ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Sviluppo di un pannello Web a supporto di un filtro DNS

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Prof. Mirko Viroli

Candidato
Alessandro Valmori

Correlatore
Dott. Nicolas Farabegoli

II Sessione di Laurea
Anno Accademico 2024-2025

Sommario

Questa tesi descrive lo sviluppo di una dashboard web in modalità readonly per la visualizzazione e l'analisi dei dati di filtraggio DNS, realizzata in collaborazione con FlashStart SRL, azienda leader nel settore del filtraggio dei contenuti web. Il progetto nasce come soluzione temporanea per colmare una lacuna funzionale della piattaforma esistente, fornendo ai clienti dell'azienda uno strumento dedicato all'analisi dei dati fino al rilascio della nuova infrastruttura aziendale.

L'obiettivo del lavoro è duplice: sviluppare un'applicazione web funzionale e sicura che implementi meccanismi di autenticazione avanzati per proteggere l'accesso ai dati sensibili, e analizzare criticamente come i principi della programmazione orientata agli oggetti guidino le scelte architetturali in un contesto industriale reale. Il sistema implementa un'architettura full-stack moderna, utilizzando tecnologie reattive per garantire scalabilità ed efficienza, con particolare attenzione alla sicurezza attraverso sistemi di autenticazione stateless e gestione sicura delle sessioni utente.

Questo lavoro rappresenta un esempio concreto di come i principi teorici dell'ingegneria del software trovino applicazione pratica nella risoluzione di problemi aziendali, evidenziando l'importanza dei design pattern e delle metodologie orientate agli oggetti nello sviluppo di software industriale.

Alla mia famiglia, a Linda e ai miei amici.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Contesto Aziendale e Motivazione del Progetto

Il presente lavoro di tesi si inserisce in un contesto industriale specifico, frutto della collaborazione con FlashStart SRL, un'azienda italiana con sede a Cesena, specializzata nello sviluppo e nella fornitura di soluzioni di filtraggio dei contenuti e protezione da minacce informatiche basate su tecnologia DNS (Domain Name System). I servizi offerti da FlashStart si rivolgono a una clientela diversificata, che include aziende, istituzioni educative e pubbliche amministrazioni, fornendo loro strumenti per garantire una navigazione sicura e controllata.

Al momento dell'inizio del percorso di tirocinio, nel mese di giugno 2025, l'azienda si trovava in una fase di significativa evoluzione tecnologica e strategica. Era infatti in corso un processo di completa reingegnerizzazione della propria piattaforma di gestione, la dashboard utilizzata dai clienti per configurare e monitorare il servizio di filtraggio. Questo processo, unito a un'operazione di rebranding aziendale, mirava a modernizzare l'infrastruttura e l'esperienza utente, con un rilascio previsto per novembre 2025.

In questo scenario di transizione, è emersa una criticità tanto specifica quanto urgente. La piattaforma allora in uso, pur essendo efficace per la gestione delle policy di protezione, presentava una notevole lacuna funzionale: l'assenza di una modalità di consultazione dei dati in sola lettura (readonly). Gli utenti, in particolare gli

amministratori di rete e i responsabili IT, manifestavano la crescente necessità di poter analizzare i report e le statistiche di navigazione senza avere i permessi di modifica, per evitare alterazioni accidentali delle configurazioni di sicurezza.

Il progetto di tesi nasce per rispondere a questa precisa esigenza. Data l'impossibilità di attendere il rilascio della nuova piattaforma, si è optato per lo sviluppo di una soluzione tattica e mirata: un'applicazione web temporanea, concepita come "ponte" (bridge) tecnologico. Lo scopo primario di questa applicazione è fornire ai clienti un pannello di controllo readonly per le funzionalità standard di analisi dei dati, garantendo continuità operativa e soddisfacendo le richieste del mercato fino alla migrazione sulla nuova infrastruttura. Questo lavoro di tesi documenta pertanto non solo la realizzazione di un prodotto software, ma anche l'approccio ingegneristico adottato per sviluppare una soluzione efficace e affidabile in un contesto agile e con vincoli temporali definiti.

1.2 Obiettivi della Tesi

A partire dal contesto delineato, questo elaborato si pone obiettivi che trascendono la semplice descrizione di un prodotto software, per configurarsi come un'analisi approfondita delle metodologie di ingegneria del software applicate a un caso di studio reale. Il fine principale della tesi è, pertanto, analizzare e documentare come i paradigmi della Programmazione Orientata agli Oggetti (OOP) e le relative pratiche di progettazione, quali i design pattern, vengano impiegati per realizzare un'applicazione web complessa in un contesto industriale. Si intende dimostrare come tali principi, spesso affrontati in ambito teorico, rappresentino strumenti pratici e fondamentali per guidare le scelte architetturali e per garantire qualità essenziali come la manutenibilità, la scalabilità e la robustezza del software, specialmente quando si opera con vincoli di tempo e risorse.

Per supportare tale analisi, un primo passo fondamentale sarà la documentazione completa e dettagliata dell'architettura full-stack dell'applicazione. Verrà illustrato il flusso di dati e le interazioni tra il frontend, sviluppato in React con TypeScript, e il backend, basato su un modello di programmazione reattiva in Java. Questo esame non si limiterà a una descrizione statica, ma includerà un'analisi critica delle decisioni tecniche prese durante lo sviluppo, giustificando la scelta dello

stack tecnologico e approfondendo l'implementazione di componenti chiave come il sistema di autenticazione sicuro basato su token JWT e l'integrazione con le API esterne di FlashStart.

Il cuore analitico della tesi si concentrerà sull'applicazione pratica dei Design Pattern. Verranno identificati e discussi esempi concreti di pattern implementati nel codice sorgente, spiegando per ciascuno il problema che risolve e i benefici apportati in termini di flessibilità e organizzazione del codice. In questo modo, si intende creare un collegamento esplicito tra i concetti teorici dell'ingegneria del software e la loro implementazione pratica. Particolare attenzione sarà data a come le scelte di design abbiano promosso un'architettura dinamica e facilmente manutenibile; un requisito fondamentale dato che l'applicazione doveva interfacciarsi con servizi interni di FlashStart a loro volta in piena fase di reingegnerizzazione. Verrà quindi evidenziato come le decisioni di design non siano state casuali, ma guidate da principi consolidati volti a massimizzare l'efficienza e la qualità, nel pieno rispetto della natura strategica ma temporanea del progetto.

Capitolo 2

Background

2.1 Paradigmi per lo Sviluppo di Interfacce Utente Moderne

Questa sezione analizza l'architettura delle interfacce utente moderne, con l'obiettivo di giustificare la scelta di un'architettura Single Page Application (SPA) costruita con React e TypeScript.

2.1.1 L'Architettura Single Page Application (SPA)

Una Single Page Application (SPA) è un'applicazione web che interagisce con l'utente riscrivendo dinamicamente la pagina corrente, invece di ricaricare intere pagine nuove dal server. Questo approccio si realizza caricando le risorse necessarie in un'unica richiesta iniziale o dinamicamente secondo necessità. La scelta tra un'architettura SPA e una tradizionale Multi-Page Application (MPA) rappresenta un compromesso fondamentale nell'ingegneria del software web. Le MPA seguono un modello classico in cui ogni interazione scatena una richiesta completa al server, che risponde con una nuova pagina HTML. Le SPA presentano un tempo di caricamento iniziale più elevato, ma le interazioni successive sono estremamente rapide, poiché vengono scambiati solo dati tramite API. In termini di esperienza utente (UX), le SPA offrono un'esperienza percepita come più fluida e interattiva, motivo della loro adozione in contesti applicativi complessi come le dashboard.

Una delle principali debolezze delle SPA risiede nell'ottimizzazione per i motori di ricerca (SEO), poiché il contenuto viene renderizzato lato client e richiede tecniche aggiuntive come il Server-Side Rendering (SSR) per una corretta indicizzazione. La selezione di un'architettura SPA per una dashboard di monitoraggio, come quella oggetto di questa tesi, è giustificata dal fatto che l'obiettivo primario è un'esperienza utente reattiva per un utente autenticato, rendendo la SEO del tutto irrilevante.

2.1.2 Il Modello a Componenti e il Framework React

L'approccio architetturale basato su componenti (Component-Based Software Engineering, CBSE) ha lo scopo di sviluppare sistemi assemblando parti riutilizzabili e autonome. Il framework React fornisce un'implementazione pratica di questi principi, scomponendo le interfacce utente in una gerarchia di componenti modulari. Ogni componente incapsula la propria logica, il proprio stato e la propria presentazione, promuovendo una forte separazione delle responsabilità e un elevato grado di riusabilità. Una delle innovazioni paradigmatiche di React è stata la messa in discussione della "separazione delle tecnologie" (file HTML, CSS, JS separati) a favore di una "separazione delle responsabilità" a livello di componente, resa possibile da JSX, un'estensione sintattica di JavaScript che permette di scrivere un markup simile a HTML direttamente nel codice. In questo modo, la logica di rendering e la struttura di presentazione sono gestite come un'unica unità coesa.

2.1.3 Analisi delle Prestazioni del Virtual DOM

Un'innovazione chiave di React è il Virtual DOM (VDOM), una rappresentazione in memoria del Document Object Model (DOM) reale del browser. La manipolazione diretta del DOM è un'operazione computazionalmente costosa. React affronta questo problema aggiornando prima il VDOM, che è un oggetto leggero, e successivamente, tramite un processo di "riconciliazione", calcola il set minimo di modifiche da applicare al DOM reale in un unico processo batch [CN19]. Questo approccio offre benefici significativi per applicazioni con aggiornamenti frequenti della UI, poiché minimizza le operazioni sul DOM e fornisce un'utile astrazione per lo sviluppatore [CN19].

2.1.4 Tipizzazione Statica con TypeScript

Nei linguaggi a tipizzazione dinamica come JavaScript, i controlli sui tipi di dato avvengono a runtime, mentre nei linguaggi a tipizzazione statica questi controlli avvengono a compile-time, intercettando errori nelle fasi iniziali dello sviluppo [MHR+12]. TypeScript è un superset di JavaScript a tipizzazione statica che viene compilato in JavaScript puro [BAT14], con l'obiettivo di introdurne i benefici in applicazioni su larga scala. Studi empirici hanno fornito prove concrete a sostegno del suo utilizzo. Una ricerca ha dimostrato che i sistemi di tipi statici offrono un vantaggio significativo per la manutenibilità del software, fungendo da documentazione efficace e aiutando gli sviluppatori a comprendere più rapidamente codice non familiare [HKR+13]. Un'analisi su larga scala di progetti su GitHub ha ulteriormente corroborato questi risultati, rilevando che le applicazioni TypeScript mostrano una qualità e una comprensibilità del codice significativamente migliore rispetto a quelle in JavaScript puro [BM22]. È importante notare che il sistema di tipi di TypeScript è intenzionalmente non "sound" (insicuro) per progettazione, una scelta di compromesso per garantire la massima compatibilità con l'ecosistema JavaScript [BAT14]. L'adozione di TypeScript in questo progetto è quindi motivata dalla necessità di migliorare la qualità e la manutenibilità a lungo termine della codebase.

