UNIVERSITÀ
degli STUDI
di CATANIA

DIPARTIMENTO DI
INGEGNERIA ELETTRICA ELETTRONICA ED INFORMATICA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

RELAZIONE TECNICA

HW1 - MICROSERVIZI PER GESTIONE UTENTI E MONITORAGGIO VOLI AEREI

Alunni:
Arena Giorgia
Tornabene Alessio

Docenti:
Di Stefano Antonella
Morana Giovanni
PhD:
Genovese Alessandro

ANNO ACCADEMICO 2025 – 2026

INDICE

1 INTRODUZIONE

1.1 Contesto e Obiettivo del Progetto

Il presente progetto, sviluppato nell'ambito del corso di *Sistemi Distribuiti e Big Data*, ha come obiettivo la progettazione e l'implementazione di un sistema software distribuito basato su un'architettura a microservizi. Lo scopo principale dell'applicazione è fornire una piattaforma scalabile e modulare per la gestione anagrafica degli utenti e il monitoraggio automatizzato del traffico aereo in tempo reale.

Il sistema risponde alla necessità di integrare dati eterogenei: da un lato la gestione strutturata e transazionale degli utenti, dall'altro l'acquisizione e l'analisi di flussi di dati volatili relativi ai voli aerei, ottenuti tramite integrazione con il servizio esterno *OpenSky Network*. L'architettura è stata progettata per garantire isolamento, manutenibilità e resilienza, sfruttando i moderni paradigmi di containerizzazione.

1.2 Funzionalità Principali

Il sistema offre due macro-funzionalità distinte ma interconnesse:

- **Gestione Utenti (User Manager Microservice):** Permette la registrazione e la cancellazione degli utenti, garantendo l'unicità delle registrazioni tramite politiche di consistenza dei dati (*At-Most-Once*).
- **Monitoraggio Voli (Data Collector Microservice):** Consente agli utenti registrati di sottoscrivere interessi specifici (monitoraggio di determinati aeroporti). Il sistema interroga periodicamente le API di OpenSky Network per raccogliere dati sui voli in partenza e offre funzionalità di analisi statistica (es. recupero dell'ultimo volo, calcolo della media dei voli giornalieri).

1.3 Stack Tecnologico

Per realizzare un'infrastruttura robusta e aderente agli standard industriali, sono state selezionate le seguenti tecnologie:

- **Linguaggio di Sviluppo:** Python 3.9, scelto per la sua versatilità e il vasto ecosistema di librerie per il networking e la gestione dati.

- **Containerizzazione & Orchestrazione:** Docker e Docker Compose, utilizzati per garantire la portabilità dell'applicazione e semplificare il deployment dei servizi e dei database.
- **Protocolli di Comunicazione:**
    - **REST (tramite Flask):** Per l'esposizione delle API pubbliche verso i client esterni.
    - **gRPC:** Per la comunicazione inter-processo (IPC) ad alta efficienza tra i microservizi interni.
- **Persistenza dei Dati (Polyglot Persistence):**
    - **PostgreSQL:** Database relazionale (RDBMS) per la memorizzazione strutturata e ACID-compliant (Atomicità, Coerenza, Isolamento, Durabilità) dei dati utente.
    - **MongoDB:** Database NoSQL per la gestione flessibile e performante dei dati di volo non strutturati (JSON).
- **Integrazione Esterna:** Libreria *requests* per l'interazione con le REST API di OpenSky Network, con implementazione di meccanismi di autenticazione OAuth2.0 e strategie di fallback.

2 ARCHITETTURA DEL SISTEMA

Il sistema progettato segue un approccio architetturale a **microservizi**, privilegiando il disaccoppiamento funzionale e la scalabilità dei singoli componenti. L'intera infrastruttura è containerizzata mediante **Docker**, garantendo portabilità e isolamento tra ambienti di sviluppo e produzione.
L'orchestrazione dei servizi è gestita tramite **Docker Compose** (versione 3.8), che definisce la topologia della rete virtuale condivisa, la gestione dei volumi persistenti per i database (*database_postgres*, *database_mongo*) e le dipendenze di avvio tra i container.

2.1 Componenti del Sistema

Il sistema si articola in quattro container principali, ciascuno con una responsabilità specifica:

1. **User Manager Service** (user-manager)
    - **Ruolo:** È il microservizio responsabile dell'identità e dell'anagrafica centralizzata. Gestisce il ciclo di vita degli utenti (registrazione e cancellazione) e funge da autorità di validazione per gli altri servizi.
    - **Stack Tecnologico:** Sviluppato in Python 3.9 (immagine python:3.9-slim) con framework **Flask**. Utilizza la libreria *psycopg* per l'interazione diretta e performante con il database relazionale, implementando meccanismi di *retry logic* all'avvio per garantire la connessione al DB.
    - **Interfacce Esposte:**
        - **REST API (Porta 5000):** Espone endpoint HTTP per l'interazione con client esterni (es. Postman) per le operazioni CRUD sugli utenti.
        - **gRPC Server (Porta 50051):** Espone la procedura remota CheckUser, utilizzata dal Data Collector per verificare l'esistenza

di un utente prima di consentire operazioni sensibili (es. aggiunta interesse).

2. **Data Collector Service** (data-collector)
    - **Ruolo:** È il cuore operativo del sistema di monitoraggio. Gestisce le preferenze di tracciamento degli utenti, l'acquisizione dati e le statistiche.
    - **Stack Tecnologico:** Sviluppato in Python con framework **Flask** e driver *pymongo*. Integra un sistema di **caching multi-thread** per ottimizzare le risposte e un modulo di *scheduling* (thread daemon) per l'esecuzione di job periodici in background.
    - **Funzionalità Chiave:**
        - **Integrazione OpenSky:** Interroga le API esterne di *OpenSky Network* (/flights/departure) utilizzando un'autenticazione OAuth2 (Client Credentials) per ottenere un *Bearer Token* e accedere ai dati storici.
        - **Fault Tolerance:** Implementa una logica di robustezza nel metodo fetch_opensky_data: in caso di errori di rete o codici HTTP 4xx/5xx dal provider esterno, il servizio attiva automaticamente una procedura di *fallback* generando dati di volo simulati (mock), garantendo la continuità operativa del sistema.
        - **Doppia Natura gRPC:** Agisce sia come **Client gRPC** (verso lo User Manager per CheckUser) sia come **Server gRPC** (sulla porta 50052) esponendo il metodo DeleteData, invocato dallo User Manager per la pulizia a cascata di tutti i dati quando un utente viene eliminato.
3. **User DB** (user-db)
    - **Tecnologia: PostgreSQL 15** (immagine postgres:15-alpine).
    - **Funzione:** Garantisce la persistenza strutturata e transazionale dei dati utente. La tabella users è progettata con vincoli di integrità (Primary Key su email), fondamentali per soddisfare il requisito di unicità.

4. **Data DB** (data-db)
    - **Tecnologia: MongoDB 6.0** (immagine mongo:6.0).
    - **Funzione:** Gestisce la memorizzazione di grandi volumi di dati eterogenei (documenti JSON) relativi ai voli e agli interessi (*interests, flights*). Utilizza indici specifici sui campi airport e timestamp per ottimizzare le query di aggregazione statistica (es. calcolo media voli).

