Università degli Studi di Modena e Reggio Emilia

Dipartimento di Ingegneria "Enzo Ferrari"
Corso di Laurea in Ingegneria Informatica

Progettazione e sviluppo di una piattaforma riutilizzabile in contesto aziendale

Relatore:
Prof. Francesco Guerra

Presentata da:
Matteo Sirri

Anno Accademico 2020-2021

Sommario

Indice

Capitolo 1

Introduzione

1.1 Obiettivo

Oggi moltissime aziende e professionisti hanno la necessità di costruire una piattaforma software in grado di supportare le esigenze del business dell'organizzazione e ottimizzare i propri processi di lavoro. Questo significa che hanno bisogno di uno strumento dinamico in grado di svolgere numerose attività.

Obiettivo della tesi è fornire la descrizione di una piattaforma *cloud based* in grado di risolvere questa necessità. Essendo le esigenze di ogni azienda uniche nel loro genere si è cercato di realizzare un software in grado di erogare alcune delle funzionalità tipicamente richieste come la gestione degli utenti oppure l'invio di notifiche. Questo significa che questa piattaforma software non sarà un sistema autonomo e pronto ad essere utilizzato in un contesto specifico ma è da considerare come le fondamenta di un sistema più complesso che verrà costruito in base alle esigenze del cliente.

La realizzazione della piattaforma è avvenuta durante il periodo di tirocinio presso Fama Labs, un'azienda software erogatrice di servizi cloud su commissione. Questo ha permesso alla azienda ospitante di ottenere un prodotto sul quale costruire nuove soluzioni per i propri clienti permettendo di ridurne costi e tempistiche.

1.2 Panoramica

La restante parte del documento vuole analizzare la piattaforma realizzata, le tecnologie e il metodo di lavoro.

Il capitolo successivo fornisce una panoramica del sistema software e verrà presentato il metodo di lavoro applicato. Nel terzo capitolo verranno presentati gli strumenti tecnologici utilizzati per lo sviluppo della piattaforma. Il quarto capitolo analizza nel

dettaglio i componenti principali della piattaforma mentre il quinto vuole soffermarsi sulle fasi di integrazione e deployment del software.

Capitolo 2

Descrizione generale

2.1 Inquadramento

La piattaforma è stata progettata con lo scopo di erogare dei servizi tipicamente richiesti in ambito aziendale come la gestione delle informazioni degli utenti, meccanismi di autenticazioni e autorizzazione, l'invio di email e la sottoscrizione a servizi generici in versione di prova. L'erogazione di questi servizi avviene grazie ai seguenti elementi (vedi Figura 2.1):

- *Client*: applicazione *front-end* che permette agli utenti di sfruttare le funzionalità della piattaforma inviando delle richieste alla API e mostrando le risposte.
- *API web*: applicazione web server che permette di gestire le richieste del client e fornisce una interfaccia REST per erogare i servizi.
- Microservizio Doc: servizio interno che permette di consultare la documentazione degli endpoint della piattaforma.
- Microservizio Mailer: servizio interno che gestisce la generazione dei template delle email e l'invio.
- *Message Broker*: permette di fare interagire API web e il microservizio Mailer.
- *Email System*: sistema esterno utilizzato per l'effettivo invio delle email agli utenti.
- Database: permette di gestire i dati in modo permanente.

Gli utenti che interagiranno con la piattaforma software possono essere catalogati in utenti semplici e amministratori. Gli utenti semplici sono da considerare privi di conoscenze tecniche. Gli amministratori che gestiranno l'API, le applicazioni client, il sistema delle mail e il database sono personale altamente qualificato in grado di effettuare operazioni di monitoraggio, modifica, correzione e aggiornamento dei vari elementi.

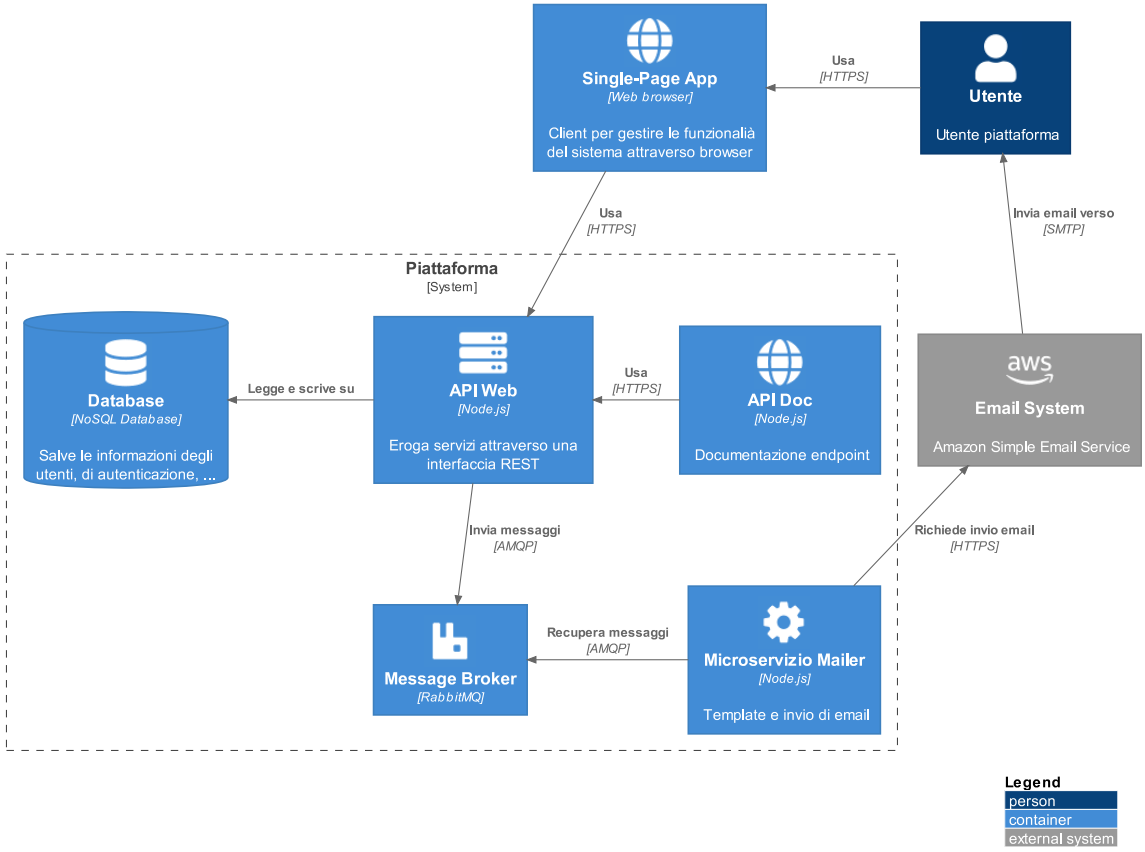

Figura 2.1: Architettura piattaforma

La piattaforma è da considerare come una base su cui costruire e aggiungere i componenti necessari per supportare in maniera efficace il business del cliente.

2.2 Vincoli generali

2.2.1 Requisiti di scalabilità

La piattaforma deve essere scalabile, ovvero in grado di supportare le operazioni degli utenti in maniera efficiente. Pertanto dovrà supportare una possibile transazione a una architettura completamente a microservizi per poter distribuire il traffico e il carico di lavoro. Anche le operazioni di supporto al processo di lavoro per la creazione del software deve essere scalabile. Verranno usate tecnologie, pratiche e metodologie per migliorare l'efficenza delle fasi di sviluppo, verifica e rilascio del prodotto.

2.2.2 Requisiti di affidabilità

La piattaforma deve garantire l'affidabilità dei servizi erogati e mantenere un livello di prestazioni costante quando viene utilizzata sotto varie condizioni per un dato periodo

di tempo. Deve garantire la presenza di meccanismi di *fault tolerance* per la gestione degli errori, malfunzionamenti o un utilizzo improprio della piattaforma garantendo la capacità di ripristinare le prestazioni e le informazioni nelle procedure più sensibili.

2.2.3 Considerazioni sulla sicurezza

Il sistema deve garantire la sicurezza degli utenti e l'accesso alle proprie informazioni sensibili. La privacy degli utenti dovrà essere garantita nel rispetto del Regolamento (UE) 2016/679 GDPR [1]. Sarà previsto un meccanismo di autenticazione basato su email e password per utenti e amministratori. Verrà poi implementato un meccanismo di autorizzazione basato sui ruoli e sui permessi di accesso alle singole risorse. Solo gli amministratori potranno accedere alla API, all'email system e al database. Per garantire la confidenzialità dei dati scambiati verrà usata una comunicazione sicura tra tutti gli elementi della piattaforma.

2.3 Metodologia di lavoro

La metodologia di lavoro adottata per la realizzazione della piattaforma si ispira alla metodologia Agile e alle pratiche di DevOps.

Sviluppo

La metodologia Agile, annunciato ufficialmente nel 2001 attraverso il Manifesto Agile [2], è un modello di sviluppo finalizzato a rilasciare frequentemente software funzionante e di qualità al cliente.

Questa metodologia prevede che le funzionalità del software vengano rilasciate costantemente secondo una cadenza molto frequente e non come un singolo prodotto finito. Il problema della realizzazione del software viene quindi affrontato con l'approccio *divide et impera*: attraverso una iterazione continua il prodotto software complesso viene suddiviso in sottoprodotti di più semplice comprensione. I singoli sottoprodotti verranno poi ricomposti come un singolo sistema software.