2.2 Architetture reattive e programmazione asincrona

Le architetture server tradizionali, come quelle basate su Java Servlet, si fondano sul modello di concorrenza **thread-per-request**. In questo paradigma, il server application assegna a ogni richiesta HTTP in ingresso un thread dedicato da un pool limitato. Questo thread gestisce l'intera logica della richiesta e, aspetto cruciale, rimane in stato di attesa bloccante durante le operazioni di I/O, come una query su un database o una chiamata a un servizio esterno. Sebbene questo modello sia concettualmente semplice, la sua efficienza si degrada rapidamente in scenari con alta concorrenza o alta latenza. Un numero elevato di richieste simultanee può portare alla saturazione del pool di thread, consumando ingenti quantità di

memoria (ogni thread ha un proprio stack) e aumentando il carico sulla CPU a causa del continuo "context switching". Le nuove richieste vengono messe in coda o respinte, e la latenza complessiva del sistema aumenta drasticamente [Vri21].

Per superare questi limiti, è stato introdotto il **modello di I/O non bloccante**, che costituisce il fondamento delle architetture reattive. In questo paradigma, un numero ridotto e fisso di thread, noto come **Event Loop**, gestisce un numero molto più elevato di connessioni concorrenti. Quando un'operazione di I/O viene avviata, il thread dell'Event Loop non attende il suo completamento; delega l'operazione al sistema operativo e registra una *callback* da eseguire quando il risultato sarà disponibile. Nel frattempo, il thread è libero di elaborare altri eventi per altre richieste. Questo approccio, ispirato a sistemi come Node.js e Nginx, permette di ottenere un'elevata efficienza nell'uso delle risorse e una scalabilità superiore, specialmente per cariche di lavoro I/O-bound, come dimostrato da numerosi studi comparativi [KS15, DK20].

Nel contesto dell'ecosistema Spring, questo paradigma è implementato dal modulo **Spring WebFlux**, che si fonda su **Project Reactor** per fornire un'API ricca e componibile per la gestione di flussi asincroni. Reactor introduce due tipi fondamentali: **Mono**, un *publisher* che rappresenta un flusso asincrono di 0 o 1 elemento (es. il risultato di una query che restituisce un singolo utente), e **Flux**, che rappresenta un flusso di 0 o N elementi (es. una lista di risultati). Attraverso questi tipi, è possibile costruire pipeline di elaborazione dati in modo dichiarativo e funzionale. Anziché scrivere codice imperativo (con cicli e costrutti condizionali), lo sviluppatore definisce una catena di operatori (es. .map() per trasformare, .filter() per selezionare, .flatMap() per operazioni asincrone nidificate) che descrivono la logica di business. Questa "ricetta" viene eseguita dal framework solo quando un client "sottoscrive" il flusso, promuovendo un codice più espressivo e disaccoppiato dalla gestione della concorrenza [Gut24].

Un principio fondamentale dei sistemi reattivi è che il paradigma non bloccante deve essere preservato **end-to-end**, dall'interfaccia di rete fino alla fonte dei dati. Utilizzare un web layer reattivo sarebbe inutile se il thread dell'Event Loop dovesse poi bloccarsi in attesa di una risposta dal database. La tradizionale API **JDBC (Java Database Connectivity)** è, per sua natura, bloccante. Per risolvere questa incompatibilità, è stata sviluppata la specifica **R2DBC (Reactive**

Relational Database Connectivity), che definisce un Service Provider Interface (SPI) per driver di database che supportano operazioni non bloccanti e basate su flussi. Per semplificare l'utilizzo di R2DBC, **Spring Data R2DBC** offre un'astrazione di alto livello, coerente con il resto dell'ecosistema Spring Data. Fornisce un modello di programmazione familiare, basato sul pattern Repository (es. `ReactiveCrudRepository`), la cui differenza cruciale è che i metodi non restituiscono entità o collezioni, ma *publishers* reattivi (Mono o Flux), integrando così l'accesso ai dati in modo trasparente all'interno della catena reattiva [Gut24].

I vantaggi di un'architettura reattiva correttamente implementata sono molteplici e significativi. Il più evidente è la scalabilità verticale e l'efficienza delle risorse: un'applicazione reattiva può gestire un numero molto più elevato di richieste concorrenti con un numero inferiore di thread, riducendo drasticamente il consumo di memoria e l'overhead della CPU. Questo si traduce in una maggiore responsività del sistema, che rimane reattivo e con bassa latenza anche sotto carichi pesanti, poiché i thread non vengono mai monopolizzati da operazioni lente. Un altro vantaggio cruciale è la resilienza. Gli errori in un flusso reattivo sono eventi di prima classe, gestiti all'interno della pipeline tramite operatori dedicati (es. .onErrorResume, .retry), permettendo di implementare logiche di fallback e di recupero dagli errori in modo più robusto e isolato. Infine, il protocollo Reactive Streams incorpora nativamente il concetto di contropressione (backpressure), un meccanismo di controllo del flusso con cui il consumatore di dati (Subscriber) segnala al produttore (Publisher) quanti elementi è in grado di processare. Questo previene che un produttore veloce possa sopraffare un consumatore più lento, garantendo la stabilità del sistema ed evitando errori di OutOfMemoryError.

2.3 Sicurezza nelle Applicazioni Web Distribuite

2.3.1 Autenticazione Stateless vs. Stateful

L'autenticazione stateful (basata su sessione) memorizza lo stato dell'utente sul server, rappresentando un collo di bottiglia per la scalabilità. L'autenticazione stateless (basata su token) supera questi limiti: il server non mantiene stato, ma emette un token auto-contenuto, come un JWT, che il client include in ogni

richiesta. Il server si limita a validare crittograficamente il token. Questo approccio è altamente scalabile, ma introduce la sfida della revoca dei token, poiché un token rimane valido fino alla sua scadenza naturale.

2.3.2 JSON Web Token (JWT): Specifica RFC 7519 e Analisi delle Vulnerabilità

Il JSON Web Token (JWT), definito nella RFC 7519, è lo standard de facto per l'implementazione dell'autenticazione stateless [JBS15]. È un formato compatto per rappresentare "claims" (asserzioni) tra due parti, composto da Header, Payload e Signature. La variante più comune, JSON Web Signature (JWS), garantisce l'integrità dei dati tramite firma digitale, ma non la confidenzialità, poiché Header e Payload sono codificati in Base64Url e pubblicamente leggibili. È quindi imperativo non memorizzare mai informazioni sensibili nel payload [RPMK24]. Vulnerabilità comuni includono l'attacco dell'algoritmo "none", l'uso di segreti deboli per la firma e la mancata verifica della firma stessa [OWA23].

2.3.3 Sinergia tra Paradigma Reattivo e Autenticazione Stateless

Il paradigma reattivo e l'autenticazione stateless tramite JWT sono un abbinamento architetturale quasi perfetto, poiché si fondano sullo stesso principio: l'assenza di stato condiviso e mutabile. Un flusso reattivo (una pipeline di 'Mono' o 'Flux') è intrinsecamente stateless; ogni richiesta viene processata come una sequenza di eventi indipendente. Se un'applicazione reattiva dovesse recuperare i dati dell'utente da un tradizionale archivio di sessioni, dovrebbe eseguire un'operazione di I/O che, se bloccante, vanificherebbe i benefici del modello non bloccante.

Qui entra in gioco la sinergia con i JWT. Un token JWT è auto-contenuto: trasporta al suo interno tutte le informazioni necessarie sull'utente (ID, ruoli, ecc.) in modo sicuro. Quando una richiesta arriva al server:

1. Un filtro di sicurezza reattivo (come quelli forniti da Spring Security) decodifica e valida il JWT all'inizio della catena di elaborazione, in modo asincrono.

2. Le informazioni sull'utente autenticato (il *Principal*) vengono estratte e inserite nel Contesto Reattivo (`Context`).

3. Questo `Context` è immutabile e viene propagato lungo l'intera pipeline reattiva, diventando accessibile a ogni operatore (es. `.map`, `.flatMap`) in modo non bloccante.

In questo modo, il contesto di sicurezza "fluisce" insieme ai dati, eliminando la necessità di qualsiasi chiamata I/O esterna per recuperare lo stato della sessione, preservando l'integrità del modello non bloccante end-to-end.

2.3.4 Gestione Avanzata dei Token: Rotazione e Revoca

Per mitigare il rischio della difficile revoca dei token, la best practice consiste nell'utilizzare access token a breve durata [Fla24]. Per non danneggiare l'esperienza utente, si adotta il pattern dei refresh token, credenziali a lunga durata usate per ottenere un nuovo access token. Per affrontare la sicurezza del potente refresh token, le più recenti best practice per OAuth 2.0 raccomandano la rotazione dei refresh token. Il meccanismo prevede che quando un client utilizza un refresh token (RT_A), il server emetta un nuovo access token e un nuovo refresh token (RT_B), invalidando contestualmente RT_A. Se un aggressore riutilizza RT_A, il server rileva il tentativo e deve invalidare l'intera famiglia di token discendenti, forzando una ri-autenticazione completa [Fla24].

2.4 Metodologie DevOps per l'Automazione del Ciclo di Vita del Software

2.4.1 Principi e Pratiche DevOps

DevOps è un movimento culturale e professionale che promuove la collaborazione tra sviluppatori (Dev) e operazioni IT (Ops) per accorciare il ciclo di vita dello sviluppo e fornire una distribuzione continua di alta qualità. Revisioni sistematiche della letteratura hanno identificato un insieme di pratiche tecniche chiave che abilitano la cultura DevOps: Continuous Integration (CI), Continuous Delivery/Deployment

(CD), Infrastructure as Code (IaC), Automated Testing e Continuous Monitoring [TPH+20].

2.4.2 Containerizzazione

La containerizzazione è un paradigma di virtualizzazione a livello di sistema operativo che consente di isolare le applicazioni in ambienti leggeri e portabili, noti come **container**. A differenza delle macchine virtuali (VM), che richiedono un intero sistema operativo guest e la virtualizzazione dell'hardware, i container condividono il kernel del sistema operativo host. Questa architettura produce un'efficienza notevolmente superiore: i container hanno un footprint di memoria e disco molto più ridotto, tempi di avvio nell'ordine dei secondi e un overhead prestazionale quasi nullo rispetto all'esecuzione nativa, specialmente per carichi di lavoro legati alla CPU e alla memoria [MV24]. Il principio fondamentale è l'incapsulamento di un'applicazione con tutte le sue dipendenze (come librerie, binari e file di configurazione) in un'unità atomica. Ciò garantisce la **portabilità** e la **consistenza** del comportamento dell'applicazione attraverso l'intero ciclo di vita dello sviluppo, eliminando la comune discrepanza tra ambienti di sviluppo, test e produzione, spesso riassunta nel problema "funziona sulla mia macchina" [Syr23].

