DIPARTIMENTO DI INGEGNERIA ELETTRICA, ELETTRONICA E INFORMATICA

CORSO DI LAUREA MAGISTRALE IN: INGEGNERIA INFORMATICA

DISTRIBUTED SYSTEMS AND BIG DATA

Homework 1

Documentazione

STUDENTI:
STEFANO CARAMAGNO
FEDERICO CALABRESE

DOCENTI:
PROF.SSA ANTONELLA DI STEFANO
PROF. GIOVANNI MORANA

ANNO ACCADEMICO 2025-2026

SOMMARIO

1. INTRODUZIONE

1.1. CONTESTO DEL PROGETTO E OBIETTIVI DIDATTICI

Il sistema sviluppato si inserisce nel contesto delle applicazioni *distributed-by-design* per il monitoraggio di eventi in tempo reale e la gestione di dati eterogenei provenienti da servizi esterni. In particolare, l'applicazione affronta il dominio del **monitoraggio del traffico aereo** sfruttando un provider di dati terzo (OpenSky Network) e una piattaforma applicativa basata su microservizi, containerizzazione e database relazionali.

L'obiettivo è progettare e realizzare un'applicazione che coniughi:

- **architettura a microservizi** con confini di responsabilità ben definiti;
- **integrazione con un servizio esterno** esposto tramite API REST autenticate tramite OAuth2;
- **persistenza strutturata** dei dati in più database separati per dominio applicativo;
- **meccanismi di raccolta periodica** e interrogazione dei dati di volo;
- **politiche di consistenza e idempotenza** sulle operazioni critiche.

Gli *obiettivi didattici* sono da intendersi in senso ampio come obiettivi di progettazione e sperimentazione:

- modellare un sistema distribuito che utilizza **più microservizi cooperanti**;
- distinguere in modo netto i **domini applicativi** (gestione utenti vs dati di volo);
- applicare **pattern di comunicazione differenti** (REST, gRPC, chiamate schedulate verso servizi esterni);
- gestire in modo consapevole **vincoli di consistenza, idempotenza e affidabilità** in presenza di chiamate ripetute, errori di rete e possibili indisponibilità del servizio esterno;
- predisporre un'infrastruttura **portabile e riproducibile** mediante Docker e Docker Compose.

Il risultato è un sistema che, pur rimanendo focalizzato su un caso d'uso specifico — il monitoraggio dei voli relativi ad aeroporti di interesse — costituisce un esempio completo di architettura distribuita moderna, applicabile a numerosi scenari di integrazione con sorgenti dati esterne.

1.2. DESCRIZIONE SINTETICA DELL'APPLICAZIONE DI FLIGHT MONITORING

L'applicazione implementa un sistema di **Flight Monitoring** che consente di:

- registrare utenti e gestirne il ciclo di vita;
- associare a ciascun utente una serie di **aeroporti di interesse**;
- raccogliere periodicamente, per tali aeroporti, i dati relativi ai voli in arrivo e in partenza tramite le API di **OpenSky Network**;

- memorizzare in un database dedicato le informazioni sui voli;
- esporre API di interrogazione per ottenere:
    - l'ultimo volo osservato per un determinato aeroporto e una specifica direzione (arrivi/partenze);
    - la media dei voli in un intervallo temporale espresso in giorni;
    - l'elenco dei voli registrati in un intervallo temporale arbitrario.

L'applicazione è articolata in due microservizi principali:

- **User Manager Service**
  Responsabile della gestione degli utenti. Offre API REST per la creazione, consultazione e cancellazione degli utenti, e mette a disposizione un servizio gRPC utilizzato dagli altri microservizi per verificare l'esistenza di un utente a partire dall'indirizzo e-mail.
- **Data Collector Service**
  Responsabile della gestione degli interessi utente verso gli aeroporti e della raccolta dei dati di volo. Espone API REST per:
    - la registrazione e gestione degli interessi utente–aeroporto;
    - l'interrogazione dei dati di volo.

  Integra uno **scheduler** che, a intervalli regolari, interroga OpenSky, elabora le risposte e aggiorna il database dei voli.

L'architettura separa inoltre due database logici:

- **User DB**, dedicato esclusivamente alla persistenza degli utenti;
- **Data DB**, dedicato ad aeroporti, interessi e registrazioni di volo.

Il collegamento tra i due domini avviene esclusivamente a livello applicativo: il Data Collector non accede direttamente al database utenti, ma si affida al servizio gRPC esposto dal User Manager per verificare la validità degli indirizzi e-mail.

1.3. REQUISITI FUNZIONALI PRINCIPALI

I principali requisiti funzionali dell'applicazione di Flight Monitoring possono essere sintetizzati come segue.

Gestione degli utenti

- Permettere la **registrazione di un nuovo utente** a partire da indirizzo e-mail e nome, garantendo l'unicità dell'e-mail.
- Permettere la **consultazione di un utente** a partire dall'indirizzo e-mail.
- Permettere la **cancellazione di un utente** identificato dall'e-mail.

Gestione degli aeroporti e degli interessi

- Mantenere un **catalogo di aeroporti** identificati da un codice univoco (es. ICAO), con informazioni descrittive quali nome, città, paese e fuso orario.
- Consentire a un utente di **registrare un interesse** verso uno o più aeroporti del catalogo.
- Garantire che per ogni coppia *(utente, aeroporto)* possa esistere **al più un interesse**, evitando duplicati e preservando la semantica di idempotenza.
- Permettere la **rimozione di un interesse** utente–aeroporto.
- Permettere la **consultazione degli interessi associati a un utente**.

