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 per conto di 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.

A testimonianza della sua stabilità ed efficacia in un contesto operativo reale, al momento della stesura di questa tesi, il software sviluppato è impiegato in ambiente di piena produzione da aziende di calibro internazionale come Telefónica e T-Mobile USA.

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 marzo 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, mira 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[1]. 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. Per raggiungere tale scopo, la tesi segue un percorso strutturato che guida il lettore dalle fondamenta teoriche fino ai dettagli implementativi e processuali.

Inizialmente, il capitolo **Background** stabilirà il contesto tecnologico necessario, introducendo i paradigmi chiave su cui si fonda il progetto: dalle architetture per interfacce utente moderne come React, alla programmazione reattiva, fino ai protocolli di sicurezza e alle metodologie DevOps che abilitano l'automazione. Successivamente, il capitolo **Analisi** definirà formalmente il perimetro del progetto attraverso la specifica dei requisiti funzionali e non funzionali, derivati dalle esigenze aziendali.

[1] In metodologie agili, una soluzione di questo tipo è talvolta definita 'spike tecnologico', ovvero un esperimento mirato a esplorare una soluzione tecnica o ridurre un rischio, in questo caso per validare le tecnologie e fornire una funzionalità richiesta in attesa della nuova piattaforma principale.

Il capitolo successivo, **Metodologia di Sviluppo**, descriverà l'approccio adottato per la gestione del progetto. Verrà spiegato come l'analisi dei requisiti e del dominio applicativo si traduca in scelte strategiche per garantire un processo di sviluppo conforme e standardizzato.

Una volta definito il processo, il cuore analitico della tesi è rappresentato dal capitolo **Design**, dove verrà documentata l'architettura e si entrerà nel merito dell'applicazione pratica dei Design Pattern. Verranno identificati e discussi esempi concreti implementati nel codice sorgente, spiegando per ciascuno il problema che risolve e i benefici apportati in termini di flessibilità e manutenibilità.

Infine, il capitolo **Implementazione** collegherà il design astratto alle scelte tecnologiche concrete. In questa sezione verranno presentati frammenti di codice significativi e verranno analizzate le strategie di testing multilivello e la pipeline di CI/CD, documentando il processo che ha portato dal codice sorgente al deployment automatizzato dell'applicazione.

Capitolo 2

Background

Questo capitolo ha lo scopo di fornire le coordinate teoriche e tecnologiche necessarie per comprendere appieno le scelte architetturali e implementative che verranno discusse nel resto della tesi. Poiché il progetto, come illustrato nell'Introduzione, consiste nella realizzazione di un'applicazione web full-stack complessa con requisiti specifici di interattività, gestione dei dati e sicurezza, è fondamentale introdurre i paradigmi, i framework e le metodologie che ne costituiscono le fondamenta. Si procederà con un'analisi stratificata, partendo dai modelli per le interfacce utente moderne, per poi approfondire le architetture di backend, i protocolli di sicurezza e, infine, le pratiche di automazione che hanno garantito l'efficienza e l'affidabilità del ciclo di vita dello sviluppo.

2.1 Paradigmi per lo Sviluppo di Interfacce Utente Moderne

La realizzazione di una dashboard per l'analisi di dati, come quella oggetto di questa tesi, richiede un'esperienza utente (UX) estremamente fluida e reattiva. Per raggiungere tale obiettivo, l'architettura frontend si è basata sui moderni paradigmi a componenti, che vengono qui analizzati.

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)[1] 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.

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

[1] Con l'SSR, la pagina richiesta viene renderizzata inizialmente sul server e inviata al client come HTML completo. Questo approccio risolve i problemi di indicizzazione dei motori di ricerca e può migliorare la percezione della velocità di caricamento iniziale (First Contentful Paint).

nel codice. In questo modo, la logica di rendering e la struttura di presentazione sono gestite come un'unica unità coesa.

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].

2.1.1 Tipizzazione Statica con TypeScript

Per garantire la robustezza di un'applicazione destinata a un contesto aziendale, l'adozione della tipizzazione statica è stata una scelta strategica. 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"[2] (insicuro) per progettazione, una

[2]Un sistema di tipi è definito 'sound' (sicuro) se garantisce che un programma ben tipizzato non possa mai generare errori di tipo a runtime.

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

La capacità di gestire in modo efficiente e scalabile le richieste di dati statistici, spesso recuperati da fonti diverse, è un requisito cardine del progetto. Le architetture server tradizionali presentano limiti noti in questi scenari. Per questo motivo, è essenziale analizzare il paradigma reattivo e l'I/O non bloccante, che rappresentano la spina dorsale del backend dell'applicazione. 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"[3]. 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

[3]Il 'context switching' è l'operazione con cui la CPU salva lo stato di un processo o thread in esecuzione per poter passare all'esecuzione di un altro.

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 [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 molte-

plici 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

In un'architettura distribuita e stateless come quella adottata, il meccanismo di autenticazione e autorizzazione deve essere altrettanto scalabile e disaccoppiato. Questa sezione esplora i principi dell'autenticazione basata su token, con un focus sullo standard JSON Web Token (JWT), scelto per la gestione della sicurezza del sistema.

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

Lo sviluppo di una soluzione in un contesto agile e con vincoli temporali definiti, come descritto nell'Introduzione, ha reso imprescindibile l'adozione di metodologie che automatizzano e rendono affidabile il processo di rilascio, le quali rientrano nelle pratiche DevOps.

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 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.

2.4.3 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 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].

2.4.4 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 ("cosa fa"), e non funzionali, che ne specificano le proprietà, le qualità e i vincoli ("come è").

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.

RF1 Autenticazione e Gestione della Sessione Utente. Il sistema deve garantire un accesso sicuro e controllato.

- Permettere l'autenticazione tramite e-mail e password.
- Gestire una sessione utente sicura dopo il login e consentire il logout.
- Rinnovare automaticamente la sessione fino alla sua scadenza definitiva, senza richiedere un nuovo login manuale.

RF2 Gestione del Contesto Operativo. L'interfaccia deve permettere all'utente di definire e persistere il proprio contesto di analisi.

- Consentire la selezione di un "Customer" attivo da una lista.
- Consentire la selezione di un "Profile" di protezione associato al customer, inclusa un'opzione per aggregare i dati di tutti i profili ("All Profiles").
- Mantenere la selezione di Customer e Profile tra le diverse sessioni di utilizzo.
- Aggiornare dinamicamente i dati visualizzati al cambio del Customer o del Profile selezionato.

RF3 Presentazione della Dashboard di Riepilogo. La pagina principale deve fornire una sintesi visiva immediata dei dati di traffico.

- Mostrare statistiche aggregate (Total Queries, Blocked, Allowed, Threats Detected) con contatori animati per un feedback visivo immediato.
- Visualizzare una serie di grafici, ciascuno con la possibilità di selezionare un intervallo temporale indipendente (24h, 48h, 7d, etc.):
  - Un grafico a barre orizzontali per le "Categorie Bloccate" (Top N).
  - Un grafico a barre orizzontali per le "Categorie Più Richieste" (Top N).
  - Un grafico a ciambella (Donut) per la distribuzione delle minacce, con il totale delle query al centro.
  - Un istogramma (barre verticali) per i "Domini Più Richiesti" (Top N).

RF4 Consultazione e Analisi delle Configurazioni di Protezione. L'applicazione deve permettere un'ispezione dettagliata delle policy di sicurezza attive.

- Mostrare la configurazione della protezione in una struttura gerarchica espandibile (Moduli, Macro Categorie, Categorie).
- Visualizzare lo stato ("Allowed" / "Blocked") per ogni categoria e per ogni paese nelle regole di Geo-blocking.
- Mostrare il contenuto delle liste Blacklist e Whitelist.
- Fornire uno strumento di "Domain Check" che permetta di interrogare un dominio specifico e visualizzare il suo stato su tutti i profili del customer, indicando la ragione di un eventuale blocco (es. "Categoria Social Network bloccata", "Dominio in blacklist").

