Politecnico di Torino

Politecnico di Torino

Corso di Laurea in Ingegneria Informatica

A.a. 2025/2026

Sessione di laurea Marzo 2026

Dai Monoliti ai Microservizi: Progettazione e Sviluppo di zMailer

Relatore: Prof. Luca Ardito

Candidato: Francesco Chiaia

Abstract

Negli ultimi anni la crescente complessità dei sistemi software ha spinto numerose aziende a riconsiderare le proprie scelte architetturali, abbandonando progressivamente i sistemi monolitici a favore di architetture a microservizi. La presente tesi si propone di analizzare tale cambiamento attraverso un percorso che, a partire da un inquadramento teorico, giunge alla descrizione dell'implementazione di un caso di studio reale.

La sezione teorica evidenzia alcuni dei limiti delle architetture monolitiche e approfondisce i principi fondamentali dei microservizi. In particolare, l'analisi si concentra sui principali design pattern architetturali e sulle modalità di comunicazione tra questi, confrontando protocolli sincroni con approcci asincroni. Inoltre, vengono analizzate le problematiche di sicurezza tipiche dei sistemi distribuiti e le strategie di migrazione da sistemi legacy. La sezione implementativa documenta la progettazione e lo sviluppo di zMailer, un microservizio per la gestione e l'invio di email, realizzato durante il periodo di stage presso l'azienda Zucchetti S.p.A. Il progetto nasce dalla necessità di estrarre questa specifica funzionalità dall'architettura monolitica esistente, con l'obiettivo di trasformarla in un servizio dedicato indipendente.

I risultati ottenuti evidenziano come un'architettura a microservizi correttamente progettata possa determinare il significativo miglioramento di una funzionalità critica del sistema precedente. Tra i contributi più rilevanti si annoverano l'implementazione di un sistema di gestione della multi-tenancy, che garantisce il completo isolamento dei dati tra realm differenti, ed il notevole incremento delle prestazioni rispetto alla soluzione originaria.

Indice

Elenco delle figure

Glossario

Realm

Un realm è un dominio definito per isolare dati e configurazioni all'interno di un sistema multi-tenant. Ogni realm rappresenta un'unità logica separata, consentendo a diverse organizzazioni o gruppi di utenti di operare in modo indipendente all'interno della stessa infrastruttura. Un realm nel microservizio zMailer è definito dalla coppia di identificatori: tenant_id e application_id .

LTS

Supporto a lungo termine, usato per programmi che indicano che una specifica versione fornirà un supporto per un periodo più lungo di tempo.

MVC

Model-View-Controller è un modello architetturale che separa un'applicazione in tre componenti interconnessi: Model, View e Controller, migliorando la manutenibilità e la scalabilità.

SDK

Software Development Kit, un insieme di strumenti di sviluppo software che consente la creazione di applicazioni per una piattaforma specifica.

ACID

Acronimo di Atomicity, Consistency, Isolation, Durability. Insieme di proprietà che garantiscono l'affidabilità delle transazioni in un database relazionale.

API

Insieme di regole e convenzioni che definiscono il modo in cui diversi programmi software comunicano tra loro.

CI/CD

Continuous Integration/Continuous Deployment, un insieme di pratiche di sviluppo software che prevedono l'integrazione continua del codice e la distribuzione automatica delle applicazioni.

CQRS

Pattern architetturale che separa le operazioni di lettura (Query) da quelle di scrittura (Command) in componenti distinti.

DKIM

Standard di autenticazione delle email che permette al mittente di firmare digitalmente i messaggi tramite crittografia asimmetrica. Il destinatario verifica l'autenticità del messaggio usando la chiave pubblica pubblicata nel DNS del dominio mittente.

Capitolo 1

Introduzione

Negli ultimi anni l'evoluzione delle architetture software ha portato ad un cambiamento significativo nel modo in cui le applicazioni vengono progettate e sviluppate. L'iniziale approccio monolitico ha mostrato limiti evidenti in termini di scalabilità, manutenibilità e flessibilità. Le applicazioni monolitiche, caratterizzate da un'unica base di codice fortemente accoppiata, sono diventate sempre più complesse da gestire, mano a mano che sono cresciute in dimensioni e funzionalità.

È Proprio in questo contesto che emerge l'architettura a microservizi la quale suddivide le applicazioni in una collezione di servizi piccoli e indipendenti. Tale approccio facilita notevolmente lo sviluppo e la scalabilità delle applicazioni in quanto divide sistemi complessi in unità più piccole e gestibili, dove ognuna di essa compie una specifica funzione e segue un proprio ciclo di vita di sviluppo e deployment [1]. Questa autonomia dei servizi si traduce in team più produttivi, rilasci più frequenti e una maggiore resilienza del sistema complessivo.

La tesi, dunque, si colloca esattamente in tale ambito e analizza lo sviluppo di zMailer, un microservizio dedicato alla gestione e all'invio di email in ambiente enterprise. Esso è nato dall'esigenza concreta di superare i limiti architetturali del sistema monolitico esistente, dimostrando come la migrazione a microservizi possa trasformare una funzionalità critica in un servizio scalabile, manutenibile e performante.

1.1 Contesto e Motivazioni

Il progetto è stato sviluppato durante il tirocinio presso l'azienda *Zucchetti S.p.A.*, una delle principali software house italiane fondate nel 1978. Come spesso accade nelle enterprise consolidate, l'azienda dispone di un software interno strutturato come monolite. Questo sistema, pur funzionante, presenta quelle che sono le tipiche problematiche delle architetture monolitiche: difficoltà di manutenzione, rigidità

nell'introduzione di nuove funzionalità e limitazioni nella scalabilità orizzontale. In particolare, l'impossibilità di scalare selettivamente le funzionalità più sollecitate comporta sovradimensionamento dell'infrastruttura e sprechi di risorse.

Per rispondere a queste criticità, l'azienda ha deciso di avviare un processo di rinnovamento architetturale e tecnologico. Esso prevede l'estrazione progressiva delle funzionalità principali dal monolite e la loro trasformazione in microservizi indipendenti, sviluppandoli con la combinazione di un linguaggio ed un framework enterprise come Java e Spring Boot.

La motivazione principale per lo sviluppo di zMailer risiede nella necessità di superare i limiti dell'approccio monolitico attraverso:

- Centralizzazione della logica di gestione email in un unico servizio separato;
- Miglioramento della manutenibilità attraverso una base di codice isolata e ben definita;
- Scalabilità granulare per gestire volumi crescenti di traffico;
- Ottimizzazione delle performance;
- Aumento dell'affidabilità attraverso meccanismi di retry, accodamento e gestione centralizzata degli errori;
- Facilitazione dell'integrazione con sistemi esterni tramite API RESTful ben documentate.

Infine, questo approccio consente anche di evolvere tecnologicamente il servizio email senza impattare il sistema principale, dimostrando un caso pratico di strangler pattern applicato a sistemi legacy enterprise.

1.2 Obiettivo della Tesi

La tesi ha, quindi, un duplice obiettivo: da un lato approfondire gli aspetti teorici delle architetture a microservizio e analizzare quella che è stata l'evoluzione delle architetture software, dall'altro documentare e descrivere l'intero processo di progettazione e implementazione di un caso di studio reale in contesto enterprise, spiegando nel dettaglio il perché delle scelte tecnologiche e strutturali.

1.3 Metodologia di Lavoro

Lo sviluppo del microservizio zMailer è stato condotto utilizzando la metodologia Agile Scrum, ampiamente adottata in contesti di sviluppo software moderno per

la sua capacità di favorire iterazioni rapide, feedback continuo e adattabilità ai cambiamenti.

Il team di progetto è stato guidato da uno Scrum Master (in questo caso il tutor aziendale), il quale ha il ruolo di gestire il team affinché questo non incontri ostacoli e sia il più produttivo possibile.

La metodologia prevedeva cicli di lavoro iterativi chiamati "sprint", della durata di una settimana ciascuno. Ogni giornata lavorativa iniziava con una breve riunione (*daily stand-up meeting*) di circa 15 minuti, durante la quale ogni membro del team condivideva le attività completate il giorno precedente e le attività pianificate per la giornata corrente [2].

Inoltre, all'inizio di ogni settimana si svolgeva la riunione di pianificazione dello sprint (*sprint planning*) durante la quale:

- Venivano analizzate e poi assegnate le user stories del backlog ai vari membri del team;
- Si effettuava la stima della complessità delle task;
- Si includevano eventuali task non completate nello sprint precedente, indagandone le cause.

Al termine di ogni sprint, il team si riuniva poi per una riunione (*sprint retrospective*) con l'obiettivo di:

- Identificare gli aspetti positivi dello sprint;
- Riconoscere eventuali criticità ed errori commessi;
- Definire azioni concrete di miglioramento per lo sprint successivo.

1.4 Struttura della Tesi

La tesi è organizzata in otto capitoli, che percorrono i fondamenti teorici dei temi trattati fino ad arrivare all'implementazione pratica del microservizio e alla valutazione dei risultati ottenuti.

Il presente capitolo ha voluto illustrare il contesto del progetto, le motivazioni che hanno portato allo sviluppo di zMailer, gli obiettivi della tesi e la metodologia di lavoro adottata.

Il secondo capitolo, invece, intende esplorare a fondo i principi fondamentali delle architetture a microservizi, i principali design patterns, le modalità di comunicazioni tra servizi e la loro sicurezza.

Per quanto riguarda il terzo capitolo, esso affronta le sfide legate all'integrazione di nuovi microservizi con sistemi legacy. Vengono analizzate le strategie di

migrazione graduale e le tecniche per garantire la sincronizzazione dei dati e la consistenza eventuale in sistemi distribuiti.

È con il quarto capitolo che si arriva al cuore applicativo della tesi. Qui, infatti, viene descritto il progetto zMailer, dall'analisi dei requisiti alla definizione dell'architettura del sistema, fino all'implementazione dei componenti principali. Vengono illustrate le scelte tecnologiche, le scelte implementative e le funzionalità principali.

Il quinto capitolo illustra l'intero ciclo di vita del microservizio in produzione, dalla containerizzazione alle strategie di deployment e gestione dell'infrastruttura. Vengono descritte le metodologie di testing adottate e le configurazioni per monitoring e observability tramite Sentry.

L'analisi critica delle principali sfide incontrate durante lo sviluppo del progetto è rappresentata nel sesto capitolo. Qui sono illustrati i risultati ottenuti attraverso metriche di performance e affidabilità, mettendo a confronto la situazione precedente con l'implementazione attuale.

Infine, il settimo capitolo, quello conclusivo, offre una sintesi del lavoro svolto ed una valutazione del raggiungimento degli obiettivi prefissati. Vengono discusse anche le possibili evoluzioni future del progetto e, gli eventuali miglioramenti identificati durante il percorso.

Capitolo 2

I Microservizi

Un'architettura a microservizi rappresenta un paradigma architetturale in cui un'applicazione viene strutturata come collezione di servizi autonomi. Ciascuno di essi progettato per eseguire una specifica funzionalità di business in modo indipendente dagli altri componenti del sistema. Ogni microservizio comunica attraverso interfacce ben definite, tipicamente API leggere, ed è caratterizzato da dimensioni ridotte, deployment indipendente, decentralizzazione delle decisioni tecnologiche e organizzative, e possibilità di essere sviluppato in linguaggi di programmazione differenti [3]. Questo paradigma architetturale è nato come risposta alle limitazioni delle architetture monolitiche tradizionali, le quali, essendo costruite come unità singole, risultavano difficili da scalare e inadatte allo sviluppo di processi complessi.

2.1 Teoria delle Architetture a Microservizi

Nel corso degli ultimi decenni l'architettura software ha attraversato diverse fasi evolutive, sintomo dell'aumento della complessità dei sistemi e l'evoluzione delle esigenze di business. Le prime applicazioni enterprise erano tipicamente sviluppate seguendo un approccio monolitico, dove l'intera applicazione costituiva un'unica unità eseguibile. In questo modello, tutti i componenti dell'applicazione erano strettamente accoppiati all'interno di un singolo processo eseguibile, condividendo memoria e risorse computazionali.

Questo approccio, sebbene inizialmente efficace per applicazioni di dimensioni contenute, ha mostrato limiti crescenti con l'espansione della complessità dei sistemi software moderni.

2.1.1 Limitazioni dell'Architettura Monolitica

L'architettura monolitica, pur avendo dominato lo sviluppo software per molti anni, presenta caratteristiche che ne hanno progressivamente evidenziato i limiti:

- Accoppiamento stretto: tutti i componenti dell'applicazione condividono lo stesso spazio di memoria e le stesse risorse, rendendo difficile modificare una parte del sistema senza impattare le altre;
- Scalabilità verticale: l'unica opzione di scalabilità consiste nell'aumentare le risorse hardware della macchina che ospita l'applicazione;
- Deploy monolitico: qualsiasi modifica, anche minima, richiede il rilascio dell'intera applicazione, aumentando i rischi e i tempi di deployment;
- Tecnologia uniforme: l'intero sistema deve essere sviluppato con lo stesso stack tecnologico, limitando la flessibilità nella scelta degli strumenti più adatti per specifiche funzionalità;
- Difficoltà di manutenzione: con la crescita del sistema, la complessità aumenta esponenzialmente, rendendo il codice sempre più difficile da comprendere e modificare.

Queste limitazioni diventano particolarmente evidenti quando le organizzazioni crescono e le applicazioni devono gestire volumi di traffico sempre maggiori, richiedendo flessibilità nel rilascio di nuove funzionalità e capacità di scalare selettivamente i componenti sotto carico.

Un caso emblematico è rappresentato da Netflix, che inizialmente utilizzava un'architettura monolitica per il suo servizio di streaming video. Tuttavia, con la crescita esponenziale della base utenti, passata da pochi milioni nel 2007 a oltre 200 milioni nel 2020, il sistema monolitico ha dimostrato evidenti difficoltà, sia nel gestire l'aumento del traffico, sia nei tempi di deployment che si erano allungati fino a diverse settimane. Inoltre, la complessità del sistema rendeva ogni modifica rischiosa e costosa. Questa esperienza ha quindi spinto Netflix a intraprendere una migrazione completa verso i microservizi, diventando uno dei casi di studio più citati nell'industria [4].

2.1.2 Nascita dei Microservizi

L'architettura a microservizi non è emersa improvvisamente, ma rappresenta l'evoluzione di concetti architetturali sviluppati nel corso di decenni. Le sue radici affondano in paradigmi precedenti, come nei principi di modularità e separazione delle responsabilità teorizzati già negli anni '70 e nella Service-Oriented Architecture (SOA), introdotta negli anni 2000 [5].

Il termine "microservizi" è stato formalmente coniato e diffuso intorno al 2011-2012, ma ha acquisito notorietà soltanto due anni dopo con la pubblicazione dell'articolo "Microservices" di James Lewis e Martin Fowler. L'articolo ha codificato i principi fondamentali e le caratteristiche distintive di questo stile architetturale, fornendo un riferimento teorico che ne ha guidato l'adozione da parte dell'industria.

Infatti, l'approccio a microservizi propone una filosofia radicalmente diversa rispetto al monolite: invece di costruire un'applicazione singola, si scompone in un insieme di "piccoli servizi" (da cui il termine "micro") che collaborano per fornire le funzionalità complessive dell'applicazione.

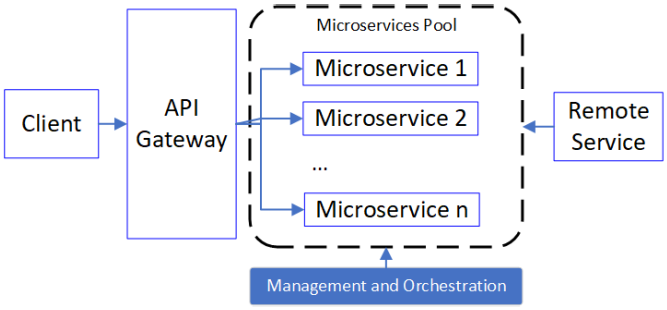

Figura 2.1: Architettura a microservizi.

2.1.3 Principi Fondamentali dei Microservizi

La progettazione e l'implementazione di microservizi efficaci richiedono l'aderenza a un insieme di principi architetturali consolidati che ne garantiscano scalabilità, manutenibilità e resilienza. Sebbene ogni sistema possa richiedere adattamenti specifici, è fondamentale che la costruzione dei microservizi segua standard comuni per promuovere il riutilizzo di componenti, facilitare il testing e garantire la coerenza architetturale dell'intero sistema.

In questa sezione verranno elencati e spiegati i principi più importanti emersi negli anni e proposti dagli esperti Martin Fowler, Sam Newman e Parnas. In particolare Fowler ha esposto nove principi che indicano le caratteristiche che i microservizi dovrebbero avere quando vengono costruiti, mentre Newman, nel suo libro "Building Microservices", evidenzia sette principi includendo linee guida per creare buoni microservizi sia al livello organizzativo che tecnologico. Infine, considerando che i microservizi sono un modo di decomporre complesse architetture, nel 1972 Parnas ha esposto ulteriori due importanti principi riguardo la divisione di sistemi in moduli [5]. Di seguito si elencano i principali:

Componentization via services

Formulato da Fowler [6], questo principio può essere collegato sia a quello di Newman [7] "Hide Internal Implementation Details", sia a quello di Parnas, "Allow modules to be reassembled and replaced without reassembly the whole system". Tale principio afferma che i microservizi debbano essere progettati come componenti autonomi ed eseguibili indipendentemente ed esposti come "black box" attraverso interfacce ben definite. Così l'implementazione interna, il linguaggio di programmazione e lo stack tecnologico utilizzato rimangono nascosti ai consumatori del servizio, che interagiscono esclusivamente tramite API pubbliche.

Questo approccio non solo consente di sostituire o aggiornare un microservizio senza impattare il resto del sistema, purché l'interfaccia pubblica rimanga compatibile, ma permette anche a team diversi di scegliere le tecnologie più adatte per il proprio dominio specifico, ottimizzando performance e produttività [5].

Organized around business capabilities

Formulato sempre da Fowler [6], tale principio può essere collegato a quello di Newman [7], "Allow one module to be written with little knowledge". Questo principio afferma che la divisione modulare dei microservizi debba essere basata su funzionalità di business e non su livelli tecnici quali presentazione, logica applicativa e accesso ai dati. A differenza delle architetture stratificate tradizionali, dove i team sono organizzati per competenza tecnologica (frontend, backend, database), nei microservizi ogni team è responsabile di un'intera business capability end-to-end. In questo modo sono ridotte le dipendenze cross-team ed è facilitata l'evoluzione indipendente di ciascun dominio [5].

Decentralized governance

Come il precedente, fu formulato da Fowler [6] ed è connesso al principio di Newman [7], "Decentralized All the Things". Questo principio afferma che i microservizi debbano godere di autonomia decisionale, così come i team che li sviluppano, i quali sono responsabili della scelta degli strumenti, tecnologie e stack più appropriati per il proprio servizio specifico. A differenza delle architetture monolitiche, dove decisioni tecnologiche centralizzate impongono framework e linguaggi uniformi a tutta l'organizzazione, nei microservizi ogni team può selezionare la soluzione ottimale per il proprio dominio. Ad esempio, un servizio di analisi dati potrebbe utilizzare Python e database NoSQL per le sue caratteristiche di elaborazione, mentre un servizio transazionale potrebbe preferire Java e database relazionali per garantire consistenza ACID. Questa flessibilità promuove l'innovazione e permette di adottare rapidamente nuove tecnologie senza vincolare l'intero sistema [5].

Infrastructure automation

Il principio fu formulato da Fowler [6] ed è collegato al principio di Newman [7], "Adopt a Culture of Automation". Esso afferma la necessità di automatizzare l'intero ciclo di vita del software, dal build al deployment, attraverso pratiche DevOps e Continuous Integration/Continuous Delivery (CI/CD). Infatti dal momento che un'architettura a microservizi può comprendere decine o centinaia di servizi indipendenti, la gestione manuale del deployment può diventare rapidamente insostenibile. L'automazione allora elimina errori umani, garantisce riproducibilità degli ambienti e accelera il rilascio di nuove versioni [5].

Design for failure

Formulato da Fowler [6], il principio è correlato al principio "Isolate Failure" proposto da Newman [7]. Esso afferma che i microservizi debbano essere progettati assumendo che i fallimenti siano inevitabili piuttosto che eccezionali. In un sistema distribuito, i componenti possono diventare temporaneamente non disponibili per guasti hardware, problemi di rete, overload o bug software. L'architettura deve quindi garantire che il fallimento di un singolo microservizio non provochi un effetto cascata che comprometta l'intero sistema [5].

Products not projects

Formulato da Fowler [6], il principio afferma la necessità di organizzare team multifunzionali cross-functional, i cosiddetti full-stack developers, responsabili dell'intero ciclo di vita del servizio: sviluppo, testing, deployment, monitoring e manutenzione operativa. Questo approccio contrasta il modello tradizionale "a progetto", nel quale un team sviluppa il software fino a considerarlo completato, per poi trasferirlo a un team operativo separato. Il modello "a prodotto", invece, promuove la ownership completa, sintetizzata dal celebre motto di Werner Vogels, CTO di Amazon, "you build it, you run it", ovvero chi costruisce il servizio ne è responsabile anche in produzione. Questa responsabilità end-to-end incentiva la scrittura di un codice robusto, osservabile e manutenibile, poiché gli sviluppatori devono gestire direttamente le conseguenze di bug o problemi di performance in produzione [5].

Smart endpoints and dumb pipes

Il principio fu formulato da Fowler [6] e afferma che i meccanismi di comunicazione tra microservizi debbano essere mantenuti semplici, spostando la logica applicativa negli endpoints piuttosto che nei layer di comunicazione. A differenza delle architetture Enterprise Service Bus (ESB), dove la logica di trasformazione, routing e

orchestrazione risiede in un middleware centralizzato complesso, nei microservizi si preferiscono protocolli "dumb", privi di logica applicativa, come HTTP/REST, che si limitano al trasporto dei messaggi. Gli endpoints sono invece "smart", e contengono tutta l'intelligenza necessaria per elaborare richieste, validare dati e gestire la business logic. Questo approccio riduce il coupling tra servizi, elimina bottleneck centralizzati e semplifica il debugging, poiché la logica è contenuta nei servizi stessi piuttosto che distribuita in layer infrastrutturali opachi. La comunicazione diventa allora un dettaglio implementativo sostituibile, permettendo ai team di evolvere i propri servizi indipendentemente, purché mantengano compatibilità nelle interfacce pubbliche [5].

Decentralized data management

Formulato da Fowler [6], il principio è connesso a quello proposto da Newman [7], "Hide Internal Implementation Details" ed a quello di information hiding di Parnas [8]. Esso afferma che ogni microservizio debba possedere e gestire autonomamente il proprio database, eliminando la dipendenza da un database condiviso centralizzato tipico delle architetture monolitiche. Tale separazione garantisce che ciascun servizio possa scegliere la tecnologia di persistenza più appropriata per le proprie esigenze: database relazionali per dati transazionali che richiedono consistenza ACID; database NoSQL per dati non strutturati o che necessitano alta scalabilità; database a grafo per relazioni complesse. L'isolamento dei dati riduce il coupling tra servizi e permette evoluzioni indipendenti dello schema senza coordinazione centralizzata. Tuttavia, questa decentralizzazione introduce sfide significative nella gestione della consistenza dei dati distribuiti, richiedendo pattern come Saga per transazioni distribuite, CQRS per separare letture e scritture, ed Event Sourcing per mantenere la tracciabilità delle modifiche. Dunque, il principio privilegia l'autonomia e la scalabilità rispetto alla consistenza immediata, adottando il modello di consistenza eventuale [5].

