UNIVERSITÀ
degli STUDI
di CATANIA

DIPARTIMENTO DI INGEGNERIA
ELETTRICA ELETTRONICA E INFORMATICA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Homework 2

Studenti:
Ficarra Filippo 1000034832
Guerrera Mario Anthony 1000035663

Anno Accademico 2025/2026

Sommario

1. Descrizione dell’applicazione

L'architettura è stata potenziata con l'introduzione di pattern e tecnologie volti a migliorare la resilienza e il disaccoppiamento tra i servizi:

- **Resilienza con Circuit Breaker:** È stato implementato un Circuit Breaker per proteggere il sistema durante le interazioni con le API esterne di *Open Sky Network*. In caso di sovraccarichi o guasti dell'API, il componente interrompe temporaneamente le chiamate, prevenendo blocchi a catena e preservando le risorse interne.
- **Architettura Event-Driven (Apache Kafka):** L'integrazione di Apache Kafka ha trasformato il sistema in un'architettura basata su eventi. Kafka funge da broker asincrono, permettendo un disaccoppiamento totale tra i microservizi e garantendo una gestione efficiente dei flussi di dati.
- **Sistema di Notifica e Gestione Soglie:** È stato introdotto un meccanismo di notifica asincrona (via email) attivato dal superamento di soglie personalizzate (*high_value* e *low_value*). L'utente può configurare o aggiornare tali parametri in modo flessibile: è possibile impostare anche una sola soglia, con il vincolo che, se presenti entrambe, la soglia alta sia superiore alla bassa.
- **API Gateway (NGINX):** NGINX è stato configurato come *Single Entry Point* per il sistema. Agendo da API Gateway, centralizza il routing del traffico verso i microservizi interni, semplificando l'accesso esterno e migliorando la sicurezza e la gestione delle politiche di rete.

2. Schema architetturale

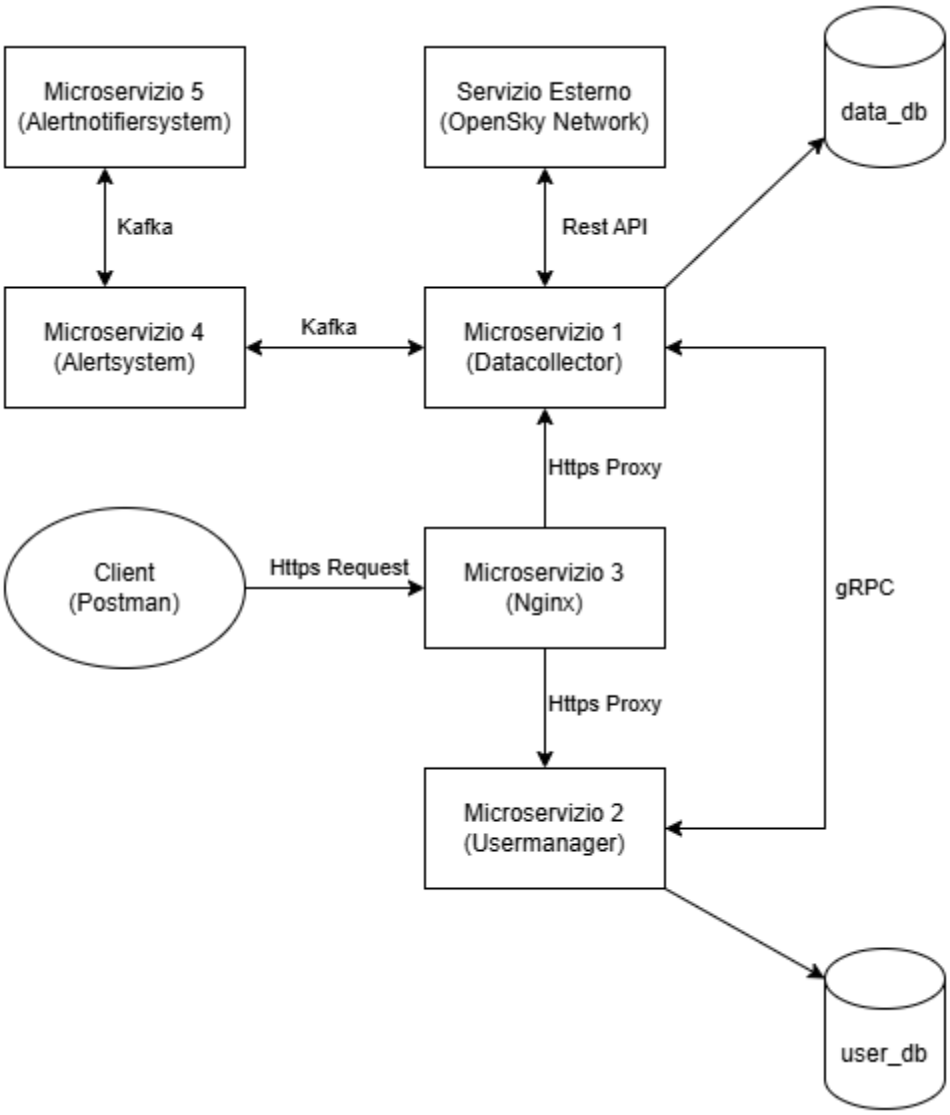


Figura 1: Schema Architetturale

L'architettura di sistema è stata sensibilmente ampliata con l'integrazione di tre nuovi microservizi specializzati, progettati per gestire il routing, la logica di monitoraggio e la notifica all'utente.

- **Microservizio NGINX (API Gateway):** Implementato come *Single Entry Point*, agisce da punto di accesso unico per tutte le richieste HTTPS provenienti dai client. Grazie alla configurazione delle *locations*, instrada il traffico verso il microservizio corretto in base all'URL fornito, garantendo una separazione netta tra la rete esterna e i servizi interni.
- **Microservizio AlertSystem (Processor):** Svolge il ruolo di intermediario logico operando sia come *Consumer* che come *Producer*.

    - **Consumer:** Legge dal topic to-alert-system i pacchetti dati inviati dal *Data Collector*, che includono il codice dell'aeroporto, il totale dei voli salvati (che comprendono sia partenze, sia arrivi), un timestamp di scrittura del messaggio, e informazioni sugli utenti interessati a quell'aeroporto, ovvero email, high_value e low_value).
    - **Producer:** Confronta i dati ricevuti con le soglie impostate. qualora il valore totale dei voli salvati sia inferiore o superiore, rispettivamente, a low_value o high_value, scrive un messaggio nel topic "to-notifier", contenente la mail dell'utente per il quale la soglia è stata superata, il codice dell'aereoporto, la condizione che si è verificata (SUPERATA_SOGLIA_ALTA o SUPERATA_SOGLIA_BASSA) e il valore che è stato superato (che può essere un high_value o un low_value).

- **Microservizio AlertNotifierSystem (Email Notifier):** Opera esclusivamente come *Consumer* leggendo dal topic to-notifier. Una volta ricevuto l'evento di allerta, utilizza una libreria dedicata per l'invio automatizzato di email verso l'indirizzo indicato nel messaggio, completando il ciclo di notifica asincrona.
- **Aggiornamento del Modello Dati:** Parallelamente all'introduzione dei nuovi servizi, il *Data Collector* è stato rifinito e lo schema del database data_db (in particolare, la tabella INTERESTS) è stato aggiornato per supportare i campi *high_value* e *low_value*. Tali parametri sono stati configurati per accettare valori nulli di default, permettendo all'utente la massima flessibilità nell'impostazione delle soglie di monitoraggio.

3. Microservizi e relative comunicazioni

L'applicazione è stata arricchita dall'aggiunta e la modifica dei seguenti microservizi:

1. Datacollector

Il seguente microservizio ha subito delle modifiche.

- Rimozione di alcune chiamate ridondanti alla funzione data_collection_job(). In questo modo, viene eseguita una sola volta all'avvio dell'applicazione, e ogni intervallo temporale specificato dalla variabile d'ambiente PERIODO nel docker-compose.yaml
  Nota: si ricorda che la funzione data_collection_job() consente di recuperare, tramite chiamata REST API ad OpenSky Network, le informazioni sui voli di interesse per l'utente.
- Implementazione del pattern CircuitBreaker. In particolare, è stata adottata la seguente strategia:
  2. Implementazione del file "circuit_breaker.py", il quale evidenzia la logica del pattern.
  3. Creazione, all'interno del datacollector/app.py, di un oggetto circuit_breaker. Con tale scelta, le richieste al servizio OpenSky Network vengano fatte da tale istanza direttamente all'interno della funzione data_collection_job(), e, nel caso in cui la richiesta fallisca per un certo numero di volte (fino ad una treshold pari a 5), l'oggetto circuit_breaker solleva un'eccezione, attivando lo stato "OPEN", in cui non vengono inviate richieste. A questo punto, dopo un recovery_time impostato a 30 secondi, si attiverà lo stato "HALF_OPEN", e verranno inviate nuovamente le richieste, con un Periodo settato nel docker-compose. La motivazione dell'attribuzione di tali valori di soglia e tempo di recupero sono da attribuire ad una maggiore velocità e facilità nel testare il funzionamento dell'applicazione.
- Modifica della funzione add_interest, con l'aggiunta della logica per l'inserimento delle soglie high_value e low_value (se non vengono forniti, saranno nulli) e per il confronto tra i due, verificando che high_value sia maggiore di low_value. Può anche essere fornito solo uno tra i due valori.
- Modifica della funzione interest, in modo da ritornare anche i due valori soglia.