In questo contesto è inoltre necessario coinvolgere il cliente in maniera costante per renderlo consapevole dell'avanzamento del progetto, per verificare la comprensione dei requisiti e individuare errori prima di passare alla realizzazione del sottoprodotto successivo. Questo porta alla realizzazione di cicli in cui interagiscono il team di sviluppo e il cliente: questi si confrontano per individuare immediatamente problemi nelle funzionalità realizzate e poter così realizzare sin da subito software di qualità. Questi cicli

si ripetono ogni volta che deve essere realizzato un nuovo sottoprodotto. Gli elementi chiave del ciclo sono:

- Pianificazione: avviene la raccolta dei requisiti connessi a una funzionalità da implementare
- Design: progettazione delle funzionalità
- Codifica: vengono implementate le funzionalità incapsulate in componenti
- Test: si verifica se ciò che è stato prodotto funziona secondo i requisiti raccolti
- Revisione: si raccolgono le opinioni del cliente. In questa fase è possibile individuare errori e verificare la comprensione dei requisiti.

Integrazione e rilascio

Una volta realizzato il sottoprodotto è necessario metterlo nell'ambiente di produzione per poter così fornire le nuove funzionalità al cliente. Questa procedura implica l'integrazione dei nuovi artefatti con quelli già prodotti. La complessità di questa operazione è notevole in quanto bisognerà configurare l'artefatto e l'ambiente per farli cooperare, integrare i nuovi elementi, testare il prodotto risultante per verificarne il funzionamento e monitorare il nuovo prodotto per individuare problemi durante l'utilizzo. Queste operazioni solitamente non vengono fatte dal team di sviluppo (*Development Team*) ma dai sistemisti (*Operations team*). Questi si ritroveranno a dover risolvere dei problemi senza nessuna conoscenza del codice e ciò potrebbe portare a malfunzionamenti del sistema rilasciato al cliente. Per evitare questa situazione è stato deciso di seguire le pratiche DevOps.

Queste pratiche hanno come scopo l'eliminazione del confine tra i due team andando a considerare questo come un unico gruppo interfunzionale in cui le risorse siano in grado di acquisire responsabilità riguardanti ogni aspetto ciclo di vita del software. Ciò fornisce a tutti i membri una conoscenza *end to end* sul sistema.

Le tecniche e i principi DevOps vengono quindi applicati nella metodologia Agile per poter ottimizzare non solo le fasi di sviluppo del software ma anche le operazioni tipicamente fatte dall'*operations team*. Questo approccio suddivide il ciclo di sviluppo software nelle fasi di pianificazione, design, codifica, build, test, rilascio, deploy, monitoraggio e operazioni. Le fasi di pianificazione, design, codifica e test sono analoghe a quelle della metodologia Agile. La fase di build prevede la compilazione dei componenti realizzati, la fase di rilascio prevede la pubblicazione del codice sorgente in una repository remota, la fase di deploy consiste nell'installazione del sistema nell'ambiente di produzione mentre le fasi di monitor e operate consistono nella supervisione dell'ambiente e nella sua configurazione. Spesso queste fasi vengono supportate con l'uso

di tecnologie in grado di automatizzare il passaggio degli artefatti tra una fase e la successiva.

Le pratiche DevOps permettono quindi di ottimizzare l'efficenza del processo di sviluppo del software fornendo degli strumenti utili per favorire la collaborazione fra i vari reparti coinvolti. Fondendo questa filosofia con la metodologia Agile, che permette di rilasciare software di qualità in un contesto in cui le esigenze sono in continua evoluzione, è possibile ottenere dei risultati più affidabili e una maggiore efficenza nella gestione complessiva della produzione del software.

Confronto e valutazioni

Emerge quindi in modo evidente la contrapposizione con i classici modelli di sviluppo come il modello a cascata. Questa metodologia ha una struttura rigida composta dalle seguenti fasi: analisi dei requisiti, progettazione, codifica, test, rilascio, manutenzione. Ogni fase produce degli artefatti che sono necessari per il completamento delle attività nella fase successiva.

Questo implica che tra il contatto con il cliente, in cui avviene la raccolta dei requisiti, e la consegna del prodotto software può passare molto tempo. Ciò introduce rischi relativi alla realizzazione di un software tecnologicamente obsoleto oppure non in grado di soddisfare le esigenze del cliente (a causa della mancanza di comunicazione). Di conseguenza questa metodologia di lavoro non si presta in modo efficiente a quelle organizzazioni dinamiche che non conoscono a priori i requisiti che vogliono vedere soddisfatti e che necessitano di cambiamenti frequenti nelle operazioni.

Nella realizzazione della piattaforma è stato deciso di organizzare in questo modo i lavori perchè si è voluto dare molta importanza alla collaborazione fra i team di lavoro, all'ottimizzazione del processo di integrazione e rilascio di aggiornamenti e alla relazione con il cliente. Inoltre, visto che il prodotto offerto sarà una personalizzazione realizzata sulla base delle esigenze del cliente è stata scelta una metodologia di lavoro anche in grado di stimolare la nascita di relazioni costruttive e utili a portare a termine con successo il progetto.

Capitolo 3

Tecnologie

3.1 Implementazione

In questa sezione verranno descritti gli strumenti utilizzati per implementare i componenti che permettono alla piattaforma di erogare i propri servizi. La motivazione principale che ha portato alla scelta delle tecnologie di seguito elencate è il *know-how* aziendale.

3.1.1 Linguaggio

Typescript

Typescript [3] è un linguaggio open source sviluppato da Microsoft.

È un *super-set* del linguaggio JavaScript che ne permette l'estensione attraverso l'introduzione di un meccanismo di tipizzazione statico e il supporto alla programmazione orientata agli oggetti. Per via della sua natura può essere utilizzato in tutti i contesti in cui viene usato JavaScript, grazie a un processo di transpilazione che traduce codice Typescript in codice JavaScript, permettendone così una successiva compilazione ed esecuzione.

3.1.2 Ambiente di sviluppo

Node.js

Node.js [4] è un ambiente runtime JavaScript open source e multipiattaforma.

Le caratteristiche fondamentali sono: l'esecuzione dell'engine V8, sviluppato da Google, che permette di compilare ed eseguire codice JavaScript al di fuori di browser web, l'uso di un insieme di primitive I/O asincrone di tipo non bloccante e l'esecuzione di applicazioni su un solo processo, senza generazione di nuovi thread per ogni richiesta.

Questo sta a significare che quando si deve eseguire una operazione I/O, come una richiesta a un web server, Node.js non blocca il thread, mettendo in attesa la CPU, ma, al contrario, la lascia libera di portare avanti altri compiti e si occuperà di ripristinare l'operazione non appena arriverà una risposta utilizzando una *callback* Grazie a queste peculiarità è possibile realizzare applicazioni performanti in grado di gestire connessioni concorrenti con un singolo server.

In questo ambiente è poi possibile utilizzare lo standard ECMAScript in modo flessibile in quanto è possibile modificare il set di funzionalità abilitate, potendo così adattarsi al meglio nei vari contesti di utilizzo.

Infine, Node.js permette anche di aumentare la produttività di un team di sviluppo perché fornisce agli sviluppatori *front-end*, che conoscono il linguaggio JavaScript, la possibilità di sviluppare codice *server-side*; senza dover imparare un linguaggio del tutto nuovo. Grazie alle sue caratteristiche Node.js risulta essere un ottimo strumento per lo sviluppo di servizi web.

3.1.3 Librerie

Node Package Manager

Node Package Manager [5] (NPM) è un *software registry* per applicazione Node.Js. Questo si compone di due parti principali: il registro e una *Command Line Interface*(CLI).

Il primo è una raccolta di librerie open source che permettono l'integrazione in una applicazione numerose funzionalità e che può favorire la condivisione di codice, anche con l'uso di registri privati.

Il secondo permette d'interagire con il registro e gestire le dipendenze del progetto. In particolare, il meccanismo di gestione delle dipendenze di NPM permette di gestire con semplicità i pacchetti sul quale dipende una applicazione grazie all'utilizzo di un file particolare chiamato *package.json*. Al suo interno sono infatti raccolte tutte le informazioni relative alla applicazione, gli script eseguibili e l'elenco delle dipendenze. Questo risulta essere di fondamentale importanza perché permette a un team di sviluppo di avere un meccanismo che garantisce consistenza tra i vari ambienti usati dagli sviluppatori.

3.1.4 Framework

NestJS

NestJS [6] (Nest) è un framework basato su Node.js per realizzare delle web *Application programming interface* (API) e microservizi. Offre supporto sia il JavaScript

che il TypeScript e combina elementi di programmazione a oggetti e programmazione funzionale.

Nel dettaglio questo framework si pone come un layer di astrazione tra lo sviluppatore e un server HTTP basato su Express.js o Fastify (due framework per realizzare server web veloci e flessibili). Grazie a questo è inoltre possibile usufruire tutti i componenti aggiuntivi compatibili con la piattaforma sottostante, con ovvi vantaggi in termini di riusabilità e flessibilità.

Altro aspetto significativo di questo framework è che guida lo sviluppatore a realizzare una applicazione con una architettura *three-tier* ("a tre strati"), ovvero suddividendo gli elementi principali in tre strati dedicati alla gestione delle richieste dell'utente, alla gestione della logica funzionale e alla gestione dei dati.

Questa architettura viene supportata grazie a dei componenti di base, offerti dal framework stesso, che possono essere estesi dallo sviluppatore in base alle proprie esigenze. Nota significativa riguardo questi componenti è che fanno largo uso della *Dependency Injection*[1], uno dei meccanismi alla base del framework.

Pertanto l'utilizzo di Nest offre agli sviluppatori utili strumenti per velocizzare lo sviluppo di una web API prestando attenzione alle performance e alla architettura software del prodotto da realizzare risultando un'ottima scelta per la realizzazione si servizi web.

3.1.5 Testing

Jest

Jest [7] è un test runner JavaScript, sviluppato da Facebook, che fornisce un tool di strumenti per testare una applicazione basata su Node.js.