Evolutionary design

Il principio, formulato da Fowler [6], afferma che l'architettura a microservizi debba facilitare l'evoluzione continua del sistema in risposta a requisiti mutevoli, tecnologie emergenti e lezioni apprese dall'operatività. In un settore tecnologico caratterizzato da rapidi cambiamenti, la capacità di sostituire componenti obsoleti o non performanti senza impattare l'intero sistema rappresenta un vantaggio competitivo fondamentale. Grazie all'isolamento e all'incapsulamento dei microservizi, è possibile riscrivere completamente un servizio con nuove tecnologie, algoritmi o architetture interne, purché l'interfaccia pubblica rimanga compatibile o venga gestita attraverso versioning [5].

Independently deployable

Il principio, ancora una volta formulato da Fowler [6], afferma che ogni microservizio debba essere deployabile indipendentemente dagli altri, senza richiedere coordinazione o deployment simultaneo di componenti correlati.

Questa indipendenza rappresenta uno dei vantaggi più significativi rispetto alle architetture monolitiche, dove qualsiasi modifica, per quanto piccola, richiede il rebuild e il redeploy dell'intera applicazione con conseguenti finestre di manutenzione e rischi di introdurre regressioni.

Nei microservizi, dunque, un team può rilasciare più volte al giorno nuove versioni del proprio servizio in produzione senza impattare altri team o servizi [5].

Highly observable

Questo principio fu formulato da Newman [7] e afferma che i sistemi a microservizi debbano essere progettati fin dall'inizio per garantire visibilità completa sul loro comportamento, performance e stato di salute in produzione.

A differenza delle architetture monolitiche, dove il debugging può essere effettuato localmente, nei sistemi distribuiti la complessità delle interazioni tra decine o centinaia di servizi rende l'osservabilità un requisito architetturale critico piuttosto che una caratteristica accessoria.

Ogni microservizio deve emettere log strutturati, metriche quantitative e trace distribuiti, che permettano di ricostruire il flusso delle richieste attraverso l'intero sistema. Strumenti come Prometheus per la raccolta di metriche, Grafana per la visualizzazione, ELK Stack (Elasticsearch, Logstash, Kibana) o Loki per l'aggregazione dei log, e sistemi di distributed tracing come Jaeger o Zipkin sono essenziali per diagnosticare problemi, identificare bottleneck e comprendere il comportamento del sistema sotto carico [5].

2.2 Design Patterns per Microservizi

I principi fondamentali descritti nella sezione precedente forniscono le linee guida concettuali per la progettazione di microservizi.

Tuttavia, la loro applicazione pratica richiede l'adozione di design patterns specifici, soluzioni ricorrenti a problemi comuni, che traducono i principi teorici in strutture architetturali concrete e riutilizzabili. I design patterns per microservizi affrontano sfide specifiche di sistemi distribuiti: dalla decomposizione del dominio alla scoperta dei servizi, dalla gestione dei dati alla migrazione graduale da architetture legacy.

Nelle sezioni seguenti verranno analizzati i design patterns più rilevanti per l'architettura a microservizi, con particolare attenzione a quelli applicati nel progetto zMailer, che verrà descritto nei capitoli successivi.

Domain-Driven Design

Domain-Driven Design (DDD) rappresenta un insieme di strumenti che supportano la progettazione e l'implementazione di software di alta qualità. Questo pattern viene utilizzato per sviluppare una visione del sistema da realizzare, generando un modello di dominio di business da cui estrarre le funzionalità di business e costruire un linguaggio comune tra sviluppatori ed esperti di dominio. La creazione di diagrammi porta diversi vantaggi: l'identificazione di sottodomini, l'individuazione di funzionalità strettamente correlate tra di loro e l'identificazione di funzionalità prioritarie per l'azienda, le dipendenze tra funzionalità, e le chiamate a sistemi esterni o servizi di terze parti. Gli elementi appena elencati devono, dunque, essere tutti inclusi nel modello di dominio.

Scomponendo un sistema in tanti piccoli microservizi che eseguono una singola funzionalità è possibile ottenere microservizi con alta coesione [5].

Service Discovery Pattern

Quando i microservizi vengono progettati e creati, generalmente viene utilizzata un'API che offre i servizi necessari affinché l'applicazione possa interagire con il backend. I microservizi hanno bisogno di conoscere dove i servizi sono hostati per potervi comunicare.

Con il Service Discovery Pattern, l'idea è che i servizi stessi, una volta avviati, si registrino in un'entità chiamata Service Registry. Quest'ultima mantiene il controllo di tutti i servizi attivi, in modo che quando si vuole consumare un servizio, si cercano le istanze disponibili nel Service Registry. Ogni servizio, all'avvio, indica il proprio nome e l'indirizzo di dove si trova, in modo tale che il Service Registry mantenga un registro con tutti i servizi disponibili. Oltre alla registrazione iniziale, i servizi devono mandare periodicamente un segnale affinché il Service Registry sappia che sono ancora disponibili. Se un servizio non si manifesta, il Service Registry sa che c'è qualcosa che non va e assume che non sia più disponibile. A questi punto, tutte le richieste verranno reindirizzate verso altre istanze [5].

Tale pattern permette di eseguire un servizio specifico senza conoscere l'indirizzo fisico del servizio. Esistono due approcci principali per il service discovery:

- Client-side discovery: il client interroga direttamente il Service Registry e sceglie un'istanza disponibile utilizzando un algoritmo di load balancing;
- Server-side discovery: il client effettua la richiesta a un load balancer che interroga il Service Registry e inoltra la richiesta a un'istanza disponibile.

Questo approccio semplifica la logica client ma introduce un potenziale single point of failure.

Database per Service Pattern

Uno dei principi fondamentali dell'architettura a microservizi è la decentralizzazione della gestione dei dati. A differenza delle architetture monolitiche, dove tutti i componenti condividono un unico database centrale, nei microservizi ogni servizio dovrebbe possedere e gestire autonomamente i propri dati. Ogni microservizio ha la proprietà esclusiva del proprio database schema (o database fisico separato) e nessun altro servizio può accedere direttamente a questi dati. Dunque, l'accesso ai dati di un microservizio può avvenire esclusivamente attraverso le sue API pubbliche. Ciò determina una serie di vantaggi:

- Indipendenza tecnologica: ogni servizio può scegliere il database più adatto alle proprie esigenze;
- Scalabilità indipendente: è possibile scalare il database di un singolo servizio senza impattare gli altri;
- Isolamento dei fallimenti: problemi di performance o corruzione dati in un database non si propagano agli altri servizi;
- Deploy indipendente: modifiche allo schema possono avvenire senza coordinazione con altri team.

Questo approccio introduce complessità nella gestione della consistenza dei dati tra servizi (eventual consistency), nelle query che necessitano di aggregare dati da servizi multipli e nella gestione di transazioni distribuite [5].

Backend for Frontend Pattern

Il pattern Backend for Frontend (BFF) è un pattern architetturale software che migliora il modo in cui i dati vengono ottenuti tra i client e i server hostati on-premises o in Cloud. L'obiettivo di questo pattern, infatti, è disaccoppiare le applicazioni frontend dall'architettura backend. Il suo principale beneficio è permettere di isolare il backend dal frontend, mentre un vantaggio di questo pattern è il riutilizzo del codice, poiché tutti i client hanno la capacità di ottenere dati dallo stesso server [5].

Adapter Microservices Pattern

Durante la transizione da sistemi legacy verso architetture a microservizi, spesso è necessario integrare API esistenti, basate su tecnologie obsolete, con la nuova

architettura RESTful o basata su eventi. L'Adapter Pattern risolve questo problema incapsulando i sistemi legacy dietro interfacce moderne che aderiscono ai principi dei microservizi. Un microservizio adapter traduce le chiamate REST moderne in invocazioni SOAP o RPC verso i sistemi esistenti, trasformando i dati tra formati diversi (ad esempio, da JSON a XML) e mappando concetti tra modelli di dominio differenti.

Questo approccio protegge i nuovi microservizi dalla complessità e dalle peculiarità dei sistemi legacy, permettendo di modernizzare gradualmente l'architettura senza richiedere la riscrittura immediata di tutti i sistemi esistenti. L'adapter deve essere progettato come componente sottile e stateless, evitando di accumulare logica di business, che lo trasformerebbe in un nuovo monolite. Dunque, la sua responsabilità è esclusivamente la traduzione tra interfacce, mentre la logica di business è delegata ai servizi appropriati [5].

Strangler Application Pattern

Il pattern strangler aiuta a gestire il refactoring di un'applicazione monolitica. L'idea è che si utilizzi la struttura di un'applicazione web in modo da dividere l'applicazione in diversi domini funzionali, i quali vengono poi sostituiti uno a uno con una nuova implementazione basata su microservizi. Nel tempo, la nuova applicazione ristrutturata sostituirà completamente l'applicazione originale, fino a spegnere l'applicazione monolitica.

Dunque, il pattern strangler include questi passaggi:

- Trasformare: consiste nel creare un nuovo sito parallelo e trasformarlo in modo incrementale;
- Coesistere: dove si hanno due siti, quello originale e quello migrato, per un certo periodo. Gradualmente si reindirizzerà dal sito esistente al nuovo sito per le funzionalità implementate;
- Rimuovere: dove la vecchia funzionalità del sito esistente viene rimossa ed il traffico viene reindirizzato verso il sito nuovo.

Ci sono due modi per applicare il pattern Strangler: ristrutturare il backend per una progettazione a microservizi o ristrutturare il frontend per accogliere i microservizi e apportare eventuali modifiche funzionali [5].

Shared Data Microservice Design Pattern

Durante i processi di migrazione da architetture monolitiche verso microservizi, può sembrare allettante mantenere temporaneamente un database condiviso tra il sistema legacy e i nuovi microservizi in modo da facilitare la transizione e ridurre

la complessità iniziale. In fase di migrazione iniziale, ad esempio, condividere il database esistente permette di:

- Evitare la duplicazione immediata dei dati;
- Ridurre la complessità della sincronizzazione;
- Accelerare il time-to-market dei primi microservizi.

Tuttavia, mantenere un database condiviso oltre la fase di transizione iniziale rappresenta un anti-pattern, che mina i benefici fondamentali dei microservizi:

- Accoppiamento stretto: i servizi diventano accoppiati tramite lo schema del database condiviso;
- Deploy non indipendente: modifiche allo schema richiedono coordinazione tra tutti i servizi che lo utilizzano;
- Scalabilità limitata: il database condiviso diventa un collo di bottiglia;
- Failure domain condiviso: problemi di performance o disponibilità del database impattano tutti i servizi simultaneamente.

La strategia corretta, quindi, è quella di utilizzare il Shared Data Pattern solo come soluzione temporanea durante la migrazione [5].

Aggregator Microservice Design Pattern

Un'entità è un oggetto che si distingue principalmente per la sua identità. Le entità, infatti, sono gli oggetti, che nel processo di modellazione, hanno identificatori univoci. Gli oggetti entità necessitano di cicli di vita ben definiti e di una buona definizione di quale sia la relazione di identità radice. Un buon modello entità-relazione ha, dunque, entità ben definite, e ogni entità ha un identificatore specifico ben noto, dal quale però potrebbe non essere mai indipendente.

Una combinazione di entità è un aggregato. Nei casi in cui si ha un gruppo di entità, che devono essere mantenute coerenti, si può fare riferimento a quelle entità come entità dipendenti. Tuttavia, bisogna assicurarsi di sapere quale sia la radice dell'aggregato, poiché la radice definisce il ciclo di vita delle entità dipendenti.

Per i team di sviluppo, i pattern di entità e aggregato sono utili per identificare concetti di business specifici mappati direttamente nei microservizi, eseguendo funzioni di business end-to-end. I microservizi di business tendono a essere stateful ed a possedere i propri dati in un database gestito da loro stessi [5].

2.3 Comunicazione tra Microservizi

In un'applicazione monolitica in esecuzione in un singolo processo, i vari componenti si richiamano l'uno l'altro, usando il metodo a livello di linguaggio o le chiamate di funzione. Questi possono essere strettamente associati se si creano degli oggetti nel codice, o possono essere richiamati in modo disaccoppiato, se si usa l'iniezione delle dipendenze facendo riferimento alle astrazioni anziché a istanze di oggetti concreti. In entrambi i casi, gli oggetti vengono eseguiti all'interno dello stesso processo.

La sfida principale, quando si passa da un'applicazione monolitica ad un'applicazione basata su microservizi, consiste nel modificare il meccanismo di comunicazione. Questa sfida porta con sé note difficoltà che sono conosciute come le fallacies of distributed computing. Quest'ultime elencano i presupposti che gli sviluppatori spesso fanno, errando, quando si passa da un'architettura monolitica a progettazioni distribuite. A tutto ciò non c'è un'unica soluzione, ma diverse. Un'applicazione basata su microservizi è un sistema distribuito in esecuzione su più processi o servizi, in genere anche più server o host. Pertanto, i servizi devono interagire usando un qualche tipo di protocollo di comunicazione tra processi [9]. I principali protocolli comunemente usati sono:

- Protocollo sincrono. In questo tipo di comunicazione il client invia una richiesta e attende una risposta del servizio.
  Indipendentemente dall'esecuzione del codice client, che potrebbe essere sincrona o asincrona, il punto importante è che il protocollo è sincrono e il codice client può continuare l'attività solo quando riceve la risposta del server;

- Protocollo asincrono. In questo tipo di comunicazione il client o il mittente del messaggio in genere non attende una risposta. Esso, infatti, invia semplicemente il messaggio come quando si invia un messaggio a una coda RabbitMQ o a qualsiasi altro broker di messaggi.

2.3.1 Comunicazione Sincrona

Nella comunicazione sincrona tra microservizi, i protocolli più comunemente utilizzati sono Representational State Transfer (REST) e GraphQL. REST è stato uno dei metodi più utilizzati per lo scambio dei dati, ma, nonostante continui ad essere una scelta diffusa, soffre di alcune criticità: ad esempio il recupero di dati non ottimale, che può risultare eccessivo (over-fetching) o insufficiente (under-fetching), per cui i dati recuperati possono superare o essere inferiori alle effettive esigenze.

Per ovviare a questi inconvenienti, GraphQL si presenta come un'alternativa interessante. Infatti, GraphQL consente ai clienti di specificare i dati di cui hanno bisogno, superando il problema dell'inefficienza di REST e offrendo agli sviluppatori di applicazioni un maggiore controllo. Tuttavia oltre a REST e GraphQL, esiste

un altro metodo per lo scambio di dati che sta ottenendo sempre più attenzioni ed è il protocollo Remote Procedure Call (gRPC). Esso offre un approccio efficiente e versatile alla comunicazione tra servizi distribuiti. Infatti a differenza dei metodi REST e GraphQL, che utilizzano il protocollo HTTP/1, gRPC impiega il protocollo HTTP/2 e supporta lo streaming dei dati. Inoltre, esso semplifica le chiamate di procedura remota (RPC) tra diversi linguaggi di programmazione, garantendo prestazioni e velocità superiori nella comunicazione tra microservizi.

Tuttavia, prima di entrare nel dettaglio di questi tre tipi di comunicazione, è utile definire cosa sia un'API. Per Application Programming Interface si intende l'insieme di regole e convenzioni che facilitano la comunicazione e l'interazione tra diversi programmi software. Questi protocolli definiscono la struttura ed il formato delle richieste e delle risposte ed i metodi per la comunicazione [10].

Representational State Transfer (REST)

REST è un'architettura per lo sviluppo di API che fornisce una comunicazione basata sul modello client-server attraverso il protocollo HTTP. REST è stato introdotto per la prima volta da Roy Fielding nella sua tesi di dottorato presso l'Università della California nel 2000. Esso utilizza il protocollo HTTP/1.1 per inviare dati dai client ai server.

Nei sistemi che adottano REST, ogni servizio espone solitamente un endpoint specifico per consentire l'interazione e lo scambio di dati tra servizi. REST mette a disposizione diversi metodi, tra cui GET, POST, PUT e DELETE e, inoltre, supporta vari formati per la rappresentazione dei dati, come JSON e XML, privilegiando JSON per la sua semplicità ed efficienza [10].

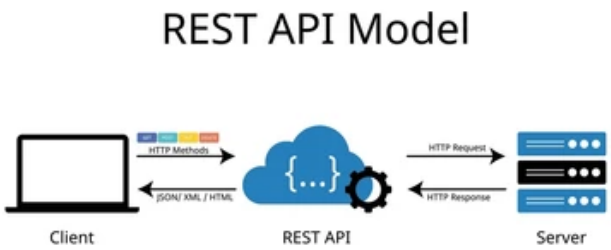

Figura 2.2: Modello di comunicazione REST.

GraphQL

GraphQL, un linguaggio di query per API, è stato creato da Facebook ed è utilizzato per la comunicazione tra client e server. Il client richiede i dati necessari tramite una query, consentendo al server di restituire una risposta conforme alla richiesta.

GraphQL rappresenta un'alternativa a REST e permette agli sviluppatori di ottenere dati specifici in modo più efficiente e flessibile. Infatti, la sua nascita è legata all'esigenza di Facebook di gestire dati complessi e superare le problematiche di REST, come l'over-fetching e l'under-fetching.

Uno dei principali vantaggi di GraphQL è la sua flessibilità: i client possono richiedere più fonti di dati con una singola richiesta, riducendo così il numero di chiamate necessarie per ottenere le informazioni desiderate.

Inoltre, i client possono validare le query grazie a tipi definiti prima di inviarle al server [10].

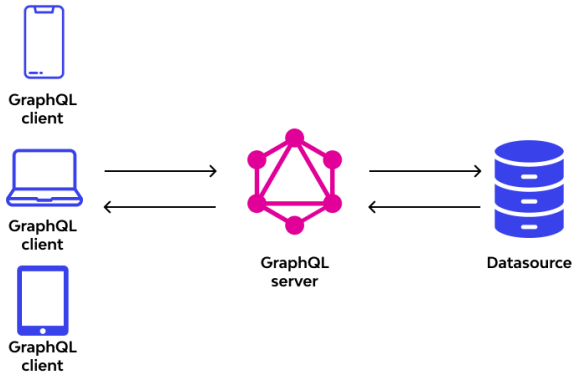

Figura 2.3: Modello di comunicazione GraphQL.

Dunque, a differenza di REST, dove ogni endpoint restituisce una struttura dati fissa, GraphQL permette al client di specificare esattamente quali campi necessita, risolvendo il problema di over-fetching e under-fetching. Inoltre, GraphQL elimina la necessità di versionare le API, dando la possibilità di aggiungere nuovi campi senza breaking changes e di rimuovere campi deprecati gradualmente. Tuttavia, ci sono degli svantaggi come la maggiore complessità server-side e la difficoltà nell'implementare caching efficace rispetto a REST [10].

gRPC

Google Remote Procedure Call (gRPC) è un framework open source ad alte prestazioni per la realizzazione di sistemi distribuiti e microservizi efficienti. Sviluppato da Google, è progettato per consentire la comunicazione tra applicazioni e servizi

in modo indipendente dal linguaggio e dalla piattaforma. gRPC permette di definire metodi di servizio e strutture dati tramite Protocol Buffers (protobuf), un linguaggio neutrale per la definizione delle interfacce.

Sulla base di queste definizioni, esso genera automaticamente il codice client e server in diversi linguaggi di programmazione. La comunicazione avviene tramite HTTP/2, sfruttando funzionalità come lo streaming bidirezionale, il multiplexing e la serializzazione efficiente. Si tratta, quindi, di un framework ad alte prestazioni per la costruzione di sistemi distribuiti, microservizi e API indipendenti dal linguaggio [10].

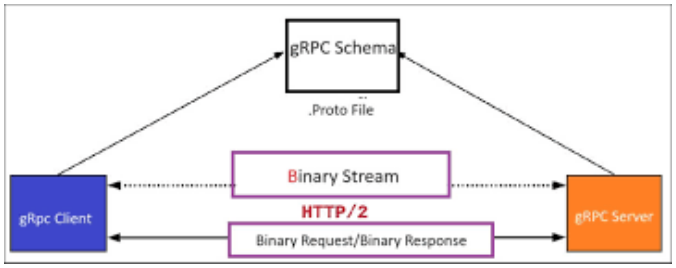

Figura 2.4: Modello di comunicazione gRPC.

gRPC offre performance significativamente superiori grazie a Protocol Buffers e HTTP/2. Diversi benchmark effettuati da altri studi [10] dimostrano che gRPC può essere 5-10 volte più veloce di REST per payloads simili.

Tuttavia, gRPC è meno human-readable e richiede code generation, risultando meno immediato per debugging rispetto a REST. Per questo motivo REST rimane preferibile per API pubbliche destinate a client esterni, mentre gRPC è la scelta ideale per comunicazioni inter-service ad alte performance [10].

2.3.2 Comunicazione Asincrona

A differenza dell'approccio sincrono, nella comunicazione asincrona il client invia la sua richiesta al server senza avere la certezza di quando e se otterrà una risposta. Infatti, la connessione viene chiusa subito dopo che la richiesta viene trasmessa.

Dato che non è possibile aspettare per una risposta immediata, la comunicazione asincrona offre l'utilizzo delle code. Tutte le richieste vengono salvate e recuperate in un secondo momento dal server. Il più grande vantaggio offerto dalla comunicazione asincrona può essere riassunto con il termine di separazione. Infatti, i microservizi, che non hanno bisogno di conoscersi a vicenda, in questo modo sono meno accoppiati tra di loro, il che permette anche di scalare orizzontalmente in maniera molto più semplice rispetto alla comunicazione sincrona.

Message Broker

I message broker sono componenti infrastrutturali che rendono possibile la comunicazione asincrona. Mentre una coda di messaggi memorizza i messaggi fino a quando non vengono processati dall'applicazione ricevente, i message broker, operando come un vero e proprio servizio separato, vanno oltre, offrendo funzionalità aggiuntive.

Una funzione chiave è il message routing: il broker può decidere dove inviare ciascun messaggio in base a regole specifiche.
Ad esempio, esso può trasmettere un singolo messaggio a più destinatari oppure indirizzarlo a un servizio specifico.

Un'altra importante caratteristica dei message broker è la trasformazione dei messaggi: il broker può modificare il formato dei messaggi per adattarlo alle esigenze dell'applicazione ricevente. Questa capacità è particolarmente utile nell'integrazione di sistemi che utilizzano formati diversi, poiché garantisce che tutte le parti del sistema possano comprendere ed elaborare le informazioni ricevute.

Inoltre, i message broker migliorano anche l'affidabilità: possono memorizzare i messaggi in modo sicuro affinché non vadano persi in caso di crash del sistema o guasto della rete. Non solo, essi possono confermare la ricezione dei messaggi e, se necessario, rinviarli, rendendo la comunicazione più robusta e affidabile.

Tra i broker più diffusi rientrano Apache Kafka, RabbitMQ e Apache Pulsar, ciascuno caratterizzato da modelli di comunicazione, prestazioni e casi d'uso differenti.