Raccolta dei dati di volo

- Interrogare periodicamente il servizio esterno **OpenSky Network** per ciascun aeroporto per cui esiste almeno un interesse utente.
- Recuperare i **voli in arrivo** e i **voli in partenza** in un intervallo temporale definito (es. una finestra di tempo recente, configurabile).
- Effettuare il **mapping dei dati grezzi** restituiti da OpenSky in entità di dominio interne (*flight records*), associandole all'aeroporto corrispondente.
- Memorizzare i dati di volo nel **Data DB**, includendo informazioni come identificativo esterno, numero del volo, direzione (ARRIVAL/DEPARTURE), orari, eventuali ritardi e timestamp di raccolta.

Interrogazione dei dati di volo

- Fornire un'API per ottenere **l'ultimo volo registrato** per un aeroporto e una direzione specifici, restituendo un errore significativo se non sono presenti voli.
- Fornire un'API per calcolare la **media dei voli** per un aeroporto e una direzione su una finestra temporale espressa in giorni, restituendo il numero totale di voli e il valore medio giornaliero.
- Fornire un'API per ottenere l'**elenco dei voli** registrati per un aeroporto e una direzione in un intervallo temporale [*from*, *to*] specificato o, in assenza di parametri, su un intervallo di default.
- Gestire parametri non validi (ad esempio intervalli temporali invertiti, valori di giorni non positivi, direzioni non ammesse) con messaggi di errore chiari e codici HTTP adeguati.

Interazione tra microservizi

- Consentire al Data Collector di **validare l'esistenza di un utente** prima di registrare un interesse, utilizzando un servizio gRPC fornito dal User Manager.
- Garantire che il Data Collector non persista interessi per utenti inesistenti, restituendo in tal caso un errore di dominio appropriato.

1.4. REQUISITI NON FUNZIONALI (SCALABILITA', MANUTENIBILITA', SEPARAZIONE DEI DOMINI, ECC.)

La progettazione del sistema è guidata da una serie di requisiti non funzionali che ne orientano la struttura e le scelte tecnologiche.

Separazione dei domini e indipendenza dei microservizi

- Il dominio *utente* e il dominio *dati di volo* sono **separati sia a livello logico che a livello di persistenza**:
    - User Manager opera su **User DB**;
    - Data Collector opera su **Data DB**.
- I microservizi comunicano esclusivamente tramite **API pubbliche** (REST e gRPC), riducendo il *coupling* e favorendo l'evoluzione indipendente dei servizi.

Scalabilità e distribuzione

- La suddivisione in microservizi consente, in prospettiva, di **scalare indipendentemente**:
    - il servizio di raccolta dati (Data Collector) in funzione del volume di richieste verso OpenSky e del numero di aeroporti monitorati;
    - il servizio di gestione utenti (User Manager) in funzione del numero di operazioni di registrazione e validazione.
- L'uso di **Docker e Docker Compose** permette di distribuire l'applicazione come insieme di container, facilitando il deploy su ambienti diversi (macchine di sviluppo, server dedicati, cluster container-based).

Manutenibilità e chiarezza del codice

- Ogni microservizio adotta una **struttura a layer chiari**:
    - controller REST;
    - servizi applicativi;
    - repository JPA;
    - componenti dedicati per l'integrazione esterna (client OpenSky, client gRPC).
- L'uso di **JPA** e **Flyway** standardizza l'accesso al database e la gestione delle migrazioni, semplificando l'evoluzione dello schema dati.
- La gestione centralizzata delle eccezioni tramite **RestExceptionHandler** contribuisce a mantenere coerenza e ridurre duplicazioni nella gestione degli errori.

Affidabilità, idempotenza e consistenza

- Le operazioni di **registrazione utente** e **registrazione interessi** sono progettate per essere *idempotenti*:

    - la registrazione utente verifica la presenza dell'e-mail prima di creare un nuovo record;
    - la registrazione di un interesse verifica l'esistenza della coppia *(utente, aeroporto)* prima di inserire un nuovo record.
- La **consistenza tra domini** è garantita applicativamente: ogni interesse utente–aeroporto viene creato solo se il servizio User Manager conferma l'esistenza dell'utente via gRPC.
- La raccolta periodica dei voli è progettata per tollerare:
    - assenze temporanee di dati (es. nessun volo nella finestra considerata);
    - specifiche risposte del servizio esterno (es. 404 per "nessun dato"), gestendo tali casi con logica applicativa esplicita e senza compromettere lo stato interno.

Robustezza e gestione delle integrazioni esterne

- Il client verso OpenSky implementa una **gestione del token OAuth2** con caching e controllo della scadenza, riducendo il numero di autenticazioni effettuate e garantendo l'uso corretto del bearer token nelle chiamate REST.
- Vengono gestiti scenari di errore tipici delle integrazioni esterne:
    - risposta 404 interpretata come assenza di voli in un intervallo temporale;
    - possibili errori di rete o indisponibilità temporanea del servizio, mitigabili con retry o politiche di log e monitoraggio (estendibili in evoluzioni future).

Portabilità e riproducibilità

- L'intero sistema è avviabile tramite **Docker Compose**, che orchestra:
    - il container PostgreSQL;
    - il container del User Manager;
    - il container del Data Collector.
- La configurazione esterna (parametri di connessione, credenziali, URL delle API esterne) è gestita tramite file di configurazione e variabili d'ambiente, favorendo l'adattamento a diversi ambienti di esecuzione senza modifiche al codice.

Questi requisiti non funzionali orientano tutte le scelte progettuali successive e costituiscono il quadro di riferimento entro cui vengono valutate architettura, modellazione dei dati, comunicazioni tra microservizi e design delle API.

2. DETTAGLIO DEI MICROSERVIZI E DELLE COMUNICAZIONI

2.1. ATTORI COINVOLTI

Il sistema di Flight Monitoring coinvolge un insieme di attori, interni ed esterni, che cooperano per fornire funzionalità di gestione utenti, definizione degli interessi sugli aeroporti e monitoraggio continuo dei voli. Gli attori principali sono:

- il **client esterno**, che interagisce con il sistema tramite API REST;
- il **microservizio User Manager**, responsabile della gestione del dominio utente;
- il **microservizio Data Collector**, responsabile della gestione degli interessi e dei dati di volo;
- il **servizio esterno OpenSky Network**, sorgente autorevole dei dati di traffico aereo.

