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

Abstract

Indice

Capitolo 1

Introduzione

1.1 Obiettivo

obiettivo della tesi

1.2 Campo di applicazione

cosa serve

1.3 Panoramica

come è strutturata la tesi

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: web Server che permette di gestire le richieste del client e fornisce una interfaccia REST per erogare i servizi.
- 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.

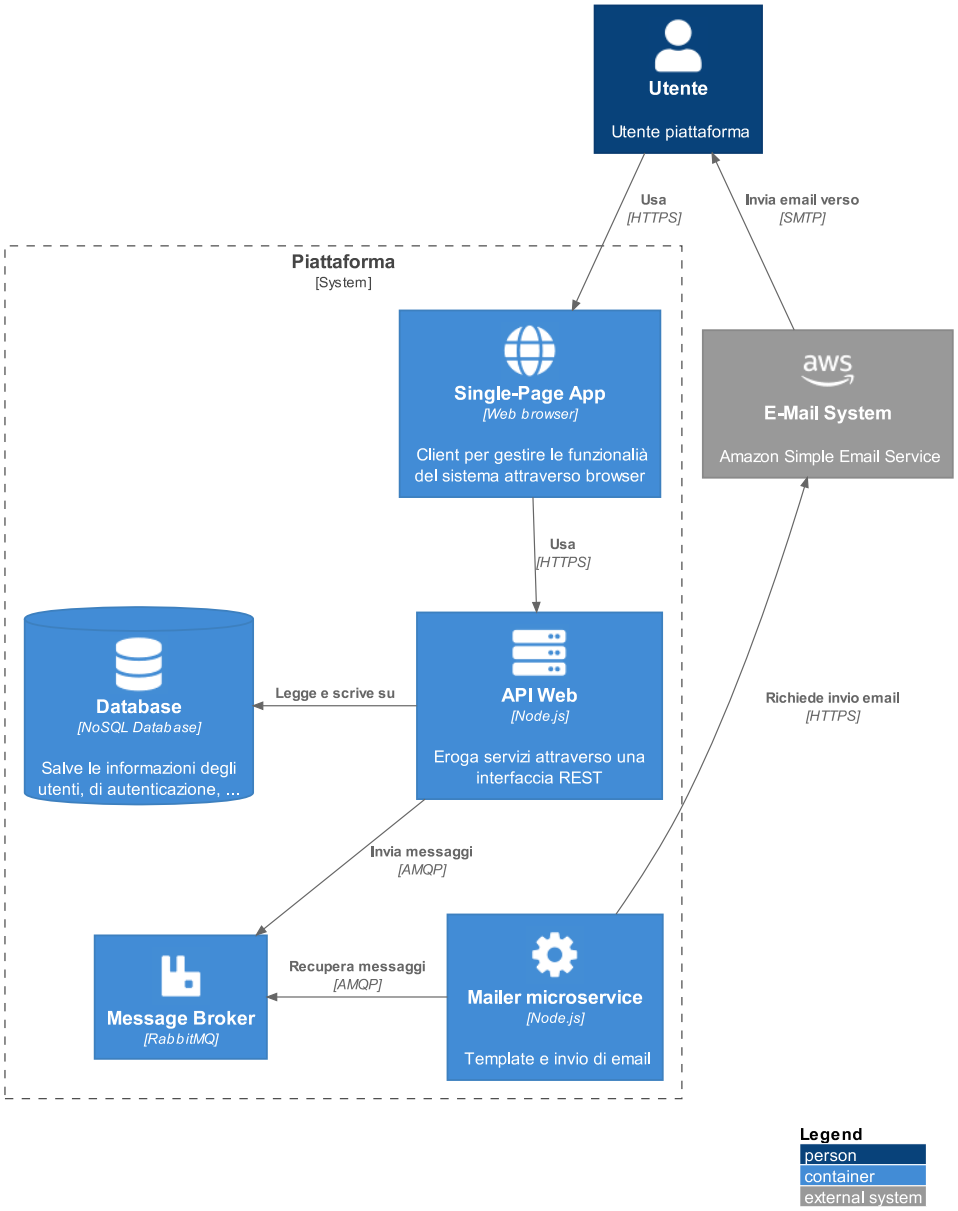

Figura 2.1: Architettura piattaforma

2.2 Macrofunzionalità del sistema

elenco funzionalità dettagliate

2.3 Caratteristiche degli utenti

utenti normali admin

2.4 Vincoli generali

Scalabilità robustezza sicurezza collaborazione

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[1] è un linguaggio open source sviluppato da Microsoft.