2.2 Diagramma Architetturale e Flussi di Comunicazione

Il diagramma seguente illustra i flussi di comunicazione implementati nel codice:

- **REST (HTTP):** Traffico esterno tra Client e Servizi (porte 5000, 5001).
- **gRPC (Protocol Buffers):**
    - Canale 1: Data Collector User Manager (Porta 50051) per validazione utente.
    - Canale 2: User Manager Data Collector (Porta 50052) per propagazione cancellazione dati.
- **Database Protocol (TCP):** Connessioni persistenti sulle porte standard 5432 e 27017.
- **HTTPS (External):** Chiamate autenticate verso le API di OpenSky Network.

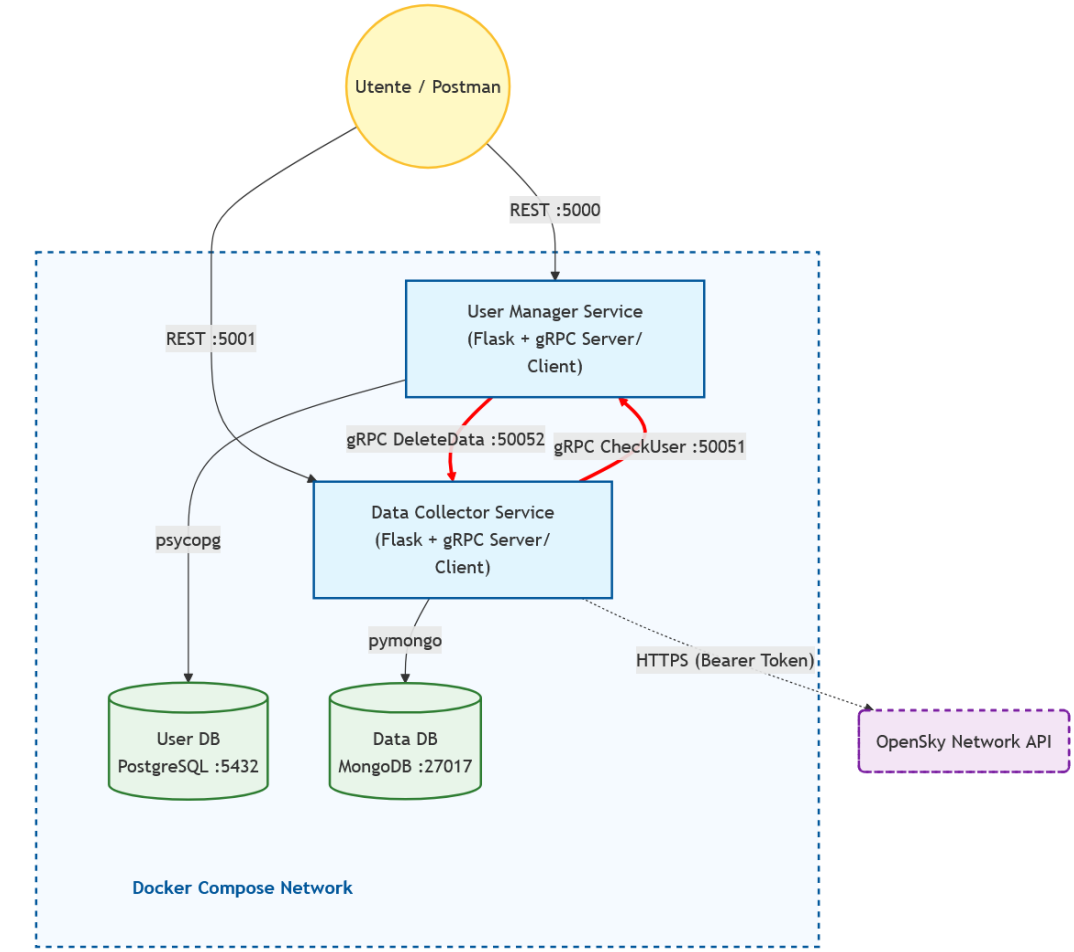


Figura 1_Diagramma_Architettura_Microservizi_e_Flussi_di_Comunicazione

3 SCELTE PROGETTUALI

3.1 Strategia di "Polyglot Persistence"

Per soddisfare in modo ottimale requisiti funzionali eterogenei, il sistema adotta un approccio di *Polyglot Persistence*, assegnando a ciascun microservizio la tecnologia di database più idonea alla natura dei dati trattati. Questa separazione è orchestrata tramite container Docker dedicati (postgres:15-alpine e mongo:6.0), garantendo isolamento e scalabilità indipendente.

3.1.1 PostgreSQL (User Manager Service)

Per il microservizio di gestione identità (*User Manager*), la scelta è ricaduta su **PostgreSQL 15**, un RDBMS *ACID-compliant*. Dal modulo *database_postgres.py* si evidenziano le motivazioni tecniche di questa scelta:

- **Integrità Referenziale e Schema Rigido:** La gestione anagrafica richiede una struttura dati definita e immutabile per garantire la coerenza. Come implementato nel metodo crea_tabella, lo schema è definito staticamente (CREATE TABLE IF NOT EXISTS users), imponendo tipi di dato precisi (VARCHAR, TIMESTAMP).
- **Garanzia di Unicità :** Il requisito non funzionale di evitare registrazioni duplicate è stato soddisfatto a livello strutturale definendo il campo email come **PRIMARY KEY**. Questo delega al motore del database il controllo di unicità, prevenendo *race conditions* in modo più efficace rispetto a controlli applicativi.
- **Gestione Transazionale:** L'utilizzo della libreria *psycopg* permette di gestire le operazioni di inserimento in modo atomico tramite il comando *conn.commit()*, assicurando che l'utente venga creato solo se l'intera transazione ha successo.
- **Resilienza alla Connessione:** È stato implementato un meccanismo di *retry logic* nel costruttore della classe Database, che tenta la connessione fino a 10 volte con backoff di 3 secondi, garantendo che il servizio non fallisca all'avvio

nel caso in cui il container del database non sia ancora pronto (problema comune nell'orchestrazione distribuita).

3.1.2 MongoDB (Data Collector Service)

Per il servizio di monitoraggio (*Data Collector*), è stato selezionato **MongoDB 6.0**, un database NoSQL orientato ai documenti. L'implementazione in *database_mongo.py* giustifica questa scelta per i seguenti motivi:

- **Flessibilità dello Schema (Schema-less):** I dati acquisiti dalle API di *OpenSky Network* sono forniti in formato JSON e possono contenere strutture annidate complesse. MongoDB permette di persistere questi oggetti direttamente tramite il metodo *salva_voli*, memorizzando in un unico documento sia i metadati (*timestamp, aeroporto*) sia il payload eterogeneo dei voli (*"data": voli*), senza necessità di normalizzazione o mapping O/R (ORM) costosi.
- **Ottimizzazione delle Performance in Lettura:** Per garantire risposte rapide alle query statistiche, sono stati definiti indici specifici (*create_index*) sui campi airport e timestamp all'inizializzazione della connessione. Questo riduce drasticamente il tempo di scansione della collezione *flights* durante le operazioni di filtraggio.
- **Capacità di Aggregazione Server-Side:** Per funzionalità analitiche come il calcolo della media voli, il sistema sfrutta la *Aggregation Pipeline* nativa di MongoDB. Nel metodo *get_media_voli*, operazioni complesse di filtraggio ($match) e raggruppamento ($group, $sum) vengono eseguite direttamente dal motore del database, minimizzando il trasferimento dati verso l'applicazione e sfruttando la potenza di calcolo del cluster dati.
- **Gestione Concorrente:** L'utilizzo del driver *pymongo* in combinazione con la logica *upsert* (update_one con upsert=True) nel metodo *aggiungi_interesse* garantisce l'idempotenza delle preferenze utente, evitando duplicati anche in scenari di richieste concorrenti.

