Studio e Implementazione di un Gateway API con Spring Cloud in Architetture a Microservizi

Andrea Donadello

17 giugno 2025

Sommario

Il presente lavoro di progetto universitario si concentra sull'analisi approfondita e sull'implementazione pratica di un Gateway API utilizzando Spring Cloud Gateway.
In un'era di trasformazione digitale rapida, la gestione strategica delle API è emersa come un fattore critico per le aziende che adottano architetture a microservizi, dove la complessità della comunicazione inter-servizio, della sicurezza e del routing può diventare proibitiva senza un punto di ingresso centralizzato.
Questo studio si propone di esplorare Spring Cloud Gateway come soluzione robusta e reattiva per affrontare tali sfide.

La metodologia adottata per questo progetto combina elementi di indagine (*Survey*), elaborazione (*Elaboration*) e applicazione pratica (*Application*).
È stata condotta un'indagine sui concetti fondamentali dei Gateway API e sulle loro funzionalità comuni, seguita da un'elaborazione dettagliata dell'architettura e delle caratteristiche specifiche di Spring Cloud Gateway.
Il cuore del progetto consiste nell'implementazione di un prototipo di Gateway API, disponibile nel repository Git fornito (`https://github.com/ddonazz/api-gateway`), che dimostra funzionalità chiave quali il routing dinamico, la gestione centralizzata della sicurezza (autenticazione e autorizzazione), la limitazione del tasso di richieste (rate limiting) e la gestione delle eccezioni.

I risultati ottenuti evidenziano i significativi vantaggi di Spring Cloud Gateway, tra cui una maggiore sicurezza, un'accelerazione dell'innovazione e tempi di commercializzazione ridotti, una migliore esperienza utente e una conformità normativa più efficiente.
Tuttavia, l'analisi critica ha anche rivelato sfide intrinseche all'adozione di un Gateway API, come l'aumento della complessità architetturale e il potenziale di introduzione di latenza aggiuntiva.
Il lavoro si conclude con una discussione sui contributi del progetto e sulle prospettive future per superare le limitazioni identificate, proponendo sviluppi che potrebbero ulteriormente migliorare la robustezza e l'applicabilità di tali soluzioni.

Indice

Elenco delle tabelle

Indice del codice

Capitolo 1

Introduzione

1.1 Contesto e Motivazioni

Il panorama dello sviluppo software contemporaneo è profondamente influenzato dall'adozione sempre più diffusa delle architetture a microservizi. Questo modello, che scompone le applicazioni monolitiche in un insieme di servizi piccoli, autonomi e accoppiati in modo lasco, offre numerosi vantaggi in termini di scalabilità, resilienza e agilità nello sviluppo. Tuttavia, l'adozione dei microservizi introduce anche nuove sfide significative, in particolare per quanto riguarda la gestione della comunicazione tra i servizi, la sicurezza, il monitoraggio e il routing delle richieste. Man mano che il numero di microservizi aumenta, la complessità di gestire le interazioni dirette tra client e servizi backend può diventare insostenibile, portando a problemi come la logica client frammentata e la difficoltà nella gestione delle preoccupazioni trasversali.

In questo contesto, il Gateway API emerge come un componente architetturale cruciale, fungendo da *edge service* e da *reverse proxy* che centralizza l'esposizione delle API e gestisce le preoccupazioni trasversali. Questo approccio risolve il problema *N+1*, dove i client dovrebbero altrimenti effettuare chiamate separate a N servizi backend, aggregando le richieste e semplificando l'interazione. La gestione strategica delle API è, infatti, un fattore critico per le aziende che perseguono la trasformazione digitale, distinguendo i leader di mercato dai ritardatari.

La scelta di Spring Cloud Gateway (SCG) come tecnologia centrale per questo studio è giustificata dalla sua rilevanza all'interno dell'ecosistema Spring, ampiamente adottato nello sviluppo di applicazioni aziendali. SCG è una soluzione leggera e reattiva, costruita su Spring WebFlux e Project Reactor, che la rende particolarmente adatta per carichi di lavoro ad alta produttività e bassa latenza. Il presente Project Work si focalizza sullo studio di questa *nuova tecnologia*, come richiesto dalle linee guida, fornendo dettagli implementativi e un'analisi critica della sua applicazione pratica. La necessità di un punto di ingresso centralizzato per le API è una conseguenza diretta dell'adozione dei microservizi; senza un Gateway API, la gestione delle interazioni tra client e servizi backend, la sicurezza e il monitoraggio diventerebbero estremamente complessi e frammentati, ostacolando l'efficienza e la scalabilità del sistema distribuito.

1.2 Obiettivi del Progetto

L'obiettivo primario di questo progetto è acquisire una comprensione approfondita, implementare e valutare criticamente Spring Cloud Gateway come soluzione per la gestione dei Gateway API in architetture a microservizi. Per raggiungere questo scopo, sono stati definiti i seguenti sotto-obiettivi specifici:

- **Conduzione di un'indagine (Survey):** Effettuare una ricerca sui concetti fondamentali dei Gateway API e sulle loro funzionalità comuni. Questo aspetto si allinea con la tipologia *Survey* del Project Work, che prevede la ricerca dei principali risultati e caratteristiche di una piattaforma tecnologica.
- **Elaborazione sull'architettura e le funzionalità di Spring Cloud Gateway:** Approfondire l'architettura di SCG, le sue caratteristiche distintive come le route, i predicati e i filtri, e i vantaggi che offre in termini di sicurezza, monitoraggio e resilienza. Questo si inquadra nella tipologia *Elaboration*, concentrandosi su un argomento specifico e il suo stato dell'arte.
- **Sviluppo di un'applicazione pratica (Application):** Realizzare un'implementazione concreta di un Gateway API utilizzando Spring Cloud Gateway. Questo prototipo, disponibile nel repository Git fornito dall'utente, dimostrerà l'applicazione pratica delle funzionalità studiate, come richiesto dalla tipologia *Application* delle linee guida, che prevede l'installazione e l'uso di una tecnologia innovativa attraverso esempi semplici.

- **Analisi critica e identificazione delle limitazioni:** Condurre un'analisi obiettiva di Spring Cloud Gateway, identificando i suoi limiti, le potenziali sfide e gli ostacoli che possono emergere durante la sua adozione. Questa componente è fondamentale per soddisfare il requisito di *Critical Thinking* delle linee guida, che invita a non *innamorarsi* delle nuove tecnologie ma a cogliere anche i loro limiti.
- **Proposta di sviluppi futuri:** Sulla base dell'analisi critica, suggerire possibili evoluzioni e miglioramenti per il progetto implementato o per la tecnologia stessa, in linea con il requisito di indicare *quali saranno gli sviluppi futuri che dovranno essere affrontati per ovviare i limiti evidenziati.*

L'esplicita correlazione degli obiettivi del progetto con le tipologie di Project Work definite nelle linee guida (Survey, Elaboration, Application) dimostra una comprensione approfondita dei requisiti dell'incarico. Ciò non si limita alla mera realizzazione tecnica, ma si estende alla capacità di inquadrare il lavoro all'interno di un rigoroso contesto accademico, evidenziando una competenza che va oltre la semplice implementazione.

1.3 Struttura del Rapporto

Il presente rapporto è strutturato per fornire una trattazione completa e progressiva dell'argomento, partendo dai concetti fondamentali fino all'implementazione pratica e all'analisi critica.

- **Capitolo 1: Introduzione** — Definisce il contesto dei Gateway API nelle architetture a microservizi, le motivazioni alla base della scelta di Spring Cloud Gateway e gli obiettivi specifici del progetto.
- **Capitolo 2: Concetti Fondamentali dei Gateway API** — Illustra la definizione e il ruolo di un Gateway API, esplorando le sue funzionalità comuni e i vantaggi architetturali che comporta.
- **Capitolo 3: Spring Cloud Gateway: Caratteristiche e Architettura** — Approfondisce i principi fondamentali di Spring Cloud Gateway, descrivendo i suoi componenti chiave (route, predicati, filtri) e le funzionalità avanzate relative a sicurezza, monitoraggio e resilienza.
- **Capitolo 4: Vantaggi e Casi d'Uso di Spring Cloud Gateway** — Sintetizza i benefici architetturali e operativi derivanti dall'adozione di SCG e presenta scenari applicativi tipici in cui questa tecnologia eccelle.
- **Capitolo 5: Implementazione Pratica: Il Progetto API Gateway** — Descrive in dettaglio l'architettura e il design del Gateway implementato, fornendo esempi concreti di configurazione, codice e dimostrazioni pratiche delle funzionalità.
- **Capitolo 6: Analisi Critica e Limitazioni** — Affronta gli ostacoli e le difficoltà incontrate durante lo studio e l'implementazione, analizzando i limiti di applicabilità di Spring Cloud Gateway e proponendo aree per futuri sviluppi.
- **Capitolo 7: Conclusioni e Sviluppi Futuri** — Riepiloga i risultati principali del progetto, evidenziando i contributi apportati e delineando le prospettive future per la ricerca e l'applicazione di Spring Cloud Gateway.
- **Biblio/Sitografia** — Elenca tutte le fonti bibliografiche e sitografiche utilizzate per la redazione del rapporto.

