
UNIVERSITÀ
degli STUDI
di CATANIA

DIPARTIMENTO DI INGEGNERIA
ELETTRICA ELETTRONICA E INFORMATICA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

---

# Homework 1

---

Studenti:
Ficarra Filippo 1000034832
Guerrera Mario Anthony 1000035663

---

Anno Accademico 2025/2026

Sommario

1. Descrizione dell’applicazione

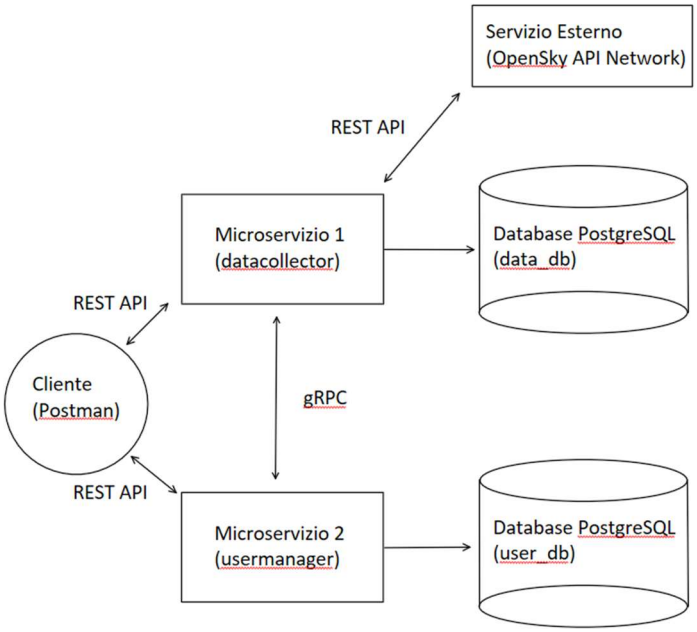
L'applicazione sviluppata è un sistema distribuito che utilizza un'architettura a microservizi per gestire e monitorare le informazioni di volo in base agli interessi degli utenti. L'obiettivo principale è fornire un servizio che raccolga dati in tempo reale da un'API esterna (OpenSky Network) e li metta a disposizione degli utenti iscritti, offrendo al contempo funzionalità di gestione utenti ed elaborazioni sui dati relativi ai voli.

Il sistema è interamente containerizzato tramite Docker e orchestrato da Docker Compose, garantendo isolamento, scalabilità e facilità di deployment. Ogni microservizio è sviluppato in Python utilizzando il framework Flask per i moduli DataCollector e UserManager, entrambi che si interfacciano con il proprio database PostgreSQL dedicato.

Le funzionalità chiave dell'applicazione includono:

- Gestione Utenti: Registrazione, visualizzazione ed eliminazione degli utenti.
- Gestione Interessi: Inserimento e visualizzazione di codici ICAO degli aeroporti di interesse per utenti registrati.
- Raccolta Dati: Schedulazione periodica per interrogare l'API OpenSkyNetwork e aggiornare il database dei voli.
- Coerenza dei Dati: Meccanismo per rimuovere automaticamente gli interessi degli utenti inattivi/eliminati.
- Statistiche: Funzionalità API per monitoraggio ed elaborazione di dati sui voli, ovvero recupero dell’ultimo volo in partenza e in arrivo da un dato aeroporto, e calcolo della media degli ultimi X giorni. Sono state implementate due funzionalità aggiuntive, che consentono di recuperare, rispettivamente, tutti i voli che si sono svolti durante un intervallo temporale, e il volo più lungo (e più breve) per un determinato aeroporto.