- Implementazione della funzione add_values, la quale permette di aggiungere uno dei due valori soglia, o anche tutti e due i valori soglia, specificando la mail dell'utente, il codice dell'aeroporto. Se la coppia utente-codice aereoporto non è presente nella tabella interest, verrà restituito un messaggio di errore. Si avrà un errore anche qualora non sia soddisfatta la condizione high-value>low_value. E' stata implementata con una logica tale che, se viene fornito uno solo dei valori, ad esempio low_value, tale confronto verrà fatto con il valore di high-value già presente in tabella. Se high_value è nullo, allora il confronto non viene fatto. La logica di implementazione è stata realizzata con l'obiettivo di renderla, oltre che una funzione per aggiungere valori di high_value e low_value, una funzione per la modifica dei valori già esistenti
- Applicazione della logica di comunicazione asincrona, mediante Apache Kafka. In particolare, il datacollector è stato configurato come producer, che scrive sul topic "to-alert-system" le informazioni sui voli raccolti. In particolare, tale logica viene implementata nella seguente maniera:
    1. Modifica della funzione data_collection_job(), la quale, dopo aver raccolto i voli in partenza o in arrivo dall'aereoporto di interesse, seleziona dalla tabella Interests tutti gli utenti, con i rispettivi high_value e low_value, associati a tale aereoporto, e li inserisce nella lista user_thresholds, contenente tanti elementi quanti sono gli utenti che hanno quell'aeroporto come interesse. Infine, chiama la funzione publish_flight_count, alla quale passa come argomenti il codice dell'aeroporto, il totale dei voli raccolti, e user_thresholds.
    2. Aggiunta della funzione publish_fligh_count, la quale riceve i parametri discussi in precedenza, e, insieme ad un timestamp che tiene traccia del momento della scrittua, vengono assemblate nel dizionario "payload". Successivamente, Il dizionario viene convertito in una stringa formattata secondo lo standard JSON. Questa viene poi trasformata in un array di byte (bytes), che è il formato effettivo che può viaggiare sulla rete, e, infine, il messaggio viene scritto sul topic "to-alert-system".

    NOTA: La scelta di creare una lista user_thresholds da scrivere all'interno del topic è dovuta alla volontà di avere un maggiore disaccoppiamento dei servizi, in quando, grazie a questa lista, l'AlertSytem non dovrà interrogare i database e comunicare in

alcun modo con gli altri microservizi, ma dovrà solo leggere i messaggi dal topic “to-alert-system” e scrivere i messaggi all’interno del topic “to-notifier”

2. Nginx

Il microservizio NGINX funge da API Gateway , stabilendo il Punto di Ingresso Unico e centralizzando la gestione del traffico per le comunicazioni sincrone (HTTP/REST) del sistema.

Il suo funzionamento si basa su tre ruoli interconnessi:

1. Accesso Centralizzato e Reverse Proxy

- Punto di Contratto (apigate.com): NGINX risponde all'indirizzo logico apigate.com (server_name), fungendo da interfaccia pubblica e unico indirizzo che i client esterni devono conoscere per accedere al sistema.
- Reverse Proxy: NGINX intercetta tutte le richieste indirizzate a apigate.com e le reindirizza (proxy_pass) verso gli indirizzi interni della rete Docker (es. http://datacollector:5000 o http://usermanager:5000).
- Vantaggi: Isola i microservizi interni, nascondendo la loro topologia e i loro indirizzi IP/Porte dal mondo esterno, migliorando la sicurezza e la flessibilità architetturale.

2. Instradamento Intelligente (Routing)

- Logica di Routing: Utilizza le direttive location per mappare l'URL richiesto al microservizio competente.
- Gestione degli Endpoint: Supporta sia le regole statiche (matching diretto del percorso) sia le regole dinamiche (tramite espressioni regolari come location ~* /last_flight/([A-Z0-9]+)$) per instradare le richieste contenenti parametri variabili. La gestione della logica dei parametri è demandata al microservizio di destinazione.

3. Funzionamento nel Contesto Asincrono

- Mediazione Sincrona**:** NGINX gestisce esclusivamente il traffico HTTP bidirezionale (richiesta → inoltro → risposta), mediando le interazioni REST/gRPC.

- Separazione: Non è coinvolto nel flusso di eventi Kafka. Questa separazione rafforza il disaccoppiamento: un errore o un *crash* nell'API Gateway (che coinvolge apigate.com) non interrompe la pipeline di notifica asincrona (Alert System/Notifier System).

3. Alertsystem

Il microservizio si comporta sia da producer, sia da consumer. Esso, infatti, legge dal topic “to-alert-system”, in cui sono presenti i messaggi scritti dal datacollector. E’ implementata una logica interna, la quale consente di stabilire se il totale di voli raccolti, presente all’interno del messaggio, supera le soglie high_value o low_value, passate anche quelle per messaggio, in modo che, in caso di superamento, scrive un messaggio all’interno del topic “to-notifier”. In particolare, dopo aver letto il messaggio dal topic, e averlo memorizzato nella variabile flight_data, si richiama la funzione process_and_produce, passando come argomento il fligh_data raccolto. A quel punto, la funzione process_and_produce, a partire dal parametro ricevuto, ricava il codice dell’aereoporto, i voli totali raccolti per quell’aeroporto, e la lista user_thresholds, raccolta all’interno della variabile thresolds. A partire da questa, verifica per ogni utente, e quindi iterando per ogni elemento della lista, se i voli totali sono maggiori o minori dei high_value e low_value analizzati per ogni utente della lista (essi possono essere anche nulli, in tal caso non viene scritto alcun messaggio), in modo che, se una delle due soglie risulta superata, viene creato un “payload”, contenente il codice dell’aeroporto, la condizione verificata (SUPERATA_SOGLIA_ALTA o SUPERATA_SOGLIA_BASSA), la soglia (tra le due) che è stata superata, e la mail dell’utente della lista per il quale si è verificata la condizione. Successivamente, Il dizionario viene convertito in una stringa formattata secondo lo standard JSON. Questa viene poi trasformata in un array di byte (bytes), che è il formato effettivo che può viaggiare sulla rete, e, infine, il messaggio viene scritto sul topic “to-notifier”.

4. Alertnotifiersystem

L'AlertNotifierSystem funge da Consumer finale nella pipeline di notifica asincrona basata su Kafka. Il suo compito principale è ricevere gli eventi di allerta e agire come gestore delle notifiche (simulando l'invio di posta elettronica). Il servizio si connette a Kafka e legge passivamente i messaggi dal topic to-notifier (prodotti dall'AlertSystem).

Una volta letto il messaggio dal topic, estrae dal payload JSON le informazioni essenziali (email, codice aeroporto, condizione superata e soglia).

A questo punto, utilizza due funzioni implementate fondamentali:

1. Send_notification, che riceve come parametri i dati raccolti dal messaggio, e, dopo aver assemblato il formato della mail da inviare (mediante la funzione format_email_body()), tramite un oggetto SMTP, che stabilisce la connessione iniziale, la messa in sicura del canale e l'autenticazione dell'utente, viene inviata la mail. Inoltre, gestisce la resilienza dell'invio:

   Successo / Errore Permanente**:** Se l'invio riesce o l'errore è permanente (es. destinatario inesistente), il messaggio viene committato (consumer.commit) per evitare rilavorazioni inutili.

   Errore Temporaneo: Se l'errore è temporaneo (es. disconnessione di rete o timeout), il messaggio non viene committato. Questo assicura che il messaggio venga riletto e l'invio ritentato al ciclo successivo, garantendo l'affidabilità della notifica.
2. format_email_body, per costruire l'oggetto e il corpo dell'email in base alla condizione rilevata (SUPERATA_SOGLIA_ALTA o SUPERATA_SOGLIA_BASSA).

Il servizio opera in un ciclo infinito (while True), in attesa del prossimo evento da notificare.

Il cuore della funzionalità di notifica nell'AlertNotifierSystem è l'uso della libreria standard Python smtplib, che implementa il protocollo SMTP.

SMTP è il protocollo universale per la trasmissione di posta elettronica attraverso le reti TCP/IP. Quando il servizio deve inviare una notifica (l'email di allerta), smtplib viene utilizzato per stabilire una connessione con un server di posta esterno (definito dalle variabili d'ambiente SMTP_SERVER e SMTP_PORT).

La scelta di utilizzare la libreria standard smtplib e il protocollo SMTP per gestire l'invio delle notifiche è motivata da due fattori principali: la simulazione realistica in un ambiente di microservizi e l'integrazione della resilienza con Kafka.

4. Lista delle API Implementate

Rispetto a quanto fatto in precedenza, sono state implementate le seguenti API aggiuntive:

- /add_values: consente di richiamare la funzione add_values. Vengono fornititi in input l'email dell'utente, il codice dell'aeroporto di interesse, e i due valori soglia high_value e low_value. E' possibile inserire anche uno tra i due valori. La logica della funzione è già stata analizzata nella sezione precedente. In caso di successo viene restituito un json contenente una stringa di modifica confermata, altrimenti una stringa di errore.