2.1.1. CLIENT ESTERNO (POSTMAN / REST CLIENT)

Il **client esterno** rappresenta qualsiasi componente in grado di invocare le API REST esposte dal sistema. Nel contesto operativo il ruolo di client è svolto tipicamente da:

- strumenti di test e ispezione delle API, come **Postman**;
- potenziali applicazioni client future (ad esempio una web dashboard, un'app mobile o un altro servizio applicativo).

Il client interagisce con il sistema eseguendo chiamate HTTP verso i microservizi esposti, in particolare:

- verso il **User Manager**, per:
    - registrare nuovi utenti;
    - recuperare le informazioni di un utente;
    - cancellare un utente esistente;
- verso il **Data Collector**, per:
    - creare, eliminare e consultare gli interessi utente–aeroporto;
    - interrogare le statistiche e l'elenco dei voli registrati.

Dal punto di vista del sistema, il client è un attore esterno non affidabile: le richieste possono essere ripetute, malformate o incoerenti, motivo per cui i microservizi implementano **validazioni esplicite** e **politiche di idempotenza** per le operazioni sensibili.

2.1.2. MICROSERVIZIO USER MANAGER

Il **User Manager** è il microservizio incaricato della gestione del dominio *utente*. Le sue responsabilità principali sono:

- mantenere il **registro degli utenti** del sistema, con attributi quali indirizzo e-mail e nome;

- garantire l'**unicità dell'e-mail** a livello di dominio, impedendo la creazione di duplicati;
- fornire operazioni per:
    - la **registrazione** di un nuovo utente;
    - la **consultazione** di un utente esistente;
    - la **cancellazione** di un utente.

Il microservizio espone:

- una **API REST** per la gestione diretta degli utenti da parte del client esterno;
- un **servizio gRPC** che funge da **punto di verità** per la validazione degli utenti, utilizzato dal Data Collector per verificare l'esistenza di un utente a partire dall'e-mail.

Il User Manager interagisce con un proprio database dedicato, il **User DB**, nel quale persiste l'informazione relativa agli utenti. Nessun altro microservizio accede direttamente a tale database; la visibilità dei dati utente verso l'esterno è mediata esclusivamente dalle API REST e gRPC del User Manager, a tutela della **separazione dei domini** e della **coerenza dei dati**.

2.1.3. MICROSERVIZIO DATA COLLECTOR

Il **Data Collector** è il microservizio responsabile della gestione del dominio relativo agli **aeroporti**, agli **interessi utente–aeroporto** e ai **dati di volo** raccolti da OpenSky.

Le sue responsabilità principali includono:

- mantenere un **catalogo di aeroporti** monitorabili, identificati da un codice univoco e arricchiti da informazioni descrittive;
- gestire gli **interessi degli utenti** rispetto agli aeroporti, permettendo di:
    - associare a un utente uno o più aeroporti di interesse;
    - evitare duplicazioni per la stessa coppia *(utente, aeroporto)*;
    - rimuovere interessi esistenti;
- gestire la **raccolta periodica** dei dati di volo (arrivi e partenze) per gli aeroporti per cui esiste almeno un interesse;
- persistere i **flight records** in un database dedicato, il **Data DB**;
- esporre **API REST** per l'interrogazione dei dati di volo, fornendo statistiche e dettagli operativi.

Il Data Collector interagisce con:

- il **User Manager**, tramite gRPC, per **validare l'esistenza degli utenti** prima di registrare interessi;

- il **servizio esterno OpenSky Network**, tramite client HTTP con autenticazione OAuth2, per ottenere i dati relativi ai voli;
- il **Data DB**, per la persistenza di aeroporti, interessi e registrazioni di volo.

L'utilizzo di uno scheduler interno consente al Data Collector di operare in modalità **proattiva**: la raccolta dei dati non dipende dalle richieste del client, ma è guidata da intervalli temporali configurati.

2.1.4. SERVIZIO ESTERNO OPENSKY NETWORK

OpenSky Network è il servizio esterno che fornisce i dati di volo utilizzati dal sistema di Flight Monitoring. In qualità di sorgente dati esterna, OpenSky:

- espone **API REST** attraverso le quali è possibile interrogare:
  - i voli in **arrivo** su un aeroporto in un certo intervallo temporale;
  - i voli in **partenza** da un aeroporto in un certo intervallo temporale;
- richiede un meccanismo di **autenticazione OAuth2** (grant type *client credentials*), che produce un *access token* da utilizzare nelle successive richieste HTTP.

Il sistema utilizza OpenSky come **data provider autorevole**, integrandolo tramite un componente dedicato (*OpenSky Client*) che si occupa di:

- ottenere e gestire il **token di accesso**;
- invocare le specifiche API (arrivals e departures) per ciascun aeroporto e intervallo temporale;
- interpretare correttamente le risposte (inclusi casi come il **404 "nessun volo"**).

2.2. SCENARIO D'USO COMPLESSIVO

Lo scenario d'uso complessivo del sistema può essere descritto come una sequenza di fasi, nelle quali interagiscono i diversi attori introdotti in precedenza.

1. **Onboarding dell'utente**
   Un client esterno invoca le API REST del User Manager per **registrare un nuovo utente**, fornendo indirizzo e-mail e nome. Il User Manager crea il profilo utente nel proprio database e diventa il riferimento per tutte le future operazioni di validazione su quell'identità.
2. **Definizione degli aeroporti di interesse**
   Una volta registrato, l'utente può indicare, tramite il Data Collector, uno o più **aeroporti di interesse**.
   - Il client invia una richiesta di creazione dell'interesse al Data Collector (API REST dedicata).
   - Il Data Collector richiede al User Manager, via gRPC, di verificare che l'e-mail fornita corrisponda a un utente esistente.
   - Se l'utente è valido e l'aeroporto fa parte del catalogo, il Data Collector registra l'interesse nel proprio database evitando duplicati.