RF5 Visualizzazione delle Configurazioni di Rete. Deve essere presente una sezione per l'analisi delle reti associate al cliente.

- Listare le configurazioni di rete (IP Statico, Dinamico, DoH/DoT) in formato tabellare.
- Visualizzare il profilo di protezione, i server DNS associati, lo stato di attività ("Up"/"Down") e l'ultimo "signin" per ogni rete.

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. Requisito primario del progetto. L'applicazione deve operare esclusivamente in modalità di consultazione. Nessuna funzionalità di modifica, creazione o cancellazione dei dati e delle configurazioni deve essere implementata.

RNF2 Sicurezza. Il sistema deve proteggere la confidenzialità e l'integrità dei dati.

- La comunicazione tra client e server deve avvenire esclusivamente tramite protocollo HTTPS.
- Le password degli utenti devono essere archiviate nel database in modo sicuro (utilizzando funzioni di hash).

RNF3 Usabilità e Esperienza Utente (UX). L'interfaccia deve essere intuitiva e comunicare chiaramente il suo stato.

- Fornire indicazioni visive dello stato di caricamento dei dati (es. spinners/loaders).
- Gestire gli errori di comunicazione con le API mostrando messaggi informativi all'utente.
- Le sessioni utente devono essere persistenti per evitare la necessità di login frequenti.

RNF4 Conformità con le prassi aziendali. Il sistema deve aderire a uno stack tecnologico e a un modello di deployment predefiniti.

- L'intera applicazione deve essere containerizzata con Docker e orchestrata tramite Docker Compose.
- Il database deve essere PostgreSQL.
- Il frontend deve utilizzare React, TypeScript e pnpm come package manager.
- Il deployment deve avvenire su un server AlmaLinux interno all'infrastruttura aziendale, utilizzando immagini Docker ospitate su un container registry.

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:

- `http://apihq.flashstart.com/`: 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.

- **`https://api.fsflt.net/`**: 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

Metodologia di Sviluppo

Data la natura del progetto, caratterizzato come una soluzione ponte strategica con vincoli temporali ben definiti, l'adozione di un approccio di sviluppo tradizionale a cascata si sarebbe rivelata inadeguata alle specifiche esigenze del contesto aziendale. Per questo motivo, si è optato per un modello di lavoro iterativo e incrementale, fondato sui principi delle metodologie agili, che ha posto al centro della strategia di sviluppo la collaborazione continua e il feedback costante da parte degli stakeholder.

La pianificazione del lavoro è stata strutturata attraverso cicli di sviluppo brevi e regolari, della durata approssimativa di una settimana ciascuno, concettualmente assimilabili agli *sprint* tipici del framework Scrum. Al termine di ogni ciclo di lavoro, veniva organizzata una sessione dimostrativa nella quale veniva presentata una versione funzionante del prodotto direttamente al *product owner* e agli stakeholder.

Queste sessioni dimostrative non costituivano una mera formalità procedurale, bensì rappresentavano un meccanismo strategico di controllo qualità e allineamento agli obiettivi. Tale approccio ha generato benefici significativi su tre livelli principali.

In primo luogo, ha consentito la validazione incrementale dei requisiti: anziché attendere la conclusione del progetto per verificare la corrispondenza tra sviluppato e richiesto, ogni singola funzionalità veniva discussa, testata e approvata progressivamente durante il suo sviluppo. Questo processo ha garantito un allineamento costante e preciso con le esigenze specifiche del business, minimizzando il rischio di deviazioni dagli obiettivi prestabiliti.

In secondo luogo, il ciclo di feedback accelerato ha permesso l'identificazione

e correzione tempestiva di incomprensioni o criticità tecniche. La possibilità di individuare problematiche nelle fasi iniziali dello sviluppo ha drasticamente ridotto il rischio di dover intraprendere costose attività di rilavorazione in fasi avanzate del progetto, ottimizzando così l'utilizzo delle risorse disponibili e mantenendo il rispetto dei tempi di consegna.

Infine, la trasparenza garantita dalle *demo* periodiche ha assicurato un'elevata visibilità del progetto all'interno dell'organizzazione aziendale, contribuendo a consolidare la fiducia del product owner nella soluzione in fase di realizzazione e nella capacità di rispettare le scadenze concordate. La condivisione regolare dei progressi ha inoltre facilitato la comunicazione interdisciplinare e ha permesso di raccogliere input preziosi da diverse prospettive aziendali.

L'adozione di questo approccio metodologico agile e collaborativo si è quindi dimostrata la strategia più efficace per gestire la complessità intrinseca del progetto, garantendo al contempo la flessibilità necessaria per adattarsi alle mutevoli esigenze del contesto operativo e il mantenimento di una rigorosa aderenza agli obiettivi strategici definiti in fase di pianificazione.

Capitolo 5

Design

Questo capitolo illustra la progettazione architetturale e di dettaglio dell'applicazione, rispondendo ai requisiti funzionali e non funzionali definiti nel capitolo 3. 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.

5.1 Progettazione dell'Architettura di Alto Livello

Per garantire la portabilità e la riproducibilità degli ambienti, 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. 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 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.

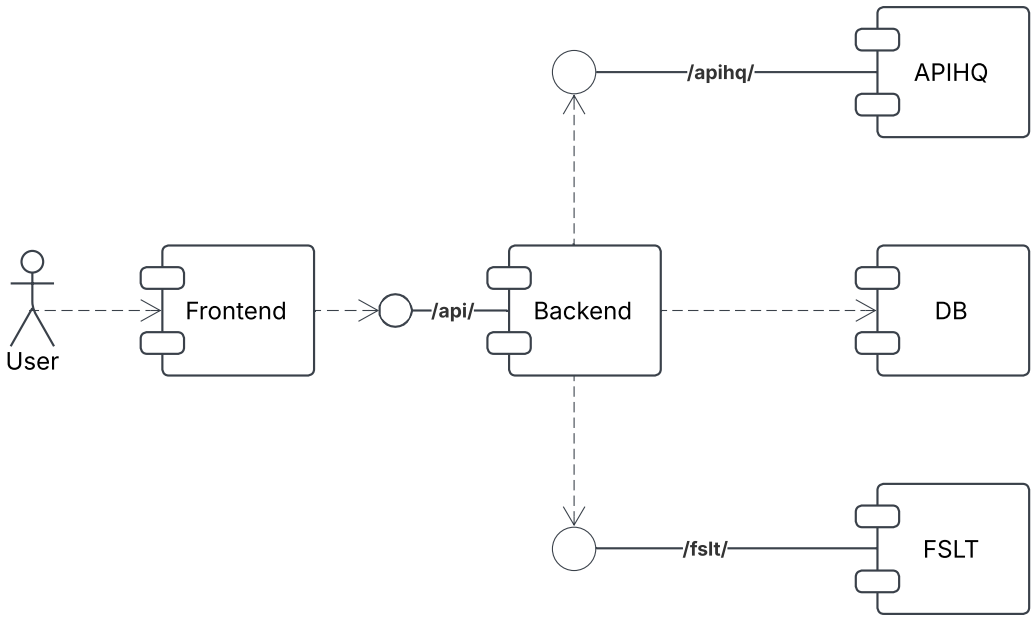

Figura 5.1: Diagramma dell'architettura logica di sistema, che illustra i componenti principali e i loro flussi di interazione.

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.

5.2 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 e facilmente estensibile. Per raggiungere tale scopo, sono state effettuate scelte di design mirate, partendo dalla definizione di un'architettura di alto livello fino all'applicazione di pattern specifici per risolvere le problematiche emerse durante la fase di analisi.

5.2.1 Architettura a Strati per la Separazione delle Responsabilità

Problema di Design Un'applicazione web complessa deve gestire diverse responsabilità: interpretare le richieste HTTP, eseguire la logica di business e interagire con un sistema di persistenza dati. Senza una struttura chiara, queste responsabilità tendono a mescolarsi, creando un codice monolitico, difficile da comprendere, testare e manutenere. Questo fenomeno, noto come alto accoppiamento e bassa coesione, rende qualsiasi modifica futura rischiosa e costosa.

