Università degli Studi di Camerino

Scuola di Scienze e Tecnologie
Corso di Laurea in Informatica (Classe L-31)

Progettazione e sviluppo di un'applicazione gestionale con architettura a microservizi

Laureando
Conti Luca

Matricola 096761

Relatore
Prof. Marcantoni Fausto

Correlatore
Dott. Azizi Endri

A.A. 2020/2021

Indice

Elenco delle figure

1 Introduzione

In questo capitolo iniziale vengono introdotti il progetto di stage, le sue motivazioni, gli obiettivi che si volevano conseguire e il contesto aziendale in cui si è svolto. Inoltre viene presentata la struttura del documento di tesi.

1.1 Progetto di stage

Il progetto di stage, dal titolo "Hexagonal architecture principles and implementation: disegno e sviluppo di un'architettura a microservizi utilizzando il modello esagonale", si è svolto in due fasi.
La prima è stata una fase di studio delle architetture da implementare e delle tecnologie da utilizzare realizzata anche mediante l'accesso a due corsi online del dott. Nicola La Rocca, incentrati il primo sulla realizzazione di backend tramite il framework SpringBoot ed il secondo sulla creazione di frontend tramite il framework Angular, dai titoli "Java Microservices con Spring Boot, Spring Cloud e AWS" [1] e "Sviluppare Full Stack Applications con Spring Boot e Angular" [2].
La seconda fase è stata l'applicazione delle conoscenze acquisite durante il periodo di studio, concretizzatasi tramite la realizzazione di un'applicazione demo.

1.2 Azienda e ambiente di lavoro

ICCS Informatica S.R.L, società del Gruppo Maggioli, è un'azienda fondata nel 2006 da Rosario Dolce con sede a Matelica.
L'azienda fornisce soluzioni software web oriented per la Pubblica Amministrazione, con particolare attenzione per la realizzazione di servizi innovativi e progetti organizzativi per supportare l'Ente pubblico nella gestione dei servizi welfare, dei servizi a domanda individuale (refezione scolastica, trasporto scolastico, asili nido) e delle attività produttive (SUAP/SUE).
L'organico aziendale è andato crescendo nel tempo, con un importante incremento negli ultimi due anni, in particolare grazie all'assunzione di giovani neolaureati. Di pari passo è aumentato anche il fatturato dell'azienda.

1.3 Obiettivi e motivazioni

L'obiettivo principale dello stage è stato lo studio di tecnologie da implementare nei prodotti software attualmente forniti dall'azienda. La necessità dell'implementazione di un'architettura a microservizi si è resa necessaria dopo il grande aumento di funzionalità che sono state aggiunte ai prodotti a causa del numero crescente di clienti acquisiti, molti dei quali con esigenze diversificate.
Il passaggio al linguaggio Java, ai framework Spring Boot e Angular e ad un database PostgreSQL dal linguaggio PHP e da un database MySQL attualmente in

uso è stata un'esigenza nata dalla necessità di uniformare i software a quelli del Gruppo Maggioli, così da fornire una migliore integrazione dei prodotti, al fine di renderli disponibili in base alle richieste dei clienti.
Il passaggio ad un'architettura a microservizi ha anche il vantaggio di rendere più semplice questa integrazione con software non prodotti e gestiti direttamente dall'azienda.

1.4 Struttura della Tesi

La tesi inizia con una presentazione generale della parte teorica per poi proseguire con il caso di studio, l'implementazione del progetto e terminare con una sezione dedicata alle conclusioni.
In particolare il documento è suddiviso nei seguenti capitoli:

- Capitolo 2: Descrive lo sviluppo delle architetture principali utilizzate nello sviluppo software, in particolare le architetture a microservizi ed esagonale, che sono state usate nel progetto;
- Capitolo 3: Illustra l'applicazione realizzata introducendola con una breve descrizione dei casi d'uso principali per proseguire con la descrizione della struttura del backend e del frontend;
- Capitolo 4: Approfondisce le tecnologie studiate nella prima fase dello stage ed utilizzate nella realizzazione del progetto;
- Capitolo 5: Contiene una breve presentazione delle tecnologie non implementate nel caso di studio, ma che dovranno essere applicate negli sviluppi futuri del progetto, oltre a considerazioni e conclusioni personali.

2 Tipologie di architetture software

Negli ultimi anni le applicazioni, in particolare quelle cloud, sono diventate sempre più grandi e complesse ed è emersa la necessità di renderle scalabili ed in grado di evolversi rapidamente. Queste esigenze hanno portato un cambiamento anche nelle architetture alla base dei software, con una progressiva evoluzione che si può dividere in quattro stati: architettura monolitica, architettura a strati, architettura orientata ai servizi e architettura a microservizi [3].
Particolare attenzione verrà dedicata anche all'architettura esagonale, proposta più recentemente come alternativa all'architettura a strati [4] [5].

2.1 Architettura monolitica

Inizialmente le applicazioni venivano sviluppate e distribuite come una singola entità: queste hanno il grande vantaggio di essere di facile implementazione in quanto hanno una sola codebase e tipicamente vengono distribuite in un unico pacchetto. Questo genere di architettura si presta bene per applicazioni piccole o poco soggette a cambiamenti, ma non è adatto per applicazioni complesse o in rapida evoluzione. In quest'ultimo caso possono diventare problematiche per dimensioni e complessità, rendendo difficilmente gestibili le fasi di sviluppo, test e implementazione. Uno sviluppatore deve avere conoscenza di tutta l'applicazione, o almeno gran parte di essa, rendendo problematico il mantenimento in caso di turnover del personale e lenta l'introduzione di nuove persone nel team. Inoltre ogni modifica, anche minima, può avere ripercussioni su tutta l'applicazione, rendendo quindi necessario un test completo prima di essere distribuita in produzione. Infine l'unico modo di poter scalare un'applicazione monolitica è quello di replicarla, operazione costosa sia dal punto di visto di investimento che di risorse necessarie.

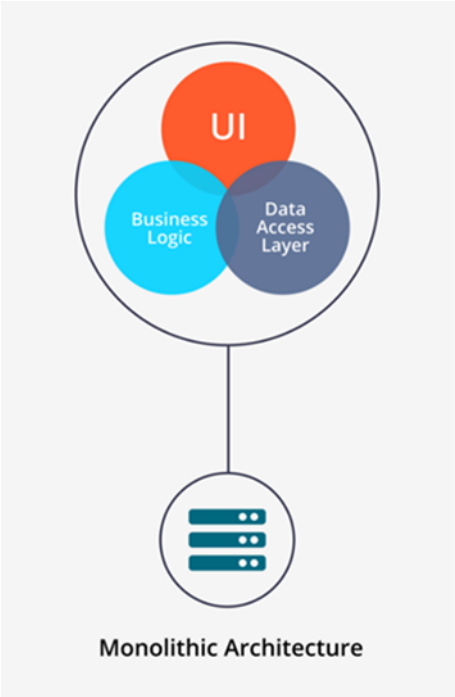

Figura 1: Schema dell'architettura monolitica [6]

2.2 Architettura a strati

I difetti dell'architettura monolitica hanno portato gli sviluppatori a cercare approcci diversi, in particolare con il principio della scomposizione, spostando così il mercato verso architetture a strati.
L'architettura multi-tier è un approccio in cui le varie funzionalità del software vengono separate a livello logico, ovvero suddivise su più strati differenti in comunicazione fra loro, in cui solo gli strati superiori possono effettuare richieste a quelli inferiori. Solitamente gli strati sono tre (architettura three-tier):

- livello di presentazione: è il livello più alto, l'interfaccia utente che mostra i dati al consumatore;
- livello di logica di business: è il nucleo dell'applicazione, in quanto permette lo svolgimento delle funzionalità effettuando decisioni, calcoli e processando i dati che riceve e inoltra agli altri due strati;
- livello di accesso ai dati: è il livello responsabile per l'interazione con le fonti dei dati (solitamente dei database).

Grazie a questo modello è possibile sostituire o aggiornare uno strato senza dover apportare modifiche agli altri. Con il crescere di dimensioni di un'applicazione di questo tipo si ripropongono i problemi caratteristici dell'architettura monolitica, con la differenza che in quella a strati solitamente sono presenti solo nel livello di logica di business.

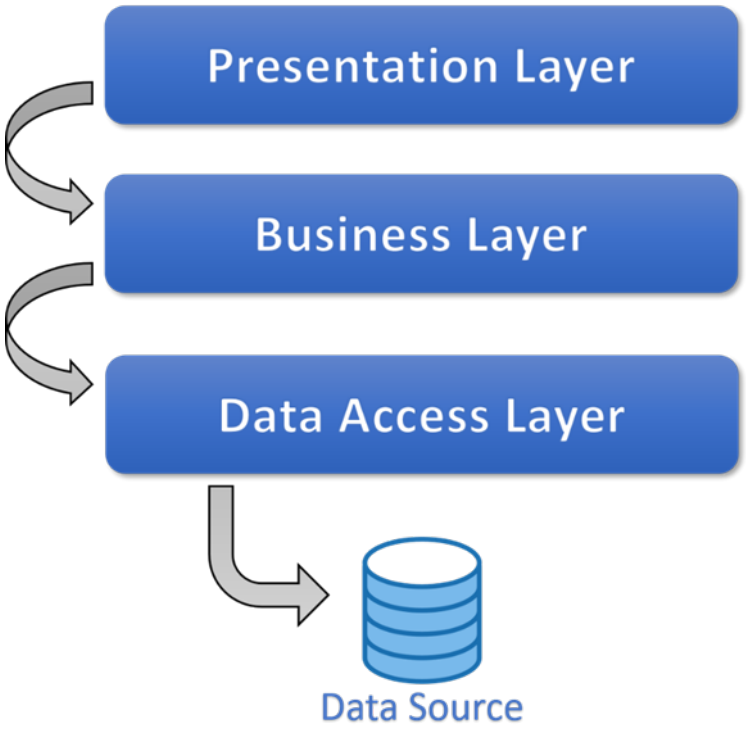

Figura 2: Schema dell'architettura a strati [7]

2.3 Architettura orientata ai servizi

Il passo successivo è stato quindi quello di scomporre le applicazioni in base alle funzionalità di business piuttosto che a livello logico. Con questo tipo di architettura si hanno vantaggi in termini di scalabilità e di una maggiore semplicità in quanto si hanno servizi separati e quindi potenzialmente più piccoli e facili da gestire.
Questo modello non ha avuto però molto successo a causa di inutili astrazioni e protocolli legacy complessi: si è presentato il bisogno di far comunicare applicazioni con tecnologie e protocolli diversi e per fare questo era necessario implementare un ulteriore livello usato per la comunicazione, l'Enterprise Service Bus, rendendolo inadatto alla dinamicità della tecnologia e del mercato attuale.

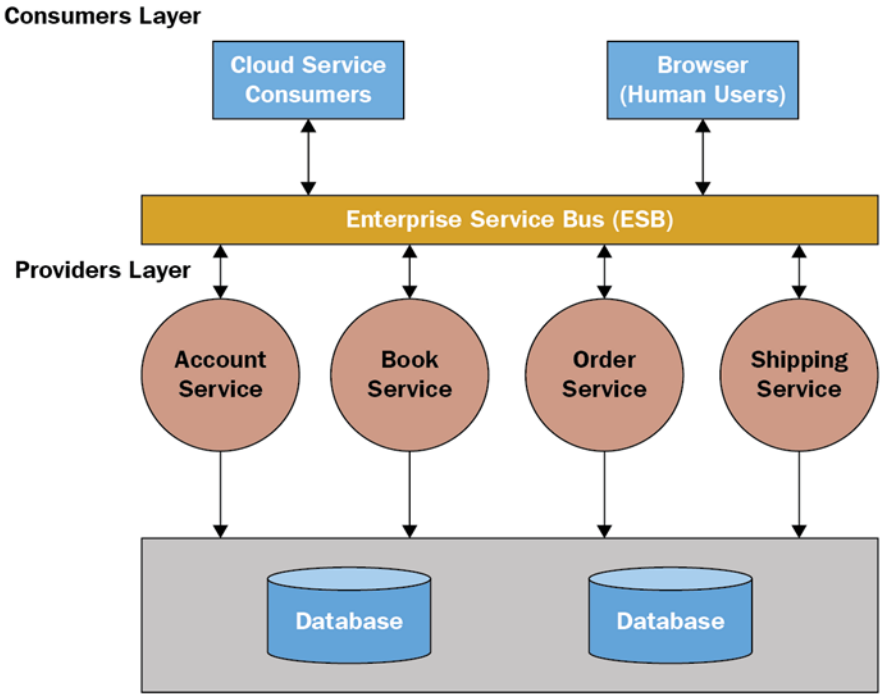

Figura 3: Schema dell'architettura orientata ai servizi [8]

2.4 Architettura a microservizi

Come nel caso precedente, anche questa architettura opera una divisione a livello funzionale. La differenza sostanziale fra un'architettura orientata ai servizi e un'architettura a microservizi è che la prima tende a far interagire n applicazioni mentre la seconda mira alla realizzazione di una singola applicazione composta da n servizi sviluppati in maniera indipendente secondo il principio della singola responsabilità.
Un'ulteriore differenza è nella comunicazione tra i servizi, in quanto nell'architettura a microservizi è solitamente basata sul protocollo HTTP tramite API RESTful.
I vantaggi principali di questo tipo di architettura sono:

- applicazioni altamente scalabili: man mano che la domanda per determinati servizi aumenta, i microservizi possono essere distribuiti su più server e infrastrutture, in base alle esigenze aziendali;
- facilità di mantenimento, correzione e aggiornamento in quanto i servizi vengono sviluppati e distribuiti indipendentemente;
- eliminazione dei singoli punti di guasto: è difficile che un problema possa riflettersi sull'intero sistema, solitamente è possibile isolare il servizio difettoso fino al momento della riparazione senza compromettere le funzionalità dell'applicazione;
- time-to-market più rapido: consentendo di abbreviare i cicli di sviluppo, un'architettura basata su microservizi supporta deployment e aggiornamenti più agili;

- accessibilità: per gli sviluppatori è molto più semplice comprendere, aggiornare e migliorare tali componenti permettendo di accelerare i cicli di sviluppo;
- maggiore apertura: grazie alle API indipendenti dal linguaggio, gli sviluppatori sono liberi di scegliere la tecnologia e il linguaggio ottimali per la funzione da creare.