Docker si è affermato come lo standard de facto per l'implementazione della tecnologia di containerizzazione, grazie a un ecosistema di strumenti che ne ha reso l'utilizzo accessibile e potente. Il processo di creazione di un container si basa su un' **immagine**, un template read-only che ne definisce lo stato. Le immagini sono costruite a partire da un **Dockerfile**, un file di testo che funge da manifesto dichiarativo, specificando una serie di istruzioni sequenziali. Una delle innovazioni chiave di Docker è il suo filesystem basato su **immagini stratificate** (layered images). Ogni istruzione in un Dockerfile crea un nuovo strato (layer) read-only che si sovrappone ai precedenti. Docker utilizza un meccanismo di **copy-on-write (CoW)**: quando un container viene avviato, viene aggiunto uno strato scrivibile sopra la pila di strati read-only dell'immagine. Qualsiasi modifica apportata dal container viene registrata in questo strato superiore. Questa architettura offre due vantaggi cruciali: efficienza nello storage, poiché gli strati comuni a più immagini vengono memorizzati una sola volta, e velocità nella distribuzione, dato che durante

un aggiornamento è necessario trasferire solo gli strati che sono stati modificati [C+19]. Le immagini vengono poi distribuite tramite un **Container Registry**, un repository centralizzato che funge da catalogo per la loro condivisione e il loro versionamento.

L'adozione della containerizzazione con Docker è diventata una pratica fondante delle metodologie DevOps e delle pipeline di **Integrazione e Distribuzione Continua (CI/CD)**. L'immagine containerizzata diventa l'artefatto immutabile che viene costruito una sola volta e promosso attraverso i vari stadi della pipeline (build, test, staging, produzione). Questo garantisce che l'unità software testata sia esattamente la stessa che viene eseguita in produzione, aumentando drasticamente l'affidabilità dei rilasci [SL24].

2.4.3 Integrazione e Distribuzione Continua (CI/CD)

La Continuous Integration (CI) è una pratica di sviluppo software che prevede l'integrazione frequente delle modifiche del codice in un repository centrale condiviso, seguita dall'esecuzione automatizzata di build e test. Questo approccio si contrappone al tradizionale modello di sviluppo "feature branch" a lunga durata, dove le modifiche vengono integrate raramente, spesso causando conflitti complessi e difficili da risolvere. La CI promuove invece commit piccoli e frequenti, tipicamente multiple volte al giorno, con l'obiettivo di identificare e risolvere rapidamente i problemi di integrazione [RAKS18]. Il processo di CI si articola in diverse fasi automatizzate. Quando uno sviluppatore effettua un commit, un CI server (come Jenkins, GitLab CI, o GitHub Actions) rileva automaticamente la modifica e avvia una build pipeline. Questa pipeline comprende tipicamente: il checkout del codice sorgente, la risoluzione delle dipendenze, la compilazione dell'applicazione, l'esecuzione di test unitari e di integrazione, l'analisi statica del codice per rilevare vulnerabilità di sicurezza o violazioni di standard di coding, e infine la generazione di artefatti deployabili (come immagini Docker). Se una qualsiasi di queste fasi fallisce, la pipeline si interrompe e gli sviluppatori ricevono un feedback immediato, permettendo una correzione rapida prima che il problema si propaghi [GdCZ19]. La Continuous Delivery (CD) estende il concetto di CI automatizzando anche il processo di rilascio, preparando ogni build che supera i test per il deployment in

produzione. Tuttavia, il rilascio effettivo rimane un'azione manuale, solitamente innescata da un'approvazione umana. Il Continuous Deployment rappresenta il livello più avanzato di automazione, dove ogni modifica che supera con successo l'intera pipeline viene automaticamente rilasciata in produzione senza intervento umano. Questo approccio richiede un'estrema fiducia nella suite di test e nei meccanisms di monitoraggio, ma permette di ottenere un feedback loop estremamente rapido dal mercato [TPH+20]. Una delle sfide tecniche più significative nell'implementazione di pipeline CI/CD efficaci è la gestione dei tempi di build. Build lente (superiori ai 10-15 minuti) compromettono il valore del feedback rapido, scoraggiando gli sviluppatori dall'effettuare commit frequenti e riducendo l'efficacia dell'intero processo. Strategie di ottimizzazione includono la parallelizzazione dei test, l'utilizzo di cache intelligenti per evitare la ricompilazione di componenti non modificati, e l'implementazione di test pyramids che privilegiano test unitari veloci rispetto a test di integrazione più lenti [GdCZ19]. L'integrazione con la containerizzazione rappresenta un'evoluzione naturale delle pipeline CI/CD. Gli artefatti prodotti non sono più semplici file binari o archivi, ma immagini Docker immutabili che incapsulano l'applicazione e tutte le sue dipendenze. Questo approccio elimina le discrepanze ambientali e garantisce che l'esatto software testato nella pipeline sia quello che viene eseguito in produzione. Inoltre, le immagini Docker possono essere taggate con metadati di versioning automatici (come commit hash, build number, timestamp), facilitando la tracciabilità e il rollback in caso di problemi.

2.4.4 Infrastructure as Code (IaC)

L'Infrastructure as Code (IaC) rappresenta un paradigma fondamentale nell'ingegneria delle infrastrutture moderne, che tratta l'infrastruttura IT come un artefatto software gestibile tramite codice sorgente. Invece di configurare manualmente server, reti, database e altri componenti infrastrutturali attraverso interfacce grafiche o comandi imperativi, l'IaC utilizza file di definizione dichiarativi (tipicamente in formato YAML, JSON, o linguaggi specifici come HCL per Terraform) per specificare lo stato desiderato dell'infrastruttura [TPH+20]. Il principio fondamentale dell'IaC è la separazione tra dichiarazione e implementazione. Lo sviluppatore o l'ingegnere DevOps definisce "cosa" vuole ottenere (ad esempio, "voglio un cluster

Kubernetes con 3 nodi, un load balancer e un database PostgreSQL"), mentre lo strumento di IaC si occupa del "come" raggiunger tale stato, traducendo la dichiarazione in una serie di API calls verso i provider cloud o i sistemi di gestione dell'infrastruttura. Questo approccio dichiarativo si contrappone al tradizionale approccio imperativo, dove è necessario specificare esplicitamente ogni passo di configurazione in sequenza. Un concetto chiave nell'IaC è l'idempotenza: l'esecuzione ripetuta della stessa definizione infrastrutturale deve produrre sempre lo stesso risultato, indipendentemente dal numero di esecuzioni. Questo è possibile grazie alla capacità degli strumenti IaC di calcolare il delta tra lo stato corrente dell'infrastruttura e quello desiderato, applicando solo le modifiche necessarie. Ad esempio, se una definizione specifica 5 istanze di un servizio ma ne esistono già 3, lo strumento creerà automaticamente solo le 2 istanze mancanti, senza toccare quelle esistenti. I vantaggi dell'IaC sono molteplici e sostanziali. La versionabilità permette di tracciare ogni modifica all'infrastruttura attraverso sistemi di version control come Git, abilitando pratiche come code review, branching strategies, e rollback sicuri. La riproducibilità garantisce che ambienti identici possano essere creati on-demand, eliminando le discrepanze tra sviluppo, testing e produzione che spesso causano il famigerato problema "funziona sulla mia macchina". La scalabilità consente di replicare facilmente configurazioni complesse su scala globale, mentre la disaster recovery diventa un processo automatizzato e testabile, poiché l'intera infrastruttura può essere ricreata da zero a partire dai file di definizione. Nel contesto delle applicazioni containerizzate, l'IaC assume forme specifiche e complementari. I Dockerfile rappresentano l'IaC a livello di runtime environment, definendo in modo dichiarativo come costruire l'immagine di un container. I file docker-compose.yml estendono questo concetto a livello di applicazione multi-container, specificando le relazioni, le reti e i volumi necessari. Per deployment su larga scala, i manifesti Kubernetes (scritti in YAML) definiscono l'orchestrazione di container attraverso concetti come Deployments, Services, ConfigMaps e Secrets. Un aspetto critico nell'implementazione di IaC è la gestione degli stati (state management). Strumenti come Terraform mantengono un state file che rappresenta la mappatura tra la dichiarazione logica dell'infrastruttura e le risorse fisiche create nei provider cloud. Questo state file deve essere condiviso tra team members e mantenuto in modo sicuro, tipicamente attraverso remote state backends che supportano locking

distribuito per evitare conflitti durante modifiche concorrenti. La sinergia tra IaC e pipeline CI/CD crea un ecosistema completamente automatizzato per la gestione dell'intero stack applicativo. Le modifiche ai file di definizione infrastrutturale vengono processate attraverso pipeline dedicate che includono validazione sintattica, testing dell'infrastruttura (attraverso strumenti come Terratest), e deployment graduale utilizzando strategie come blue-green deployment o canary releases. Questo approccio trasforma l'infrastruttura da un elemento statico e difficile da modificare a un componente agile e continuamente evolutivo, perfettamente allineato con i principi DevOps di integrazione e deployment continui.

Capitolo 3

Analisi

In questo capitolo si definiscono le fondamenta progettuali dell'applicazione web. Partendo dal contesto aziendale e dalle motivazioni esposte nell'Introduzione, verranno delineati in modo formale i requisiti che la soluzione software dovrà soddisfare. L'analisi si articola nella definizione delle funzionalità attese (requisiti funzionali), dei vincoli qualitativi e operativi (requisiti non funzionali) e del panorama di sistemi esterni con cui l'applicazione dovrà necessariamente interagire. Questo capitolo ha lo scopo di definire il perimetro e gli obiettivi del sistema, fungendo da guida per le successive fasi di progettazione e implementazione.

3.1 Analisi dei Requisiti

La definizione dei requisiti costituisce il primo passo del processo ingegneristico, traducendo le esigenze degli stakeholder in specifiche precise. Tali requisiti vengono classificati in funzionali, che descrivono il comportamento del sistema, e non funzionali, che ne specificano le proprietà e i vincoli.