Permette una facile implementazione di unit test e integration test, con la possibilità di sfruttare i *mock objects* ("oggetti simulati"). Fornisce inoltre strumenti statistici per analizzare la *code coverage* ("copertura del codice").

3.2 Gestione dati

In questa sezione verranno descritti gli strumenti utilizzati per la gestione e l'aggiornamento delle informazioni necessarie per il funzionamento della piattaforma. La mo-

[1] La *Dependency Injection* è un meccanismo che permette di applicare l'inversione del controllo a un componente software. In generale permette a una classe di non dover configurare le proprie dipendenze in modo statico perché vengono configurate dall'esterno. Ciò offre grossi vantaggi in termini di riusabilità e rende la fase di test molto più semplice.

tivazione principale che ha portato alla scelta delle tecnologie di seguito elencate è il *know-how* aziendale.

3.2.1 Database

MongoDB

MongoDB [8] è un database management system (DBMS) che offre la possibilità di gestire database NoSQL basati sull'utilizzo di documenti flessibili per la gestione di dati in vario formato[9].

L'unità fondamentale in un database basato su MongoDB sono i documenti. Questi rappresentano l'informazioni da memorizzare. Sono formattati in formato *Binar y JSON* (BSON) e possono includere varie tipologie di dati che vanno dai comuni valori numerici o stringhe sino ai più complessi oggetti annidati e liste. La particolarità di questi documenti è che non hanno uno schema rigido: possono subire delle modifiche nel tempo. Permettono quindi di adattare i dati alla applicazione e non viceversa. Questo offre un notevole vantaggio per lo sviluppatore in quanto può strutturare i dati nella maniera più consona per facilitarne l'utilizzo. Inoltre i documenti facenti parti di una stessa categoria vengono poi raccolti in collezioni e tutti questi dati possono poi essere ovviamente sottoposti a query e ad aggregazioni.

MongoDB risulta essere anche molto performante grazie alla possibilità d'immagazzinare documenti innestati, riducendo l'attività I/O del sistema DB, e all'utilizzo del meccanismo d'indicizzazione che permette di effettuare query molto velocemente.

Altre caratteristiche fondamentali di questa tecnologia sono l'alta disponibilità e la possibilità di scalare orizzontalmente il sistema. La prima è ottenuta con l'uso dei *replica set*, che permettono di gestire database replicati, in modo da garantire ridondanza. La seconda è invece realizzabile con il metodo dello *Sharding* distribuito che permette la distribuzione del carico computazionale, per la gestione delle richieste, su più server; fornendo così una esecuzione più efficiente delle operazioni rispetto all'utilizzo di una solo server.

3.3 Servizi esterni

In questa sezione verranno descritti i servizi esterni integrati dalla piattaforma. La loro integrazione ha permesso alla piattaforma di erogare servizi complessi, riducendo spese e costi di gestione e sviluppo e aumentando la produttività del team.

3.3.1 Amazon Simple Email Service

Amazon Simple Email Service [10] (SES) è un servizio e-mail scalabile e conveniente che offre a uno sviluppatore un'interfaccia semplice per inviare e-mail da qualsiasi applicazione. Offre la possibilità di gestire l'invio di e-mail transazionali, di business o in massa permettendo così di potersi adattare a vari contesti di utilizzo.

Questo servizio fornisce anche un pannello di controllo con il quale è possibile effettuare il monitoraggio e l'analisi dei problemi, che potrebbero diminuire l'efficacia del recapito delle e-mail, e delle statistiche relative all'invio delle comunicazioni, con le quali si può misurare il grado di coinvolgimento dei clienti Permette inoltre di gestire il piano di utilizzo con il quale è possibile ottimizzare le spese di gestione.

L'integrazione di questo servizio ha permesso di gestire l'invio di e-mail in modo flessibile, performante ed economicamente conveniente senza dover aggiungere complessità alla piattaforma.

3.4 Gestione codice condiviso

In questa sezione verranno presentate le tecnologie utilizzate per supportare le operazioni del team di sviluppo e aumentarne la produttività. La motivazione principale che ha portato alla scelta delle tecnologie di seguito elencate è il *know-how* aziendale.

3.4.1 Git

Git [11] è un sistema di controllo versione distribuito (DVCS), gratuito e open source. Nato nel 2005 grazie all'operato della comunità Linux e di Linus Torvalds, il creatore di Linux, è a oggi uno strumento che permette a un team di sviluppo di collaborare in modo facile, veloce ed efficiente su grandi progetti.

Essendo un sistema di controllo versione permette di registrare, nel tempo, i cambiamenti a un file o a una serie di file, così da poter richiamare una specifica versione in un secondo momento [12]. Ciò offre numerosi vantaggi tra cui quello di poter ripristinare un file o un intero progetto a uno stato precedente a una modifica, vedere chi e quando ha cambiato qualcosa e recuperare file cancellati. Inoltre offre un sistema di ramificazione (branching) per lo sviluppo non lineare.

È stato utilizzato nel contesto dello sviluppo della applicazione attraverso GitLab, una piattaforma web open source che consente la gestione di repository Git.

3.4.2 Monorepo

Una *monorepo* [13] (*mono repository*) è una strategia di sviluppo software dove il codice relativo a vari progetti o applicazioni risiede in uno stesso luogo, solitamente una *repository* condivisa gestita con un sistema di controllo versione. Si contrappone al modello di singola *repository* per progetto.

Di seguito alcuni vantaggi offerti dall'utilizzo della strategia monorepo [14].

- Gestione semplificata delle dipendenze: le singole applicazioni condividono le stesse dipendenze. Questo stimola la coesione negli strumenti e librerie utilizzati e aumenta la produttività degli sviluppatori.
- Organizzazione semplificata: è possibile progettare una gerarchia logica fra i progetti per renderli logicamente relazionati anche all'interno della repository.
- Semplificazione processi di release: esiste una pipeline condivisa per la build e il deploy del sistema.
- Condivisione codice: è possibile utilizzare delle librerie interne per condividere codice fra le varie applicazioni riducendo così la duplicazione e favorendo la creazione di codice migliore.
- Standard e convenzioni: è possibile introdurre un utilizzo coeso relativo alle convenzioni di sviluppo e all'utilizzo delle tecnologie usate per testing, debug e build dei progetti. Queste permette di creare codice consistente e di qualità.

L'utilizzo di una monorepo permette quindi di creare piattaforme complesse, composte da applicazioni e microservizi, in modo coeso, dal punto di vista degli strumenti e tecnologie usate, e permettendo agli sviluppatori di avere le conoscenze adatte per lavorare in modo efficiente ed efficace ai vari progetti della organizzazione. Nonostante ciò l'utilizzo di questa strategia introduce un certo livello di rigidità nella gestione delle applicazioni e un grado di complessità, proporzionale alle dimensioni della monorepo, nell'introduzione di nuovi membri nel team di sviluppo.

Nello sviluppo della piattaforma è stata utilizzata questa strategia perchè si è voluto dare importanza alla coesione dei processi di compilazione, test, gestione delle dipendenze e deploy del software.

Nx

Nx [15] è uno strumento open source, creato dal team Nrwl [16], che permette a una organizzazione di gestire una monorepo. Il suo funzionamento è basato sulla CLI di Angular, un framework per lo sviluppo di applicazioni front-end, ed è composto da un insieme di strumenti che permettono di gestire in modo flessibile una monorepo per

progetti basati su Node.js, React o Angular. Alcune delle funzionalità offerte sono la generazione automatica della gerarchia di file e cartelle per generare una nuova applicazione o libreria, possibilità di eseguire script in un contesto globale o relegato alle singole applicazioni e meccanismi di ottimizzazione per le fasi di build e test grazie all'utilizzo dei un meccanismo di caching.

Nello sviluppo della piattaforma è stato decisa l'introduzione questo strumento in quanto permette di semplificare notevolmente la fase di design della monorepo stessa.

3.5 Integrazione e rilascio del software

In questa sezione sono descritte le tecnologie utilizzate nelle fasi d'integrazione e rilascio del software.

3.5.1 Docker

Docker [17] è una tecnologia open source per creare, distribuire e gestire applicazioni sotto forma di *container*. Un container Docker è un ambiente isolato in cui può essere messa in esecuzione una applicazione. Più nello specifico si crea un container basato su Linux in cui viene caricata un'immagine di una applicazione con le relative dipendenze.

Vantaggio offerto dalla piattaforma Docker è il fatto che questi container sono isolati li uni dagli altri e ciò permette di poterli eseguire in sicurezza e indipendentemente. Altra nota positiva è la leggerezza. Infatti i container Docker, a differenza delle macchine virtuali, non necessitano di un *hypervisor* ma vengono eseguiti direttamente dal kernel della macchina host.

L'utilizzo dei container permette poi di disaccoppiare le fasi di rilascio e deployment delle applicazioni e offre la garanzia di avere un ambiente in cui si ha la certezza che il comportamento della applicazione sarà quello previsto. Questo risulta vantaggioso anche per la produttività degli sviluppatori che possono così condividere il proprio lavoro più facilmente.

Questa tecnologia è stata usata nello sviluppo della piattaforma per supportare le operazioni d'integrazione e distribuzione. Viene inoltre utilizzato per effettuare il deployment, ovvero il rilascio dell'applicazione al reparto di produzione.

3.5.2 Jenkins

Jenkis [18] è uno strumento open source che fa parte della categoria degli *automation server*, ovvero applicazioni server utilizzate per automatizzare le task relative alle fasi di build, test, rilascio e deploy del software.

Basa il suo funzionamento su una pipeline, ovvero un insieme di step, detti *stage*, da eseguire in sequenza per portare a termine un processo. Ogni step potrà contenere a sua volte delle istruzioni, detti *jobs*, da eseguire per portare a termine un dato compito quale potrebbe essere la build, i test o il deployment. L'esecuzione della pipeline può avvenire modalità pianificata o utilizzando dei *trigger*, come ad esempio il commit su una repository o il fallimento di un test.