3.2 Comunicazione Ibrida (REST + gRPC)

L'architettura del sistema adotta un paradigma di comunicazione ibrido, distinguendo nettamente tra l'interfaccia verso l'esterno (*Public API*) e la comunicazione tra microservizi (*Inter-Service Communication*). Questa separazione è stata implementata per ottimizzare le prestazioni interne mantenendo al contempo la massima compatibilità verso i client esterni.

3.2.1 REST per l'Interfaccia Pubblica

Per l'esposizione dei servizi verso gli utenti finali e i client di test (es. Postman), è stato adottato lo stile architetturale **REST** su protocollo HTTP/1.1.

- **Implementazione:** Come visibile nel modulo app.py, il framework **Flask** gestisce le richieste HTTP, mappando le risorse su endpoint specifici (es. POST /interests, GET /flights/last).
- **Motivazione:** REST garantisce interoperabilità universale e facilità di debug. I payload JSON utilizzati nelle risposte (es. jsonify({...})) sono human-readable e facilmente consumabili da qualsiasi client web o mobile.

3.2.2 gRPC per la Comunicazione Interna

Per la comunicazione interna tra i container, il sistema abbandona il protocollo HTTP testuale in favore di **gRPC** (basato su HTTP/2 e Protocol Buffers), ottenendo vantaggi significativi in termini di latenza, efficienza di banda e tipizzazione forte dei dati.

L'aspetto più innovativo dell'architettura risiede nella natura di **Comunicazione Reciproca** dei microservizi: essi non agiscono secondo un rigido schema Client-Server ma invertono i propri ruoli in base alla procedura richiesta (Pattern Unary RPC).

A. Flusso di Validazione (Data Collector→User Manager) Quando il *Data Collector* riceve una richiesta di aggiunta interesse (add_interest in app.py):

1. Agisce come **Client gRPC**, istanziando uno stub (user_pb2_grpc.UserManagerStub).
2. Invia una richiesta sincrona CheckUser allo *User Manager* sulla porta **50051** (definita in docker-compose.yml).
3. La richiesta è fortemente tipizzata grazie al messaggio CheckUserRequest definito in user.proto, che incapsula client_id, message_id (UUID per il tracciamento) ed email.
4. Questo approccio riduce l'overhead di rete rispetto a una chiamata REST interna, fondamentale per operazioni frequenti di validazione.

B. Flusso di Consistenza Dati (User Manager→ Data Collector) Per garantire la coerenza dei dati distribuiti in caso di cancellazione di un utente, il sistema inverte i ruoli:

1. Il *Data Collector* agisce come **Server gRPC**. Nel file app.py, la classe DataCollectorGRPC implementa il servicer generato da user_pb2_grpc.DataCollectorServicer.
2. Un thread dedicato (start_grpc_server) mantiene in ascolto un server gRPC sulla porta **50052**.
3. Lo *User Manager* invoca la procedura remota DeleteData definita in user.proto.
4. Alla ricezione della chiamata, il *Data Collector* esegue la pulizia locale tramite mongo_db.rimuovi_interessi_utente(email), garantendo che non rimangano dati orfani ("Data Consistency Pattern").

3.2.3 Gestione della Concorrenza

L'implementazione di questa architettura ibrida nel *Data Collector* ha richiesto una gestione avanzata della concorrenza tramite il modulo threading di Python:

- Il **Main Thread** gestisce il server Flask (porta 5001).
- Un **Daemon Thread** esegue il server gRPC (porta 50052) per rispondere alle chiamate dello User Manager senza bloccare l'interfaccia REST.

- Un ulteriore thread gestisce il ciclo di monitoraggio (monitoraggio_ciclico), assicurando che l'acquisizione dati e la comunicazione interna non degradino le performance delle API pubbliche.

3.3 Gestione della Fault Tolerance e Resilienza gRPC

In un'architettura distribuita, la rete non è affidabile al 100%. Per gestire i **guasti transitori** (*Transient Faults*) come micro-interruzioni di rete o riavvi dei container, abbiamo implementato una strategia di resilienza avanzata basata sulla **Service Config** nativa di gRPC, evitando logiche di retry manuali nel codice.
Abbiamo configurato il canale di comunicazione con una politica di **Exponential Backoff** definita tramite JSON. I parametri chiave adottati sono:

- **timeout:** Definisce il tempo massimo totale concesso all'operazione. 5 secondi rappresentano un compromesso ideale per garantire reattività senza bloccare il client indefinitamente.
- **retryableStatusCodes:** limitiamo i retry solo quando il server è irraggiungibile. Errori logici (es. dati non validi) non vengono ritentati.
- **maxAttempts:** Numero massimo di tentativi(5), sufficiente a coprire disservizi di 3-4 secondi.
- **initialBackoff & backoffMultiplier:** Utilizziamo una crescita esponenziale (0.5s 1.0s 2.0s) per ridurre il carico sul server in fase di recupero.

Scenario di Recovery: Se lo *User Manager* invoca la cancellazione dati mentre il *Data Collector* si sta riavviando, il canale gRPC attende e riprova automaticamente secondo la policy configurata. L'operazione ha successo non appena il servizio torna online, risultando trasparente per l'utente finale.

3.4 Robustezza e Fault Tolerance (Integrazione OpenSky)

L'integrazione con il provider dati esterno *OpenSky Network* rappresenta un punto critico dell'architettura, essendo soggetta a latenze di rete, limitazioni di rate-limiting e

potenziali indisponibilità del servizio. Per mitigare questi rischi, il microservizio *Data Collector* implementa pattern architetturali specifici per la resilienza.

3.4.1 Autenticazione OAuth 2.0 (Client Credentials Flow)

A differenza delle implementazioni base che utilizzano *Basic Auth*, il sistema adotta lo standard **OAuth 2.0** per garantire un accesso sicuro e scalabile alle API.

- **Implementazione:** Come definito nella funzione *get_opensky_token*, il servizio effettua una richiesta POST all'endpoint di autenticazione OpenID Connect (https://auth.opensky-network.org/...).
- **Sicurezza:** Utilizzando il grant type client_credentials con CLIENT_ID e CLIENT_SECRET, il sistema ottiene un **Access Token** temporaneo. Questo token viene poi iniettato nell'header Authorization: Bearer ... delle richieste successive, disaccoppiando le credenziali a lungo termine dal traffico dati e migliorando la sicurezza complessiva.

3.4.2 Strategia di Fallback e Graceful Degradation

La funzione *fetch_opensky_data* è progettata per gestire il fallimento delle chiamate esterne senza impattare sulla stabilità del sistema ("Fail-Safe").

- **Gestione Eccezioni:** La chiamata HTTP verso OpenSky è incapsulata in un blocco try...except che gestisce timeout di rete (timeout=10s) ed errori di connessione.
- **Logica di Fallback:** In caso di codici di stato HTTP di errore (es. 401 Unauthorized, 429 Too Many Requests, 500 Server Error) o eccezioni di rete, il flusso di esecuzione non si interrompe. Il sistema attiva automaticamente una procedura di fallback generando un oggetto mock_flight.
    - Questo oggetto rispetta rigorosamente lo schema dati atteso dal database (campi icao24, firstSeen, estDepartureAirport), garantendo che il thread di monitoraggio (monitoraggio_ciclico) possa procedere con il

salvataggio su MongoDB senza causare crash o inconsistenze nei dati a valle.

