PeerConnectionStatus.Opened) {
      this.localUser.eventEmitter.emit(new
          PeerJsEvent(PeerJsEventType.RemoteUserConnected, {
         peerId: peerId
      }))
   }
   ...
}
```

Listato 5.17: Solo se il collegamento è istanziato è possibile proseguire.

La connessione e disconnessione degli utenti è quindi gestita a basso livello da `peerjs.service`, ma gli eventi da esso scaturiti sono osservati direttamente da `call.service`, che reagisce opportunamente ai cambiamenti di stato. Una banale aggiunta o rimozione del video è di conseguenza gestita grazie al riferimento al componente della view, nel quale il `call.service` è libero di agire.

```
manageRemoteUserConnected(peerId: string) {
   const observedStream = this.peerjsService.getUserStream(peerId)
   // Subscribe to future events
   observedStream.subscribe(mediaStream => {
      this.addVideoStream(peerId, mediaStream)
   })
   // For events arrived before subscription
   if (observedStream.value != null) {
      this.addVideoStream(peerId, observedStream.value)
   }
}

manageRemoteUserDisconnected(peerId: string) {
   this.removeVideoStream(peerId)
}

private addVideoStream(title: string, mediaStream: MediaStream,
    isSelfVideo: boolean = false) {
   this.videosContainer.addVideo(title, mediaStream, isSelfVideo)
}

private removeVideoStream(title: string) {
   this.videosContainer.removeVideo(title)
}
```

Listato 5.18: La gestione dei video interna al `call.service`.

Per quanto riguarda le criticità analizzate nella sezione 3.5.1, è stato deciso di semplificare la gestione di tali aspetti, rimandando ad eventuali futuri sviluppi per la loro correzione. Il modo in cui si affronta una delle problematiche, ovvero quello degli utenti non più disponibili, è molto semplice: alla connessione di un nuovo utente, se quest'ultimo rileva che è impossibile instaurare un collegamento verso un `peer` manda un messaggio al *backend* chiedendo la sua rimozione dalla chiamata e avvisando nello stesso tempo tutti i partecipanti di questo avvenimento. Tale operazione ha diverse criticità da correggere, ma al momento, nella versione prototipale, risulta comunque funzionante nella maggior parte dei casi.

```
private askForParticipantRemove(peerId: string) {
   this.signalUserUnreachable(peerId).toPromise()
   .then(participants => {
      this.participants = participants
      this.peerjsService.resetConnectionWith(peerId)
   })
}

private signalUserUnreachable(peerId) {
   let participant = this.participants.find(p => p.peerId == peerId)

   if (participant != null) {
      return this.http.post<Participant[]>(
      `${this.baseUrl}/${this.BaseRoute}/${this.room.id}/participants/
      ${participant.user.id}/${peerId}/not-reachable`, { })
      .pipe(
         take(1),
         catchError(this.handleError)
      )
   }
}
```

Listato 5.19: Rimozione forzata di un utente nel caso in cui non si riesca a instaurare un collegamento con esso.

5.4 Test

In ultima fase, si vuole ripercorrere tutta l'analisi effettuata, compresi i casi d'uso, per verificare se una tale implementazione possa soddisfare i requisiti. Nelle successive fasi si segue il flusso naturale dell'applicazione: esso determina anche il soddisfacimento dei requisiti, man mano ripercorsi durante il test.

L'avvio dell'applicazione procede utilizzando un processo per ogni microservizio disponibile; viene inoltre lanciata l'istanza dei due database in Docker, il server di PeerJS e il *frontend*. Successivamente al lancio, vi è necessità di recarsi sull'indirizzo specificato per poter visitare il sito web costruito, nel quale appare inizialmente soltanto la schermata di login. Dopo aver inserito delle credenziali predefinite, si apre la principale schermata dell'applicazione.