L'utilizzo di Jenkins permette di accelerare lo sviluppo software grazie agli automatismi introdotti che permettono di compilare, testare e rilasciare il software e individuare possibili problemi prima che questi giungano al reparto produzione.

3.6 Deployment

In questa sezione sono descritte le tecnologie utilizzate per il deploy della piattaforma.

3.6.1 Amazon EC2

Amazon Elastic Compute Cloud (Amazon EC2) è un servizio erogato dalla piattaforma di cloud computing Amazon web Services di Amazon che fornisce capacità di elaborazione sicura e scalabile nel cloud. In particolare permette di eseguire delle applicazioni o dei servizi web su risorse che vengono noleggiate in base all'utilizzo previsto. Questa pratica, detta provisioning delle risorse, permette di eliminare la necessità di effettuare investimenti anticipati in hardware, software e per la configurazione e gestione di queste risorse poiché l'investimento viene personalizzato sulla base delle esigenze richieste. Altra particolarità è che il provisioning di risorse è immediato. Questo significa che è possibile scalare orizzontalmente o verticalmente i propri servizi o applicazioni in poco tempo e ciò permette di avere sempre una piattaforma flessibile e performante.

Nella piattaforma è stato utilizzato questo servizio per il deploy della piattaforma.

Capitolo 4

Architettura

4.1 Panoramica

In questo capitolo si andranno a descrivere le caratteristiche dei componenti chiave della piattaforma quali il ruolo, le relazioni con gli altri elementi, i protocolli e gli standard utilizzati, i servizi erogati e i principi di design utilizzati per implementarli. In particolare ci si focalizzerà sui seguenti componenti: API web, *message broker*, microservizio Mailer e Doc.

4.2 API web

4.2.1 Descrizione generale

L'API web è il componente della piattaforma che permette di gestire le richieste degli utenti attraverso una interfaccia REST.

Essendo la piattaforma ideata come rappresentazione di una base riutilizzabile in vari contesti aziendali sono stati implementati alcuni dei servizi tipicamente richiesti come l'autenticazione e l'autorizzazione degli utenti, l'accesso a servizi demo sotto previa registrazione, l'invio di notifiche e la gestione delle informazioni dell'utente. Questo permetterà future implementazioni personalizzate riducendo notevolmente i costi e i tempi di sviluppo potendo così dedicare tutte le risorse sulle funzionalità richieste dalla clientela.

L'applicazione è basata su Node ed è stato utilizzato il framework Nest. La comunicazione tra client e API web verrà gestita con la suite di protocolli TCP/IP e in particolare verrà usato il protocollo HTTPS per gestire le richieste e le risposte. Si interfaccia con il microservizio del Mailer, attraverso un *message broker* RabbitMQ, per gestire l'invio di email e con il database server, basato su MongoDB, per gestire i dati

necessari per erogare i servizi richiesti. Inoltre interagisce con il microservizio Doc per mostrare la documentazione degli endpoint della piattaforma attraverso una interfaccia interattiva.

4.2.2 Modellazione dati

L'erogazione dei servizi da parte della API web necessità di una memorizzazione permanente di dati, incapsulati all'interno di entità.

Nel dettaglio si hanno:

- User: incapsula le informazioni dell'utente.
- Auth Mail: utilizzata per tenere traccia delle email di verifica dell'account e recupero della password.
- User Identity: rappresenta un'identità di terze parti.
- Refresh Token: rappresenta il refresh token di proprietà di un utente.
- Demo Subscription: identifica una sottoscrizione a un servizio demo da parte di un utente.

Per l'implementazione è stato deciso di utilizzare MongoDB, un database non relazionale basato su documenti. Per interfacciarsi con esso è stato deciso di utilizzare Typegoose [19]: una libreria *Object Data Modelling* (ODM) per gestire le relazioni tra i dati, validare gli schemi e convertire il formato delle informazioni usato nel codice e nel database; tutto ciò con la possibilità di sfruttare la tipizzazione tipica di Typescript. Ogni entità verrà poi incapsulata in componenti detti models (modelli) che forniranno tutte le funzionalità necessarie per comunicare con il database.

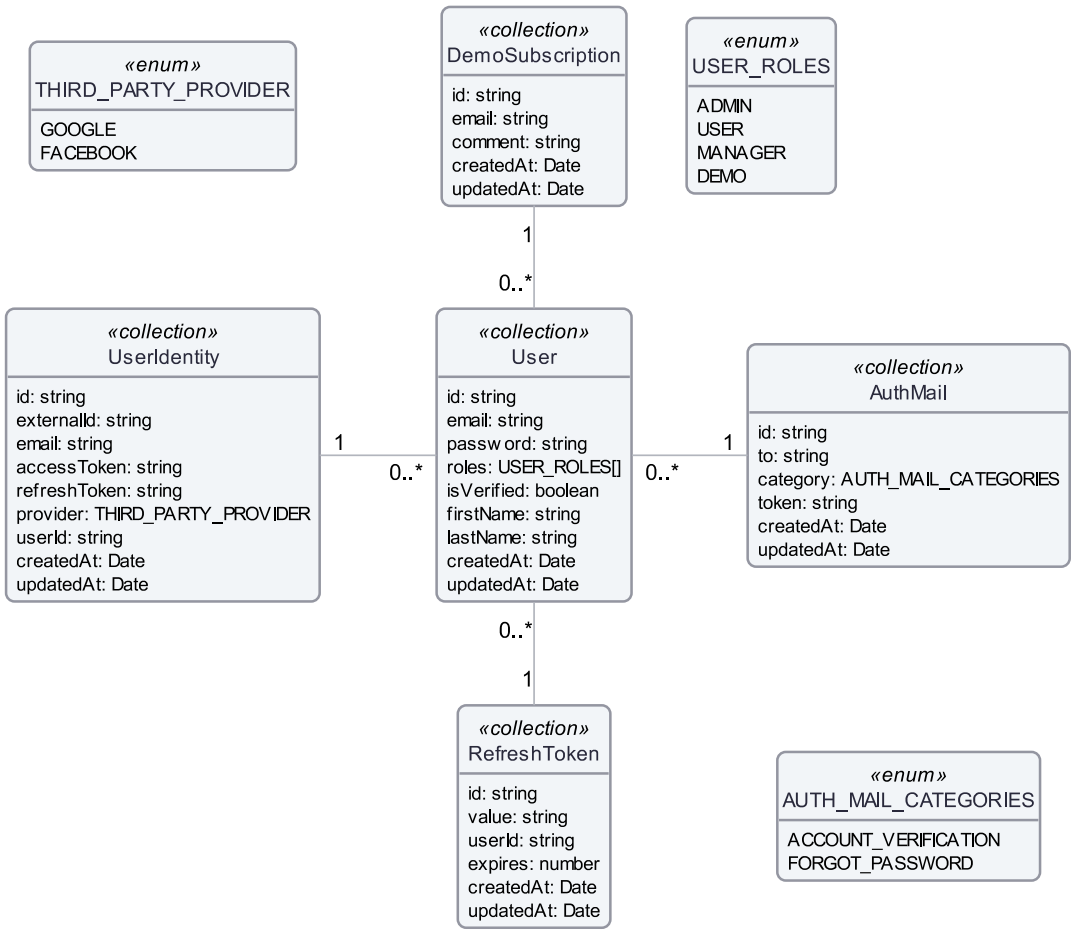

Figura 4.1: MongoDB Class Diagram

4.2.3 Principi di design

La piattaforma è organizzata in moduli relazionati tra loro prestando attenzione alla creazione di componenti coesi, disaccoppiati e che rispettino i principi SOLID. Un modulo esiste per incapsulare dei servizi che vengono erogati attraverso dei componenti che collaborano per portare a termine il compito richiesto. Ciascuno di essi può definire i moduli da importare per sfruttarne i servizi, i controller da esporre per gestire le richieste e i servizi che esporta. Di conseguenza un modulo espone una interfaccia software costituita dai servizi che esporta e una interfaccia REST costituita dagli endpoint, definiti nei controller esposti, che permettono di interagire con l'applicazione attraverso richieste HTTP.

Nella piattaforma avremo i seguenti moduli funzionali:

- Auth Module: si occupa di tutti gli aspetti inerenti alla autenticazione e autorizzazione degli utenti.
- User Module: si occupa della gestione delle informazioni degli utenti.
- Demo Module: permette la registrazione a servizi in versione demo.

- Mail Module: permette di gestire le richieste di invio email al *message broker*.

Ognuno di questi moduli è organizzato con una architettura a tre strati (vedi Figura 4.2) così definita:

- Controller Layer: contiene gli elementi, detti controllers, che permettono all'utente di interagire con i servizi offerti dalla piattaforma.
- Service Layer: contiene gli elementi, detti services, che hanno il compito di elaborare i dati (business logic).
- Data Access Layer: contiene gli elementi, detti models, che permettono di interagire con il database.

I moduli, i controller e i servizi sono delle istante *singleton*, create all'avvio dell'applicazione e condivise al suo interno. Inoltre ogni componente sfrutta il meccanismo della *dependency injection* per risolvere le proprie dipendenze e ciò comporta notevoli vantaggi per quando riguarda la riusabilità del codice e la verifica del comportamento dei componenti nella fase di test. Nella applicazione sarà poi presenta anche un modulo di *root*, detto *application module*, che permette si risolvere le dipendenze tra i vari elementi della applicazione e ne esegue l'avvio. Inoltre ogni modulo potrà poi essere convertito in microservizio nel caso in cui si voglia avere la possibilità di scalare indipendentemente i vari servizi in base alle esigenze richieste dalle operazioni degli utenti. Questi elementi supportano la realizzazione di una architettura scalabile, coesa, disaccoppiata e mantenibile.