Questa struttura è stata concepita per guidare il lettore attraverso un percorso logico, dalla teoria alla pratica, fino a una valutazione ponderata, rispondendo in modo esaustivo a tutti i requisiti delle linee guida del Project Work.

Capitolo 2

Concetti Fondamentali dei Gateway API

2.1 Cos'è un API Gateway e il suo Ruolo nelle Architetture a Microservizi

Un API Gateway è un componente architetturale che funge da singolo punto di ingresso per tutte le richieste dei client verso un sistema di microservizi backend. In essenza, agisce come un *reverse proxy* o un *edge service* che si interpone tra i client e i servizi backend.
Questo modello risolve una problematica comune nelle architetture a microservizi, nota come *N+1 problem*, dove un client dovrebbe altrimenti conoscere e interagire direttamente con N servizi backend distinti per soddisfare una singola richiesta complessa. L'API Gateway aggrega queste interazioni, presentando un'unica interfaccia semplificata ai client.

Il ruolo di un API Gateway è multifunzionale e critico per la gestione efficace di sistemi distribuiti. Esso non si limita al semplice routing delle richieste al servizio appropriato, ma centralizza anche una serie di preoccupazioni trasversali che altrimenti dovrebbero essere implementate in ogni singolo microservizio. Questa centralizzazione è fondamentale per mantenere la coerenza, ridurre la ridondanza del codice e accelerare lo sviluppo.

La funzione di un API Gateway come punto di ingresso unificato semplifica intrinsecamente l'integrazione dei client e disaccoppia i client dall'evoluzione dei servizi di backend. Se i client fossero costretti a interagire direttamente con ogni microservizio, dovrebbero essere a conoscenza degli indirizzi specifici, dei protocolli e delle modifiche di ciascuno.
Un Gateway API astrae questa complessità, offrendo un'interfaccia coerente. Questa separazione significa che le modifiche ai servizi di backend (ad esempio, la fusione o la divisione di servizi, o la migrazione di tecnologie) possono essere implementate senza richiedere aggiornamenti sul lato client. Ciò riduce significativamente il sovraccarico di manutenzione e accelera i cicli di sviluppo, poiché i team possono concentrarsi sulla logica di business senza preoccuparsi delle implicazioni per i client.

2.2 Funzionalità Comuni dei Gateway API

Un Gateway API robusto e completo offre una vasta gamma di funzionalità essenziali per la gestione efficiente e sicura delle API in un ambiente a microservizi.
Queste funzionalità sono progettate per affrontare le sfide intrinseche dei sistemi distribuiti e per centralizzare le preoccupazioni trasversali.

Le funzionalità comuni includono:

- **Routing:** La capacità fondamentale di dirigere le richieste in ingresso al servizio backend corretto in base a regole predefinite, come il percorso dell'URL, gli header o i parametri della richiesta.
- **Autenticazione e Autorizzazione:** Applicazione centralizzata delle politiche di sicurezza. Questo include la gestione dell'autenticazione (ad esempio, Basic Auth, OAuth2, OpenID Connect) e dell'autorizzazione (controlli di accesso basati sui ruoli). La funzionalità di Single Sign-On (SSO) è spesso integrata per semplificare l'accesso tra diverse applicazioni.

- **Limitazione del Tasso (Rate Limiting):** Controllo del numero di richieste che un client può effettuare in un determinato periodo di tempo per prevenire abusi, attacchi Denial of Service (DoS) e garantire un uso equo delle risorse.
- **Bilanciamento del Carico (Load Balancing) e Tolleranza ai Guasti (Fault Tolerance):** Distribuzione uniforme del traffico in ingresso tra più istanze di un servizio backend per migliorare le prestazioni e la disponibilità, garantendo che il sistema rimanga reattivo anche in caso di guasti o sovraccarichi di singoli servizi.
- **Monitoraggio e Logging:** Raccolta centralizzata di metriche e log relativi al traffico API, alle prestazioni e agli errori, fornendo visibilità sull'operatività del sistema e facilitando il debugging.
- **Trasformazione di Protocollo e Formato Dati:** Capacità di convertire richieste e risposte tra diversi protocolli (ad esempio, HTTP a gRPC) o formati di dati (ad esempio, JSON a XML), facilitando l'integrazione tra sistemi eterogenei.
- **Versionamento delle API:** Gestione di più versioni di un'API, consentendo agli sviluppatori di introdurre nuove funzionalità o modifiche senza interrompere i client esistenti.
- **Caching:** Memorizzazione di risposte a richieste frequenti per ridurre il carico sui servizi backend e migliorare i tempi di risposta.
- **Circuit Breaker (Interruttore di Circuito):** Un pattern di resilienza che previene i guasti a cascata in un sistema distribuito, isolando i servizi che non rispondono e fornendo risposte di fallback.
- **Header di Sicurezza:** Applicazione automatica di header di sicurezza (ad esempio, Cache-Control, X-Content-Type-Options, Strict-Transport-Security) per rafforzare la postura di sicurezza complessiva.

L'aggregazione di queste preoccupazioni trasversali all'interno del Gateway API le trasforma da responsabilità individuali di ciascun servizio in politiche centralizzate e gestibili.
Questo approccio riduce il carico di sviluppo sui team, che possono concentrarsi sulla creazione di valore di business piuttosto che sulla ricostruzione di componenti infrastrutturali.
Il risultato è un'accelerazione dell'innovazione e del tempo di commercializzazione, oltre a una conformità più efficiente e un'esperienza cliente superiore, derivanti dall'applicazione coerente di best practice di sicurezza e gestione del traffico.

Tabella 2.1: Funzionalità Comuni di un API Gateway (con librerie standard)

Funzionalità	**Descrizione**	**Beneficio Principale**
Routing	Inoltra le richieste in ingresso al servizio backend appropriato.	Semplifica l'integrazione client e la gestione del traffico.
Autenticazione & Autorizzazione	Applica politiche di sicurezza centralizzate per l'accesso alle API.	Migliora la sicurezza e la conformità, riduce il rischio.
Limitazione del Tasso (Rate Limiting)	Controlla il numero di richieste per prevenire abusi e sovraccarichi.	Garantisce stabilità del sistema e equità d'uso.
Bilanciamento del Carico	Distribuisce il traffico tra le istanze del servizio per ottimizzare le prestazioni.	Aumenta la disponibilità e la scalabilità.
Tolleranza ai Guasti	Gestisce i fallimenti dei servizi backend per mantenere la disponibilità.	Aumenta la resilienza del sistema.
Monitoraggio & Logging	Raccoglie dati sul traffico API, le prestazioni e gli errori.	Fornisce visibilità operativa e facilita il debugging.
Trasformazione Protocollo/Dati	Converte richieste/risposte tra diversi formati o protocolli.	Facilita l'integrazione con sistemi eterogenei.
Versionamento API	Gestisce più versioni di un'API contemporaneamente.	Consente evoluzione delle API senza interrompere i client esistenti.
Caching	Memorizza le risposte per richieste frequenti.	Riduce la latenza e il carico sui servizi backend.
Circuit Breaker (Interruttore di Circuito)	Isola i servizi problematici per prevenire guasti a cascata.	Migliora la resilienza del sistema in ambienti distribuiti.
Header di Sicurezza	Applica automaticamente header HTTP per rafforzare la sicurezza.	Migliora la postura di sicurezza complessiva.

Capitolo 3

Spring Cloud Gateway: Caratteristiche e Architettura

3.1 Panoramica e Principi Fondamentali

Spring Cloud Gateway (SCG) si presenta come un Gateway API leggero e reattivo, costruito sopra il framework Spring.
È stato progettato per fornire un metodo semplice ma efficace per instradare le richieste alle API e per gestire le preoccupazioni trasversali quali sicurezza, monitoraggio/metriche e resilienza.
La sua architettura è saldamente radicata su tecnologie moderne dell'ecosistema Spring: Spring Framework 5, Project Reactor e Spring Boot 2.0. Questa fondazione è cruciale, poiché conferisce a SCG la sua natura reattiva e non bloccante, un aspetto distintivo che lo rende particolarmente adatto per gestire un elevato numero di richieste concorrenti con bassa latenza.