Soluzione di Design Per risolvere questo problema, il design del backend adotta il pattern architetturale **Layered Architecture** (Architettura a Strati). Questo pattern scompone l'applicazione in due tipologie di componenti: un insieme di strati orizzontali con responsabilità comportamentali, e un layer trasversale che contiene i modelli dei dati.

Model Layer (Strato dei Modelli Dati) Questo non è uno strato comportamentale, ma un componente trasversale che definisce gli oggetti di dominio e le strutture dati dell'applicazione. È utilizzato da tutti gli altri strati. Il design distingue due tipi di modelli:

- Le **Entità del Dominio** che rappresentano i dati nel loro stato più puro e strutturato. Contengono la "verità" del sistema e vengono utilizzate principalmente dal Service Layer e dal Data Access Layer.
- I **Data Transfer Objects (DTO)**, che fungono da "contratto" dati per la comunicazione con l'esterno (il frontend) o tra gli strati stessi. Sono strutture dati semplici, ottimizzate per il trasferimento e spesso modellate per nascondere o aggregare i dati delle entità sottostanti.

Strati Comportamentali Gli strati che contengono la logica applicativa sono tre e comunicano in modo rigorosamente unidirezionale (dall'alto verso il basso):

- **Presentation Layer (Strato di Presentazione):** È il punto di ingresso dell'applicazione. La sua unica responsabilità è quella di gestire la comunicazione HTTP. Riceve le richieste, valida e mappa i dati in ingresso in *Data Transfer Objects (DTO)* e li passa allo strato di servizio. Al ritorno, riceve i DTO dal Service Layer e li serializza nella risposta HTTP. In questo progetto, questo strato è rappresentato dai *Controller*.
- **Service Layer (Strato di Servizio):** Rappresenta il cuore dell'applicazione. Contiene tutta la logica di business e orchestra le operazioni. Opera sulle *Entità* del dominio, applica le regole di business e coordina le interazioni con il Data Access Layer. È completamente agnostico rispetto al protocollo HTTP o alla fonte di persistenza.
- **Data Access Layer (Strato di Accesso ai Dati):** La sua unica responsabilità è la comunicazione con il database. Incapsula tutta la logica per mappare le *Entità* del dominio da e verso le tabelle del database. In questo progetto, tale strato è implementato tramite il **Repository Pattern**.

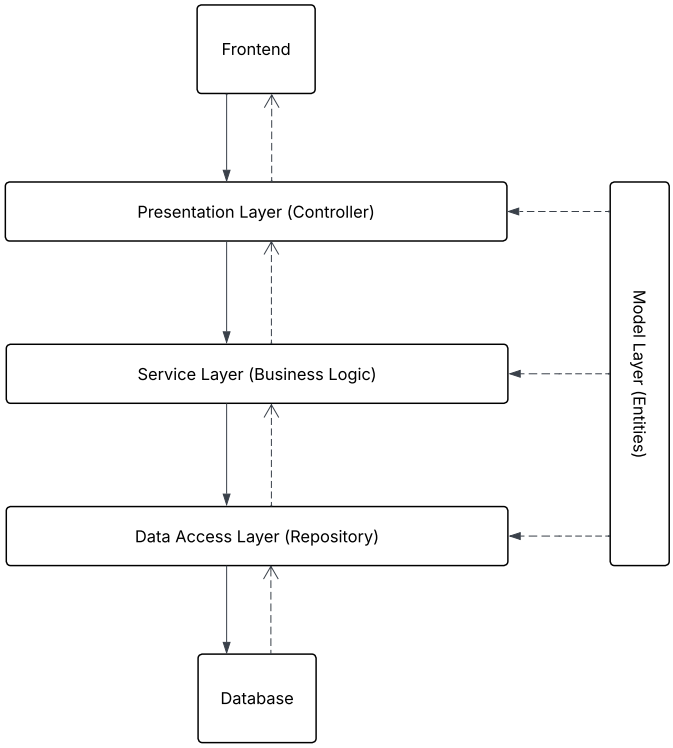

Figura 5.2: Diagramma che illustra l'architettura a tre strati del backend e il flusso di comunicazione unidirezionale.

Questa struttura garantisce un basso accoppiamento e un'alta coesione, rendendo il sistema più testabile (ogni strato può essere testato in isolamento, mockando le sue dipendenze) e più facile da manutenere ed estendere.

5.2.2 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.

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.

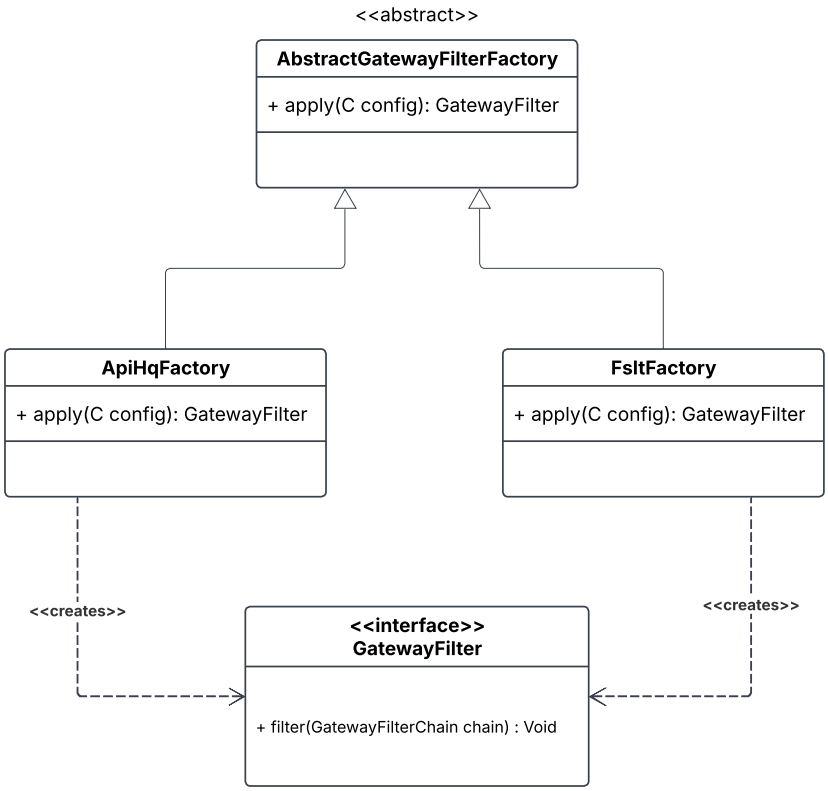

Figura 5.3: Diagramma delle classi che illustra l'applicazione del pattern Factory Method per la creazione dei filtri del gateway.

5.2.3 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 adotta il pattern Singleton. Tuttavia, invece di ricorrere a un'implementazione manuale e statica del pattern, che può introdurre rigidità e problemi di testabilità, il design prevede di delegare la gestione del ciclo di vita di questi oggetti a un componente infrastrutturale basato sul principio di Inversion of Control (IoC).

Secondo questo approccio, i componenti di servizio (come quelli per l'autenticazione o la generazione di token) e di accesso ai dati vengono registrati presso un gestore centrale. Sarà responsabilità di questo gestore istanziarli una sola volta (garantendo l'unicità del Singleton) e fornire tale istanza a qualsiasi altro componente che ne dichiari una dipendenza. Questo meccanismo, noto anche come Dependency Injection, permette ai componenti di essere ignari dei dettagli di creazione e del ciclo di vita delle proprie dipendenze.

5.2.4 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 e un servizio per la gestione della sessione, dei quali molteplici interagiscono anche con il componente di persistenza dati. 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.