È un *super-set* del linguaggio JavaScript, permettendone l'estensione con 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[2] 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 svi-

luppare 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 Node Package Manager

Node Package Manager[3] (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 dipendenze 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 [4] (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.

[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.

3.1.5 Testing

Jest

Jest[5] 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 motivazione principale che ha portato alla scelta delle tecnologie di seguito elencate è il *know-how* aziendale.

3.2.1 Database

MongoDB

MongoDB[6] è un database management system (DBMS) che offre la possibilità di gestire database NoSQL [2] basati sull'utilizzo di documenti flessibili per la gestione di dati in vario formato.[7]

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 ai dati di adattarsi 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. Questi documenti vengono poi raccolti in collezioni che vengono usate come archivi per documenti facenti parte di una stessa categoria d'informazione. 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.

[2] dall'inglese *Not only SQL*; non solo SQL.

3.3 Servizi cloud esterni

In questa sezione verranno descritti i servizi cloud 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 [8] (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 [9] 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. [10]. 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*[11] (*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.[12]

- 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 stato deciso di utilizzare questa strategia e ciò ha permesso al team di sviluppo di operare in modo efficiente e coeso.

Nx

Nx [13] è uno strumento open source, creato dal team Nrwl [14], 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 [15] 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 [16] 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

3.6.1 AWS

Testo

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 e microservizio Mailer.

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 fornendo degli endpoint per poter svolgere varie operazioni.

Essendo la piattaforma ideata come rappresentazione di una base riutilizzabile in vari contesti aziendali è stato deciso d'implementare alcuni servizi generici 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.

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.