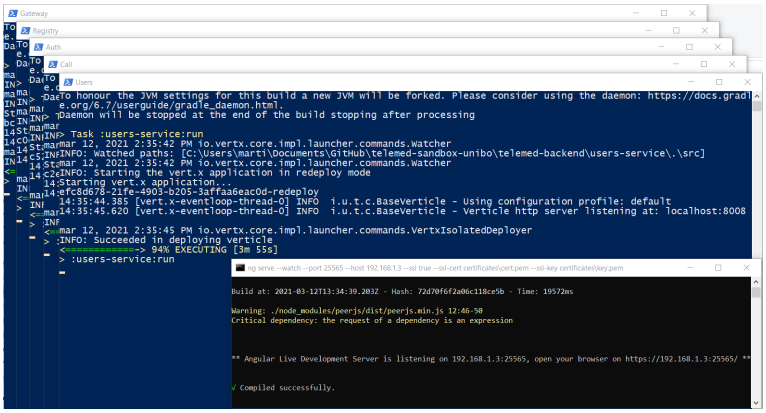


Figura 5.1: Il lancio dell'applicazione (omettendo i servizi lanciati su Docker).

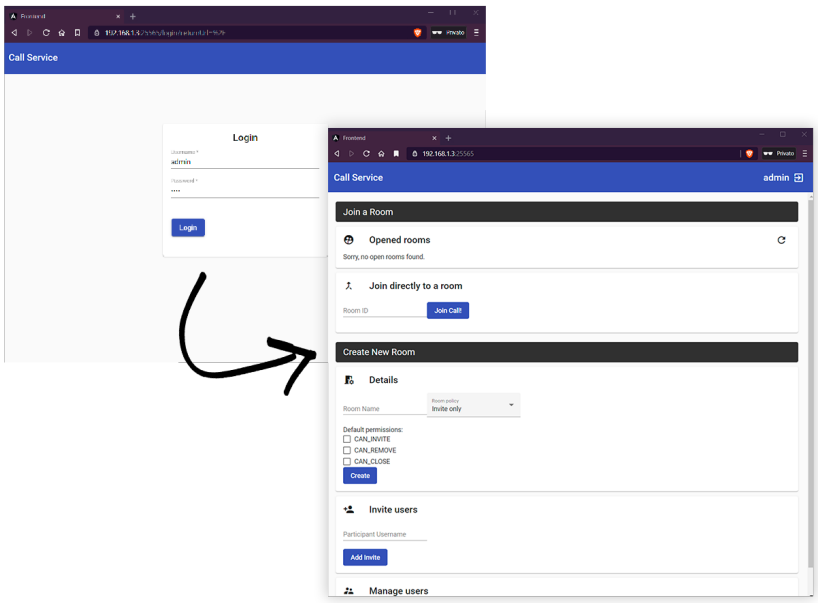


Figura 5.2: Passaggio dalla schermata login a quella di home.

Dalla schermata iniziale si possono vedere le stanze aperte e si può creare una stanza, specificandone i parametri ed invitando utenti. A scopo dimostrativo, si procede alla creazione di una stanza pubblica con il nome "*RoomTest1*" e tutti i permessi di default negati. Essendo una stanza visibile a tutti, è possibile accedere con un secondario account su un dispositivo mobile e verificare se tale affermazione sia vera.

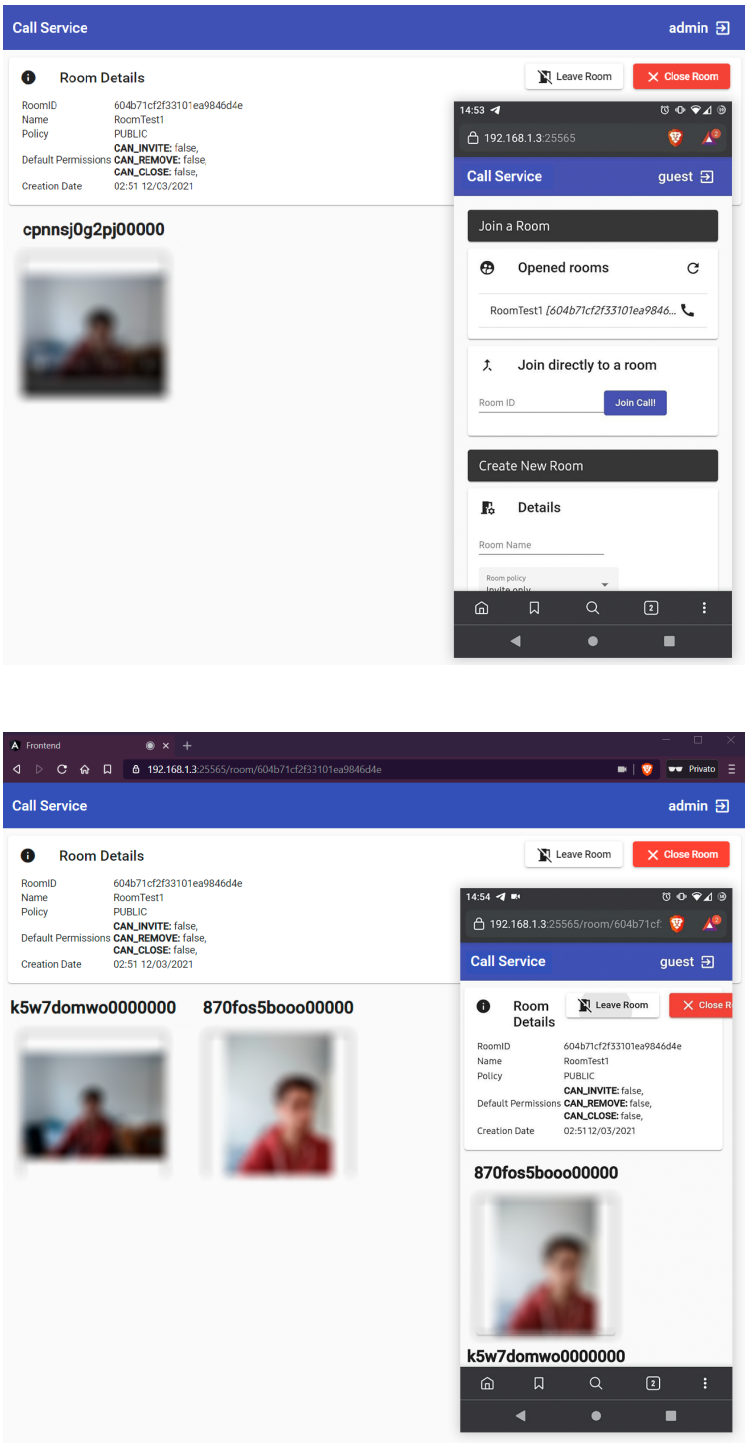


Figura 5.3: La chiamata risulta essere attiva ed è visibile dal dispositivo mobile.

Le maggiori criticità rilevate durante il test (oltretutto anticipate nell'analisi dell'architettura stessa) riguardano la difficoltà del sistema di reggere più di 4 connessioni, dopo le quali le performance venivano degratate in maniera rilevabile, ma non impedendo la prosecuzione dell'incontro. Il quinto `peer` segnava il limite di tale architettura, rendendo la soluzione basata su PeerJS non perfettamente adatta ad ambienti con un numero elevato di partecipanti. Tale aspetto, grazie alla modularità del sistema, potrà essere revisionato agevolmente in un prossimo aggiornamento.