Ad esempio, RabbitMQ è un message broker tradizionale basato sul protocollo AMQP ed è particolarmente indicato per scenari di messaggistica affidabile e routing complesso. Esso offre meccanismi avanzati di gestione delle code, acknowledgements e delivery garantita, risultando adatto a sistemi che richiedono un controllo fine sul flusso dei messaggi.

Apache Kafka, invece, è progettato come una piattaforma di streaming distribuito ad alte prestazioni.
Si basa su un modello publish/subscribe orientato ai log e garantisce elevata scalabilità e throughput, rendendolo ideale per la gestione di grandi volumi di eventi e per casi d'uso come event streaming, log aggregation e data pipeline.

Per quanto riguarda Apache Pulsar, esso rappresenta una soluzione più recente che combina caratteristiche di messaging tradizionale e streaming di eventi. Pulsar separa il livello di storage da quello di serving, migliorando la scalabilità e la flessibilità del sistema. Inoltre, supporta nativamente più modelli di consumo, come code, publish/subscribe e stream processing, oltre a funzionalità avanzate quali multi-tenancy, geo-replicazione e garanzie di consegna configurabili. Queste caratteristiche lo rendono particolarmente adatto a sistemi distribuiti complessi e a scenari cloud-native.

Event-driven architecture

Event-driven architecture (EDA) è uno stile architetturale nello sviluppo del software che coinvolge la generazione, la scoperta ed il processamento di eventi. Questo approccio consente un'architettura reattiva e a basso accoppiamento per la costruzione di sistemi progettati per la comunicazione asincrona, rendendola adatta a sistemi distribuiti su larga scala come i microservizi. Il concetto chiave è che le dipendenze tra i vari componenti devono essere minime: chi produce eventi non deve sapere nulla riguardo a chi consumerà quegli eventi.

I microservizi basati su un'architettura asincrona event-driven implicano che i servizi possano lavorare in parallelo tra di loro e comunicare usando solamente gli eventi. Una delle prime caratteristiche, che si integra perfettamente con la comunicazione asincrona, delle architetture event-driven è che i servizi che mandano un messaggio non si aspettano una risposta prima di passare alla successiva attività. Questo tipo di comunicazione, come anche spiegato precedentemente, non solo non blocca l'applicazione, ma migliora la sua performance e la sua capacità di offrire numerosi servizi contemporaneamente. Infatti, l'indipendenza di questi sistemi ne permette la costruzione e la manutenibilità in maniera molto più semplice, dato che non è necessario utilizzare altri servizi per far eseguire quello di interesse.

Tuttavia, questo stile di comunicazione presenta alcuni problemi, soprattutto nella gestione del progresso in differenti servizi. Infatti, poiché molti servizi vengono eseguiti contemporaneamente, spesso è difficile identificare lo stato attuale di un servizio, soprattutto se non è presente un meccanismo di gestione dello stato. Inoltre, rendere coerente lo stato del sistema tra tutti i servizi rappresenta un ulteriore difficoltà. Ad esempio, durante l'invio di un evento, poiché il servizio ricevente non fornisce una risposta diretta, è difficile determinare se l'evento sia stato correttamente elaborato. Per questo motivo EDA utilizza la consistenza eventuale, che è un fenomeno importante nei sistemi distribuiti, come i microservizi basati su eventi. I sistemi event-driven sono diversi dai sistemi convenzionali che richiedono la coerenza di tutti i servizi in un determinato momento. Si presume che, in un sistema basato su eventi, tutti i servizi diventino coerenti col tempo. Questo è particolarmente vero quando i dati sono duplicati in diversi servizi e il sistema deve garantire che, alla fine, tutte le modifiche siano riflesse in ciascuno di essi [11].

Publish-Subscribe pattern

Il pattern publish-subscribe (Pub-Sub) è uno dei modelli architetturali più utilizzati nei sistemi basati su eventi. In questo modello, i publisher emettono eventi senza conoscere i destinatari, mentre diversi subscriber si registrano autonomamente per ricevere solo gli eventi di loro interesse. Il Pub-Sub offre il vantaggio di disaccoppiamento temporale e spaziale: infatti, i produttori e i consumatori non

devono essere attivi contemporaneamente, né conoscere reciprocamente la loro esistenza o posizione. Questo consente ai servizi di evolvere indipendentemente, facilitare l'aggiunta di nuovi consumer senza modificare i publisher e migliorare significativamente la scalabilità del sistema. Rispetto a un'architettura monolitica, dove l'aggiunta di nuove funzionalità richiederebbe modifiche al codice esistente e ridistribuzione dell'intera applicazione, il Pub-Sub permette di estendere il sistema semplicemente aggiungendo nuovi subscriber. Dunque, il pattern è particolarmente adatto in situazioni in cui molteplici componenti devono reagire allo stesso evento in modi diversi, come notifiche, logging, analytics e aggiornamenti di cache [11].

2.3.3 Confronto e Criteri di Scelta

La scelta del protocollo di comunicazione in un'architettura a microservizi rappresenta una decisione architetturale critica che influenza direttamente le prestazioni complessive del sistema, la sua scalabilità e la complessità di sviluppo e manutenzione. A differenza delle architetture monolitiche, dove la comunicazione tra componenti avviene prevalentemente attraverso chiamate di funzione in-process con overhead trascurabile, i microservizi richiedono meccanismi di comunicazione inter-processo che introducono latenza di rete, serializzazione dei dati e necessità di gestione dei fallimenti distribuiti.

Nel contesto della comunicazione sincrona, REST mantiene il vantaggio della semplicità e dell'adozione universale, risultando particolarmente adatto per API pubbliche dove l'interoperabilità e la compatibilità con client eterogenei sono prioritarie. Tuttavia, REST presenta limitazioni intrinseche legate all'inefficienza nel recupero dei dati come i già menzionati problemi dell'over-fetching, dove il client riceve più informazioni del necessario aumentando il carico di rete, e dell'under-fetching, che richiede multiple richieste sequenziali per ottenere dati correlati.

GraphQL emerge come soluzione a queste criticità, consentendo ai client di specificare esattamente i dati necessari attraverso un linguaggio di query dichiarativo. Questa flessibilità riduce significativamente il numero di round-trip necessari, risultando particolarmente vantaggiosa in scenari con strutture dati complesse o client con requisiti eterogenei. Tuttavia, tale flessibilità comporta un costo computazionale non indifferente e alcune ricerche empiriche hanno dimostrato che GraphQL presenta un utilizzo della CPU superiore rispetto a REST, a causa della necessità di interpretare dinamicamente ogni query e costruire risposte personalizzate.

gRPC rappresenta un'evoluzione significativa per la comunicazione inter-servizio ad alte prestazioni. L'adozione di HTTP/2 e Protocol Buffers conferisce vantaggi prestazionali sostanziali. Il multiplexing, infatti, permette richieste concorrenti su una singola connessione TCP riducendo l'overhead, mentre la serializzazione binaria risulta più efficiente rispetto a JSON in termini di dimensione payload e velocità di

parsing. Inoltre, studi comparativi hanno evidenziato che gRPC presenta tempi di risposta mediamente inferiori rispetto a REST e GraphQL, con un vantaggio particolarmente marcato in scenari a carico medio-basso. Tuttavia, sotto carichi elevati, le prestazioni possono degradare a causa delle limitazioni dei buffer di rete e del maggiore consumo di memoria. Anche la minore leggibilità del formato binario e la necessità di code generation rendono gRPC meno immediato per il debugging rispetto agli approcci basati su JSON, fatto che ne limita l'adozione per API rivolte a sviluppatori esterni.

La comunicazione asincrona basata su message broker e architetture event-driven offre un paradigma complementare, particolarmente vantaggioso in scenari che richiedono elevato disaccoppiamento temporale e spaziale. L'eliminazione dell'attesa sincrona delle risposte consente ai servizi di operare indipendentemente, migliorando la resilienza complessiva del sistema e facilitando la scalabilità orizzontale. L'adozione del pattern Publish-Subscribe permette quindi l'evoluzione del sistema senza modifiche invasive: infatti, nuovi subscriber possono essere aggiunti per reagire a eventi esistenti senza alterare i publisher. Esso rappresenta un vantaggio significativo rispetto alle architetture monolitiche, in cui l'introduzione di nuove funzionalità richiede tipicamente modifiche al codice esistente e ridistribuzione dell'intera applicazione. Tuttavia, la comunicazione asincrona introduce complessità nella gestione della consistenza dei dati, richiedendo l'adozione del modello di consistenza eventuale, in cui lo stato del sistema diventa coerente nel tempo e non istantaneamente.

La scelta del message broker appropriato dipende dalle caratteristiche specifiche del sistema. RabbitMQ eccelle in scenari che richiedono routing complesso e garanzie di consegna affidabili, risultando particolarmente adatto per architetture che necessitano di controllo granulare sul flusso dei messaggi. Apache Kafka, progettato come piattaforma di streaming distribuito, garantisce invece throughput elevato e scalabilità orizzontale, rendendolo ideale per gestione di grandi volumi di eventi, log aggregation e data pipeline. Apache Pulsar, infine, rappresenta una sintesi evolutiva, che combina messaggistica tradizionale e streaming di eventi con funzionalità avanzate, quali multi-tenancy e geo-replicazione, particolarmente adatte a sistemi distribuiti complessi in ambienti cloud-native.

Nella pratica, architetture enterprise mature adottano frequentemente un approccio ibrido che combina comunicazione sincrona e asincrona secondo le caratteristiche specifiche di ciascuna interazione. Per operazioni che richiedono risposta immediata, validazione in tempo reale o transazioni con consistenza forte, la comunicazione sincrona risulta preferibile. Si consiglia allora gRPC per comunicazioni inter-servizio ad alte prestazioni, REST per API pubbliche o quando la semplicità è prioritaria e GraphQL quando i client hanno requisiti di dati eterogenei e la flessibilità di query giustifica l'overhead computazionale. Per processi batch, notifiche, audit

logging, analytics e scenari dove il disaccoppiamento temporale costituisce un vantaggio architetturale, la comunicazione asincrona rappresenta la soluzione ottimale, permettendo ai servizi di evolversi indipendentemente.

In conclusione, non esiste un protocollo universalmente superiore, ma piuttosto scelte architetturali contestuali basate su requisiti funzionali e non funzionali specifici. La comprensione approfondita dei trade-off di ciascun approccio costituisce un requisito fondamentale per progettare architetture a microservizi efficaci, bilanciando prestazioni, complessità, scalabilità e manutenibilità nel lungo periodo.

2.4 Sicurezza nei Microservizi

La sicurezza dei microservizi presenta numerose sfide legate alla fiducia e alla protezione.

Il tema principale sarà sempre legato alle abitudini di programmazione e agli errori umani, che aumentano poiché i microservizi trasformano normalmente una semplice applicazione in una vasta superficie di attacco sotto la responsabilità di molte persone. Alla luce di questo, amministratori di sistema, amministratori di database, fornitori di soluzioni cloud e middleware API gateway distribuiti sulla rete devono tutti comunicare e agire come un'orchestra, in cui ciascuno suona però il proprio strumento tramite un servizio di streaming senza errori. È importante tenere presente un aspetto fondamentale: i microservizi sono spesso progettati per fidarsi dei propri pari e compromettere efficacemente uno di essi può portare a una divulgazione completa e vantaggiosa degli altri. Gli hacker sanno come funzionano i sistemi e conoscono le abitudini degli utenti, quindi garantirne la sicurezza è della massima importanza.

Per questo motivo, dato che le applicazioni sono distribuite su diversi servizi o piattaforme, molte delle quali nel cloud pubblico come PaaS, SaaS, IaaS, la loro protezione rappresenta una vera e propria sfida a causa della complessità degli sviluppi, della difficoltà di monitorare ed effettuare il debug e l'audit dell'intera applicazione in ambienti esterni. Tale sfida, inoltre, si fa ancora più difficile per via di un livello di trasparenza comune a questi tipi di servizi, che deve essere nascosto in quanto parte del business dei fornitori.

In un ambiente cloud, utilizzare soluzioni di terze parti completamente fuori dalla portata in termini di sicurezza, negando agli sviluppatori il controllo o l'accesso alla soluzione acquistata e lasciandoli con il prodotto senza i segreti che lo compongono, significa chiedere loro di fare un salto nel buio dal punto di vista della sicurezza. Essi possono usare questi servizi per ospitare i loro microservizi, come database o API gateway, a loro rischio, assicurandosi però che ciò sia chiaramente indicato nel loro EULA (End User License Agreement). Gli hacker, allora, sfruttano la mancanza di astrazione per attaccare applicazioni che utilizzano questo tipo di

architettura distribuita su più soluzioni, mentre i fornitori cloud continuano a non disporre di meccanismi per controllare o supportare i clienti nel monitoraggio del loro sistema distribuito con tutti i componenti che si fidano l'uno dell'altro.

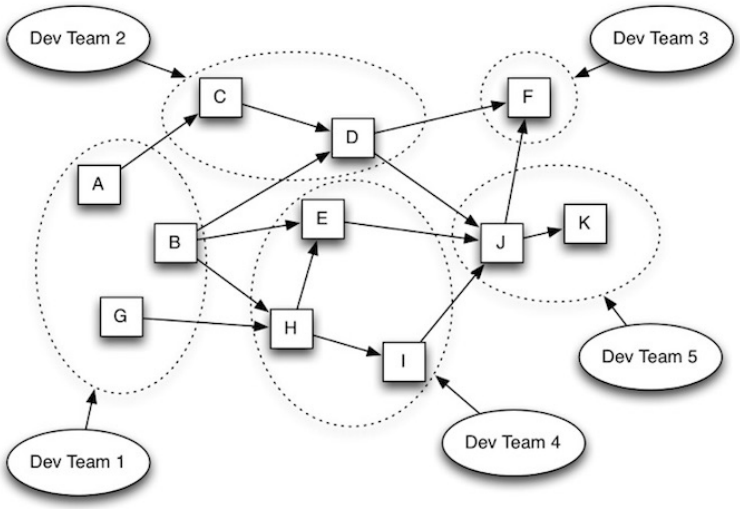

Figura 2.5: Struttura a microservizi.

Se un attaccante compremette un singolo servizio sfruttando una vulnerabilità in un microservizio esposto pubblicamente con la quale riesce a fare privilege escalation sulla VM in cui il microservizio è in esecuzione, può verificarsi una vera e propria catastrofe. Di conseguenza, i singoli microservizi potrebbero non essere affidabili.

Per questa serie di motivazioni è importante mettere in discussione il nostro giudizio quando scegliamo un partner per l'uso dei servizi cloud. Le domande riguardanti il modo in cui mantengono la sicurezza e come lo fanno sono fondamentali, indipendentemente dalle dimensioni o dalla qualità del fornitore. Infatti, le dimensioni non contano: tutti sono un bersaglio quando si parla di sicurezza. Se il sottodominio di una determinata applicazione viene compromesso e da quel dominio un attaccante serve qualsiasi contenuto nel contesto del dominio della webapp, se tutti i cookie degli utenti posso essere accessibili da qualsiasi sottodominio, allora un attaccante che controlla un sottodominio può manomettere l'autenticazione e, in ultima analisi, i dati [12]. Dunque, lo sviluppo sicuro è qualcosa che richiede molto impegno e dedizione da parte del team sviluppo.

Nelle prossime sezioni verranno elencati i punti più importanti che bisogna tenere in mente durante lo sviluppo di un microservizio per mantenere un livello di sicurezza alto.

2.4.1 Possibili Rischi

Complessità delle password

Innanzitutto, è fondamentale promuovere servizi e meccanismi che obblighino utenti, amministratori e sviluppatori ad utilizzare password complesse ed uniche. Infatti, bisogna imporre l'uso di caratteri maiuscoli dalla A alla Z con segni diacritici, caratteri greci e cirillici, caratteri non alfanumerici o qualsiasi unicode classificato come carattere alfabetico ma non maiuscolo o minuscolo [12].

Autenticazione

A volte l'autenticazione non viene considerata una priorità durante lo sviluppo ed il risultato può essere grave. Oggi le applicazioni si interconnettono tramite API per facilitare l'interazione dell'utente e per fare tutto in maniera trasparente. C'è una tendenza a trascurare tali questioni a costo elevato, dato che molte applicazioni posseggono meccanismi per elaborare acquisti.

E' importante bloccare gli account anche dopo pochi tentativi di accesso falliti e verificare che il software di terze parti, integrato tramite API, faccia lo stesso. Per servizi critici, ad esempio sanitari, bancari e militari, l'autenticazione deve essere assegnata in modo sicuro e le credenziali mantenute segrete, invece di essere definite dall'utente. Dunque, è fondamentale implementare protezioni contro gli attacchi di enumerazione degli account durante il reset delle password e utilizzare servizi di autenticazione multifattore in microservizi separati [12].

Vulnerabilità della sicurezza web

I microservizi non sono altro che un'evoluzione delle piattaforme e dei servizi informatici standard, quindi rimane la stessa vulnerabilità anche se ciò che prima era un singolo punto di vulnerabilità ora è distribuito.

Ciò che bisogna ricordare è che la sicurezza deve essere riconosciuta e discussa da sviluppatori e project manager e, che l'argomento di discussione deve essere la OWASP TOP 10 in scenari distribuiti [12].

Attacchi

L'effetto umano circonda tutto influenzando e infettando i sistemi in modi positivi e negativi. Per questo motivo deve essere necessario garantire che i team di sviluppo integrino la sicurezza sia ricorrendo a framework e standard specifici, sia promuovendo un'adeguata formazione. Questa'ultima deve includere attività esplicite sui principi di sicurezza e sullo sviluppo sicuro del software, in modo che progettisti, sviluppatori e tester possano comprendere i principi di base come il controllo degli accessi. La chiave è l'uso del ciclo PDCA (Plan, Do, Check, Act)

[12], in particolare nel contesto dei microservizi in cui la superficie di attacco è distribuita su decine o centinaia di servizi. L'obiettivo è una solida politica interna di sicurezza operativa e semantica dei microservizi composta da:

- Pianificazione della continuità operativa;
- Controllo degli accessi al sistema;
- Sviluppo e manutenzione del sistema;
- Sicurezza fisica e ambientale;
- Conformità e sicurezza personale;
- Sicurezza organizzativa, gestione sicura di sistemi e rete;
- Classificazione degli asset e controllo dei servizi.

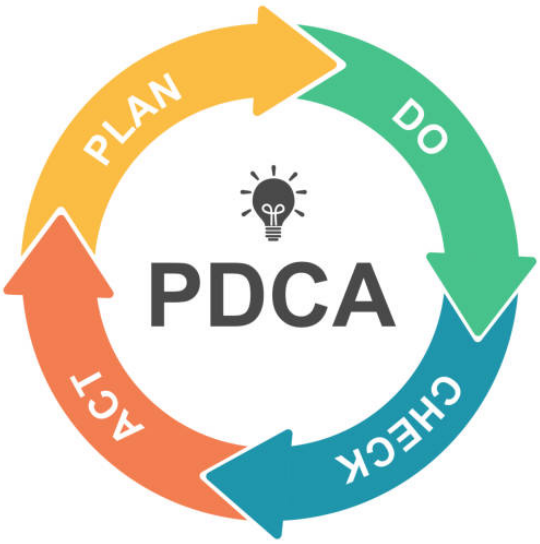

Figura 2.6: PDCA.

Per ragioni contenutistiche non sono inseriti né tutti i vari tipi di attacchi esistenti né il modo in cui essi colpiscono direttamente il cuore delle applicazioni basate su microservizi. Tuttavia, di seguito viene riportata una lista delle le 10 vulnerabilità più critiche per le applicazioni web secondo OWASP. Ecco i candidati per l'anno 2025:

- A1 Broken Access Control;
- A2 Security Misconfiguration;
- A3 Software Supply Chain Failures;
- A4 Cryptographic Failures;
- A5 Injection;
- A6 Insecure Design;
- A7 Authentication Failures;
- A8 Software or Data Integrity Failures;
- A9 Security Logging and Alerting Failures;
- A10 Mishandling of Exceptional Conditions.

Nel contesto dei microservizi, alcune di queste vulnerabilità assumono particolare rilevanza. Ad esempio, il Broken Access Control (A1) diventa critico quando ogni servizio deve validare autonomamente i permessi, mentre le Security Misconfiguration (A2) si moltiplicano proporzionalmente al numero di servizi deployati. O ancora le Software Supply Chain Failures (A3) sono amplificate dall'uso intensivo di container e dipendenze esterne in ogni microservizio.

Un aspetto che merita un'analisi più approfondita è l'SSRF (Server-Side Request Forgery). Quando si affronta il tema della sicurezza bisogna farlo in senso ampio, nonostante il rischio sia apparentemente basso. I microservizi distribuiscono una determinata fonte di dati di un'applicazione attraverso servizi in una rete di richieste app-server, riducendo la necessità di utilizzare pesantemente un database o un servizio ospitato su un server. Comunemente, o questi database o servizi sono suddivisi in più piccole richieste di dati oppure l'applicazione utilizza diversi database in cui memorizza e recupera informazioni, mantenendo ridondanza e servizi rapidi.

Nonostante le applicazioni basate su microservizi distribuiscano le informazioni in diversi piccoli frammenti di dati, ciò non significa che l'SSRF non sia un rischio. Infatti, le applicazioni possono generare richieste inter-server per recuperare risorse, sfruttando il legame di fiducia tra server e servizi.

Nelle architetture a microservizi, le richieste inter-server sono generalmente sicure se vengono applicate difese come firewall o segregazione di rete. Quando questi aspetti non vengono rispettati, il risultato può essere un attacco SSRF, in cui un aggressore sfrutta il minimo vantaggio e abusa della relazione di fiducia tra server. In questo modo, egli aggira la IP whitelist ed i servizi di autenticazione basati su host, legge risorse non accessibili come metadati, API in un ambiente cloud, esegue

una scansione della rete a cui il server è connesso e recupera informazioni sensibili come l'indirizzo IP di un web server dietro un reverse proxy [12].

2.4.2 Migliori Contromisure

Tutto ciò che è stato appena descritto può essere mitigato attraverso diverse soluzioni specifiche per architetture distribuite. A differenza dei monoliti, dove la sicurezza può essere implementata in un unico perimetro, nei microservizi è necessario: applicare il principio del minimo privilegio a ogni singolo servizio; stabilire confini di fiducia chiari; ridurre la superficie di attacco; controllare e limitare il più possibile l'accesso; implementare sistemi di difesa su più livelli e responsabilizzare progettisti e gli sviluppatori nel controllo dei livelli di fiducia, evitando eccessi.
I dati devono essere protetti non solo a livello logico, ma anche durante il transito da un punto A a un punto B [12].

Autenticazione e Autorizzazione

Il processo di autenticazione conferma che un determinato soggetto sia effettivamente chi dichiara di essere, mentre l'autorizzazione consiste nel meccanismo che definisce e consente le azioni che tale soggetto può compiere all'interno di ciascun sistema.