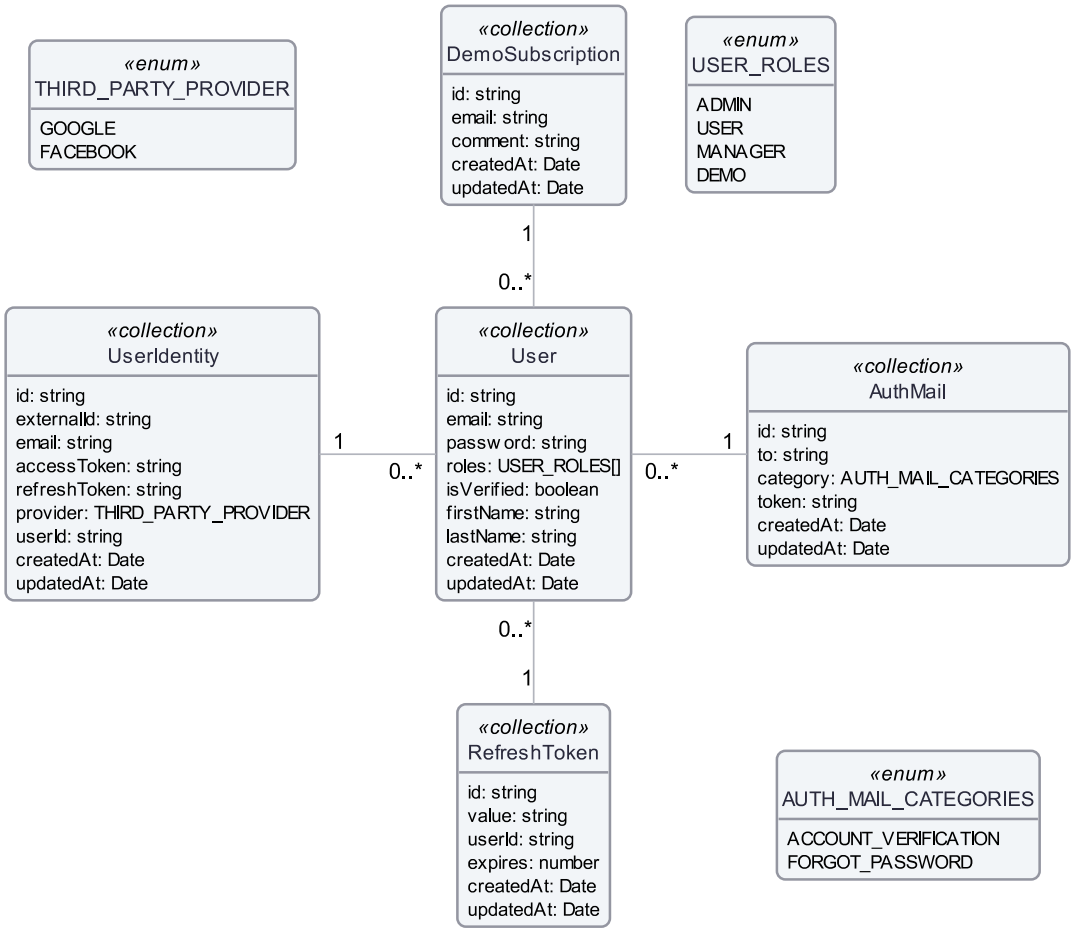

Figura 4.1: Modellazione dati

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[17]: 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.

4.2.3 Autenticazione

L'autenticazione è un metodo che permette l'identificazione degli utenti sulla base di determinate credenziali e la verifica della legittimità di una utenza.

L'implementazione presente all'interno della API web si basa su una coppia di credenziali: email e password.

Per garantire una archiviazione sicura delle password è stato deciso di imporre regole sul formato della password per renderla 'poco' 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 input alla funzione di hash sequenze di caratteri casuali mentre gli attacchi dizionario utilizzano stringhe in lingua naturale ed entrambe 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. L'aggiunta di questo valore di *salt* rende la funzione di *hash* non deterministica ed evita così di generare *message digest* uguali per lo stesso input.

Nella piattaforma è stata utilizzata come funzione di hash *bcrypt*[18]. 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.

4.2.4 Autorizzazione

L'autorizzazione è un metodo che permette di garantire la riservatezza e la disponibilità dei dati. Attraverso l'implementazione di policy è infatti possibile definire dei protocolli che permettono di 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 [19] e sull'utilizzo di un modello di controllo sugli accessi basato sui ruoli dei singoli utenti.

OAuth 2.0

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[20] (JWT) che permette di codificare un oggetto JSON in un token che può essere cifrato e firmato con vari algoritmi (HMAC, RSA, ECDSA). 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:

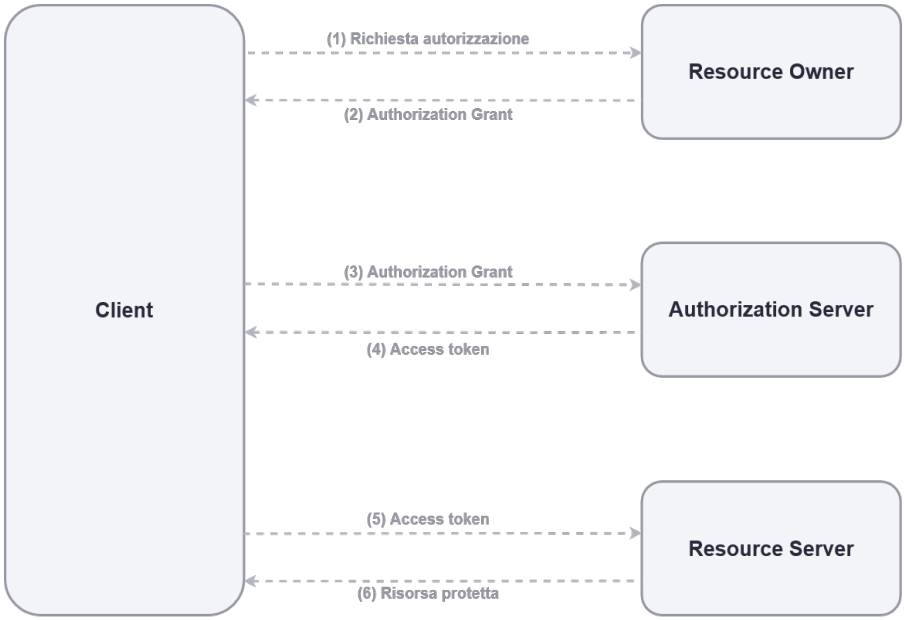

Figura 4.2: Flusso del protocollo OAuth 2.0

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.