2. Schema architetturale

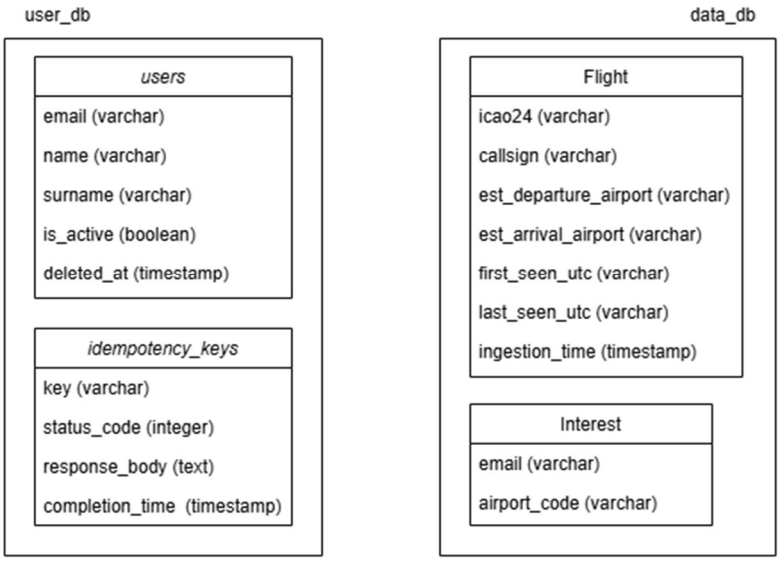



L'architettura consiste in un sistema a microservizi disaccoppiato, con chiara separazione delle responsabilità (Separation of Concerns).

Lo schema architetturale è composto da:

1. Microservizi: Sono presenti due componenti applicativi distinti: User Manager e Data Collector.

2. Database dedicati: Ogni microservizio possiede e gestisce in modo esclusivo il proprio database PostgreSQL (User DB e Data DB).




3. Clienti (Postman/Utente Finale): I client interagiscono con i microservizi quasi esclusivamente tramite REST API. In questo caso, le API sono esposte sulle porte 5000 e 5001 del Docker Compose.

4. Servizio Esterno: Il Data Collector si interfaccia con un External Service (OpenSky Network API) per la raccolta dei dati di volo tramite API REST.

## 3. (Micro)Servizi e Relative Comunicazioni

Il sistema si basa su due microservizi principali che utilizzano protocolli diversi per le loro interazioni.

- User Manager: Si occupa della gestione anagrafica dell' utente. È connesso al database PostgreSQL "user_db", che possiede le tabelle "users" e "idempotency_keys". Utilizzando il protocollo PostgreSQL grazie alla rete docker effettua le operazioni CRUD (Create, Read, Update, Delete) sul database (es. /add_user, /delete_user). Inoltre, viene implementata la politica at-most-once sul metodo /add_user per prevenire duplicazioni. Infine, espone il servizio "UserManagerService" sulla porta 50051 utilizzato dal Data Collector.

**Funzioni**

- CheckUserStatus(self, request, context): È l'implementazione del servizio gRPC chiamato dal Data Collector. Riceve un'e-mail e interroga lo user_db per verificare lo stato dell'utente. Restituisce uno stato specifico: ACTIVE se l'utente esiste ed è attivo, DELETED se esiste ma è stato eliminato logicamente (soft-delete), o NOT_FOUND se non esiste nel database.
- Serve(): Inizializza e avvia il server gRPC sulla porta configurata (GRPC_PORT). Mantiene il server in esecuzione in attesa di richieste.

- Data Collector: Si occupa di raccolta dati da OpenSkyNetwork, memorizzazione dei voli, gestione degli interessi degli utenti e fornitura di statistiche (elencate nel primo paragrafo della documentazione). È connesso al database "data_db", che comprende le tabelle "flights" e "interests". Espone API REST per aggiungere interessi (/add_interest) e per richiedere dati e statistiche sui voli (/flights, /average_flights, ecc.). Inoltre, interroga periodicamente, il database di OpenSky Network, tramite API REST, per recuperare i dati dei voli.

  **Funzioni**

  - CheckUserStatus(e-mail): Contatta in modo sincrono lo User Manager tramite gRPC per ottenere lo stato di un utente (es. ACTIVE, DELETED).
  - get_access_token():Esegue una richiesta POST all'endpoint di autenticazione per ottenere un Token Bearer OAuth2 (Access Token) necessario per interrogare le API di OpenSky Network.
  - verify_users(): Interroga lo UserManager per eliminare gli utenti non più attivi dalla tabella "Interests". Per ogni e-mail, chiama CheckUserStatus() sullo User Manager. Se un utente non è più attivo, elimina i suoi interessi dalla tabella "Interests" per mantenere la coerenza tra i servizi.
  - data_collection_job(): È il core del microservizio.
    1. Esegue prima la verify_users().
    2. Recupera i codici aeroportuali di interesse.
    3. Per ciascuno, interroga le API di OpenSkyNetwork per ottenere i dati di voli in partenza e in arrivo nelle ultime 24 ore e li salva/aggiorna nel "data_db".

