

# Relazione Homework 2 DSBD

Lorenzo Varsallona (1000084765) - Paola Pappalardo (1000035439)

## Indice

<b>1</b>	<b>Introduzione e Obiettivi</b>	<b>2</b>
<b>2</b>	<b>Architettura del sistema e decomposizione</b>	<b>2</b>
2.1	Architettura Generale . . . . .	3
2.2	Database . . . . .	3
<b>3</b>	<b>Modifiche al Data Collector</b>	<b>3</b>
3.1	Gestione Interessi Utente . . . . .	3
3.2	Circuit Breaker . . . . .	5
3.3	Comunicazione tramite Kafka . . . . .	5
<b>4</b>	<b>Nuovi microservizi che utilizzano Kafka</b>	<b>6</b>
4.1	Alert System . . . . .	6
4.1.1	Avvio e Connessione . . . . .	6
4.1.2	Configurazione Kafka . . . . .	6
4.1.3	Strategia di Commit . . . . .	8
4.1.4	Logica di Elaborazione . . . . .	8
4.2	Alert Notifier System . . . . .	9
<b>5</b>	<b>API Gateway con Nginx</b>	<b>9</b>
5.1	Sicurezza e Networking . . . . .	9
5.2	Strategia di Load Balancing e Scalabilità Futura . . . . .	10

# 1 Introduzione e Obiettivi

Il presente documento illustra le modifiche apportate e le decisioni progettuali prese per la seconda parte dell'homework. L'obiettivo principale è stato quello di evolvere l'architettura del sistema verso un approccio a microservizi più robusto e scalabile, introducendo meccanismi di comunicazione asincrona tramite Kafka, pattern di resilienza come il Circuit Breaker e un sistema di notifica via email.

Nelle sezioni successive, si andranno ad approfondire le nuove funzionalità introdotte nell'applicazione, tra le quali l'integrazione con Apache Kafka, l'implementazione del Circuit Breaker nel Data Collector e la gestione delle notifiche di allerta.

# 2 Architettura del sistema e decomposizione

In questa sezione si approfondiranno gli aspetti implementativi adottati all'interno dell'applicativo, con particolare attenzione alle nuove scelte tecnologiche e architetturali introdotte per soddisfare i requisiti della seconda parte dell'homework.

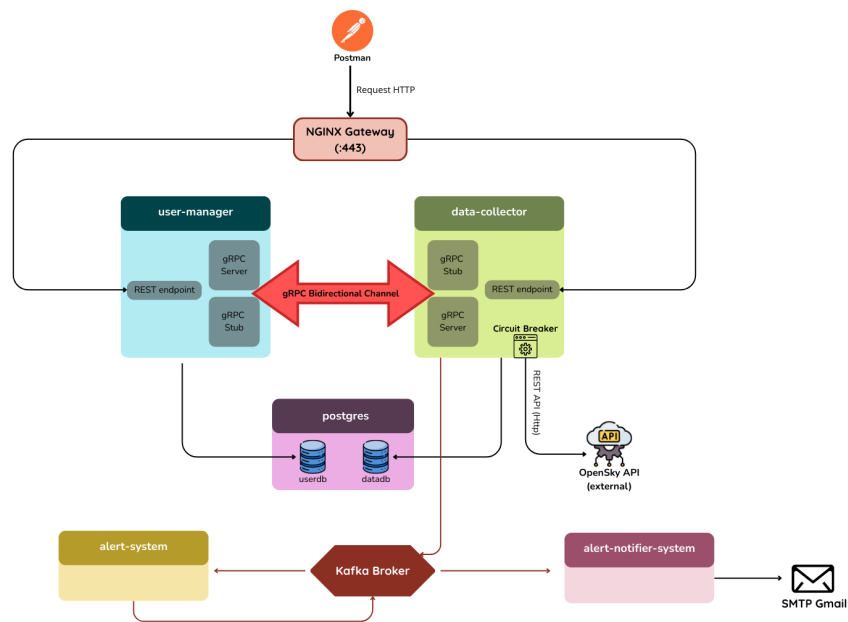


Figura 1: Diagramma architetturale generale del sistema.

## 2.1 Architettura Generale

L'applicativo è stato strutturato secondo un'architettura a microservizi, che è stata estesa rispetto alla versione precedente per garantire maggiore scalabilità e resilienza. Oltre ai microservizi originali (**User-Manager** e **Data-Collector**), l'architettura ora comprende:

- **Alert-System**: nuovo microservizio responsabile dell'elaborazione degli eventi di volo e della verifica delle soglie di allerta.
- **Alert-Notifier-System**: nuovo microservizio responsabile dell'invio delle notifiche via email.