3.5 Concorrenza e Background Jobs

Il microservizio *Data Collector* deve soddisfare requisiti operativi contrastanti: rispondere istantaneamente alle richieste API (bassa latenza) e, contemporaneamente, eseguire operazioni di rete intensive e a lunga durata (scaricamento dati da OpenSky). Per evitare che le operazioni di I/O bloccanti degradino la reattività del sistema, è stato implementato un modello di **concorrenza basato su Threading**.
L'analisi del modulo app.py evidenzia l'orchestrazione di tre flussi di esecuzione paralleli all'interno dello stesso container:

3.5.1 Main Thread: Interfaccia REST

Il *Main Thread* (thread principale del processo Python) è dedicato esclusivamente all'esecuzione del server web **Flask** (porta 5001).

- **Funzione:** Gestisce il ciclo richiesta-risposta HTTP per gli endpoint pubblici (/interests, /flights/last).
- **Isolamento:** Mantenendo questo thread libero da task pesanti, il sistema garantisce che le API rimangano sempre responsive, anche mentre in background avvengono operazioni complesse di aggiornamento dati.

3.5.2 Daemon Thread: Server gRPC e Thread Pool

Per gestire la comunicazione interna in ingresso dallo *User Manager* (es. notifiche di cancellazione dati), viene avviato un thread dedicato tramite threading.Thread(target=start_grpc_server, daemon=True).

- **Gestione della Concorrenza (Thread Pool):** All'interno di questo thread, il server gRPC è istanziato utilizzando un futures.ThreadPoolExecutor configurato con max_workers=10. Questa scelta progettuale è fondamentale: permette al

servizio di gestire fino a 10 richieste gRPC simultanee in parallelo, prevenendo colli di bottiglia nella comunicazione *inter-service* in scenari di alto carico.

- **Modalità Daemon:** Il flag daemon=True assicura che il server gRPC termini automaticamente e graziosamente alla chiusura del processo principale, semplificando il ciclo di vita del container Docker.

3.5.3 Background Worker: Monitoraggio Ciclico

Il terzo componente concorrente è il *Worker Thread* dedicato alla funzione *monitoraggio_ciclico*, anch'esso avviato come *daemon*.

- **Logica di Scheduling:** Questo thread esegue un ciclo infinito (while True) che interroga il database per ottenere gli aeroporti attivi, scarica i dati da OpenSky e li persiste su MongoDB.
- **Disaccoppiamento Temporale:** Tra un ciclo e l'altro è inserita una pausa esplicita (time.sleep(600)), che definisce la frequenza di aggiornamento (ogni 10 minuti). L'esecuzione in un thread separato è l'unica soluzione che permette di sospendere il processo di aggiornamento (sleep) senza bloccare l'intero microservizio, che altrimenti smetterebbe di rispondere alle chiamate HTTP e gRPC durante l'attesa.

4 Dettagli Implementativi

Questa sezione analizza le soluzioni tecniche adottate a livello di codice per garantire i requisiti non funzionali di unicità, consistenza e concorrenza.

4.1 Implementazione della Politica "At-Most-Once" e Idempotenza

Il requisito fondamentale di garantire che una richiesta venga elaborata al **massimo una volta** (**evitando duplicazioni** in caso di ritrasmissioni o errori di rete) è stato implementato su due livelli distinti. I due livelli distinti sono: il **livello di persistenza (Business Logic/Database)**, che garantisce l'unicità del dato tramite vincoli relazionali, e il **livello applicativo (gRPC)**, che garantisce l'idempotenza delle comunicazioni tramite un sistema di caching basato su Request-ID.

4.1.1 Livello Persistenza (Business logic/Database): Unicità Transazionale (PostgreSQL)

Per la registrazione degli utenti, la garanzia "At-Most-Once" è demandata **strutturalmente** al database PostgreSQL. Analizzando il file *database_postgres.py* (metodo crea_tabella), la tabella *users* è definita con il vincolo esplicito:

email VARCHAR(255) PRIMARY KEY

L'utilizzo della PRIMARY KEY sulla colonna email trasforma il requisito logico in un vincolo fisico:

1. Quando lo *User Manager* riceve una POST /users, tenta l'inserimento.
2. Se arriva una seconda richiesta identica (duplicata), il database solleva un'eccezione di violazione del vincolo di unicità.
3. Il codice Python intercetta questa eccezione e impedisce la creazione di una "doppia identità", garantendo che l'effetto collaterale (la registrazione) avvenga esattamente una sola volta (o zero, in caso di errore), **ma mai due**.

4.1.2 Livello Applicativo(gRPC): Deduplica tramite Request-ID e Cache

Per le comunicazioni interne via gRPC, è stato implementato un meccanismo di **idempotenza** basato su identificativi univoci, gestito dalla classe custom Cache definita nel file *cache.py*.

- Nel file user.proto, i messaggi di richiesta (es. CheckUserRequest) sono stati estesi con due campi fondamentali:
  - string client_id = 1; (Identificativo del chiamante)
  - string message_id = 2; (UUID univoco della singola richiesta) Questo permette di distinguere una *nuova richiesta* da un *retry* di una richiesta precedente.
- **Logica di Caching (cache.py):** La classe Cache implementa un dizionario in-memory (*self.cache*) protetto da un **Mutex** (threading.Lock()) per garantire la *thread-safety* in ambiente concorrente. Il flusso di eliminazione dei duplicati funziona come segue:
  1. **Check:** Prima di eseguire una logica onerosa, il server interroga la cache con la chiave composta f"{client_id}:{message_id}" (metodo get_response).
  2. **Hit:** Se la chiave esiste, significa che la richiesta è già stata elaborata. Il sistema restituisce immediatamente la risposta salvata (return data['response']), realizzando la semantica *At-Most-Once* **senza ri-eseguire l'operazione**.
  3. **Miss & Save:** Se la chiave non esiste, il server elabora la richiesta e salva il risultato tramite *save_response*, associandovi un timestamp.
- **Gestione della Memoria (TTL):** Essendo la nostra applicazione poco costosa in termini computazionali, abbiamo optato per una cache volatile, piuttosto che mantenere i dati in una memoria onerosa. Per evitare che la cache cresca indefinitamente, il costruttore *__init__* avvia un thread daemon (*self.pulizia*) che esegue il metodo *pulisci_cache*. Questo *thread* interno scansiona periodicamente il dizionario e rimuove le voci più vecchie del *Time-To-Live* configurato (300 secondi).