La scelta di basare SCG su Project Reactor e Spring WebFlux non è un mero dettaglio tecnico, ma una decisione architetturale fondamentale che lo distingue dai gateway tradizionali basati su modelli di programmazione bloccanti.
I modelli di programmazione reattiva sono intrinsecamente progettati per gestire un gran numero di connessioni concorrenti con un numero ridotto di thread, ottimizzando l'utilizzo delle risorse e migliorando le prestazioni sotto carichi pesanti. Questo è particolarmente vantaggioso in ambienti a microservizi, dove la capacità di gestire efficacemente un flusso elevato di traffico è essenziale per garantire *carichi di lavoro a bassa latenza e alta produttività.*

Il ciclo di vita di una richiesta all'interno di Spring Cloud Gateway segue un flusso ben definito:

1. **Client Request:** Una richiesta viene inviata dal client al Gateway API.
2. **Gateway Handler Mapping:** Il Gateway determina se la richiesta corrisponde a una route definita.
3. **Gateway Web Handler:** Se una corrispondenza viene trovata, la richiesta viene inoltrata a questo handler.
4. **Filter Chain:** L'handler esegue la richiesta attraverso una catena di filtri specifici per quella richiesta. Questi filtri possono modificare la richiesta prima che venga inviata al servizio di destinazione (pre-filtri) o la risposta che ritorna al client (post-filtri).
5. **Downstream Service:** La richiesta modificata viene inviata al servizio backend appropriato.
6. **Response:** La risposta dal servizio backend ritorna attraverso la catena di filtri (post-filtri) prima di essere inviata al client.

Questo modello di elaborazione basato su filtri e route consente una grande flessibilità e modularità nella gestione del traffico API.

3.2 Componenti Chiave: Route, Predicati e Filtri

Spring Cloud Gateway si basa su tre componenti fondamentali per la sua operatività: Route, Predicati e Filtri. La modularità di questi elementi consente una configurazione dichiarativa e un'estensibilità notevole, permettendo di gestire logiche di routing complesse e preoccupazioni trasversali con grande flessibilità.

Route: Una Route è il *blocco di costruzione fondamentale* del gateway. È definita da un ID univoco, un URI di destinazione (il servizio backend a cui la richiesta deve essere inoltrata), una collezione di predicati e una

collezione di filtri. Una route viene considerata *matched* (corrispondente) se l'operazione logica AND su tutti i suoi predicati restituisce true.

```yaml
spring
  cloud
    gateway
      routes
        - id: book-service-route
          uri: lb://BOOK-SERVICE # Nome del servizio utilizzando Eureka
          predicates
            - Path=/api/books/** # Predicato di percorso
          filters
            - StripPrefix=2 # Rimuove /api/books dal percorso prima di inoltrare
```

Listing 3.1: Esempio di configurazione di una route in YAML

Predicate: I Predicati sono funzioni booleane (Java 8 Function Predicate) che permettono di far corrispondere le route a qualsiasi attributo della richiesta HTTP. Questo include header, parametri, metodi HTTP, host, e persino il tempo. È possibile combinare più predicati con operatori logici AND per creare regole di routing altamente specifiche. Esempi comuni di predicati includono: Path, Host, Method, Query, Header, Cookie, RemoteAddr, After, Before, Between. I predicati After, Before e Between sono particolarmente utili per scenari come le finestre di manutenzione, permettendo di instradare il traffico in base a intervalli temporali specifici.

Filter: I Filtri forniscono un meccanismo per modificare le richieste HTTP in ingresso e le risposte HTTP in uscita. Esistono due tipi principali di filtri:

- **GatewayFilter:** Applicati a una specifica route.
- **GlobalFilter:** Applicati a tutte le route. Spring Cloud Gateway include molti filtri predefiniti (ad esempio, AddRequestHeader, AddResponseHeader, RateLimiter, RewritePath). È anche possibile implementare filtri personalizzati per esigenze di business uniche. Ad esempio, un filtro personalizzato potrebbe essere utilizzato per aggiungere un timestamp alla richiesta in ingresso prima che venga inoltrata (AddRequestTimeHeaderPreFilter) o per simulare una validazione e bloccare la richiesta se un header di autorizzazione non è presente.

L'enfasi ricorrente su Route, Predicati e Filtri sottolinea la loro centralità nel potere di Spring Cloud Gateway. La capacità di definire le route in modo dichiarativo, di combinare i predicati per una corrispondenza granulare e di applicare i filtri per la modifica delle richieste/risposte significa che le politiche di gestione delle API, anche le più complesse, possono essere costruite a partire da blocchi più piccoli e riutilizzabili. Questa modularità rende il gateway altamente configurabile ed estensibile, consentendo agli sviluppatori di aggiungere facilmente nuove funzionalità o di implementare filtri personalizzati per requisiti specifici. Questo principio di design è fondamentale per la sua adattabilità in diversi ambienti a microservizi.

3.3 Funzionalità Avanzate: Sicurezza, Monitoraggio, Resilienza

Oltre alle capacità fondamentali di routing e trasformazione, Spring Cloud Gateway eccelle nella gestione di funzionalità avanzate che sono cruciali per la robustezza e la sicurezza delle architetture a microservizi. Queste includono la sicurezza centralizzata, il monitoraggio e le metriche, e la resilienza del sistema.

Sicurezza: SCG centralizza le preoccupazioni di sicurezza, riducendo la necessità di implementare logiche di sicurezza in ogni singolo microservizio. Questo include:

- *Autenticazione e Autorizzazione:* Supporta vari meccanismi di autenticazione come Basic Auth e OAuth2/OpenID Connect. Le funzionalità di Single Sign-On (SSO) sono particolarmente rilevanti, in quanto possono essere configurate una sola volta a livello di gateway, eliminando la necessità di implementazioni diverse per ogni sistema backend. Ciò consente un controllo degli accessi basato sui ruoli e semplifica notevolmente la conformità normativa, riducendo i costi di conformità fino al 40% per le organizzazioni con controlli di sicurezza centralizzati.
- *Header di Sicurezza:* L'applicazione automatica di header di sicurezza (es. Cache-Control, X-Content-Type-Options, Strict-Transport-Security) a livello globale per tutte le route è una best practice che rafforza la postura di sicurezza complessiva.
- *Gestione Globale delle Eccezioni:* Implementare una gestione globale delle eccezioni in SCG è essenziale per catturare e gestire in modo uniforme gli errori che si verificano all'interno del gateway. Questo previene risposte di errore frammentate e incoerenti, migliorando l'esperienza utente e facilitando il debugging.

Tabella 3.1: Esempi di Predicati di Route in Spring Cloud Gateway

Predicato	Descrizione	Parametri Esempio	Esempio di Configurazione
Path	Corrisponde al percorso dell'URL della richiesta.	`/api/users/**`	`predicates: - Path=/api/users/**`
Host	Corrisponde all'header Host della richiesta.	`*.example.com`	`predicates: - Host=*.example.com`
Method	Corrisponde al metodo HTTP della richiesta.	GET, POST	`predicates: - Method=GET,POST`
Query	Corrisponde a un parametro di query specifico.	param=value	`predicates: - Query=param,value`
Header	Corrisponde a un header HTTP specifico e al suo valore.	`X-Request-Id, d+`	`predicates: - Header=X-Request-Id, d+`
After	Corrisponde alle richieste che avvengono dopo una data e ora specificate.	2017-01-20T...	`predicates: - After=2017-01-20T17:42:47.789Z[UTC]`
Before	Corrisponde alle richieste che avvengono prima di una data e ora specificate.	2017-01-21T...	`predicates: - Before=2017-01-21T17:42:47.789Z[UTC]`
Between	Corrisponde alle richieste che avvengono tra due date e ore specificate.	datetime1, datetime2	`predicates: - Between=...,` `...`
RemoteAddr	Corrisponde all'indirizzo IP remoto della richiesta.	192.168.1.1/24	`predicates: - RemoteAddr=192.168.1.1/24`

Monitoraggio/Metriche: SCG può essere integrato con strumenti di monitoraggio e logging esterni come Splunk o ELK Stack. Questa integrazione fornisce una visibilità centralizzata sul traffico API, sulle prestazioni e sugli errori. La capacità di registrare ogni eccezione per analisi future è una best practice fondamentale.

Resilienza: La resilienza è vitale in ambienti distribuiti per prevenire i guasti a cascata. SCG offre funzionalità come:

- *Integrazione con Circuit Breaker:* L'integrazione con circuit breaker come Hystrix (o Resilience4j in versioni più recenti) consente di isolare i servizi che non rispondono, evitando che un singolo guasto si propaghi attraverso l'intero sistema.
- *Gestione Intelligente del Traffico e Tolleranza ai Guasti:* SCG offre funzionalità avanzate di gestione del traffico e alta disponibilità che possono mantenere prestazioni consistenti anche durante i picchi di traffico. Questo include il bilanciamento del carico e la gestione dei fallimenti.

La centralizzazione di funzionalità avanzate come la sicurezza e la resilienza a livello di gateway le trasforma da correzioni reattive in salvaguardie architetturali proattive. Questo approccio migliora significativamente la robustezza complessiva del sistema e la sua postura di conformità. Gestendo questi aspetti in modo centralizzato, il sistema diventa più resistente ai guasti e agli attacchi, e la conformità normativa è più facile da gestire, riducendo potenziali sanzioni. Questo non si limita alla mera funzionalità, ma si estende all'impatto strategico di tali funzionalità sull'integrità operativa del sistema e sui risultati di business.