La comunicazione tra i servizi è stata arricchita: mentre User-Manager e Data-Collector continuano a comunicare via gRPC, l'interazione verso i nuovi servizi di allerta avviene in modo asincrono tramite il message broker **Apache Kafka**. Inoltre, per migliorare la robustezza delle chiamate verso API esterne, è stato introdotto il pattern **Circuit Breaker** nel Data-Collector.

## 2.2 Database

Le modifiche allo schema dati si sono concentrate sul database **datadb**, specificamente per supportare la gestione delle soglie di allerta personalizzate.

**Tabella user\_airports** Questa tabella, che memorizza le associazioni tra utenti e aeroporti, è stata modificata aggiungendo le colonne per le soglie:

Campo	Tipo	Vincolo
id	INTEGER	PK, Autoincrement
user_email	VARCHAR(255)	NOT NULL
airport_code	VARCHAR(10)	NOT NULL
high_value	INTEGER	Nullable
low_value	INTEGER	Nullable

## 3 Modifiche al Data Collector

Il componente **data-collector** è stato aggiornato per migliorare l'affidabilità e disaccoppiare la raccolta dati dall'elaborazione degli allarmi.

### 3.1 Gestione Interessi Utente

L'API `/register_airports` è stata estesa per supportare la definizione di soglie personalizzate per ogni aeroporto monitorato. Ora, oltre al codice dell'aeroporto, l'utente può specificare opzionalmente:

- **high\_value**: soglia massima di voli, superata la quale si desidera ricevere un allarme.
- **low\_value**: soglia minima di voli, al di sotto della quale si desidera ricevere un allarme.

È possibile registrare un aeroporto senza specificare alcuna soglia. Le soglie possono essere aggiunte successivamente, sia singolarmente che entrambe, tramite una nuova chiamata allo stesso endpoint, che aggiornerà i valori esistenti. Il sistema valida che, se presenti entrambi, il valore massimo sia strettamente maggiore del minimo.

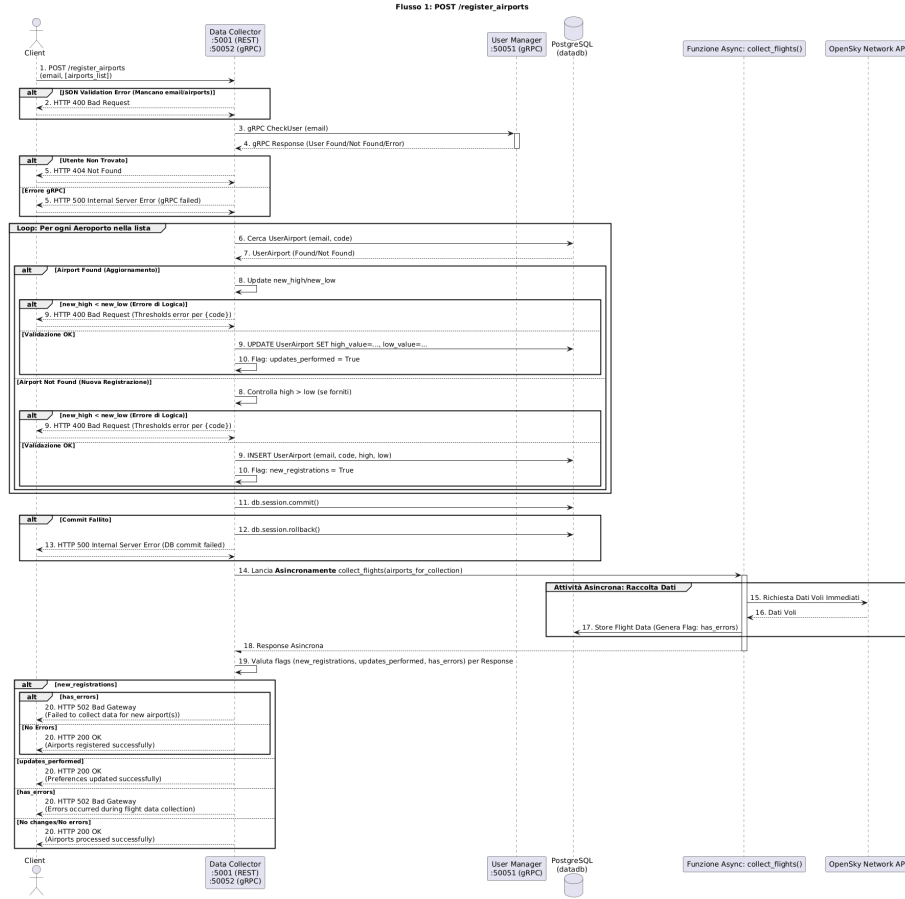


Figura 2: Diagramma di sequenza POST/register\_airports

Inoltre, è stata introdotta una nuova rotta per la gestione delle thresholds:

- **/airports/deleteThresholds**: (metodo DELETE, esposta dal Data Collector) consente di rimuovere le soglie personalizzate (**high\_value** e **low\_value**) per un aeroporto specifico, mantenendo però l'interesse attivo. In questo modo l'utente continua a

monitorare l'aeroporto ma non riceverà più allarmi basati sulle soglie precedentemente impostate.

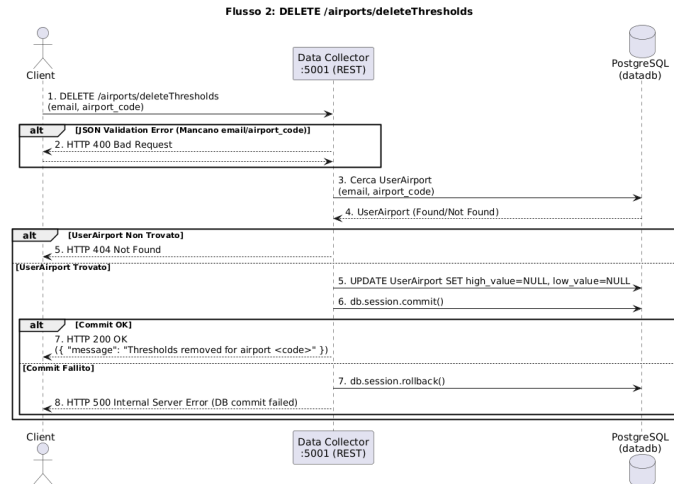


Figura 3: Diagramma di sequenza DELETE/airports/deleteThresholds

### 3.2 Circuit Breaker

Per gestire eventuali fallimenti o latenze eccessive nelle chiamate alle API esterne di OpenSky, è stato implementato il pattern **Circuit Breaker**. Il meccanismo è configurato con una soglia di fallimento (`failure_threshold`) di 3 tentativi e un timeout di recupero (`recovery_timeout`) di 60 secondi. Questo previene il sovraccarico del servizio remoto in caso di disservizi e permette al sistema di fallire velocemente (fail-fast) senza bloccare le risorse.

### 3.3 Comunicazione tramite Kafka

Una volta raccolti i dati, il sistema agisce come un **Kafka Producer**. Nello specifico, ogni volta che viene eseguita la funzione `collect_flights` (invocata periodicamente o scatenata dalla registrazione di nuovi aeroporti tramite `/register_airports`), il servizio pubblica un messaggio sul topic `to-alert-system`. Questo messaggio contiene il conteggio aggiornato dei voli per ogni aeroporto monitorato, permettendo un'elaborazione asincrona e scalabile da parte del sistema di allerta.

Per ottimizzare l'affidabilità e l'efficienza della trasmissione, il Producer è stato configurato con i seguenti parametri:

- `acks='all'`: garantisce la massima durabilità dei dati, richiedendo la conferma di scrittura da tutte le repliche in-sync (ISR) prima di considerare il messaggio inviato con successo.
- `retries=5`: aumenta la resilienza del sistema, permettendo al producer di riprovare l'invio in caso di errori transitori (es. elezione di un nuovo leader) senza perdere messaggi.
- `batch.size=32768` (32KB): incrementa la dimensione del batch per favorire l'invio cumulativo di messaggi, migliorando il throughput di rete.
- `linger.ms=20`: introduce una latenza intenzionale di 20ms per permettere al buffer di riempirsi maggiormente prima dell'invio, riducendo il numero di richieste I/O verso il broker.

## 4 Nuovi microservizi che utilizzano Kafka

In questa sezione verranno analizzati i due nuovi microservizi implementati: Alert System e Alert Notifier System.

### 4.1 Alert System

Il nuovo microservizio `alert-system` è stato introdotto per gestire la logica di business relativa al monitoraggio delle soglie.