3.1.1 Requisiti Funzionali

I requisiti funzionali descrivono le capacità che l'applicazione dovrà offrire ai suoi utenti per risolvere il problema di business identificato. Il sistema dovrà quindi essere in grado di:

- **RF1: Fornire un accesso autenticato.** Il sistema dovrà garantire che solo gli utenti autorizzati possano accedere ai dati. Sarà necessario un meccanismo di login basato su credenziali (e-mail e password) per proteggere l'accesso alle informazioni.

- **RF2: Presentare una dashboard di riepilogo.** Dopo l'autenticazione, l'applicazione dovrà visualizzare una pagina principale che offra una sintesi immediata dello stato del servizio, presentando statistiche aggregate sul traffico DNS e sulle minacce rilevate in un determinato intervallo temporale.

- **RF3: Permettere la consultazione delle policy di protezione.** L'utente dovrà poter navigare e ispezionare le configurazioni di protezione attive associate ai propri profili. Ciò include la visualizzazione delle categorie di contenuti filtrate, delle regole di geo-blocking e delle liste di eccezioni (blacklist e whitelist).

- **RF4: Mostrare le configurazioni di rete.** Sarà necessario fornire una sezione dedicata alla visualizzazione delle reti associate all'account del cliente, distinguendo tra indirizzi IP statici, utenze dinamiche e connessioni sicure tramite protocolli crittografati (DoH/DoT).

- **RF5: Offrire uno strumento di analisi per domini specifici.** L'applicazione dovrà includere una funzionalità di "lookup" che permetta all'utente di interrogare lo stato di un singolo dominio per verificare come verrebbe gestito dalle policy di ogni profilo (se bloccato o permesso) e per quale motivo.

- **RF6: Gestire il contesto multi-profilo e multi-cliente.** Poiché un utente può gestire più clienti e ciascun cliente può avere più profili di protezione, l'interfaccia dovrà permettere una facile selezione del contesto (cliente e profilo) per cui si desidera visualizzare i dati.

3.1.2 Requisiti Non Funzionali

I requisiti non funzionali impongono vincoli sulla qualità e sulle modalità operative del sistema, influenzando profondamente le scelte architetturali.

- **RNF1: Modalità di Sola Lettura.** Si tratta del requisito non funzionale più stringente: l'applicazione dovrà operare esclusivamente in modalità di consultazione. Nessuna funzionalità di modifica, creazione o cancellazione dei dati e delle configurazioni dovrà essere progettata o implementata.

- **RNF2: Gestione delle Operazioni di I/O Concorrenti.**L'applicazione dovrà gestire un numero significativo di operazioni di I/O concorrenti, dovute alle chiamate verso i servizi API esterni. Pertanto, l'architettura del backend dovrà essere progettata per gestire tali operazioni in modo asincrono e non bloccante, al fine di evitare la saturazione dei thread e ottimizzare l'uso delle risorse di sistema.

- **RNF3: Sicurezza delle Sessioni Utente.** Il meccanismo di autenticazione dovrà essere di tipo stateless, per favorire la scalabilità, ma al contempo garantire un'elevata sicurezza. Per mitigare i rischi associati a token di lunga durata, si dovrà studiare e implementare una strategia di rotazione dei token, in particolare per i refresh token utilizzati per il rinnovo delle sessioni.

- **RNF4: Portabilità e Standardizzazione dell'Ambiente.** La soluzione software dovrà essere pacchettizzata in modo da garantire la coerenza e la riproducibilità dell'ambiente operativo. L'adozione della tecnologia di containerizzazione, come Docker, è identificata come la scelta strategica per raggiungere questo obiettivo.

- **RNF5: Vincoli di Progetto.** Lo sviluppo del sistema è vincolato da una precisa finestra temporale, essendo una soluzione "ponte" strategica. Questo requisito impone l'adozione di un approccio agile, favorendo scelte tecnologiche e architetturali che bilancino la velocità di realizzazione con la robustezza e la manutenibilità necessarie per un prodotto di livello industriale.

- **RNF6: Conformità allo Stack Tecnologico Aziendale.** Per garantire continuità, manutenibilità e integrazione con le competenze e le infrastrutture esistenti in FlashStart, lo sviluppo del progetto è vincolato all'utilizzo di uno stack tecnologico predefinito. Nello specifico, i requisiti impongono l'uso di React con TypeScript per lo sviluppo dell'interfaccia utente, Nginx come web

server per il servizio di frontend e Docker per la containerizzazione dell'intera applicazione.

3.2 Analisi del Contesto e delle Integrazioni

L'applicazione oggetto di studio non è un sistema autocontenuto, ma deve operare all'interno dell'ecosistema tecnologico di FlashStart per adempiere ai suoi requisiti funzionali. In particolare, per recuperare i dati necessari alla visualizzazione (come report, policy di protezione e configurazioni di rete), l'applicazione dovrà interfacciarsi con dei servizi API preesistenti.

Un'analisi dell'infrastruttura esistente ha identificato due distinti endpoint di servizio che espongono le informazioni richieste:

- **Endpoint API HQ**: Un servizio specializzato che fornisce funzionalità di diagnostica e amministrazione. Ai fini di questo progetto, l'endpoint di maggior interesse è quello relativo al "lookup" di un dominio, che ne restituisce lo stato di filtraggio. Per comunicare con questo servizio, le richieste HTTP devono essere autenticate tramite il meccanismo di Basic Auth.
- **Endpoint API Principale (fsflt)**: Costituisce la fonte dati primaria per la maggior parte delle funzionalità del pannello. Da questo servizio è possibile recuperare tutte le informazioni operative, tra cui le configurazioni dei profili, le liste di protezione, i dati statistici per i report e l'elenco delle reti dei clienti. Le richieste a questo endpoint richiedono un'autenticazione basata su una chiave API (API key) univoca, associata all'utente che effettua la chiamata.

La necessità di interagire con questi due sistemi eterogenei, ciascuno con un proprio meccanismo di autenticazione, rappresenta un vincolo tecnico significativo. La soluzione software dovrà essere in grado di gestire questa complessità, orchestrando le chiamate verso entrambi gli endpoint per aggregare e presentare i dati all'utente in modo trasparente.

Capitolo 4

Design

Questo capitolo illustra la progettazione architetturale e di dettaglio dell'applicazione web, rispondendo ai requisiti funzionali e non funzionali definiti nel capitolo di Analisi. Verranno descritte le scelte strategiche relative alla struttura del software, le interazioni tra i componenti e i design pattern della programmazione orientata agli oggetti impiegati per garantire un'architettura robusta, manutenibile e scalabile.

4.1 Progettazione dell'Architettura di Alto Livello

Per rispondere ai requisiti di portabilità e riproducibilità degli ambienti (RNF4), il sistema è stato progettato secondo un'architettura a servizi. Questo approccio favorisce una netta separazione delle responsabilità (Separation of Concerns), disaccoppiando i componenti logici principali, semplificandone lo sviluppo e garantendo la coerenza del sistema nelle diverse fasi del suo ciclo di vita.

La struttura logica del sistema è decomposta in tre componenti principali:

Componente di Presentazione (Frontend) Questo componente ha la responsabilità di gestire l'interfaccia utente. È disegnato come una Single Page Application (SPA) servita da un web server. Il suo unico scopo è renderizzare le

viste, gestire le interazioni con l'utente e comunicare con il componente applicativo per il recupero e la visualizzazione dei dati.

Componente Applicativo (Backend) Rappresenta il nucleo logico del sistema e agisce come unico punto di contatto per il componente di presentazione. Il suo design prevede una duplice responsabilità:

1. Gestire la logica di business interna, come l'autenticazione degli utenti e la gestione delle sessioni.

2. Fungere da intermediario (secondo il pattern Architetturale API Gateway) verso i servizi esterni, astraendo la loro complessità dal frontend.

Componente di Persistenza Dati (Database) Questo componente è dedicato alla memorizzazione e al recupero dei dati necessari al funzionamento intrinseco dell'applicazione. In accordo con il requisito di sola lettura (RNF1) per i dati di business, il suo perimetro è strettamente limitato alla persistenza dei dati di sessione e delle anagrafiche utente.

Il flusso di interazione tra questi componenti è lineare e disaccoppiato. Il client utente interagisce esclusivamente con il Componente di Presentazione. Le richieste di dati vengono inoltrate da quest'ultimo al Componente Applicativo, che si occupa di elaborarle, interagendo a sua volta con il Componente di Persistenza per le operazioni relative all'autenticazione, o con i sistemi esterni (come descritto in Sezione 3.2) per i dati di business.

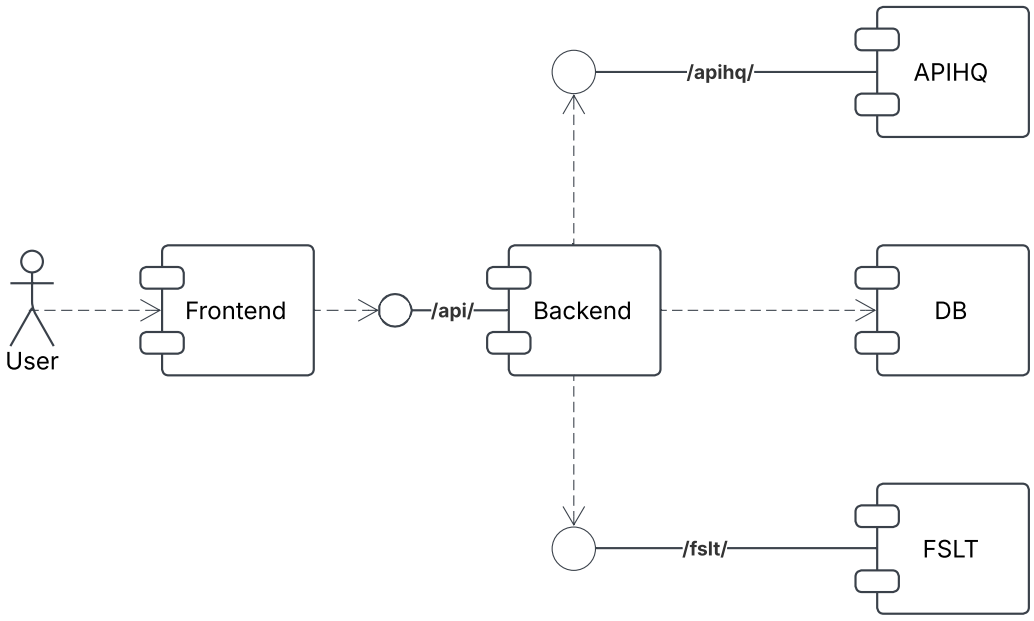

Figura 4.1: Diagramma dell'architettura logica di sistema, che illustra i componenti principali e i loro flussi di interazione.

Questa architettura a tre livelli logici permette di isolare le diverse responsabilità, migliorando la coesione di ciascun componente e riducendo l'accoppiamento tra di essi, a totale beneficio della manutenibilità e della testabilità del sistema.