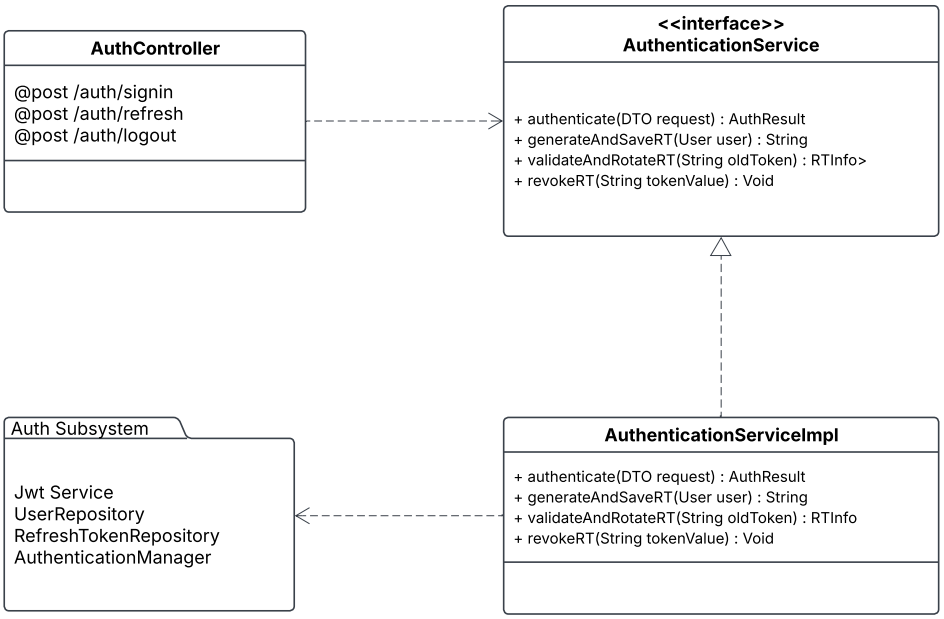

Figura 5.4: Diagramma delle classi che mostra come `AuthenticationService` agisca da Facade, semplificando l'accesso al sottosistema di autenticazione.

5.2.5 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.

5.3 Design del Frontend

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.

5.3.1 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.

5.3.2 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.

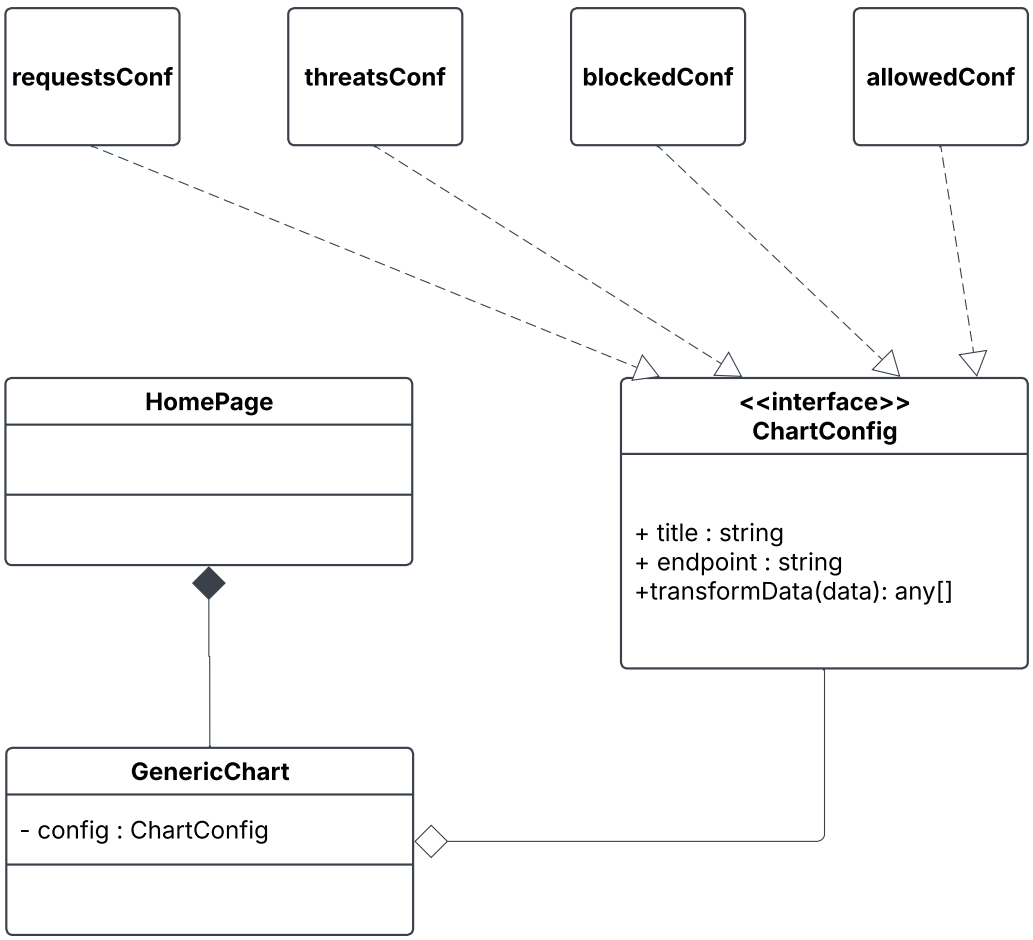

Figura 5.5: Diagramma che illustra come il componente `GenericChart` utilizzi diversi oggetti `ChartConfig` (strategie) per renderizzare grafici differenti.

5.3.3 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.

5.3.4 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 compo-

nente 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 6

Implementazione

Questo capitolo illustra come i principi e le architetture definiti nel capitolo 5 siano stati tradotti in scelte tecnologiche concrete e codice sorgente. L'obiettivo è dimostrare come le decisioni di progettazione abbiano guidato la selezione e l'utilizzo dello stack tecnologico per realizzare gli obiettivi del progetto.

6.1 Traduzione del Design in Scelte Tecnologiche

La fase implementativa ha richiesto di selezionare gli strumenti più adatti a realizzare l'architettura a strati, i pattern di comunicazione e la gestione dello stato delineati nel design.

6.1.1 Backend: la Scelta di un Ecosistema Reattivo con Spring WebFlux

Il design del backend (Sezione 5.2) richiedeva un'architettura a strati robusta, la gestione di servizi stateless (Singleton) e, soprattutto, la capacità di gestire in modo efficiente un alto numero di operazioni di I/O concorrenti, data la necessità di interfacciarsi con API esterne.

Per tradurre questi requisiti in una soluzione concreta, la scelta è ricaduta sull'ecosistema *Java* e sul framework *Spring Boot*. Java offre robustezza e un ecosistema maturo, mentre Spring Boot fornisce un potente meccanismo di *Inversion of Control (IoC)* e *Dependency Injection*, che rappresenta l'implementazione pratica

perfetta per il design basato sul pattern Singleton per la gestione dei servizi, astraendo completamente lo sviluppatore dalla loro creazione e dal loro ciclo di vita.

Tuttavia, per ovvire al problema dell'I/O concorrente, si è fatta una scelta implementativa fondamentale: adottare il paradigma di programmazione reattiva. All'interno dell'ecosistema Spring, questo ha significato scegliere Spring WebFlux al posto del tradizionale Spring MVC. Questa scelta permette di realizzare un backend non bloccante, che è la soluzione tecnica ideale per un'applicazione il cui carico di lavoro è dominato da attese di rete. Per mantenere la coerenza del modello reattivo end-to-end, anche per l'accesso ai dati è stata fatta una scelta conseguente: utilizzare Spring Data R2DBC invece del classico JDBC, garantendo che nessuna operazione, neanche sul database, potesse bloccare i thread principali dell'applicazione.

6.1.2 Frontend: Realizzazione dei Pattern di Design con React e Axios

Il design del frontend (Sezione 5.3) prevedeva un'architettura a componenti, una gestione centralizzata dello stato (pattern Observer) e un meccanismo robusto e centralizzato per la comunicazione con le API (pattern Singleton e Decorator).

Componenti e Stato Il design basato sulla composizione di oggetti è stato implementato utilizzando la libreria React, come dettato dal requisito non funzionale RNF4. La sua architettura nativamente basata su componenti ha permesso di tradurre direttamente il design di componenti "Container" e "Presentazionali". Per realizzare il pattern Observer per la gestione dello stato globale, si è scelto di utilizzare la React Context API. Questa soluzione, interna a React, è stata preferita a librerie esterne più complesse (come Redux) perché offriva il livello di funzionalità necessario per il progetto (sottoscrizione e notifica dei cambiamenti di stato) in modo più semplice e integrato.