Nelle applicazioni monolitiche questo è meno complesso, poiché l'applicazione stessa gestisce autenticazione e autorizzazione. Framework e linguaggi come Spring, Php, Angular e Python constentono di implementare la gestione degli utenti in modo semplice e sicuro. Nei microservizi, invece, è necessario adottare schemi più avanzati. Infatti, non sarebbe pratico se ogni utente dovesse effettuare login separati per sistemi diversi, utilizzando credenziali differenti, aumentando la complessità e richiedendo un broker per gestire il processo. In questi casi, le astrazioni possono essere strumenti potenti. L'obiettivo è promuovere ed implementare sistemi con identità unica, in grado di garantire un'autenticazione affidabile. E' importante ricordare che, per definizione, i microservizi si fidano reciprocamente. Tra le soluzioni più efficaci per una forte autenticazione e autorizzazione vi è l'utilizzo di Single Sign-On (SSO) gateways, che evitano l'impiego di librerie condivise e riducono la duplicazione del codice.
I gateway fungono da proxy, gestendo il processo di handshake con i provider di identità, centralizzando il comportamento e semplificando la gestione [12].

Autenticazione ed Autorizzazione tra Servizi

Nei microservizi, i servizi comunicano tra loro in modo implicito e sono esposti ad attacchi "man-in-the-middle". E' raccomandato utilizzare HTTPS anziché

l'autenticazione HTTP di base, non solo per cifrare le credenziali, ma anche per garantire che il client comunichi con il server corretto, proteggendo così il traffico da intercettazioni e manipolazioni. Questa soluzione è sicura ma complessa. Infatti, la gestione dei certificati SSL in scenari multi-macchina è difficile, i processi di emissione sono rischiosi e complessi e alcuni certificati sono difficili da revocare.

Una soluzione appropriata è l'implementazione del Single Sign-On con protocolli come SAML o OpenID, ideali sia per l'autenticazione degli utenti sia per quella tra servizi.

In alternativa, i certificati client su TLS possono confermare l'identità e garantire autenticazioni robuste, sebbene richiedano una gestione accurata. Altre soluzioni includono HMAC su HTTP, che utilizza codici di autenticazione basati su hash per garantire l'integrità e l'autenticità delle richieste. Il client firma ogni richiesta combinandola con una chiave privata condivisa e applicando una funzione hash (es. SHA-256). Il server, possedendo la stessa chiave privata, ricalcola l'hash dalla richiesta ricevuta e lo confronta con quello inviato dal client: se coincidono la richiesta è autentica e non è stata manomessa durante il transito.

Le API sono centrali nei microservizi, poiché favoriscono la comunicazione e l'integrazione con servizi esterni. L'uso di API Keys è comune, ma deve essere gestito con attenzione. E' essenziale archiviare queste informazioni in ambienti sicuri, utilizzando servizi dedicati alla cifratura e alladecifratura dei dati. Ad esempio, server SQL offrono supporto nativo per la crittografia, in modo da tenere al sicuro le informazioni [12].

Protezione dei dati a riposo

Proteggere i dati attualmente non utilizzati è un punto fondamentale. Anche in ambienti sicuri può spesso capitare che gli attaccanti riescano ad accedere ad informazioni sensibili. Per questo motivo è necessario garantire che i dati siano sempre cifrati, evitando algoritmi proprietari o poco conosciuti, poiché mancano di validazione e revisione critica. Algoritmi consolidati come AES-256 sono preferibili, dal momento che possiedono aggiornamenti regolari e test di sicurezza. I dati devono essere cifrati at rest, ovvero quando vengono memorizzati, e decifrati in memory solo durante l'elaborazione, minimizzando il tempo in cui sono in chiaro [12].

Difesa in profondità

I microservizi operano per livelli, quindi progettare un sistema "a cipolla" con diversi strati di sicurezza è essenziale. Tecnologie come DPI (Deep Packet Inspection) e firewall rappresentano l'ultima linea di difesa.
L'architettura deve prevedere firewall sulle principali interfacce di servizio, controllando porte e protocollo e ricorrendo a IDS/IPS integrati con firewall stateful.

Infatti, uno dei vantaggi dei Microservizi è la possibilità di distribuire i servizi su reti o subnet diverse e controllarli con firewall o regole IPTABLES,definendo connessioni, porte attive e indirizzi IP/MAC [12].

Tracciamento

Infine, il tracciamento è un classico della sicurezza. Monitoriare chi fa cosa è essenziale e per farlo si ricorre ai log, che consentono di analizzare gli exploit, l'utilizzo del sistema e le vulnerabilità.

Dunque, nell'ambito della sicurezza informatica, in particolare per i microservizi che aumentano significativamente il livello di complessità dell'intero sistema distribuito, vale il principio fondamentale "nulla è totalmente sicuro" [12]. Tuttavia, l'adozione sistematica delle contromisure analizzate consente di costruire architetture a microservizi significativamente più resilienti agli attacchi. L'obiettivo non è l'eliminazione completa del rischio, che rimane un'illusione, ma l'innalzamento della barriera di complessità per gli attaccanti, in modo da rendere il sistema economicamente non conveniente come bersaglio rispetto ad alternative più vulnerabili. Paradossalmente, sebbene i microservizi introducano una superficie di attacco distribuita su molteplici servizi, l'isolamento e la segregazione, che caratterizzano questo paradigma architetturale, possono, se correttamente implementati, contenere l'impatto di una compromissione localizzata, impedendo la propagazione laterale che nelle architetture monolitiche può compromettere l'intero sistema da un singolo punto di ingresso. La sicurezza nei microservizi richiede, quindi, un cambio di paradigma: da un perimetro difensivo centralizzato a un modello zero-trust, in cui ogni servizio è potenzialmente ostile fino a prova contraria, implementando meccanismi di verifica continua dell'identità e dell'integrità ad ogni interazione [12].

Capitolo 3

Migrazione da Sistemi Legacy a Microservizi

Nei capitoli precedenti sono stati analizzati i principi fondamentali, i pattern architetturali e i vantaggi intrinseci dell'architettura a microservizi rispetto alle soluzioni monolitiche. Tuttavia, la migrazione da un sistema legacy esistente a un'architettura distribuita basata su microservizi rappresenta una delle trasformazioni tecnologiche più complesse che un'organizzazione possa intraprendere.

A differenza dello sviluppo greenfield, dove l'architettura può essere progettata exnovo seguendo le migliori pratiche fin dall'inizio, questi tipi di migrazione richiedono di operare su sistemi in produzione, i quali devono mantenere la continuità operativa durante l'intera transizione [13].

E' proprio questo processo di migrazione che il seguente capitolo intende esaminare, analizzando le motivazioni che spingono le organizzazioni verso i microservizi, le difficoltà tecniche e organizzative comuni e le strategie di migrazione più valide. Particolare attenzione viene anche dedicata alle evidenze quantitative che guidano le decisioni architetturali e permettono di valutare quando la migrazione è effettivamente vantaggiosa e quale approccio adottare per minimizzare i rischi.

3.1 Motivazioni della Migrazione

La decisione di migrare da un'architettura monolitica a microservizi non dovrebbe essere guidata da mode tecnologiche, ma da esigenze concrete e misurabili. Recenti studi, condotti su un campione di diverse organizzazioni che hanno intrapreso la migrazione, hanno rilevato tre ragioni principali che spingono verso questa trasformazione [13].

La motivazione più frequente della migrazione è la scarsa manutenibilità del sistema monolitico. Infatti, i principali problemi che si manifestano tipicamente

quando le applicazioni monolitiche superano determinate soglie dimensionali, sia in termini di linee di codice che di complessità dell'intero sistema sono: la perdita di visione d'insieme della codebase, dove nessun singolo sviluppatore comprende più l'intero sistema, modifiche eccessivamente costose in termini di tempo e risorse e l'alta probabilità di effetti collaterali indesiderati quando si modifica una funzionalità [13].

I problemi di scalabilità rappresentano la seconda motivazione. Nei sistemi monolitici, l'impossibilità di scalare selettivamente i componenti sotto carico costringe a replicare l'intera applicazione, comportando spreco di risorse computazionali e costi operativi sproporzionati. Un modulo che richiede elevata capacità computazionale, come l'elaborazione di immagini o l'analisi dati in tempo reale, forza la replicazione anche di componenti che non necessitano risorse aggiuntive. I microservizi risolvono questo problema permettendo la scalabilità granulare dei soli servizi che effettivamente sono sotto pressione, ottimizzando l'utilizzo delle risorse e riducendo i costi infrastrutturali [13].

L'ultimo motivo è rappresentato dalle problematiche operative. I sistemi monolitici soffrono di tempi di avviamento lunghi che rallentano sia il deployment, sia i test sia i tempi di inattività durante gli aggiornamenti, impattando negativamente gli utenti. Le organizzazioni che operano in contesti competitivi dove il time-to-market è critico trovano questi limiti particolarmente dannosi per la loro volontà di innovare rapidamente [13].

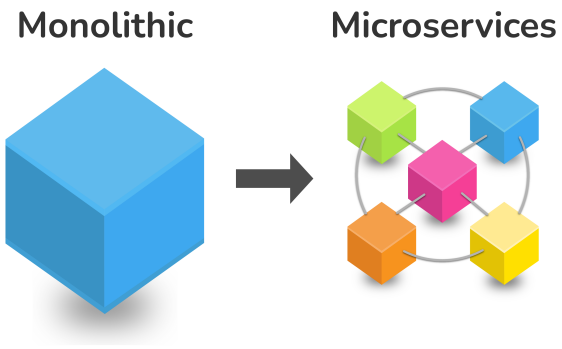

Figura 3.1: Confronto tra architettura monolitica e architettura a microservizi.

3.2 Benefici della Migrazione

3.2.1 Miglioramenti Operativi

Le organizzazioni che hanno completato con successo la migrazione riportano miglioramenti quantificabili in diverse dimensioni operative. L'aspetto che è migliorato maggiormente è la scalabilità dell'applicazione: infatti, l'85% delle aziende ha riportato un miglioramento netto, con valutazione di 4.5 su 5 rispetto alla situazione precedente alla migrazione [14]. La capacità di scalare selettivamente i microservizi sotto carico ha permesso di ottimizzare l'utilizzo delle risorse computazionali, riducendo i costi operativi e migliorando le performance complessive del sistema.

Un altro significativo beneficio, riportato in questo studio, è la velocità di rilascio delle nuove funzionalità, ovvero il tempo di deployment è migliorato notevolmente per l'80% delle organizzazioni coinvolte, con una valutazione media di 4.3 su 5 [14]. La possibilità di deployare servizi indipendentemente ha accelerato i cicli di rilascio, permettendo alle organizzazioni di rispondere più rapidamente alle esigenze del mercato e dei clienti.

Infine un ultimo aspetto da evidenziare è il miglioramento della produttività del team di sviluppo, con il 65% delle organizzazioni che ha segnalato un miglioramento significativo, con una valutazione media di 3.9 su 5 [14]. La decomposizione del sistema in microservizi ha permesso ai team di lavorare in modo più autonomo, riducendo le dipendenze e i conflitti che spesso rallentano lo sviluppo in ambienti monolitici.

3.2.2 Miglioramenti Performativi

Oltre ai benefici operativi, le organizzazioni hanno riscontrato miglioramenti significativi nelle performance dell'applicazione al termine della migrazione. In particolare, il 75% delle organizzazioni ha riportato un miglioramento importante nel tempo di risposta (latenza), con una valutazione media di 4.4 su 5 [14]. La capacità di scalare i microservizi in base alla domanda ha contribuito a ridurre i tempi di risposta, migliorando l'esperienza utente complessiva.

Inoltre, il 65% delle organizzazioni ha segnalato un miglioramento significativo nella throughput dell'applicazione, con una valutazione media di 4.1 su 5 [14]. Dunque, la decomposizione del sistema in microservizi ha permesso di ottimizzare le risorse e migliorare la capacità di gestire un maggior numero di richieste simultanee, aumentando l'efficienza complessiva del sistema.

Infine, sono migliorate anche la tolleranza ai guasti e la resilienza del sistema, con il 60% delle organizzazioni che ha riportato un notevole miglioramento, con una valutazione media di 4.0 su 5 [14]. La natura distribuita dei microservizi ha

permesso di isolare i guasti e garantire la continuità operativa anche in presenza di problemi, aumentando l'affidabilità complessiva dell'applicazione.

3.3 Sfide e Complessità della Migrazione

Nonostante i benefici documentati, la migrazione da monolite a microservizi introduce complessità significative che devono essere attentamente valutate e gestite.

3.3.1 Complessità Tecniche

Gestione dei Dati Distribuiti

Tipicamente, nei sistemi monolitici tutti i componenti condividono lo stesso database, mentre nell'architettura a microservizi ogni servizio possiede idealmente il proprio database per garantire il disaccoppiamento. Questa transizione richiede una ristrutturazione profonda dell'architettura dati [13]. Le principali difficoltà includono la gestione delle transazioni distribuite. Le transazioni ACID (Atomicity, Consistency, Isolation, Durability), che nel monolite garantivano consistenza immediata attraverso singole operazioni database, devono essere sostituite con pattern come Saga, i quali decompongono transazioni complesse in sequenze di transazioni locali. Questo introduce complessità nella gestione dei fallimenti parziali e nella definizione di operazioni che gestiscano questi fallimenti [14]. La consistenza eventuale sostituisce la consistenza forte tipica dei monoliti: i sistemi a microservizi devono tollerare finestre temporali in cui i dati distribuiti su servizi diversi non sono perfettamente sincronizzati. Questo modello di consistenza, sebbene necessario per mantenere il disaccoppiamento e la scalabilità, richiede riprogettazione delle logiche applicative e può complicare operazioni che prima avvenivano atomicamente. Le query cross-service, che aggregano dati da servizi multipli, rappresentano un'ulteriore complessità. L'utilizzo di pattern come CQRS (Command Query Responsibility Segregation) può aiutare a mitigare questo problema, nonostante introduca overhead di sincronizzazione e duplicazione dei dati.

In questo modo, nelle architetture a microservizio si passa da ACID a BASE (Basically Available, Soft state, Eventually consistent) per la gestione dei dati [14].

Comunicazione tra Servizi

Nei sistemi monolitici i componenti comunicano tipicamente attraverso chiamate di metodo all'interno della stessa codebase, mentre nei microservizi, essendo entità separate in ambienti diversi, è necessario determinare come i servizi interagiranno tra loro. Questa transizione da chiamate in-process a comunicazione di rete introduce latenza, necessità di gestione dei fallimenti, serializzazione/deserializzazione dei dati

e complessità nella scoperta dei servizi [14]. La scelta dei protocolli di comunicazione, analizzati nel capitolo precedente, diventa allora critica: protocolli sincroni, come REST, gRPC o GraphQL per operazioni request-response immediate, o protocolli asincroni basati su message broker per disaccoppiamento temporale. Ciascun approccio presenta vantaggi e svantaggi specifici in termini di latenza, throughput, complessità implementativa e resilienza ai fallimenti.

Osservabilità e Debugging Distribuito

Il debugging di sistemi distribuiti è intrinsecamente più complesso rispetto ai sistemi monoliti. Un'operazione utente può attraversare decine di microservizi, rendendo difficile tracciare il flusso di esecuzione e identificare la causa radice dei problemi. Dunque l'implementazione di distributed tracing, di logging centralizzato aggregato e di monitoring comprensivo diventa requisito architetturale fondamentale piuttosto che caratteristica opzionale [13].

3.3.2 Complessità Organizzative

Recenti studi hanno anche evidenziato come sia necessario un cambio culturale all'interno dell'organizzazione. Secondo gli studi di Smith et al. (2022) e di Li e Zhang (2023) è importante che ci sia uno switch mentale all'interno delle organizzazione, in modo da poter concentrarsi di più sulla collaborazione tra i vari team di sviluppo e dominio [14]. Infatti, l'architettura a microservizi richiede di riorganizzare i team secondo il principio "you build it, you run it" per allinearsi con la migrazione tecnica. Così i team organizzati per competenza tecnologica (frontend, backend, DBA) devono trasformarsi in team cross-funzionali responsabili di interi domini business end-to-end. Questo cambiamento organizzativo può incontrare resistenza, specialmente in organizzazioni con culture gerarchiche consolidate. La transizione richiede investimenti significativi in formazione, ridefinizione di responsabilità e cambiamento nella mentalità organizzativa da project-oriented a product-oriented, discusso nel secondo capitolo [14].

3.4 Strategie di Migrazione

Esistono diverse strategie per affrontare la migrazione da monolite a microservizi. La scelta della strategia più appropriata può dipendere da diversi fattori: lo stato del sistema legacy, i vincoli temporali e di budget, le competenze del team e i requisiti di continuità operativa.

3.4.1 Strangler Fig Pattern

Lo Strangler Fig Pattern rappresenta l'approccio più diffuso e raccomandato per la migrazione. Invece di sostenere una riscrittura completa, questa tecnica si concentra sulla migrazione incrementale e graduale, che mantiene la continuità operativa durante tutta la transizione. Questa strategia fu coniata da Martina Fowler ed il suo nome deriva dalla pianta strangolatrice, che cresce attorno ad un albero esistente fino ad ucciderlo ed a sostituirlo [15]. Secondo uno studio, questa tecnica mostra un tasso di successo dell'80% [14], significativamente superiore a molti altri approcci. Il pattern prevede tre fasi sovrapposte e iterative.

Fase 1 - Trasformazione. Si identifica un dominio funzionale specifico nel monolite candidato per l'estrazione. I criteri di selezione includono chiara delimitazione dei confini del dominio, dipendenze minime con altri moduli, potenziali benefici immediati in termini di scalabilità o performance, complessità gestibile per limitare il rischio. Il dominio selezionato viene reimplementato come microservizio indipendente. Un API Gateway o un router viene configurato per intercettare le richieste destinate a quella funzionalità e reindirizzarle al nuovo servizio, mentre il resto dell'applicazione continua a essere servito dal monolite.

Fase 2 - Coesistenza. Il sistema opera in modalità ibrida dove alcune funzionalità sono servite dal monolite e altre dai nuovi microservizi. Questa fase può durare mesi o anni, permettendo di validare ogni estrazione in produzione prima di procedere con la successiva. Durante la coesistenza, particolare attenzione deve essere dedicata alla gestione dei dati. Se il microservizio e il monolite condividono ancora accesso allo stesso database, si introduce un anti-pattern temporaneo, che deve essere risolto implementando database separati e meccanismi di sincronizzazione.

Fase 3 - Eliminazione. Una volta che tutte le funzionalità di un dominio sono migrate e stabilizzate in produzione, il codice corrispondente viene rimosso dal monolite. Il traffico viene reindirizzato completamente verso i microservizi e le dipendenze con il vecchio sistema legacy vengono eliminate.

3.4.2 Big Bang

Quando compariamo l'approccio Strangler con altri metodi di migrazione, emergono chiaramente tutti i vantaggi che questo approccio offre. Per esempio un altro approccio alla migrazione è il Big Bang, che prevede la riscrittura completa e simultanea dell'applicazione monolitica in microservizi, ma presenta tassi di successo significativamente inferiori, circa del 55% [14]. Inoltre, gli studi indicano, frequenti fallimenti dovuti alla sottostima della complessità del sistema esistente, impossibilità di testare adeguatamente l'intero nuovo sistema prima del deployment e la troppa pressione insostenibile sui team di sviluppo [14]. La strategia incrementale, invece, permette di validare i benefici in produzione prima di investire ulteriormente, mantenendo il sistema operativo durante l'intera migrazione.

Per concludere, la migrazione da sistemi monolitici a microservizi rappresenta una trasformazione complessa che richiede un'approfondita pianificazione e significativi investimenti sia tecnici che organizzativi. Tuttavia, per sistemi che presentano reali problemi di scalabilità, manutenibilità o velocità di evoluzione, i benefici documentati in questo capitolo giustificano gli sforzi. Le evidenze indicano chiaramente che approcci incrementali come lo Strangler Fig Pattern offrono le migliori probabilità di successo. Al contrario, approcci Big Bang o migrazioni intraprese senza adeguata preparazione mostrano tassi di fallimento elevati. Tuttavia, è fondamentale valutare preliminarmente se la propria applicazione e la propria organizzazione possano realmente beneficiare di questo tipo di architettura. Un'analisi onesta e approfondita è, dunque, imprenscindibile.

Nel prossimo capitolo, i principi e le strategie presentate finora verranno applicati al caso studio concreto del progetto zMailer. Verrà, infatti, riportato il processo di estrazione del servizio di gestione email dal monolite esistente, incluse le sfide incontrate e le soluzioni adottate, in modo da fornire un esempio pratico dei concetti teorici appena discussi.

Capitolo 4

Caso di Studio - zMailer

4.1 Contesto Aziendale

Il progetto zMailer nasce dall'esigenza di migliorare il sistema di gestione e di invio email presente all'interno del prodotto software dell'azienda. Come accennato precedentemente, l'azienda dispone di un'architettura software monolitica consolidata che presenta significative limitazioni strutturali.

4.1.1 Il Sistema Esistente

Il sistema attuale integra la funzionalità di invio email direttamente all'interno dell'applicazione principale. Questa soluzione presenta diverse cricità, già discusse nei capitoli precedenti in relazione alle architetture monolitiche: ad esempio, l'accoppiamento stretto con il componente principale, che rende complessa qualsiasi modifica poiché tende ad impattare l'intero sistema. Ne deriva un impedimento dell'evoluzione tecnologica di questa funzionalità, che è così costretta a sincronizzarsi con i cicli di rilascio dell'applicazione principale. Un ulteriore aspetto negativo è rappresentato dall'impossibilità di scalare selettivamente questa specifica funzione in presenza di picchi di traffico. L'unica strategia possibile risulta, quindi, il sovradimensionamento dell'intera infrastruttura, con conseguenti sprechi di risorse e costi operativi elevati. Inoltre, un problema particolarmente rilevante riguarda la scarsa affidabilità del sistema. Infatti, l'assenza di meccanismi di retry e di una gestione efficace degli errori può comportare la perdita di messaggi in caso di malfunzionamenti dei provider esterni. Infine, è assente una gestione centralizzata dei log, il che rende difficile l'analisi delle performance e il monitoraggio dello stato delle email inviate.

4.1.2 Obiettivi del Progetto

Il progetto zMailer si pone come soluzione a queste problematiche appena descritte attraverso l'estrazione della funzionalità di invio email dal monolite e la sua trasformazione in un microservizio dedicato. L'obiettivo primario è quello di creare un unico servizio che possa fungere da unico punto di accesso per tutte le applicazioni dell'azienda che necessitano di inviare una email, in modo da evitare la duplicazione del codice nei diversi sistemi. Un altro obiettivo è sicuramente quello di risolvere il problema della scalabilità, consentendo il dimensionamento indipendente del servizio email in base al carico effettivo e ottimizzando i costi. È importante aumentare anche l'affidabilità per garantire che nessun messaggio venga perso e per fornire visibilità sullo stato di ogni singola email. Questo comprende l'integrazione di un sistema di logging strutturato per monitorare la salute del servizio. Si vedrà in seguito come ciò è stato implementato anche grazie all'utilizzo di Sentry.

Per quanto riguarda i principali obiettivi funzionali, tra i principali figura il support a diversi provider di invio come Amazon SES, SMTP e Microsoft Graph API, al fine di assicurare le stesse funzionalità offerte dal servizio precedente. Ovviamente l'architettura dovrà essere progettata per consentire, in futuro, l'aggiunta di nuovi provider senza alcuna difficoltà. Infine, è necessario garantire la completa separazione dei dati tra le differenti applicazioni e tenant. Infatti, la gestione della multi-tenancy è cruciale affinché ogni realm possa operare in un ambiente isolato con configurazioni e dati completamente indipendenti.