4.2 Design del Backend

Il componente backend rappresenta il nucleo logico dell'intera applicazione. La sua progettazione è stata affrontata con un approccio orientato agli oggetti, con l'obiettivo di creare un sistema modulare, poco accoppiato e facilmente estensibile. Per raggiungere tale scopo, sono stati impiegati diversi design pattern fondamentali, scelti per risolvere specifiche problematiche emerse durante la fase di analisi.

4.2.1 Pattern Factory Method per la Creazione di Filtri Gateway

Problema di Design Dall'analisi delle integrazioni (Sezione 3.2) è emersa la necessità di arricchire le richieste inoltrate ai due diversi endpoint esterni (API HQ e API Principale) con meccanismi di autenticazione differenti (Basic Auth per uno, API Key per l'altro). La sfida di design consisteva nel trovare un modo flessibile ed estensibile per creare e applicare queste logiche di modifica delle richieste in modo dinamico e disaccoppiato dalla configurazione delle rotte.

Soluzione di Design Per risolvere questo problema, è stato adottato il pattern creazionale **Factory Method**. Il design prevede la definizione di una interfaccia o classe base astratta, la `GatewayFilterFactory`, che dichiara un "metodo fabbrica" (*factory method*) per la creazione di oggetti di tipo `GatewayFilter`. Un `GatewayFilter` è un oggetto la cui responsabilità è intercettare e modificare una richiesta HTTP.

Il design si completa con la creazione di due classi "fabbrica" concrete:

- `ApiHqAuthGatewayFilterFactory`: Una fabbrica concreta che implementa il factory method per produrre un'istanza di `GatewayFilter` specializzata nell'aggiungere l'header di autenticazione Basic Auth, necessario per le chiamate verso l'endpoint API HQ.

- `FsfltApiKeyGatewayFilterFactory`: Un'altra fabbrica concreta il cui factory method produce un'istanza di `GatewayFilter` che si occupa di recuperare la chiave API dell'utente autenticato e di inserirla in un header specifico per le chiamate verso l'API Principale.

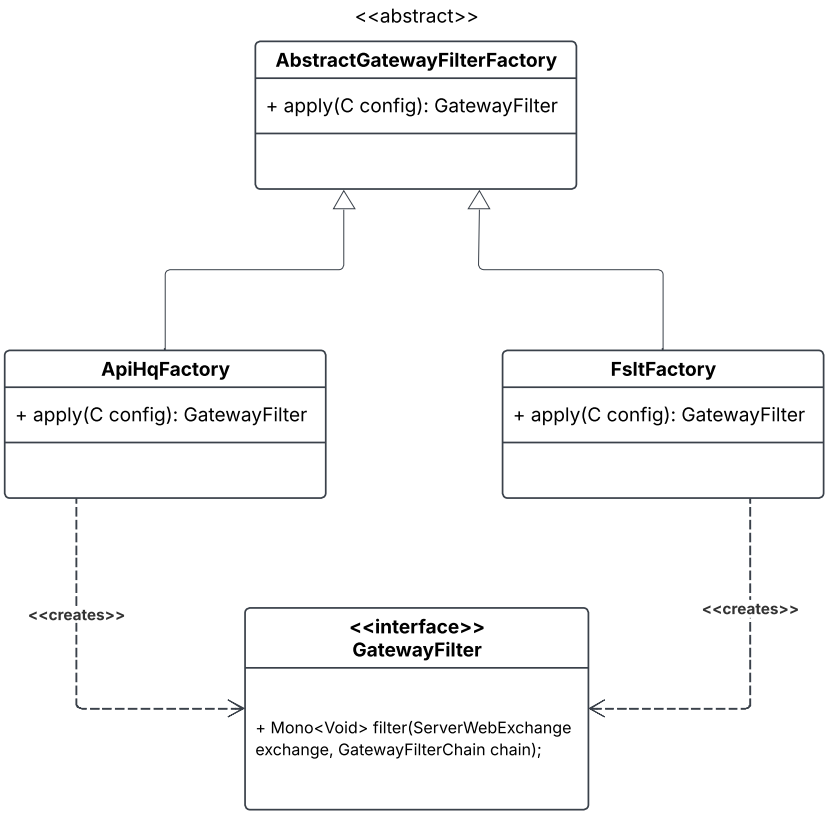

Figura 4.2: Diagramma delle classi che illustra l'applicazione del pattern Factory Method per la creazione dei filtri del gateway.

Questo approccio delega la responsabilità dell'istanziazione dei filtri alle sottoclassi, permettendo al sistema principale di configurazione delle rotte di operare a un alto livello di astrazione, senza conoscere i dettagli concreti di ciascun filtro. Ciò aumenta la modularità e semplifica l'eventuale aggiunta di nuovi filtri in futuro.

4.2.2 Pattern Singleton per la Gestione dei Servizi

Problema di Design Molti componenti del backend, come i servizi per la gestione della logica di business (es. autenticazione, generazione JWT) o i repository per l'accesso ai dati, sono intrinsecamente *stateless* (privi di stato di istanza). La creazione di più istanze di tali oggetti sarebbe inefficiente e potrebbe portare a

comportamenti anomali, rendendo necessaria una strategia per garantire che esista una e una sola istanza di questi componenti per tutta l'applicazione.

Soluzione di Design Per questo requisito, il design si affida all'applicazione del pattern **Singleton**. Piuttosto che implementare manualmente il pattern in ogni classe, si è deciso di delegarne la gestione a un container *Inversion of Control* (IoC), un componente fondamentale del framework scelto. Il design prevede che i componenti di servizio (`AuthenticationService`, `JwtService`) e di accesso ai dati (`UserRepository`) vengano definiti come "bean" gestiti dal container. Sarà responsabilità del container stesso garantire che per ciascuno di questi bean venga creata una sola istanza (ambito singleton) e che questa venga condivisa e "iniettata" in tutti gli altri componenti che ne dichiarano una dipendenza. Questo approccio, oltre a risolvere il problema dell'istanza unica, promuove un basso accoppiamento e semplifica la gestione delle dipendenze del sistema.

4.2.3 Pattern Facade per l'Autenticazione e la Gestione delle Sessioni

Problema di Design Il processo di autenticazione e gestione delle sessioni utente, per sua natura, è un'operazione complessa che richiede la coordinazione di molteplici componenti. Un client che volesse eseguire l'autenticazione dovrebbe interagire con un gestore degli utenti, un componente per la validazione delle credenziali, un servizio per la creazione dei token JWT e un repository per la gestione dei refresh token nel database. Esporre questa complessa rete di collaborazioni al client (in questo caso, il layer dei Controller) creerebbe un forte accoppiamento e renderebbe il codice difficile da comprendere, utilizzare e manutenere.

Soluzione di Design Per nascondere questa complessità e fornire un punto di accesso unificato e semplice, è stato applicato il pattern strutturale **Facade**. Il design prevede la creazione di una classe `AuthenticationService` che agisce, per l'appunto, da "facciata" per l'intero sottosistema di autenticazione.

Questa classe espone un set di metodi ad alto livello, come `autenticaUtente(...)` o `ruotaRefreshToken(...)`. Al suo interno, la facciata orchestra le chiamate ai

vari componenti del sottosistema (il repository degli utenti, il gestore dei token, il validatore delle password, etc.), ma nasconde completamente questi dettagli di interazione al chiamante. In questo modo, il client (il `AuthController`) dipende unicamente dall'interfaccia semplificata della Facade, risultando completamente disaccoppiato dalla logica interna del sottosistema. Questo non solo semplifica il codice del client, ma permette anche di modificare e far evolvere il sottosistema di autenticazione in modo indipendente, senza impattare il resto dell'applicazione.

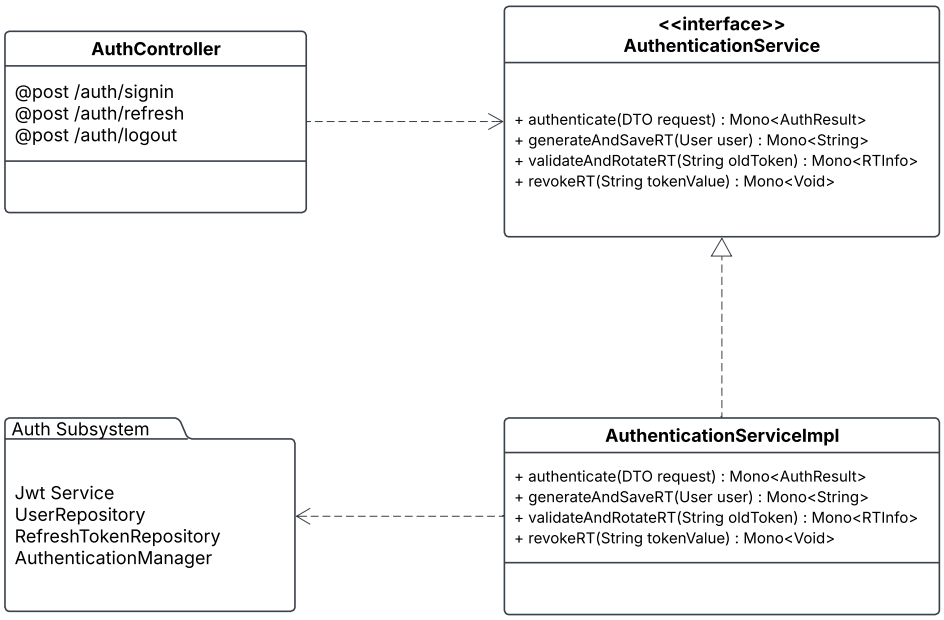

Figura 4.3: Diagramma delle classi che mostra come `AuthenticationService` agisca da Facade, semplificando l'accesso al sottosistema di autenticazione.

4.2.4 Pattern Data Transfer Object (DTO) per lo Scambio di Dati

Problema di Design La comunicazione tra i diversi layer dell'applicazione (es. tra Controller e Service) e, soprattutto, tra il backend e il client frontend, richiede uno scambio di dati strutturati. Utilizzare direttamente gli oggetti del dominio (le "entità" che rappresentano i dati nel database, come `User` o `RefreshToken`) per

questo scopo è una pratica sconsigliata. Esporre il modello di dominio interno potrebbe rivelare dettagli implementativi, creare dipendenze indesiderate e introdurre vulnerabilità di sicurezza, oltre a rendere l'API rigida e difficile da far evolvere.

Soluzione di Design Per risolvere questo problema di disaccoppiamento e sicurezza, è stato adottato il pattern **Data Transfer Object (DTO)**. Il design prevede la creazione di un insieme di classi semplici, il cui unico scopo è quello di fungere da contenitori di dati (data carrier) per le informazioni scambiate attraverso i confini dell'API.