Comunicazione API Centralizzata Il design di un punto di accesso unico e decorato per le chiamate API ha guidato la scelta e l'utilizzo della libreria Axios.

Per implementare il pattern Singleton, è stata creata un'unica istanza configurata di Axios (`axiosInstance.ts`), esportata e utilizzata in tutta l'applicazione. Il pattern Decorator è stato realizzato sfruttando la funzionalità nativa degli intercettori di Axios. Sono stati configurati due intercettori:

- Un *intercettore di richiesta*, che "decora" ogni chiamata in uscita aggiungendo dinamicamente l'header di autenticazione `Authorization`.
- Un *intercettore di risposta*, che "decora" la gestione degli errori intercettando le risposte `401 Unauthorized` per avviare in modo trasparente il flusso di rinnovo del token.

6.2 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, come conseguenza del requisito RNF2. e.

Generazione dei Token e Gestione dei Cookie Al momento del login, l' `Auth Controller` 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`[1], `Secure` (i.e. utilizzare solo HTTPS), `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*.

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.

[1] L'attributo HttpOnly impedisce che il cookie possa essere letto tramite JavaScript. Fondamentale per mitigare attacchi di tipo Cross-Site Scripting (XSS), in cui uno script malevolo potrebbe tentare di rubare il token di sessione.

- 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.

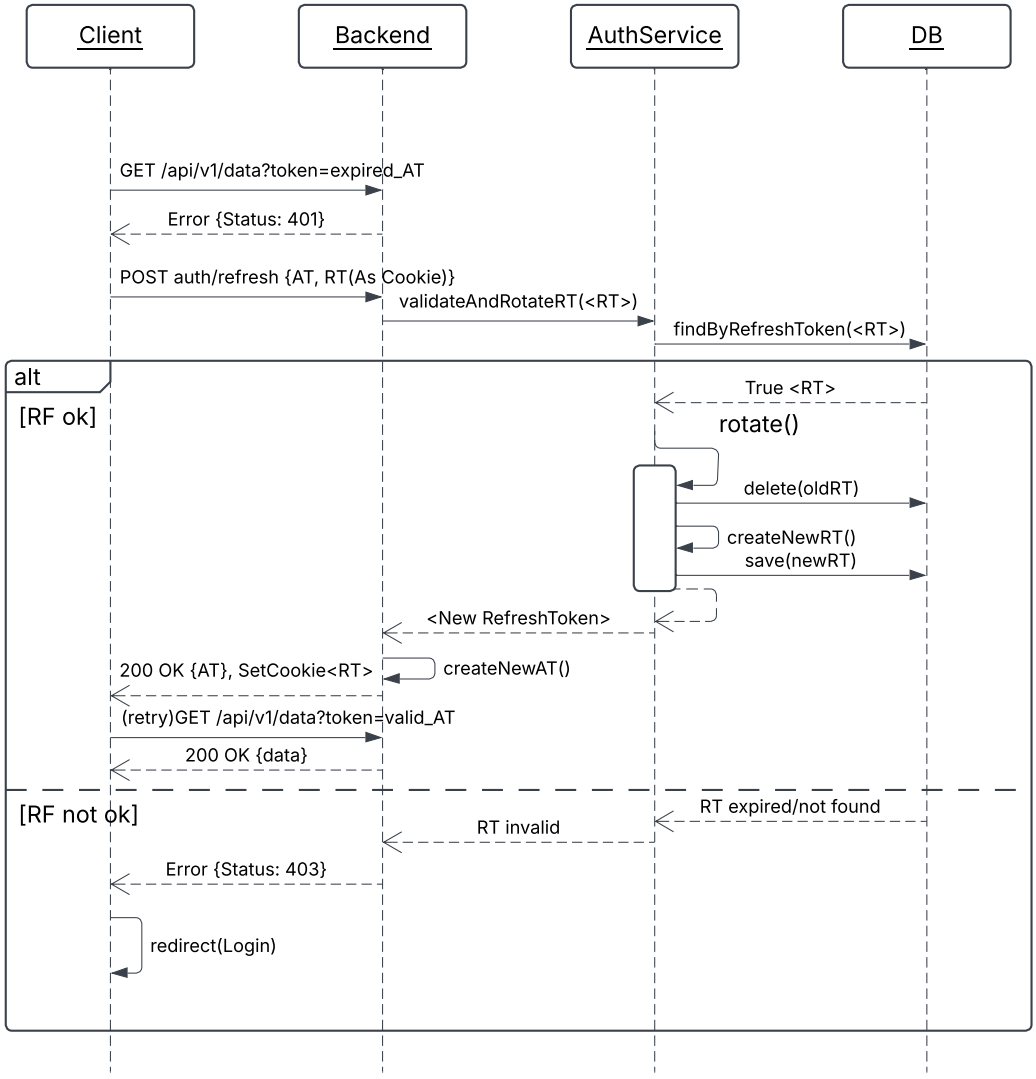

Figura 6.1: Diagramma di sequenza che illustra il flusso di gestione di un Access Token scaduto e la rotazione del Refresh Token, evidenziando le interazioni tra client, backend e database.

6.2.1 Struttura dell'Interfaccia Utente e Navigazione

L'intera interfaccia è sovrastata da un'intestazione (*header*) persistente che ospita gli elementi di controllo del contesto globale: due menu a tendina permettono all'utente di selezionare il `Customer` e il `Profile` attivi, i cui valori vengono mantenuti tra le sessioni. Accanto a questi, è presente il pulsante di logout per terminare la sessione.

Pagina di Login Costituisce il punto di accesso sicuro al sistema, in risposta al requisito RF1. L'interfaccia è volutamente minimale per ridurre le distrazioni e si compone di due campi di input per e-mail e password, un pulsante di submit e un meccanismo per la visualizzazione di messaggi di errore. Qualsiasi tentativo di accesso non autenticato ad altre pagine scatena il `redirect` a questa pagina.

Dashboard Principale È la landing page dopo l'autenticazione (Figura 6.2) e risponde al requisito RF3. Offre una sintesi visiva immediata dello stato del servizio tramite quattro riquadri principali che mostrano le statistiche aggregate: *Threats Detected*, *Blocked Requests*, *Not Existing Domains* e *Total DNS Requests*. La parte inferiore della pagina è dedicata a quattro grafici di analisi, ciascuno dotato di un selettore di intervallo temporale indipendente (24h, 7d, etc.) per consentire analisi granulari.