4.1.3 Il Concetto di Realm

Nel contesto di zMailer, il realm rappresenta l'unità fondamentale di isolamento del sistema e costituisce il contesto operativo all'interno del quale ogni richiesta viene eseguita. Un realm è definito dalla combinazione di due entità distinte: un tenant e un'applicazione.

Il tenant rappresenta un'organizzazione o un cliente che utilizza la piattaforma, l'applicazione, invece, identifica uno specifico sistema software appartenente a quel tenant che interagisce con zMailer. Questa distinzione è necessaria poiché uno stesso tenant può disporre di più applicazioni che necessitano di inviare email, ciascuna con configurazioni di provider e mittenti differenti.

Ogni richiesta effettuata verso zMailer è quindi sempre associata a uno specifico realm, che determina quali dati sono accessibili, quale provider di invio utilizzare e quali configurazioni applicare. Questo modello garantisce un isolamento completo tra i diversi contesti operativi, impedendo che dati o configurazioni di un realm possano essere accessibili o influenzare un altro realm.

4.1.4 Requisiti Funzionali

In questa sezione vengono elencati i principali requisiti funzionali individuati dall'analisi delle esigenze aziendali e dalle limitazioni del sistema esistente.

Tabella 4.1: Requisiti funzionali del sistema

Requisito	**Descrizione**
FR1	**Gestione workflow invio email.**
FR1.1	Il sistema deve poter ricevere richieste di invio email.
FR1.2	Il sistema deve validare i dati della richiesta inclusi mittente, destinatari (to, cc, bcc), oggetto e corpo dell'email.
FR1.3	Il sistema deve verificare la dimensione totale degli allegati.
FR1.4	Il sistema deve memorizzare i metadati delle email.
FR1.5	Il sistema deve memorizzare gli allegati.
FR1.6	Il sistema deve accodare le email validate in una coda per l'invio asincrono.
FR1.7	Il sistema deve processare la coda e inviare le email tramite il provider configurato per il realm corrente.
FR1.8	Il sistema deve aggiornare lo stato dell'email dopo ogni tentativo di invio.
FR2	**Configurazione provider email.**
FR2.1	Il sistema deve permettere la configurazione di Amazon SES come provider email per ciascun realm.
FR2.2	Il sistema deve permettere la configurazione di server SMTP come provider email, supportando autenticazione semplice (username/password).
FR2.3	Il sistema deve supportare l'autenticazione OAuth2 per server SMTP compatibili (Microsoft Azure, Google Workspace).
FR2.4	Il sistema deve permettere la configurazione di Microsoft Graph API come provider email.
FR2.5	Il sistema deve selezionare dinamicamente il provider da utilizzare in base alla configurazione del realm richiedente.
FR2.6	Il sistema deve validare le configurazioni del provider al momento del salvataggio.
FR3	**Gestione verifica identità.**
FR3.1	Il sistema deve permettere la richiesta di verifica degli indirizzi email.
FR3.2	Il sistema deve integrare il processo di verifica con Amazon SES inviando email di conferma all'indirizzo da verificare.

FR3.3	Il sistema deve gestire il processo di verifica DKIM per domini, generando i record DNS necessari.
FR3.4	Il sistema deve fornire i record DNS DKIM in formato testuale scaricabile dal client.
FR3.5	Il sistema deve monitorare periodicamente lo stato delle verifiche email e DKIM in modo asincrono.
FR3.6	Il sistema deve aggiornare lo stato delle verifiche nel database.
FR3.7	Il sistema deve notificare il client del completamento delle verifiche.
FR3.8	Il sistema deve impostare un timeout di 48 ore per le verifiche, marcandole come failed se non completate.
FR4	**Autenticazione e autorizzazione.**
FR4.1	Il sistema deve autenticare i client.
FR4.2	Il sistema deve emettere un access token con validità di 10 minuti per i client autenticati.
FR4.3	Il sistema deve validare i JWT token in ingresso per tutte le API protette.
FR4.4	Il sistema deve permettere il logout invalidando il token corrente.
FR5	**Sistema di notifiche.**
FR5.1	Il sistema deve consentire ai client di ricevere notifiche automatiche al verificarsi di determinati eventi.
FR5.2	Il sistema deve inviare notifiche ai client registrati al verificarsi degli eventi sottoscritti.
FR5.3	Il sistema deve eseguire l'invio delle notifiche in modo completamente asincrono rispetto al flusso principale.
FR6	**Sicurezza Dati.**
FR6.1	Il sistema deve cifrare tutti i dati sensibili a riposo.
FR6.2	Il sistema deve utilizzare algoritmi di cifratura conformi agli standard.
FR6.3	Il sistema deve decifrare i dati solo al momento dell'utilizzo effettivo.
FR6.4	Il sistema deve hashare le password client utilizzando algoritmi sicuri.
FR7	**Rate Limit.**
FR7.1	Il sistema deve limitare il numero di invii simultanei verso ciascun provider esterno.
FR7.2	Il sistema deve prevenire il superamento dei limiti imposti da ciascun provider di invio email.

FR7.3	Il sistema deve bloccare temporaneamente l'accettazione di nuove email quando vengono raggiunti i limiti di invio.
FR8	**Gestione allegati.**
FR8.1	Il sistema deve verificare che la dimensione totale degli allegati non superi il limite configurato.
FR8.2	Il sistema deve permettere il recupero degli allegati fino al momento dell'invio con successo dell'email.
FR8.3	Il sistema deve eliminare automaticamente gli allegati dopo l'invio con successo dell'email o dopo un timeout configurabile.
FR9	**Gestione priorità email.**
FR9.1	Il sistema deve permettere di assegnare un livello di priorità alle email.
FR9.2	Il sistema deve ordinare la coda di invio in base alla priorità decrescente delle email.
FR10	**Gestione multi-tenancy.**
FR10.1	Il sistema deve supportare l'isolamento completo dei dati tra realm differenti.
FR10.2	Il sistema deve supportare operazioni globali (cross-realm) solo per servizi autorizzati.
FR11	**Gestione errori.**
FR11.1	Il sistema deve implementare meccanismi di retry automatico per email fallite.
FR11.2	Il sistema deve configurare un numero massimo di tentativi di invio.
FR11.3	Il sistema deve loggare ogni tentativo di invio con timestamp, esito e messaggio di errore.

4.1.5 Requisiti Non Funzionali

In questa sezione vengono elencati i requisiti non funzionali individuati che il sistema dovrà soddisfare.

Tabella 4.2: Requisiti non funzionali del sistema

ID	**Categoria**	**Descrizione**	**Valore**
NFR1	Reliability	Il sistema deve garantire che ogni email accettata e confermata al client non venga persa, anche in caso di errori o crash del servizio.	Durability $\geq$ 99.999%

NFR2	Availability	Il sistema deve essere disponibile e funzionante per la maggior parte del tempo.	Availability $\geq$ 99.9%
NFR3	Scalability	Il sistema deve poter essere scalato orizzontalmente tramite l'esecuzione di più istanze concorrenti senza perdita di funzionalità o consistenza.	Min 2 istanze
NFR4	Scalability	Il sistema deve mantenere le performance sotto carico elevato.	Performance fino a 10.000 email/ora per istanza
NFR5	Efficiency	Le chiamate API sincrone devono rispondere rapidamente	Response time $\leq$ 100ms
NFR6	Efficiency	Il throughput delle API deve supportare traffico intenso	$\geq$ 100 richieste/secondo per istanza
NFR7	Maintainability	Il codice deve avere una copertura di test adeguata	Code coverage $\geq$ 80%
NFR8	Usability	Le API devono essere completamente documentate	

4.2 Tecnologie Utilizzate

La scelta del giusto stack tecnologico è cruciale per la scalabilità, l'efficienza ed il successo di un'azienda. Il processo di scelta è molto complesso e può essere influenzato da diversi fattori che spaziano da fattori economici a fattori strategici [16].

La scelta delle diverse tecnologie per il progetto zMailer è stata guidata, ovviamente, da criteri di robustezza enterprise, performance ed allineamento con le competenze del team di sviluppo. In questa sezione verranno, quindi, descritte nel dettaglio le scelte tecnologiche adottate e le motivazioni che le hanno determinate.

4.2.1 Linguaggio e Framework

Il microservizio è stato sviluppato utilizzando Java 25, l'ultima versione LTS disponibile. La scelta di Java si fonda su diversi fattori, quali la maturità dell'ecosistema enterprise, ottime performance per applicazioni server-side ed una vasta disponibilità di librerie e framework. Java 25 introduce, inoltre, miglioramenti significativi in termini di performance e gestione della memoria rispetto alle versioni precedenti.

Come framework applicativo è stato scelto Spring Boot 4, che rappresenta lo standard per lo sviluppo di microservizi enterprise in ambiente Java. Spring Boot offre numerosi vantaggi, come la configurazione automatica che riduce il boilerplate code, il supporto nativo per pattern architetturali cloud-native ed una community vasta e attiva. La versione 4, uscita proprio a Novembre del 2025, porta ulteriori ottimizzazioni e supporto migliorato per architetture moderne.

Per la gestione delle dipendenze e il processo di build è stato adottato Apache Maven, tool consolidato che garantisce build riproducibili e gestione dichiarativa delle dipendenze. È stato preferito Maven in quanto rappresenta il tool di riferimento in ambito Java all'interno di Zucchetti. In futuro si sta valutando una migrazione a Gradle, in quanto offrre una maggiore flessibilità ed espressività nella definizione del processo di build.

4.2.2 Persistenza e Storage

La strategia di persistenza scelta adotta un approccio multi-storage, dove ciascuna tipologia di dato è gestita dalla tecnologia più appropriata in base alla natura del dato stesso e al modo in cui bisogna accedere ad esso. Come database relazionale principale è stato scelto PostgreSQL 18 che viene utilizzato per la gestione dei dati principali. PostgreSQL è il database open source relazionale a oggetti più avanzato al mondo ed è progettato proprio per prestazioni di livello enterprise ed, a partire proprio dalla versione 18, esso supporta nativamente OAuth 2.0 come metodo di autenticazione. [17]. La scelta è motivata, dunque, dai molteplici vantaggi offerti:

- Affidabilità: PostgreSQL ha un'alta tolleranza di errore, mantiene la durabilità dei dati e massimizza il tempo di attività [17];
- Sicurezza: supporta un modello di autenticazione e autorizzazione avanzato, nonché diversi metodi di crittografia, tra cui la crittografia end-to-end dei dati con SSL [17];
- Alto rendimento: PostgreSQL archivia i dati in modo strutturato, il che consente di inserirli, eliminarli e modificarli in modo efficace. È anche performante nelle operazioni di ricerca e join. Inoltre, PostgreSQL può scalare con più CPU in parallelo, velocizzando ulteriormente le query [17];
- Conformità: i database PostgreSQL sono conformi ad ACID, il che significa che i dati sono validi anche in caso di interruzioni hardware, software o di rete [17];
- Massima estendibilità: PostgreSQL supporta un'ampia gamma di tipi di dati, più linguaggi di programmazione e la possibilità di scrivere funzioni personalizzate [17];

- Row-Level Security (RLS): PostgreSQL permette di definire policy di accesso granulari a livello di singola riga direttamente nel database, migliorando la sicurezza e l'isolamento dei dati in ambienti multi-tenant [18];
- PostgreSQL integra un potente motore di Full-Text Search (FTS) che consente di indicizzare e cercare testo in modo rapido e intelligente [19];
- Supporto per gli UUID: PostgreSQL fornisce supporto nativo per la generazione di UUID utilizzando gli algoritmi UUIDv4 e UUIDv7 [20].
- Semplici da monitorare: PostgreSQL fornisce diverse statistiche per supportare la raccolta e la generazione di report sull'attività del server [17].

Per ottimizzare le performance di indicizzazione sono stati utilizzati UUIDv7 come chiave primaria. A differenza degli UUIDv4, che sono gli unici totalmente casuali, la versione 7 è basata su timestamp, che genera valori monotonicamente crescenti e ordinabili temporalmente. Questa caratteristica rende le query basate sul tempo più efficienti, se opportunamente accompagnate da indici B-tree [21]. Inoltre, gli UUIDv7 garantiscono unicità globale anche in ambienti distribuiti, evitando il rischio di collisioni [21]. A differenza degli UUIDv1, che si basano sull'indirizzo MAC della macchina che genera l'UUID, gli UUIDv7 non hanno questo difetto e, quindi, non espongono nessuna informazione sensibile, migliorando la privacy e la sicurezza [21]. Per specifici tipi di dati, inoltre sono stati configurati indici hash al fine di migliorare ulteriormente le performance di query specifiche.
È inoltre in fase di analisi l'adozione degli UUID74, una variante che restituisce al client UUID con i bit del timestamp rimescolati in base a un seed configurabile, in modo da apparire esternamente come UUIDv4 pur mantenendo internamente l'ordinabilità temporale, evitando così di esporre informazioni sul momento di creazione della risorsa.

Infine, per le tabelle che lo richiedono, sono state abilitate le cosidette Row-Level Security (RLS), le quali permettono di definire policy di accesso ai dati a livello di singola riga [18]. Grazie a questa funzionalità è possibile implementare un controllo degli accessi più sicuro, assicurando che ogni realm possa accedere esclusivamente ai propri dati e garantendo isolamento e sicurezza tra i diversi tenant.

```
1 DROP POLICY IF EXISTS email_senders_access_policy ON email_senders
    ;
2
3 ALTER TABLE email_senders ENABLE ROW LEVEL SECURITY;
4
5 CREATE POLICY email_senders_access_policy ON email_senders
6 USING (
7     app_id = NULLIF(current_setting('app.application_id', true),''
    )::uuid
8 )
```

```
9  WITH CHECK(
10     app_id = NULLIF(current_setting('app.application_id', true),''
       )::uuid
11 );
12
13 ALTER TABLE email_senders FORCE ROW LEVEL SECURITY;
```

Listing 4.1: Esempio di policy RLS.

Per garantire un'evoluzione consistente e versionata dello schema del database nei diversi ambienti, il microservizio utilizza come strumento di migrazione Flyway. Esso permette di gestire le modifiche dello schema del database in modo dichiarativo e tracciabile attraverso script SQL versionati. L'esempio riportato sopra mostra proprio una delle migrazione che Flyway andrà ad eseguire per creare la policy di RLS sulla tabella email_senders.

È stata decisa una convenzione per il naming dei file di migrazione che segue il pattern: `V<YYYY_MM_DD_HH_MM>_<sequence>__<description>.sql`

- `V` prefisso che indica una migrazione versionata;
- `<YYYY_MM_DD_HH_MM>` rappresenta la data e l'ora di creazione della migrazione;
- `<sequence>` numero sequenziale per distinguere più migrazioni create nello stesso momento;
- `<description>` breve descrizione dell'obiettivo della migrazione separata da underscore.

Dunque, con questa convezione, l'esempio sopra riportato ha come nome file: `V2025_12_01_16_37_1__create_policy_email_senders_access.sql`

Un diverso tipo di storage è utilizzato per quanto riguarda le memorizzazione dei metadati delle email, i quali vengono salvati su Elasticsearch 7.17.28.
Elasticsearch è un motore di ricerca e analisi open source basato sulla libreria Apache Lucene [22]. Esso offre funzionalità di ricerca estremamente scalabili e performanti, acquisendo tipi di dati non strutturati da varie fonti e memorizzandoli in formati specializzati per ricerche ottimizzate basate sul linguaggio [22]. Tutte queste motivazioni rendono Elasticsearch la scelta ideale per memorizzare i metadati delle email, che richiedono ricerche rapide e analisi complesse su grandi volumi di dati. La versione 7.17.28 è stata selezionata poiché già disponibile nel cluster aziendale esistente. È però pianificato un aggiornamento alla versione 8, dato che la versione 7 è attualmente deprecata.

Una soluzione differente è stata adottata per lo storage degli allegati delle email,

i quali vengono memorizzati su Amazon S3. Amazon Simple Storage Service è un servizio di archiviazione di oggetti che offre scalabilità, disponibilità dei dati, sicurezza e prestazioni elevate [23]. Con S3 è possibile archiviare praticamente qualsiasi quantità di dati fino a exabyte ed è completamene elastico, cresce e si riduce automaticamente, mano a mano che si aggiungono o rimuovono dati [23]. Tutto ciò lo rende la soluzione perfetta per memorizzare gli allegati, che possono variare notevolmente in dimensione e quantità.

Per mantenere un certo ordine e un adeguato isolamento tra realm, il sistema crea una directory virtuale per ciascuna combinazione di applicazione e tenant. Tutte le directory risiedono in un unico bucket dedicato al microservizio, mentre la suddivisione logica viene ottenuta costruendo dinamicamente un percorso (chiave S3) che segue la seguente struttura:
`s3://<bucket>/<base_directory>/<tenant_id>/<application_id>/attachments/<attachment_id>`

- bucket: nome del bucket S3 dedicato al microservizio;
- base_directory: eventuale directory radice (opzionale);
- application_id: identificativo univoco dell'applicazione associata al realm corrente;
- tenant_id: identificativo univoco del tenant associato al realm corrente;
- attachment_id: identificativo univoco dell'allegato.

4.2.3 Provider Email

Come già accenato in precedenza, il microservizio supporta differenti provider di invio email, permettendo a ciascuna applicazione di scegliere quello più adatto alle proprie esigenze. Attualmente sono supportati tre provider principali: Amazon SES, server SMTP e Microsoft Graph API.

Amazon SES è il provider predefinito ed è progettato per l'invio di un grosso volume di email. L'integrazione avviene tramite l'SDK ufficiale AWS per Java.
Per quanto riguarda l'invio tramite server SMTP viene utilizzata la libreria `spring-boot-starter-mail` di Spring boot che utilizza Jakarta Mail, ma con una configurazione molto più semplice. Ciò permette l'invio delle email a qualsiasi server SMTP con supporto per diversi metodi di autenticazione, come da esempio l'autenticazione semplice tramite username e password oppure il protocollo OAuth 2.0 per Microsoft Azure e Google Workspace.

L'ultimo provider supportato (ad oggi) è Microsoft Graph API, il quale permette di inviare email tramite l'infrastruttura Microsoft 365 e Azure Active Directory. L'integrazione avviene tramite l'SDK ufficiale di Microsoft Graph.

4.2.4 Virtual Threads

Una scelta tecnologica particolarmente significativa nel progetto è stata l'adozione dei virtual threads, introdotti in Java 19 come preview feature e stabilizzati in Java 21 [24]. I virtual threads sono thread leggeri che permettono di gestire migliaia di connessioni concorrenti con un overhead minimo di risorse. Infatti, un virtual thread, anche se esegue sempre il suo codice su un thread del sistema operativo sottostante, non è legato, come i tradizionali platform thread, ad un thread del sistema operativo [25]. Proprio per questo, quando il codice in esecuzione in un virtual thread chiama un'operazione di I/O bloccante, il runtime Java sospende il virtual thread finché non può essere ripreso. Il thread del sistema operativo associato al virtual thread sospeso è ora libero di eseguire operazioni per altri virtual thread [25].
Tutte queste caratterisiche rendono i virtual threads particolarmente adatti per applicazioni server-side ad alta concorrenza come il microservizio zMailer, dove la maggior parte delle operazioni sono I/O-bound, come l'invio di email tramite provider esterni oppure l'accesso al database o a qualunque sistema di storage precedentemente elencato. Dunque, i virtual threads sono perfetti per creare molti thread sprecando pochissime risorse.
Per valutare concretamente i benefici dei virtual threads nel contesto del progetto, sono stati condotti benchmark comparativi tra platform threads e virtual threads su diverse combinazioni di JVM e servlet container. I test hanno simulato operazioni di scrittura su Elasticsearch, rappresentative delle operazioni I/O-bound tipiche dell'applicazione.
La metodologia di test ha previsto l'utilizzo dello strumento `wrk`, famoso strumento di benchmarking HTTP, con il seguente comando:

```
wrk -t4 -c500 -d60s -s config.lua http://localhost:8080/api/send
```

configurato per simulare 500 connessioni concorrenti per una durata di 60 secondi, inviando richieste POST con un certo payload JSON.
Sono state valutate cinque distribuzioni JVM (GraalVM, Temurin, Azul Zulu, Corretto e OpenJDK) in combinazione con tre servlet container (Tomcat, Jetty e Undertow[1]).

[1]Undertow risulta attualmente incompatibile con Spring Boot 4 e non è stato incluso nei test.

Configurazione	**Tomcat**		**Jetty**	
	Platform	**Virtual**	**Platform**	**Virtual**
GraalVM	11.045	13.695	10.742	13.774
Temurin	10.786	13.693	11.012	13.648
Azul Zulu	10.917	14.361	11.383	13.596
Corretto	10.931	14.574	10.812	14.095
OpenJDK	11.193	14.959	10.899	14.161

Tabella 4.3: Throughput medio (req/s) per configurazione

I risultati mostrano un miglioramento prestazionale significativo e consistente con l'adozione dei virtual threads. L'incremento di throughput medio si attesta tra il +24% e il +34% rispetto ai platform threads, con la configurazione OpenJDK su Tomcat che risulta essere la migliore con un picco di 14.959 richieste al secondo, utilizzando virtual threads, rispetto alle 11.193 con platform threads.
Questi risultati confermano l'idoneità dei virtual threads per il microservizio, dove la gestione efficiente di un elevato numero di operazioni I/O concorrenti è fondamentale per garantire prestazioni ottimali.

4.3 Architettura del Sistema

L'architettura del microservizio zMailer è stata progettata seguendo i principi del pattern MVC adattato al contesto di un'architettura RESTful APIs. Questo approccio consente una chiara separazione delle responsabilità tra i diversi livelli dell'applicazione per mantenere il codice modulare, testabile e facilmente manutenibile.

4.3.1 Panoramica Architetturale

Il sistema è strutturato come applicazione Spring Boot MVC, dove lo scambio di informazioni tra client e microservizio avviene attraverso il formato JSON. Esternamente, zMailer espone API REST ed una tipica richiesta segue il flusso descritto di seguito:

1. Il client invia una richiesta HTTP al microservizio zMailer tramite un endpoint REST;

2. La richiesta attraversa immediatamente un layer di autenticazione che estrae il JWT token dall'header Authorization, ne verifica la firma e ne estrae il claim nonce, che viene validato su Redis. Questo meccanismo consente di invalidare selettivamente uno o più token in qualsiasi momento, ad esempio in seguito a una rotazione delle credenziali o logout, senza dover attendere la naturale scadenza del token.

3. La richiesta viene poi intercetta da un interceptor, che verrà trattato nel dettaglio nelle prossime sezioni, il quale, a partire dal JWT precedentemente validato, estrae i claim necessari per risolvere e impostare il realm per l'intera durata della richiesta;

4. Successivamente la richiesta viene poi elaborata dal controller layer che si occupa di validare i dati ricevuti e chiamare i metodi del service layer;

5. Il service layer contiene la logica di business principale interagendo con i layer di persistenza per salvare o recuperare dati;

6. Il repository layer si interfaccia con il database utilizzando Spring Data JPA che semplifica l'implementazione del layer di persistenza riducendo il codice necessario per l'accesso ai dati;

7. Infine, il controller formatta la risposta del service in JSON e la invia al client.

4.3.2 Componenti Architetturali

Data Transfer Object

I DTO sono classi utilizzate per impacchettare e trasferire dati tra i diversi layer dell'applicazione in modo standardizzato.