#### 4.1.1 Avvio e Connessione

All'avvio del servizio, è stato introdotto un ritardo intenzionale (`time.sleep(20)`) prima di tentare la connessione a Kafka. Questa attesa è necessaria per garantire che i broker Kafka siano completamente avviati e pronti ad accettare connessioni, evitando errori di connessione ("Broker not available") durante la fase di startup dell'infrastruttura Docker Compose.

#### 4.1.2 Configurazione Kafka

Il servizio utilizza la libreria `confluent-kafka` sia come Consumer che come Producer.

A livello di infrastruttura (Docker Compose), è stata adottata la modalità **KRaft (Kafka Raft Metadata mode)**. Questa architettura rimuove la dipendenza da Zookeeper, semplificando il deployment e la gestione del cluster. Il broker agisce quindi autonomamente sia per la gestione dei dati che per il coordinamento del cluster (controller). Di seguito i parametri di configurazione utilizzati per abilitare questa modalità in un ambiente a singolo nodo:

- **KAFKA\_NODE\_ID**: 1: identificativo univoco del nodo nel cluster.
- **KAFKA\_PROCESS\_ROLES**: 'broker,controller': il nodo agisce sia come broker (gestione dati) che come controller (gestione metadati).
- **KAFKA\_CONTROLLER\_QUORUM\_VOTERS**: '1@kafka:9093': definisce il quorum dei votanti per il controller; essendo un solo nodo, vota se stesso.
- **KAFKA\_LISTENERS** e **KAFKA\_ADVERTISED\_LISTENERS**: configurano gli endpoint di ascolto. La porta 9092 è usata per i client (PLAINTEXT), mentre la 9093 per il traffico interno del controller.
- **KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP**: mappa i nomi dei listener ai protocolli di sicurezza (qui PLAINTEXT non criptato).
- **KAFKA\_INTER\_BROKER\_LISTENER\_NAME**: 'PLAINTEXT': definisce il protocollo per la comunicazione tra broker.
- **KAFKA\_CONTROLLER\_LISTENER\_NAMES**: 'CONTROLLER': definisce il listener usato dal controller.
- **KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR**: 1: imposta la replica del topic degli offset a 1.
- **KAFKA\_TRANSACTION\_STATE\_LOG\_REPLICATION\_FACTOR**: 1: imposta la replica del log delle transazioni a 1.
- **KAFKA\_TRANSACTION\_STATE\_LOG\_MIN\_ISR**: 1: numero minimo di repliche in-sync richiesto per le transazioni (1 per single-node).
- **KAFKA\_GROUP\_INITIAL\_REBALANCE\_DELAY\_MS**: 0: rimuove il ritardo iniziale per il rebalance dei gruppi consumer, velocizzando l'avvio in sviluppo.
- **CLUSTER\_ID**: ID univoco del cluster Kafka generato per la modalità KRaft.

Per il **Consumer**, sono stati impostati i seguenti parametri:

- **group.id**: impostato a "alert-system-group", identifica il gruppo di consumo permettendo a Kafka di gestire gli offset per questa specifica applicazione.
- **auto.offset.reset**: impostato a "earliest", garantisce che, in assenza di un offset salvato (es. primo avvio), il consumer legga tutti i messaggi presenti nel topic dall'inizio, evitando perdita di dati pregressi.
- **enable.auto.commit**: impostato a **False** per disabilitare il commit automatico degli offset, delegando all'applicazione la responsabilità di confermare l'elaborazione (come dettagliato nella sezione successiva).

Per il **Producer**, sono stati replicati gli stessi parametri di ottimizzazione utilizzati nel Data Collector (`acks='all'`, `retries=5`, `batch.size=32768`, `linger.ms=20`).

#### 4.1.3 Strategia di Commit

Il servizio agisce come un **Kafka Consumer**, sottoscrivendosi al topic `to-alert-system`. Per garantire l'affidabilità dell'elaborazione ("at-least-once delivery"), è stata scelta una strategia di **Commit Manuale Sincrono** (`commitSync`). A differenza dell'Auto Commit (che committa periodicamente in base al tempo, rischiando di perdere messaggi in caso di crash o di processarli due volte in caso di rebalance lento), il commit manuale viene eseguito solo *dopo* che il messaggio è stato completamente elaborato e le eventuali notifiche sono state inviate al topic successivo. Sebbene questa scelta possa ridurre leggermente il throughput rispetto al commit asincrono, garantisce che nessun messaggio venga perso.