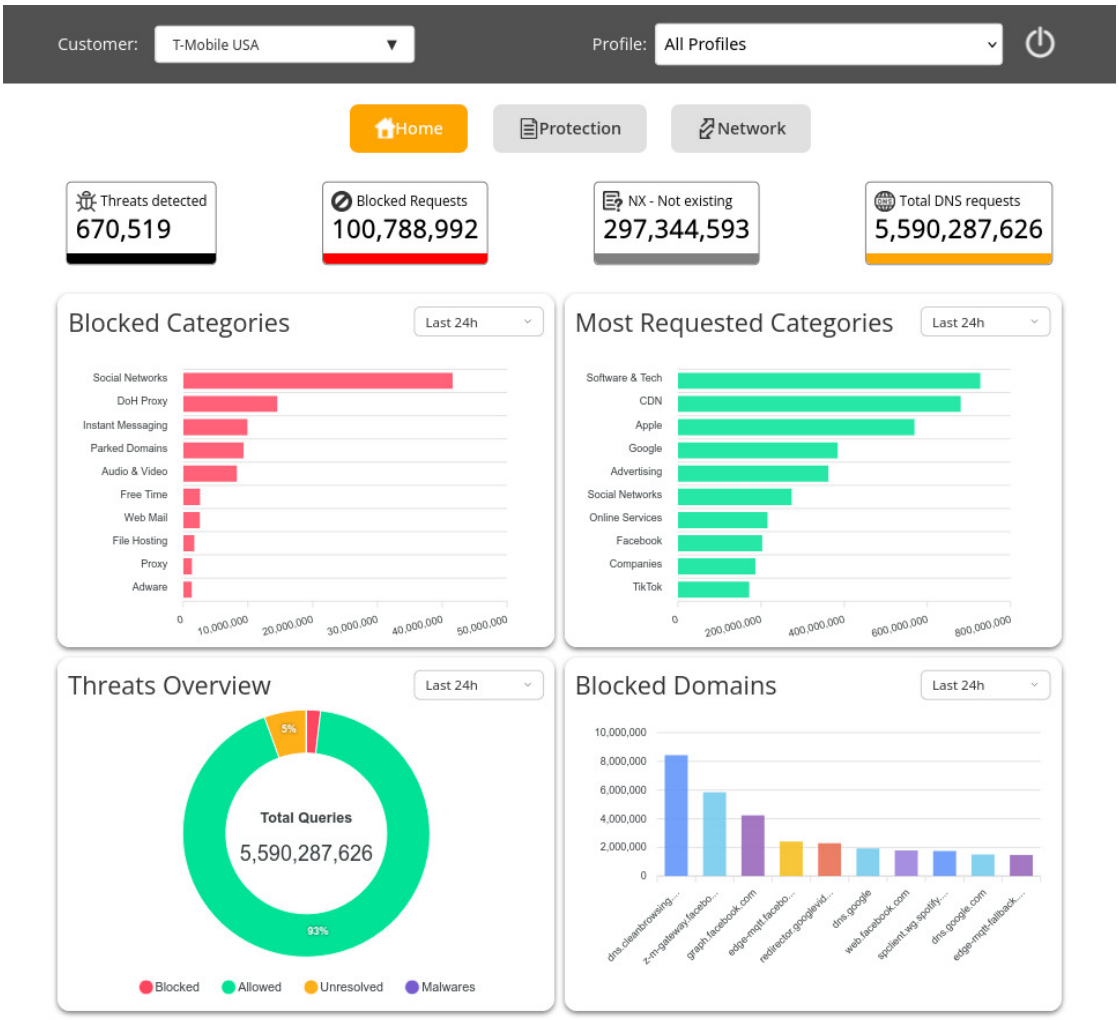

Figura 6.2: La Dashboard Principale, con i riquadri delle statistiche e i quattro grafici di analisi.

Pagina delle Policy di Protezione Questa sezione è dedicata alla consultazione delle policy di sicurezza (requisito RF4). La sua funzionalità principale è il **Domain Check Tool** (Figura 6.3), uno strumento interattivo che si interfaccia direttamente con l'endpoint `http://apihq.flashstart.com` per fornire un'analisi in tempo reale dello stato di un dominio su tutti i profili del cliente.

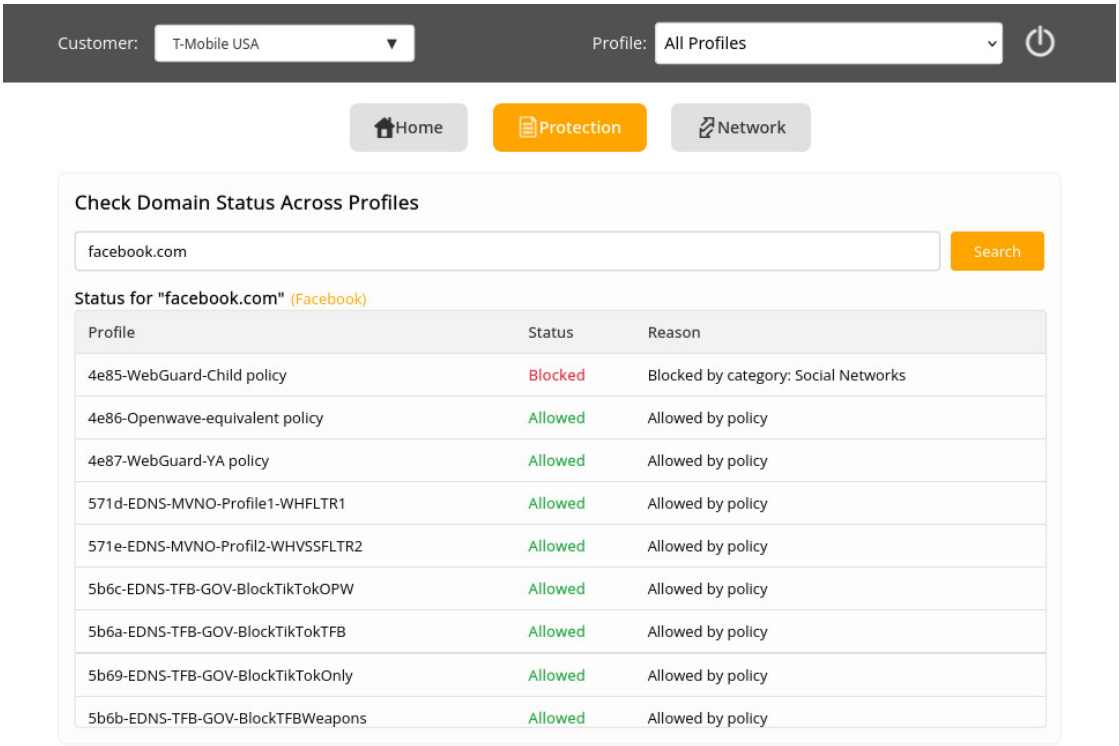

Figura 6.3: Dettaglio dello strumento "Domain Check Tool" all'interno della pagina di protezione.

Inoltre, la pagina (Figura 6.4) utilizza una serie di tabelle espandibili (accordion) per presentare in modo ordinato le complesse configurazioni di blocco, incluse le categorie di contenuti e le regole di geo-blocking, permettendo all'utente di ispezionare facilmente le policy attive.

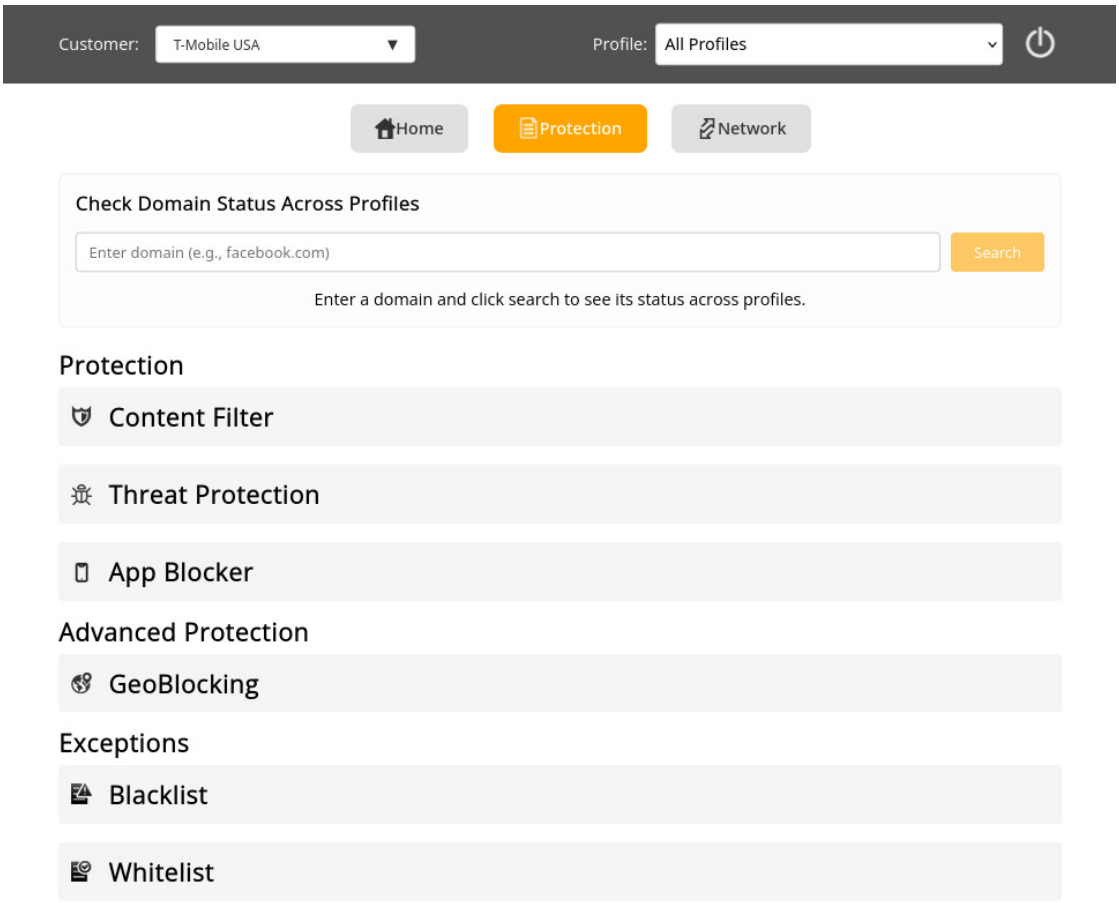

Figura 6.4: La pagina di consultazione delle policy, con le tabelle espandibili per le categorie.