Gli svantaggi principali sono invece:

- compilazione più complessa: occorre dedicare tempo all'identificazione delle dipendenze fra i servizi. A causa di tali dipendenze, quando si esegue una compilazione può essere necessario eseguirne anche molte altre;
- comunicazione tra i servizi: è necessario un modo efficace per farli comunicare. Scambiarsi dati sulla rete introduce latenza e potenziali fallimenti, che possono interferire con l'esperienza dell'utente;
- garantire la coerenza dei dati: più database replicati e lo scambio costante di dati può facilmente portare a incoerenze. Senza l'uso di meccanismi adeguati, implementare meccanismi di comunicazione tra processi per i casi d'uso che si estendono su più servizi senza l'utilizzo di transazioni distribuite è difficile;
- gestione delle versioni: l'aggiornamento a una nuova versione potrebbe compromettere la retrocompatibilità con gli altri servizi;
- deployment: occorre investire notevolmente in soluzioni di automazione. La complessità dei microservizi renderebbe il deployment manuale estremamente difficile;
- test: mentre testare un singolo servizio diventa una cosa semplice, non è altrettanto semplice implementare i test di integrazione. Anche qui è necessario automatizzare il più possibile e questo richiede un impegno non indifferente.

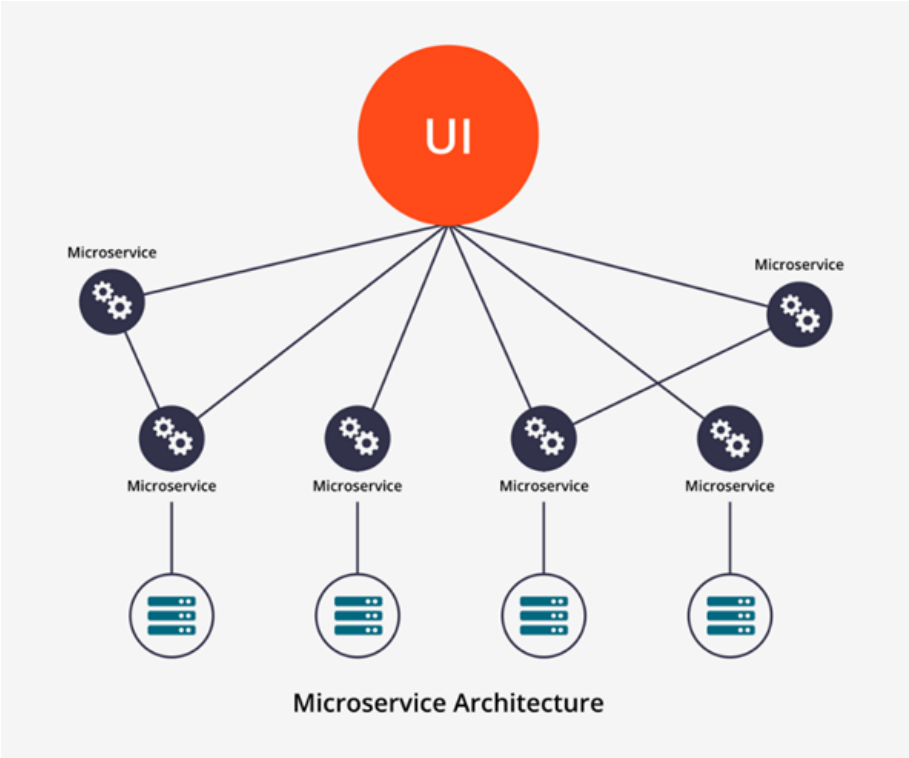

Figura 4: Schema dell'architettura a microservizi [6]

2.5 Architettura esagonale

L'architettura esagonale (o architettura Port and Adapter) è nata con l'obiettivo di creare un'applicazione disaccoppiabile e di evitare quindi dipendenze tra strati. L'idea alla base è di rappresentare i livelli in modo concentrico e non come una pila: al centro viene posto il core dell'applicazione, ovvero gli oggetti di dominio che rappresentano la logica e le regole di business. All'interno del core non ci sono riferimenti espliciti a dettagli tecnologici che vengono invece situati all'esterno.
Intorno al core si trovano le porte che fungono da confine tra gli oggetti del dominio e l'esterno: esse sono rappresentazioni astratte di un servizio e permettono il disaccoppiamento tra il core domain e i servizi esterni.
Le porte possono essere di due tipi: inbound port, che espongono il core all'esterno con un'interfaccia che può essere invocata da componenti esterni, e outbound port, che permettono l'accesso a funzionalità esterne da parte del core, come ad esempio la comunicazione con un database.
In un livello più esterno si trovano gli adattatori, cioè l'implementazione delle relative porte, che permettono la comunicazione con il mondo esterno. Essi si dividono in primary o input adapters se usano un'inbound port e secondary o output adapters se implementano un'outbound port.

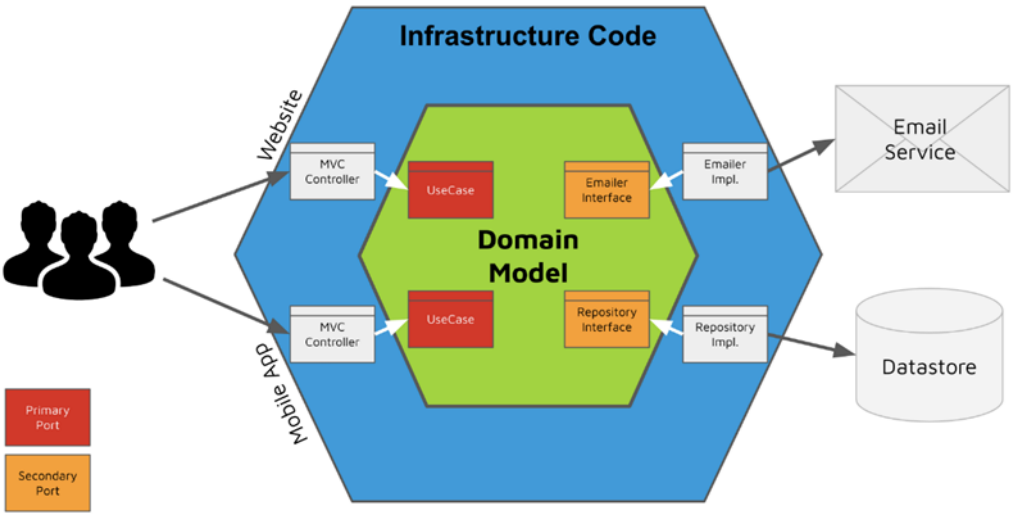

Figura 5: Esempio di architettura esagonale [9]

3 Caso di studio

L'applicazione delle conoscenze acquisite si è concretizzata nella realizzazione di un'applicazione full stack per la gestione di un servizio di anagrafica. I casi d'uso principali sono quelli relativi alla visualizzazione, creazione, eliminazione e modifica delle anagrafiche, inoltre è possibile aggiungere enti, nazioni, città e cittadinanze nel caso in cui essi non siano presenti fra quelli disponibili. Alcune delle operazioni sono disponibili solo se si dispone di un determinato tipo di autorizzazione.

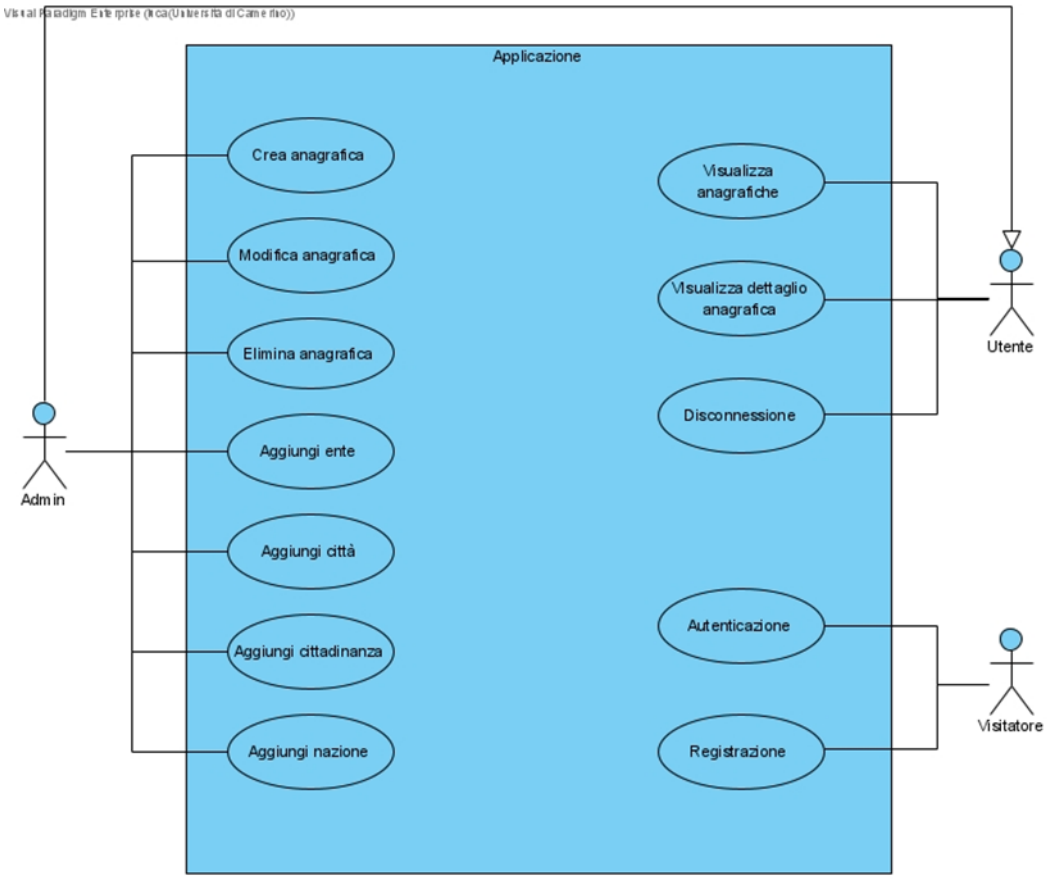

Figura 6: Diagramma dei casi d'uso dell'applicazione

In un primo momento è stato realizzato il backend seguendo i principi dell'architettura a microservizi e di quella esagonale per i singoli servizi, in seguito è stata realizzata un'applicazione frontend con Angular per simulare almeno in parte la soluzione che in futuro verrà utilizzata anche in azienda.

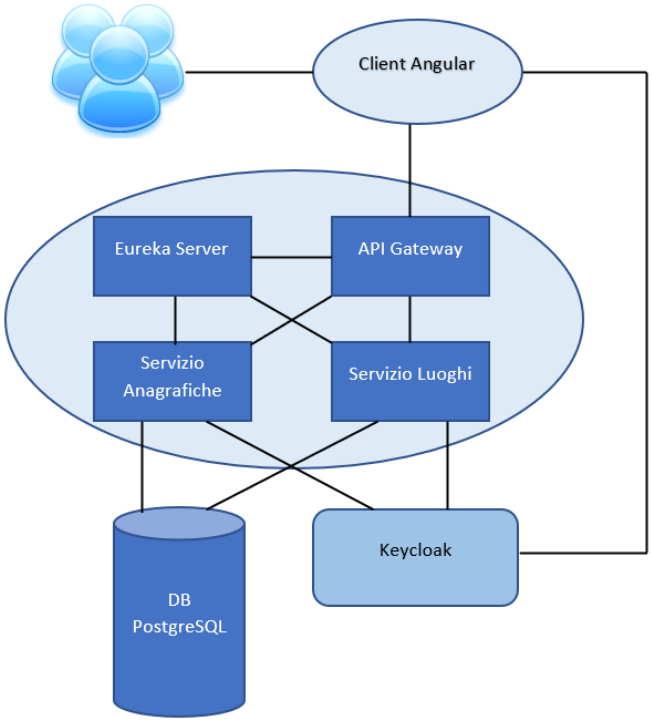

Figura 7: Schema del progetto

3.1 Backend

Il backend è composto da quattro microservizi che interagiscono fra loro, realizzati utilizzando il framework Spring Boot. Il codice sorgente è disponibile su GitHub [10].