Tabella 3.2: Esempi di Filtri in Spring Cloud Gateway

Filtro	**Tipo**	**Descrizione**	**Caso d'Uso Esempio**
AddRequestHeader	GatewayFilter	Aggiunge un header alla richiesta in uscita.	`filters: - AddRequestHeader=X-Request-Foo, Bar`
AddResponseHeader	GatewayFilter	Aggiunge un header alla risposta in uscita.	`filters: - AddResponseHeader=X-Response-Bye, Bye`
StripPrefix	GatewayFilter	Rimuove un prefisso dal percorso della richiesta.	`filters: - StripPrefix=1` per /api/v1/users → /users
RewritePath	GatewayFilter	Riscrive il percorso della richiesta usando un'espressione regolare.	`filters: - RewritePath=/foo/(?<segment>.*), /${segment}`
RequestRateLimiter	GatewayFilter	Limita il tasso di richieste per utente/IP.	`filters: - RequestRateLimiter=` (con config.)
CircuitBreaker	GatewayFilter	Implementa un pattern Circuit Breaker per la resilienza.	`filters: - CircuitBreaker=myServiceCircuit`
SecureHeaders	GlobalFilter	Applica globalmente header di sicurezza HTTP.	`tanzu: api-gateway: secure-headers: deactivated: false`
CustomGlobal-ExceptionHandler	GlobalFilter	Gestisce eccezioni a livello globale per risposte uniformi.	Intercetta HttpClientErrorException per risposte di errore consistenti
Authorization Filter	GlobalFilter	Filtro personalizzato per autenticazione/autorizzazione.	Verifica token OAuth2 o credenziali Basic Auth
AddRequestTime-HeaderPreFilter	GlobalFilter	Filtro personalizzato che aggiunge un timestamp alla richiesta.	`filters: - AddRequestTimeHeaderPreFilter`

Capitolo 4

Vantaggi e Casi d'Uso di Spring Cloud Gateway

4.1 Benefici Architetturali e Operativi

L'adozione di Spring Cloud Gateway in un'architettura a microservizi porta a una serie di benefici significativi, sia a livello architetturale che operativo. Questi vantaggi non sono solo tecnici, ma si traducono direttamente in risultati di business misurabili.

- **Integrazione Client Semplificata:** I client interagiscono con un unico endpoint esposto dal Gateway API, semplificando la loro logica applicativa e riducendo la complessità di dover conoscere e gestire più servizi backend.
- **Sicurezza Migliorata:** La centralizzazione delle politiche di sicurezza, inclusa l'autenticazione, l'autorizzazione e l'applicazione di header di sicurezza, riduce il rischio di incidenti di sicurezza e i costi di conformità. Le organizzazioni con pratiche mature di sicurezza API registrano il 60% in meno di incidenti di sicurezza.
- **Innovazione Accelerata e Tempo di Commercializzazione Ridotto:** I team di sviluppo possono concentrarsi sulla logica di business principale piuttosto che sulla ricostruzione di componenti infrastrutturali come la sicurezza o la gestione del traffico. La ricerca McKinsey indica che le organizzazioni con una gestione API semplificata rilasciano nuove funzionalità 2-3 volte più velocemente della concorrenza.
- **Prestazioni e Scalabilità Migliorate:** L'architettura reattiva e non bloccante di SCG, basata su Spring WebFlux e Project Reactor, consente di gestire un gran numero di richieste concorrenti con un utilizzo efficiente delle risorse. Funzionalità come la gestione intelligente del traffico, il bilanciamento del carico e la tolleranza ai guasti contribuiscono a mantenere prestazioni elevate. SCG ha dimostrato risultati impressionanti, con tempi di risposta API fino a 6 volte più veloci, latenza ridotta fino al 50% e oltre l'80% di riduzione dei tassi di errore rispetto a opzioni concorrenti.
- **Migliore Monitoraggio e Visibilità:** La centralizzazione dei log e delle metriche per le preoccupazioni trasversali fornisce una visibilità completa sul comportamento delle API e facilita l'identificazione e la risoluzione dei problemi.
- **Conformità Cost-Effective:** I controlli di sicurezza centralizzati consentono di ridurre i costi di conformità fino al 40%. SCG implementa questi controlli in modo coerente, creando audit trail e applicando politiche di sicurezza con un overhead minimo.
- **Versionamento API e Compatibilità Retroattiva:** SCG può gestire più versioni di un'API, consentendo agli sviluppatori di introdurre nuove funzionalità o apportare modifiche senza interrompere i client esistenti, garantendo una transizione più fluida e riducendo il rischio di interruzioni del servizio.

I vantaggi tecnici di Spring Cloud Gateway si traducono direttamente in risultati di business tangibili, come la riduzione dei costi operativi, una risposta più rapida al mercato e un'esperienza cliente superiore.
Questo collegamento tra capacità tecniche e obiettivi strategici di business è fondamentale per comprendere il valore di SCG, in linea con il requisito di spiegare *quali vantaggi produce questa tecnologia in quali settori applicativi.*

4.2 Scenari Applicativi Tipici

La versatilità di Spring Cloud Gateway lo rende idoneo per un'ampia gamma di scenari applicativi, affrontando diverse sfide architetturali in contesti moderni e legacy.

- **Orchestrazione di Microservizi:** Il caso d'uso più comune è l'orchestrazione del traffico in ingresso verso un'architettura a microservizi. SCG agisce come il punto di ingresso che instrada le richieste ai vari servizi backend, gestendo le preoccupazioni trasversali in modo centralizzato.
- **Integrazione di Sistemi Legacy:** SCG può fungere da ponte tra sistemi legacy e applicazioni moderne. Grazie alle sue capacità di trasformazione di protocollo e formato dati (ad esempio, da JSON a XML o da HTTP a gRPC), può facilitare la comunicazione e l'integrazione di servizi più datati con nuove applicazioni basate su microservizi.
- **Mobile Backend for Frontend (BFF):** In scenari dove diverse tipologie di client (es. web, mobile) necessitano di API ottimizzate per le loro esigenze specifiche, SCG può essere configurato per fornire un *Backend for Frontend* (BFF), esponendo API su misura per ogni client.
- **Esposizione di API Pubbliche:** Per le aziende che desiderano esporre le proprie API a partner esterni o sviluppatori, SCG offre un robusto strato di sicurezza e gestione del traffico, proteggendo i servizi interni e fornendo un'interfaccia controllata e monitorata.
- **Aggregazione Dati:** In alcuni casi, una singola richiesta client potrebbe necessitare di dati provenienti da più microservizi. SCG può essere configurato per aggregare le risposte da diversi servizi backend in una singola risposta consolidata prima di inviarla al client.
- **Applicazioni Real-time:** Sebbene SCG sia reattivo e adatto per carichi di lavoro ad alta concorrenza, è importante riconoscere alcune limitazioni per esigenze real-time molto specifiche, come la diffusione di messaggi in broadcast a un gran numero di client WebSocket. Per tali scenari, potrebbero essere necessarie soluzioni complementari o architetture più complesse.

La vasta applicabilità di Spring Cloud Gateway gli consente di affrontare un'ampia gamma di sfide architetturali contemporanee, dall'implementazione di nuovi microservizi all'integrazione di sistemi legacy.
Questo lo posiziona come un componente strategico per la trasformazione digitale, non solo come strumento di nicchia ma come elemento fondamentale per la costruzione di piattaforme digitali adattabili e scalabili, in linea con la tipologia di progetto *applicazione o processo innovativo*.

Capitolo 5

Implementazione Pratica: Il Progetto API Gateway

Questa sezione descrive l'implementazione pratica di un Gateway API utilizzando Spring Cloud Gateway, come dimostrato nel repository Git fornito (`https://github.com/ddonazz/api-gateway`).
L'obiettivo è illustrare come i concetti teorici discussi nei capitoli precedenti siano stati tradotti in una soluzione funzionante, evidenziando le scelte di design e i dettagli implementativi.

5.1 Architettura e Design del Gateway Implementato

L'architettura del sistema implementato è composta da un servizio Spring Cloud Gateway che funge da punto di ingresso centrale e da uno o più semplici microservizi backend.
Per scopi dimostrativi, è stato configurato un servizio di backend minimale (ad esempio, un `product-service` o un `user-service` come si vede in progetti simili), che espone alcune API REST.

Il flusso delle richieste è il seguente:

- Un client (ad esempio, un browser o un'applicazione mobile) invia una richiesta HTTP al Gateway API.
- Il Gateway API, basato su Spring Cloud Gateway, intercetta la richiesta.
- Utilizzando le route e i predicati configurati, il Gateway determina il microservizio backend appropriato a cui inoltrare la richiesta.
- I filtri configurati (globali o specifici per la route) vengono applicati per modificare la richiesta in ingresso o la risposta in uscita.
- La richiesta viene inoltrata al microservizio backend.
- Il microservizio elabora la richiesta e restituisce una risposta al Gateway.
- Il Gateway applica eventuali filtri post-elaborazione alla risposta prima di inviarla al client.