5.4.1 Soddisfacimento Requisiti

Si può di conseguenza confermare che i seguenti requisiti, trattati nella sezione 2.1.1, possono essere ritenuti soddisfatti:

- 1.1 - completamente;
- 1.2 - può essere migliorato il punto 1.2.4;
- 1.3 - può essere migliorato il punto 1.3.2;
- 2.1 - completamente;
- 2.2 - completamente;
- 2.3 - possono essere migliorati i punti 2.3.1 e 2.3.2;
- 4.1 - completamente;
- 4.2 - completamente;
- 4.3 - completamente;

Per quanto riguarda la sezione 3, vi sono da notare particolari aspetti relativi all'architettura. La sua progettazione ha permesso l'individuazione di alcuni moduli principali, i quali possono essere ulteriormente isolati e utilizzati anche in contesti diversi, soddisfacendo sia il requisito 3.1 che il 3.2. Inoltre, come si può notare dagli *screenshot* dell'applicazione reale, l'interfaccia grafica è minimale ma permette un'interazione essenziale con i partecipanti della chiamata, per cui anch'esso può essere ritenuto soddisfatto.

Per quanto riguarda i punti migliorabili, si tratta essenzialmente di piccoli accorgimenti implementativi, sia lato *frontend* che *backend*. La loro corretta progettazione ha permesso una composizione graduale dell'applicazione durante lo sviluppo e il perfezionamento di tali parti non richiede un eccessivo sforzo.