3.1.1 Microservizi principali: AnagraficaUtenti e LuoghiService

Oltre ai servizi di utilità, come Gateway ed EurekaServer, nella fase iniziale il backend era stato concepito come un unico microservizio (chiamato AnagraficaUtenti) che andasse ad interagire con il database: questa decisione era stata presa perché la parte iniziale dello stage era incentrata sullo studio dell'architettura esagonale e si era quindi deciso di concentrarsi sullo strutturare il servizio su quel modello. Quando si è arrivati allo studio dell'architettura a microservizi è stato modificato il microservizio AnagraficaUtenti e creato un secondo microservizio (inizialmente chiamato ResidenzaService e poi LuoghiService), che si occupasse in un primo momento della sola gestione dei dati relativi alla residenza degli utenti, funzionalità inizialmente gestita dal servizio AnagraficaUtenti, e infine dei dati relativi a qualsiasi tipo di luogo, incentrando così i due microservizi finali su due domini ben distinti, quello delle persone e quello dei luoghi.

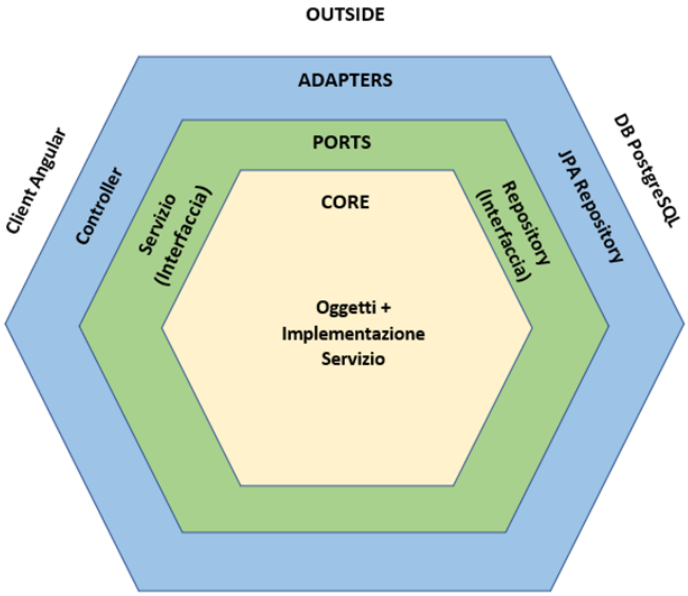

Figura 8: Struttura di base dei microservizi

3.1.2 Altri microservizi: Gateway e EurekaServer

I microservizi Gateway e EurekaServer sono, almeno per quanto riguarda le funzionalità di base, comuni a qualsiasi infrastruttura a microservizi. Verranno trattati nella parte relativa alle tecnologie utilizzate, in quanto allo stato attuale del progetto sono microservizi abbastanza basilari, la cui importanza è data proprio dalle dipendenze che usano: Spring Cloud Gateway e Netflix Eureka.

3.2 Frontend

Il frontend permette di visualizzare i dati presenti in anagrafica e di modificarne, eliminarne e crearne alcuni. Permette inoltre di registrare e di autenticare un utente, prerequisito necessario per effettuare alcune delle operazioni sui dati.
Questo componente è stato sviluppato come una web app, accessibile come una semplice pagina web tramite un qualsiasi browser. Per realizzarlo è stato utilizzato Angular, un potente framework che permette la creazione di Single Page Application (SPA).
Anche il codice del frontend è disponibile su GitHub [11].

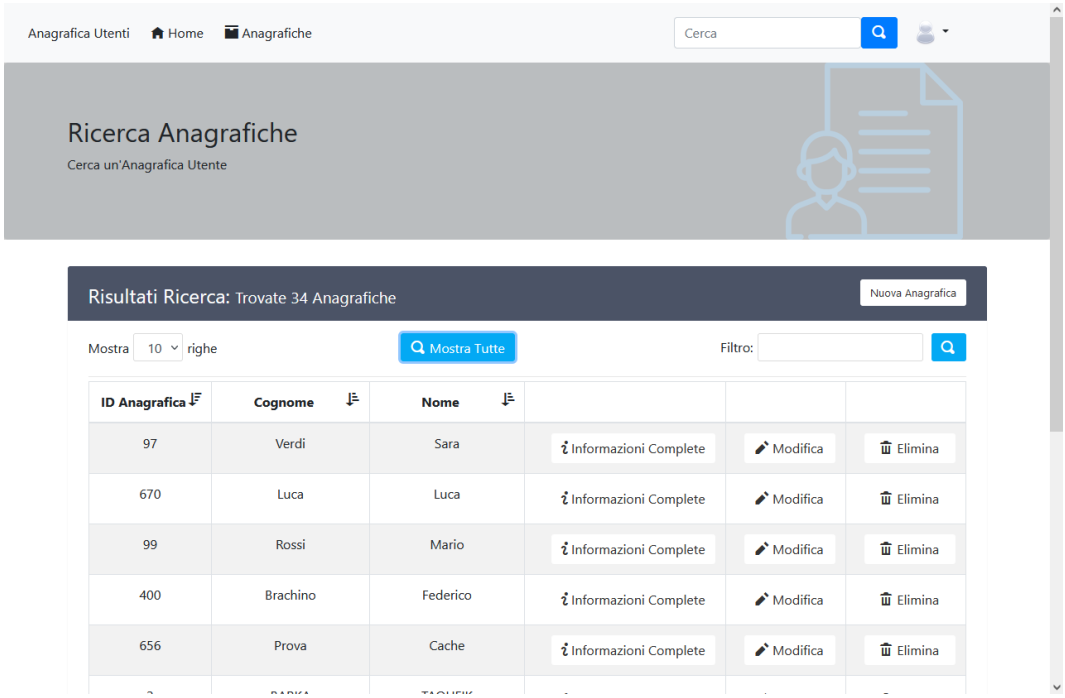

Figura 9: Esempio di una pagina del frontend dell'applicazione

3.3 Database

I microservizi interagiscono con un unico database PostgreSQL, formato da uno schema che contiene sette tabelle: le query ad una determinata tabella vengono effettuate solo da un determinato microservizio. Questa modalità non è quella più usata nelll'architettura a microservizi, in quanto solitamente ogni microservizio interagisce con un database diverso, ma data la piccola quantità di dati nel database e il basso numero di richieste, si è scelto di dividere i dati accessibili dai microservizi per tabelle anziché per database.

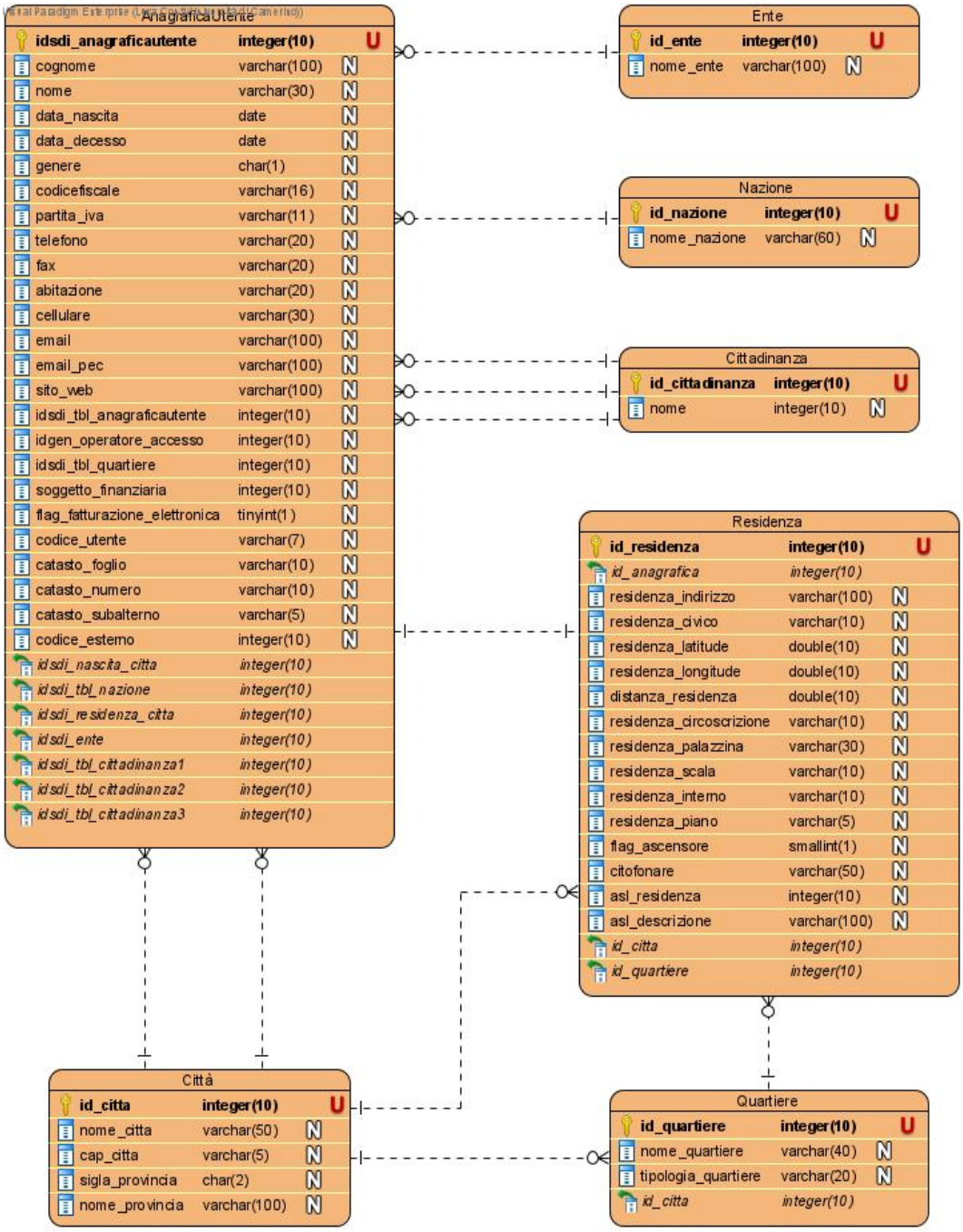

Figura 10: Diagramma ER del database

4 Tecnologie utilizzate

Di seguito verranno descritte le principali tecnologie utilizzate nel progetto.

4.1 Linguaggi

Per i microservizi è stato usato il linguaggio Java, mentre per il frontend sono stati utilizzati TypeScript, HTML e CSS.

4.1.1 Java

Java [12] è un linguaggio di programmazione di alto livello orientato agli oggetti progettato per essere il più possibile indipendente dalla piattaforma di esecuzione. Il codice compilato non deve essere ricompilato se eseguito su un'altra piattaforma, infatti la compilazione genere un prodotto in formato bytecode che può essere eseguito da qualunque implementazione di un processore virtuale chiamato Java Virtual Machine.

4.1.2 TypeScript

TypeScript [13] è un superset di JavaScript open source sviluppato da Microsoft.
Il linguaggio estende la sintassi di JavaScript rendendo qualunque programma scritto con quest'ultimo linguaggio in grado di funzionare anche con TypeScript senza effettuare nessuna modifica. Le differenze principali sono:

- TypeScript è un linguaggio di programmazione orientato agli oggetti, mentre JavaScript è un linguaggio di scripting;
- TypeScript, al contrario di Javascript, utilizza la tipizzazione statica;
- TypeScript supporta i moduli, mentre Java no;
- Con TypeScript è possibile usare le interfacce, opzione non disponibile con JavaScript;
- TypeScript è un linguaggio compilato: cioè rende possibile mostrare gli errori durante la fase di sviluppo, riducendo quindi gli errori al momento di esecuzione. Diversamente JavaScript è un linguaggio interpretato, rendendo così gli errori visibili solo al momento dell'esecuzione.

4.1.3 HTML e CSS

L'HTML [14] (HyperText Markup Language) è un linguaggio di formattazione che descrive le modalità di impaginazione o visualizzazione grafica del contenuto, testuale e non, di una pagina web attraverso tag di formattazione. Sebbene l'HTML supporti l'inserimento di script e oggetti esterni quali immagini o filmati, non è un linguaggio di programmazione: non prevedendo alcuna definizione di variabili, strutture dati, funzioni o strutture di controllo che possano realizzare programmi, il suo codice è in grado soltanto di strutturare e decorare dati

testuali.
Il CSS [15] (Cascading Style Sheets) è un linguaggio usato per la formattazione di documenti HTML. L'introduzione del CSS si è resa necessaria per separare i contenuti delle pagine HTML dalla loro formattazione o layout e permettere una programmazione più chiara e facile da utilizzare, sia per gli autori delle pagine stesse sia per gli utenti, garantendo contemporaneamente anche il riutilizzo di codice ed una sua più facile manutenzione

4.2 Eclipse e Visual Studio Code

Per il backend è stato utilizzato l'ambiente di sviluppo integrato Eclipse [16]. La scelta è ricaduta su questo IDE perché ha un'ottima integrazione con il framework Spring, la Spring Tool Suit e Maven.
Per il frontend è stato invece scelto Visual Studio Code [17], che, nonostante non sia definito un IDE, ma un semplice editor di testo, permette una vasta personalizzazione tramite plugin rendendolo, con le giuste estensioni, una delle opzioni migliori attualmente disponibili per lo sviluppo di codice TypeScript.