3. **Raccolta periodica dei dati di volo**
   Un componente scheduler all'interno del Data Collector, con frequenza configurabile, attiva il processo di **raccolta dei voli**:
   - il Data Collector ricava la lista degli aeroporti per cui esiste almeno un interesse;
   - per ciascuno di essi interroga OpenSky Network per ottenere in un intervallo temporale predefinito:
     - i voli in arrivo;
     - i voli in partenza;
   - i dati ricevuti vengono trasformati in **entità di dominio interne** e memorizzati nel Data DB.
4. **Interrogazione dei dati di volo**
   Il client può interrogare l'applicazione per ottenere **informazioni aggiornate sui voli**:
   - ultimo volo per aeroporto e direzione (arrivo/partenza);
   - media dei voli registrati su un certo numero di giorni;
   - elenco completo dei voli registrati in un intervallo temporale.
     Il Data Collector elabora le richieste, esegue le query sul Data DB, costruisce le risposte in formato JSON e le restituisce al client.

Lo scenario complessivo realizza quindi una pipeline applicativa in cui:

- l'utente configura i propri interessi;
- il sistema raccoglie autonomamente i dati dai servizi esterni;
- il client ottiene informazioni aggiornate e strutturate sul traffico aereo relativo agli aeroporti di interesse.

2.3. FLUSSI INFORMATIVI DI ALTO LIVELLO

I flussi informativi di alto livello descrivono come i dati si muovono tra gli attori e i componenti principali del sistema, distinguendo tra:

- **flussi di controllo e configurazione** (gestione utenti, definizione interessi);
- **flussi di dati** (raccolta e interrogazione dei voli).

I principali flussi possono essere riassunti come segue.

Flusso 1 – Gestione utenti (Client ↔ User Manager ↔ User DB)

- Il client invia richieste REST al User Manager per creare, leggere o cancellare utenti.
- Il User Manager accede al **User DB** per:
  - verificare l'esistenza di un utente (in particolare a partire dall'e-mail);

    - creare nuovi record;
    - aggiornare lo stato applicativo con operazioni di cancellazione.
- L'informazione di ritorno (utente creato, trovato o non trovato) viene restituita al client in formato JSON, con codici HTTP aderenti al risultato dell'operazione.

Flusso 2 – Gestione interessi (Client ↔ Data Collector ↔ User Manager ↔ Data DB)

- Il client invia richieste REST al Data Collector per registrare o rimuovere interessi *utente–aeroporto*.
- Il Data Collector:
    - interroga il **User Manager** via gRPC per verificare che l'indirizzo e-mail fornito appartenga a un utente esistente;
    - consulta il **Data DB** per verificare l'esistenza dell'aeroporto e dell'eventuale interesse già associato;
    - aggiorna il Data DB creando o eliminando i record di interesse.
- Il risultato dell'operazione (interesse creato, già esistente, rimosso o errore) viene comunicato al client tramite risposta REST.

Flusso 3 – Raccolta dei voli (Scheduler/Data Collector ↔ OpenSky ↔ Data DB)

- Lo scheduler interno al Data Collector attiva periodicamente il processo di **raccolta**.
- Il Data Collector:
    - legge dal Data DB la lista di aeroporti per cui è presente almeno un interesse;
    - per ciascun aeroporto invoca le API di **OpenSky Network** (servizio esterno) per ottenere i voli in arrivo e in partenza in un intervallo temporale definito;
    - trasforma la risposta del servizio esterno in **flight records** coerenti con il modello interno;
    - persiste i nuovi record nel **Data DB**.
- Questo flusso non coinvolge direttamente il client esterno, ma aggiorna lo stato informativo su cui si basano le successive interrogazioni.

Flusso 4 – Interrogazione dei voli (Client ↔ Data Collector ↔ Data DB)

- Il client invia richieste REST al Data Collector per ottenere:
    - il **singolo ultimo volo** per aeroporto/direzione;
    - la **media dei voli** su finestra temporale in giorni;
    - l'**insieme dei voli** in un intervallo temporale arbitrario.

- Il Data Collector:
    - valida i parametri della richiesta (direzione, intervalli temporali, numero di giorni);
    - interroga il **Data DB** con query appropriate (ricerca del massimo per actual time, conteggi su range temporali, selezione di liste ordinate);
    - costruisce la risposta JSON includendo i campi rilevanti (codice aeroporto, direzione, identificativi di volo, orari, stato, ritardi, timestamp di raccolta).
- Gli esiti vengono restituiti al client con codici HTTP coerenti (200 per successo, 400 per parametri non validi, 404 per risorse mancanti).

Questi flussi delineano una chiara separazione tra:

- **flussi di configurazione** (creazione utenti, definizione interessi), che modellano il *cosa* monitorare;
- **flussi di ingestione** (raccolta dai servizi esterni), che popolano il sistema di dati aggiornati;
- **flussi di consultazione** (interrogazione dei voli), che espongono verso l'esterno il valore informativo generato dal sistema.

3. ARCHITETTURA DEL SISTEMA

3.1. PANORAMICA ARCHITETTURALI

L'architettura del sistema di **Flight Monitoring** è basata su un approccio *microservices-oriented*, in cui le responsabilità applicative sono suddivise in servizi autonomi e cooperanti, orchestrati all'interno di un ambiente containerizzato. I due domini principali – gestione degli utenti e gestione dei dati di volo – sono modellati come **microservizi indipendenti**, ciascuno dotato di un proprio database logico e di interfacce ben definite verso l'esterno.