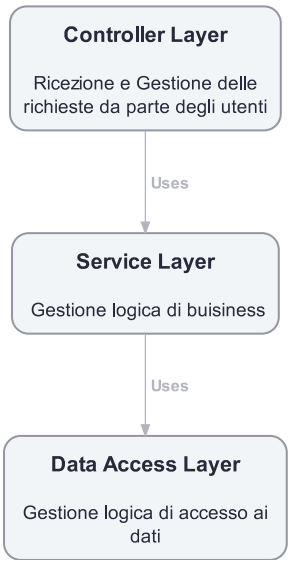

Figura 4.2: Schema riassuntivo della architettura *three-tier*

4.2.4 Auth Module

Il modulo auth si occupa dei servizi autenticazione e autorizzazione. In particolare supporta le operazioni di registrazione e accesso alla piattaforma (anche attraverso provider di terze parti), recupero password e verifica dell'account. Il modulo auth dipende dal modulo user per gestire le procedure di registrazione e dal modulo mail per inviare le notifiche di verifica dell'account e per il recupero della password.

Autenticazione

L'autenticazione all'interno della API web si basa su una coppia di credenziali: email e password. Questo metodo permette l'identificazione degli utenti sulla base di determinate credenziali e la verifica della legittimità di una utenza.

Per garantire una archiviazione sicura delle password è stato deciso di imporre regole sul suo formato per renderla meno vulnerabile (lunga e con una sequenza confusa di numeri, lettere e simboli) e applicare una policy basata su funzioni di *hash* e di *salt*.

Una funzione di *hash* è una funzione non invertibile che partendo da un messaggio di dimensione qualsiasi genera una stringa binaria di dimensione fissa, chiamata *message digest* oppure valore di *hash*, che identifica in modo univoco il messaggio originale.

L'applicazione della sola funzione di hash non è però sufficiente a proteggere una password a causa degli attacchi basati su dizionari, su tabelle di associazione hash-password (*hash table*) e di tipo brute force. Tutte queste tipologie hanno come obiettivo l'individuazione di un valore di *hash*. Le principali differenze sono che gli attacchi brute force utilizzano come ingresso della funzione di hash sequenze di caratteri casuali mentre gli attacchi dizionario utilizzano stringhe in lingua naturale ed entrambi richiedono una computazione runtime per calcolare il valore di hash. Le *hash table* sono invece tabelle di valori di hash precedentemente calcolati: questo riduce molto i tempi per scoprire il valore di una password perché partendo da un database di valori di hash è possibile fare una associazione inversa e individuare l'input originale.

Per mitigare questi attacchi è stato necessario introdurre l'utilizzo delle funzioni di *salt*. Queste permettono di aggiugnere all'input della funzione di *hash* un valore generato con una funzione di crittografia sicura andando così a creare dei *message digest* unici. Pertanto l'aggiunta di questo valore di *salt* rende la funzione di *hash* non deterministica.

Nella piattaforma è stata utilizzata come funzione di hash *bcrypt* [20]. Questa funzione richiede un salt, un parametro costo e una chiave primaria (la password) e ha come caratteristica il fatto che il processo di hashing avviene in più cicli (tanti quanto il costo). Questo significa che il processo di generazione sarà tanto più lento quanto è alto

il valore costo passato alla funzione e ciò permette di adeguarne l'efficacia nei confronti della potenza di calcolo dei computer che cresce ogni anno.

Autorizzazione

L'autorizzazione viene usata per garantire la riservatezza e la disponibilità dei dati attraverso l'implementazione di policy finalizzate a restringere l'accesso alle risorse, rendendole disponibili solo agli utenti autorizzati. Le policy definite all'interno della API web si basano sul protocollo di autorizzazione OAuth 2.0 [21] e sull'utilizzo di un modello di controllo sugli accessi.

Nel dettaglio OAuth 2.0 è un framework autorizzativo standard per applicazioni web. Scopo del framework è delegare a un'utenza qualsiasi l'accesso limitato a una risorsa di proprietà di un'entità differente (detta *resource owner*).

Gli attori coinvolti nell'architettura del framework sono:

- *Resource Owner*: il proprietario della risorsa esposta. Può essere una applicazione o un utente.
- *Resource Server*: è il server che detiene la risorsa esposta.
- *Authorization Server*: è il server o il modulo applicativo che rilascia i token di accesso (detti *access token*) al client dopo aver autenticato con successo il *resource owner* e aver ottenuto l'autorizzazione.
- *Client*: è l'applicazione che richiede l'accesso alla risorsa. Può essere sia una applicazione *client-side* che *server-side*. Deve registrarsi presso l'*authorization server* per ottenere delle credenziali: *client secret* e *client identifier*.

Altri elementi significativi sono l'*access token* e l'*authorization grant*.

Il primo è una stringa che rappresenta le credenziali necessarie per ottenere l'accesso alla risorsa protetta sul resource server. Pratica comune è utilizzare il formato Json web Token [22] (JWT) che permette di codificare un oggetto JSON in un token che può essere cifrato e firmato con vari algoritmi (HMAC, RSA). Questo permette di trasmettere in modo sicuro e compatto delle informazioni fra due interlocutori garantendone integrità e confidenzialità.

L'*authorization grant* rappresenta le credenziali che verificano l'autorizzazione fornita dal *resource owner* e usate dal *client* per ottenere un *access token*. Il framework ne definisce quattro tipologie: *authorization code*, *implicit*, *resource owner password credentials* e *client credentials*.

Il processo di base definito dal protocollo per autenticare il *client* e fornire un *access token* è il seguente (vedi Figura 4.3):

1. Il *client* richiede l'autorizzazione al *resource owner*.
2. Il *client* riceve un *authorization grant*, ovvero delle credenziali che rappresentano l'autorizzazione del *resource owner*.
3. Il *client* richiede un *access token* autenticandosi con l'*authorization server* e fornendo l'*authorization grant*.
4. L'*authorization server* autentica il client validando l'*authorization grant*. Se è valido fornisce un *access token*.
5. Il *client* richiede l'accesso alla risorsa protetta presente nel *resource server* e si autentica presentando l'*access token*.
6. Il *resource server* valida l'*access token* e, se è valido, gestisce la richiesta.

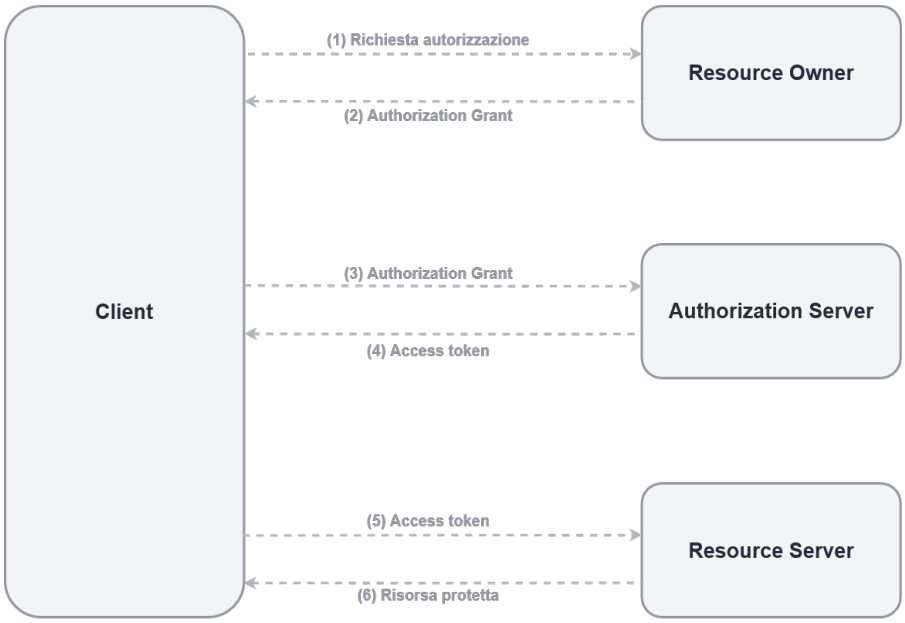

Figura 4.3: Flusso del protocollo OAuth 2.0

Nella piattaforma è stato deciso di utilizzare l'*authorization grant* di tipo *authorization code* per supportare le procedure di registrazione degli utenti attraverso i provider di terze parti (Google e Facebook).

Questo meccanismo disaccoppia completamente le credenziali utente dall'autorizzazione per accedere. In particolare si basa sull'introduzione di un nuovo elemento chiamato *authorization code* che viene ottenuto grazie all'*authorization server*, che si comporta come intermediario fra il *client* e il *resource owner*. Invece di richiedere l'autorizzazione al *resource owner*, il *client* indirizza quest'ultimo verso un *authorization server* che richiede al *resource owner* di autenticarsi. Se l'autenticazione ha successo l'*authorization server* crea un *authorization code* e reindirizza il *resource owner* al *client*.

Questo procede con l'invio di una richiesta di autorizzazione, diretta all'*authorization server*, con la quale può ottenere l'*access token*. A questo punto il *client* effettuerà la richiesta per ottenere la risorsa protetta e ritornerà queste informazioni al *resource owner*. I benefici di questa tipologia di *authorization grant* sono dovuti al fatto che il *resource owner* si autentica direttamente con il *authorization server* e di conseguenza le sue credenziali non vengono mai condivise con il *client*.