Le scelte di design hanno privilegiato una configurazione dichiarativa delle route e dei filtri tramite file `application.yml`, sebbene Spring Cloud Gateway supporti anche la configurazione programmatica in Java.
Per la scoperta dei servizi è stata utilizzato un server Eureka. L'ambiente di sviluppo è stato avviato utilizzando Spring Initializr, che facilita la creazione di progetti Spring Boot con le dipendenze necessarie, supportando sia Gradle che Maven.

Una chiara rappresentazione dell'architettura e delle scelte di design dimostra non solo la capacità tecnica, ma anche una profonda comprensione dei principi architetturali, collegando efficacemente la teoria (Capitoli 2 e 3) con la pratica.
La selezione di servizi backend semplici ma rappresentativi consente di illustrare in modo efficace le capacità di routing e filtraggio del gateway.

5.2 Dettagli Implementativi e Configurazione

Questa sezione fornisce dettagli specifici sull'implementazione delle funzionalità chiave all'interno del progetto API Gateway, con esempi di configurazione e codice.

5.3 Definizione delle Route e dei Predicati

5.3.1 Eureka e la Scoperta dei Servizi

La scoperta dei servizi su un server **Eureka** semplifica l'individuazione e l'interazione tra i microservizi. Invece di configurare manualmente gli indirizzi di rete per ogni servizio, i servizi si registrano con Eureka, che mantiene un registro aggiornato delle istanze di servizio disponibili. Quando un client necessita di comunicare con un servizio specifico, interroga Eureka per ottenere la posizione attuale di un'istanza sana di quel servizio.

Le route, come quelle definite nel file `application.yml` per chiarezza e facilità di configurazione, sfruttano questa capacità di scoperta.

Ogni route specifica un **ID** univoco, l'**URI** del servizio di destinazione (spesso un nome logico registrato con Eureka, ad esempio `lb://NOME-DEL-SERVIZIO`), e una serie di **predicati** per la corrispondenza delle richieste in entrata. Questi predicati determinano se una richiesta deve essere instradata a un particolare servizio. Ad esempio, una route potrebbe specificare che tutte le richieste che iniziano con `/api/utenti` devono essere indirizzate al servizio `lb://SERVIZIO-UTENTI`. Questo approccio disaccoppia i client dalle posizioni fisiche dei servizi, migliorando la resilienza e la manutenibilità dell'architettura a microservizi.

Un esempio di configurazione di una route in YAML è il seguente:

```yaml
spring
  cloud
    gateway
      routes
        - id: user_service_route
          uri: http://localhost:8082 # Indirizzo del microservizio utente
          predicates
            - Path=/api/users/**
            - Method=GET,POST
          filters
            - StripPrefix=1 # Rimuove '/api' dal percorso prima dell'inoltro
        - id: product_service_route
          uri: http://localhost:8083 # Indirizzo del microservizio prodotto
          predicates
            - Path=/api/products/**
            - Header=X-Version, v2 # Richiede un header specifico per questa route
          filters
            - AddRequestHeader=X-Forwarded-By, ApiGateway
```

Listing 5.1: Configurazione delle rotte

In questo esempio, la route `user_service_route` inoltra le richieste GET e POST con percorso `/api/users/**` al servizio utente, rimuovendo il prefisso `/api`.
La route `product_service_route` inoltra le richieste con percorso `/api/products/**` e un header `X-Version` con valore `v2` al servizio prodotto, aggiungendo un header `X-Forwarded-By`.
L'uso di predicati come Path, Method e Header dimostra la flessibilità nella definizione delle regole di routing.

5.4 Implementazione dei Filtri Personalizzati

Sono stati implementati filtri personalizzati per dimostrare la capacità di estensione di Spring Cloud Gateway oltre i filtri predefiniti.

```java
@Component
public class AuthorizationHeaderGatewayFilterFactory extends
    ↩ AbstractGatewayFilterFactory<AuthorizationHeaderGatewayFilterFactory.Config> {

    private static final Logger LOG = LoggerFactory.getLogger(AuthorizationHeaderGatewayFilterFactory.class);

    private final ValidateJwtUtil jwtUtil;
    private final RouterValidator routerValidator;

    public AuthorizationHeaderGatewayFilterFactory(ValidateJwtUtil jwtUtil, RouterValidator routerValidator) {
        super(Config.class);
        this.jwtUtil = jwtUtil;
        this.routerValidator = routerValidator;
    }

    // La classe Config e' vuota perche' le dipendenze sono iniettate nella factory,
    // e non ci sono parametri specifici da passare tramite YAML a questo filtro.
    public static class Config {
        // Nessun campo necessario qui per la tua logica attuale
    }
```

```
20
21      @Override
22      public GatewayFilter apply(Config config) {
23          return (exchange, chain) → {
24              ServerHttpRequest request = exchange.getRequest();
25
26              if (routerValidator.isSecured.test(request)) {
27                  if (!request.getHeaders().containsKey(HttpHeaders.AUTHORIZATION)) {
28                      ServerHttpResponse response = exchange.getResponse();
29                      response.setStatusCode(HttpStatus.UNAUTHORIZED);
30                      LOG.warn("Missing Authorization header for secured route.");
31                      return response.setComplete();
32                  }
33
34                  String authHeader = request.getHeaders().getOrEmpty(HttpHeaders.AUTHORIZATION).get(0);
35                  String token = authHeader.replace("Bearer ", "");
36
37                  try {
38                      jwtUtil.validateToken(token);
39                      Claims claims = jwtUtil.extractAllClaims(token);
40
41                      List<?> rawAuthorities = claims.get("authorities", List.class);
42                      if (rawAuthorities != null) {
43                          List<String> authorities = rawAuthorities.stream()
44                                  .map(Object::toString)
45                                  .toList();
46
47                          ServerHttpRequest modifiedRequest = request.mutate()
48                                  .header("X-Auth-User", claims.getSubject())
49                                  .header("X-Auth-Roles", String.join(",", authorities))
50                                  .build();
51
52                          LOG.info("AuthorizationHeaderFilter applied. User: {}", claims.getSubject());
53                          return chain.filter(exchange.mutate().request(modifiedRequest).build());
54                      }
55                  } catch (Exception e) {
56                      ServerHttpResponse response = exchange.getResponse();
57                      response.setStatusCode(HttpStatus.UNAUTHORIZED);
58                      LOG.warn("Invalid Authorization header: {}", e.getMessage());
59                      return response.setComplete();
60                  }
61              }
62              return chain.filter(exchange);
63          };
64      }
65
66  }
```

Listing 5.2: Esempio di GatewayFilter per la validazione di un header di autorizzazione

Questo filtro personalizzato (simile all'esempio di validazione precedente) verifica la presenza dell'header `Authorization` e, in caso di assenza, genera un errore 401 (Unauthorized).

5.4.1 Sicurezza Delegata nel Microservizio

In un'architettura a microservizi, la gestione della sicurezza è spesso centralizzata a livello di API Gateway. Il Gateway ha il compito di autenticare le richieste provenienti dai client esterni (ad esempio, validando un token JWT) e, una volta verificata l'identità dell'utente, arricchisce la richiesta inoltrata ai servizi interni con le informazioni necessarie per l'autorizzazione.

Questo capitolo descrive l'implementazione di un filtro di sicurezza all'interno di un singolo microservizio, sviluppato con Spring WebFlux, che opera secondo questo principio di *autenticazione delegata*. Il servizio non esegue una nuova autenticazione, ma si fida delle informazioni (come utente e ruoli) inserite negli header HTTP dal Gateway.

Configurazione della Catena di Filtri di Sicurezza

Il cuore della configurazione di sicurezza risiede nella classe `SecurityConfig`. Grazie alle annotazioni `@EnableWebFluxSecurity` e `@EnableReactiveMethodSecurity`, attiviamo il supporto di Spring Security per applicazioni reattive e per la sicurezza a livello di metodo.