5.5 Contrapposizione Servizio Televisita

Una buona progettazione dell'architettura del "*Call Service*" (e la sua implementazione prototipale) offrono una solida base per il Sistema di Televisite. A causa dell'elevato effort nello studio e organizzazione di tutte le parti relative a questo progetto, non vi è stato modo di proseguire con l'implementazione di un proof-of-concept calato più specificatamente nell'ambito telemedico. Si è potuto constatare però che la realizzazione di "*Call Service*" ha perfettamente rispecchiato l'architettura prevista in tutte le sezioni relative alla sua strutturazione, ottenendo oltretutto molteplici spunti per diverse migliorie del codice finale.

Di conseguenza, è possibile immaginare un fluido proseguimento nell'implementazione del servizio relativo all'ambito sanitario, seguendo le linee tracciate già dal prototipo realizzato: è soltanto necessario implementare i nuovi servizi e predisporre l'interfaccia grafica, considerando il modello a disposizione. Il modulo delle videochiamate verrà quindi inglobato all'interno della soluzione finale, in modo tale da usarlo in maniera completamente trasparente.

Si prevede quindi una strutturazione del tutto simile a quella già affrontata: lato *backend*, ulteriori *verticle* verranno aggiunti per implementare i nuovi servizi richiesti in campo della Televisita. Oltre ad essi, verrà modificato il servizio relativo agli utenti, per permettere una coerente e unica fonte di informazioni per quanto li riguarda. L'aggiunta del "*Notification Serivce*" potrà portare ad eventuali considerazioni relative al suo utilizzo anche per estendere le capacità del "*Call Service*", nel quale le notifiche non erano un requisito, ma risultano essere decisamente comode per informare degli eventi capitati all'interno del servizio.

Successivamente, lato *frontend*, si procederà all'aggiunta di ulteriori componenti per gestire aspetti puramente grafici e all'aggiunta di servizi responsabili a contattare il *backend*. Le parti relative al "*Call Service*" verranno di conseguenza raggruppate in un unico modulo Angular, capace di offrire le funzionalità richieste, nascondendo il funzionamento effettivo.

Conclusioni

La progettazione di sistemi complessi che richiedono diverse sfide è una capacità non banale: con questa tesi, avente come obiettivo quello di progettare e sviluppare soluzioni interoperabili e modulari, si è potuto constatare l'effettiva difficoltà di tale percorso. Nonostante un'elevata richiesta di abilità da parte di uno studente nell'affrontare una tale realizzazione, si è potuto constatare come una buona analisi e un'attenta progettazione architetturale si rivelino una robusta base dalla quale poi si può procedere con la vera implementazione. La necessità di considerare numerosi aspetti caratterizzanti una soluzione complessa, nonchè il rientro nei limiti posti dai requisiti, può trasformarsi in una non triviale attività di ricerca della perfezione, ottenuta non solo a livello della realizzazione, ma anche della sola progettazione di tali architetture.

Lo sforzo iniziale per approfondire argomenti in un ambito completamente nuovo e sconosciuto, causato *anche* dai vincoli iniziali, si è rivelato in realtà un eccellente modo per definire ciò che riguarda il modello del dominio di entrambe le soluzioni. Avendo una corretta impostazione del problema, è stato del tutto naturale proseguire con ulteriori approfondimenti del problema stesso, in pieno rispetto dei contesti analizzati. Il risultato che ne deriva è un'eccellente architettura, basata su moderne soluzioni miranti ad essere il più possibile conformi allo standard dell'Industria 4.0.

La sfida affrontata offre di conseguenza un modello che copre, in ogni suo punto essenziale, la realizzazione di sistemi modulari: da una parte il "*Call Service*", capace di poter essere calato in ambito sanitario, dall'altra il "*Servizio di Televisite*", costruito direttamente sulla base del primo. Le parti che compongono entrambi i sistemi, progettate in vista di questo aspetto, possono essere adattate anche in ulteriori ambiti, grazie alla predisposizione dei microservizi (con Vert.X) e di Angular a creare applicazioni capaci di affrontare questo tipo di problematiche. Una corretta strutturazione architetturale di entrambi i sistemi ha permesso di avere in mano un ottimo esempio di isolamento delle funzionalità, riusabile in ulteriori contesti.