5. Diagrammi di interazione

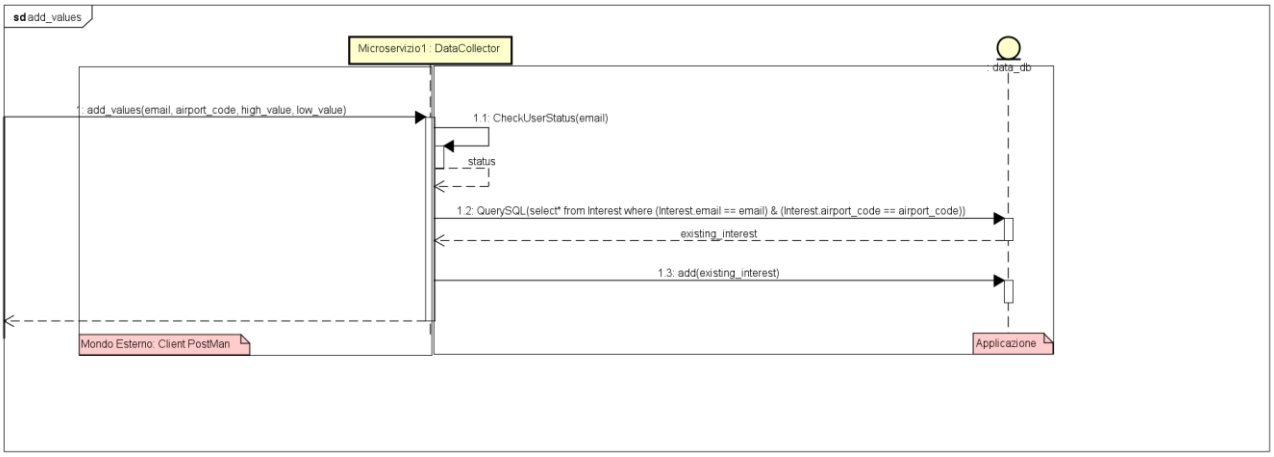


Figura 2: add_values

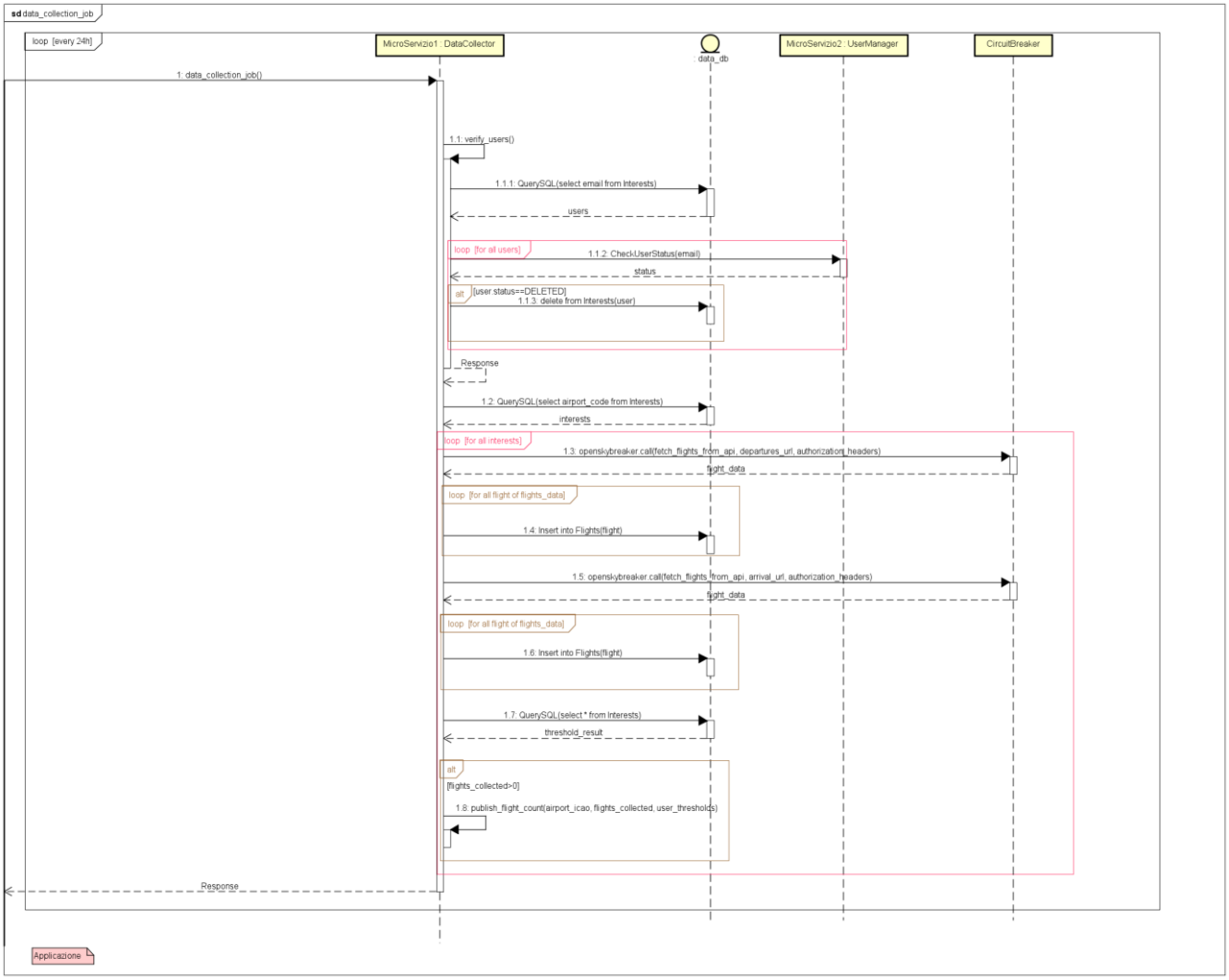


Figura 3: data_collection_job

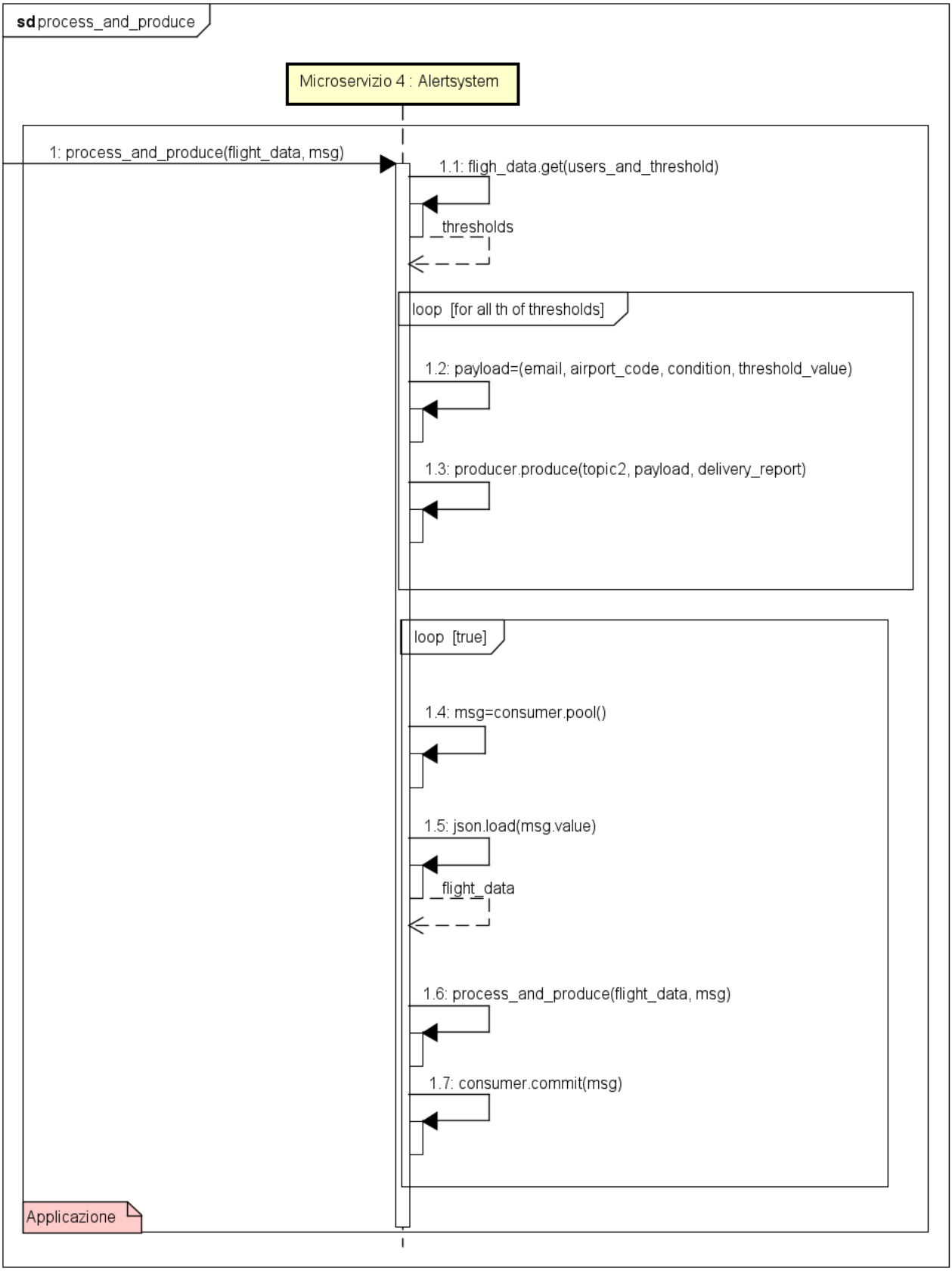


Figura 4: process_and_produce

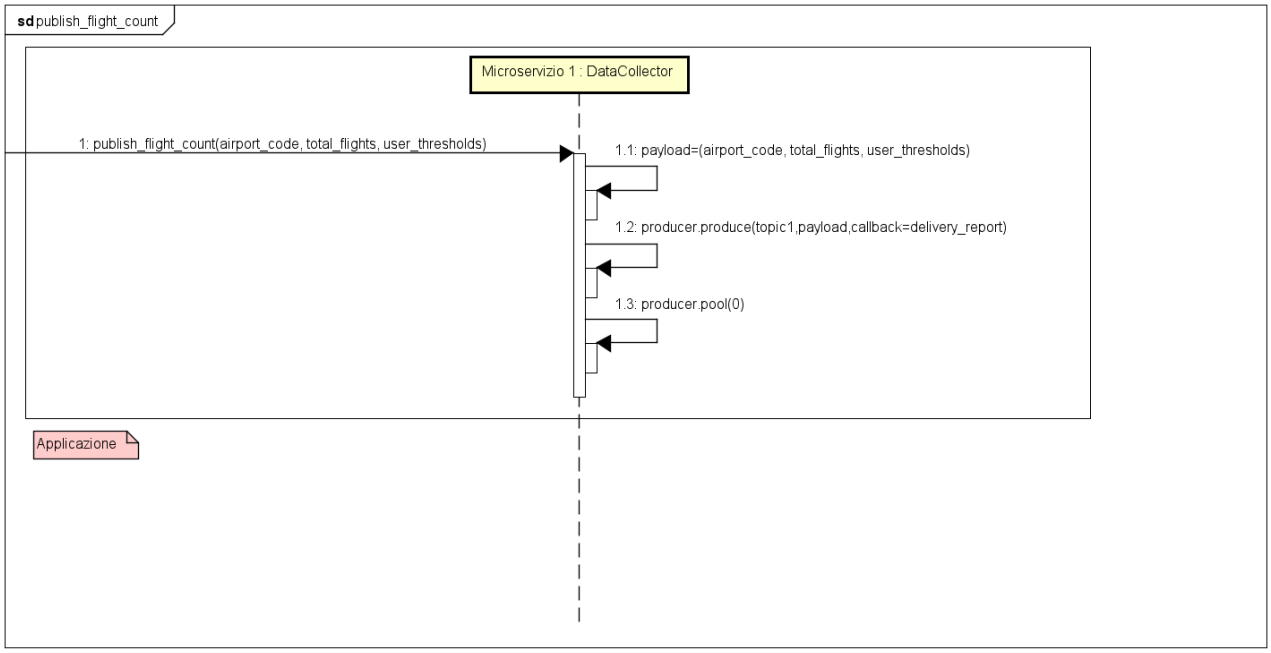


Figura 5: publish_flight_count

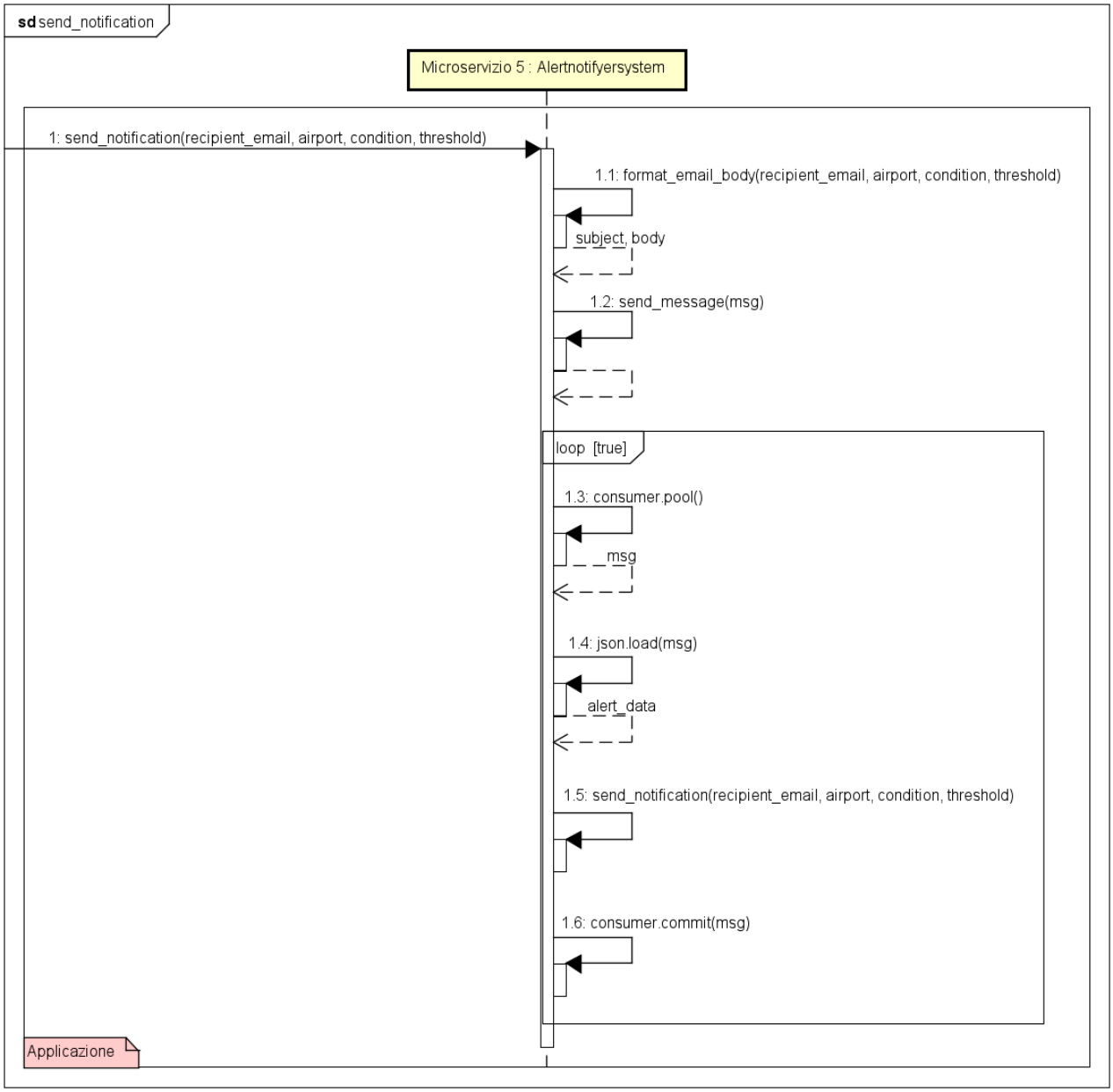
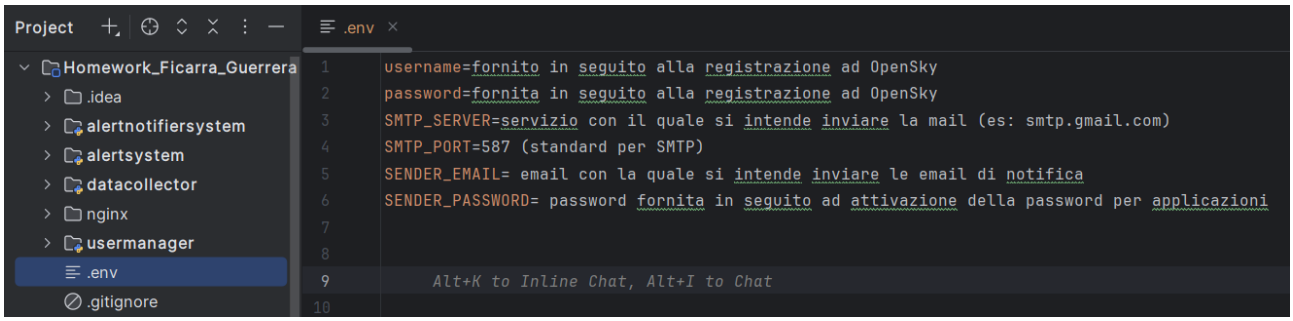


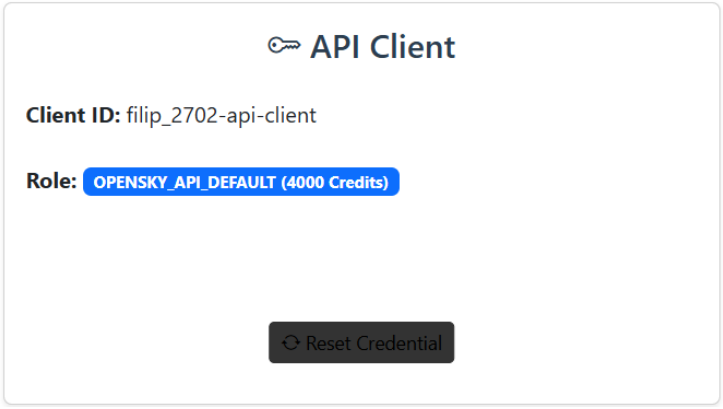
Figura 6: send_notification

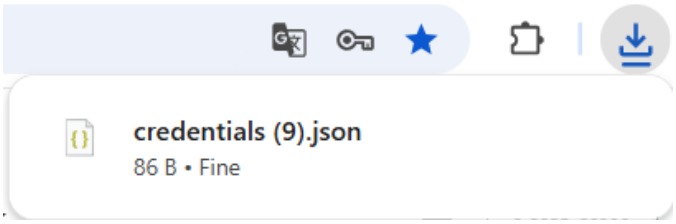
6. Test dell'applicazione e risultati

Per effettuare un test del funzionamento dell'applicazione, è opportuno creare, all'interno della repository principale, un file .env in cui inserire le proprie credenziali:



Per ottenere username e password, occorre registrarsi al sito OpenSky Network, e ottenere, cliccando su "Reset Credential" nella sezione "API Client" un json contenente il client-id (username) e il client secret (password).