4.3 Git e GitHub

Git [18] è un software di controllo versione distribuito utilizzabile da interfaccia a riga di comando che permette una gestione efficace del codice di un progetto.
Git viene implementato da GitHub [19], un servizio di hosting per progetti software accessibile da browser o da applicazione desktop, che permette l'interazione di altri utenti con il codice sorgente dei progetti.
Nello specifico sono state create due repository, rese accessibili anche al tutor aziendale, in cui sono stati caricati i codici sorgente del backend e del frontend. In esse sono state salvate anche versioni alternative del progetto in cui sono state provate tecnologie diverse da quelle poi scelte, come l'OpenFeign, il ModelMapper, l'Hystrix e il Circuit Breaker.

4.4 Docker

Docker [20] è una piattaforma software che permette di semplificare il deploy di un'applicazione integrando in un unico pacchetto, detto container, tutto il software e le dipendenze necessarie per il suo funzionamento. In precedenza la soluzione più usata era l'impacchettamento del software in un'istanza di una macchina virtuale, ma in questo modo era necessario gestire e deployare le macchine virtuali, inoltre molte risorse venivano sprecate per il sistema operativo delle VM. Docker è una tecnologia che consente la creazione e l'utilizzo dei container Linux: Docker utilizza il kernel di Linux e le sue funzionalità per isolare i processi in modo da poterli eseguire in maniera indipendente, richiedendo meno risorse di una macchina virtuale.

Docker è stato usato nel progetto per istanziare il database PostgreSQL tramite Docker Compose, uno strumento che permette di definire i parametri dei container da un singolo file di configurazione, solitamente chiamato docker-compose.yml.
Un'immagine Docker distribuita dagli sviluppatori era stata inizialmente usata per installare Keycloak, ma un bug della stessa impediva il riavvio del container dopo che esso fosse stato stoppato. Per questo motivo si è scelto di utilizzare la versione standalone di Keycloak.

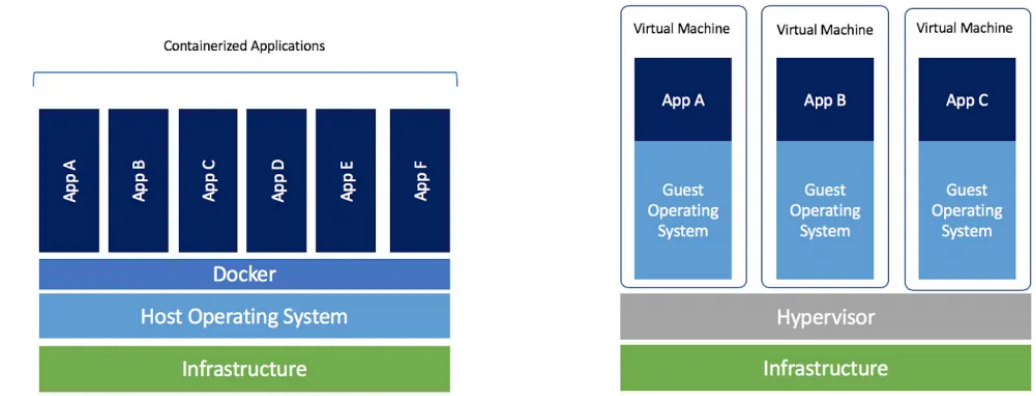

Figura 11: Confronto fra container Docker e Virtual Machine [9]

4.5 PostgreSQL

PostgreSQL [21] è un sistema di gestione di database relazionale ad oggetti (DBMS ad oggetti) open source. Esso utilizza molti degli standard SQL ed è completamente compatibile con le proprietà ACID:

- Atomicità: al termine della transazione gli effetti di quest'ultima devono essere totalmente resi visibili oppure nessun effetto deve essere mostrato;
- Consistenza: garantisce che al termine dell'esecuzione di una transazione, i vincoli di integrità sono soddisfatti, ovvero che il database si trova in uno stato coerente sia prima della transazione che dopo la transazione;
- Isolamento: garantisce che ogni transazione deve essere indipendente dalle altre, ovvero l'eventuale fallimento di una o più transazioni non deve interferire con altre transazioni in esecuzione;
- Durabilità: garantisce che i risultati di una transazione completata con successo siano permanenti nel sistema, ovvero non devono essere persi.

PostgreSQL usa il linguaggio SQL per eseguire delle query sui dati. Questi sono conservati come una serie di tabelle con chiavi esterne che servono a collegare i dati correlati.
La programmabilità di PostgreSQL è il suo principale punto di forza ed il principale vantaggio verso i suoi concorrenti: esso permette agli utenti la definizione di nuovi tipi basati sui normali tipi di dato SQL, rendendo più semplice passare dalla programmazione ad oggetti ai dati SQL.

Altro elemento a favore è l'essere supportato da una vasta gamma di interfacce grafiche che consentono un'amministrazione dei dati semplificata, come pgAdmin, phpPgAdmin o DBeaver.

4.6 Keycloak

Keycloak [22] è un progetto open source nato per semplificare la gestione dell'autorizzazione e dell'autenticazione nelle applicazioni. Una volta installato permette di accedere ad una console come amministratore e creare un reame, ovvero un oggetto che permette la gestione di un insieme di client, utenti, credenziali, ruoli e gruppi, offrendo anche una grandissima possibilità di configurazioni e opzioni dove necessario. I reami sono oggetti completamente isolati uno dall'altro e possono gestire ed autenticare solo gli utenti al loro interno. I client sono entità che possono richiedere l'autenticazione di un utente e solitamente sono applicazioni o servizi.
Per il progetto è stato creato un reame in cui sono stati registrati come client i due microservizi principali e la web app Angular, inoltre sono stati creati degli utenti, ad ognuno dei quali sono assegnati dei ruoli costumizzabili.
Keycloak utilizza un sistema di autenticazione SSO che permette che il login e il logout da un client registrato nel reame siano validi per tutti i client dello stesso reame, rendendolo un'ottima scelta per l'autenticazione in un'infrastruttura a microservizi.
Quando da un client si cerca di navigare in una pagina che richiede l'autenticazione Keycloak reindirizza ad una pagina personalizzabile che permette il login e la registrazione senza che lo sviluppatore debba scrivere nemmeno una riga di codice.
Dalla console amministratore è inoltre possibile assegnare dei ruoli personalizzabili agli utenti. I ruoli vengono poi utilizzati nel frontend per decidere se la route che si sta visitando (e di conseguenza la pagina) possa essere visualizzata dall'utente autenticato. Allo stesso modo permettono anche di gestire gli endpoint raggiungibili nel backend e quindi di consentire o bloccare determinate operazioni sul database.

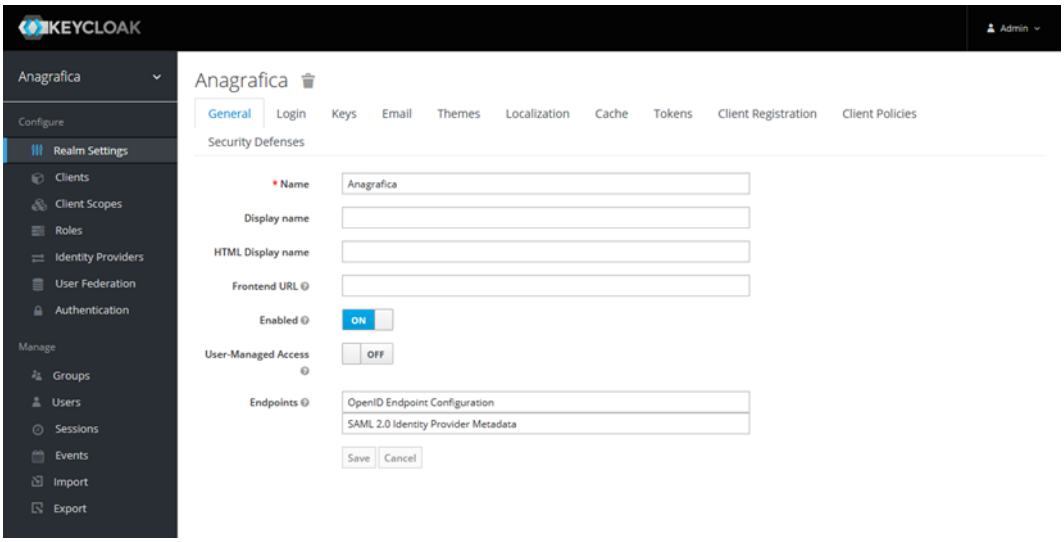

Figura 12: Schermata di gestione di un reame con Keycloak

4.7 Postman

Postman [23] è un'applicazione disponibile via browser o come app desktop che permette di testare e documentare API più velocemente. Tramite di esso è possibile effettuare delle chiamate API senza dover toccare il codice, ma tramite un'utile interfaccia grafica. Le chiamate possono essere effettuate sia verso un server locale che un server online impostando tutti i dati tipici di una chiamata ad una API, come headers e body.
Le caratteristiche principali di Postman sono:

- Consente di creare velocemente le API;
- Cronologia delle chiamate effettuate;
- Personalizzazione con script;
- Robustezza dei framework dei test;
- Creazione e gestione delle collezioni di API;
- Organizzazione delle API in workspace per collaborazione tra sviluppatori.

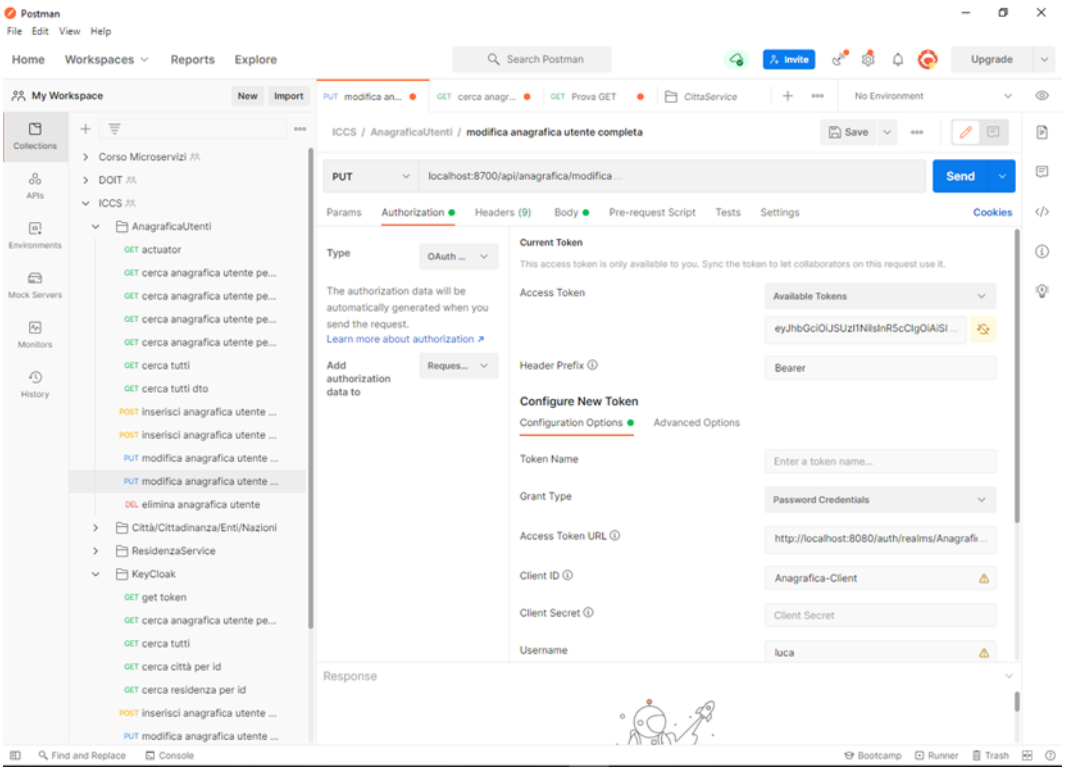

Figura 13: Schermata di configurazione dell'autenticazione di una chiamata POST dell'applicazione desktop Postman

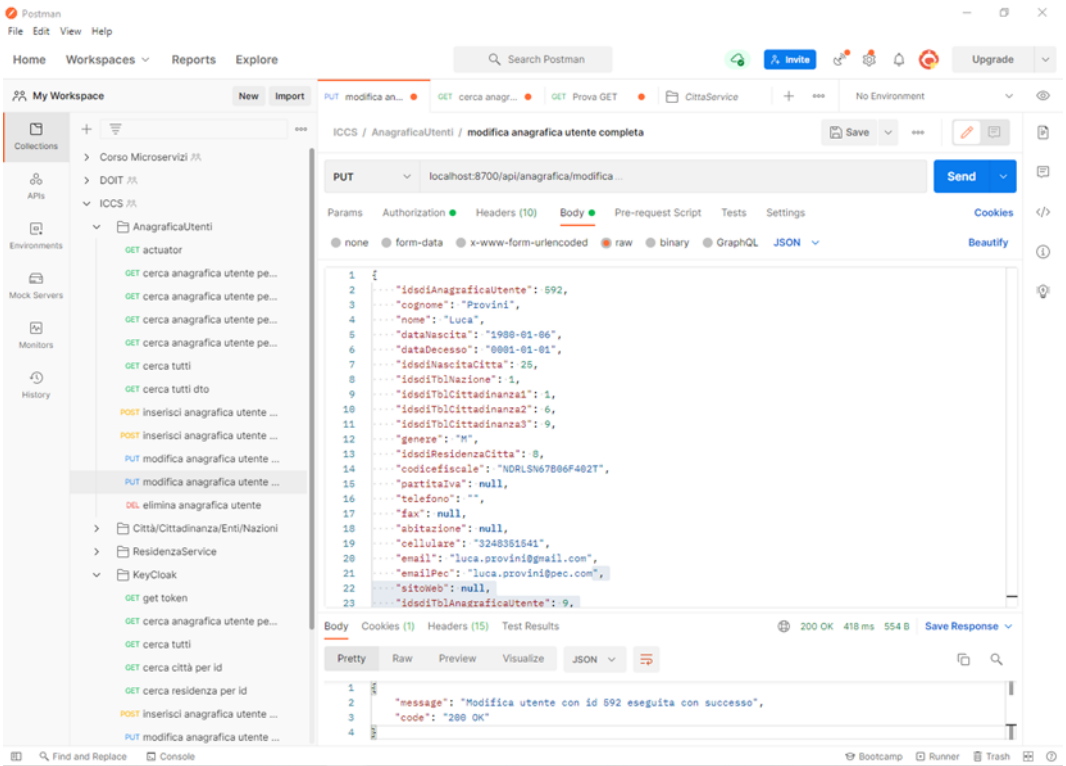

Figura 14: Schermata d'inserimento del body e risposta di una chiamata POST dell'applicazione desktop Postman