Controller

I Controller rappresentano il punto di ingresso delle richieste HTTP e hanno la responsabilità esclusiva di ricevere, trasformare e passare i dati al service layer, delegando ogni altra operazione ai livelli inferiori. Nel contesto delle operazioni CRUD, zMailer adotta una struttura standardizzata che implementa cinque endpoint principali per ogni risorsa:

- **Index** (`GET /api/{version}/resources`): restituisce una lista paginata delle risorse;
- **Show** (`GET /api/{version}/resources/{id}`): recupera una singola risorsa dato il suo ID;
- **Store** (`POST /api/{version}/resources`): crea una nuova risorsa;
- **Update** (`PUT /api/{version}/resources/{id}`): aggiorna una risorsa esistente;
- **Delete** (`DELETE /api/{version}/resources/{id}`): elimina una risorsa.

La nomenclatura degli endpoint segue una sintassi formale che garantisce coerenza all'intera API, dove le risorse sono sempre espresse al plurale e possono essere annidate per rappresentare relazioni gerarchiche.

Request e Validazione

Le classi Request rappresentano i dati in ingresso, tipicamente il body di richieste POST, PUT o PATCH. Queste classi sono responsabili della validazione dei dati, che avviene tramite annotazioni appartenenti a Jakarta Bean Validation, e della loro trasformazione in oggetti DTO, che poi vengono passati ai layer inferiori. Questo approccio garantisce che il Service layer riceva sempre dati già validati e strutturati secondo il modello di dominio dell'applicazione.
Per implementare questo tipo di logica ogni Request class estende la classe astratta `RequestDTO` ed implementa l'interfaccia `ExportServiceDTO` che definisce il metodo `toServiceDTO` che la classe dovrà implementare per convertire i dati nel DTO specifico del service che andrà a chiamare.

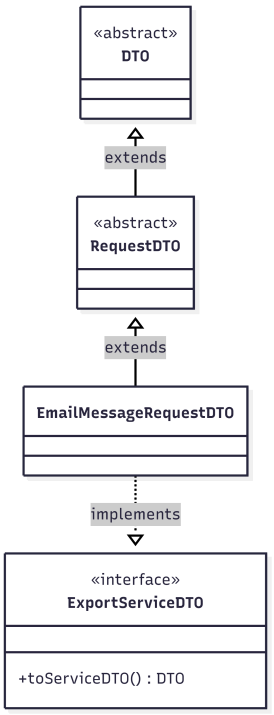

Figura 4.1: Esempio di struttura delle classi contenute nel package dto/request.

Service

Il service layer implementa la logica di business dell'applicazione. I servizi sono definiti come classi annotate con `@Service` e contengono metodi che eseguono operazioni specifiche sul dominio dell'applicazione. Questi metodi interagiscono con i repository per accedere ai dati e possono anche pubblicare eventi interni per notificare altri componenti dell'applicazione riguardo a cambiamenti di stato o azioni completate.

Repository

I Repository gestiscono l'interazione con il layer di persistenza. Nel progetto sono definiti come interfacce annotate con `@Repository` o estensioni dirette di interfacce base di Spring Data JPA come `JpaRepository`. Questo approccio sfrutta la generazione automatica delle query di Spring Data, semplificando il layer di persistenza ed eliminando codice non necessario. I bean Repository vengono iniettati direttamente nei servizi tramite constructor injection.

Entity

Le Entity rappresentano le entità del database. Per default, i modelli estendono la classe astratta `BaseEntity` e implementano l'interfaccia `Serializable`, fornendo una struttura base comune per tutte le entità del sistema.

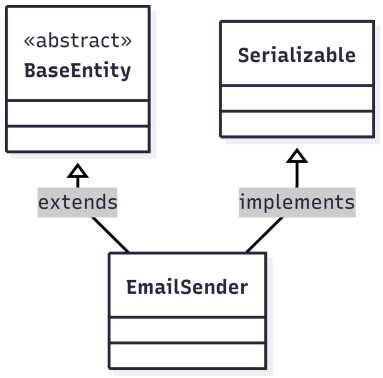

Figura 4.2: Esempio di struttura delle classi contenute nel package entity.

4.3.3 Gestione Eventi interni

Il microservizio zMailer utilizza un sistema di eventi basato sull'EDA pattern per gestire la comunicazione asincrona tra i vari componenti interni dell'applicazione. Il sistema di eventi è organizzato su tre livelli gerarchici:

1. **BaseEvent**: radice della gerarchia che estende `ApplicationEvent` di Spring Framework, fornendo la struttura fondamentale comune a tutti gli eventi;

2. **Eventi di Dominio**: per ogni dominio è definita una classe astratta che estende `BaseEvent` e funge da base per tutti gli eventi specifici di quel dominio;

3. **Eventi Concreti**: implementazioni specifiche che rappresentano eventi del sistema.

La pubblicazione degli eventi avviene tipicamente nel service layer attraverso `ApplicationEventPublisher`, mentre i listener interessati si registrano implementando `ApplicationListener`, per reagire agli eventi in modo sincrono o asincrono. Gli eventi seguono una convenzione di nomenclatura `{Entity}{Action}Event` e trasportano informazioni specifiche necessarie ai listener per eseguire le azioni appropriate.

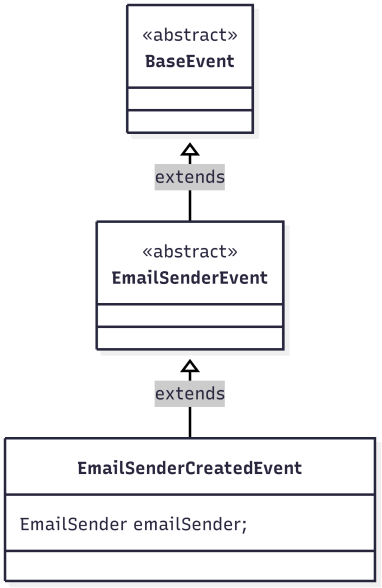

Figura 4.3: Esempio di struttura delle classi contenute nel package event.

4.3.4 Gestione delle Eccezioni

Il sistema implementa un'architettura di gestione delle eccezioni strutturata e gerarchica, progettata per fornire risposte di errore consistenti, controllo flessibile del logging e facile estensibilità per eccezioni specifiche di dominio.
Alla radice della gerarchia si trova `BaseException`, che fornisce gestione degli identificatori, mappatura ai codici di stato HTTP e possibilità di sopprimere il reporting con Sentry per eccezioni attese.
Sono state definite quattro annotazioni per configurare il comportamento delle eccezioni:

- `@Identifier`: assegna un identificatore univoco all'eccezione, come ad esempio `"exception.emailSender.notFound"`;
- `@HttpStatusCode`: mappa l'eccezione a un codice di stato HTTP;
- `@LoggingLevel`: specifica il livello di log (ERROR, WARN, INFO, DEBUG, TRACE);
- `@MuteReporting`: sopprime il reporting verso sistemi di monitoring esterni come Sentry. È molto utile per eccezioni non critiche, come, ad esempio, una risorsa non trovata.

Le eccezioni sono organizzate in package specifici per dominio e vengono intercettate dal `GlobalExceptionHandler`, un componente annotato con

`@RestControllerAdvice` che le converte in oggetti `ErrorResponseDTO` standardizzati.

```
1 @StandardException
2 @HttpStatusCode(404)
3 @Identifier("exception.emailSender.notFound")
4 public class EmailSenderNotFoundException extends
    BaseEmailSenderException {}
```

Listing 4.2: Esempio di eccezione personalizzata.

4.4 Implementazione dei Componenti Principali

Questa sezione descrive l'implementazione dei componenti architetturali più significativi del microservizio zMailer, ponendo particolare attenzione agli aspetti innovativi e alle sfide tecniche affrontate durante lo sviluppo.

4.4.1 Layer di Autenticazione

Il microservizio adotta un'architettura OAuth2 con un duplice ruolo: cioè funge sia da Authorization Server, ovvero firma e rilascia i token di accesso, sia da Resource Server, ovvero verifica la validità dei token. Infatti, ogni client, che desidera accedere alle risorse del microservizio, deve prima autenticarsi attraverso l'endpoint dedicato in cui fornisce `client_id` e `client_secret`, e in caso di successo, ottiene un JWT access token. Questo token deve poi essere incluso in ogni richiesta API per essere validato ed accedere alle risorse. Questa scelta progettuale elimina la dipendenza da provider di identità esterni, mantenendo l'autenticazione completamente gestita all'interno del microservizio.

L'implementazione si basa sul pattern OAuth2 Client Credentials Grant Type, che è lo standard per l'autenticazione server-to-server di applicazioni fidate senza il coinvolgimento di un utente finale [26]. Questa scelta risulta particolarmente appropriata nel contesto di zMailer dove i client sono applicazioni server-side e non degli utenti finali. Inoltre, questa modalità di autenticazione non prevede l'utilizzo di refresh token [26], poiché i client possono semplicemente richiedere un nuovo access token una volta scaduto quello in uso.
Nel ruolo di **Authorization Server**, il sistema:

- Emette i JWT access token per i client autenticati con un tempo di validità di 5 minuti;
- Valida le credenziali client confrontandole con quelle memorizzate nel database;

- Firma i token utilizzando una coppia di chiavi asimmetriche generate con l'algoritmo ECDSA con una curca ellittca P-512 solamente al primo avvio;
- Aggiunge dei claim personalizzati nei token che verranno utilizzati per definire il contesto applicativo ad ogni richiesta.

Nel ruolo di **Resource Server**, il sistema:

- Valida i JWT token in ingresso per le API protette;
- Estrae i dettagli di autenticazione dai claim del token;
- Verifica che il token sia ancora valido attraverso un meccanismo di nonce basato su Redis.

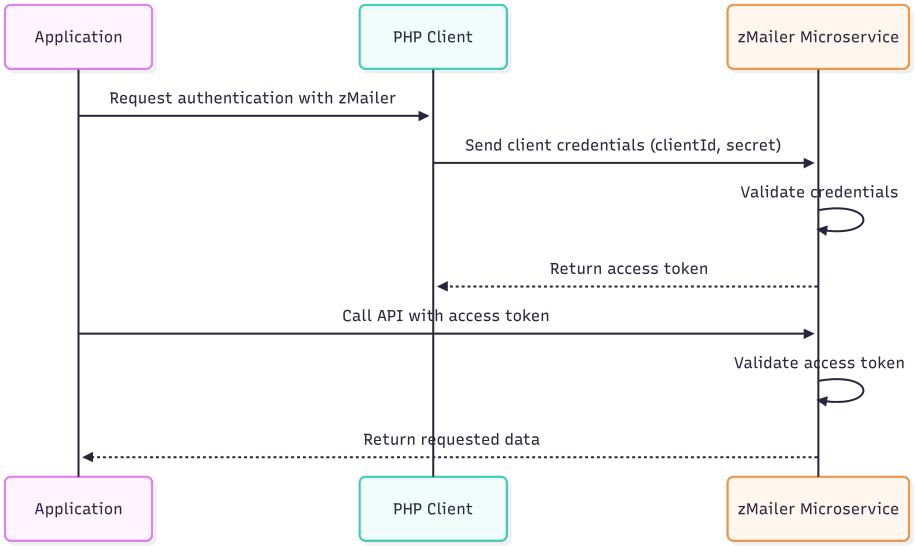

Figura 4.4: Flusso di Autenticazione.

Una caratteristica distintiva dell'implementazione è l'utilizzo di un repository client dinamico implementato attraverso l'interfaccia `RegisteredClientRepository`, messa a disposizione dalla libreria `spring-security-oauth2-authorization-server`. Questo componente fa da ponte tra Spring Authorization Server e il database applicativo, nel nostro caso PostgreSQL, consentendo il caricamento delle applicazioni client in modo completamente dinamico. Infatti, le informazioni necessarie all'autenticazione vengono recuperate dal database solo al momento della richiesta, permettendo così la registrazione di nuove applicazioni anche a runtime. Inoltre, il metodo `toRegisteredClient` converte le entità `Application` del dominio in oggetti `RegisteredClient` utilizzati da Spring Authorization Server.

Token JWT Personalizzati

Il sistema estende i token JWT standard con claim personalizzati attraverso la classe `CustomTokenCustomizer`, che implementa l'interfaccia `OAuth2TokenCustomizer`, che definisce il metodo `customize`. Durante la generazione di ogni access token, il customizer aggiunge tre claim:

- `user_type`: definisce lo scope del client autenticato. Sono previsti due valori possibili: `ADMIN`, che può eseguire qualsiasi operazione amministrativa come la creazione e l'eliminazione di applicazioni e tenant, e `TENANT`, che può eseguire le operazioni applicative ordinarie;
- `application_id`: UUID dell'applicazione autenticata, utilizzato successivamente per risolvere il contesto realm;
- `nonce`: identificatore univoco, sempre in formato UUID, del token, fondamentale per il meccanismo di logout e validazione dei token. Infatti esso viene memorizzato su Redis al momento dell'emissione del token e verificato ad ogni richiesta API.

Flusso di Autenticazione e Logout

Il processo completo di autenticazione è orchestrato dalla classe `AuthService` e segue questo flusso:

1. Il client invia una richiesta POST a `/api/v1/oauth2/token` con `client_id` e `client_secret`;
2. Il servizio recupera l'applicazione dal database tramite `ApplicationService`;
3. Viene verificata la corrispondenza del secret e lo stato attivo dell'applicazione;
4. L'Authorization Server valida le credenziali, genera il token JWT con i claim personalizzati e memorizza il nonce su Redis;
5. Il token di accesso viene restituito al client nel response body.

Il meccanismo di logout sfrutta il sistema di nonce per invalidare un token. La classe `NonceManager` gestisce questa funzionalità:

- Estrae il claim `nonce` dal token JWT presente nel Security Context;
- Elimina la corrispondente entry da Redis;
- Qualsiasi successiva richiesta con quel token fallirà.

Validazione Token e Authentication Filter

Ogni richiesta API, ad eccezione di quella di login, attraversa l'`AuthenticationFilter`, che è una classe che estende la classe astratta `OncePerRequestFilter`, della quale Spring garantisce che venga eseguita esattamente una volta per ogni richiesta.

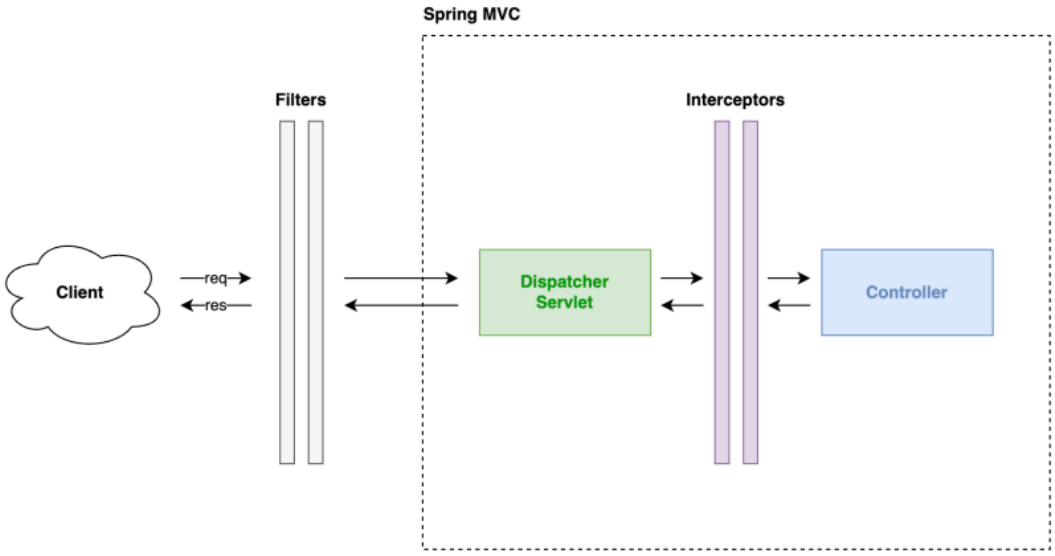

Figura 4.5: Posizionamento dell'Authentication Filter nella struttura di Spring.

Il filtro implementa la seguente logica di validazione:

1. Verifica se la richiesta è in whitelist (solo endpoint di login);
2. Invoca `NonceManager.validate()` per controllare l'esistenza del nonce su Redis;
3. Se il nonce non esiste o è scaduto lancia un'eccezione di tipo `UnauthorizedException`;
4. Se la validazione ha successo, passa la richiesta al filtro successivo nella catena.

L'utilizzo di Redis per la gestione dei nonce offre diversi vantaggi, tra cui quello di delegare direttamente a Redis la scadenza automatica dei token, senza dover mantenere uno stato applicativo complesso. Inoltre, garantisce anche che le entry vengano rimosse automaticamente alla scadenza del ttl.

4.4.2 Sistema di Risoluzione Realm

Il sistema di risoluzione realm gestisce il contesto multi-tenant dell'applicazione, estraendo automaticamente da ogni richiesta HTTP due entità fondamentali: tenant e application. Queste entità vengono risolte una sola volta per richiesta e rese disponibili a tutti i layer dell'applicazione attraverso un context holder thread-safe. L'intero sistema, dunque, è composto da quattro componenti principali, che collaborano per fornire un accesso trasparente al contesto multi-tenant:

1. **Realm**: classe immutabile che rappresenta il contesto per la richiesta corrente. Contiene tre campi: `RealmType`, `Tenant` e `Application`;
2. **RealmContextHolder**: fornisce accesso thread-safe al realm della richiesta corrente tramite l'utilizzo di `ThreadLocal`, che permette di memorizzare dei dati che saranno accessibili solo dallo specifico thread. Esso espone metodi getter che sollevano eccezioni esplicite `RealmAccessException` quando un'entità richiesta non è disponibile;
3. **RealmInterceptor**: interceptor Spring MVC che viene eseguito dopo il filtro di autenticazione. Si occupa di risolvere il realm all'inizio della richiesta e di ripulire il context holder al termine;
4. **RealmResolver**: contiene la logica di risoluzione delle singole entità per costruire il realm dato il JWT.

Flusso di Risoluzione

Il processo di risoluzione del realm segue un flusso sequenziale ben definito. Nel metodo `preHandle`, il `RealmInterceptor` invoca il `RealmResolver` che, a partire dal JWT presente nel `SecurityContextHolder`, esegue le operazioni di risoluzione per costruire il realm:

- **Application**: il metodo `resolveApplication` verifica che l'authentication sia di tipo `JwtAuthenticationToken`, estrae il claim `application_id` e recupera l'entità dal database;
- **Tenant**: individuato a partire dall'applicazione precedentemente risolta, poiché ogni applicazione è sempre associata ad uno specifico tenant;
- **RealmType**: determinato a partire dal claim `user_type` presente nel JWT, che distingue i casi `ADMIN` e `TENANT`.

Il realm risolto viene poi memorizzato nel `RealmContextHolder` e la richiesta procede attraverso i layer Controller e Service, che possono accedere al realm tramite i metodi statici del context holder. Al termine della richiesta, il metodo

`afterCompletion` del `RealmInterceptor` invoca `RealmContextHolder.clear()` in un blocco finally, garantendo la pulizia del ThreadLocal anche in caso di eccezioni.

4.4.3 Gestione della Multi-Tenancy

Una delle sfide più complesse affrontate nello sviluppo di zMailer è stata l'implementazione di un sistema di multi-tenancy che garantisse contemporaneamente da un lato l'isolamento completo dei dati tra i diversi realm, dall'altro la semplicità di sviluppo e le performance del sistema. La soluzione adottata utilizza Hibernate Discriminator, in cui tutti i dati sono memorizzati in un unico database e schema e sono partizionati per ogni realm tramite un discriminatore, rappresentato dall'id dell'applicazione. Questo approccio utilizza una singola connessione al database per tutti i tenant ed ad ogni query viene aggiunta automaticamente una condizione basata sul discriminatore, che consente di filtrare i dati in base al realm corrente [27].

Il componente centrale di questo meccanismo è il `DiscriminatorResolver`, che implementa l'interfaccia Hibernate `CurrentTenantIdentifierResolver`. Hibernate invoca automaticamente questo resolver prima di creare ogni sessione con il database, costruendo dinamicamente il discriminatore appropriato basanin base al realm corrente, disponibile nel `RealmContextHolder` descritto in precedenza. Se un componente del realm non è disponibile, il resolver utilizza un realm di default con un application_id costante, permettendo così anche le operazioni che non richiedono un contesto completo. Il resolver implementa inoltre un controllo sul discriminatore `root` attraverso il metodo `isRoot`, messo a disposizione dall'interfaccia, come verrà illustrato in seguito, particolarmente utile per specifiche query che non devono essere filtrate. Le entità soggette a questo filtraggio introducono un campo aggiuntivo `app_id`, annotato con l'annotazione `@TenantId` di Hibernate, che istruisce il framework ad utilizzare questa colonna per il filtraggio automatico delle query.

Un aspetto critico di questa implementazione riguarda il modo in cui Spring gestisce le sessioni. Infatti, di default, Spring, quando arriva una nuova richiesta HTTP, apre una nuova Hibernate Session a livello dell'HTTP Filter, quindi, prima che qualsiasi interceptor venga eseguito, come si vede nella figura 4.5 mostrata precedentemente. Inoltre, ogni volta che l'applicazione richiede una sessione, Spring riutilizza quella già aperta per tutta la durata della richiesta HTTP [28]. Come menzionato prima, il `DiscriminatorResolver` viene invocato da Hibernate al momento della creazione della sessione per determinare il discriminatore; tuttavia in questa fase il `RealmInterceptor` non è stato ancora eseguito, quindi il `RealmContextHolder` contiene un realm nullo. Di conseguenza il resolver non sarebbe in grado di costruire un discriminatore valido poiché il realm non è ancora disponibile, compromettendo l'intero meccanismo di multi-tenancy.

Per risolvere questa problematica, è stato fondamentale l'utilizzo della proprietà `spring.jpa.open-in-view` di Spring, che deve essere esplicitamente impostata a `false`. Con `open-in-view=false`, infatti, al contrario di quanto accadeva prima, la sessione Hibernate non viene aperta a livello del filter HTTP, ma viene creata nel Service layer al momento della prima annotazione `@Transactional`. A questo punto, il `RealmInterceptor` ha già risolto ed impostato il realm nel context holder, permettendo al `DiscriminatorResolver` di costruire correttamente il discriminatore. Questa sequenza temporale garantisce che ogni query Hibernate venga eseguita con il discriminatore appropriato per il realm della richiesta corrente.

Operazioni Cross-Tenant e Aspect-Oriented Programming

Un caso particolare riguarda le operazioni che necessitano di accedere ai dati di tutti i realm contemporaneamente, evitando il filtraggio automatico. Un esempio concreto sono i job schedulati che verificano periodicamente lo stato di verifica delle email e dei domini. Senza un meccanismo apposito, la query iniziale, per recuperare tutte le email non ancora verificate verrebbe, filtrata per il realm corrente, impedendo al job di operare correttamente sull'intero insieme di dati.
In Java un aspect è un modulo che incapsula della logica trasversale separandola dal codice principale dell'applicazione, permettendo di aggiungere comportamento a codice esistente senza modificarlo [29].