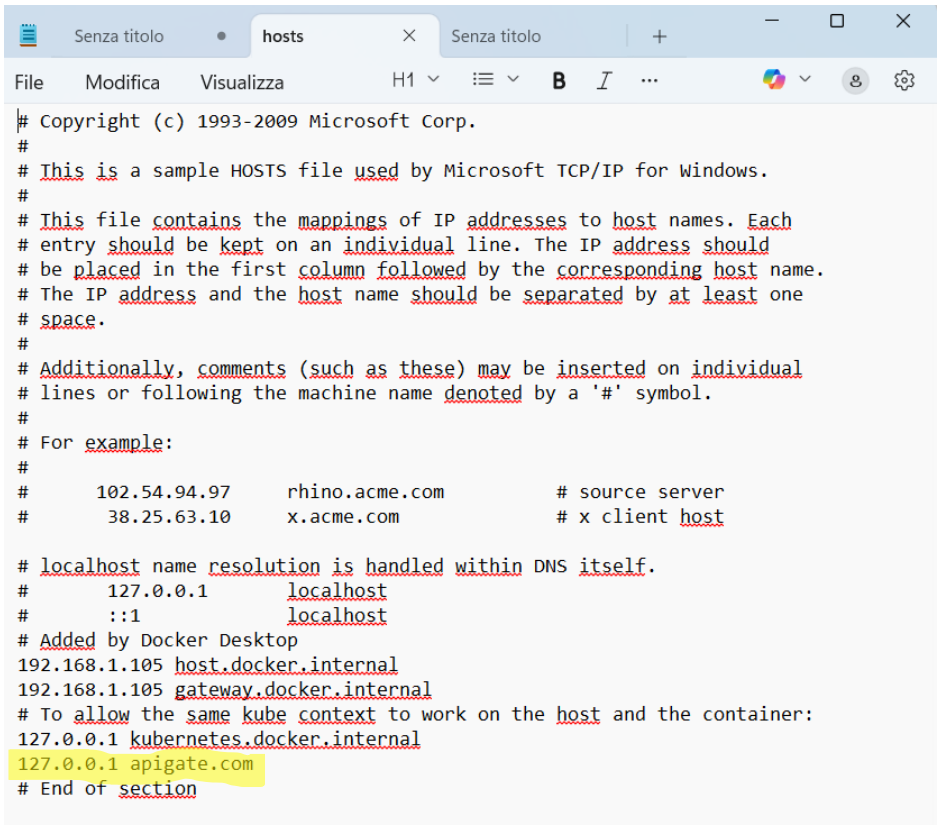




Per ottenere, SENDER_PASSOWRD, supponendo che si desideri inviare le mail di notifica attraverso GMAIL, occorre cliccare sul seguente link: Account SMTP Google

Da qui, si attiva la password per applicazioni, da inserire nel campo relativo alla SENDER_PASSWORD.

Dopo aver settato il file .env, per garantire il funzionamento di NGINX, occorre cercare il path C:\Windows\System32\drivers\etc, aprire il file hosts tramite blocco note, e inserire la riga "127.0.0.1 apigate.com", come nel seguente screen:



Una volta assegnato al DNS locale, tramite il file hosts, il dominio apigate.com con l'indirizzo di localhost (127.0.0.1), dobbiamo inserire i certificati SSL nella directory nginx/ .

Per fare ciò bisogna prima creare una cartella SSL_CERTIFICATE all'interno della directory nginx del progetto. Successivamente, ci spostiamo sul pathname nginx/SSL_CERTIFICATE ed eseguiamo i seguenti comandi