4.8 Spring Framework

Da alcuni anni lo standard de facto per la realizzazione di applicativi web è rappresentato dal framework Spring [24]. Spring è stato creato principalmente come container per Dependency Injection, cioè un pattern secondo il quale le varie dipendenze di un oggetto vengono soddisfatte non delegando il compito all'oggetto stesso ma attraverso l'iniezione della dipendenza dall'esterno. Il framework è delegato alla gestione del ciclo di vita di questi componenti e quindi anche a soddisfarne le dipendenze. Questo approccio, dove è lo stesso framework a farsi carico della gestione degli oggetti e di parti del programma, è chiamato Inversion of Control. In Spring l'Inversion of Control è gestita dall'ApplicationContext che è responsabile di istanziare, configurare ed assemblare gli oggetti, soprannominati bean, e gestirne il ciclo di vita. Le specifiche dei singoli bean possono venire definite in diversi modi: file XML, Java annotations o attraverso classi di configurazione. Per ottimizzare e velocizzare lo sviluppo di applicativi web, il team di sviluppo di Spring ha creato una versione preconfigurata del framework che provvede a fornire al programmatore un insieme di bean già configurati e pronti all'uso secondo il principio "Convention over Configuration", integrando il tutto con l'application server Tomcat in modo da rendere l'applicativo deployabile ovunque come un semplice file jar, dando origine a Spring Boot. Gli applicativi Spring Boot fanno uso di alcune annotazioni specifiche; le principali usate nel progetto sono:

- *@SpringBootApplication*: riunisce le tre annotazioni *@Configuration*, *@EnableAutoConfiguration* e *@ComponentScan* che hanno il compito di attivare la configurazione automatica di Spring Boot e rilevare gli altri componenti dichiarati all'interno del programma;
- *@Component*: viene usata per creare classi gestite dal framework;
- *@Bean*: viene usata dove non è possibile sfruttare i meccanismi automatici di Spring Boot per ritornare un oggetto istanziato tramite un metodo;
- *@Autowired*: viene usata quando si ha bisogno di usare un bean o un component in un'altra classe.

Il pattern di Dependency Injection su cui si basano Spring e Spring Boot rende quasi indispensabile uno strumento di build automation: i più diffusi sono Gradle e Maven, quest'ultimo usato nel caso di studio.
Maven [25] usa un file XML noto come POM (Project Object Model) che descrive le dipendenze fra il progetto e le varie versioni di librerie necessarie. Maven effettua automaticamente il download di librerie Java e plug-in dai vari repository definiti scaricandoli in locale o in un repository centralizzato lato sviluppo. Questo permette di recuperare in modo uniforme i vari file JAR e di poter spostare il progetto indipendentemente da un ambiente all'altro avendo la sicurezza di utilizzare sempre le stesse versioni delle librerie.
Nella parte successiva verranno introdotte alcune delle librerie principali usate.

4.8.1 Actuator

Actuator [26] introduce funzionalità pronte per la produzione all' applicazione senza che lo sviluppatore debba implementarle, rendendo triviale monitorare l'applicazione, raccogliere metriche, comprendere il traffico o lo stato del database L'Actuator viene utilizzato principalmente per esporre informazioni sull'applicazione in esecuzione: integrità, metriche, informazioni, dump, env, ecc. Utilizza endpoint HTTP o bean JMX per consentire di interagire con esso. Una volta che questa dipendenza viene importata, diversi endpoint sono disponibili immediatamente senza bisogno di definirli. Come con la maggior parte dei moduli Spring, si può facilmente configurarlo o estenderlo in molti modi.
Nel progetto sono stati usati gli endpoint health, metrics e info, quest'ultimo personalizzato con informazioni generali relative all'applicazione.

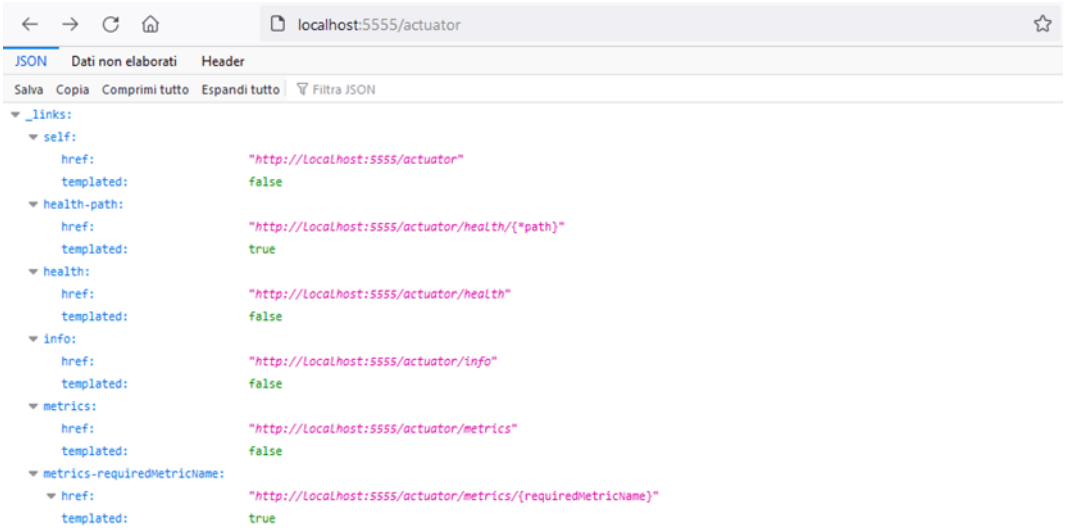

Figura 15: Schermata con endpoint dell'Actuator disponibili per il microservizio AnagraficaUtenti

4.8.2 Swagger

Swagger [27] è uno strumento che semplifica lo sviluppo e la documentazione di API. Tramite un file di configurazione e delle annotazioni nelle classi che si vogliono documentare, permette la definizione di specifici parametri per la documentazione automatica di metodi e oggetti. Swagger è utile soprattutto per la documentazione degli endpoint che si vogliono esporre, in quanto, tramite un'interfaccia grafica chiamata Swagger UI disponibile in un endpoint predefinito, permette agli sviluppatori di testarli senza che debbano implementare alcuna logica.

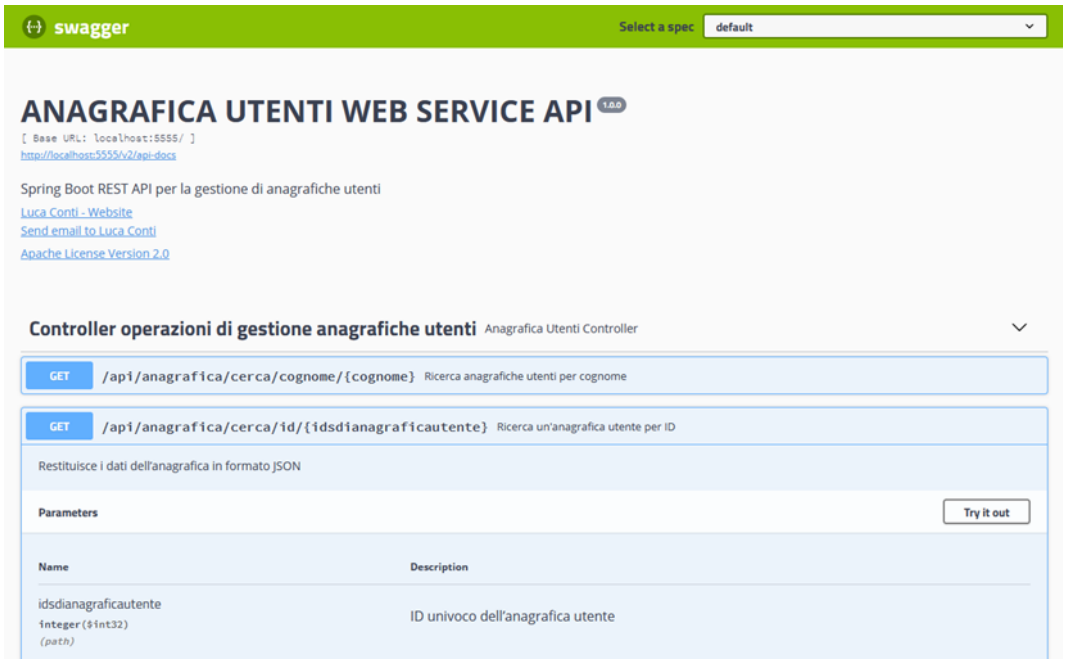

Figura 16: Esempio della documentazione creata da Swagger

4.8.3 Spring Security e Keycloak

Spring Security [28] è un framework che si concentra sulla gestione di autenticazione e autorizzazione alle applicazioni Java. Come tutti i progetti Spring, la vera forza di Spring Security si trova nella facilità con cui può essere estesa per soddisfare i requisiti personalizzati. Nel progetto è stato esteso tramite l'adapter Keycloak per Spring Boot [29], una dipendenza che con poche righe di configurazione permette di collegare l'applicazione con il reame creato sul server Keycloak.

4.8.4 Spring Data e PostgreSQL

Spring Data è un framework che si occupa della persistenza dei dati nei database, in particolare nel progetto viene usata la specifica JPA (Java Persistence API), che fornisce:

- una mappatura tra classi Java e tabelle del database;
- un linguaggio per effettuare query SQL, chiamato JPQL (Java Persistence Query Language), che è indipendente dalla DBMS utilizzato;
- varie API per la gestione e manipolazioni degli oggetti Java che mappano le tabelle del database.

Per fare ciò Spring Data incorpora una delle implementazioni più famose della specifica JPA [30]: Hibernate. Esso fornisce un servizio di object-relational mapping (ORM) che consente di mappare tramite annotazioni le classi Java in tabelle di un database relazionale; sulla base di questo mapping Hibernate gestisce il salvataggio e il reperimento, producendo ed eseguendo automaticamente le query SQL, di tali classi sul database.
Il collegamento al database avviene tramite dei parametri inseriti nel file di

configurazione application.yml grazie all'apposita dipendenza Postgresql importata nei microservizi che hanno bisogno di interagire con il database.

4.8.5 OpenFeign

Spring Cloud OpenFeign [31] è una libreria che permette di effettuare richieste HTTP fra servizi: è progettato per consentire agli sviluppatori di utilizzare un modo dichiarativo per creare client HTTP mediante la creazione di interfacce annotate senza scrivere alcun codice boilerplate.
Annotando l'interfaccia con *@FeignClient* è possibile effettuare richieste al servizio con nome uguale a quello inserito come parametro nell'annotazione, rendendo quindi semplicissima la comunicazione fra due servizi.
Spring Cloud OpenFeign utilizza Ribbon per fornire load balancing lato client e anche per integrarsi al meglio con altri servizi cloud, come Eureka per il service discovery e Hystrix per la tolleranza ai guasti.

4.8.6 Eureka

Eureka [32] è il componente che funge da service registry, cioè un registro in cui i vari servizi si registrano indicando il loro indirizzo fisico, la porta su cui sono in ascolto ed un service ID, rendendo così possibile ad altri servizi di interagire con questi senza dover conoscere a priori la loro posizione. Questo approccio prende il nome di service discovery ed è di estrema importanza nelle architetture distribuite: dal momento che per utilizzare un servizio non è necessario conoscerne la posizione effettiva ma solo il nome, è possibile aggiungere o togliere istanze di un certo servizio pur mantenendolo sempre raggiungibile, in questo modo aumentando la flessibilità e la scalabilità e permettendo di aumentare la resilienza dell'applicazione in caso di malfunzionamento di un servizio dato che la richiesta fallita può essere dirottata su di una replica funzionante dello stesso servizio. Per creare un'istanza di Eureka, è sufficiente importare la dipendenza Spring Cloud Starter Netflix Eureka Server, inserire l'annotazione *@EnableEurekaServer* all'interno della classe di bootstrap del servizio e configurare alcuni parametri nel file di configurazione. In questo modo, sarà possibile accedere ad un'interfaccia grafica che contiene tutte le informazioni relative ai servizi registrati. La registrazione di un servizio con Eureka avviene importando la dipendenza Spring Cloud Starter Eureka Client, aggiungendo alcuni parametri nel file di configurazione, facendo attenzione che coincidano con quelli inseriti nel server, e annotando la classe di avvio del servizio con *@EnableEurekaClient*.