Pagina di Gestione della Rete La pagina Network (Figura 6.5) risponde al requisito RF5, fornendo un elenco completo delle configurazioni di rete associate al cliente. L'interfaccia utilizza lo stesso componente *accordion* utilizzato dalla pagina Protection per raggruppare logicamente le diverse tipologie di rete, come indirizzi IPv4, IPv6, e connessioni crittografate DoH (DNS Over HTTPS) e DoT (DNS Over TLS). Una caratteristica di questa schermata è la sua natura dinamica: il numero e il tipo di opzioni visualizzate possono variare in base al piano di sottoscrizione del cliente, mostrando solo le configurazioni a cui ha effettivamente diritto.

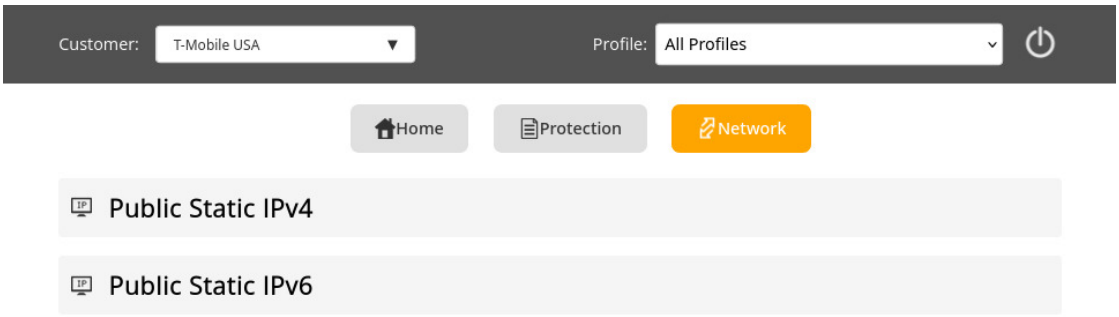

Figura 6.5: La pagina Network, che mostra le diverse configurazioni di rete raggruppate in sezioni.

6.3 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.

6.3.1 Testing del Backend

La strategia di testing per il backend è stata strutturata su livelli multipli per validare non solo la logica di business, ma anche l'infrastruttura di routing e

sicurezza. Come framework di riferimento è stato scelto JUnit 5[2], supportato dalla libreria Mockito[3] per la gestione delle dipendenze simulate.

Test Unitari I test unitari si sono concentrati sulla verifica della correttezza della logica implementata nei servizi, isolando completamente le dipendenze esterne. L'esempio principale è il testing del servizio `AuthenticationService`. Per validare il comportamento del metodo `validateAndRotateRefreshToken`, che restituisce un oggetto `Mono`, è stata utilizzata la libreria `reactor-test` con il suo `StepVerifier`. Le dipendenze sono state mockate con Mockito e la suite ha coperto scenari di successo (token valido) e di errore (token scaduto o non trovato), verificando la corretta emissione di eventi `onNext` o `onError`.

Test della Configurazione del Gateway e dei Filtri Una parte fondamentale della strategia di testing ha riguardato la validazione del componente Spring Cloud Gateway, che agisce come unico punto di accesso (Single Point of Entry) e orchestra le chiamate verso i servizi esterni di FlashStart. Questi test, a differenza dei classici test unitari, verificano la corretta configurazione e il comportamento dell'infrastruttura.

- **Test dei Filtri Personalizzati:** Sono stati creati test specifici per i filtri custom. Utilizzando `WebTestClient`, fornito da Spring Boot, questi test avviano un server embedded con una rotta di prova che applica il filtro in esame. Viene quindi inviata una richiesta HTTP a questo endpoint e si asserisce che il filtro abbia modificato la richiesta come atteso, ad esempio aggiungendo correttamente l'header `X-API-KEY` o `Authorization`. Questo garantisce che la logica di manipolazione delle chiamate, cruciale per l'integrazione con le API esterne, sia corretta e non soggetta a regressioni.
- **Test della Configurazione delle Rotte:** Il file `GatewayConfigTest.java` verifica la corretta costruzione delle rotte a partire dal file di configurazione. Questo test assicura che i predicati (es. path) e i filtri definiti in `application.properties` vengano caricati e applicati correttamente,

[2] `https://junit.org/junit5/`
[3] `https://site.mockito.org/`

garantendo che le richieste in arrivo siano instradate verso gli upstream corretti.

Test di Integrazione I test di integrazione hanno validato il funzionamento congiunto dei diversi strati applicativi. La base di questi test è la classe `BackendApplicationTests.java`, che con il suo test `contextLoads()` funge da "smoke test" per assicurare che l'intero contesto di Spring possa essere caricato senza errori. Per le interazioni con il database, è stato impiegato un database H2 in-memory con il driver R2DBC, garantendo un ambiente isolato ma fedele. Il test di integrazione dell'autenticazione in `AuthControllerTest.java` esemplifica l'approccio: dopo aver inserito un utente nel database H2, si invoca l'endpoint di login e si verifica che il `RefreshToken` sia stato correttamente persistito.

6.3.2 Testing del Frontend

La strategia di testing adottata per il frontend si basa sull'utilizzo del framework **Jest**[4], una piattaforma completa che integra le funzionalità di test runner, libreria di asserzioni e sistema di gestione dei mock. L'approccio metodologico seguito privilegia la verifica del comportamento dell'applicazione dal punto di vista dell'esperienza utente, anziché concentrarsi sui dettagli implementativi interni dei componenti.

Per facilitare l'interazione con i componenti React all'interno dell'ambiente di test, è stata integrata la libreria React Testing Library (RTL), che adotta come principio fondamentale: *"Più i tuoi test assomigliano al modo in cui il tuo software viene usato, più fiducia ti possono dare"* [D$^+$24].

Definizione dei Criteri di Successo La valutazione della correttezza dei componenti è stata standardizzata attraverso la verifica della percezione e dell'interattività degli elementi dell'interfaccia utente. Questo approccio utilizza le utility di RTL per eseguire query di selezione basate sui principi di accessibilità web.

A titolo esemplificativo, per la pagina di Login, un test di rendering corretto deve verificare che:

[4] `https://jestjs.io/`

- Sia presente un campo di input per l'email, identificabile attraverso la sua etichetta semantica utilizzando `screen.getByLabelText('Email')`;
- Esista un campo di input per la password, anch'esso accessibile tramite la corrispondente etichetta;
- Sia disponibile un pulsante di autenticazione, localizzabile mediante `screen .getByRole('button',{ name: /login/i })`.

Il successo di queste asserzioni costituisce il criterio di accettazione per un rendering corretto, garantendo che tutti gli elementi necessari all'interazione siano presenti e accessibili secondo gli standard di usabilità.

Simulazione delle Interazioni Utente La validazione della reattività dei componenti viene effettuata attraverso la simulazione delle interazioni che un utente reale eseguirebbe, impiegando la libreria **`@testing-library/user-event`**. Questa libreria fornisce un'API ad alto livello che replica fedelmente il comportamento degli eventi del browser.