    Il meccanismo di raccolta dati tramite data_collection_job() non è triggerato da un endpoint REST, ma da un thread di schedulazione interno (run_scheduler). Questo job asincrono è essenziale per disaccoppiare la logica di raccolta (che può richiedere tempo) dalle richieste API dei client, migliorando la reattività del servizio.
  - run_scheduler(): Avvia un thread per eseguire la funzione data_collection_job() a intervalli regolari (definiti dalla variabile d'ambiente PERIODO).

La comunicazione tra i due microservizi è cruciale per la coerenza del sistema e avviene tramite gRPC (sulla porta 50051). Infatti, quando un utente viene eliminato tramite API REST sulla route “/delete_user” nello User Manager, viene effettuata una soft-delete, cioè un cambio di stato dell’utente da ACTIVE a DELETED. In questo modo il service Data Collector può periodicamente verificare (tramite la funzione verify_users() )quali utenti hanno richiesto la rimozione dell’account ed eliminare gli interessi conservati nel suo “data-db”. Lo User Manager periodicamente effettuerà la pulizia del suo “user_db” dagli utenti in stato DELETED per liberare le risorse.

La politica At-Most-Once è realizzata nell'endpoint /add_user dello User Manager utilizzando il pattern di Idempotenza. Quando arriva una richiesta, viene generata una chiave (un hash dell’e-mail) e viene cercata nella tabella “idempotency_keys”. Se la chiave esiste, il sistema blocca la riesecuzione della richiesta API allo User Manager e restituisce la risposta in cache, garantendo che l'operazione di creazione utente avvenga al massimo una singola volta, anche in caso di invii multipli accidentali della stessa richiesta. La tabella “idempotency_keys” viene periodicamente svuotata delle tuple scadute da almeno 5 minuti dall’istante di prima richiesta.

## 4. Lista delle Api implementate

| **Nome API** | **Obiettivo** | **Parametri forniti** | **Parametri ritornati** |
|---|---|---|---|
| Add_user | Registra un nuovo utente | email(string)<br>name(string)<br>surname(string) | message(json) |
| Delete_user | Elimina un utente registrato | email(string) | message(json) |
| Add_interest | Aggiunge uno o più aeroporti di interesse per un utente registrato | email(string)<br>airport(List) | message(json) |
| Flights | Recupera i voli associati agli aeroporti di interesse di un utente registrato | email(string) | Per ogni volo:<br>icao_24(string)<br>callsign(string)<br>est_departure_airport(string)<br>est_arrival_airport (string)<br>first_seen_utc (string)<br>last_seen_utc (string)<br>ingestion_time(timestamp) |
| Interest | Recupera gli aeroporti di interesse degli utenti registrati | email(string) | email (string)<br>airport_code (string) (per ogni aereoporto d'interesse dell'utente) |
| Last_flight | Recupera l'ultimo volo in partenza e in arrivo da un dato aeroporto | airport_code(string) | icao24_departure (string)<br>callsign_departure (string)<br>est_departure_airport_departure (string)<br>est_arrival_airport_departure (string)<br>first_seen_utc_departure (string)<br>last_seen_utc_departure (string)<br>icao24_arrival (string)<br>callsign_arrival (string)<br>est_departure_airport_arrival (string)<br>est_arrival_airport_arrival (string) |

| | | | |
|---|---|---|---|
| | | | first_seen_utc_arrival (string)<br>last_seen_utc_arrival (string) |
| Average_flights | Recupera la media degli ultimi X giorni sul numero di voli in partenza e/o in arrivo da un dato aeroporto | airport_code(string)<br>days (int) | aeroporto_icao (string)<br>periodo_giorni (int)<br>voli_totali (int)<br>media_giornaliera (double) |
| Flights_by_period | Recupera i voli svolti in un determinato intervallo temporale | start_date(string)<br>end_date(string) | start_date_str (string)<br>end_date_str (string)<br>aeroporto (string)<br>voli_totali (int) |
| Flight_duration | Recupera il volo più lungo e più breve per un dato aeroporto | airport_code(string) | Per il volo più breve e il volo più lungo:<br>icao24 (string)<br>callsign (string)<br>durata_secondi (int)<br>durata_formattata (strinig) |