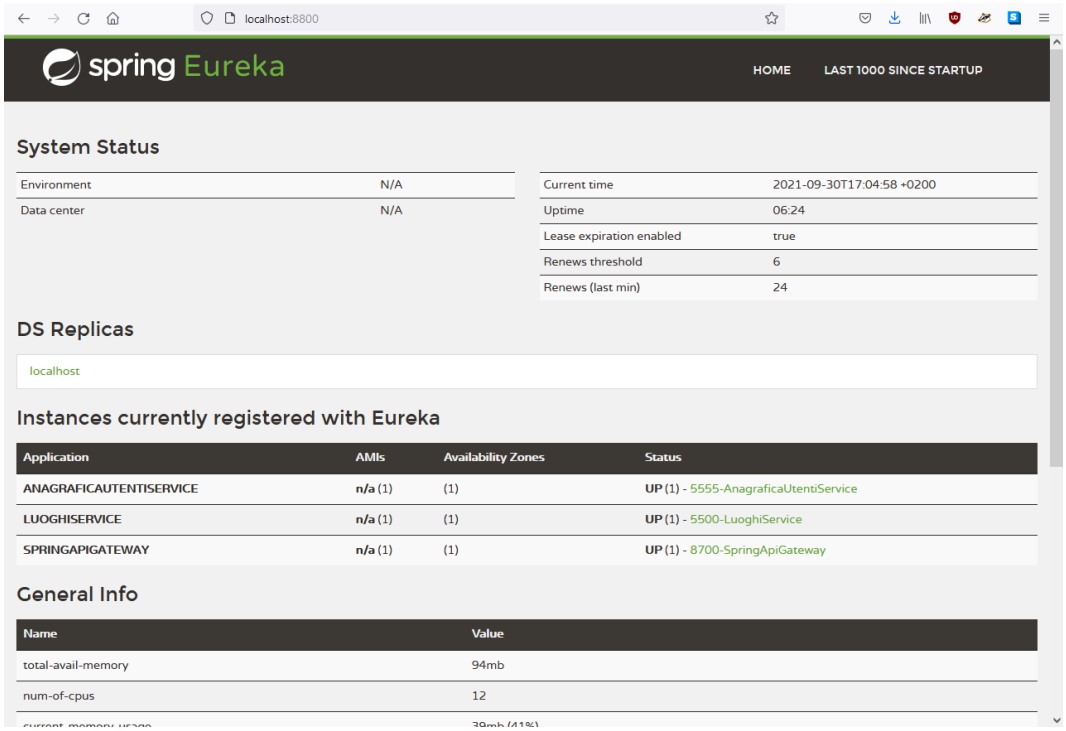

Figura 17: Schermata di monitoraggio delle istanze tramite interfaccia grafica Eureka

4.8.7 Gateway e Zuul

Nell'accezione più generica del termine un gateway è un punto di accesso ad una rete, in questo caso di microservizi. Una delle librerie principali per implementare questa funzionalità in un progetto Spring Boot era Netflix Zuul, ma dalla versione 2.4 non è più supportata. Al suo posto è stato sviluppato Spring Cloud Gateway [33], che è la dipendenza usata nel microservizio Gateway del progetto.
Spring Cloud Gateway è una libreria che viene usata nelle architetture a microservizi per nascondere i servizi dietro una singola facciata. La funzione principale è quella di determinare cosa fare con le richieste che soddisfano una specifica route. La configurazione delle rotte può essere effettuata sia tramite file di configurazione sia tramite bean.
I componenti principali dell'API sono le route: esse sono definite da un identificatore, un URI di destinazione e un insieme di predicati e filtri. I predicati vengono usati per controllare la corrispondenza della richiesta con gli endpoint permessi. I filtri permettono la manipolazione delle richieste e delle risposte, prima e dopo che vengano inoltrate.

4.8.8 Spring Web e CORS

Per abilitare la comunicazione fra due domini diversi, come il frontend ed un microservizio, è necessario abilitare il Cross-Origin Resource Sharing. Tramite il framework Spring Boot Web è possibile modificare vari parametri delle richieste, come le origini (in questo caso l'host dell'applicazioni Angular), i metodi HTTP e

gli headers permessi e i path a cui viene applicato [34].
Questi parametri vengono inseriti all'interno di un bean definito in una classe di configurazione o direttamente nel file di configurazione application.yml.
Queste impostazioni devono essere inserite anche nel Gateway, ma in questo caso va aggiunto il filtro *DedupeResponseHeader = Access-Control-Allow-Origin*. Questo filtro permette di rimuovere gli header doppi: senza di esso la richiesta avrebbe due header dello stesso tipo, uno aggiunto dal frontend e uno dal gateway, e quindi la richiesta dal gateway allo specifico microservizio non andrebbe a buon fine.

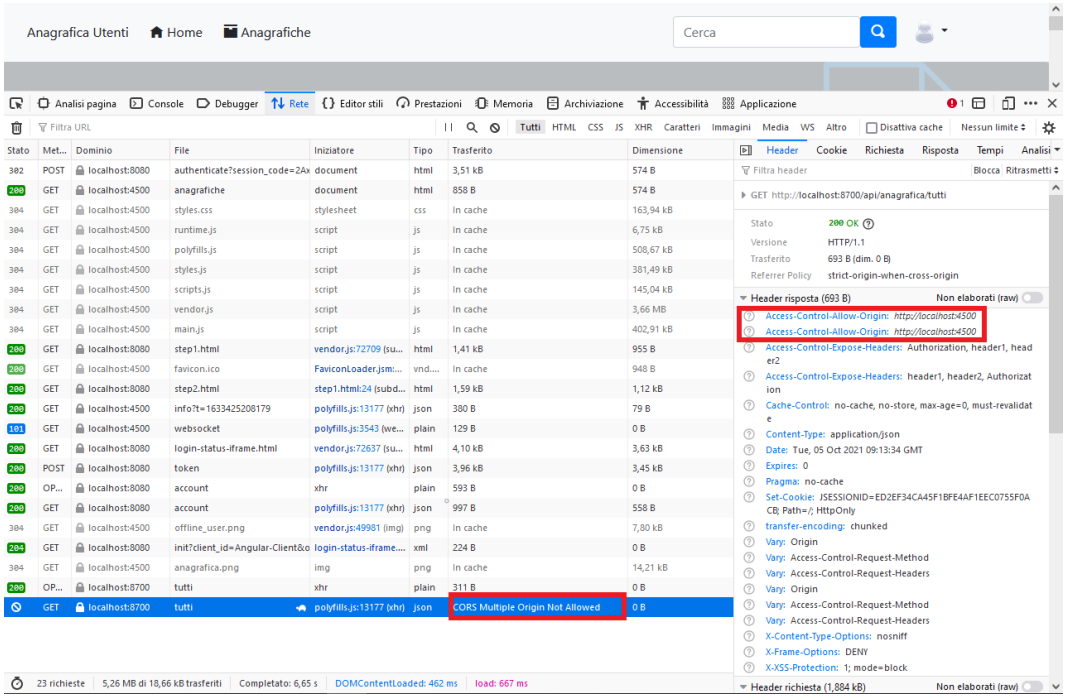

Figura 18: Esempio dell'errore relativo al CORS descritto

4.8.9 MapStruct e ModelMapper

Una parte fondamentale di ogni applicazione è la manipolazione degli oggetti.
In particolare nei microservizi che compongono il progetto vengono definite due classi per ogni oggetto concettuale: la entity e il DTO (Data Transfer Object).
Le entity, identificate dall'annotazione *@Entity* sono classi che vengono mappate tramite JPA sulle entità tabellari di un RDBMS. Le classi DTO implementano invece l'omonimo design pattern che viene usato per trasferire dati tra sottosistemi di un'applicazione software.
La conversione fra questi due tipi di dato, e in generale tra due tipi di dato qualsiasi, può essere semplificata tramite apposite librerie come ModelMapper [35] e MapStruct [36].
La prima può effettuare la conversione tra due tipi di dato definendo un bean in una classe di configurazione. La seconda permette di dichiarare interfacce che verranno implementate automaticamente durante il build dell'applicazione. La scelta è ricaduta su MapStruct in quanto riesce ed effettuare autonomamente il

mapping anche in caso di oggetti complessi, mentre con ModelMapper bisogna specificare i campi nel caso in cui i tipi non corrispondano.

4.8.10 Cache e Hazelcast

Per migliorare le performance è indispensabile implementare una cache per ridurre il numero di chiamate al database, laddove si effettui una richiesta già fatta.
La cache può essere facilmente aggiunta importando la dipendenza Spring Boot Cache [37] e aggiungendo l'annotazione *@EnableCaching* nella classe di boot. Nello strato di servizio, aggiungendo l'annotazione *@CacheConfig* e *@Cacheable* si può decidere quale metodo la utilizza. Inoltre può anche essere eliminata con l'annotazione *@CacheEvict*, per far in modo che la cache non contenga valori obsoleti in caso di modifica o eliminazione di dati dal database.
Hazelcast [38] è un IMDG (In Memory Data Grid). Questa architettura supporta un'alta scalabilità e la distribuzione di dati in un ambiente clusterizzato. Nelle architetture a microservizi, la dipendenza Hazelcast viene usata per fornire una cache che può essere condivisa tra le diverse istanze di un microservizio. Essa utilizza le stesse annotazioni della cache, ma richiede la creazione di un file di configurazione aggiuntivo.

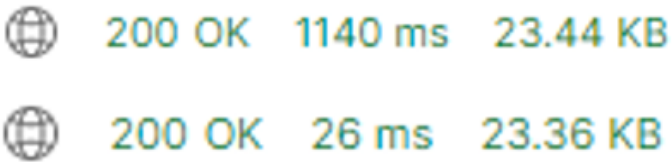

Figura 19: Tempi di risposta di una stessa richiesta GET con dati ottenuti da database (sopra) e da cache (sotto)

4.8.11 Hystrix e CircuitBreaker

Altro concetto fondamentale nelle architetture a microservizi è la tolleranza ai guasti, un pattern molto usato per la loro gestione è il circuit breaker.
Fino a poco tempo fa l'unica implementazione di questo pattern disponibile per i progetti Spring Boot era Spring Cloud Netflix Hystrix [39]. Il principio di funzionamento è simile a quello dei circuiti nell'elettronica: Hystrix monitora i metodi e quando rileva un fallimento apre il circuito e inoltra la chiamata ad un metodo di fallback. Dopo una certa soglia di fallimenti il circuito viene lasciato aperto e tutte le chiamate seguenti verranno inoltrate direttamente al metodo di fallback. Dopo un determinato intervallo di tempo il circuito passerà in uno stato definito semiaperto, dopodichè se la richiesta successiva avrà esito positivo il circuito verrà chiuso, altrimenti resterà aperto.
Recentemente è stata sviluppata una libreria chiamata Spring Cloud Circuit Breaker [40], che fornisce un'astrazione permettendo di scegliere l'implementazione di circuit breaker diversi (supporta Netflix Hystrix,

Resilience4J, Sentinel e Spring Retry) senza dover modificare l'applicazione.
Nel caso di studio questo pattern è stato applicato a livello di metodi, ma in fase di produzione può e dovrebbe essere applicato anche a livello di microservizi, cosicché se un'istanza è irraggiungibile o troppo lenta nel rispondere le richieste possano venir inoltrate ad un'altra istanza.

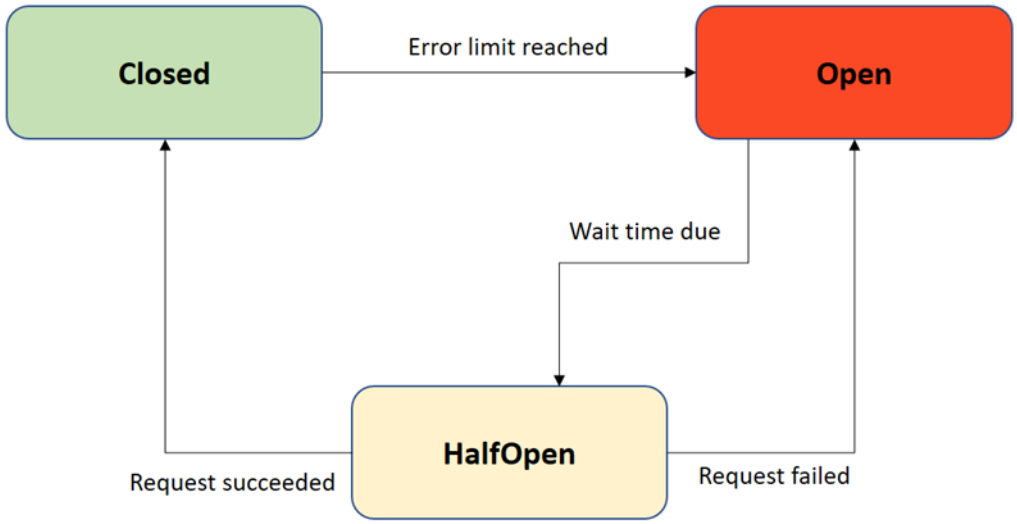

Figura 20: Schema di funzionamento del pattern circuit breaker [41]

4.9 Angular

Angular [42] è un progetto open source sviluppato da Google e rilasciato a partire dal 2016 e negli anni si è imposto come uno dei framework più utilizzati per lo sviluppo di Single Page Application. Questo tipo di applicativi web consistono in una pagina web in grado di renderizzare in modo dinamico i suoi elementi senza necessità di ricaricare la pagina, grazie a codice Javascript che si occupa di gestire la composizione della pagina e l'aggiornamento dei dati visualizzati, infatti, una volta effettuata la prima richiesta al web server che ospita la SPA richiesta, tutti i file che la implementano vengono scaricati sul client e non sono necessarie altre richieste.
L'applicazione è in grado di gestire la richiesta di dati ad API esterne tramite richieste HTTP rendendo possibile interagire con i vari microservizi per ottenere i dati desiderati e permetterne la manipolazione o la visualizzazione.
Uno dei principali vantaggi di Angular rispetto ad altri framework di questo tipo è rappresentato dall'utilizzo del linguaggio TypeScript al posto di JavaScript. Inoltre per semplificare l'attività di setup dell'ambiente di sviluppo il team di Angular ha sviluppato Angular CLI [43], un ambiente a riga di comando per creare la struttura di un'applicazione Angular già configurata secondo le linee guida ufficiali. Angular CLI è basato su nodejs e permette di creare una nuova applicazione preconfigurata con il solo comando "*ng new appName*".
Nelle pagine seguenti verranno introdotti gli elementi principali di Angular e

quelli più usati nel caso di studio.