[schema oauth flow]

Nella piattaforma è stato poi deciso di sfruttare gli *access token* anche per gestire le autorizzazioni degli utenti interni alla piattaforma. Nello specifico vengono usati insieme a un *refresh token* che permette di ottenere un nuovo *access token* quando quest'ultimo scade. Questo evita di dover richiedere all'utente di autenticarsi e riottenere l'autorizzazione, ottimizzando così il carico del lavoro sul server web e l'esperienza utente. In particolare l'*access token* verrà passato come valore nell'header della richiesta per accedere alle risorse che necessitano tale livello di autorizzazione mentre il *refresh token* verrà passato come cookie HttpOnly, per evitare che sia accessibile agli script lato client e mitigare gli attacchi XSS più comuni.

Inoltre è presente anche una implementazione di un modello di controllo sugli accessi attraverso il quale è possibile definire delle *access control list* che determinano chi ha il diritto di accedere a una risorsa. Nel dettaglio è possibile rendere una risorsa accessibile solo al proprietario, a un qualsiasi utente autenticato o a tutti. Inoltre è anche possibile definire il controllo di accesso sulla base del ruolo di un utente.

4.2.5 Demo module

Il modulo demo permette agli utenti di richiedere l'accesso a una servizio generico in versione di prova.

In fase di progettazione si è vista la necessità di erogare questo tipologia di servizi in quanto è plausibile che una azienda che eroga servizi web voglia dare la possibilità a futuri clienti di vivere l'esperienza data dalla loro piattaforma.

Questo modulo offre agli utenti un endpoint con il quale è possibile creare un profilo di prova e ricevere una mail contente un link temporaneo attraverso il quale si potrà ottenere un token di accesso. L'utente avrà così modo di accedere per un periodo di prova al servizio richiesto.

4.2.6 User module

Il modulo user implementa le funzionalità necessarie per la gestione degli utenti. Permette di accedere alle proprie informazioni, modificarle, aggiornare la password e cancellare

il proprio account. Inoltre è presente una implementazione del modello User in figura 4.1 definendo anche un servizio utile per effettuare richieste al database.

Nella piattaforma esistono quattro categorie di utenti: gli user, gli admin, i manager e gli utenti demo. Gli utenti sono membri della piattaforma e possono accedere e modificare solo le proprie informazioni, i manager hanno privilegi maggiori, gli admin sono coloro che hanno accesso a tutta la piattaforma mentre gli utenti demo sono coloro che hanno richiesto l'accesso a un servizio di prova.

4.2.7 Mail module

Il modulo mail ha la responsabilità di gestire l'invio di messaggi al *message broker* per richiedere l'invio di una mail.

Viene esposto un servizio che permette ad altri componenti di richiedere l'invio di mail al microservizio mailer. In particolare è presente una interfaccia che definisce i campi che devono essere presenti nel messaggio che verrà recapitato al microservizio mailer per permettergli di capire quale template renderizzare e come inviare la mail.

Questo servizio si occuperà poi di incapsulare tutti i dettagli necessari per poter comunicare con il *message broker*. In particolare viene utilizzato un modulo wrapper della libreria *amqplib* che permette di comunicare con il *message broker* utilizzato nella piattaforma. Con l'utilizzo di questo modulo è possibile definire un meccanismo interno per inviare le mail attraverso l'uso di una interfaccia. Inoltre si mantiene il componente aperto alle estensioni in caso si voglia utilizzare un protocollo diverso per comunicare con il *message broker*.

4.3 Message broker

4.3.1 Descrizione generale

Un *message broker* è un software che permette di fare interagire varie applicazioni in un contesto distribuito. Il suo compito principale è permettere a servizi eterogenei di comunicare con successo. Viene solitamente utilizzato per la comunicazione fra microservizi in una architettura distribuita e per operazioni che richiedono molto tempo in modo da ridurre il carico di lavoro sul web server.

Nel dettaglio un *message broker* è un software che implementa il paradigma *Message Oriented Middleware* (MOM) e offre servizi per distribuire i messaggi (routing e politiche di recapito). Ciò significa che verrà usato come *gateway* di livello applicativo in un sistema in cui le applicazioni comunicano utilizzando messaggi che vengono inseriti in code e dove vengono utilizzati dei nodi di smistamento per distribuirli verso la

destinazione. Questa tipologia di sistemi è detta *message-queuing system* [23] in cui è garantita la persistenza grazie all'utilizzo delle code e la possibilità di creare topologie punto-punto o punto-multipunto con in nodi di smistamento. In particolare avremo che il *message broker* riceve dei messaggi da applicazioni dette *publishers* e li indirizzerà verso applicazioni dette *consumers* che processeranno questi messaggi.

Nella piattaforma è stato deciso di utilizzare un broker RabbitMQ [24] usando come protocollo di messaggistica l'*Advanced message Queuing Protocol 0-9-1* (AMQP) [25], per permettere alla API d'interagire con il microservizio del Mailer.

4.3.2 Protocollo di messaggistica

L'AMQP 0-9-1 è un protocollo di livello applicativo che permette a delle applicazioni client d'interagire con i *message broker*. Per rendere completa questa interazione è necessario che fra le parti venga specificata la sintassi da usare e la semantica dei messaggi per i servizi offerti dal broker. Il protocollo AMQP li definisce entrambi:

- *Advanced Message Queuing protocol* (AMQP): è il protocollo che definisce la sintassi dei messaggi che permettono ai client d'interagire con il broker.
- *Advanced Message Queuing model* (AMQ model): definisce un insieme di elementi e standard che devono essere implementati dal broker in modo tale da supportare le interazioni fra *publishers* e *consumers*.

Il protocollo AMQP è un protocollo di livello applicativo diviso in due layer:

- *Functional Layer*: definisce i comandi che permettono alle applicazioni di svolgere operazioni sul broker
- *Transport Layer*: gestisce il multiplexing del canale, il framing, la codifica e il trasporto dei messaggi. Permette anche la gestione degli errori, la comunicazione asincrona e funzionalità di heartbeat.

Il modello AMQ definisce tre tipologie di componenti, che vengono poi connessi e processati dal broker per erogare i servizi richiesti. Nel dettaglio:

- *Exchange*: componente che indirizza i messaggi alle code
- *Message queue* ("coda di messaggi"): una struttura dati che memorizza i messaggi
- *Binding*: regola che definisce la relazione tra l'*exchange* e la coda.

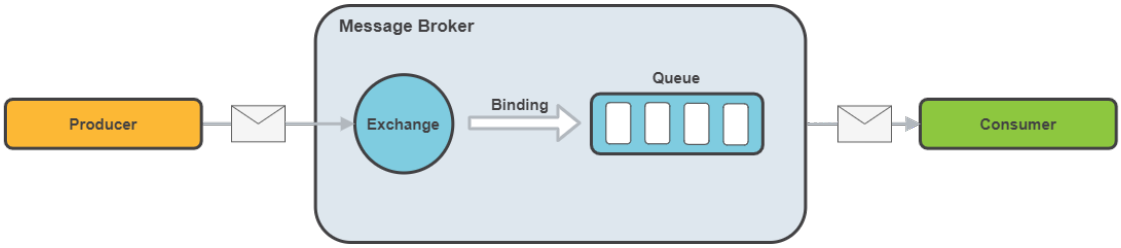

Figura 4.4: Modello AMQ

4.3.3 Principi di design

Nella piattaforma è stato utilizzato un broker RabbitMQ per supportare l'interazione tra API e Mailer e disaccoppiare la fase di gestione della richiesta utente dalla fase di renderizzazione e invio email al destinatario. Vista la possibilità di dover gestire numerose email contemporaneamente, è stato deciso di utilizzare il modello a *consumers* concorrenti (*Competing Consumers Pattern*) [26]. Questo prevede la creazione di molteplici *consumers* collegati a uno stesso canale in modo tale che questi possano processare più messaggi contemporaneamente. Quando un messaggio arriva sul canale, uno qualsiasi dei *consumers* potrebbe riceverlo e processarlo andando così a competere l'uno con l'altro.

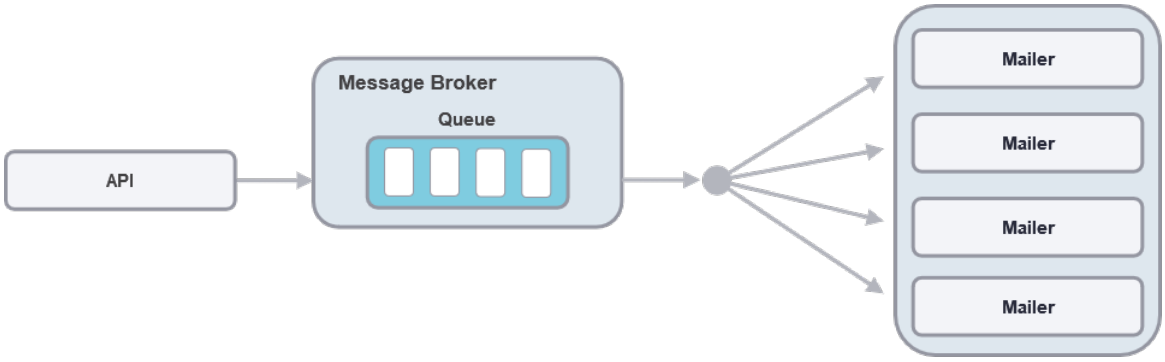

Figura 4.5: Rappresentazione semplificata del *Competing Consumer pattern*

Per implementare questo modello si è deciso di utilizzare i seguenti elementi:

- *Task Queue*: coda usata per distribuire su più *consumers* varie operazioni che richiedono molto tempo.
- Distribuzione *Round Robin*: il broker manderà un messaggio a ogni *consumer* in maniera sequenziale (ogni *consumer* riceverà in media lo stesso numero di messaggi). Questo garantisce che tutti i messaggi verranno recapitati almeno una volta.

- *Message acknowledgment* (ACK): vengono utilizzati degli ACK per confermare la corretta elaborazione di un messaggio da parte di un *consumer*. Se l'ACK non viene inviato al broker, il messaggio verrà rimesso in coda.