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \

  -keyout nginx-selfsigned.key \

  -out nginx-selfsigned.crt \

  -subj "/C=IT/ST=Italy/L=CT/O=Uni/OU=Dev/CN=localhost"
```

Di seguito si riportano gli screen e logs (relativi ai microservizi e ai topics coinvolti nell'applicazione) di alcune funzionalità dell'applicazione. Onde evitare una lunghezza eccessiva, non sono riportati gli screen di tutte le funzioni, ma di alcune per le quali si desidera porre una maggiore attenzione, considerando le aggiunte effettuate.

Nella maggior parte dei casi si è scelto di riportare screen che raffigurano gli scenari di successo, onde evitare di allungare troppo la documentazione.

Si noti come, nelle seguenti immagini, gli URL usati per invocare le opportune funzioni, rispettano il funzionamento di NGINX, e di come viene utilizzato "apigate.com" al posto dell'indirizzo del microservizio.

1.1 add_interest (scenario di successo)

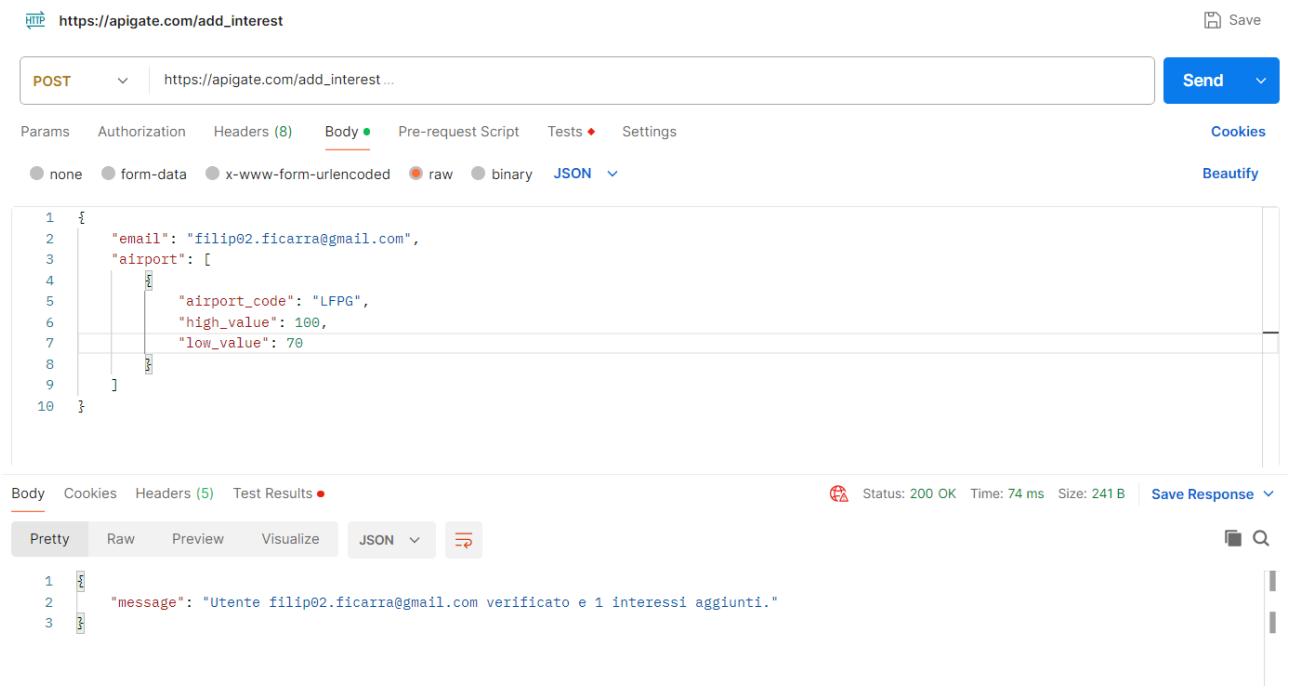


Figura 7: add_interest (con high_value e low_value)

E' possibile anche inserire l'interesse senza specificare high_value e low_value:

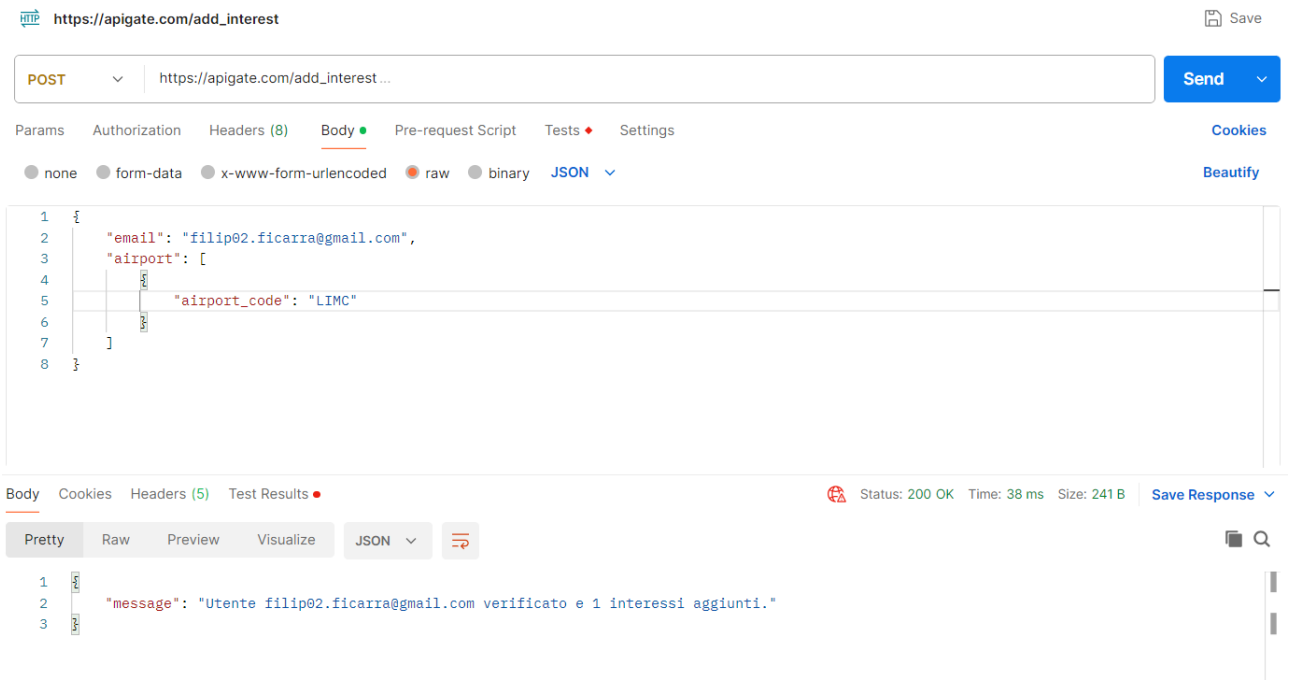


Figura 8: add_interest (senza hig_value e low_value)

1.2 interest

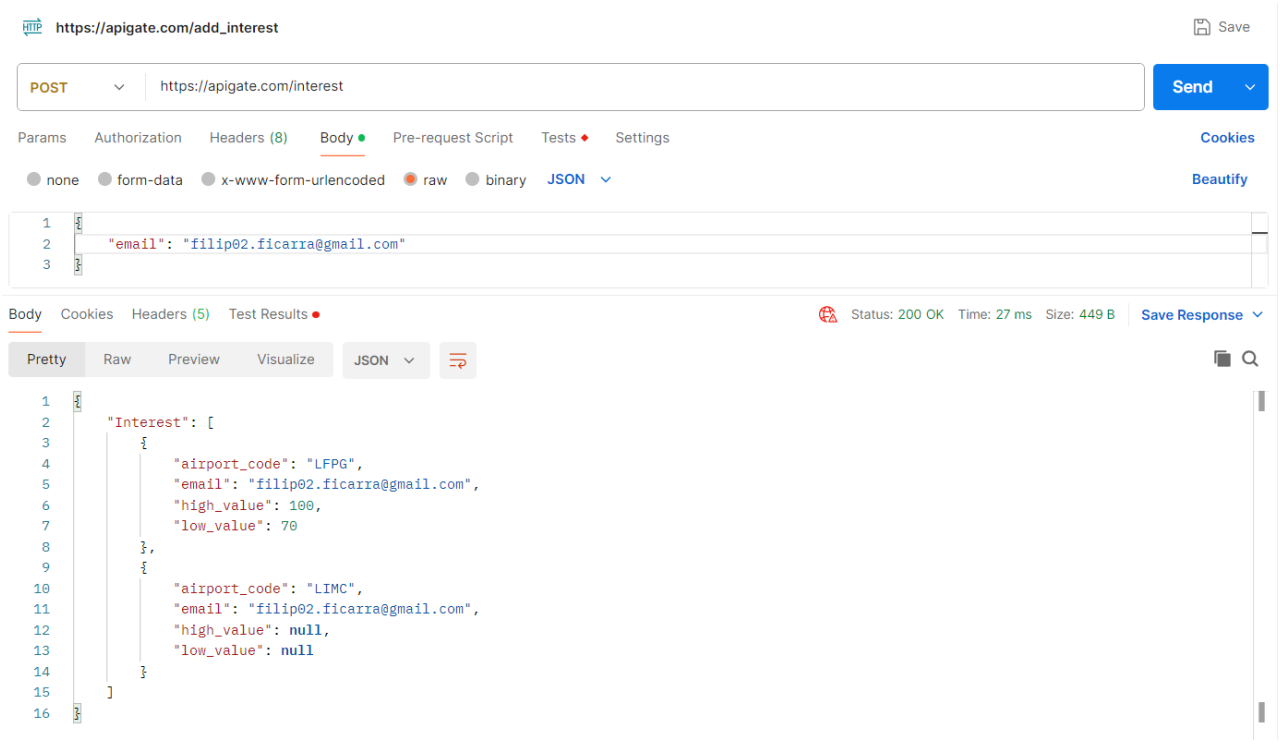


Figura 9: interests

1.3 add_values

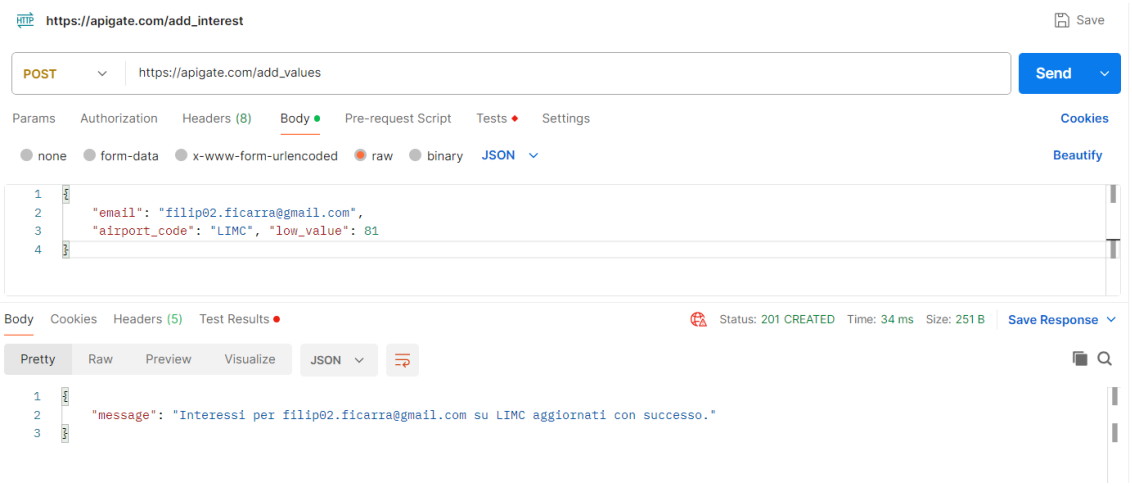


Figura 10: add_values

Si nota come è anche possibile inserire un solo valore tra le due soglie.