Il sistema è composto da:

- un **microservizio User Manager**, che si occupa del ciclo di vita degli utenti e fornisce funzionalità di validazione tramite gRPC;
- un **microservizio Data Collector**, che gestisce aeroporti, interessi utente–aeroporto e raccolta dei dati di volo da OpenSky, nonché le API di interrogazione dei voli;
- un'istanza di **PostgreSQL** che ospita due database logici distinti (User DB e Data DB), ciascuno di competenza di un microservizio;
- un **client esterno** che interagisce con i microservizi tramite API REST;
- il **servizio esterno OpenSky Network**, raggiunto tramite client HTTP autenticato con OAuth2.

Le interazioni tra gli elementi architetturali adottano protocolli standard:

- **HTTP/REST** per le comunicazioni tra client e microservizi;
- **gRPC** per la validazione remota degli utenti tra Data Collector e User Manager;
- **JDBC/JPA** per l'accesso ai database;
- **HTTP/REST + OAuth2** per le richieste verso l'API di OpenSky.

Tutti i componenti applicativi e il database sono eseguiti come container Docker, orchestrati tramite Docker Compose, al fine di garantire **portabilità**, **riproducibilità** e facilità di messa in esercizio dell'intero sistema.

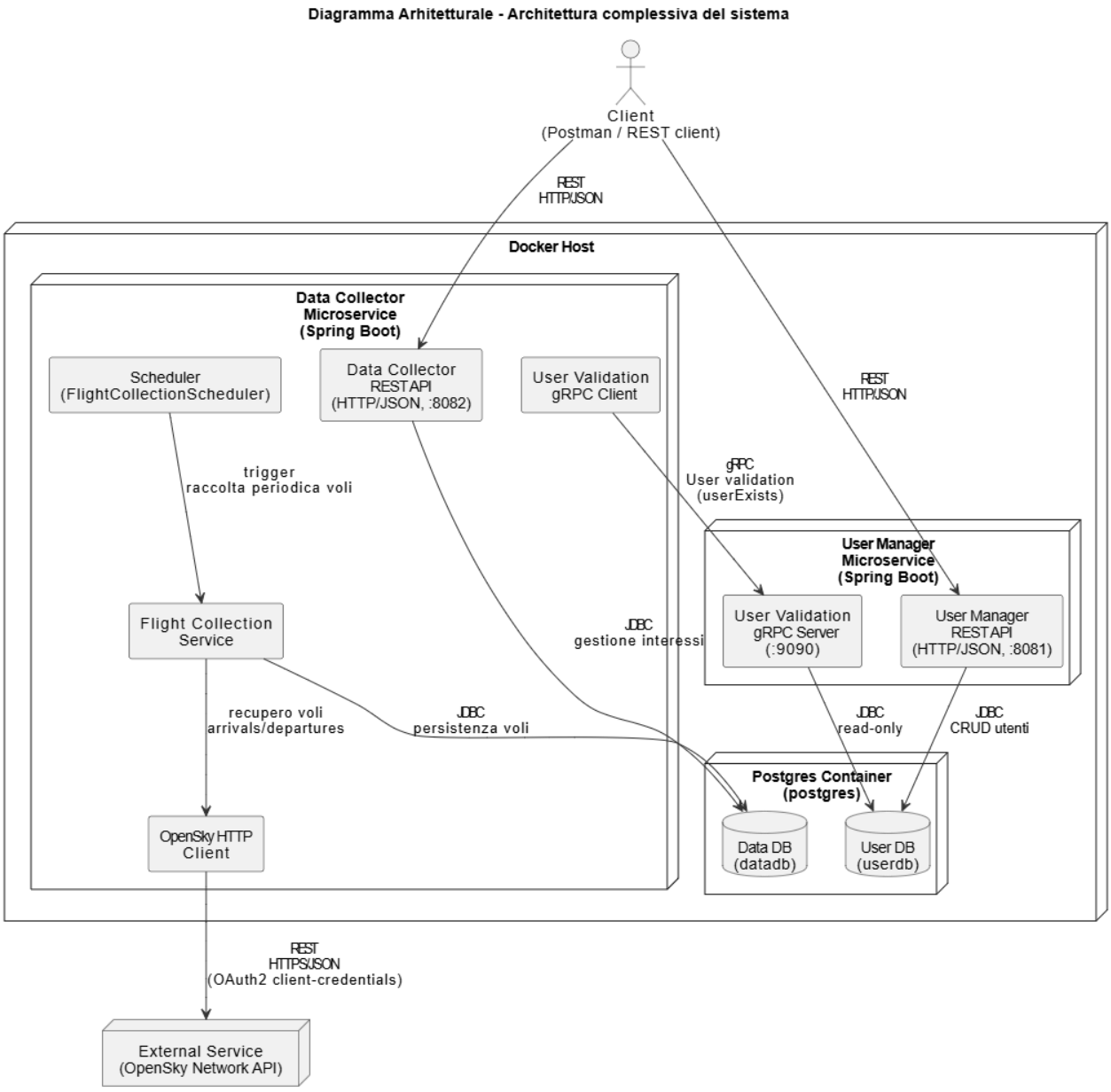
3.1.1. DIAGRAMMA ARCHITETTURALE – ARCHITETTURA COMPLESSIVA DEL SISTEMA

In questa sezione viene inserita l'immagine dell'"**diagramma architetturale – architettura complessiva del sistema**" che rappresenta l'architettura complessiva del sistema di Flight Monitoring. Il diagramma evidenzia:

- il **client esterno**, collegato tramite REST al microservizio User Manager e al microservizio Data Collector;
- il **microservizio User Manager**, con le sue interfacce REST e gRPC, associato al **User DB** all'interno dell'istanza PostgreSQL;

- il **microservizio Data Collector**, con le sue API REST, il componente scheduler interno, il servizio di raccolta dei voli e il client OpenSky, associato al **Data DB**;
- il **servizio esterno OpenSky Network**, raggiunto tramite il client HTTP autenticato;
- i protocolli di comunicazione fondamentali (REST/HTTP, gRPC, JDBC) e i confini tra domini applicativi.