Continuando con l'esempio della pagina di Login, la sequenza di test delle interazioni comprende:

1. **Simulazione dell'inserimento dati:** Utilizzo di `userEvent.type()` per emulare l'immissione dell'indirizzo email e della password nei rispettivi campi di input;
2. **Simulazione dell'azione di submit:** Impiego di `userEvent.click()` per replicare la pressione del pulsante di autenticazione.

Successivamente a questa sequenza di interazioni, il test procede con la verifica della risposta dell'applicazione, che tipicamente prevede l'esecuzione di una chiamata API per la validazione delle credenziali.

Isolamento delle Dipendenze Esterne e Verifica degli Effetti Per garantire l'isolamento della logica di interfaccia utente dalle dipendenze del backend e assicurare test deterministici, è stata implementata una strategia di mocking utilizzando le funzionalità native di Jest. Le chiamate API, gestite tramite la

libreria Axios, vengono intercettate e simulate per controllare i diversi scenari di risposta.

La metodologia di testing prevede:

- **Verifica delle chiamate API:** Dopo l'esecuzione dell'azione simulata, il test utilizza `expect(axios.post).toHaveBeenCalledWith(...)` per validare che la chiamata API sia stata effettuata con l'endpoint corretto e il payload appropriato;
- **Testing degli scenari di successo:** Configurando il mock di Axios per restituire una risposta positiva, il test verifica che l'applicazione esegua correttamente la navigazione verso la dashboard;
- **Gestione degli scenari di errore:** Simulando risposte di errore (ad esempio, stato HTTP 401), il test impiega le query asincrone di RTL per attendere e verificare la corretta visualizzazione dei messaggi di errore, come "Credenziali non valide".

Questa metodologia integrata, orchestrata dal framework Jest, consente una validazione completa del flusso di interazione di ciascun componente, dalla fase di rendering iniziale fino alla gestione delle risposte asincrone del backend.

6.4 Infrastruttura di Build e Deployment

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).

Containerizzazione con Docker Il fondamento dell'infrastruttura è l'implementazione della containerizzazione tramite **Docker**. L'intera applicazione (frontend, backend, database) è definita come un insieme di servizi in un file `docker-compose.yaml`. Per ottimizzare le immagini, sono state utilizzate build multi-stage nei `Dockerfile`, separando le dipendenze di build da quelle di runtime per produrre artefatti finali leggeri.

Pipeline Ibrida con GitHub Actions È stata implementata un'architettura di CI/CD ibrida. **GitHub Actions** è stato utilizzato come orchestratore per definire il flusso di lavoro, che si attiva automaticamente al push sul branch `main`. I job di build e deployment vengono eseguiti su un **Self-Hosted Runner** installato direttamente sul server aziendale, garantendo sicurezza e accesso controllato alle risorse interne.

Flusso di Deployment a Fasi Il flusso di lavoro implementato è suddiviso in quattro fasi sequenziali per garantire un rilascio sicuro e controllato:

1. **Build e Push delle Immagini:** Il primo job costruisce le immagini Docker per frontend e backend. Ogni immagine viene taggata sia con `latest` sia con l'hash del commit Git, per garantire un riferimento immutabile e tracciabile. Le immagini vengono quindi caricate su Docker Hub.

2. **Deployment in Staging:** La pipeline procede al deployment automatico in un ambiente di *staging*, una replica di quello di produzione, per permettere la validazione manuale e i test esplorativi in un contesto realistico.

3. **Gate di Approvazione Manuale:** Un punto di controllo umano cruciale. La pipeline si mette in pausa e crea una "Issue" su GitHub, richiedendo un'approvazione esplicita. Questo previene rilasci non autorizzati e crea una cronologia documentata delle decisioni di deployment.

4. **Deployment in Produzione:** Solo dopo l'approvazione, il flusso procede all'ultimo passo. Il runner esegue il pull delle medesime immagini verificate in staging e aggiorna i servizi nell'ambiente di produzione, garantendo la massima coerenza.

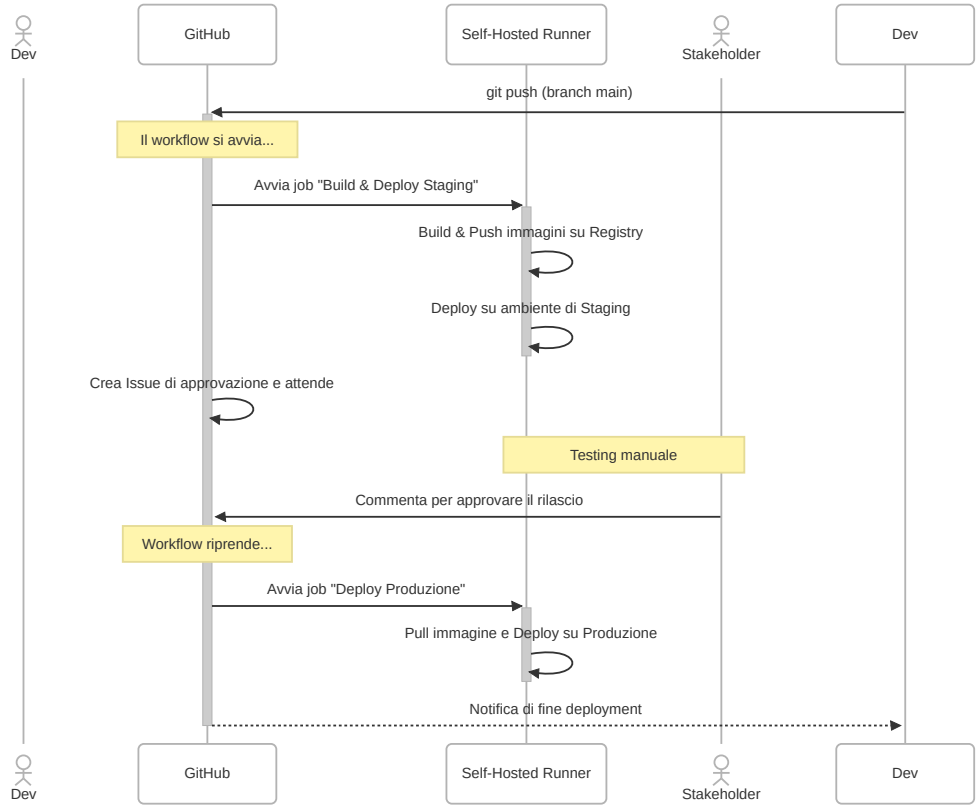

Figura 6.6: Diagramma che illustra il flusso di Continuous Integration e Continuous Deployment, dal push su Git al deployment sul server.

Capitolo 7

Conclusioni e Sviluppi Futuri

Dal punto di vista pratico, l'obiettivo è stato pienamente raggiunto. È stata sviluppata e rilasciata un'applicazione web full-stack, basata su un'architettura moderna che gode dell'implementazione di un robusto flusso di autenticazione. Il successo e la stabilità della soluzione sono testimoniati dalla sua adozione in ambiente di piena produzione da parte di clienti di calibro internazionale come Telefónica e T-Mobile USA.

Dal punto di vista accademico e formativo, il progetto ha rappresentato un'opportunità unica di lavoro indipendente, fornendo piena libertà per quanto riguarda la metodologia di sviluppo e gran parte della progettazione.

In virtù della sua natura di "soluzione ponte", progettata per colmare un vuoto funzionale fino al rilascio della nuova piattaforma aziendale, e del pieno raggiungimento degli obiettivi prefissati, non sono previste evoluzioni sostanziali del software. Il suo ciclo di vita si conclude con successo con l'attuale versione in produzione, avendo adempiuto al suo scopo strategico.

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.

[D+24] Kent C. Dodds et al. React testing library. `https://testing-library.com/docs/react-testing-library/intro/`, 2024. Guiding Principle: "The more your tests resemble the way your software is used, the more confidence they can give you".

[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.

[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*. PhD thesis, 07 2021.

Ringraziamenti

Optional. Max 1 page.