È inoltre di indubbio valore anche la standardizzazione di tali soluzioni: non si trattava, infatti, soltanto di definire un'API capace di essere sfruttata indipendentemente dal client, ma di offrire la capacità di utilizzo di un'eventuale implementazione di tale servizio in maniera completamente trasparente. L'applicazione di eventuali standard, come quello relativo alla realizzazione di `RestError`, risulta essere un chiaro esempio di come la previsione e progettazione di minuscoli aspetti può rivelarsi di estrema importanza per la realizzazione e riuscita di un buon progetto.

Quest'ultimo aspetto, nonostante la sua rilevante influenza, risulta essere marginale di fronte all'interoperabilità garantita sin dalla fase di progettazione grazie all'applicazione dello standard FHIR. La sua esplorazione prima di calarsi sull'analisi dei requisiti ha permesso di contestualizzare meglio il tema del progetto, rendendo disambigua la terminologia utilizzata nelle successive fasi. La sua applicazione durante l'effettiva progettazione, inoltre, ha reso possibile la concezione di un sistema capace di interoperare con ulteriori soluzioni che applicano lo stesso riferimento. Tutto ciò senza complicare eccessivamente l'organizzazione di tale servizio, grazie alla facilità con la quale è possibile far riferimento soltanto ad una parte dello standard FHIR, senza scendere ai dettagli eccessivamente sovradimensionati previsti per un sistema nettamente più complesso.

Di certo, l'intera organizzazione del progetto non ha soltanto aspetti puramente positivi: vi è infatti del lavoro non banale da svolgere per un prossimo futuro. Nonostante una minuziosa ricerca di dettagli, vi sono criticità non coperte dall'analisi che devono essere rivisitate per poter garantire il corretto funzionamento di entrambe le soluzioni. È infatti da riconsiderare l'approccio alternativo a PeerJS, a causa dei limiti rilevati durante la fase di test. Inoltre, sono da definire in maniera più precisa protocolli relativi alle notifiche e al pagamento per quello che riguarda il "*Servizio di Televisite*". Nonostante queste imperfezioni, il lavoro svolto mostra quanto sia possibile ottenere ponendo particolare attenzione alle singole parti componenti un progetto più grande.

Lo sviluppo effettivo del "*Servizio di Televisite*" è inoltre garantito: infatti, da parte dell'azienda, che ha dato origine a tutto il progetto, è possibile notare un elevato entusiasmo per il suo completamento. La volontà di concludere una tale sfida è risentita anche all'interno del *working group* che vi ha lavorato. In un prossimo futuro è prevista quindi la continuazione di tale sviluppo, in modo da mettere completamente a disposizione il prima possibile un prototipo per quello che risulta essere, al giorno di oggi, uno scenario pandemico con diverse carenze nell'ambito della sanità digitale.

L'architettura prevista e una prima soluzione prototipata soddisfano completamente le aspettative, nonchè i requisiti, posti all'inizio di tutto il percorso. Le varie sfide affrontate durante il lavoro svolto hanno fatto emergere diversi modi per affrontare le problematiche, nonchè ampi spazi di crescita personale e professionale in numerosi ambiti. Lo svolgimento delle attività, sia in autonomia che considerando l'interazione con il *working group*, ha reso noto come l'affrontare uno scenario reale possa sia convergere che divergere dai problemi puramente accademici. Tale evidenza ha richiesto specifiche modalità di ragionamenti che hanno garantito il soddisfacimento degli obiettivi assegnati.

Tutto il lavoro svolto risulta quindi essere il miglior modo possibile per completare un percorso di studi: un progetto con enormi potenzialità, diverse difficoltà e soluzioni non banali, che permette di sfruttare tutte le conoscenze acquisite e di impararne nuove, destinato indubbiamente a continuare in un prossimo futuro.

Ringraziamenti

In primo luogo, si ringrazia tutto il *working group* per il percorso affrontato insieme. Particolari meriti vanno all'azienda ospitante (I-TEL) che ha pienamente coinvolto lo studente nella realizzazione del progetto svolto con l'attiva presenza di Jean Claude Correale e Matteo Orzes, sempre disponibili a chiarire dubbi e proporre interessanti spunti durante i meeting svolti.