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*.

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 dei *refresh token* che permettono di ottenere un nuovo *access token* quando quest'ultimo scade. Questo evita di dover richiedere all'utente di autenticarsi e ottenere l'autorizzazione ottimizzando così il carico del lavoro sul server web e l'esperienza utente.

4.2.5 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. Inoltre 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 4.3) 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.

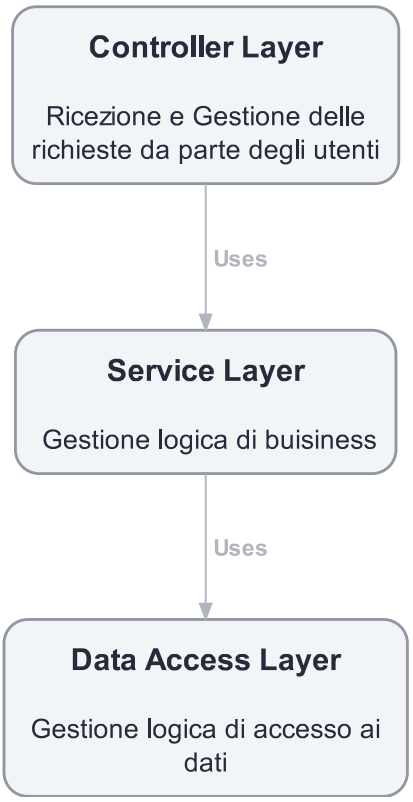

Figura 4.3: Schema riassuntivo della architettura *three-tier*

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. Questi elementi supportano la realizzazione di una architettura scalabile, coesa, disaccoppiata e mantenibile.

4.2.6 Auth Module

testo

4.2.7 Demo module

testo

4.2.8 User module

testo

4.2.9 Mail module

testo

4.3 Mailer microservice

4.3.1 Descrizione generale

testo

4.3.2 Principi di design

testo

4.3.3 Template Service

testo

4.3.4 Transport Service

testo

4.4 Message broker

4.4.1 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*[21] 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[22] usando come protocollo di messaggistica l'*Advanced message Queuing Protocol 0-9-1* (AMQP)[23], per permettere alla API d'interagire con il microservizio del Mailer.

4.4.2 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.

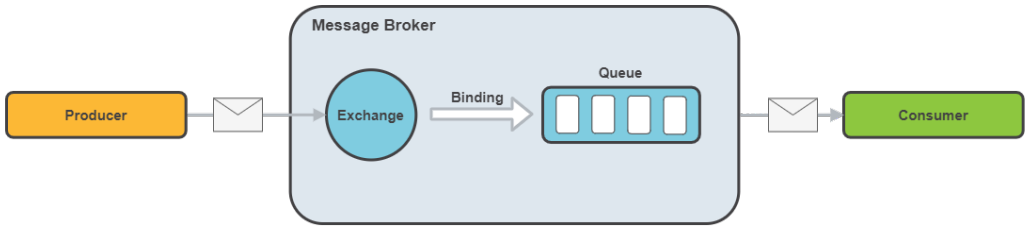

Figura 4.4: Modello AMQ

4.4.3 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)[24]. 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.

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.

Capitolo 5

Distribuzione

5.1 Descrizione generale

5.2 CI/CD pipeline

5.2.1 Descrizione generale

testo

5.2.2 Motivazioni

testo

5.2.3 Integrazione continua

testo

5.2.4 Distribuzione continua

testo

5.2.5 Deployment continuo

Capitolo 6

Conclusioni

6.1 Valutazioni complessive

6.2 Sviluppi futuri

Fonti bibliografiche e sitografia

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

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

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

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

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

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

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

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

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

[10] 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

[11] 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.

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

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

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

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

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

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

[18] 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.

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

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

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

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

[23] 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

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

Elenco delle figure