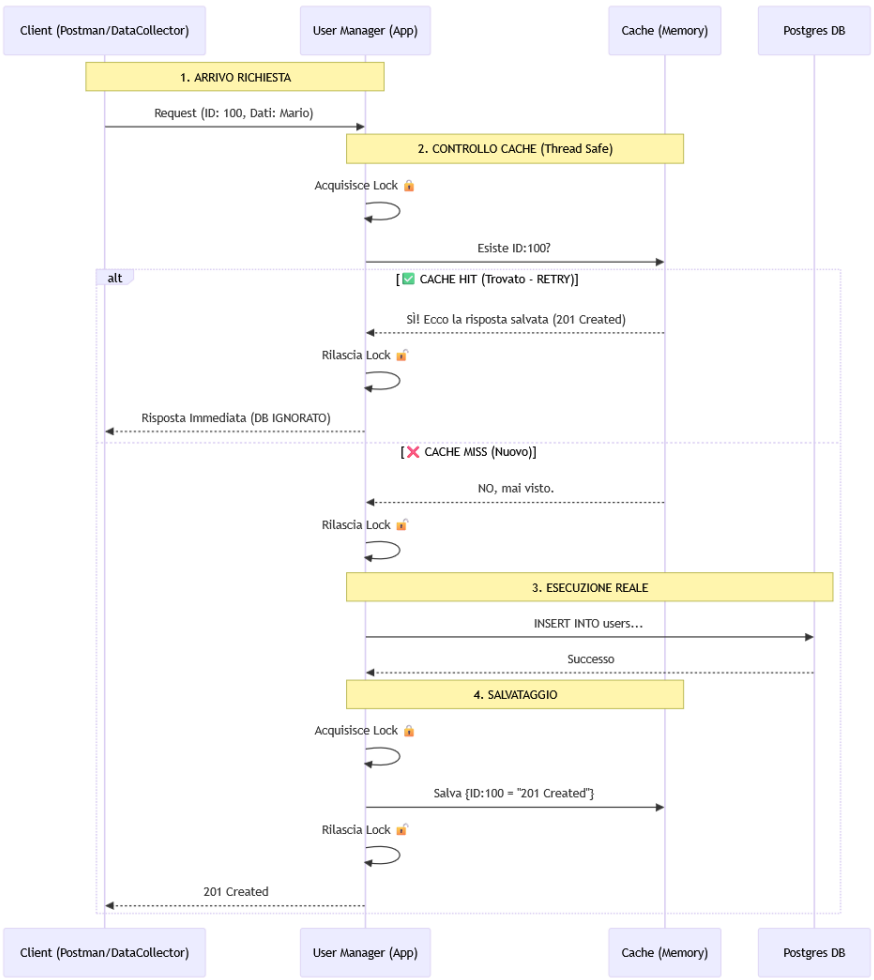
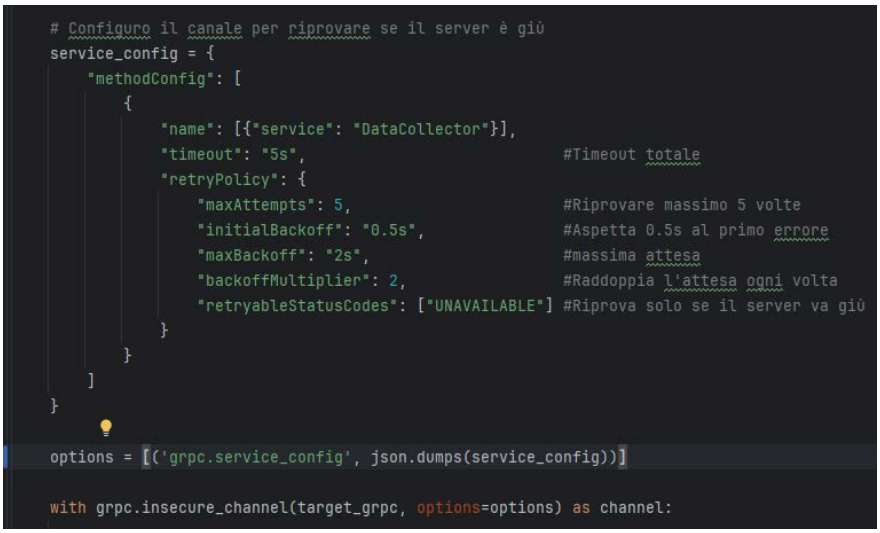


Figura 2_Politica_di_funzionamento_AtMostOnce_e_Cache

4.2 Configurazione del Canale gRPC (Service Config)

A livello di codice Python, la configurazione di resilienza descritta nel capitolo 3 è stata implementata iniettando un payload JSON durante la creazione del canale. È stato necessario utilizzare la libreria json per serializzare il dizionario di configurazione, poiché la libreria core gRPC (scritta in C) richiede che l'opzione grpc.service_config sia passata come stringa.

Snippet implementativo (presente sia in user_manager che data_collector):



```
# Configuro il canale per riprovare se il server è giù
service_config = {
    "methodConfig": [
        {
            "name": [{"service": "DataCollector"}],
            "timeout": "5s",                          #Timeout totale
            "retryPolicy": {
                "maxAttempts": 5,                     #Riprovare massimo 5 volte
                "initialBackoff": "0.5s",             #Aspetta 0.5s al primo errore
                "maxBackoff": "2s",                   #massima attesa
                "backoffMultiplier": 2,               #Raddoppia l'attesa ogni volta
                "retryableStatusCodes": ["UNAVAILABLE"] #Riprova solo se il server va giù
            }
        }
    ]
}

options = [('grpc.service_config', json.dumps(service_config))]

with grpc.insecure_channel(target_grpc, options=options) as channel:
```

Questa implementazione garantisce che entrambi i microservizi siano robusti sia in fase di validazione utente (add_interest) sia in fase di pulizia dati (delete_user).

4.2 Gestione della Concorrenza nel Data Collector

L'analisi del file app.py evidenzia un'architettura multi-thread avanzata per il servizio *Data Collector*, necessaria per gestire contemporaneamente traffico HTTP, gRPC e operazioni di background. All'avvio dell'applicazione vengono istanziati tre contesti di esecuzione paralleli:

1. **Main Thread (Server REST):** Il thread principale esegue app.run(), mettendo in ascolto il server Flask sulla porta 5001 per rispondere alle richieste dei client.
2. **gRPC Server Thread:** Un thread dedicato (grpc_thread), avviato come daemon, esegue la funzione start_grpc_server. Questa funzione inizializza un ThreadPoolExecutor con 10 worker per gestire le chiamate gRPC in ingresso (es. DeleteData sulla porta 50052) in modo non bloccante.
3. **Monitor Thread:** Un terzo thread (bg_thread) esegue il ciclo infinito monitoraggio_ciclico. Questo componente è disaccoppiato dalle interfacce di rete e si occupa di orchestrare il download periodico dei dati da OpenSky (ogni 600 secondi) e il salvataggio su MongoDB.

4.3 Integrazione e Resilienza (OpenSky API)

L'interazione con il servizio esterno è incapsulata nella funzione fetch_opensky_data (app.py), che implementa pattern di resilienza specifici:

- **Autenticazione OAuth2:** La funzione get_opensky_token gestisce il flow *Client Credentials*, ottenendo dinamicamente un Bearer Token per l'accesso alle API storiche.
- **Fallback Strategy:** In caso di eccezioni di rete o codici di errore HTTP (es. 403 Forbidden se il token scade o l'account è limitato), il codice non interrompe il servizio. Viene invece generato un oggetto mock_flight sintatticamente valido. Questo garantisce che il thread di monitoraggio possa completare il ciclo di scrittura su MongoDB (mongo_db.salva_voli) mantenendo il sistema in uno stato consistente anche in assenza di connettività esterna.

4.4 Aggregazione Dati (MongoDB Pipeline)

L'analisi statistica (es. media voli) non viene effettuata trasferendo i dati all'applicazione, ma sfruttando il motore di calcolo di MongoDB. Nel file database_mongo.py, il metodo get_media_voli costruisce una *Aggregation Pipeline* che filtra i documenti per intervallo temporale ($match) e calcola la somma ($sum) direttamente lato database server, ottimizzando drasticamente le performance di I/O.