#### 4.1.4 Logica di Elaborazione

Alla ricezione di un messaggio di aggiornamento ("`update_completed`"), il sistema:

1. Estrae i conteggi dei voli per ogni aeroporto.
2. Interroga il database per recuperare le preferenze degli utenti (soglie minime e massime) per quegli aeroporti.
3. Verifica se le condizioni di allerta sono soddisfatte (es. numero di voli superiore alla soglia massima).

Se viene rilevata un'anomalia, l'`alert-system` produce un nuovo messaggio sul topic `to-notifier`, contenente i dettagli dell'allerta (email destinatario, aeroporto, condizione violata).

Nell'immagine sotto viene fornita una visione d'insieme dell'intero flusso di notifica Kafka.

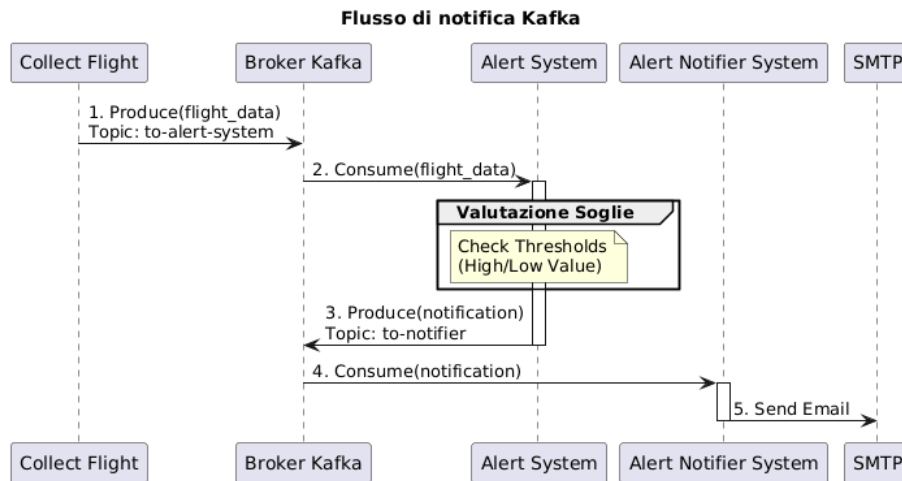


Figura 4: Flusso di notifica Kafka

## 4.2 Alert Notifier System

L'`alert-notifier-system` è il componente finale della catena di allerta. Esso consuma i messaggi dal topic `to-notifier` e si occupa dell'invio effettivo delle notifiche agli utenti. L'invio delle email è implementato utilizzando il server SMTP di **Gmail** (`smtp.gmail.com`).

## 5 API Gateway con Nginx

Per semplificare l'accesso ai microservizi e fornire un unico punto di ingresso (entry point) per il client, è stato introdotto **Nginx** configurato come *Reverse Proxy*. Questa configurazione permette di instradare le richieste HTTP verso i container corretti (`user-manager` o `data-collector`) in base al percorso dell'URL (es. `/users` verso User Manager, `/flights` verso Data Collector), nascondendo la complessità dell'architettura sottostante e facilitando eventuali operazioni di bilanciamento del carico in futuro.

### 5.1 Sicurezza e Networking

- **Isolamento dei Servizi (expose vs ports):** rispetto alla versione precedente del progetto, è stata migliorata la sicurezza di rete modificando la visibilità dei container. I microservizi *User Manager* e *Data Collector* non espongono più le proprie porte direttamente sull'host (direttiva `ports`). Al contrario, si è optato per l'uso di `expose`, che limita l'accessibilità delle porte (5000, 5001, 50051, 50052) alla sola rete interna di Docker. Di conseguenza, l'unico punto di ingresso pubblico è la porta 80 gestita da Nginx, obbligando tutte le richieste a transitare attraverso l'API Gateway. Per

quanto riguarda i database, si è scelto di mantenere temporaneamente l'accesso diretto dall'host per agevolare le attività di debug e manutenzione, con la prospettiva di rimuoverlo in produzione per blindare ulteriormente l'infrastruttura.

## 5.2 Strategia di Load Balancing e Scalabilità Futura

Nonostante l'attuale configurazione preveda una singola istanza per ogni microservizio, l'architettura di Nginx è stata predisposta per facilitare una futura scalabilità orizzontale. Attualmente, il blocco `upstream` utilizza l'algoritmo predefinito *Round Robin*, adeguato per lo scenario corrente. Tuttavia, in un'ottica di espansione con repliche multiple, è stata valutata l'adozione dell'algoritmo `least_conn`, che risulta essere più efficiente al crescere dell'applicativo.