4.9.1 Componenti

I componenti [44] sono l'unità fondamentale di un applicativo Angular: essi definiscono le unità funzionali e grafiche che, composte, permettono la creazione delle pagine visualizzate e l'implementazione delle funzionalità.
Tramite Angular CLI è possibile creare un componente con il comando "*ng generate component <component-name>*" che crea:

- una cartella con il nome del componente;
- un file TypeScript, "*<component-name>.component.ts*";
- un file HTML, "*<component-name>.component.html*";
- un file CSS, "*<component-name>.component.css*";
- un file per le specifiche per il testing, "*<component-name>.component.spec.ts*";

Le classi TypeScript associate ad un componente sono identificabili dal decoratore *@Component()* e richiedono alcuni metadati necessari alla corretta visualizzazione della view associata al componente (predefiniti se il componente è stato creato con Angular CLI):

- *templateUrl*: indica il riferimento al template del componente, un file HTML che verrà usato come modello per il rendering della view del component;
- *styleUrls*: indice una lista di fogli di stile CSS che verranno usati per personalizzare l'estetica del componente;
- *selector*: è un selettore CSS che permette di inserire il componente all'interno del template di altri componenti utilizzando il tag con il nome qui definito.

4.9.2 Servizi

I service [45] sono gli elementi predisposti ad ottenere e salvare dati. I servizi vengono distinti dai componenti per incrementare la modularità e la riusabilità: un servizio solitamente ha uno scopo preciso e può quindi essere usato dai componenti che ne hanno bisogno tramite dependency injection (infatti le classi di servizio vengono decorate con *@Injectable()*).
Anche i service possono essere facilmente generati con Angular CLI tramite il comando "*ng generate service <service-name>*", che genera una directory con il nome del service che contiene i due file TS.
I servizi possono essere iniettati a livello del componente che ne ha bisogno, tramite la proprietà "*providers*" del decoratore *@Component()*, creando un'istanza diversa per ogni componente che lo richiede, o a livello di root (come viene preconfigurato con Angular CLI) così che venga creata una singola istanza del servizio che viene iniettata nei servizi che lo richiedono tramite il costruttore.

4.9.3 Direttive

Il framework mette a disposizione delle classi particolari chiamate direttive [46] che permettono di modificare la composizione del template in base al valore delle variabili del componente.
In Angular esistono tre tipi di direttive:

- I componenti rappresentano una particolare direttiva caratterizzata dalla presenza di un template;
- Le direttive di attributo, il cui nome deriva dal fatto che vengono applicate ad un elemento come se fossero degli attributi, permettono di modificare l'aspetto o il comportamento di un elemento o del componente ad esso associato. Le principali sono *ngClass*, che permette di aggiungere o rimuovere classi CSS da un elemento in maniera dinamica, e *ngStyle*, che permette di modificare lo stile di un certo elemento in base al valore ottenuto dalla valutazione di una certa espressione;
- Le direttive strutturali consentono di mutare la struttura di un'applicazione andando a modificare direttamente il DOM e permettendo di aggiungere o rimuovere degli elementi. Le principali sono *ngIf*, che aggiunge o rimuove un elemento e i suoi discendenti dal DOM in base al valore dell'espressione assegnata, *ngFor*, che permette di inserire nel DOM una serie di elementi simili a partire da una lista di valori, e *ngSwitch*, che permette di selezionare un solo elemento, fra un gruppo di diverse possibili scelte, da inserire nel DOM in base a una certa condizione. Le direttive strutturali sono precedute da un asterisco o sono denotate esplicitamente con il tag *<ng-template>*.

4.9.4 Data Binding

Una delle funzionalità più interessanti fornite dal framework è la possibilità di legare variabili del componente ad elementi della view permettendo di sincronizzare il valore di un dato con quello visualizzato. Questo meccanismo prende il nome di data binding [47] ed in Angular può avvenire in una direzione (one-way data binding: dal componente alla vista o viceversa) o in entrambe (two-way data binding).
L'interpolazione è un tipo di data binding unidirezionale dal componente alla vista caratterizzato dalla notazione *{{ variabile }}* che permette di aggiornare dinamicamente il valore della variabile nel template in base al valore che la variabile ha nel componente.
L'interpolazione è una tecnica usata per le stringhe, negli altri casi si usa il property binding. Il property binding è denotato con *[proprietà]* e permette il binding della proprietà del componente all'elemento del DOM.
L'event binding è invece una particolare forma di binding unidirezionale dalla vista al componente denotato dalla sintassi *(evento target)="funzione()"*: questo permette di eseguire la funzione definita nel componente al verificarsi di un evento nel DOM, come il click su un bottone.

Combinando il property binding e l'event binding si ottiene il data binding bidirezionale: esso permette di mantenere sempre sincronizzato il valore di un dato visualizzato nella view anche quando questo viene direttamente modificato. Anche la sintassi è una sintesi dei due tipi di data binding esaminati in precedenza, ad esempio *<input type="number" [(ngModel)]="numero"/>* permette di aggiornare la variabile *numero* nella classe del componente quando viene inserito un input (cioè quando si verifica un evento) e allo stesso tempo di mostrare il valore correntemente associato alla variabile numero quando non si fornisce nessun numero in input.

4.9.5 Direttive usate nel data binding

La sopracitata *ngModel* è una direttiva predefinita usata per il two-way binding che in congiunzione con *ngForm* permette di verificare la validità di un form prima di passare i dati al componente.
ngValue permette invece di assegnare un valore ad un *ngModel* all'interno del tag *<select>* quando il valore che si vuole selezionare è associato ad un tag *<radio>* o *<option>*. Può essere inoltre utilizzato per effettuare one-way data binding. Sostituire *ngValue* a *ngModel* nell'esempio dell'input permetterebbe di mostrare il valore della variabile del componente nella casella di input, ma inserendo un valore diverso in essa il valore della variabile non verrebbe aggiornato.

4.9.6 Moduli

Le applicazioni Angular sono modulari e si basano su un sistema di moduli chiamati NgModules [48]. Essi sono contenitori per un blocco di codice coeso solitamente riferito ad un dominio, un workflow o a delle funzionalità correlate. I moduli contengono componenti, servizi, direttive, funzioni e costanti e possono esportare/importare funzionalità a/da altri moduli. Ogni applicazione Angular è composta da almeno un NgModule, il modulo root, che viene generato anche con Angular CLI, viene chiamato AppModule e risiede in un file chiamato *app.module.ts*.
Un modulo è definito da una classe decorata con *@NgModule()*. Questo decoratore è una funzione che prende un oggetto di metadati, le cui proprietà descrivono il modulo stesso. Le più importanti sono:

- *declarations*: i componenti, le direttive e le pipe che appartengono al modulo;
- *exports*: il sottoinsieme di declarations che sono visibili ed utilizzabili dai template dei componenti degli altri NgModules;
- *imports*: gli altri moduli le cui classi esportate sono richieste dai template dei componenti di questo modulo;
- *providers*: creatori dei servizi che li rendono disponibili in ogni parte dell'applicazione (possono essere specificati anche a livello di singoli componenti);

- *bootstrap*: la vista principale dell'applicazione, chiamata componente root, in cui vengono aggiunte tutte le altre viste dell'applicazione (solo l'NgModule root dovrebbe impostare questa proprietà).

4.9.7 Routing

Essendo una SPA composta da una sola pagina, il framework ha il compito di gestire la navigazione tra le viste e riflettere lo stato dell'applicazione nel relativo URL. Il modulo che gestisce questo aspetto in Angular è chiamato router [49].
Creando l'applicazione con Angular CLI verrà creato in automatico anche un file chiamato *app-routing.module.ts*, in cui sono già importate le classi necessarie relative al routing del modulo router di Angular.
All'interno di questo file vengono definite le associazioni tra URL e componente che implementa la pagina. Navigando da un indirizzo all'altro all'interno dell'applicazione sarà possibile passare alla visualizzazione della vista associata al componente desiderato. Un altro aspetto importante del Router è la capacità di imporre condizioni all'accesso degli URL tramite le Route Guards. Le Route Guards sono semplici classi Typescript che determinano se un utente ha i permessi necessari per navigare verso la pagina richiesta. Nel caso di studio questa classe è stata creata come estensione della classe astratta *KeycloakAuthGuard*, importata dal modulo *keycloak-angular*, che permette di controllare che il ruolo richiesto per accedere alla route sia presente fra quelli assegnati all'utente mediante keycloak.

4.9.8 Observable

Degli elementi spesso usati per il passaggio di dati da una parte ad un'altra di un'applicazione sono gli observable [50], una classe messa a disposizione dal modulo RxJS, che si occupa del trattamento di flussi di dati in modo asincrono e basato sugli eventi.
L'observable viene generato anche dalle richieste HTTP che vengono gestite dal servizio *HttpClient*, importabile dal modulo *http* di Angular: esso è l'oggetto che media il passaggio di dati tra un produttore, in questo caso il client HTTP, e un consumatore, il componente. Quando un componente intende ricevere i dati emessi dall'observable chiama la funzione *subscribe()*. All'interno della funzione è possibile indicare la callback che verrà chiamata quando un nuovo dato verrà emesso, una che verrà chiamata in caso di errore e una ulteriore che verrà chiamata solo dopo che una delle due precedenti ha terminato l'esecuzione. Nell'observable infatti, il producer può pubblicare i dati chiamando la funzione *next()* che scatena l'emissione di un nuovo dato e allo stesso modo chiama la funzione *error()* in caso di errori. Nel caso del client HTTP, che viene iniettato dentro la classe che implementa il servizio, quando viene fatta una richiesta HTTP, viene ritornato un observable che emetterà il dato ricevuto dal back end una volta che qualche componente si sarà iscritto e che il dato sarà disponibile.
Una volta che il componente non ha più bisogno di ricevere dati può disiscriversi

da un observable con la funzione *unsubscribe()*, solitamente chiamata alla distruzione del componente. Con il client HTTP di Angular questo passaggio non è necessario in quanto quando viene ricevuto il dato richiesto, viene chiamata sull'observable la funzione *complete()* che disiscrive gli observer ancora iscritti.

5 Conclusioni e Sviluppi Futuri

Questo capitolo sarà diviso in tre sezioni:

- Tecnologie che sono state studiate durante lo stage, ma non applicate al progetto per mancanza di tempo, ma che dovrebbero essere implementate in un'eventuale versione futura;
- Progetti aziendali in cui verranno applicate le tecnologie studiate ed applicate al progetto;
- Conclusioni e considerazioni personali.

5.1 Tecnologie non implementate

5.1.1 Spring Cloud Config

In un'architettura composta da molti microservizi è consigliabile crearne uno che gestisca le configurazioni di tutti gli altri. Per fare ciò è disponibile un progetto di Spring chiamato Spring Cloud Config [51] che permette di implementare in modo semplice un Configuration Server per centralizzare ed esternalizzare le configurazioni, permettendo di suddividerle e richiamarle per ambiente. Le configurazioni possono essere esternalizzate su vari sistemi di storage, anche se solitamente si usano repository Git.
Il Config Server è un'applicazione Spring Boot in cui la classe di avvio viene annotata con *@EnableConfigServer* e nel file di configurazione vengono impostati parametri relativi al luogo da cui recuperare i file di configurazione per i client (spesso una repository GitHub). I file qui presenti devono seguire una determinata nomenclatura, così da coincidere con i parametri presenti nel file di configurazione dei client.
Affinché il client legga le configurazioni dal server è sufficiente che abbia la dipendenza Spring Cloud Starter Config e che nel file di configurazione includa il nome dell'applicazione e l'URI del Server Config. Per attivare un determinato profilo nel client sarà ora sufficiente passare il nome al momento dell'avvio del client, così che il Config Server possa cercarlo nella repository.