5 Istruzioni per Build & Deploy

Il deployment dell'intero stack applicativo è automatizzato tramite **Docker Compose** (versione schema 3.8). La configurazione definita nel file *docker-compose.yml* garantisce il provisioning deterministico di tutti i microservizi, dei database e della rete virtuale necessaria per la comunicazione interna.

5.1 Prerequisiti Software

Per eseguire il progetto è necessario disporre di:

- **Docker Engine** (v20.10+) e **Docker CLI**.
- **Docker Compose** (v2.0+).
- Connessione Internet attiva (per il pull delle immagini base python:3.9-slim, postgres:15-alpine e mongo:6.0).

5.2 Configurazione dell'Ambiente

Il sistema è progettato per essere configurabile tramite variabili d'ambiente, evitando l'hardcoding di credenziali nel codice sorgente. Prima dell'avvio, è possibile configurare il file *docker-compose.yml* (sezione environment del servizio data-collector) per abilitare l'accesso ai dati storici reali:

environment:

- OPENSKY_CLIENT_ID=tuo_username_opensky

- OPENSKY_CLIENT_SECRET=tua_password_opensky

Nota: Se queste variabili vengono lasciate vuote o omesse, il sistema attiverà automaticamente la modalità "Mock", generando dati simulati per garantire la testabilità delle API.

5.3 Procedura di Avvio (Deployment)

1. **Clonazione e Posizionamento:** Aprire il terminale nella directory radice del progetto (dove risiede il file *docker-compose.yml*).
2. **Build e Orchestrazione:** Eseguire il comando per costruire le immagini e avviare i container in background:

docker-compose up --build -d

Dettagli tecnici dell'operazione:

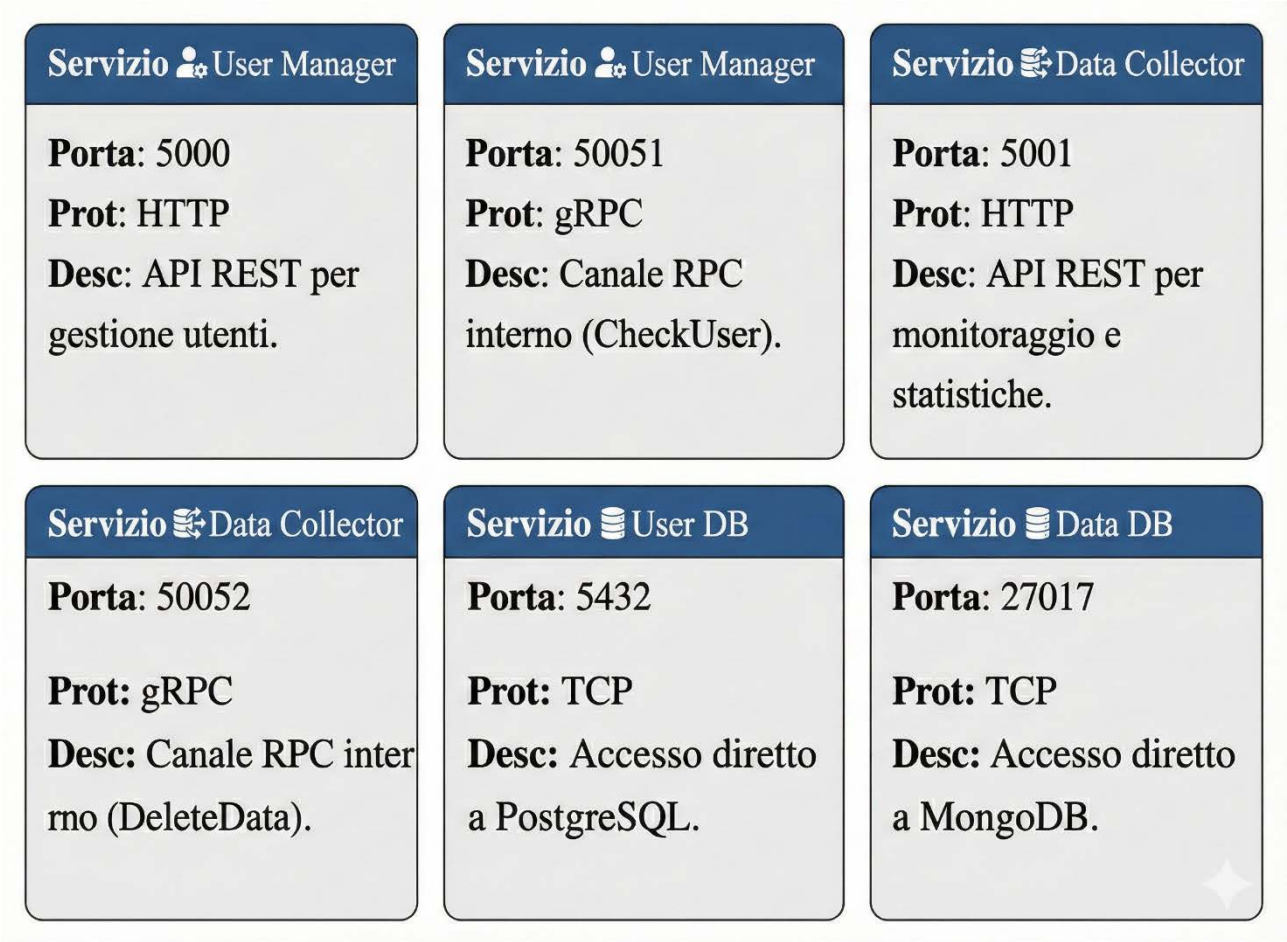
- Vengono costruite le immagini custom per *user-manager* e *data-collector* utilizzando i rispettivi Dockerfile basati su python:3.9-slim.
- Viene creata una rete bridge dedicata (app-network) per isolare il traffico dei container.
- Vengono montati i volumi persistenti (*postgres_data* e *mongo_data*) per garantire che i dati sopravvivano al riavvio dei container.

Verifica dello Stato: Controllare che i quattro servizi siano attivi (Up):

docker-compose ps

5.4 Accesso ai Servizi

Una volta avviato lo stack, i servizi espongono le seguenti interfacce sull'host locale (localhost):



5.5 Arresto e Pulizia (Teardown)

Per arrestare il sistema preservando i dati nei volumi:

docker-compose stop

Per arrestare il sistema e **rimuovere definitivamente** i volumi (utile per resettare i database e ripartire da un ambiente pulito per i test):

docker-compose down -v

6 Documentazione API (Reference)

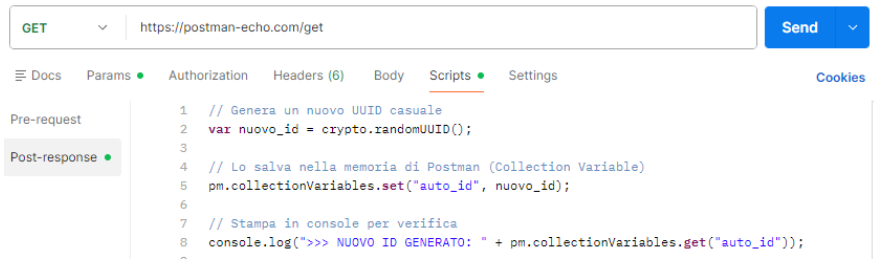
Di seguito vengono dettagliate le specifiche delle interfacce REST esposte dai microservizi. Ogni endpoint è descritto in termini di metodo HTTP, URI, parametri di richiesta e formato della risposta, riflettendo l'implementazione del codice sorgente.