Per esempio, per una richiesta di autenticazione, il client non invia un oggetto di dominio, ma un `AuthRequestDTO` contenente solo i campi strettamente necessari (email e password). Analogamente, la risposta del backend non sarà l'oggetto `User` completo, ma un `AuthResponseDTO` contenente solo il token di accesso e le informazioni sulla sua scadenza.

Questo approccio offre molteplici vantaggi:

- **Disaccoppiamento**: Il modello dati dell'API (i DTO) è indipendente dal modello dati di persistenza (le entità). È possibile modificare la struttura del database senza impattare i client dell'API.

- **Sicurezza**: Vengono esposti solo i dati strettamente necessari, nascondendo informazioni sensibili (come le password hashate o altri dati interni dell'entità `User`).

- **Ottimizzazione**: I DTO possono essere modellati per aggregare dati provenienti da più entità, riducendo il numero di chiamate necessarie al client per ottenere le informazioni di cui ha bisogno.

L'uso dei DTO definisce un "contratto" chiaro e stabile per le API, fondamentale per un'architettura a servizi in cui frontend e backend evolvono in modo indipendente.

4.3 Design del Frontend e Principi OOP Applicati

Sebbene il paradigma dominante nello sviluppo con React sia funzionale/dichiarativo, i principi fondamentali della programmazione orientata agli oggetti – come l'incapsulamento, la composizione e la separazione delle responsabilità – sono stati la guida per la progettazione del frontend. L'obiettivo era creare un'architettura a componenti che fosse non solo reattiva e performante, ma anche logicamente strutturata e scalabile.

4.3.1 Architettura a Componenti come Composizione di Oggetti

Problema di Design La costruzione di un'interfaccia utente complessa e interattiva come quella richiesta rischia di portare a un codice monolitico, difficile da comprendere, testare e far evolvere. Era necessario un approccio che permettesse di gestire la complessità attraverso la decomposizione.

Soluzione di Design Il design del frontend si basa su un'architettura a componenti, che è l'analogo del principio di **composizione** nella programmazione orientata agli oggetti. L'interfaccia utente è stata scomposta in una gerarchia di componenti React, ciascuno dei quali può essere visto come un "oggetto" con le proprie proprietà (*props*), il proprio stato interno (*state*) e il proprio comportamento (metodi e gestori di eventi).

Questo approccio ha permesso di distinguere due tipologie di componenti:

- **Componenti "Container" (o Smart)**: Hanno la responsabilità di gestire la logica e lo stato. Si occupano di recuperare i dati (interagendo con i servizi API) e di passarli ai componenti sottostanti.

- **Componenti "Presentazionali" (o Dumb)**: La loro unica responsabilità è quella di visualizzare i dati ricevuti tramite *props* e di notificare ai componenti genitori eventuali interazioni dell'utente. Sono altamente riutilizzabili.

4.3.2 Pattern Strategy per Componenti Configurabili

Problema di Design La dashboard (pagina Home) richiede la visualizzazione di molteplici grafici. Sebbene ogni grafico abbia una logica di base simile (titolo, selettore temporale, recupero dati, gestione del caricamento), ognuno di essi si differenzia per l'endpoint da interrogare, la trasformazione da applicare ai dati e le opzioni di visualizzazione (es. grafico a barre vs. grafico a torta). Creare un componente specifico per ogni grafico avrebbe comportato una notevole duplicazione di codice.

Soluzione di Design Per risolvere questo problema in modo elegante, è stato applicato un design che ricalca il **pattern Strategy**. È stato progettato un singolo componente generico, `GenericChart`, che agisce come "contesto". Questo componente non contiene la logica specifica di nessun grafico, ma è progettato per ricevere un oggetto `config` tramite le sue *props*.

Questo oggetto di configurazione definisce la "strategia" completa per un particolare grafico: l'endpoint da chiamare, i parametri della query, la funzione per trasformare la risposta dell'API in un formato compatibile con la libreria di grafici, e le opzioni di rendering. Le diverse strategie sono definite centralmente nel file `chartConfigs.ts`. In questo modo, per renderizzare un nuovo tipo di grafico è sufficiente definire una nuova strategia, senza modificare il componente `GenericChart`, promuovendo i principi di Open/Closed e di riuso del software.

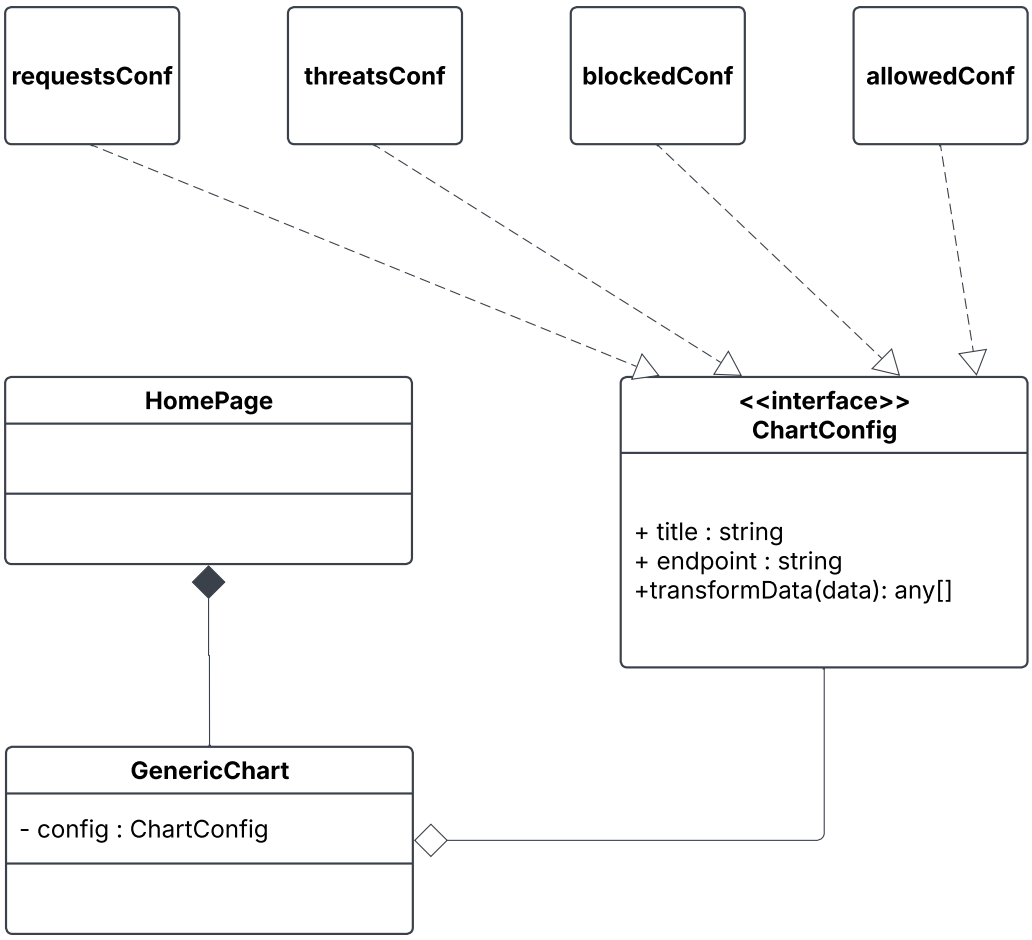

Figura 4.4: Diagramma che illustra come il componente `GenericChart` utilizzi diversi oggetti `ChartConfig` (strategie) per renderizzare grafici differenti.

4.3.3 Gestione dello Stato Globale con il Pattern Observer

Problema di Design In un'applicazione complessa, numerosi componenti distribuiti nell'albero della UI necessitano di accedere e reagire a cambiamenti di uno stesso stato condiviso. Esempi tipici in questo progetto sono lo stato di autenticazione dell'utente o il codice del cliente e del profilo attualmente selezionati. Far transitare questi dati attraverso l'intera gerarchia di componenti tramite le

props (una pratica nota come "prop drilling") è inefficiente, crea un forte accoppiamento tra i componenti e rende l'applicazione estremamente rigida e difficile da manutenere.

Soluzione di Design Per risolvere questo problema di gestione dello stato globale in modo pulito e scalabile, il design adotta un'architettura che ricalca il pattern comportamentale **Observer**. La soluzione prevede la creazione di "soggetti" osservabili centralizzati (*subjects*), denominati `AuthProvider` e `CustomerProvider`, che detengono lo stato condiviso.

I componenti che necessitano di accedere a questo stato agiscono come "osservatori" (*observers*), "sottoscrivendo" l'interesse verso le notifiche di cambiamento tramite un meccanismo di accesso al contesto (i custom hook `useAuth` e `useCustomer`). Quando lo stato in un *provider* cambia (ad esempio, l'utente effettua il logout o seleziona un nuovo cliente), il *provider* notifica automaticamente tutti i componenti sottoscrittori, che possono così reagire e aggiornare la propria vista di conseguenza. Questo design disaccoppia efficacemente i componenti, che non necessitano di una comunicazione diretta, ma reagiscono in modo indipendente ai cambiamenti di uno stato centralizzato.

4.3.4 Centralizzazione delle Chiamate API (Singleton e Decorator)

Problema di Design Le chiamate asincrone verso il backend sono una responsabilità trasversale a molti componenti. Ogni chiamata deve includere il token di autenticazione e deve gestire correttamente la possibile scadenza di tale token, avviando un flusso di rinnovo. Implementare questa logica in ogni singolo componente che recupera dati porterebbe a una massiccia duplicazione di codice e a una manutenibilità quasi nulla.

Soluzione di Design Il design affronta questo problema attraverso la creazione di un servizio API centralizzato. Viene definita una singola istanza di un client HTTP, configurata per l'intera applicazione, che agisce come un **Singleton**. Questo

garantisce che tutte le impostazioni di comunicazione (come l'URL di base o i timeout) siano coerenti.

Su questa istanza unica, viene applicato un pattern riconducibile al **Decorator**, utilizzando il meccanismo degli "intercettori".

- **Intercettore di Richiesta**: "Decora" ogni richiesta in uscita, aggiungendo dinamicamente l'header `Authorization` con il token JWT dell'utente. I componenti che effettuano la chiamata non devono preoccuparsi di questo dettaglio.
- **Intercettore di Risposta**: "Decora" la gestione degli errori, intercettando specificamente le risposte di tipo `401 Unauthorized` (token scaduto). In questo caso, l'intercettore gestisce in modo trasparente l'intero flusso di rinnovo del token, mettendo in pausa la richiesta originale, ottenendo un nuovo token e, infine, ripetendo la richiesta fallita con le nuove credenziali.