Il bean `SecurityWebFilterChain` definisce la catena di filtri che ogni richiesta deve attraversare.

```
1  @Configuration
```

```java
@EnableWebFluxSecurity
@EnableReactiveMethodSecurity
public class SecurityConfig {

    @Bean
    SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {
        AuthenticationWebFilter authenticationWebFilter = new
        ↩ AuthenticationWebFilter(getAuthenticationManager());
        authenticationWebFilter.setServerAuthenticationConverter(new HeaderAuthenticationConverter());

        return http
                .authorizeExchange(exchanges → exchanges
                        .anyExchange().authenticated()
                )
                .addFilterAt(authenticationWebFilter, SecurityWebFiltersOrder.AUTHENTICATION)
                .httpBasic(ServerHttpSecurity.HttpBasicSpec::disable)
                .formLogin(ServerHttpSecurity.FormLoginSpec::disable)
                .csrf(ServerHttpSecurity.CsrfSpec::disable)
                .logout(ServerHttpSecurity.LogoutSpec::disable)
                .build();
    }

    /**
     * Un AuthenticationManager che si fida dell'autenticazione creata dal converter.
     * Dato che il nostro converter crea un token gia' "autenticato" (con ruoli),
     * questo manager non deve fare complesse validazioni.
     * Restituiamo un manager che accetta semplicemente l'oggetto.
     */
    private ReactiveAuthenticationManager getAuthenticationManager() {
        return Mono::just;
    }

}
```

Listing 5.3: Definizione della catena di filtri di sicurezza in `SecurityConfig.java`.

I punti salienti di questa configurazione sono:

- **Disabilitazione dei meccanismi standard**: Le funzionalità di login basate su form (`formLogin`), autenticazione HTTP Basic (`httpBasic`), protezione CSRF (`csrf`) e logout sono disabilitate. Questo perché il servizio non gestisce sessioni utente tradizionali, ma opera in modo *stateless*, processando ogni richiesta in base alle informazioni che essa contiene.
- **Autorizzazione predefinita**: Con `anyExchange().authenticated()`, si stabilisce che ogni richiesta, a qualsiasi endpoint, deve essere autenticata per poter essere processata.
- **Filtro personalizzato**: Viene creato e inserito un `AuthenticationWebFilter`. Questo filtro è il componente chiave che orchestra il processo di estrazione dell'identità. Viene posizionato esattamente al punto della catena in cui Spring Security si aspetta che avvenga l'autenticazione (`SecurityWebFiltersOrder.AUTHENTICATION`).

5.4.2 Estrazione dell'Identità dagli Header HTTP

Il compito di leggere gli header HTTP e tradurli in un oggetto `Authentication` comprensibile a Spring Security è delegato alla classe `HeaderAuthenticationConverter`.

```java
public class HeaderAuthenticationConverter implements ServerAuthenticationConverter {

  private static final String USER_HEADER = "X-Auth-User";
    private static final String ROLES_HEADER = "X-Auth-Roles";

    @Override
    public Mono<Authentication> convert(ServerWebExchange exchange) {
        HttpHeaders headers = exchange.getRequest().getHeaders();

        String username = headers.getFirst(USER_HEADER);
        String rolesHeader = headers.getFirst(ROLES_HEADER);

        if (username == null || rolesHeader == null) {
            return Mono.empty();
        }

        List<SimpleGrantedAuthority> authorities = Arrays.stream(rolesHeader.split(","))
                .map(String::trim)
                .filter(role → !role.isEmpty())
                .map(SimpleGrantedAuthority::new)
```

```
                .toList();

        Authentication authentication = new UsernamePasswordAuthenticationToken(username, null, authorities);

        return Mono.just(authentication);
    }

}
```

Listing 5.4: Implementazione di `HeaderAuthenticationConverter.java`.

Questo convertitore implementa una logica semplice ma efficace:

1. Cerca due header specifici nella richiesta in arrivo: `X-Auth-User` per l'identificativo dell'utente e `X-Auth-Roles` per i suoi ruoli, separati da virgola.
2. Se gli header non sono presenti, restituisce un `Mono.empty()`, segnalando che non è stato possibile costruire un'identità per la richiesta corrente.
3. Se gli header sono presenti, costruisce un oggetto `UsernamePasswordAuthenticationToken`, che è un'implementazione standard di `Authentication`. Questo oggetto contiene il principal (l'utente), le credenziali (impostate a `null` perché non pertinenti in questo contesto) e le autorità (i ruoli).

È fondamentale notare che questo componente si fida implicitamente del fatto che gli header siano stati inseriti da un'entità affidabile (l'API Gateway) e che non siano stati manomessi.

5.4.3 Autorizzazione Basata su Ruoli

Una volta che il `SecurityContext` è stato popolato con l'oggetto `Authentication`, il microservizio può implementare logiche di autorizzazione granulari a livello di singolo endpoint. Questo viene realizzato tramite l'annotazione `@PreAuthorize`, che sfrutta Spring Expression Language (SpEL) per definire le regole di accesso.

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

  @GetMapping("/who-am-i")
  public Mono<String> whoAmI() {
    return Mono.just("Hello! You are an authenticated user.");
  }

  @GetMapping("/admin-dashboard")
  @PreAuthorize("hasRole('ADMIN')")
  public Mono<String> getAdminDashboard() {
    return Mono.just("Welcome to the Admin Dashboard!");
  }

  @GetMapping("/user-profile")
  @PreAuthorize("hasAnyRole('USER', 'ADMIN')")
  public Mono<String> getUserProfile() {
    return Mono.just("Welcome to your User Profile!");
  }

  @GetMapping("/authenticated-user-info")
  public Mono<String> authenticatedUserInfo(Mono<Principal> principal) {
    return principal.map(Principal::getName);
  }

}
```

Listing 5.5: Esempi di autorizzazione nel `ProductController.java`.

Come mostrato nel `ProductController` (listato 5.5):

- L'endpoint `/admin-dashboard` è accessibile solo agli utenti che possiedono il ruolo `'ADMIN'`.
- L'endpoint `/user-profile` è accessibile sia agli utenti con ruolo `'USER'` che `'ADMIN'`.
- È inoltre possibile accedere programmaticamente all'identità dell'utente autenticato iniettando un `Mono<Principal>` nel metodo del controller, come dimostra l'endpoint `/authenticated-user-info`.

Questo approccio crea un sistema di sicurezza efficiente e disaccoppiato, dove il microservizio delega l'onere dell'autenticazione primaria e si concentra esclusivamente sulla logica di autorizzazione basata sulle informazioni affidabili fornite dal Gateway. //

5.5 Altre Funzionalità Implementate

- **Limitazione del Tasso (Rate Limiting):** È stata configurata la limitazione del tasso per alcune route, utilizzando il filtro `RequestRateLimiter` integrato di Spring Cloud Gateway. Questo filtro consente di definire quante richieste un utente (identificato per IP attraverso la condifuirazione di un bean *ipKeyResolver*) può effettuare in un dato periodo di tempo.