Grazie a questi elementi è stato possibile creare una architettura di messaggistica in grado di scalare orizzontalmente con facilità (aggiungendo dei *consumers*), disaccoppiata, affidabile e resiliente.

4.4 Microservizio Mailer

4.4.1 Descrizione generale

Il microservizio mailer è il componente della piattaforma che permette di inviare delle email.

L'applicazione è basata su Node ed è stato utilizzato il framework Nest. Si interfaccia con l'API web, attraverso un *message broker* RabbitMQ, per recuperare i messaggi contenenti le informazioni utili per generare la email e con il servizio esterno Simple Email Service per inviare le email.

In fase di progettazione è stato deciso di disaccoppiare questa funzionalità dalla API web a causa della complessità necessaria per effettuare il render di una mail e per inviare il messaggio al destinatario. Inoltre, essendo il microservizio un componente indipendete dalla API web, sarà possibile scalarlo orizzontalmente in modo da poter gestire in maniera più efficace le operazioni richieste.

4.4.2 Principi di design

Il microservizio è composto da tre elementi principali: *consumer service*, *template service* e *transport service*.

Il *consumer service* incapsula i dettagli relativi all'interazione con il *message broker* RabbitMQ. Questo servizio si occupa quindi di recuperare i messaggi dalla coda del *message broker*, per poterli elaborare, e di comunicare al broker un messaggio di tipo ACK per confermare la corretta elaborazione del messaggio.

Il *template service* si occupa di effettuare l'operazione di rendering di una mail a partire dal messaggio recuperato. Per supportare questa operazione viene usato Nunjucks, un template engine che permette di creare documenti dinamici sulla base di un set di campi prefissati. L'utilizzo di questa tecnologia ha permesso di definire la struttura di un documento HTML, il cui contenuto verrà poi creato dinamicamente sulla base delle informazioni contenute nel messaggio, e che verrà usato come corpo della email da inviare.

Infine il *transport service* ha il compito di incapsulare la logica di invio della mail. Per supportare questa operazione è stato deciso di integrare il servizio esterno Simple Email Service, facente parte degli Amazon Web Services.

[schemino]

4.5 Microservizio Doc

4.5.1 Descrizione generale

Il microservizio Doc rende accessibile la documentazione della piattaforma. L'erogazione di questo servizio avviene grazie a un file conforme alla specifica OpenAPI [27], un insieme di direttive che permettono a una macchina di descrivere, produrre e utilizzare un servizio REST.

In particolare questo microservizio fornisce una interfaccia interattiva realizzata con SwaggerUI [28], una tecnologia che partendo dal file OpenAPI genera un client in grado comunicare con l'API, direttamente dal browser. L'interfaccia è accessibile attraverso l'API web che si comporta come API gateway, andando a fare il proxy delle richieste verso il microservizio Doc.

Durante lo sviluppo della piattaforma la possibilità di interagire con l'API web in modo rapido ha permesso di testare direttamente il comportamento degli endpoint. Inoltre il file di specifica, erogato dal microservizio, potrà poi essere utilizzato dal team front-end per ridurre i tempi necessari per integrare l'API nella applicazione client attravero Swagger Codegen [29], un generatore di librerie client.

Capitolo 5

Gestione e rilascio dei componenti

5.1 Descrizione Generale

Le operazioni di gestione e rilascio dei componenti della piattaforma è stato fatto con l'obiettivo di seguire le pratiche DevOps per ottimizzare le fasi di integrazione, distribuzione e deployment del software. In particolare è stato usato il metodo della containerizzazione per gestire l'installazione dei componenti nei vari ambienti ed è stata implementata una pipeline per supportare il processo di *continuous integration* e *continous deployment*.

5.2 Container

5.2.1 Introduzione

Tutti i componenti della piattaforma sono gestiti con il supporto di container utilizzando Docker.

Un container Docker è un ambiente isolato in cui può essere messa in esecuzione una applicazione. Questi vengono eseguiti direttamente dal kernel della macchina host e non richiedono un *hypervisor* come le macchine virtuali. Questo rende i container molto più leggeri. Altra caratteristica è che i singoli container, quando eseguiti, vengono gestiti come processi separati. Questo significa che godono di un elevato livello di isolamento con la macchina host e gli altri processi garantendone la sicurezza. Inoltre la gestione delle applicazioni in container permette di velocizzare e rendere più efficente il ciclo di sviluppo del software in quanto ne aumenta la portabilitità, rendendo più semplice le fasi di build, test e deploy.

Il meccanismo che porta alla creazione di un container Docker si basa sui seguenti elementi:

- Docker Engine: server o processo deamon che effettua tutte le operazioni necessarie per gestire container Docker.
- Docker CLI: interfaccia a riga di comando con cui interagire con il server Docker.
- Immagine: elemento che incapsula un insieme di direttive e di software che permettodno di creare il container.
- Registro: permette di salvare le immagini in raccolte pubbliche o private. Viene usato DockerHub, un *image registry* accessibile da remoto.
- Container: elemento che include un'immagine. Può essere messo in esecuzione rendendo accessibile l'applicazione.

Notevole importanza risiede quindi nelle immagini in quanto permettono di definire il contenuto del container.

Nel seguito verrà descritto come sono state create e definite le immagini per la containerizzazione dei componenti della piattaforma e come avviene la loro orchestrazione.

5.2.2 Creazione immagini

Le immagini vengono create con dei file speciali detti *DockerFile*. Questi contengono le direttive per includere il codice sorgente, le dipendenze ed eseguire script e vengono utilizzati dai container per mettere in esecuzione l'applicazione. Nel dettaglio ciò che avviene è che ogni istruzione presente nel *DockerFile* aggiunge un layer all'immagine che viene quindi realizzata attraverso vari stage. Questo significa che ad ogni layer sarà associato un artefatto diverso rispetto al layer precedente e bisognerà quindi prestare attenzioni agli elementi effettivamente coinvolti nelle varie operazioni.

Per ottimizzare la gestione e la dimensione delle immmagini è quindi necessario ricordarsi di ripulire gli elementi non necessari per il layer successivo. A tal fine nella piattaforma è stato deciso di utilizzare il *multi-stage build pattern* per gestire i *DockerFile*. Questo permette di creare basi differenti su cui costruire i propri artefatti. Ciò significa che è possibile fare riferimento gli artefatti realizzati nei singoli stage invece di gestire il risultato finale. Questo è vantaggioso in quanto permette di operare effettivamente solo con gli artefatti necessari.

Nel listing 1 viene riportato il *DockerFile* utilizzato per creare le immagini dei componenti della piattaforma. Questo prevede l'utilizzo di sei basi differenti: *node*, *dev*, *source*, *test*, *preprod* e *prod*. La base *node* è una immagine basata sul progetto Alpine Linux contente Node.js e gli strumenti necessari per creare l'ambiente in cui sarà possibile eseguire l'applicazione. La base *dev* copia i file *package.json* e *package-lock.json*

contenenti l'elenco delle dipendenze dei componenti della applicazione poi con il comando *npm intall* le installa. La base *source* effettua la build dei vari componenti mentre la base *test* utilizza lo stage *source* per eseguire i test. Questo permetterà di verificare il comportamento dei componenti durante l'esecuzione delle procedure automatizzate di integrazione e deploy del software. La base *preprod*, partendo dall'artefatto generato dalla stage *dev*, elimina le dipendenze non necessarie per l'ambiente di produzione ottimizzando così la dimensione dell'immagine che verrà usata come base nello stage *prod*. Questo copia i file compilati, le dipendenze e i file di specifica per poi includere la direttiva per eseguire i componenti.

Listing 1: *DockerFile*

```
FROM node:15-alpine as node

FROM node as dev
WORKDIR /app
COPY package*.json ./
RUN npm ci

FROM dev as source
COPY . ./
RUN npm run build:all

FROM source as test
RUN npm run test

FROM dev as preprod
RUN npm prune --production

FROM node as prod
ENV NODE_ENV production
WORKDIR /app
COPY --from=preprod /app/package*.json ./
COPY --from=preprod /app/node_modules ./node_modules
COPY --from=source /app/dist ./dist
USER node
ENTRYPOINT ["/bin/sh"]
```

5.2.3 Orchestrazione container

L'esecuzione dei i singoli componenti della piattaforma avviene in container separati utilizzando Docker Compose. Questo è un servizio di orchestrazione di container che permette di automatizzare la configurazione, la coordinazione e la gestione dei servizi. Docker compose si basa su un file di specifica in formato YAML, il *docker-compose.yml* file, che permette di definire un insieme di container da avviare e le loro proprietà a runtime. Ogni contaier viene considerato un servizio che interagisce con gli altri container secondo le direttive specificate. Tutti i container verranno poi eseguiti e configurati con il comando *docker-compose up*.

Nella piattaforma ci saranno i servizi api, doc e mailer che costiuiscono i componenti funzionali. Poi ci sarà il servizio database che si basa su una immagine Mongo e un servizio *message broker* basato su una immagine RabbitMQ. Tutti i servizi sono configurati per riavviarsi automaticamente in caso di errore. Nel caso del servizio database e *message broker* è stato creato anche uno script per eseguire l'*healthcheck* in modo da verificare che l'avvio sia avvenuto con successo.

L'utilizzo di Docker Compose permette l'orchestrazione di vari container in modo rapido ed efficace. Fornisce la possibilità di gestire manualmente i singoli servizi e interconneterli tra loro. Offre inoltre la possibilità di specificare quanti container avviare per un servizio in modo da poterlo scalare facilmente.