Ulteriori ringraziamenti vanno direttamente all'Università di Bologna, la quale attraverso le figure di diversi docenti e assistenti ha permesso l'acquisizione di numerose conoscenze necessarie per affrontare tali sfide. Particolari meriti si devono direttamente al Prof. Alessandro Ricci e al Dott. Ing. Angelo Croatti, che hanno assistito lo studente durante tutto lo sviluppo, sin dai primi giorni del tirocinio. Di certo non sono gli unici a dover essere citati in questa fase, ma sono quelli che hanno maggiormente segnato il percorso universitario dello studente.

Un infinito grazie anche a chi in tutto questo periodo ha dimostrato di esserci: le diverse difficoltà in tutto il percorso universitario non sarebbero state affrontate senza un attivo supporto da parte di Giulia Bianchi e Giacomo Frisoni; menzioni che possono essere estese anche a Simone Di Lorenzo, Giacomo Vignali, Matteo Michelotti, Nadir Santolini e Giacomo Succi, i quali hanno saputo distrarmi nei momenti difficili. Inoltre, una qualsiasi sfida non potrebbe essere superata senza l'aiuto della propria famiglia: nonostante diversi inconvenienti nel frattempo capitati, è riuscita sempre a dimostrarsi pronta a "*mettere i puntini sulle i*".

Infine, si vogliono ringraziare tutte le persone che abbiano contribuito, anche in maniera minima, allo sviluppo di questa tesi e a tutto l'insieme di particolari che caratterizzano oggi in modo imprescindibile l'autore di questa tesi.

Bibliografia

[1] A. Dearden et al. "User-centred design and pervasive health: A position statement from the User-Centred Healthcare Design project". In: *2010 4th International Conference on Pervasive Computing Technologies for Healthcare*. 2010, pp. 1–4. DOI: 10.4108/ICST.PERVASIVEHEALTH2010.8837.

[2] H. Grain. "eHealth - layers to achieve safe, efficient and cost effective solutions to information exchange". In: *2013 3rd International Conference on Instrumentation, Communications, Information Technology and Biomedical Engineering (ICICI-BME)*. 2013, pp. 4–7. DOI: 10.1109/ICICI-BME.2013.6698454.

[3] I. Macía. "Towards a semantic interoperability environment". In: *2014 IEEE 16th International Conference on e-Health Networking, Applications and Services (Healthcom)*. 2014, pp. 543–548. DOI: 10.1109/HealthCom.2014.7001900.

[4] Ministero della Salute. "TELEMEDICINA - Linee di indirizzo nazionali". In: (2014).

[5] J. Favela et al. "Living Labs for Pervasive Healthcare Research". In: *IEEE Pervasive Computing* 14.2 (2015), pp. 86–89. DOI: 10.1109/MPRV.2015.37.

[6] C. Juhra et al. "The health policy guidance and practice of introducing technologies in health system in Europe". In: *2016 Digital Media Industry Academic Forum (DMIAF)*. 2016, pp. 35–36. DOI: 10.1109/DMIAF.2016.7574897.

[7] M. Alloghani et al. "Healthcare Services Innovations Based on the State of the Art Technology Trend Industry 4.0". In: *2018 11th International Conference on Developments in eSystems Engineering (DeSE)*. 2018, pp. 64–70. DOI: 10.1109/DeSE.2018.00016.

[8] M. L. Braunstein. "Health care in the age of interoperability part 5: the personal health record". In: *IEEE Pulse* 10.3 (2019), pp. 19–23. DOI: 10.1109/MPULS.2019.2911804.

[9] Janya Chanchaichujit et al. “An Introduction to Healthcare 4.0”. In: *Healthcare 4.0: Next Generation Processes with the Latest Technologies*. Singapore: Springer Singapore, 2019, pp. 1–15. ISBN: 978-981-13-8114-0. DOI: 10.1007/978-981-13-8114-0_1. URL: https://doi.org/10.1007/978-981-13-8114-0_1.