```
data_db=# SELECT * FROM interests;
           email           | airport_code | high_value | low_value
---------------------------+--------------+------------+-----------
 filip02.ficarra@gmail.com | LFPG         |        100 |        70
 filip02.ficarra@gmail.com | LIMC         |            |        81
(2 rows)
```

1.4 add_values (scenario in cui non è presente un interesse con quell'airport_code)

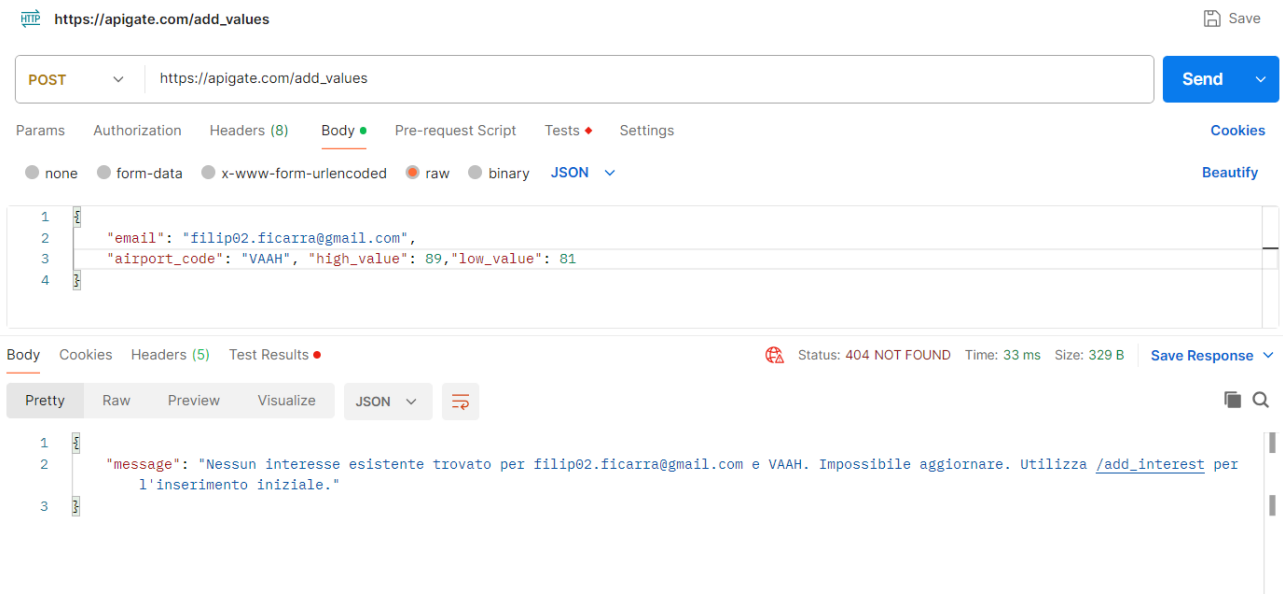


Figura 11: add_values (airport_code non presente)

1.5 add_values (scenario in cui la condizione high_value>low_value non è soddisfatta)

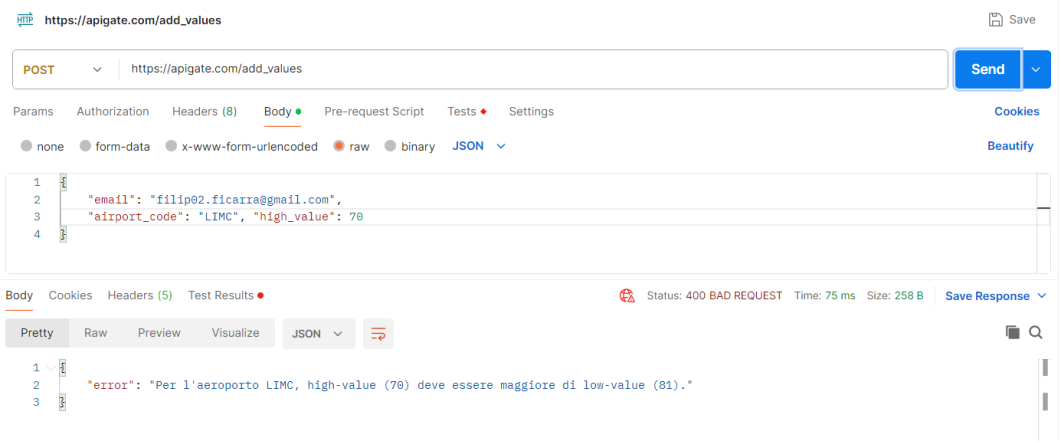


Figura 12: add_values (high_value<low_value)

Si noti come il confronto viene eseguito anche se si inserisce uno solo tra i valori, in quanto la funzione, in assenza del valore fornito, raccoglie il vecchio valore (se presente).

1.6 data_collection_job()

```
datacollectorcontainer  | Crediti OpenSky Rimanenti: 2530
datacollectorcontainer  | Crediti OpenSky Rimanenti: 2500
datacollectorcontainer  | Salvati 33 voli totali per LIMC.
datacollectorcontainer  | /app/app.py:126: DeprecationWarning: datetime.datetime.utcnow() is deprecated and scheduled for removal in a future version. Use timezone-aware objects t
o represent datetimes in UTC: datetime.datetime.now(datetime.UTC).
datacollectorcontainer  |   "timestamp": datetime.utcnow().isoformat(),
datacollectorcontainer  | [KAFKA PRODUCER] Inviati dati su LIMC a to-alert-system.
```

I dati prodotti dalla data_collection_job() sono scritti correttamente all'interno del topic

In questo caso, durante la scrittura tra un messaggio e un altro, la quale avviene ogni 5


```
{"airport_code": "LIMC", "total_flights": 33, "timestamp": "2025-12-15T21:03:56.127233", "users_and_thresholds": [{"email": "filip02.ficarra@gmail.com", "high_value": null, "low_v
alue": 81}]}
{"airport_code": "LIMC", "total_flights": 34, "timestamp": "2025-12-15T21:09:11.416739", "users_and_thresholds": [{"email": "filip02.ficarra@gmail.com", "high_value": 100, "low_va
lue": 81}]}
{"airport_code": "LIMC", "total_flights": 33, "timestamp": "2025-12-15T21:16:07.890510", "users_and_thresholds": [{"email": "filip02.ficarra@gmail.com", "high_value": 100, "low_va
lue": 81}]}
{"airport_code": "LIMC", "total_flights": 35, "timestamp": "2025-12-15T21:21:21.487658", "users_and_thresholds": [{"email": "filip02.ficarra@gmail.com", "high_value": 50, "low_val
ue": 40}]}
{"airport_code": "LIMC", "total_flights": 32, "timestamp": "2025-12-15T21:26:35.430923", "users_and_thresholds": [{"email": "filip02.ficarra@gmail.com", "high_value": 20, "low_val
ue": 10}]}
{"airport_code": "LIMC", "total_flights": 30, "timestamp": "2025-12-15T21:31:50.377340", "users_and_thresholds": [{"email": "filip02.ficarra@gmail.com", "high_value": 90, "low_val
ue": 10}]}
```


minuti, dal momento che la variabile d'ambiente PERIODO è stata impostata a 300 secondi, sono stati modificate le soglie, con la funzione add_values(), per testare i diversi scenari.

1.7 Lettura dei messaggi dell'AlertSystem e process_and_produce()

```
alertsystemcontainer  | [RECEIVED] Dati volo da DC: LIMC (33 voli)
alertsystemcontainer  | Check and notify con 1 profili
alertsystemcontainer  | [ALERT PRODUCER] Notifica inviata per filip02.ficarra@gmail.com: soglia SUPERATA_SOGLIA_BASSA su LIMC
alertsystemcontainer  | [COMMIT SUCCESS] Messaggio 0 elaborato e offset committato.
alertsystemcontainer  | [RECEIVED] Dati volo da DC: LIMC (34 voli)
alertsystemcontainer  | Check and notify con 1 profili
alertsystemcontainer  | [ALERT PRODUCER] Notifica inviata per filip02.ficarra@gmail.com: soglia SUPERATA_SOGLIA_BASSA su LIMC
alertsystemcontainer  | [COMMIT SUCCESS] Messaggio 1 elaborato e offset committato.
alertsystemcontainer  | [RECEIVED] Dati volo da DC: LIMC (33 voli)
alertsystemcontainer  | Check and notify con 1 profili
alertsystemcontainer  | [ALERT PRODUCER] Notifica inviata per filip02.ficarra@gmail.com: soglia SUPERATA_SOGLIA_BASSA su LIMC
alertsystemcontainer  | [COMMIT SUCCESS] Messaggio 2 elaborato e offset committato.
alertsystemcontainer  | [RECEIVED] Dati volo da DC: LIMC (35 voli)
alertsystemcontainer  | Check and notify con 1 profili
alertsystemcontainer  | [ALERT PRODUCER] Notifica inviata per filip02.ficarra@gmail.com: soglia SUPERATA_SOGLIA_BASSA su LIMC
alertsystemcontainer  | [COMMIT SUCCESS] Messaggio 3 elaborato e offset committato.
alertsystemcontainer  | [RECEIVED] Dati volo da DC: LIMC (32 voli)
alertsystemcontainer  | Check and notify con 1 profili
alertsystemcontainer  | [ALERT PRODUCER] Notifica inviata per filip02.ficarra@gmail.com: soglia SUPERATA_SOGLIA_ALTA su LIMC
alertsystemcontainer  | [COMMIT SUCCESS] Messaggio 4 elaborato e offset committato.
```

I messaggi ricevuti ed elaborati vengono scritti correttamente all'interno del topic "to-notifier":

```
{"email": "filip02.ficarra@gmail.com", "airport_code": "LIMC", "condition_met": "SUPERATA_SOGLIA_BASSA", "threshold_value": 81}
{"email": "filip02.ficarra@gmail.com", "airport_code": "LIMC", "condition_met": "SUPERATA_SOGLIA_BASSA", "threshold_value": 81}
{"email": "filip02.ficarra@gmail.com", "airport_code": "LIMC", "condition_met": "SUPERATA_SOGLIA_BASSA", "threshold_value": 81}
{"email": "filip02.ficarra@gmail.com", "airport_code": "LIMC", "condition_met": "SUPERATA_SOGLIA_BASSA", "threshold_value": 40}
{"email": "filip02.ficarra@gmail.com", "airport_code": "LIMC", "condition_met": "SUPERATA_SOGLIA_ALTA", "threshold_value": 20}
```

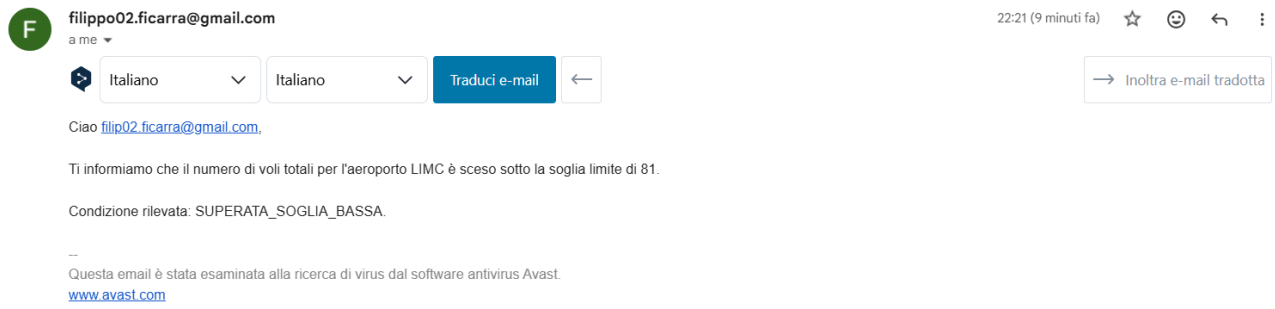
Si noti come, nel caso in cui le soglie non sono superate, non viene scritto alcun messaggio nel topic "to-notifier"