Questo design a strati astrae completamente la complessità della comunicazione autenticata, lasciando ai componenti della UI la sola responsabilità di richiedere i dati di cui hanno bisogno.

Capitolo 5

Implementazione

Questo capitolo illustra i dettagli implementativi salienti del progetto, con l'obiettivo di dimostrare come i concetti e i pattern architetturali discussi nel capitolo di `Design` (Capitolo 4) siano stati realizzati concretamente nel codice sorgente.

5.1 Dettagli Implementativi del Backend

Il componente backend è stato sviluppato utilizzando Java 17 e il framework Spring Boot 3, con un focus sul paradigma reattivo offerto da Spring WebFlux. Di seguito vengono analizzate le implementazioni dei principali pattern di design adottati.

5.1.1 Implementazione del Pattern Factory Method nei Filtri Gateway

Come descritto nella sezione di design 4.2.1, il pattern Factory Method è stato utilizzato per creare filtri dinamici nel gateway.

Realizzazione della Factory Concreta La classe `ApiHqAuthGatewayFilter Factory` rappresenta una delle fabbriche concrete. L'annotazione `@Component` la registra come bean di Spring, rendendola disponibile per l'iniezione delle dipendenze. Il metodo `apply`, che è l'effettivo "factory method", restituisce una lambda expression che implementa l'interfaccia funzionale `GatewayFilter`. Questa lambda

incapsula la logica per aggiungere l'header di autenticazione `Basic Auth` a ogni richiesta.

Consumo della Factory nella Configurazione delle Rotte La factory viene poi utilizzata in modo dichiarativo nella configurazione delle rotte del gateway, come mostrato in `GatewayConfig.java`. Il framework invoca il metodo `apply` al momento opportuno per ottenere l'istanza del filtro da applicare alla rotta specificata.

5.1.2 Implementazione del Pattern Facade nel Servizio di Autenticazione

Il pattern Facade, discusso nella sezione 4.2.3, è stato implementato nella classe `AuthenticationService` per semplificare il complesso sottosistema di gestione delle sessioni.

Il Metodo `authenticate` Il metodo pubblico `authenticate` espone un'operazione di business semplice, ma al suo interno orchestra la collaborazione di molteplici componenti: invoca il `ReactiveAuthenticationManager` di Spring Security per validare le credenziali e, in caso di successo, chiama un metodo privato per generare e salvare il refresh token, nascondendo questa complessità al suo client, l' `AuthController`.

5.1.3 Implementazione dei Data Transfer Objects (DTO)

Per realizzare lo scambio di dati disaccoppiato descritto nella sezione 4.2.4, sono state create delle semplici classi POJO (Plain Old Java Object). Le classi `AuthRequestDTO.java` e `AuthResponseDTO.java` definiscono il "contratto" dati per l'API di autenticazione. Il loro utilizzo è evidente nelle firme dei metodi del `AuthController`, dove vengono usate per mappare il corpo delle richieste e delle risposte HTTP.

5.2 Dettagli Implementativi del Frontend

Il frontend è stato realizzato con React 18 e TypeScript, sfruttando le funzionalità degli hook per la gestione dello stato e del ciclo di vita dei componenti.

5.2.1 Implementazione del Pattern Strategy per i Grafici

Come progettato nella sezione 4.3.2, il pattern Strategy permette di avere un componente `GenericChart` altamente riutilizzabile.

Definizione delle Strategie Le strategie concrete sono definite come oggetti di configurazione nel file `chartConfigs.ts`. Ogni oggetto implementa l'interfaccia `ChartConfig`, specificando l'endpoint, la funzione per i parametri e la funzione per la trasformazione dei dati.

Utilizzo del Componente Contestuale La pagina `Home` istanzia poi molteplici volte il componente `GenericChart`, passando a ciascuno una diversa strategia tramite la prop `config`. Questo dimostra come lo stesso componente possa esibire comportamenti radicalmente diversi in base alla strategia iniettata.

5.3 Implementazione del Flusso di Sicurezza End-to-End

Questa sezione descrive la realizzazione pratica del flusso di sicurezza, traducendo il design concettuale in componenti software specifici sia nel backend che nel frontend. Il sistema implementa un'autenticazione stateless basata su JSON Web Tokens (JWT) con un meccanismo di rotazione per i refresh token, in accordo con il requisito RNF3.

5.3.1 Implementazione nel Backend

La logica principale di generazione, validazione e gestione dei token è interamente confinata nel backend, sviluppato con Spring Boot.

Generazione dei Token e Gestione dei Cookie Al momento del login, l' `AuthController` riceve le credenziali e le delega all' `AuthenticationService`. Questo servizio, agendo come **Facade**, orchestra la validazione e, in caso di successo, invoca un `JwtService` dedicato. Questa classe utilizza la libreria `io.jsonwebtoken.jjwt` per creare e firmare i token.

- L'**Access Token** viene generato come un JWT firmato, contenente i *claims* dell'utente (username e ruoli) e una scadenza breve di 15 minuti.
- Il **Refresh Token** viene generato come stringa crittograficamente sicura e con una scadenza lunga (7 giorni). L'hash del token viene salvato nel database PostgreSQL tramite il `RefreshTokenRepository` per la validazione futura.

L' `AuthController` si occupa poi di inviare i token al client. L'Access Token viene restituito nel corpo della risposta JSON, mentre il Refresh Token viene inserito in un cookie. Per questa operazione si utilizza la classe helper `ResponseCookie` di Spring, che permette di configurare in modo dichiarativo gli attributi di sicurezza `HttpOnly`, `Secure`, `Path` e `Max-Age`, garantendo che il refresh token sia protetto da accessi via JavaScript (XSS) e trasmesso solo su connessioni HTTPS.

Implementazione della Rotazione del Token Il cuore della sicurezza del sistema è implementato nel metodo `validateAndRotateRefreshToken` all'interno dell' `AuthenticationService`. Quando l'endpoint `/auth/refresh` viene chiamato, questo metodo esegue la logica di rotazione. L'intero processo è annotato con `@Transactional` di Spring, per assicurare che la ricerca, la cancellazione del vecchio token e il salvataggio del nuovo avvengano come una transazione atomica sul database, prevenendo condizioni di gara (*race conditions*) e garantendo la consistenza dei dati. Se il token fornito è valido, viene immediatamente invalidato e sostituito, rendendolo a tutti gli effetti un token monouso per prevenire attacchi di tipo *replay*.

5.3.2 Implementazione nel Frontend

Il frontend React è responsabile della gestione dell'Access Token e dell'orchestrazione trasparente del processo di rinnovo.

Memorizzazione dell'Access Token e Gestione dello Stato In accordo con le best practice di sicurezza, l'Access Token non viene mai memorizzato nel `localStorage` o `sessionStorage`. Viene invece mantenuto esclusivamente nello stato in memoria dell'applicazione, gestito tramite un **React Context** (`AuthContext`). Questo minimizza l'esposizione del token a vulnerabilità di tipo Cross-Site Scripting. I componenti che effettuano chiamate API protette recuperano il token corrente dal contesto tramite un custom hook `useAuth()`.

Intercettore per il Rinnovo Automatico La logica di rinnovo è centralizzata in un **intercettore** configurato sull'istanza globale di **Axios** (`axiosInstance.ts`). Questo intercettore ispeziona ogni risposta proveniente dal server.

- Se la risposta ha uno stato HTTP `401 Unauthorized`, l'intercettore "cattura" l'errore.

- Mette in pausa la richiesta originale fallita e avvia una nuova chiamata all'endpoint `/auth/refresh`. Essendo una normale richiesta HTTP, il browser allega automaticamente il cookie `HttpOnly` contenente il Refresh Token.

- In caso di successo, il nuovo Access Token viene salvato nel `AuthContext`, e la richiesta originale viene ritentata con il nuovo token.

- In caso di fallimento del refresh, l'utente viene disconnesso e reindirizzato alla pagina di login.

Questo approccio astrae completamente la complessità della gestione delle sessioni dai componenti della UI, che possono effettuare chiamate API senza doversi preoccupare della scadenza dei token.

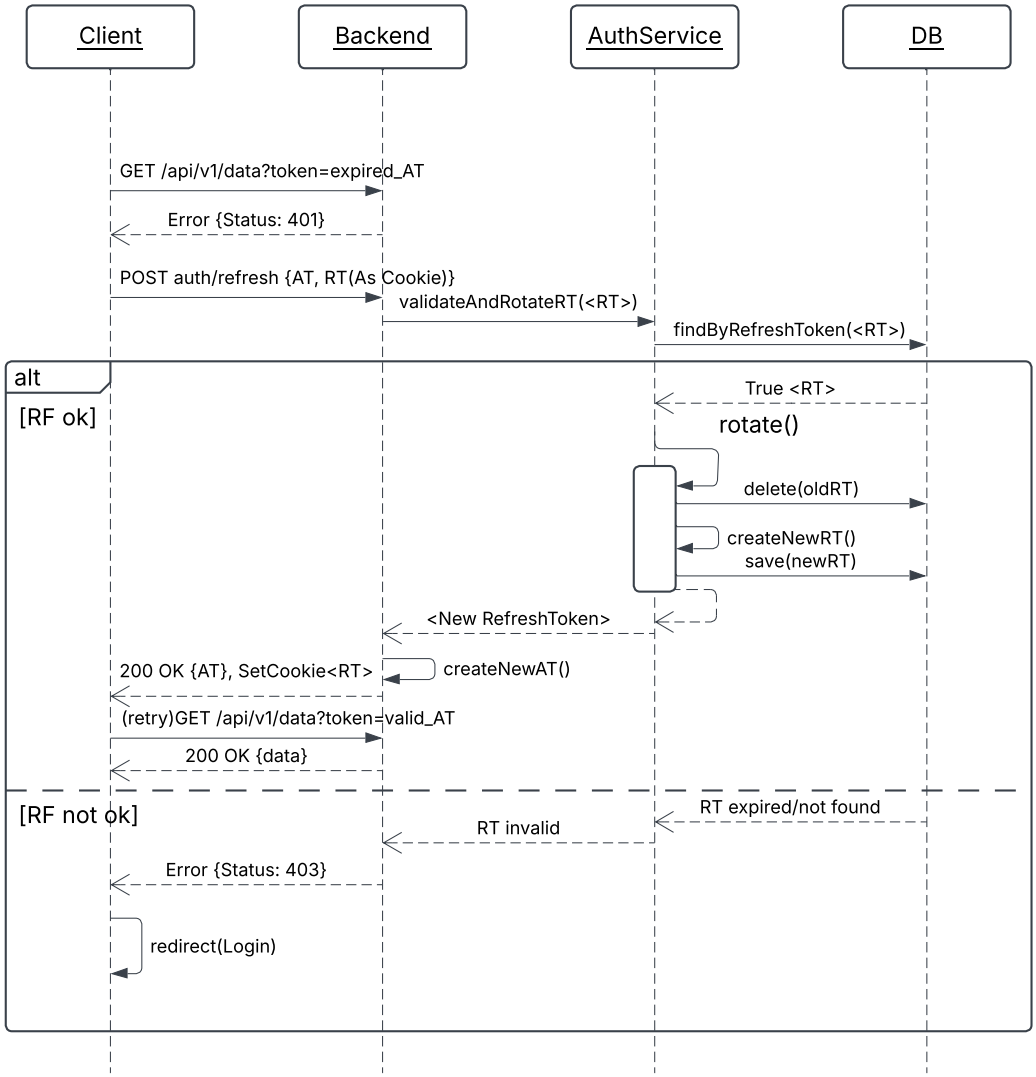

Figura 5.1: Diagramma di sequenza che illustra il flusso di gestione di un Access Token scaduto e la rotazione del Refresh Token, evidenziando le interazioni tra client, backend e database.