Dunque, la soluzione adottata sfrutta proprio questo meccanismo, definendo una custom annotation `@CrossTenant` da applicare ai metodi che devono operare senza filtraggio. L'aspect intercetta tutte le esecuzioni dei metodi annotati con `@CrossTenant` e imposta nel `RealmContextHolder` un realm speciale di tipo root, riconosciuto dal `CurrentTenantIdentifierResolverImpl` come indicatore per disattivare il filtraggio. Al termine dell'esecuzione del metodo, il realm originale viene ripristinato, garantendo che le operazioni successive tornino a funzionare con il contesto multi-tenant corretto.

```
1  @Aspect
2  @Component
3  @RequiredArgsConstructor
4  @Order(Ordered.HIGHEST_PRECEDENCE)
5  public class Aspect {
6
7      @Pointcut("annotation(@CrossTenant)")
8      public void pointCut() {}
9
10     @Around("pointCut()")
11     public void disableTenancy(ProceedingJoinPoint joinPoint)
       throws Throwable {
12         Realm realmBuffer = RealmContextHolder::getRealm()
13         try {
```

```
14              this.switchToRootApplication()
15              joinPoint.proceed()
16          }
17          finally {
18              if ( realmBuffer != null ) {
19                      RealmContextHolder.setRealm(realmBuffer)
20                      rlsService.activateForRealm(realmBuffer)
21              }
22              else {
23                      RealmContextHolder.clear()
24              }
25          }
26      }
27
28      private void switchToRootApplication() {
29          try {
30              Application application = new Application()
31              application.setId(ApplicationStdIdentifiers.ROOT)
32              Realm realm = new Realm(tenant: null, application:
    application, type: RealmType.TENANT)
33              RealmContextHolder.setRealm(realm)
34              rlsService.deactivate()
35          } catch (Exception ex) {
36              throw RootModeSwitchFailedTenancyException(ex)
37          }
38      }
39  }
```

Listing 4.3: Aspect per la gestione delle operazioni cross-realm.

La soluzione fin qui descritta offre numerosi vantaggi. Innanzitutto, in questo modo l'isolamento dei dati è garantito a livello di framework, riducendo drasticamente il rischio di errori umani che potrebbero esporre dati di un realm ad un altro. Inoltre, il codice all'interno dei service rimane pulito e focalizzato esclusivamente sulla logica di dominio, mentre viene delegata la gestione della multi-tenancy completamente all'infrastruttura.

Tuttavia, l'implementazione presenta anche alcune complessità. Infatti, l'esecuzione di codice prima di ogni query può impattare le performance, mentre la configurazione `open-in-view=false` richiede particolare attenzione alla gestione delle lazy-loading associations, che possono causare `LazyInitializationException` se non gestite correttamente.

4.4.4 Verifica Email e DKIM

Il microservizio zMailer implementa un sistema di verifica per indirizzi email e domini, garantendo che le email inviate siano autenticate correttamente e non

vengano classificate come spam. Il sistema gestisce l'intero ciclo di vita della verifica, dalla richiesta iniziale al monitoraggio asincrono dello stato, fino alla notifica del completamento tramite webhook.

Il sistema di verifica si integra con Amazon Simple Email Service come provider di autenticazione, sfruttando le sue capacità di verifica di identità email e configurazione DKIM per i domini. L'architettura si basa su quattro componenti tecnologici: Spring Scheduler per l'esecuzione periodica dei job di verifica, ShedLock per la gestione dei lock distribuiti, Redis per la memorizzazione dei lock e l'SDK AWS per Java per la comunicazione con SES.

Verifica Indirizzi Email

Il processo di verifica per indirizzi email individuali inizia quando un client invia una richiesta POST all'endpoint `/api/senders/verify` specificando l'indirizzo email da verificare. Il sistema esegue validazioni preliminari sull'email, verifica i permessi del tenant corrente e controlla che non esistano già verifiche attive per lo stesso indirizzo.
Superati i controlli, il sistema salva un'entità `EmailSender` nel database con stato `PENDING` e invoca il metodo `verifyEmailIdentity` dell'SDK AWS SES. Questo trigger fa sì che Amazon SES invii automaticamente un'email all'indirizzo specificato contenente un link di verifica. L'utente finale deve cliccare questo link per confermare il possesso dell'indirizzo email. Il sistema restituisce immediatamente al client una risposta con stato `PENDING` e avvia il processo di verifica asincrona.

Il componente `EmailVerificationJob` è uno Spring scheduled task annotato con `@Scheduled` che viene eseguito a intervalli regolari configurati tramite la proprietà `job.email-verification.interval`. Il job implementa la seguente logica: recupera tutti gli `EmailSender` con stato `PENDING` invocando il metodo `findAllEmailsByVerificationStatus` del service layer, che, come abbiamo spiegato nella precedente sezione, è annotato con `@CrossTenant` per ottenere tutte le email ancora `PENDING` senza distinzioni di realm.
Per ciascuna email pendente, il job invoca un metodo del service che effettua una chiamata all'API AWS SES per controllare lo stato di verifica di più indirizzi contemporaneamente, ottimizzando così il numero di chiamate di rete. Il metodo restituisce una mappa che associa ogni `EmailSender` al suo `EmailVerificationStatus` corrente.

A questo punto emerge una problematica critica relativa al contesto multitenant. Il job viene eseguito in un thread dello scheduler che non ha un realm impostato, mentre l'aggiornamento dello stato di verifica di un'email richiede il contesto corretto per garantire che i filtri Hibernate e le validazioni scope funzionino correttamente. Per risolvere questo problema è stato definito il componente `RealmExecutorService`, che fornisce il metodo `runAwareJob`, per eseguire delle

funzioni in un contesto specifico in cui il realm è definito. In particolare, questo metodo accetta un `Realm` esplicito e un `Runnable` rappresentante il lavoro da eseguire. Internamente, viene utilizzato l'`ExecutorService` che lancia un thread, il quale, prima di eseguire il job, invoca `RealmContextService.setRealm` per impostare il realm specifico della email da verificare. Questo garantisce che tutte le operazioni successive avvengano nel contesto corretto. Infine, in un blocco finally, il realm viene sempre ripulito tramite `realmContextService.clear`, prevenendo contaminazione tra le diverse esecuzioni dei thread.

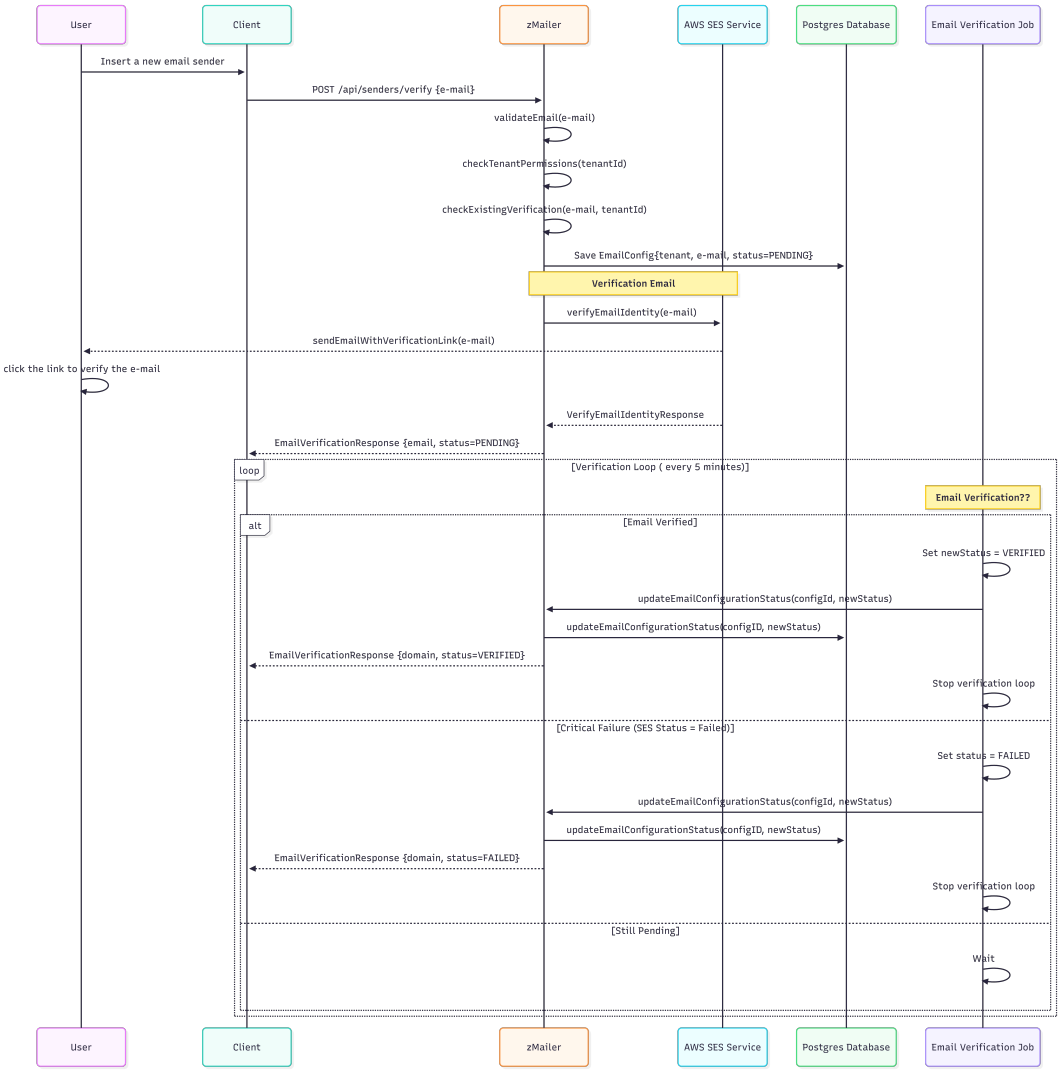

Figura 4.6: Flusso di verifica di un'email.

Verifica DKIM

La verifica del dominio, nonostante sia molto simile, in realtà è più complessa, poiché richiede la configurazione di record DNS da parte dell'utente. Il processo inizia con una richiesta POST all'endpoint `/api/dkim/verify` contenente il dominio da verificare. Il sistema valida il formato del dominio, verifica i permessi del tenant e controlla che non esistano configurazioni pendenti o verificate per lo stesso dominio. A questo punto, il sistema invoca `verifyDomainDkim` dell'SDK AWS SES, passando il nome del dominio, mentre Amazon SES risponde con un set di token DKIM univoci per quel dominio. Questi token vengono utilizzati dal sistema per generare i record DNS che l'utente dovrà configurare presso il proprio provider DNS. Tipicamente, si tratta di tre record CNAME nel formato `<token>._domainkey.<dominio>` che puntano a indirizzi forniti da AWS [30]. I record DNS generati vengono salvati nel database insieme ad un'entità `Domain` con stato `PENDING`. Il sistema restituisce immediatamente al client la lista di record DNS da configurare e lo stato `PENDING`, permettendo all'utente di procedere con la configurazione presso il proprio DNS. Il client può anche scaricare successivamente questi record tramite un endpoint dedicato che recupera i dati dal database e li formatta come file di testo.

Il componente `DKIMVerificationJob` opera in modo simile al precedente job. Per ogni dominio con stato `PENDING`, il job invoca `DKIMVerificationService.check` che interroga AWS SES sullo stato di verifica DKIM del dominio. Quando tutti i record richiesti sono presenti e validi, SES marca il dominio come verificato. Una volta che il job, allora, rileva questo cambiamento di stato e aggiorna il database, pubblica un `DKIMVerificationCompletedEvent`. Se SES segnala un fallimento critico o se viene superato il timeout di 48 ore, il sistema imposta lo stato a `FAILED`. Per semplicità non viene riportato il diagramma di sequenza per questo tipo di verifica data la sua somiglianza con quello della verifica email.

Gestione Lock Distribuiti con ShedLock

In un'architettura in cui è possibile avere molteplici repliche del microservizio zMailer in esecuzione contemporaneamente, è fondamentale garantire che i job vengano eseguiti esattamente una volta, evitando operazioni duplicate che causerebbero potenziali race condition sul database e spreco di risorse. Per questo motivo viene utilizzato ShedLock, una libreria Java che assicura che i job vengano eseguiti esattamente uno alla volta. In questa soluzione viene utilizzato Redis come storage condiviso tra tutte le istanze per salvare i lock e capire se un job è già in esecuzione.

```
1  @Scheduled(cron = "${job.email-verification.cron}")
2  @SchedulerLock(
3    name = "emailVerificationStatusLock",
4    lockAtMostFor = "${job.email-verification.lock-at-most-for}",
5    lockAtLeastFor = "${job.email-verification.lock-at-least-for}"
6    )
7  public void handle() { // Logica del job di verifica email }
```

Listing 4.4: Esempio di utilizzo di ShedLock.

Come si può vedere dal codice riportato sopra, ogni job è annotato con l'annotazione `@SchedulerLock`, specificando il nome di un lock univoco e due parametri temporali:

- `lockAtMostFor` definisce la durata massima per cui il lock può essere trattenuto, prevenendo deadlock nel caso in cui un'istanza fallisse mentre detiene il lock;
- `lockAtLeastFor` definisce la durata minima di trattenimento del lock, prevenendo esecuzioni troppo frequenti, indipendentemente dalla rapidità con cui il job completa l'esecuzione.

Quando arriva il momento dell'esecuzione, tutte le istanze tentano simultaneamente di acquisire il lock. Tuttavia, solo un'istanza riesce ad acquisire il lock, mentre le altre ricevono un risultato negativo e saltano l'esecuzione per quell'intervallo. L'istanza vincitrice esegue il job e, al completamento o allo scadere di `lockAtMostFor`, rilascia automaticamente il lock.

Sistema di Notifica Webhook

Il completamento delle verifiche descritte nelle sezioni precedenti devono essere comunicate alle applicazioni client che hanno richiesto la verifica. Per fare ciò è implementato un meccanismo di webhook basato su eventi, che fornisce notifiche push asincrone.

Tutti gli eventi, che devono essere notificati esternamente, implementano l'interfaccia `ExternalEvent` e sono annotati con `@ExternalIdentifier`, per specificare un identificatore univoco nel formato `event.<dominio>.<azione>`, che identifica l'evento. Questo identificatore permette ai client di sottoscriversi solo gli eventi di loro interesse. Inoltre, l'evento implementa il metodo `getPayload()`, che costruisce un oggetto contenente i dati rilevanti dell'evento.

A questo punto, le applicazioni client possono registrare sottoscrizioni webhook specificando l'evento di interesse e l'URL di callback. Quando un evento viene pubblicato, l'`ExternalEventDispatcherService` recupera tutte le sottoscrizioni attive e invia notifiche HTTP POST asincrone ai rispettivi endpoint.

Un aspetto importante dell'implementazione è che l'invio delle notifiche webhook avviene in modo completamente asincrono rispetto al flusso principale. Infatti, il dispatcher sottomette ogni invio HTTP a un `ExecutorService` dedicato, che esegue le richieste in thread separati. Questo design previene peggioramenti delle performance del sistema.

4.4.5 Processo di Invio

Il microservizio gestisce l'invio delle email attraverso un'architettura asincrona in cui sono presenti due fasi principali: l'accodamento della richiesta e il successivo invio asincrono. Questo disaccoppiamento, reso possibile dall'utilizzo di Apache Pulsar come message broker, garantisce che il client riceva una risposta immediata senza dover attendere il completamento dell'invio effettivo. In questo modo, aumenta la resilienza del sistema in caso di malfunzionamenti temporanei dei provider esterni.

Apache Pulsar

Per la gestione della coda dei messaggi è stato scelto Apache Pulsar come message broker. Pulsar è una piattaforma di messaggistica distribuita open source e cloud-native che combina le funzionalità dei sistemi di messaggistica tradizionali con quelle dei sistemi publish/subscribe, rendendolo particolarmente adatto per casi d'uso come la comunicazione tra microservizi. Tra le caratteristiche che hanno motivato questa scelta vi sono il supporto nativo alla multi-tenancy, fondamentale nel contesto di zMailer, la capacità di scalare orizzontalmente per gestire grandi volumi di messaggi e il supporto a subscription di tipo condiviso, che permette di distribuire i messaggi tra più consumer, abilitando il modello a worker pool utilizzato per l'invio asincrono delle email [31]. Inoltre, grazie alla replica su più nodi, garantisce che i messaggi non vengano persi nemmeno in caso di guasti hardware aumentando così l'affidabilità complessiva del sistema di invio [31].

Un ulteriore vantaggio è il supporto nativo per topic distinti, che in zMailer viene sfruttato per separare le email ad alta priorità da quelle a priorità normale, garantendo che le prime vengano elaborate indipendentemente dal volume delle seconde.

Accodamento

Il processo ha inizio quando un client invia una richiesta POST all'endpoint dedicato, fornendo nel body tutti i dati necessari: mittente, destinatari, oggetto, corpo del messaggio, eventuali allegati e altri metadati. Prima di essere accettata, la richiesta passa attraverso una serie di controlli:

- Validazione della struttura dei campi tramite Jakarta Bean Validation;

- Verifica della dimensione degli allegati;
- Eventuali vincoli applicativi del realm corrente.

Una volta superata la fase di validazione, il sistema procede con la persistenza dei dati in modo atomico:

- I metadati dell'email vengono salvati su Elasticsearch con stato `PENDING`;
- Gli allegati vengono caricati su Amazon S3.

Completata la persistenza, viene pubblicato un messaggio su Apache Pulsar che funge da notifica di una nuova email in coda. Il messaggio contiene l'identificativo del documento salvato su Elasticsearch e l'`app_id` del realm corrente, due informazioni sufficienti per ricostruire completamente il contesto necessario durante la fase di invio. Il client riceve immediatamente una risposta con l'identificativo dell'email con il quale potrà monitorare lo stato.

Come accennato, le email supportano due livelli di priorità, `HIGH` e `NORMAL`, che si traducono nell'utilizzo di topic Pulsar distinti. Questo garantisce che le email ad alta priorità vengano elaborate indipendentemente dal volume di quelle normali, evitando che un'elevata mole di email ordinarie possa ritardare la consegna di messaggi urgenti.

Scodamento

Sul lato consumer, un pool di worker thread rimane in ascolto sui topic Pulsar. Quando viene intercettato un nuovo messaggio, il worker avvia il processo di invio seguendo questi passi:

1. Recupera i metadati dell'email da Elasticsearch tramite l'identificativo contenuto nel messaggio;
2. Ricostruisce eventuali allegati da Amazon S3;
3. Ricostruisce il realm a partire dall'`app_id` contenuto nel messaggio, secondo il meccanismo descritto nelle sezioni precedenti;
4. Verifica che il realm ricostruito corrisponda a quello che ha originato la richiesta;
5. Seleziona dinamicamente il provider di invio corretto in base alla configurazione del realm;
6. Invia l'email tramite il provider selezionato e aggiorna lo stato su Elasticsearch in `SENT` o `FAIL`.

Rate Limit

Per garantire il rispetto dei limiti imposti dai provider esterni e prevenire l'invio concorrente di troppe email simultaneamente, è stato implementato un sistema di rate limiting basato su un semaforo distribuito, realizzato con Redis attraverso la classe `RedisSemaphoreService`. La struttura dati utilizzata è un hash set Redis in cui ogni entry rappresenta un invio in corso e ha un proprio TTL. In questo modo si garantisce la rimozione automatica delle entry anche in caso di crash del worker, evitando che il semaforo rimanga in uno stato inconsistente.

Il meccanismo segue un approccio ottimistico nel quale ogni worker aggiunge la propria entry al set, e successivamente verifica che la dimensione corrente del set rientri nel limite massimo configurato per il provider corrente. Se il controllo ha esito positivo, il worker procede con l'invio, al termine del quale rimuove la propria entry dal set. Se invece il controllo ha esito negativo, ovvero la dimensione supera il limite, il worker rimuove immediatamente la propria entry senza effettuare l'invio e ripubblica il messaggio su Pulsar con un delay, in modo che venga processato nuovamente dopo un intervallo di tempo, consentendo agli invii già in corso di completarsi.

Tale strategia ottimistica è stata preferita rispetto ad un approccio bloccante, poiché da un lato evita che i thread rimangano in attesa, consumando risorse inutilmente, dall'altro permette al sistema di mantenere la propria reattività anche in presenza di carichi elevati.

Capitolo 5

Deployment e Operatività

5.1 Containerizzazione di zMailer

La containerizzazione di zMailer rappresenta un passaggio fondamentale per garantire la portabilità e la riproducibilità del microservizio nei diversi ambienti. L'applicazione viene impacchettata in un'immagine container, che include il codice compilato e testato insieme alla JVM necessaria all'esecuzione.

Una scelta progettuale rilevante riguarda l'immagine base utilizzata per la JVM. Infatti, è stata adottata una *distroless image*, ovvero un'immagine che include solanto i componenti strettamente necessari all'esecuzione dell'applicazione, escludendo così qualsiasi altro strumento di sistema non essenziale. Questo approccio riduce drasticamente la superficie di attacco del container, minimizzando il numero di vulnerabilità potenzialmente sfruttabili, e produce immagini di dimensioni significativamente inferiori rispetto alle immagini base tradizionali.

5.2 Deployment

Il processo di deployment è completamente automatizzato tramite una pipeline GitLab CI/CD, che si occupa sia della fase di build che della distribuzione sul cluster Kubernetes. La pipeline ha due compiti principali: la costruzione e il packaging dell'applicazione, con la successiva pubblicazione dell'immagine su un container registry privato e la distribuzione automatica dell'immagine sul cluster Kubernetes dedicato.
All'interno dello stesso cluster è presente, oltre a zMailer, anche Apache Pulsar per la gestione della messaggistica.
Tutta l'infrastruttura del cluster e la relativa connettività sono gestite attraverso OpenTofu, adottando un approccio di Infrastructure as Code (IaC). Questo garantisce ripetibilità, versionamento e controllo delle modifiche all'infrastruttura,

trattando la configurazione infrastrutturale con lo stesso rigore applicato al codice applicativo.

5.3 Monitoring

Un sistema di monitoring efficace è essenziale per garantire la salute operativa del microservizio e individuare tempestivamente anomalie o degradazioni delle performance. zMailer adotta una strategia di observability basata sul monitoraggio degli errori tramite Sentry.

5.3.1 Monitoraggio degli Errori con Sentry

Sentry è integrato nel microservizio come sistema di error tracking in tempo reale. Ogni eccezione non gestita o errore applicativo viene automaticamente catturato, arricchito con il contesto di esecuzione e inviato alla piattaforma Sentry. Quest'utlima provvede a raggruppare gli errori simili, a tracciarli nel tempo e a notificare il team di sviluppo. Oltre al tracciamento degli errori, Sentry offre funzionalità di performance monitoring che permettono di analizzare i tempi di risposta delle chiamate API attraverso la visualizzazione grafica della sequenza temporale di tutte le operazioni che avvengono durante una singola richiesta HTTP, incluse le query al database e le chiamate verso servizi esterni, con i relativi tempi di esecuzione. Per arricchire il contesto degli eventi tracciati, il sistema associa a ciascuno di essi una serie di tag personalizzati: l'ambiente di esecuzione, il tenant e l'applicazione che hanno originato la richiesta, e il correlation-id, supportato come header opzionale nelle chiamate HTTP. Quest'ultimo risulta particolarmente utile per ricostruire il flusso completo di una richiesta distribuita in caso di errore. È inoltre prevista l'integrazione con ClickUp, che consentirà la creazione automatica di task al verificarsi di errori, facilitando il tracciamento e la risoluzione sistematica dei problemi in produzione.