```yaml
spring
  application
    name: api-gateway
  cloud
    gateway
      server
        webflux
           routes
             - id: authentication_service
               uri: lb://AUTHENTICATION-SERVICE
               predicates
                 - Path=/auth/**
               filters
                 - name: RequestRateLimiter
                   args
                     key-resolver: '#{@ipKeyResolver}'
                     redis-rate-limiter.replenishRate: 5
                     redis-rate-limiter.burstCapacity: 10
                     redis-rate-limiter.requestedTokens: 1
```

Listing 5.6: Configurazione YAML per Rate Limiting

- **Integrazione Circuit Breaker:** Sebbene non esplicitamente implementato nel repository fornito dall'utente, un'integrazione con un circuit breaker (es. Resilience4j, successore di Hystrix) sarebbe configurata per proteggere le route da servizi backend lenti o non disponibili.
- **Logging e Monitoraggio:** La configurazione di logging standard di Spring Boot è stata utilizzata per monitorare il traffico e gli errori. Per un sistema di produzione, si integrerebbe con soluzioni esterne come Splunk o ELK Stack per un monitoraggio più avanzato.

Tabella 5.1: Funzionalità Implementate nel Progetto API Gateway

Funzionalità	**Descrizione dell'Implementazione**	**Riferimento Codice/Configurazione**	**Endpoint di Test/Esempio**
Routing Basico	Route configurate per inoltrare richieste a servizi utente e prodotto.	`application.yml` (route `user_service_route`, `product_service_route`)	`GET /api/users/1`, `GET /api/products/abc`
Predicati Multipli	Uso combinato di Path e Method, e Path e Header.	`application.yml` (route `user_service_route`, `product_service_route`)	`GET /api/users/1`, `GET /api/products/abc` con `X-Version: v2`
Filtro Personalizzato (Logging)	GlobalFilter per registrare il tempo di elaborazione di ogni richiesta.	`RequestTimeLoggingFilter`	Tutte le richieste al gateway (output console)
Filtro Personalizzato (Validazione Header)	GatewayFilter per verificare la presenza dell'header Authorization.	`AuthorizationHeaderFilterFactory`	`GET /api/protected/resource` (senza header Authorization → 401)
Limitazione del Tasso	Configurazione del filtro RequestRateLimiter per una route pubblica.	`application.yml` (route `public_api_rate_limited`)	Richieste multiple a `GET /api/public/data` (superando il limite → 429 Too Many Requests)

La dettagliata implementazione di funzionalità specifiche, come i filtri personalizzati e la gestione della sicurezza, dimostra l'estensibilità e l'adattabilità di Spring Cloud Gateway oltre le sue capacità predefinite. Ciò evidenzia un livello più profondo di padronanza, andando oltre una semplice indagine (*Survey*) per abbracciare pienamente gli aspetti di applicazione ed elaborazione (*Application* ed *Elaboration*) del progetto. Gli esempi specifici di configurazione e codice fungono da prova concreta del lavoro svolto e della comprensione dello studente.

5.6 Esempi di Utilizzo e Dimostrazione Pratica

Per validare il corretto funzionamento dell'API Gateway e l'integrazione con i microservizi sottostanti, è stato predisposto uno script di test automatizzato. Questo script, basato su comandi `curl` e `jq`, simula le

interazioni di un client, verificando sistematicamente le diverse funzionalità implementate, come l'autenticazione, l'autorizzazione basata su ruoli, il rate limiting e i predicati di routing.

Si assume che il Gateway sia in ascolto sulla porta `8080` e che tutti i microservizi (autenticazione, prodotti, utenti) siano attivi e raggiungibili dal Gateway.

- **Esecuzione dello script di test:** Per avviare la dimostrazione, è sufficiente eseguire lo script Bash, che si occuperà di orchestrare tutte le chiamate API e di riportare l'esito di ogni test.
- **Sezione 1: Servizio di Autenticazione e Generazione Token**
  - *Richiesta:* Ottenere i token JWT per un utente standard (`user`) e un amministratore (`admin`).
  - *Logica di Test:* Lo script invia due richieste `POST` all'endpoint `/auth/login` con le rispettive credenziali. I token restituiti vengono salvati per i test successivi. Viene inoltre testato un tentativo di login con password errata.
  - *Risultato Atteso:* I primi due login hanno successo (stato HTTP 200) e forniscono un token valido. Il login con credenziali errate fallisce, restituendo un errore (nello script, si attende un 500 Internal Server Error, gestito dal servizio di autenticazione).
- **Sezione 2: Dimostrazione del Rate Limiter**
  - *Richiesta:* Sovraccaricare un endpoint per attivare il limite di richieste.
  - *Logica di Test:* Lo script esegue un ciclo che invia 15 richieste consecutive e in rapida successione all'endpoint `/auth/login`.
  - *Risultato Atteso:* Le prime richieste (fino al limite configurato, ad esempio 10 al secondo) ricevono uno stato HTTP 200. Superata la soglia, il Gateway risponde con uno stato HTTP 429 Too Many Requests, bloccando le richieste in eccesso senza inoltrarle al servizio sottostante.
- **Sezione 3: Verifica dell'Accesso Protetto al Product Service**
  - *Richiesta:* Accedere a endpoint con diverse regole di autorizzazione.
  - *Logica di Test:* Vengono eseguiti molteplici test:
    1. Accesso senza token: fallisce con stato 401 Unauthorized.
    2. Accesso con token valido ma senza l'header `X-Version: v2` richiesto dalla rotta: il Gateway non trova una rotta corrispondente e il test fallisce.
    3. Utente `user` accede a una risorsa per utenti (`/api/products/user-profile`): successo (200 OK).
    4. Utente `user` tenta di accedere a una risorsa per amministratori (`/api/products/admin-dashboard`): accesso negato (403 Forbidden).
    5. Utente `admin` accede sia alla risorsa per amministratori che a quella per utenti: successo in entrambi i casi (200 OK).
  - *Risultato Atteso:* I test dimostrano che il Gateway valida correttamente la presenza e la validità del token JWT, applica i predicati sugli header (`X-Version`) e inoltra gli header di autenticazione (`X-Auth-User`, `X-Auth-Roles`) al servizio `product-service`, che a sua volta applica l'autorizzazione basata sui ruoli.
- **Sezione 4: Verifica dei Predicati di Metodo sul User Service**
  - *Richiesta:* Accedere a un endpoint con un metodo HTTP consentito e uno non consentito.
  - *Logica di Test:* La rotta per `/api/users/**` è configurata per accettare solo richieste `GET`. Lo script tenta prima una richiesta `GET` e poi una `DELETE`.
  - *Risultato Atteso:* La richiesta `GET` viene inoltrata correttamente, risultando in uno stato 200 (se la risorsa esiste) o 404 (se non esiste). La richiesta `DELETE`, invece, non corrisponde a nessuna rotta configurata e viene respinta dal Gateway con uno stato 404 Not Found, senza mai raggiungere il microservizio.

Questi test pratici, orchestrati dallo script, confermano che l'API Gateway implementato agisce come un punto di controllo centralizzato, gestendo in modo efficace la sicurezza, il routing avanzato e la protezione dei servizi backend, soddisfacendo così i requisiti del progetto.

Capitolo 6

Analisi Critica e Limitazioni

L'adozione di qualsiasi tecnologia, per quanto promettente, richiede un'analisi critica che vada oltre i soli vantaggi, esplorando anche gli ostacoli, le difficoltà e i limiti di applicabilità. Questa sezione, un requisito fondamentale del "Critical Thinking", mira a fornire una prospettiva equilibrata su Spring Cloud Gateway e sui Gateway API in generale.

6.1 Ostacoli e Difficoltà Incontrate

L'introduzione di un API Gateway, sebbene benefica, non è priva di sfide e può introdurre nuove complessità nell'architettura di un sistema.

- **Complessità Aggiuntiva:** L'API Gateway introduce un ulteriore strato nell'architettura, che gli sviluppatori devono comprendere e gestire. Questo richiede conoscenze, competenze e strumenti aggiuntivi, aumentando la complessità complessiva del sistema.
- **Single Point of Failure (SPOF):** Se non configurato correttamente, il Gateway API può diventare un singolo punto di fallimento per l'intero sistema. Un'interruzione o problemi di prestazioni del gateway possono compromettere l'intera applicazione. È fondamentale garantire un'adeguata ridondanza, scalabilità e tolleranza ai guasti durante il deployment.
- **Latenza:** L'API Gateway aggiunge un "hop" extra nel percorso richiesta-risposta, il che può introdurre una certa latenza. Sebbene l'impatto sia solitamente minimo e mitigabile tramite ottimizzazioni delle prestazioni, caching e bilanciamento del carico, in applicazioni ad altissima frequenza o sensibili al tempo, questo potrebbe essere un fattore da considerare.
- **Overhead di Manutenzione:** Un Gateway API richiede monitoraggio, manutenzione e aggiornamenti regolari per garantirne la sicurezza e l'affidabilità. Questo può aumentare l'overhead operativo per il team di sviluppo, specialmente in caso di gestione autonoma del gateway.
- **Complessità di Configurazione:** I Gateway API spesso offrono un'ampia gamma di funzionalità e opzioni di configurazione. L'impostazione e la gestione di queste configurazioni possono essere complesse e richiedere tempo, soprattutto in ambienti multi-ambiente o deployment su larga scala.

Specificamente per Spring Cloud Gateway, alcune difficoltà possono includere:

- **Curva di Apprendimento per la Programmazione Reattiva:** Essendo basato su Spring WebFlux e Project Reactor, SCG richiede una familiarità con il paradigma di programmazione reattiva, che può presentare una curva di apprendimento per gli sviluppatori abituati a modelli sincroni.
- **Debugging in un Sistema Distribuito:** La natura distribuita del sistema, con il gateway che si interpone tra client e servizi, può rendere il debugging più complesso, richiedendo strumenti di tracciamento distribuiti.
- **Configurazione di Regole Complesse:** Sebbene i predicati e i filtri offrano grande flessibilità, la configurazione di regole di routing e logiche di filtro molto complesse può diventare intricata e difficile da mantenere.