Il diagramma costituisce il riferimento visivo principale per la comprensione delle relazioni tra i microservizi, il database e i sistemi esterni.



3.2. MICROSERVIZIO USER MANAGER

Il microservizio **User Manager** implementa il dominio relativo alla gestione degli utenti del sistema. Opera come *authority* per tutte le informazioni identificative degli utenti e funge da punto di verità per la loro validazione, esponendo sia un'interfaccia REST verso i client esterni sia un'interfaccia gRPC verso gli altri microservizi. Questo servizio è progettato in modo da essere **indipendente** dal resto dell'ecosistema e da poter evolvere autonomamente, purché venga preservato il contratto delle interfacce esposte.

3.2.1. RESPONSABILITA' E CONFINI DEL DOMINIO

Il dominio del User Manager è circoscritto alla gestione dell'entità **utente** e comprende le seguenti responsabilità:

- **Creazione dell'utente** a partire da e-mail e nome, con vincolo di unicità sull'indirizzo e-mail;
- **Lettura dei dati utente** a partire dalla chiave primaria (e-mail), per permettere al client di verificare o ispezionare le informazioni registrate;
- **Cancellazione dell'utente**, con eliminazione del relativo record dal User DB;
- **Validazione dell'esistenza dell'utente** come servizio offerto ad altri microservizi, in particolare al Data Collector, attraverso un'API gRPC.

I confini del dominio sono definiti in modo netto:

- il User Manager **non gestisce interessi, aeroporti o voli**;
- il suo database è limitato alla tabella degli utenti;
- qualsiasi verifica sull'esistenza di un utente deve essere delegata a questo microservizio, evitando accessi diretti al User DB da parte di altri componenti.

Questa separazione garantisce che l'integrità del dominio utente sia mantenuta sotto il controllo di un unico servizio, semplificando la gestione della coerenza e dell'evoluzione del modello dati utente.

3.2.2. COMPONENTI INTERNI (LAYER REST, SERVICE, REPOSITORY, gRPC)

Dal punto di vista interno, il User Manager adotta una struttura a **layer**:

- **Layer REST**
  Implementato da controller annotati con le convenzioni tipiche di Spring Web, si occupa di:
  - esporre le API HTTP per la registrazione, consultazione e cancellazione degli utenti;
  - validare i payload di input tramite annotazioni come `@Valid` e vincoli sui DTO;
  - tradurre le richieste HTTP in invocazioni al layer di servizio.

- **Layer Service**
  Contiene la logica di **business** del dominio utente. Le sue responsabilità includono:
  - applicare la politica di **idempotenza** sulla registrazione, verificando l'esistenza di un utente prima di crearne uno nuovo;
  - orchestrare le chiamate al repository per operazioni CRUD;
  - esporre metodi dedicati alla **validazione dell'utente** (esistenza per e-mail) utilizzati dal servizio gRPC.
- **Layer Repository (JPA)**
  Implementato tramite interfacce Spring Data JPA, fornisce l'accesso al **User DB**:
  - mapping dell'entità utente alla tabella `users`;
  - gestione di metodi come `findById`, `existsById`, `save`, delegando a JPA la generazione delle query SQL.
- **Servizio gRPC**
  Il componente gRPC implementa il servizio di **User Validation** definito tramite il relativo file `.proto`. Le sue caratteristiche principali sono:
  - ricezione di richieste di validazione contenenti l'e-mail dell'utente;
  - invocazione del layer di servizio per verificare l'esistenza dell'utente;
  - restituzione di una risposta strutturata (ad esempio flag `exists`) ad altri microservizi.

La gestione delle eccezioni HTTP (ad esempio errori di validazione) viene centralizzata in un componente di **exception handling** dedicato, che mappa le eccezioni applicative in codici di stato HTTP e payload di errore coerenti.

3.3. MICROSERVIZIO DATA COLLECTOR

Il microservizio **Data Collector** gestisce il dominio relativo agli **aeroporti**, agli **interessi utente–aeroporto** e ai **dati di volo** raccolti dal servizio esterno OpenSky. Opera come punto di accesso per tutte le funzionalità di configurazione degli interessi e per l'interrogazione dei voli, oltre a inglobare la logica di raccolta periodica dei dati.

3.3.1. RESPONSABILITA' E CONFINI DEL DOMINIO

Le principali responsabilità del Data Collector sono:

- **Gestione del catalogo aeroporti**
  Mantiene la lista degli aeroporti monitorabili, identificati da un codice (es. ICAO) e associati a metadati quali nome, città, paese e fuso orario.
- **Gestione degli interessi utente–aeroporto**
  Consente di:
  - registrare l'interesse di un utente verso un aeroporto presente nel catalogo;
  - garantire che per ogni coppia *(utente, aeroporto)* esista al più un interesse;

    - rimuovere interessi esistenti;
    - interrogare gli interessi associati a un determinato utente.
- **Raccolta periodica dei dati di volo**
  Per ciascun aeroporto per cui esiste almeno un interesse:
    - interroga OpenSky per ottenere le liste di voli in arrivo e in partenza in una finestra temporale configurata;
    - effettua il mapping delle risposte grezze in **flight records** coerenti con il modello interno;
    - persiste tali record nel **Data DB**.
- **Interrogazione dei dati di volo**
  Espone API REST per:
    - ottenere l'ultimo volo registrato per aeroporto e direzione;
    - calcolare la media dei voli su una finestra temporale espressa in giorni;
    - recuperare l'elenco di voli in un intervallo temporale specificato.

I confini del dominio del Data Collector escludono esplicitamente la gestione diretta degli utenti: la verifica dell'esistenza di un utente è delegata al **User Manager** tramite gRPC. Il Data Collector non accede al User DB e si limita a utilizzare, come identificatore utente, l'e-mail validata dal servizio esterno.