Capitolo 6

Risultati e Valutazioni

Questo capitolo presenta una valutazione complessiva di zMailer, analizzandone le performance attraverso test di carico e confrontando il sistema realizzato con la soluzione precedente. Vengono, inoltre, discusse le principali sfide tecniche incontrate durante lo sviluppo e le soluzioni architetturali adottate per risolverle.

6.1 Metodologia dei Test di Carico

I test di carico sono stati condotti utilizzando `wrk`, uno strumento di benchmarking HTTP moderno e ad alte prestazioni, progettato per generare carichi significativi operando su una singola CPU multi-core [32].

L'endpoint sottoposto a stress test è quello relativo all'accodamento delle email, che rappresenta il punto di ingresso principale del microservizio. Esso utilizza un tipo di codifica definito multipart/form-data per accettare richieste HTTP, che includono sia campi di testo che file binari. Per rendere il test rappresentativo di un caso d'uso reale, ogni richiesta ha incluso un singolo allegato di circa 1 MB. Il comando utilizzato per tutti i test è stato il seguente:

```
wrk -t3 -c20 -d10s -s multipart.lua
```

I parametri configurati sono:

- `-t3`: 3 thread paralleli per la generazione del carico;
- `-c25`: 25 connessioni HTTP concorrenti mantenute aperte durante l'intera durata del test;
- `-d10s`: durata del test fissata a 10 secondi;
- `-s multipart.lua`: script Lua custom che costruisce ad ogni iterazione una richiesta `multipart/form-data` completa.

I test sono stati eseguiti in tre configurazioni distinte per valutare il comportamento del sistema al variare delle risorse disponibili:

- **No-stress**: nessun vincolo di risorse imposto al container, per misurare le performance massime raggiungibili;
- **Stress 1GB RAM, 0.5 CPU**: configurazione fortemente vincolata, rappresentativa di ambienti di produzione con risorse ridotte al minimo;
- **Stress 2GB RAM, 1 CPU**: configurazione intermedia, più vicina a un tipico deployment di produzione.

Per garantire la validità statistica dei risultati, ciascuna configurazione è stata sottoposta a 10 esecuzioni indipendenti del benchmark, i cui valori sono stati successivamente aggregati tramite media aritmetica.

6.2 Analisi dei Risultati

Metrica	**No-stress**	**1GB - 0.5 CPU**	**2GB - 1 CPU**
Totale richieste	3285	612	1260
Richieste per secondo	328.50	61.20	125.99
Tempo medio di risposta	3.04 ms	16.34 ms	7.94 ms
Richieste fallite/timeout	0	0	0
Tetto massimo req/s	328	61	126

Tabella 6.1: Risultati dei test di carico.

I risultati mostrano che il microservizio si comporta in modo eccellente in tutti e tre gli ambienti testati. In particolare, in nessuno scenario si sono verificate richieste fallite o timeout, a conferma della robustezza e dell'affidabilità del sistema anche sotto carico.

Nell'ambiente senza vincoli di risorse, zMailer raggiunge un throughput di 328.50 richieste al secondo con un tempo medio di risposta di soli 3.04 ms, dimostrando le elevate performance del sistema in condizioni ottimali. Anche negli ambienti con risorse limitate il sistema mantiene un comportamento stabile, con 2GB di RAM e 1 CPU il throughput si attesta a circa 126 req/s con un tempo medio di risposta di 7.94 ms, mentre con sole risorse da 1GB e 0.5 CPU il sistema riesce comunque a gestire oltre 61 req/s. Questi risultati confermano che zMailer scala in modo proporzionale alle risorse disponibili, senza degradazioni anomale o comportamenti instabili.

Dunque, il confronto tra il sistema precedente e zMailer evidenzia un miglioramento significativo su tutte le dimensioni rilevanti.

Dal punto di vista delle performance, la precedente soluzione integrata nel monolite era in grado di gestire circa 20 richieste al secondo. zMailer, anche nell'ambiente più vincolato testato (1GB RAM, 0.5 CPU), raggiunge 61 req/s, rappresentando un miglioramento circa tre volte superiore. Nell'ambiente con risorse standard (2GB RAM, 1 CPU) il throughput sale a 126 req/s, ovvero più di sei volte rispetto alla soluzione precedente. In condizioni ottimali, il sistema supera le 328 req/s.

Dal punto di vista dell'affidabilità, la soluzione precedente non disponeva di meccanismi di retry né di gestione strutturata degli errori, con conseguente perdita di messaggi in caso di malfunzionamenti dei provider esterni. zMailer introduce un sistema di accodamento persistente su Apache Pulsar che prevede il reinvio con delay nel caso in cui si superino i limiti consentiti dai provider o si verifichi qualsiasi altro errore, fino ad un massimo di 3 tentativi. In questo modo si garantisce che nessun messaggio venga perso.

Infine, dal punto di vista della scalabilità, il vecchio sistema poteva essere scalato solo insieme all'intera applicazione monolitica. zMailer può essere scalato in modo indipendente in base al carico effettivo, ottimizzando i costi operativi e permettendo una gestione granulare delle risorse.

6.3 Qualità del Codice

Per garantire un elevato standard qualitativo del codice sorgente, sono stati integrati nel processo di build diversi strumenti di analisi statica e di uniformazione della formattazione.

In particolare, per l'analisi statica sono stati adottati PMD, applicato al codice sorgente principale per rilevare pattern problematici, codice ridondante e potenziali bug, e SpotBugs con il plugin FindSecBugs per l'identificazione di vulnerabilità di sicurezza comuni, come injection, gestione scorretta delle credenziali e problemi crittografici.

Per quanto riguarda la formattazione, è stato integrato Spotless con un ruleset Eclipse personalizzato, che garantisce l'uniformità stilistica dell'intera base di codice e previene inconsistenze introdotte da diversi sviluppatori.
L'obiettivo è che queste verifiche vengano eseguite automaticamente nella pipeline di build, in modo da impedire l'integrazione di codice che non rispetti gli standard definiti.

Questo investimento nella qualità del codice rappresenta un netto miglioramento rispetto alla soluzione precedente, nella quale l'assenza di strumenti di analisi automatica rendeva difficile mantenere standard omogenei nel tempo.

6.4 Test

Un ulteriore ambito in cui è stato effettuato un miglioramento netto riguarda l'introduzione di test automatizzati, completamente assenti nel sistema precedente.

Per zMailer sono stati sviluppati sia unit test sia feature test, progettati per verificare rispettivamente il corretto funzionamento dei singoli componenti e il comportamento dei principali flussi applicativi end-to-end. Grazie a questa attività è stata raggiunta una copertura del codice pari a circa il 90%.

Oltre ai test automatizzati, è previsto che in azienda vengano eseguite anche attività di verifica manuale delle funzionalità applicative e penetration test, con l'obiettivo di valutare la sicurezza complessiva del servizio.

L'adozione di queste diverse modalità di verifica consente di validare il sistema sia dal punto di vista funzionale sia sotto il profilo della robustezza e della sicurezza.

6.5 Documentazione

Un ulteriore aspetto differenziante rispetto alla soluzione precedente è il forte investimento nella documentazione, progettata per rendere zMailer un servizio facilmente installabile, integrabile e manutenibile nel tempo.
La documentazione prodotta si articola su più livelli:

- Documentazione del funzionamento interno: illustra l'architettura del microservizio, i flussi di elaborazione delle email, il modello di multi-tenancy adottato e il comportamento generale del sistema;
- Documentazione di integrazione: fornisce le informazioni necessarie ai team che devono integrare zMailer nei propri sistemi;
- Documentazione degli endpoint tramite Swagger: gli endpoint REST esposti dal microservizio sono documentati tramite OpenAPI/Swagger, con descrizione dei parametri, dei tipi di dato accettati, dei codici di risposta e degli esempi di utilizzo, accessibile direttamente tramite interfaccia web interattiva.

Questo approccio riduce significativamente il tempo necessario all'inserimento di nuovi sviluppatori nel progetto e abbassa il rischio di errori di configurazione e integrazione, problemi che invece affliggevano il sistema precedente per via della documentazione assente.

6.6 Sfide e Difficoltà Incontrate

Lo sviluppo di zMailer ha presentato diverse sfide tecniche significative, alcune delle quali hanno richiesto soluzioni architetturali non banali.

L'obiettivo principale del microservizio zMailer era garantire elevate performance e affidabilità nel processo di invio delle email, evitando la perdita di messaggi anche in presenza di errori temporanei dei provider esterni. Per raggiungere questo obiettivo è stata progettata un'architettura basata sulla separazione tra la fase di accodamento sincrono della richiesta e la fase di invio asincrono delle email.

Quando un client effettua una richiesta di invio email, il microservizio gestisce inizialmente l'operazione in modo sincrono. Durante questa fase vengono salvati i metadati dell'email all'interno di Elasticsearch, mentre eventuali allegati vengono memorizzati su Amazon S3. Una volta completata la persistenza dei dati, il sistema accoda un messaggio contenente le informazioni necessarie per l'invio all'interno di Apache Pulsar. Al termine di questa operazione viene restituito al client l'identificativo univoco dell'email, permettendo così al chiamante di tracciare successivamente lo stato del messaggio.

L'invio effettivo dell'email avviene, invece, in modo asincrono attraverso un consumer che elabora i messaggi presenti nella coda Pulsar. Questo approccio consente di disaccoppiare completamente il tempo di risposta dell'API dal tempo necessario per l'invio dell'email, migliorando significativamente la latenza percepita dal client e aumentando la capacità del sistema di gestire elevati volumi di richieste.

Durante la fase di invio asincrono, il sistema gestisce eventuali errori dei provider implementando un meccanismo di retry con ritardo. In caso di fallimento dell'invio, il messaggio viene reinserito nella coda fino a un numero massimo di tentativi configurato. Se anche l'ultimo tentativo non va a buon fine, lo stato dell'email viene aggiornato come `FAIL`. In ogni fase del processo lo stato del messaggio viene mantenuto aggiornato all'interno di Elasticsearch, consentendo di monitorare in modo consistente il ciclo di vita completo dell'email.

Questa architettura asincrona ha permesso, dunque, come accennato in precedenza, di ottenere un sistema resiliente e scalabile, capace di garantire l'affidabilità dell'invio anche in presenza di carichi elevati o malfunzionamenti temporanei dei servizi esterni.

Tuttavia, una delle sfide più complesse è stata certamente l'implementazione del sistema di multi-tenancy. Infatti, la progettazione del sistema ha richiesto una valutazione delle principali strategie architetturali comunemente adottate. Come descritto nei capitoli precedenti, esistono 3 approcci principali per isolare le informazioni in questi sistemi multi-tenant, che vanno di pari passo con diverse definizioni dello schema del database e configurazioni JDBC. La scelta per questo progetto è ricaduta sull'utilizzo di un unico database in cui tutti i realm condividono lo stesso schema relazionale, mentre la separazione dei dati viene garantita attraverso una colonna discriminatrice presente nelle tabelle interessate. Questa scelta ha permesso di minimizzare la complessità infrastruturale e mantenere alte le performance riducendo la frammentazione dei dati.

Ciò ha comportato, però, anche la necessità di gestire il ciclo di vita della

sessione Hibernate in relazione alla risoluzione del realm. Infatti, l'integrazione con Hibernate, che supporta il modello discriminator tramite meccanismi di filtraggio automatico delle query basati sul realm corrente, ha evidenziato un problema di sincronizzazione tra l'apertura della sessione e la disponibilità del realm nel context holder. Inizialmente, la sessione veniva aperta prima che il realm fosse risolto, causando errori di accesso ai dati. La soluzione adottata è stata quella di disabilitare esplicitamente la funzionalità `open-in-view` di Spring, che mantiene la sessione aperta per tutta la durata della richiesta, e di gestire manualmente l'apertura e chiusura della sessione, in modo da garantire che avvenga solo dopo che il realm sia stato correttamente risolto.

Un'altra sfida rilevante ha riguardato la gestione del contesto multi-tenant nei job schedulati, che per natura operano al di fuori del ciclo di vita di una normale richiesta HTTP. La soluzione tramite `@CrossTenant` e `RealmExecutorService` ha richiesto una progettazione accurata per garantire che ogni operazione avvenisse nel realm corretto senza contaminazione tra thread concorrenti.

Infine, la progettazione del sistema di rate limiting ha richiesto una riflessione approfondita su quale potesse essere la strategia più adatta al contesto di invio asincrono delle email. Si è optato per un approccio ottimistico basato sull'utilizzo di un hash set su Redis per la gestione del TTL, al fine di evitare possibili situazioni di deadlock e migliorare le prestazioni del sistema.

Capitolo 7

Conclusioni e Sviluppi Futuri

7.1 Conclusioni

Il lavoro presentato in questa tesi ha descritto la progettazione e lo sviluppo di zMailer, un microservizio dedicato alla gestione centralizzata dell'invio di email in un contesto multi-tenant. Il sistema è stato realizzato con l'obiettivo di superare i limiti della soluzione precedente, integrata nel monolite applicativo, che non garantiva affidabilità, osservabilità né scalabilità adeguate.

I risultati ottenuti confermano il raggiungimento di questi obiettivi. I test di carico hanno dimostrato che zMailer è in grado di gestire fino a 328 richieste al secondo in condizioni ottimali, con un tempo medio di risposta di 3.04 ms e zero richieste fallite o andate in timeout in tutti gli scenari testati. Anche nelle configurazioni più vincolate, il sistema mantiene un comportamento stabile e proporzionale alle risorse disponibili, raggiungendo 61 req/s con sole risorse da 1GB di RAM e 0.5 CPU. Rispetto alla soluzione precedente, che si attestava a circa 20 req/s, si registra, dunque un miglioramento delle performance, che va da un minimo di tre volte fino a oltre sedici volte in condizioni ottimali.

Al di là delle performance, zMailer introduce miglioramenti strutturali significativi: il sistema di accodamento persistente su Apache Pulsar garantisce che nessun messaggio venga perso in caso di malfunzionamenti, il tracciamento dello stato delle email su Elasticsearch offre visibilità completa sul ciclo di vita di ogni messaggio e l'integrazione con Sentry consente il monitoraggio proattivo degli errori in produzione. L'adozione di Infrastructure as Code tramite OpenTofu e il deployment automatizzato su Kubernetes completano un sistema progettato per essere affidabile, osservabile e operativamente maturo.

7.2 Sviluppi Futuri

Il lavoro svolto pone le basi per ulteriori evoluzioni, sia a livello di architettura condivisa tra i microservizi che a livello di ottimizzazione.

7.2.1 zMscore

Durante lo sviluppo di zMailer è emersa la necessità di una base solida e condivisa per tutti i futuri microservizi dell'ecosistema. La logica trasversale attualmente integrata nel microservizio, che comprende il sistema di multi-tenancy, la gestione dell'autenticazione e le funzionalità infrastrutturali comuni, verrà estratta in una libreria dedicata denominata `zMscore`. Questa libreria sarà distribuita come dipendenza Maven e costituirà il punto di partenza obbligatorio per tutti i microservizi futuri, garantendo coerenza architetturale, riducendo la duplicazione del codice e capitalizzando le soluzioni alle sfide tecniche affrontate durante lo sviluppo di zMailer.

7.2.2 Compilazione Nativa con GraalVM

Un'altra direzione di sviluppo riguarda la compilazione nativa dell'applicazione tramite GraalVM Native Image in sostituzione del classico artefatto JAR eseguito sulla JVM. La compilazione nativa produce un eseguibile autonomo, che non richiede una JVM installata, con tempi di avvio nell'ordine dei millisecondi e un consumo di memoria significativamente ridotto rispetto all'esecuzione tradizionale. Questo si tradurrebbe in immagini container più leggere e in un comportamento più reattivo durante le fasi di scaling orizzontale sul cluster Kubernetes.

Tuttavia, questa transizione presenta delle sfide non banali. La principale criticità riguarda i tempi di compilazione: la generazione di un'immagine nativa richiede diversi minuti, rendendo la pipeline CI/CD inadatta alla distribuzione rapida di hotfix urgenti. Una possibile soluzione prevede la coesistenza di due pipeline distinte: una pipeline standard basata su JAR, adatta ai rilasci frequenti, e una pipeline nativa dedicata ai rilasci pianificati, che produca immagini ottimizzate per l'ambiente di produzione. La definizione della strategia più adatta a bilanciare velocità di rilascio e ottimizzazione delle prestazioni del sistema è oggetto di valutazione futura.

Bibliografia

[1] *What are microservices and why use them?* LogicMonitor. Nov. 2024. URL: https://www.logicmonitor.com/blog/what-are-microservices (cit. a p. 2).

[2] Sarah Laoyan. *What is Agile methodology?* asana. Feb. 2025. URL: https://asana.com/it/resources/agile-methodology (cit. a p. 4).

[3] *Che cos'è l'architettura dei microservizi?* Google. URL: https://cloud.google.com/learn/what-is-microservices-architecture?hl=it (cit. a p. 6).

[4] Nupura Torveka e Pravin Game. «Microservices and Its Applications An Overview». In: *International Journal of Computer Sciences and Engineering* 4 (apr. 2019), pp. 803–809 (cit. a p. 7).

[5] Victor Velepucha e Pamela Flores. «A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges». In: *IEEE Access* 11 (gen. 2023), pp. 88339–88358 (cit. alle pp. 7–16).

[6] J. Lewis e M. Fowler. «Building Microservices: Designing Fine-Grained». In: *MartinFowler.com* 25.14–26 (2014), p. 12 (cit. alle pp. 9–12).

[7] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA, USA: O'Reilly Media, 2015 (cit. alle pp. 9–12).

[8] D. L. Parnas. «On the criteria to be used in decomposing systems into modules». In: *Commun. ACM* 15.12 (Dec 1972), pp. 1053–1058 (cit. a p. 11).

[9] *Comunicazione in un'architettura di microservizi*. Microsoft. URL: https://learn.microsoft.com/it-it/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture (cit. a p. 17).

[10] Muhammad Niswar, Reza Arisandy Safruddin, Anugrayani Bustamin e Iqra Aswad. «Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC». In: *International Journal of Advanced Computer Science and Applications (IJACSA)* 13.10 (2022), pp. 123–131 (cit. alle pp. 18–20).

[11] Ashwin Chavan. «Exploring Event-Driven Architecture in Microservices: Patterns, Pitfalls and Best Practices». In: *International Journal of Science and Research Archive* 04.01 (2021), pp. 229–249 (cit. alle pp. 22, 23).

[12] Nuno Mateus-Coelho, Manuela Cruz-Cunha e Luís Gonzaga Ferreira. *Security in Microservices Architectures*. Vol. 181. Procedia Computer Science. Vilamoura, Portugal: Elsevier, 2021, pp. 1225–1236 (cit. alle pp. 26–28, 30–32).

[13] Martin Kaloudis. «Evolving Software Architectures from Monolithic Systems to Resilient Microservices: Best Practices, Challenges and Future Trends». In: *International Journal of Advanced Computer Science and Applications (IJACSA)* 15.9 (2024), pp. 1–8. URL: https://www.ijacsa.thesai.org (cit. alle pp. 33, 34, 36, 37).

[14] Sanghamithra Duggirala e Avneesh Kumar. «Microservices Adoption and Migration: Strategies for Modernizing Legacy Applications». In: *Journal of Emerging Technologies and Innovative Research (JETIR)* 12.2 (2025), h829–h836. ISSN: 2349-5162. URL: https://www.jetir.org/view?paper=JETIR2502790 (cit. alle pp. 35–38).

[15] *Strangler fig pattern*. Wikipedia. 2025. URL: https://en.wikipedia.org/wiki/Strangler_fig_pattern (cit. a p. 38).

[16] Digital OIT. *Choosing the right technology for business success*. URL: https://digitaloit.com/choosing-the-right-technology-for-business-success/ (cit. a p. 45).

[17] Google Cloud. *What is PostgreSQL?* URL: https://cloud.google.com/discover/what-is-postgresql?hl=it (cit. alle pp. 46, 47).

[18] PostgreSQL Global Development Group. *Row Security Policies*. 2025. URL: https://www.postgresql.org/docs/current/ddl-rowsecurity.html (cit. a p. 47).

[19] Gabriele Romanato. *Abilitare e usare la ricerca FULLTEXT in PostgreSQL*. 2025. URL: https://gabrieleromanato.com/2025/08/abilitare-e-usare-la-ricerca-fulltext-in-postgresql (cit. a p. 47).

[20] PostgreSQL Global Development Group. *UUID Type*. 2025. URL: https://www.postgresql.org/docs/current/datatype-uuid.html (cit. a p. 47).

[21] UUID7.com. *UUID7 – The time-sortable identifier for modern databases*. URL: https://uuid7.com/ (cit. a p. 47).

[22] IBM. *What is Elasticsearch?* URL: https://www.ibm.com/it-it/think/topics/elasticsearch (cit. a p. 48).

[23] Amazon Web Services. *Amazon S3: Cloud Object Storage*. URL: https://aws.amazon.com/it/s3/ (cit. a p. 49).

[24] Ron Pressler e Alan Bateman. *JEP 444: Virtual Threads*. `https://openjdk.org/jeps/444`. Created: 2023-03-06; Updated: 2025-10-30. 2023 (cit. a p. 50).

[25] Oracle Corporation. *Virtual Threads — Java SE 21 Documentation*. 2023. URL: `https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html` (cit. a p. 50).

[26] Dick Hardt, cur. *The OAuth 2.0 Authorization Framework*. RFC. Section 4.4: Client Credentials Grant. Internet Engineering Task Force, ott. 2012. URL: `https://datatracker.ietf.org/doc/html/rfc6749` (cit. a p. 57).

[27] Hibernate Team. *Chapter 16: Multi-tenancy*. 2011. URL: `https://docs.hibernate.org/devguide/en-US/html/ch16.html` (cit. a p. 62).

[28] Ali Dehghani. *A Guide to Spring's Open Session in View*. Baeldung. 2019. URL: `https://www.baeldung.com/spring-open-session-in-view` (cit. a p. 62).

[29] Spring Framework Team. *AOP Concepts: Introduction and Definitions*. Spring Framework Reference Documentation. VMware, Inc. 2025. URL: `https://docs.spring.io/spring-framework/reference/core/aop/introduction-defn.html` (cit. a p. 63).

[30] Amazon Web Services. *VerifyDomainDkim – Amazon Simple Email Service API Reference*. Amazon Web Services, Inc. 2026. URL: `https://docs.aws.amazon.com/ses/latest/APIReference/API_VerifyDomainDkim.html` (cit. a p. 67).

[31] Alice Gomstyn e Alexandra Jonker. *Cos'è Apache Pulsar*. IBM. 2025. URL: `https://www.ibm.com/it-it/think/topics/apache-pulsar` (cit. a p. 69).

[32] Felipe Dutra Tine e Silva. *Intelligent Benchmark with wrk*. Medium. 2018. URL: `https://medium.com/@felipedutratine/intelligent-benchmark-with-wrk-163986c1587f` (cit. a p. 74).