5.3 CI/CD

5.3.1 Introduzione

CI/CD è un metodo per la distribuzione frequente di software ai clienti, che prevede l'utilizzo di meccanismi di automazione nelle varie fasi di sviluppo del prodotto che si vuole rilasciare. L'acronimo CI/CD ha vari significati ma solitamente fa riferimento a dei concetti di integrazione, distribuzione e deployment continuo.

In particolare *CI* fa riferimento all'integrazione continua. Questo è un processo di automazione utile agli sviluppatori in cui le nuove modiche apportate al codice sorgente del software vengono sottoposte a controlli prima di essere pubblicate nel repository condiviso dell'organizzazione.

Il termine *CD* può assumere un duplice significato: distribuzione continua e/o deployment continuo. Solitamente il processo di distribuzione continua porta alla pubblicazione del nuovo software sul repository condiviso dell'organizzazione mentre il deployment continuo fa riferimento al suo rilascio automatico all'ambiente di produzione e quindi al cliente. Questi sono concetti correlati e spesso vengono usati in modo inter-

cambiabile in quanto fanno riferimento alla automatizzazione delle procedure successive a quelle previste dall'integrazione continua.

Il metodo CI/CD implica quindi la definizione di una serie di processi automatici che devono essere eseguiti per fornire una nuova versione del software al cliente in maniera rapida e sicura. Nella piattaforma è stato possibile applicare questa metodologia grazie alla definizione di una pipeline CI/CD che ha introdotto l'automazione nella metodologia di lavoro adottata nello sviluppo della piattaforma, permettendo di migliorare l'efficenza e la rapidità con cui le modifiche vengono integrate e rilasciate al cliente.

5.3.2 Pipeline CI/CD

La pipeline CI/CD implementata nella piattaforma suddivide i processi di integrazione e deployment continuo in un sottoinsieme di attività distinte.

Il flusso di integrazione continua è stato progettato con l'obiettivo di favorire la collaborazione contemporanea di più sviluppatori sul singolo prodotto mantenendo però un elevato livello di indipendenza tra loro. Ciò significa che ogni sviluppatore apporta modifiche, in modo indipendente, al software e le pubblica nel repository condiviso. Tuttavia questa modalità di integrazione potrebbe portare alla generazione di conflitti fra le modifiche apportare dai vari sviluppatori.

Per risolvere questo problema è stato deciso di implementare un processo di convalida tramite il quale è possibile garantire la compatibilità fra gli aggiornamenti apportati al software dai membri del team. Questo processo viene eseguito ogni volta che uno sviluppatore effettua la pubblicazione dei *commit* sul repository condiviso e, nel nostro caso, avvia la compilazione del software e l'esecuzione di unit e integration test. Questo permette di verificare che le nuove modifiche non introducano problemi nel software.

Il flusso di deployment continuo è invece finalizzato ad automatizzare il rilascio di software pronto ad essere distribuito nell'ambiente di produzione. Nello sviluppo della piattaforma il processo di deployment continuo prevede la creazione di una *build production ready* e la sua pubblicazione sul *production server*. Questo processo automatico ha il vantaggio di rendere attiva una modifica apportata da uno sviluppatore poco dopo che è stata scritta. Ciò permette di ricevere un feedback immediato dal cliente e con una cadenza più costante; potendo così apportare le dovute modifiche al software prima di procedere all'implementazione di nuove funzionalità.

Nella piattaforma queste procedure di automazione sono state realizzate sfruttando Jenkins, un *automation server* che permette di definire un flusso di operazioni da eseguire per supportare i processi di integrazione e deployment continuo. La pipeline viene eseguita ogni volta che un commit viene pubblicato nel repository condiviso sulla piattaforma GitLab ed è stata suddivisa in tre fasi: *build*, *test* e *deploy*. La fase di

build prevede il checkout del codice dal repository remoto con la successiva build dei componenti attraverso lo strumento Docker Compose. La fase di *test* viene utilizzata per validare le modidiche apportate al codice. Al termine si ha la fase di *deploy* in cui vengono pubblicate le immagini sul registro privato e vengono installati gli strumenti necessari sul *production server* per eseguire la piattaforma. Anche in questo caso si fa affidamento a Docker Compose per mettere in esecuzione i container sul *production server*. In caso di successo o fallimento della pipeline è stato inoltre predisposto un meccanismo di messagistica per notificare gli sviluppatori riguardante l'esito della procedura.

Questo processo ha permesso di rendere lo sviluppo più efficente e ha inoltre contribuito ad aumentare la sicurezza e la stabilità del software.

Capitolo 6

Conclusioni

6.1 Valutazioni complessive

La piattaforma software è stata realizzata con lo scopo di divenire una piattaforma basata sul cloud in grado di soddisfare le esigenze di una vasta gamma di clienti attraverso l'implementazione di funzionalità personalizzate. Durante l'analisi del prodotto realizzato sono state descritte le tecnologie e gli strumenti utilizzati per implementare i componenti fondamentali della piattaforma nonchè il metodo di lavoro e gli automatismi adottati per erogare agilmente il software al cliente.

Un esempio di utilizzo della piattaforma risiede nella Medical Adaptive Platform (MAP), un nuovo servizio cloud based erogato dall'azienda Fama Labs. Questo è una suite di soluzioni digitali per le Società Scientifiche, aziende biofarmaceutiche e professionisti sanitari che offre servizi per gestire l'amministrazione della società, organizzare eventi di formazione a distanza, corsi e congressi.

La piattaforma realizzata verrà inoltre sfruttata dall'azienda Fama Labs in una applicazione interna per il management dei servizi erogati e delle risorse.

6.2 Sviluppi futuri

Sviluppi futuri riguardano il tema della scalabilità e un aumento del livello di disaccoppiamento tra i servizi erogati per favorire maggiormente una architettura orientata ai microservizi. In particolare, per quanto riguarda la scalabilità, verranno introdotte soluzioni di orschestrazione dei container basate su tecnologie come Docker Swarm o Kubernetes con lo scopo di poter fornire maggiori opzioni al cliente in fase contrattuale. Infatti in questa prima versione della piattaforma si ha che tutti i componenti risiedono in un unico host e nonostante sia possibile scalare i singoli servizi i limiti dati da questa situazione sono evidenti. Verranno quindi progettate soluzioni alternative in cui

in componenti verranno messi in esecuzione su host separati per poter offrire un livello di scalabilità orizzontale più elevato sui servizi di maggiore interesse per il business del cliente.

Fonti bibliografiche e sitografia

[1] (2016) Regulation (eu) 2016/679, general data protection regulation. european commission. European Commission. Visitato il 08/2021. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679

[2] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," 2001, visitato il 09/2021. [Online]. Available: http://www.agilemanifesto.org/

[3] Typescript. Sito web typescript. Visitato il 08/2021. [Online]. Available: https://www.typescriptlang.org/

[4] Node.js. Sito web node.js. Visitato il 08/2021. [Online]. Available: https://nodejs.org/en/about/

[5] NPM. Documentazione npm. Visitato il 08/2021. [Online]. Available: https://docs.npmjs.com

[6] NestJS. Documentazione nestjs. Visitato il 08/2021. [Online]. Available: https://docs.nestjs.com/

[7] Jest. Sito web jest. Visitato il 08/2021. [Online]. Available: https://jestjs.io/

[8] MongoDB. Sito web mongodb. Visitato il 08/2021. [Online]. Available: https://www.mongodb.com/

[9] IBM. What is mongodb. Visitato il 08/2021. [Online]. Available: https://www.ibm.com/cloud/learn/mongodb

[10] AWS. Sito web aws-ses. Visitato il 08/2021. [Online]. Available: https://aws.amazon.com/it/ses/

[11] Git. Sito web git. Visitato il 08/2021. [Online]. Available: https://git-scm.com/

[12] S. Chacon and B. Straub, *Pro git: Everything you need to know about Git*, 2nd ed. Apress, 2014. [Online]. Available: https://git-scm.com/book/en/v2

[13] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. K. Smith, C. Winter, and E. Murphy-Hill, "Advantages and disadvantages of a monolithic repository: A case study at google," *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017.

[14] G. Brito, R. Terra, and M. T. Valente, "Monorepos: A multivocal literature review," *ArXiv*, vol. abs/1810.09477, 2018.

[15] Nrwl. Sito web nx. Visitato il 08/2021. [Online]. Available: https://nx.dev/

[16] ——. Sito web nrwl. Visitato il 08/2021. [Online]. Available: https://nrwl.io/

[17] Docker. Sito web docker. Visitato il 08/2021. [Online]. Available: https://www.docker.com/

[18] Jenkins. Sito web jenkins. Visitato il 08/2021. [Online]. Available: https://www.jenkins.io

[19] Typegoose. Sito web typegoose. Visitato il 08/2021. [Online]. Available: https://typegoose.github.io/typegoose

[20] N. Provos and D. Mazières, "A future-adaptive password scheme," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '99. USA: USENIX Association, 1999, p. 32, visitato il 08/2021.

[21] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749, Oct. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6749.html

[22] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, May 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7519.txt

[23] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2nd Edition)*. USA: Prentice-Hall, Inc., 2006.

[24] RabbitMQ. Sito web rabbitmq. Visitato il 08/2021. [Online]. Available: https://www.rabbitmq.com

[25] S. Aiyagari, M. Arrott, M. Atwell, J. Brome, A. Conway, R. Godfrey, and et al., "Advanced message queuing protocol," 2008. [Online]. Available: https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf

[26] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2019.

[27] Openapi. OpenAPI. Visitato il 09/2021. [Online]. Available: https://www.openapis.org/

[28] (2016) Swaggerui. SmartBear. Visitato il 09/2021. [Online]. Available: https://swagger.io/tools/swagger-ui/

[29] (2016) Swagger codegen. SmartBear. Visitato il 09/2021. [Online]. Available: https://swagger.io/tools/swagger-codegen/

Elenco delle figure