3.3.2. COMPONENTI INTERNI (REST API, SCHEDULER, SERVICE, CLIENT OpenSky)

Internamente, il Data Collector è articolato nei seguenti componenti:

- **REST API (controller)**
  Espongono gli endpoint per:
    - gestione degli interessi (`POST`, `DELETE`, `GET` su `/api/interests`);
    - interrogazione dei voli (`GET` su `/api/flights/last`, `/api/flights/average`, `/api/flights`).

  I controller:
    - effettuano la **validazione dei parametri di input**;
    - delegano la logica applicativa ai servizi interni;
    - gestiscono codici di ritorno HTTP coerenti con gli esiti delle operazioni.
- **Servizi applicativi (service layer)**
  Gestiscono la logica di business relativa a:
    - creazione e cancellazione degli interessi, con politiche di idempotenza;
    - orchestrazione della raccolta dei voli (per singolo aeroporto o per tutti gli aeroporti interessati);
    - esecuzione delle query sul Data DB per il recupero dei flight records;

    - calcolo delle statistiche (ad esempio media giornaliera dei voli).

- **Scheduler**
  Implementato tramite annotazioni come `@Scheduled`, si occupa di:
    - pianificare l'esecuzione periodica della procedura di raccolta dei voli;
    - calcolare la finestra temporale di interesse (ad esempio le ultime N ore);
    - invocare i metodi del servizio dedicato alla raccolta per avviare il flusso di ingestione verso OpenSky.
- **Client OpenSky**
  È il componente di integrazione con il servizio esterno OpenSky Network. Le sue responsabilità includono:
    - gestione dell'**autenticazione OAuth2** con grant type *client credentials*;
    - caching del token di accesso e verifica della sua validità prima di effettuare chiamate agli endpoint di arrivo e partenza;
    - invio delle richieste HTTP verso gli endpoint di OpenSky, con gestione degli header di autorizzazione;
    - interpretazione delle risposte, inclusi casi come:
        - risposta vuota o priva di voli;
        - errori specifici (es. HTTP 404 come "nessun volo disponibile nell'intervallo").
- **Repository JPA**
  Forniscono l'accesso strutturato al **Data DB** per le entità:
    - *Airport*;
    - *UserAirportInterest*;
    - *FlightRecord*.

  I repository espongono metodi dedicati per:
    - selezionare gli aeroporti di interesse attuale;
    - verificare l'esistenza di un interesse (per idempotenza);
    - contare e selezionare i voli in base a criteri temporali e di direzione.

La gestione centralizzata delle eccezioni applicative è affidata a un componente di **exception handling** che traduce errori di dominio (utente inesistente, aeroporto non trovato, volo non presente, parametri non validi) in risposte HTTP con payload di errore omogenei.

3.4. COMPONENTI INFRASTRUTTURALI

L'architettura applicativa è supportata da un insieme di componenti infrastrutturali che garantiscono persistenza, orchestrazione e integrazione con servizi esterni. Questi

componenti includono il database relazionale, l'ambiente di containerizzazione e l'integrazione con il provider di dati esterno.

3.4.1. DATABASE PostgreSQL (User DB E Data DB)

Il sistema utilizza un'istanza di **PostgreSQL** come motore di database relazionale. All'interno di questa istanza sono definiti due database logici separati:

- **User DB**
  Contiene la tabella degli utenti, indicizzata sull'indirizzo e-mail, che funge da chiave primaria. È utilizzato esclusivamente dal microservizio User Manager. La gestione dello schema è automatizzata tramite **Flyway**, che applica le migrazioni SQL all'avvio del servizio.
- **Data DB**
  Contiene le tabelle relative al dominio dei voli:
  - tabella *airports*, con il catalogo degli aeroporti monitorati;
  - tabella *user_airport_interest*, che rappresenta l'associazione tra utenti (referenziati tramite e-mail) e aeroporti;
  - tabella *flight_records*, che memorizza le informazioni sui voli raccolti da OpenSky.
    Anche in questo caso, **Flyway** gestisce la creazione e l'evoluzione dello schema.

La scelta di utilizzare un'unica istanza PostgreSQL con database logici distinti consente di ottenere una **separazione dei domini a livello logico**, mantenendo al contempo una gestione semplificata dell'infrastruttura (un singolo container per il database).

3.4.2. AMBIENTE DOCKER E DOCKER COMPOSE

Tutti i componenti del sistema sono eseguiti all'interno di container **Docker**, orchestrati tramite **Docker Compose**. La definizione del file `docker-compose.yml` include:

- il container **Postgres**, con:
  - configurazione di database, utente e password;
  - volume per la persistenza dei dati;
- il container **User Manager**, con:
  - immagine derivata dalla build del progetto Spring Boot;
  - variabili d'ambiente per la configurazione della connessione al User DB;
  - esposizione delle porte HTTP (per REST) e gRPC;
- il container **Data Collector**, con:
  - immagine derivata dalla build del secondo progetto Spring Boot;
  - configurazione per l'accesso al Data DB;
  - configurazione per l'accesso al servizio User Manager via gRPC;

    - configurazione delle credenziali e dell'URL del servizio OpenSky.

Docker Compose definisce inoltre la **rete interna** che permette ai container di comunicare fra loro tramite hostname logici, evitando la dipendenza da indirizzi IP statici. Questo approccio consente di avviare l'intero sistema con un singolo comando, garantendo coerenza tra ambienti di esecuzione differenti.

3.4.3. INTEGRAZIONE CON IL SERVIZIO ESTERNO OpenSky (Oauth2 + REST)

L'integrazione con **OpenSky Network** si basa su:

- un endpoint di **token** OAuth2, utilizzato per ottenere un *access token* tramite grant type *client credentials*;
- due endpoint REST principali:
    - uno per i voli in arrivo su un determinato aeroporto in un intervallo temporale;
    - uno per i voli in partenza da un determinato aeroporto nello stesso intervallo.