6.1 User Manager Service

Il servizio gestisce l'anagrafica utenti interagendo con il database PostgreSQL.

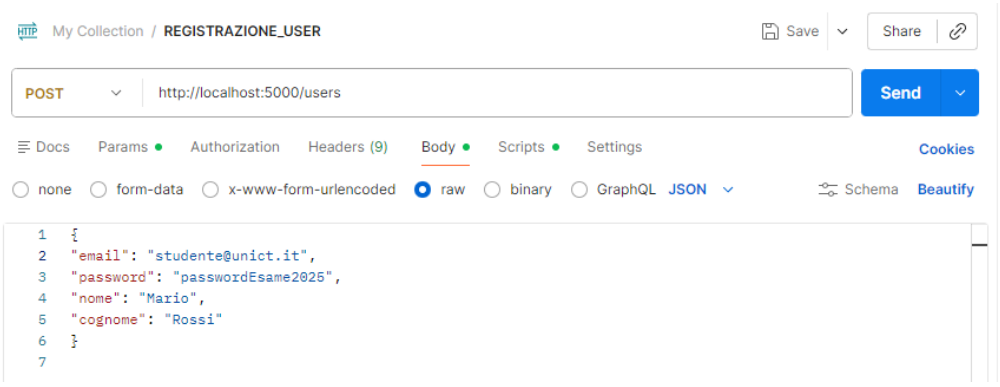
Base URL: http://localhost:5000

Request ID: generato randomicamente mediante uno script. Il seguente parametro è contenuto negli header della richiesta.



6.1.1 Registrazione Utente

Crea una nuova identità nel sistema. Implementa la politica "At-Most-Once".

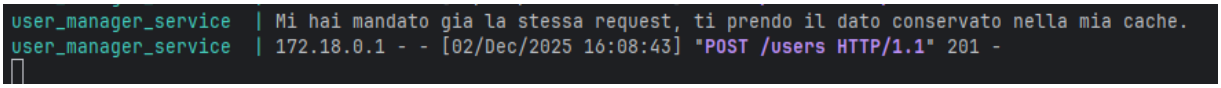


- **Risposte:** per vedere i log delle risposte direttamente da terminale utilizzare il comando:

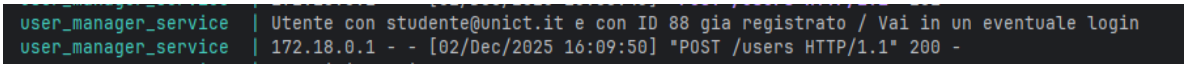
  docker-compose logs -f user-manager

- 201 Created: Utente registrato con successo.

Per il meccanismo dell' At Most Once, se l'utente rinvia richiesta 2 o più volte con lo stesso Request ID, prende il dato dalla Cache.



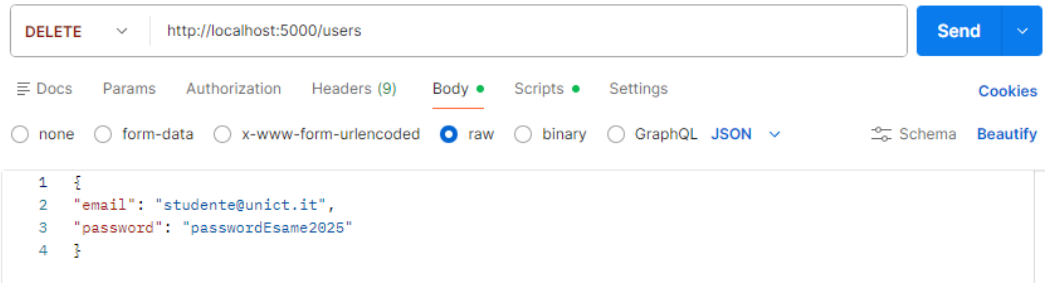
Se invece cambia la Request ID, ma le credenziali dell'utente rimangono invariate, viene visualizzato il seguente messaggio.



- 400 Bad Request: Dati mancanti o malformati.

6.1.2 Cancellazione Utente

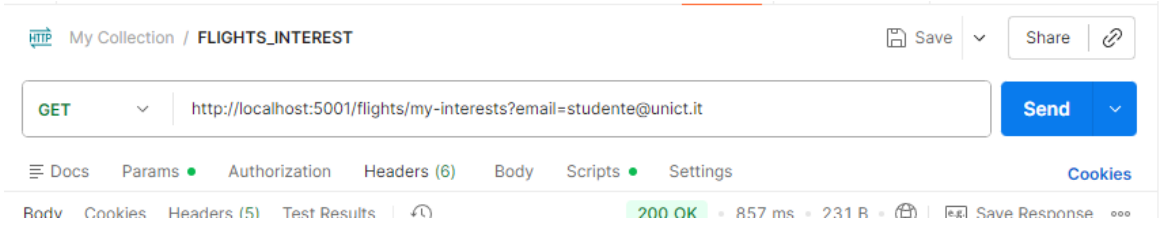
Rimuove un utente dal database PostgreSQL.



- **Risposte:** 200 OK: Cancellazione effettuata e propagata.

- 404 Not Found: Utente non presente.

Nota Architetturale: Questa operazione innesca una chiamata gRPC verso il Data Collector per garantire la pulizia a cascata dei dati correlati (interessi).



```
data_collector_service  | Recupero interessi per studente@unict.it:
data_collector_service  | {
data_collector_service  |     "user": "studente@unict.it",
data_collector_service  |     "total_flights_found": 0,
data_collector_service  |     "flights": []
data_collector_service  | }
data_collector_service  | ----------------------------
data_collector_service  | 172.18.0.1 - - [02/Dec/2025 16:51:48] "GET /flights/my-interests?email=studente@unict.it HTTP/1.1" 200 -
```

6.2 Data Collector Service

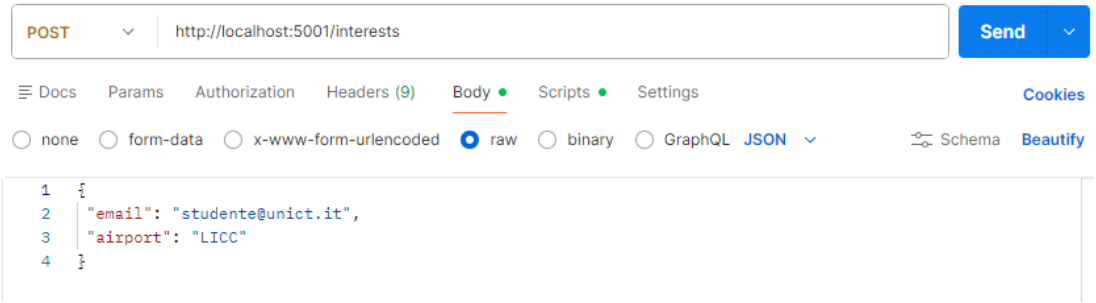
Base URL: http://localhost:5001

Il servizio espone le funzionalità di monitoraggio e analisi, interagendo con MongoDB e OpenSky Network.

6.2.1 Aggiunta Interesse (Monitoraggio)

Sottoscrive un utente al monitoraggio di uno specifico aeroporto.