Capitolo 6

Metodologia e Infrastruttura di Sviluppo

Oltre alla progettazione e all'implementazione del software, un progetto di successo si basa su una solida metodologia di sviluppo e su un'infrastruttura di supporto affidabile. Questo capitolo descrive il processo operativo adottato per la realizzazione del progetto, dalla gestione agile alla strategia di testing, fino all'architettura della pipeline di Continuous Integration e Continuous Deployment (CI/CD) che automatizza il rilascio dell'applicazione.

6.1 Metodologia di Sviluppo Agile

Data la natura del progetto come "ponte tecnologico" e i vincoli temporali definiti (RNF5), si è scelto di non adottare un modello a cascata rigido, ma un approccio iterativo e incrementale, ispirato ai principi delle metodologie agili.

Collaborazione con gli Stakeholder Lo sviluppo è avvenuto in stretta collaborazione con il *product owner* e gli stakeholder aziendali di FlashStart. Questo ha garantito un allineamento costante con le esigenze del business e ha permesso di ricevere feedback tempestivi.

Iterazioni e Demo Il lavoro è stato organizzato in cicli di sviluppo brevi, assimilabili a degli *sprint*, della durata di circa due settimane. Al termine di ogni

ciclo, veniva presentata una demo dello stato di avanzamento del prodotto. Questo approccio ha permesso di:

- Validare i requisiti in modo incrementale.
- Identificare e correggere eventuali incomprensioni o problemi in una fase precoce.
- Mantenere alta la visibilità del progetto all'interno dell'azienda, rafforzando la fiducia degli stakeholder.

Questa metodologia si è rivelata vincente per un progetto con requisiti chiari ma che richiedeva flessibilità e velocità di esecuzione.

6.2 Strategia di Testing e Validazione

Per garantire la qualità, la robustezza e la non regressione del software, è stata implementata una strategia di testing a più livelli, coprendo sia il backend che il frontend.

6.2.1 Testing del Backend

Per il backend Java, sono state implementate due tipologie principali di test, sfruttando il framework di testing JUnit 5 e la libreria Mockito per la creazione di mock.

Unit Test I test unitari si concentrano sulla verifica del comportamento di singole classi o metodi in isolamento. Un esempio tipico è la verifica dei servizi di autenticazione, dove le dipendenze esterne (come repository di dati e servizi di validazione) vengono sostituite con oggetti mock. Questo approccio permette di testare in modo isolato la logica di business critica, come la rotazione dei token, la validazione delle credenziali e la gestione degli errori, senza dipendere da risorse esterne come database o servizi di rete.

Integration Test Per verificare l'interazione tra più componenti, come la logica di servizio e il layer di persistenza, sono stati scritti test di integrazione. Questi test utilizzano un database in-memory H2, configurato per emulare PostgreSQL, come specificato nel file di properties dei test. Ciò permette di testare il flusso completo di salvataggio e recupero dati (es. la creazione di un utente e del suo refresh token) in un ambiente controllato ma realistico.

6.2.2 Testing del Frontend

Per il frontend React, è stata utilizzata la libreria React Testing Library in combinazione con Jest. L'approccio si è concentrato sul testare i componenti dal punto di vista dell'utente.

Component Test I test verificano che i componenti si renderizzino correttamente e rispondano alle interazioni dell'utente. Un esempio è il test per la pagina di Login, `Login.test.tsx`, che simula l'inserimento di testo da parte dell'utente, il click sul pulsante di submit e verifica che vengano mostrati i messaggi di errore o di successo appropriati.

Mocking delle Chiamate API Per isolare i componenti frontend dal backend durante i test, le chiamate API effettuate tramite Axios sono state intercettate e simulate (mocking). Come visibile nei test, l'istanza di `axiosInstance` viene "mockata" per restituire risposte predefinite, permettendo di testare il comportamento del componente in caso di successo, fallimento o altri scenari di rete.

6.3 Deployment e Pipeline di CI/CD

Per automatizzare il processo di rilascio e garantire la coerenza degli ambienti, come richiesto da RNF4, è stata progettata e implementata una pipeline di Continuous Integration e Continuous Deployment (CI/CD).

6.3.1 Containerizzazione con Docker

Il fondamento dell'infrastruttura è la containerizzazione. L'intera applicazione (frontend, backend, database) è definita come un insieme di servizi. Questo garantisce che ogni sviluppatore e ogni ambiente di deployment esegua il software con le stesse dipendenze e configurazioni. Sono state utilizzate build multi-stage nei Dockerfile per creare immagini finali ottimizzate e leggere, separando le dipendenze di build da quelle di runtime.

6.3.2 Pipeline Ibrida con GitHub Actions e Self-Hosted Runner

È stata scelta un'architettura di CI/CD ibrida:

- **GitHub Actions** viene utilizzato come orchestratore del flusso di lavoro. La pipeline si attiva automaticamente a ogni push su rami specifici (es. `main` o `staging`).
- **Self-Hosted Runner** anziché un runner gestito da GitHub, è stato installato un agente (runner) direttamente sul server di deployment aziendale. Questa scelta strategica garantisce un maggiore controllo e sicurezza: le operazioni di build e deployment avvengono all'interno dell'infrastruttura aziendale, senza esporre credenziali o artefatti all'esterno.

Flusso di Deployment Il flusso tipico della pipeline è il seguente:

1. Lo sviluppatore effettua un push sul repository GitHub.
2. GitHub Actions avvia il workflow definito.
3. Il job viene assegnato al self-hosted runner.
4. Il runner esegue i seguenti passi:
    - Esegue il build delle immagini Docker per frontend e backend.
    - Esegue i test automatici per validare la build.

- Se i test passano, effettua il push delle nuove immagini su un registry privato (es. Docker Hub).
- Esegue il pull delle nuove immagini sul server di deployment.
- Riavvia i servizi utilizzando `docker compose up -d` per applicare l'aggiornamento senza downtime significativo.

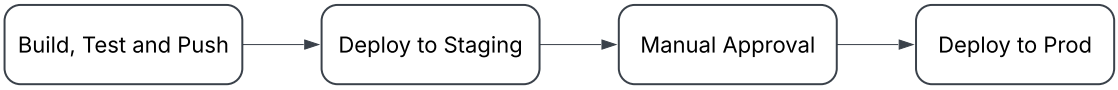

Figura 6.1: Diagramma che illustra il flusso di Continuous Integration e Continuous Deployment, dal push su Git al deployment sul server.

Bibliografia

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[BM22] Justus Bogner and Manuel Merkel. To type or not to type?: a systematic comparison of the software quality of javascript and typescript applications on github. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 658–669. ACM, May 2022.

[C+19] Cristian Coha et al. Evaluating docker storage performance: from workloads to graph drivers. *Journal of Cloud Computing*, 8(1):1–17, 2019.

[CN19] Dariusz Chęć and Ziemowit Nowak. The performance analysis of web applications based on virtual dom and reactive user interfaces. In Piotr Kosiuczenko and Zbigniew Zieliński, editors, *Engineering Software Systems: Research and Praxis*, pages 420–429, Cham, 2019. Springer International Publishing.

[DK20] Anna Derezinska and Karol Kwasnik. Performance-based refactoring of web application: A case of public transport. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, pages 611–618. SCITEPRESS, 2020.

[Fla24] Heather Flanagan. Token lifetimes and security in oauth 2.0: Best practices and emerging trends. *IDPro Body of Knowledge*, 1, 11 2024.

[GdCZ19] Taher Ahmed Ghaleb, Daniel Alencar da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24(4):2102–2139, 8 2019.

[Gut24] Felipe Gutierrez. *Spring Boot Reactive*, pages 327–362. Apress, Berkeley, CA, 2024.

[HKR+13] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19, 10 2013.

[JBS15] M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519, 2015.

[KS15] Anton Krasinskiy and Oleksandr Spivak. Performance evaluation of blocking and non-blocking i/o based web frameworks. Master's thesis, Blekinge Institute of Technology, 2015.

[MHR+12] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. volume 47, pages 683–702, 10 2012.

[MV24] Martin Moravcik and Blesson Varghese. Experimental assessment of containers running on top of virtual machines. *arXiv preprint arXiv:2401.07539*, 2024.

[OWA23] OWASP Foundation. JSON Web Token for Java Cheat Sheet. OWASP Cheat Sheet Series, 2023.

[RAKS18] Akond Rahman, Amritanshu Agrawal, Rahul Krishna, and Alexander Sobran. Characterizing the influence of continuous integration: empirical results from 250+ open source and proprietary projects. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*, ESEC/FSE '18, page 8–14. ACM, November 2018.

[RPMK24] Manish Rana, Ayush Pandey, Ankit Mishra, and Vishal Kandu. International journal on recent and innovation trends in computing and communication enhancing data security: A comprehensive study on the efficacy of json web token (jwt) and hmac sha-256 algorithm for web application security. *International Journal on Recent and Innovation Trends in Computing and Communication*, 11:4409–4416, 09 2024.

[SL24] Muhammad Shafique and Lucy Lwakatare. Containerization and its impact on devops practices. *arXiv preprint arXiv:2402.18435*, 2024.

[Syr23] Joona Syrjämäki. Exploring the advantages: A review of docker container technology in the devops operating model. In *Trepo, Tampere University Institutional Repository*, 2023.

[TPH+20] Daniel Teixeira, Ruben Pereira, Telmo Henriques, Miguel Silva, and João Faustino. A systematic literature review on devops capabilities and areas. *International Journal of Human Capital and Information Technology Professionals*, 11:1–22, 04 2020.

[Vri21] Alex Vrincean. Optimizing request handling using blocking & non-blocking i/o middleware. Master's thesis, 07 2021.

Ringraziamenti

Optional. Max 1 page.