Riconoscere le sfide e le difficoltà dimostra un'onestà intellettuale e una comprensione matura del fatto che nessuna tecnologia è una soluzione universale. Questo si allinea pienamente con il requisito di "Critical Thinking", che esorta a non "innamorarsi" delle nuove tecnologie, ma a cogliere anche gli ostacoli e i limiti.

6.2 Limiti di Applicabilità e Sviluppi Futuri

Nonostante i numerosi vantaggi, esistono scenari in cui l'adozione di un API Gateway potrebbe essere eccessiva o presentare limitazioni intrinseche.

- **Overkill per Applicazioni Semplici:** Per applicazioni monolitiche molto semplici o con un numero limitato di endpoint, l'introduzione di un API Gateway potrebbe aggiungere complessità non necessaria senza un ritorno significativo sull'investimento.
- **Limitazioni di Scalabilità per Casi Estremi:** Sebbene SCG sia altamente scalabile, alcuni Gateway API (come l'esempio di AWS API Gateway citato) possono avere quote restrittive per richieste al secondo o connessioni concorrenti in scenari di scala estremamente elevata. Per carichi di lavoro massivi, potrebbe essere necessario un'attenta ottimizzazione o l'esplorazione di soluzioni personalizzate.
- **Gestione dello Stato delle Connessioni WebSocket e Broadcasting:** La gestione dello stato delle connessioni WebSocket non scala bene intrinsecamente in alcuni Gateway API, richiedendo l'archiviazione di metadati in database esterni. Inoltre, la capacità di trasmettere messaggi in broadcast a un gran numero di client connessi (un pattern comune per applicazioni real-time come aggiornamenti di punteggi o quotazioni di borsa) non è nativamente supportata da alcuni gateways e richiede implementazioni punto-punto, che possono colpire i limiti di chiamata API.
- **Difficoltà di Global Availability:** Rendere un Gateway API globalmente disponibile può essere complesso. Alcuni servizi di gateway sono regionali, e la creazione di un'architettura multi-regione, sebbene migliore, introduce una complessità significativa nella configurazione e nell'orchestrazione dei componenti.
- **Vendor Lock-in:** L'utilizzo di un servizio Gateway API gestito da un fornitore di cloud specifico può portare a una dipendenza dalla loro infrastruttura, prezzi e set di funzionalità, rendendo più difficile la migrazione a un fornitore o piattaforma diversa in futuro.

Sulla base delle limitazioni identificate, si propongono i seguenti sviluppi futuri per il progetto implementato o per Spring Cloud Gateway in generale:

- **Monitoraggio e Alerting Avanzati:** Integrare il gateway con sistemi di monitoraggio e alerting più sofisticati (es. Prometheus e Grafana) per ottenere una visibilità in tempo reale sulle prestazioni, la latenza e gli errori, e configurare alert proattivi.
- **Integrazione con Service Mesh:** Esplorare l'integrazione di SCG con una service mesh (es. Istio o Linkerd). Una service mesh può gestire la comunicazione inter-servizio, la resilienza e l'osservabilità a un livello più profondo, complementando le funzionalità del gateway per il traffico "north-south" (esterno-interno) e "east-west" (interno-interno).
- **Esplorazione di Opzioni Serverless:** Valutare l'implementazione di un Gateway API in un contesto serverless (es. AWS Lambda con API Gateway) per ridurre l'overhead di manutenzione e scalare automaticamente in base alla domanda.
- **Miglioramento del Supporto WebSocket:** Per applicazioni che richiedono una gestione avanzata di WebSocket, esplorare soluzioni dedicate o pattern architetturali che consentano il broadcasting efficiente e la gestione dello stato delle connessioni.
- **Politiche di Sicurezza più Sofisticate:** Implementare politiche di sicurezza più granulari, come la validazione dello schema delle richieste (API schema validation) o la protezione da attacchi specifici (es. iniezione SQL, XSS).

Identificare i limiti e proporre sviluppi futuri dimostra lungimiranza e capacità di risoluzione dei problemi, trasformando un rapporto descrittivo in un'analisi orientata al futuro. Questo risponde direttamente al requisito delle linee guida di evidenziare i limiti e gli sviluppi futuri necessari per superarli.

Tabella 6.1: Vantaggi e Svantaggi di Spring Cloud Gateway

Vantaggi	**Svantaggi**
Centralizzazione delle preoccupazioni trasversali: Sicurezza, monitoraggio, resilienza gestiti in un unico luogo.	Complessità aggiuntiva: Introduce un nuovo strato architetturale da gestire.
Architettura reattiva e non bloccante: Ottimizzata per alta concorrenza e bassa latenza.	Single Point of Failure (SPOF): Se non configurato con ridondanza, può bloccare l'intero sistema.
Integrazione client semplificata: Un unico punto di ingresso per i client.	Latenza aggiuntiva: Ogni "hop" può introdurre un minimo ritardo.
Accelerazione dell'innovazione: I team si concentrano sulla logica di business.	Overhead di manutenzione: Richiede monitoraggio, aggiornamenti e gestione continua.
Sicurezza e conformità migliorate: Controlli centralizzati e applicazione di best practice.	Complessità di configurazione: Ampia gamma di funzionalità e opzioni.
Flessibilità di routing: Basato su predicati e filtri configurabili.	Curva di apprendimento: Richiede familiarità con la programmazione reattiva.
Resilienza integrata: Supporto per Circuit Breaker e gestione del traffico.	Limitazioni per WebSocket: Difficoltà nella gestione dello stato e del broadcasting di messaggi.
Versionamento API: Gestione fluida di più versioni API.	Difficoltà di global availability: Implementazione multi-regione complessa.

Capitolo 7

Conclusioni e Sviluppi Futuri

7.1 Riepilogo dei Risultati

Il presente Project Work ha affrontato lo studio e l'implementazione di un *Gateway API* con `Spring Cloud Gateway`, un componente cruciale nelle moderne architetture a microservizi. Gli obiettivi iniziali del progetto, che includevano l'indagine sui concetti fondamentali dei Gateway API, l'elaborazione sulle caratteristiche di `Spring Cloud Gateway`, lo sviluppo di un'applicazione pratica e un'analisi critica della tecnologia, sono stati pienamente raggiunti.

Attraverso l'indagine, è stata consolidata la comprensione del ruolo essenziale di un Gateway API come *punto di ingresso unificato*, capace di semplificare l'integrazione dei client e di centralizzare preoccupazioni trasversali come la *sicurezza*, il *rate limiting* e la *resilienza*. L'analisi di `Spring Cloud Gateway` ha evidenziato la sua *robustezza*, derivante dalla sua fondazione su un'architettura *reattiva e non bloccante* (`Spring WebFlux` e `Project Reactor`), che lo rende particolarmente performante in scenari ad alta concorrenza. La *modularità* intrinseca di SCG, basata su route, predicati e filtri, è stata riconosciuta come un fattore chiave per la sua flessibilità ed estensibilità.

L'implementazione pratica ha dimostrato con successo le capacità di `Spring Cloud Gateway` nel *routing dinamico*, nella gestione centralizzata dell'*autenticazione e dell'autorizzazione* (anche tramite `OAuth2`), nell'applicazione di *filtri personalizzati* per la manipolazione delle richieste/risposte e nella *limitazione del tasso di richieste*. Questi esempi concreti hanno convalidato la comprensione teorica e l'applicabilità della tecnologia.

L'analisi critica ha fornito una prospettiva equilibrata, riconoscendo i numerosi vantaggi di `SCG` in termini di sicurezza, accelerazione dello sviluppo e miglioramento delle prestazioni, ma anche evidenziando le sfide. Tra queste, l'introduzione di *complessità architetturale*, il potenziale di *latenza aggiuntiva* e le sfide legate alla gestione dello stato delle *connessioni WebSocket* in scenari specifici. Questa valutazione ponderata è fondamentale per una comprensione completa della tecnologia.

7.2 Contributi e Prospettive Future

Il principale contributo di questo Project Work risiede nella combinazione di una rigorosa *analisi teorica* con un'*implementazione pratica tangibile* di `Spring Cloud Gateway`. Il progetto fornisce una risorsa completa per la comprensione di questa tecnologia, fungendo da guida per futuri sviluppi o per l'adozione in contesti reali. La documentazione dei dettagli implementativi e degli esempi di utilizzo pratico offre un punto di riferimento prezioso per chiunque intenda esplorare o implementare SCG.

Guardando al futuro, diverse aree di sviluppo potrebbero ulteriormente migliorare il progetto e la comprensione di `Spring Cloud Gateway`:

- **Resilienza Avanzata:** Approfondire l'implementazione di pattern di resilienza come *Circuit Breaker* (utilizzando `Resilience4j`) e *Retry*, configurando fallback complessi per gestire i guasti dei servizi backend in modo più robusto.
- **Monitoraggio e Tracciamento Distribuito:** Integrare il Gateway con sistemi di monitoraggio e tracciamento distribuiti (es. `Zipkin` o `OpenTelemetry`) per ottenere una visibilità end-to-end sulle richieste attraverso l'intera catena di microservizi, facilitando il debugging e l'ottimizzazione delle prestazioni.
- **Gestione Avanzata di WebSocket:** Esplorare soluzioni e pattern architetturali per superare le limitazioni nella gestione dello stato delle connessioni `WebSocket` e nella capacità di broadcasting, rendendo il gateway più adatto per applicazioni real-time su larga scala.
- **Automazione del Deployment (`CI/CD`):** Sviluppare una pipeline `CI/CD` per automatizzare il building, il testing e il deployment del Gateway API e dei suoi servizi backend, migliorando l'efficienza operativa.

- **Test di Performance e Carico:** Eseguire test di performance e carico per valutare la scalabilità del Gateway in diverse condizioni di traffico e identificare potenziali colli di bottiglia.

Questi sviluppi futuri non solo affronterebbero le limitazioni identificate, ma spingerebbero anche l'applicazione di `Spring Cloud Gateway` verso scenari più complessi e di produzione, dimostrando una comprensione continua e un approccio proattivo alla risoluzione dei problemi. Il lavoro svolto in questo progetto serve da solida base per ulteriori ricerche e applicazioni pratiche nel campo dei Gateway API e delle architetture a microservizi.