[10] Chaloner Chute e Tara French. “Introducing Care 4.0: An Integrated Care Paradigm Built on Industry 4.0 Capabilities”. In: *International Journal of Environmental Research and Public Health* 16.12 (2019). ISSN: 1660-4601. DOI: 10.3390/ijerph16122247. URL: https://www.mdpi.com/1660-4601/16/12/2247.

[11] N. Mohamed e J. Al-Jaroodi. “The Impact of Industry 4.0 on Healthcare System Engineering”. In: *2019 IEEE International Systems Conference (SysCon)*. 2019, pp. 1–7. DOI: 10.1109/SYSCON.2019.8836715.

[12] Ward EC. Scriven H Doherty DP. “Evaluation of a multisite telehealth group model for persistent pain management for rural/remote participants”. In: (2019). DOI: https://doi.org/10.22605/RRH4710. URL: https://www.rrh.org.au/journal/article/4710.

[13] J. Al-Jaroodi, N. Mohamed e E. Abukhousa. “Health 4.0: On the Way to Realizing the Healthcare of the Future”. In: *IEEE Access* 8 (2020), pp. 211189–211210. DOI: 10.1109/ACCESS.2020.3038858.

[14] T. Cai et al. “SC4 - Telemedicine and teleconsulting in andrology at the time of COVID-19 pandemic: Is this the right way?” In: *European Urology Open Science* 20 (2020). Abstracts from The Italian Society of Urology 93rd National Congress 2020, S33. ISSN: 2666-1683. DOI: https://doi.org/10.1016/S2666-1683(20)35326-X. URL: https://www.sciencedirect.com/science/article/pii/S266616832035326X.

[15] D. Jakhar, S. Kaul e I. Kaur. “WhatsApp messenger as a teledermatology tool during coronavirus disease (COVID-19): from bedside to phone-side”. In: *Clinical and Experimental Dermatology* 45.6 (2020), pp. 739–740. DOI: https://doi.org/10.1111/ced.14227. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/ced.14227. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/ced.14227.

[16] H. Ren et al. “5G Healthcare Applications In COVID-19 Prevention And Control”. In: *2020 ITU Kaleidoscope: Industry-Driven Digital Transformation (ITU K)*. 2020, pp. 1–4. DOI: 10.23919/ITUK50268.2020.9303191.

[17] Sayed E. Wahezi et al. "Telemedicine and current clinical practice trends in the COVID-19 pandemic". In: *Best Practice & Research Clinical Anaesthesiology* (2020). ISSN: 1521-6896. DOI: https://doi.org/10.1016/j.bpa.2020.11.005. URL: https://www.sciencedirect.com/science/article/pii/S1521689620301105.

[18] Reggie Tiong Saldivar et al. "Goals of care conversations and telemedicine". In: *Journal of Geriatric Oncology* (2021). ISSN: 1879-4068. DOI: https://doi.org/10.1016/j.jgo.2021.02.016. URL: https://www.sciencedirect.com/science/article/pii/S1879406821000400.

Sitografia

[19] *Angular.*
https://angular.io/,
https://angular.io/start.

[20] *FHIR.*
https://www.hl7.org/fhir/.

[21] *FSE - Fascicolo Sanitario Elettronico.*
https://www.fascicolo-sanitario.it/fse/.

[22] *JWT.*
https://jwt.io/,

[23] *Kotlin.*
https://kotlinlang.org/,

[24] *MongoDB.*
https://www.mongodb.com/it,

[25] *OpenVidu.*
https://openvidu.io/.

[26] *PeerJS.*
https://peerjs.com/.

[27] *REST.*
https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods,
https://blog.restcase.com/7-rules-for-rest-api-uri-design/.

[28] *RFC 7807 e guida pratica.*
https://tools.ietf.org/html/rfc7807,
https://www.baeldung.com/rest-api-error-handling-best-practices,

[29] *Ricetta Online.*
https://www.fascicolosanitario.regione.lombardia.it/ricette/.

[30] *TypeScript.*
https://www.typescriptlang.org/,

[31] *Vert.x*.
`https://vertx.io/`,
`https://vertx.io/get-started/`.

[32] *WebRTC*.
`https://webrtc.org/`,
`https://webrtc.org/getting-started/overview`.