```
alertsystemcontainer  | [RECEIVED] Dati volo da DC: LIMC (30 voli)
alertsystemcontainer  | Check and notify con 1 profili
alertsystemcontainer  | [COMMIT SUCCESS] Messaggio 5 elaborato e offset committato.
```

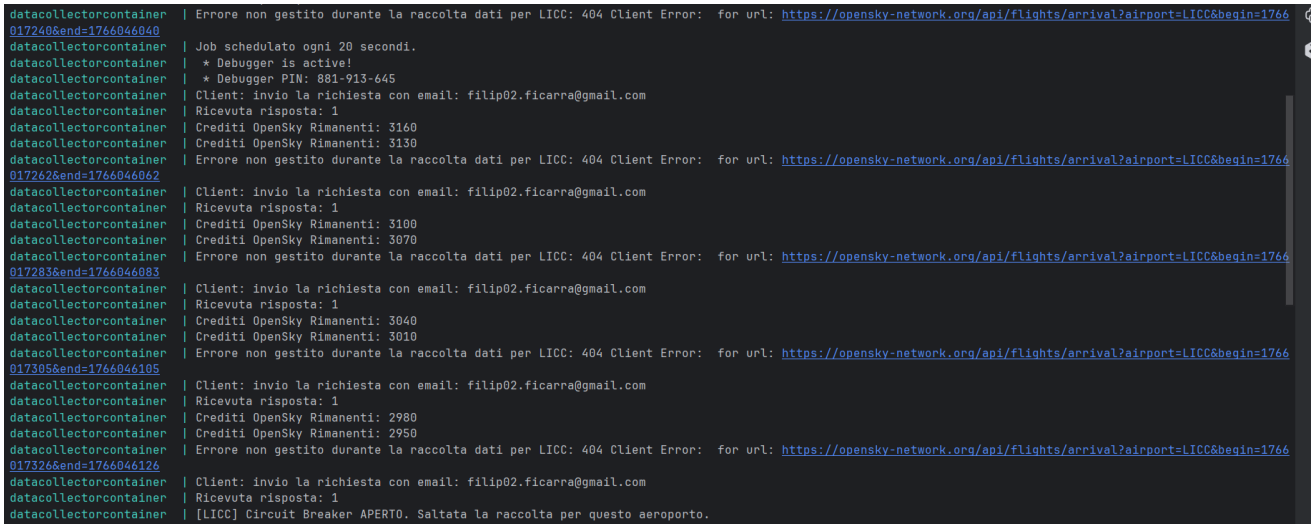
1.8 Lettura dei messaggi dell'AlertNotifierSystem e send_notification()

I messaggi vengono letti correttamente dall' AlertNotifierSystem, e la mail viene inviata

```
alertnotifiersystemcontainer  | [EMAIL SUCCESS] Notifica inviata a filip02.ficarra@gmail.com.
alertnotifiersystemcontainer  | [EMAIL SUCCESS] Notifica inviata a filip02.ficarra@gmail.com.
alertnotifiersystemcontainer  | [EMAIL SUCCESS] Notifica inviata a filip02.ficarra@gmail.com.
alertnotifiersystemcontainer  | [EMAIL SUCCESS] Notifica inviata a filip02.ficarra@gmail.com.
alertnotifiersystemcontainer  | [EMAIL SUCCESS] Notifica inviata a filip02.ficarra@gmail.com.
```

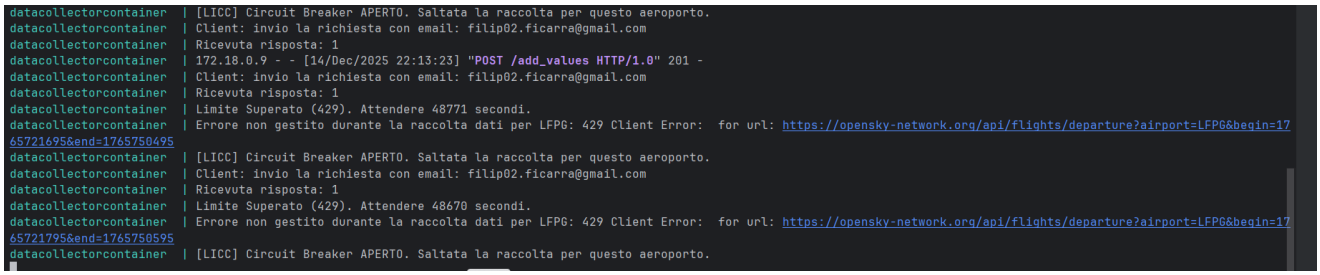


1.9 Attivazione dello stato OPEN del CircuitBreaker



In questo test del CircuitBreaker, il PERIODO è impostato a 20 secondi. Il funzionamento è corretto, in quanto, dopo il fallimento di 5 richieste consecutive, con conseguente superamento della soglia, il CircuitBreaker passa allo stato OPEN

In quest'altro test, eseguito in un momento differente (e con un PERIODO diverso) rispetto a quanto mostrato in precedenza, si vede come, Dopo il recovery_time di 30 secondi, il CircuitBreaker passa allo stato HALF_OPEN, e ricomincia ad inviare le richieste, ma se questa fallisce, ritorna allo stato di OPEN



7. Scelte progettuali

L'applicazione, basata sul paradigma dei sistemi distribuiti, utilizza varie tecnologie allo scopo di rispettare le specifiche assegnate.

Si propone di REST API e gRPC per le interazioni sincrone. Nello specifico viene impiegato REST API come interfaccia fra il client, Postman, e il sistema, per permettere agli utenti di utilizzare i servizi messi a disposizione. REST API permette grazie al formato JSON una maggiore flessibilità, inoltre, ogni richiesta REST è stateless, cioè priva di stato, il che ci permette di rispondere ad ogni singola richiesta senza dover richiedere ulteriori informazioni. Per la comunicazione interna fra microservizi, come nel caso del DataCollector che interroga lo UserManager, si è scelto di utilizzare gRPC. La scelta è motivata dal fatto che per la comunicazione fra microservizi, a differenza di REST che permette di scambiare solo dati formato JSON, gRPC permette, grazie ai Protobuf, la serializzazione dei dati risultando più efficiente, riducendo la banda utilizzata e velocizzando le comunicazioni.

Per quanto riguarda le interazioni asincrone si è scelto di impiegare Apache Kafka. Questo sistema di messaggistica permette ai microservizi di comunicare senza che ognuno sappia dell'esistenza dell'altro, ciò permette un alto disaccoppiamento. Se per esempio il microservizio AlertNotifierSystem dovesse cadere i messaggi a lui destinati non vengono persi ma memorizzati. I microservizi si limitano solamente a pubblicare messaggi nei topic in cui altri microservizi si occuperanno di consumare quei messaggi. Kafka permette l' ordinamento corretto dei messaggi in modo da ricevere le notifiche di soglia in maniera coerente ai dati raccolti, garanzia dell' invio grazie al commit dei messaggi che avviene solo dopo che il messaggio è stato processato (es. commit dopo aver inviato l'email di notifica), persistenza e durability dei messaggi anche quando i producer/consumer dovessero cadere.

L'app dispone di un single-entry point implementato con Nginx: il servizio funge da API gateway e Reverse Proxy nascondendo tutta la complessità del sistema e la granularità dei servizi. Nginx permette di disaccoppiare il client dai dettagli implementativi fornendo un'interfaccia unificata e sicura grazie a SSL. Il microservizio Nginx conosce i nomi dei microservizi in maniera dettagliata e grazie alle funzioni di reverse proxy renderizza il client al microservizio corretto, anche se quest'ultimo dovesse cambiare posizione fisica (IP). L'utilizzo di SSL, infine, ci permette di cifrare/decifrare i dati grazie alla chiave privata contenuta in nginx-selsigned.key e

alla chiave pubblica contenuta nel certificato nginx-selfsigned.crt. Le richieste in arrivo alla porta 443, grazie al protocollo TLS, vengono quindi cifrate grazie ad una chiave di sessione scambiata fra le parti usando una cifratura asimmetrica, garantendo riservatezza alle comunicazioni.

Per prevenire la creazione di richieste di registrazione duplicate si è scelto di implementare una politica at-most-once grazie ad una chiave di idempotenza (l'hash dell'email) conservata per un tempo limitato nel database del sistema.

Per proteggere il sistema da malfunzionamenti dell'API esterna è stato implementato un circuit-breaker nel DataCollector. Il circuit-breaker previene il cascading failure: cioè che un guasto dell'API faccia, a cascata, guastare l'intero servizio; permette di risparmiare le risorse facendo in modo che il sistema riprovi a fare richieste solo dopo un certo intervallo di tempo; infine, permette di riprendere il servizio dopo un guasto temporaneo dell' API esterna.

8. Indice delle figure