5.1.2 Spring Cloud Bus e RabbitMQ

Nel caso di pochi servizi in esecuzione, quando ci sono modifiche ai file di configurazione è possibile effettuare un aggiornamento dinamico tramite l'endpoint *refresh* dell'Actuator, ma quando si hanno servizi con molte repliche questa diventa un'operazione scomoda. Per effettuare dinamicamente l'aggiornamento si può utilizzare lo Spring Cloud Bus [52]: importando la dipendenza Spring Cloud Starter Bus Amqp, usando l'annotazione *@RefreshScope* ed esponendo l'endpoint *bus-refresh* dell'Actuator, è possibile eseguire il refresh della configurazione di tutte le repliche del servizio mediante

una richiesta all'endpoint /actuator/busrefresh di una singola replica.
Perché questa comunicazione avvenga è però necessario un message broker, cioè un software che permetta a varie applicazioni di comunicare fra loro e scambiarsi informazioni tramite un protocollo comune. Il message broker usa un sistema di messaggistica asincrona, chiamato coda di messaggi, che ordina e archivia i messaggi finché le applicazioni che ne fanno uso non sono in grado di elaborarli, riuscendo così a garantire che la consegna dei messaggi avvenga una sola volta e nell'ordine corretto.
Per connettersi al server RabbitMQ è sufficiente impostare indirizzo del server, porta e credenziali nel file di configurazione del client. RabbitMQ mette a disposizione anche una dashboard da cui è possibile visualizzare lo stato della coda, la cronologia ed altre informazioni relative ai messaggi.

5.1.3 Zipkin e Sleuth

In un'architettura complessa non è raro che un microservizio ne invochi altri, che magari a loro volta effettuano chiamate ad altri microservizi. Questo rende difficile capire dove si generi un eventuale rallentamento. Per risolvere questi casi si utilizza un metodo chiamato tracing distribuito, reso possibile da due librerie: Sleuth e Zipkin.
Sleuth [53] permette di assegnare un id univoco ad ogni richiesta esterna del client (trace id) e uno ad ogni sotto-richiesta gestita da un altro servizio invocata in seguito alla richiesta originale (span id). In questo modo il trace id sarà uguale per l'intero flusso della richiesta, ma gli span id sono diversi ad ogni passaggio.
Zipkin [54] permette invece di inviare, ricevere, archiviare e visualizzare i dettali delle tracce e relativi span. Tramite una dashboard è quindi possibile analizzare le tracce nel dettaglio per osservare il tempo trascorso nei vari servizi e se le operazioni sono fallite o meno, permettendo così di rilevare eventuali blocchi e rallentamenti.

5.1.4 ELK Stack

Con l'aumento del numero di microservizi diventa cruciale avere un sistema centralizzato di raccolta e gestione dei log, per questo motivo si usa il cosiddetto ELK Stack [55], composto da tre prodotti open source:

- Elasticsearch: un database NoSQL basato sul motore di ricerca Lucene;
- Logstash: uno strumento di pipeline di log che accetta input da varie fonti, esegue diverse trasformazioni ed esporta i dati verso vari target (ad esempio Elasticsearch);
- Kibana: una dashboard di visualizzazione dei dati in Elasticsearch tramite tabelle e grafici.

L'ELK stack fornisce quindi una piattaforma unica per l'aggregazione, la ricerca, la gestione e l'analisi dei log.

Figura 21: Struttura di base dell'ELK Stack [56]

5.2 Sviluppi futuri dall'architettura studiata

Il caso di studio presentato è nato in previsione dell'applicazione delle tecnologie descritte ai prodotti software dell'azienda. In particolare si vogliono ammodernare le applicazioni che si occupano dei servizi di anagrafica, che sono state sviluppate con tecnologie diverse, e si vuole creare un nuovo prodotto che permetta la gestione tramite un unico portale di tutti i servizi di welfare per la popolazione più anziana.
Per fare ciò oltre alle tecnologie utilizzate nel progetto presentato, verranno sicuramente introdotte anche quelle descritte nell'ultima sezione.

5.3 Considerazioni e conclusioni personali

L'architettura a microservizi si è dimostrata particolarmente adatta per essere implementata in prodotti dalle dimensioni medio-grandi o in continuo sviluppo, grazie alla possibilità di eseguire modifiche mirate in un solo servizio senza intaccare gli altri, mentre la conversione non è un valido investimento quando si ha già un applicativo monolitico funzionante e non si ha intenzione di apportare modifiche significative o di espanderlo. Inoltre è un modello facile da implementare anche quando si inizia da una situazione in cui non si hanno conoscenze al riguardo.
L'architettura esagonale si è invece rivelata più semplice di quanto previsto in fase di avvio di stage, in quanto, almeno in programmi semplici come quelli studiati e quello proposto, l'implementazione è molto simile all'architettura a strati, già studiata durante il percorso universitario, con le maggiori differenze presenti a livello concettuale.

Bibliografia

[1] N. La Rocca, «Java Microservices con Sring Boot, Spring Cloud e AWS | Udemy,» [Online]. Available: https://www.udemy.com/course/java-microservices-con-spring-boot-e-spring-cloud/.

[2] N. La Rocca, «Sviluppare Full Stack Applications con Spring Boot e Angular | Udemy,» [Online]. Available: https://www.udemy.com/course/sviluppare-full-stack-applications-con-spring-boot-e-angular/.

[3] G. Capodieci, «Microservices Architecture: Il Pattern Architetturale Emergente Per Le Grandi Applicazioni Moderne,» [Online]. Available: http://losviluppatore.it/microservices-architecture-il-pattern-architetturale-emergente-per-le-grandi-applicazioni-moderne/.

[4] «Hexagonal Architecture (software) - Wikipedia,» [Online]. Available: https://en.wikipedia.org/wiki/Hexagonal_architecture_(software).

[5] A. Chatterjee, «Hexagonal Architecture in Hava – Expat Dev | Tech Blog,» [Online]. Available: https://www.expatdev.com/posts/hexagonal-architecture-in-java/#3-domain-object.

[6] [Online]. Available: https://miro.medium.com/max/6996/1*xu1Ge_Cew0DHdSU6ETcpLQ.png.

[7] [Online]. Available: https://miro.medium.com/max/875/1*G3NfSzSVh_cbzWyBF7YA8A.png.

[8] [Online]. Available: https://www.oreilly.com/library/view/hands-on-microservices/9781789133608/assets/ab95727d-9b5b-4831-a2b5-e4c0b5a1d11e.png.

[9] [Online]. Available: https://i1.wp.com/www.docker.com/blog/wp-content/uploads/Blog.-Are-containers-..VM-Image-1-1024x435.png?ssl=1.

[10] L. Conti, «lucaUni18/ICCS,» [Online]. Available: https://github.com/lucaUni18/ICCS.

[11] L. Conti, «lucaUni18/ICCSAngular,» [Online]. Available: https://github.com/lucaUni18/ICCSAngular.

[12] «Java | Oracle,» Oracle Corporation, [Online]. Available: https://www.java.com/it/.

[13] «TypeScript: JavaScript With Syntax For Types,» Microsoft, [Online]. Available: https://www.typescriptlang.org/.

[14] «HTML Standard,» World Wide Web Consortium, [Online]. Available: https://html.spec.whatwg.org/.

[15] «CSS Snapshot 2020,» World Wide Web Consortium, [Online]. Available: https://www.w3.org/TR/CSS/.

[16] «Enabling Open Innovation & Collaboration | The Eclipse Foundation,» Eclipse Foundation, [Online]. Available: https://www.eclipse.org/.

[17] «Visual Studio Code – Code Editing, Redefined,» Microsoft, [Online]. Available: https://code.visualstudio.com/.

[18] L. Torvalds, «Git,» [Online]. Available: https://git-scm.com/.
[19] «GitHub: When the world builds software – GitHub,» GitHUb Inc., [Online]. Available: https://github.com/.
[20] «Empowering App Development for Developers | Docker,» Docker Inc., [Online]. Available: https://www.docker.com/.
[21] «PostgreSQL: The world's most advanced open source database,» PostgreSQL Global Development Group, [Online]. Available: https://www.postgresql.org/.
[22] «Keycloak,» [Online]. Available: https://www.keycloak.org/.
[23] «Postman API Platform | Sign Up For Free,» [Online]. Available: https://www.postman.com/.
[24] «Spring | Home,» VMware Inc., [Online]. Available: https://spring.io/.
[25] «Maven - Welcome to Apache Maven,» The Apache Software Foundation, [Online]. Available: https://maven.apache.org/.
[26] J. C. V. Sanchez, «Spring Boot Actuator | Baeldung,» [Online]. Available: https://www.baeldung.com/spring-boot-actuators.
[27] Baeldung, «Setting Up Swagger 2 with a Spring REST API | Baeldung,» [Online]. Available: https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api.
[28] «Spring Security,» VMware Inc., [Online]. Available: https://spring.io/projects/spring-security.
[29] M. Good, «A Quick Guide to Using Keycloak with Spring Boot | Baeldung,» [Online]. Available: https://www.baeldung.com/spring-boot-keycloak.
[30] E. Paraschiv, «A Guide to JPA with Spring | Baeldung,» [Online]. Available: https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa.
[31] Baeldung, «A Guide to JPA with Spring | Baeldung,» [Online]. Available: https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa.
[32] Baeldung, «Introduction to Spring Cloud Netflix - Eureka | Baeldung,» [Online]. Available: https://www.baeldung.com/spring-cloud-netflix-eureka.
[33] Baeldung, «Exploring the New Spring Cloud Gateway | Baeldung,» [Online]. Available: https://www.baeldung.com/spring-cloud-gateway.
[34] Baeldung, «CORS with Spring | Baeldung,» [Online]. Available: https://www.baeldung.com/spring-cors.
[35] J. Halterman, «ModelMapper - Spring Integration,» [Online]. Available: http://modelmapper.org/user-manual/spring-integration/.
[36] «MapStruct - Java bean mappings, the easy way!,» [Online]. Available: https://mapstruct.org/.
[37] E. Paraschiv, «A Guide To Caching in Spring | Baeldung,» [Online]. Available: https://www.baeldung.com/spring-cache-tutorial.
[38] «Hazelcast - The Real-Time Intelligent Applications Platform,» Hazelcast Inc., [Online]. Available: https://hazelcast.com/.
[39] Baeldung, «A Guide to Spring Cloud Netflix - Hystrix | Baeldung,» [Online]. Available: https://www.baeldung.com/spring-cloud-netflix-hystrix.

[40] C. Burcea, «Quick Guide to Spring Cloud Circuit Breaker | Baeldung,» [Online]. Available: https://www.baeldung.com/spring-cloud-circuit-breaker.
[41] [Online]. Available: https://www.oreilly.com/library/view/implementing-azure-cloud/9781788393362/assets/7aad97ab-bc69-4353-a5a3-e9086ebac44a.png.
[42] «Angular,» Google LLC, [Online]. Available: https://angular.io/.
[43] «@angular/cli - npm,» [Online]. Available: https://www.npmjs.com/package/@angular/cli.
[44] «Angular - Angular Components Overview,» [Online]. Available: https://angular.io/guide/component-overview.
[45] «Angular - Introduction to services and dependecy injection,» [Online]. Available: https://angular.io/guide/architecture-services.
[46] «Angular - Built-in directives,» [Online]. Available: https://angular.io/guide/built-in-directives.
[47] «Angular - Binding syntax,» [Online]. Available: https://angular.io/guide/binding-syntax.
[48] «Angular - NgModules,» [Online]. Available: https://angular.io/guide/ngmodules.
[49] «Angular - Common Routing Tasks,» [Online]. Available: https://angular.io/guide/router.
[50] «Angular - Using observables to pass values,» [Online]. Available: https://angular.io/guide/observables.
[51] Baeldung, «Quick Intro to Spring Cloud Configuration | Baeldung,» [Online]. Available: https://www.baeldung.com/spring-cloud-configuration.
[52] Baeldung, «Spring Cloud Bus | Baeldung,» [Online]. Available: https://www.baeldung.com/spring-cloud-bus.
[53] «Spring Cloud Sleuth,» [Online]. Available: https://spring.io/projects/spring-cloud-sleuth.
[54] «OpenZipkin: A distributed tracing system,» [Online]. Available: https://zipkin.io/.
[55] «ELK Stack: Elasticsearch, Logstash, Kibana | Elastic,» Elasticsearch B.V., [Online]. Available: https://www.elastic.co/what-is/elk-stack.
[56] [Online]. Available: https://media.geeksforgeeks.org/wp-content/uploads/20200728122427/gog111.png.
[57] [Online]. Available: https://miro.medium.com/max/875/0*kS_Y9t9gpdjYssd1.png.

Ringraziamenti

Ringrazio di cuore il mio relatore Prof. Fausto Marcantoni che è sempre stato disponibile per consigli e suggerimenti, fin dall'inizio dello stage, e il mio correlatore Dott. Endri Azizi per la disponibilità e i costanti confronti di fronte ad un caffè.
Un ringraziamento anche a tutto il personale dell'azienda ICCS Informatica SRL, che mi ha subito accolto, ed in particolare al tutor aziendale Dott. Danilo Pazzelli e ai Dott. Rosario Dolce e Nadia Santamarianova, per la disponibilità mostrata.
Infine un enorme grazie a mia madre, mio padre e mia sorella per il loro supporto e la loro pazienza, e agli amici storici, Dario e Marco, che ci sono sempre stati.