Logica: Prima di salvare, verifica l'esistenza dell'utente tramite chiamata gRPC sincrona (CheckUser) allo User Manager. Se l'utente è valido, scarica immediatamente i dati preliminari da OpenSky.



- **Risposte:** per vedere i log delle risposte direttamente da terminale utilizzare il comando:

  docker-compose logs -f data-collector

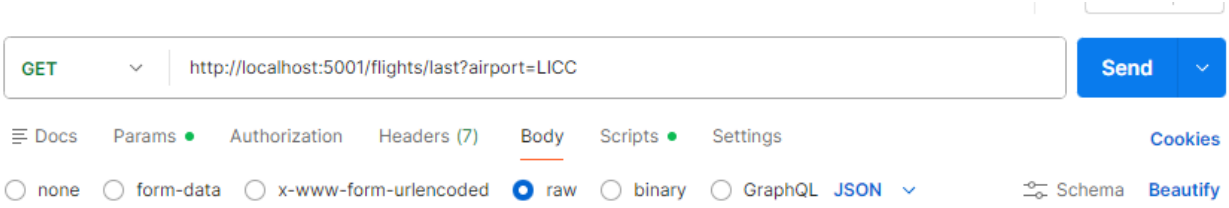
  - 200 OK: Interesse aggiunto e primo download completato.

```
data_collector_service  | Download immediato dati per LICC...
data_collector_service  | Token ottenuto, eseguo richiesta autenticata per LICC...
data_collector_service  | 172.18.0.1 - - [02/Dec/2025 16:15:34] "POST /interests HTTP/1.1" 200 -
```

  - 404 Not Found: Utente non registrato (Verifica gRPC fallita).
  - 503 Service Unavailable: User Manager non raggiungibile via gRPC.

6.2.2 Visualizzazione Ultimo Volo

Interroga MongoDB per restituire i dettagli del volo più recente registrato per un dato aeroporto.



Risposte:

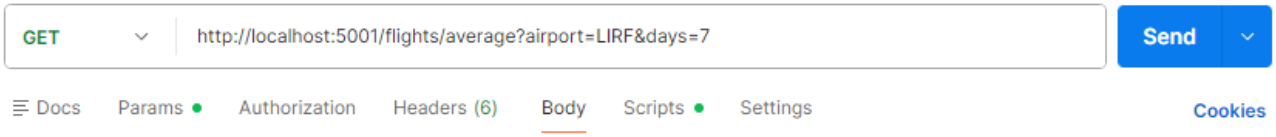
- 200 OK: Volo trovato.

```
data_collector_service  |  Ultimo volo trovato per LICC:
data_collector_service  | {
data_collector_service  |     "icao24": "3b775c",
data_collector_service  |     "firstSeen": 1764688550,
data_collector_service  |     "estDepartureAirport": "LICC",
data_collector_service  |     "lastSeen": 1764692125,
data_collector_service  |     "estArrivalAirport": null,
data_collector_service  |     "callsign": "CTM2007 ",
data_collector_service  |     "estDepartureAirportHorizDistance": 8986,
data_collector_service  |     "estDepartureAirportVertDistance": 1458,
data_collector_service  |     "estArrivalAirportHorizDistance": null,
data_collector_service  |     "estArrivalAirportVertDistance": null,
data_collector_service  |     "departureAirportCandidatesCount": 7,
data_collector_service  |     "arrivalAirportCandidatesCount": 0
data_collector_service  | }
data_collector_service  | ----------------------------
data_collector_service  | 172.18.0.1 - - [02/Dec/2025 16:17:10] "GET /flights/last?airport=LICC HTTP/1.1" 200 -
```

- 404 Not Found: Nessun dato disponibile per l'aeroporto richiesto.

6.2.3 Calcolo Media Voli

Esegue una *Aggregation Pipeline* su MongoDB per calcolare la media giornaliera dei voli rilevati in un intervallo temporale.

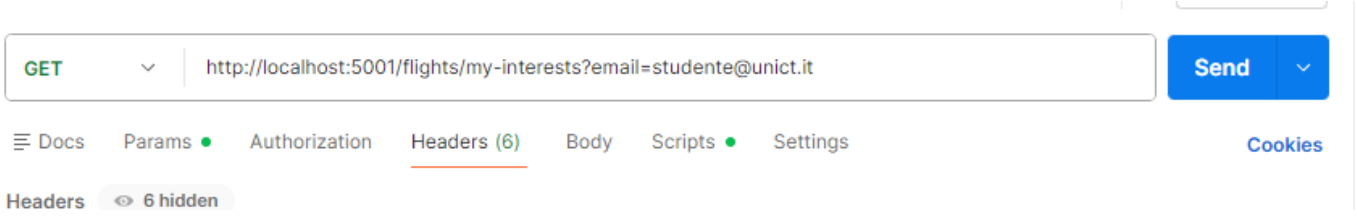


Risposte:

```
data_collector_service  | Calcolo Media:
data_collector_service  | {
data_collector_service  |     "airport": "LIRF",
data_collector_service  |     "days": 7,
data_collector_service  | 172.18.0.1 - - [02/Dec/2025 16:19:22] "GET /flights/average?airport=LIRF&days=7 HTTP/1.1" 200 -
data_collector_service  |     "average_flights": 5.0
data_collector_service  | }
data_collector_service  | ----------------------------
```

6.2.4 Visualizzazione Voli Utente

Endpoint aggregatore che restituisce tutti i voli tracciati per tutti gli aeroporti seguiti da uno specifico utente. Esegue una join applicativa tra la collezione interests e flights.



- **Risposte:**
  - 200 OK: Lista completa dei voli.

  - 400 Bad Request: Parametro e-mail mancante.

CONCLUSIONI

Il progetto ha portato alla realizzazione di un sistema distribuito completo che soddisfa pienamente i requisiti funzionali e non funzionali proposti, dimostrando l'efficacia dell'architettura a microservizi in scenari reali di integrazione dati.

7.1 Sintesi dei Risultati Tecnici

- **Persistenza Poliglotta:** L'uso combinato di PostgreSQL e MongoDB ha ottimizzato rispettivamente la consistenza transazionale e la velocità di ingestione dati.
- **Comunicazione Resiliente:** L'implementazione della **Service Config gRPC** con *Exponential Backoff* ha reso il sistema più robusto. I microservizi sono in grado di tollerare guasti temporanei della rete o riavvi dei container senza generare errori all'utente o disallineamenti nei dati .
- **Idempotenza:** La gestione tramite Request-ID e cache applicativa garantisce la correttezza delle operazioni anche in caso di retry multipli dal client.
- **Robustezza Esterna:** Il sistema di fallback sui dati Mock per OpenSky garantisce continuità operativa anche in assenza di connettività esterna.

7.2 Sviluppi Futuri

Sebbene il sistema sia pienamente operativo in ambiente Docker Compose, possibili evoluzioni future potrebbero includere:

1. La migrazione dell'orchestrazione su **Kubernetes** per gestire la scalabilità orizzontale dei microservizi *stateless*.
2. L'estenalizzazione della cache di idempotenza (attualmente in-memory nel modulo cache.py) su un datastore distribuito come **Redis**, per permettere la persistenza dello stato anche in caso di riavvio dei container.

   In conclusione, il progetto rappresenta un esempio concreto di ingegneria del software applicata ai sistemi distribuiti, coniugando moderne tecnologie di containerizzazione con pattern architetturali solidi per garantire modularità, scalabilità e affidabilità.