Il client OpenSky:

- legge le **credenziali applicative** (client id e client secret) dalla configurazione o dalle variabili d'ambiente;
- invia una richiesta al token endpoint per ottenere il **bearer token**;
- memorizza il token insieme alla sua scadenza, così da riutilizzarlo fino alla scadenza, riducendo il numero di autenticazioni;
- include il token nell'header `Authorization` delle richieste HTTP ai servizi di arrivi e partenze;
- gestisce le risposte HTTP, distinguendo tra:
    - risposte con dati di volo validi;
    - errori di rete o di autenticazione;
    - risposte che indicano l'assenza di voli dell'intervallo (es. 404 interpretato come "nessun volo").

Questa integrazione è incapsulata all'interno del microservizio Data Collector, che si occupa di trasformare i dati grezzi in entità di dominio e di persisterli nel Data DB.

3.5. MOTIVAZIONI DELLE SCELTE ARCHITETTURALI

Le scelte architetturali adottate sono guidate dalla volontà di ottenere una soluzione **modulare**, **scalabile** e **manutenibile**, in grado di gestire in modo robusto l'integrazione con un servizio dati esterno e la separazione dei domini applicativi.

3.5.1. SEPARAZIONE DEI DOMINI APPLICATIVI (userdb VS datadb)

La decisione di separare il dominio *utente* (User DB) dal dominio *dati di volo* (Data DB) risponde a diversi obiettivi:

- **Isolamento dei dati sensibili**
  I dati identificativi degli utenti sono concentrati in un unico microservizio e in un unico database logico, facilitando l'applicazione di politiche di sicurezza e protezione dei dati.
- **Chiarezza dei confini di responsabilità**
  Il User Manager è l'unico responsabile del ciclo di vita degli utenti, mentre il Data Collector governa esclusivamente aeroporti, interessi e voli. Questa ripartizione facilita l'evoluzione dei domini senza interferenze reciproche.
- **Facilità di evoluzione e migrazione**
  In prospettiva, ciascun dominio potrebbe essere spostato su infrastrutture diverse (ad esempio database separati, istanze gestite, cluster distinti) senza richiedere interventi profondi sull'altro dominio.

La separazione logica dei database, anche se ospitati nella stessa istanza PostgreSQL, consente di rispettare i principi di **bounded context** tipici dell'approccio domain-driven, pur mantenendo una complessità infrastrutturale contenuta.

3.5.2. ADOZIONE DI MICROSERVIZI E COMUNICAZIONI REST/gRPC

L'adozione di una **architettura a microservizi** permette di:

- distribuire le responsabilità su servizi autonomi, ciascuno con il proprio ciclo di vita;
- scalare indipendentemente in base al carico: ad esempio il Data Collector può richiedere maggiori risorse per la raccolta massiva dei voli, mentre il User Manager potrebbe avere carichi più moderati ma costanti;
- semplificare il *deployment* e l'aggiornamento di parti del sistema senza dover riavviare l'intera applicazione.

Per le comunicazioni sono stati adottati due paradigmi complementari:

- **REST/HTTP**
  Utilizzato per le interazioni tra client esterno e microservizi. Questo approccio è ampiamente supportato da strumenti di test, linguaggi e framework, e risulta naturale per l'esposizione di API verso l'esterno.
- **gRPC**
  Utilizzato per la comunicazione *service-to-service* tra Data Collector e User Manager. Tale scelta consente:
  - una definizione rigorosa dei contratti tramite file `.proto`;
  - la generazione automatica di client e server type-safe;
  - maggiore efficienza nella serializzazione e nei tempi di risposta rispetto ad una comunicazione REST interna, soprattutto in scenari con molte chiamate tra microservizi.

Questo mix di tecnologie di comunicazione permette di sfruttare i punti di forza di ciascun paradigma in base al tipo di interazione e al grado di *coupling* desiderato.

3.5.3. SCELTE TECNOLOGICHE (Spring Boot, JPA, Flyway, Postgres)

Le tecnologie adottate sono state selezionate in base a criteri di maturità, diffusione e capacità di supportare in modo naturale i requisiti del sistema:

- **Spring Boot**
  Fornisce un ambiente consolidato per lo sviluppo di microservizi in Java, con:
  - integrazione nativa per REST, gRPC (tramite estensioni), pianificazione (`@Scheduled`);
  - gestione semplificata della configurazione;
  - supporto integrato per osservabilità e logging.
- **Spring Data JPA**
  Consente di interfacciarsi al database relazionale tramite un livello di astrazione basato su entità e repository, riducendo il codice boilerplate per la gestione delle query e rendendo il modello dati più espressivo.
- **Flyway**
  Gestisce le **migrazioni dello schema** in modo dichiarativo e versionato, garantendo:
  - consistenza tra ambienti (sviluppo, test, produzione);
  - controllo puntuale delle modifiche al modello dati;
  - facilità di rollback in caso di errori.
- **PostgreSQL**
  È un database relazionale maturo, open-source e largamente utilizzato, che offre:
  - robustezza e affidabilità;
  - supporto per transazioni, vincoli e indici;
  - buona integrazione con Spring Boot e gli strumenti di migrazione.

Queste scelte tecnologiche concorrono alla realizzazione di un sistema che combina **solidità infrastrutturale** e **flessibilità architetturale**, riducendo il rischio di complessità accidentale e favorendo l'estensione futura delle funzionalità.

4. MODELLAZIONE E GESTIONE DEI DATI

4.1. REQUISITI INFORMATIVI DEL SISTEMA

La modellazione dei dati è guidata dai requisiti informativi derivanti dai due domini principali del sistema:

- **dominio utente**, gestito dal microservizio *User Manager*;
- **dominio aeroporti–interessi–voli**, gestito dal microservizio *Data Collector*.

Dal punto di vista informativo, il sistema deve poter rappresentare in modo strutturato:

- l'**identità degli utenti**, identificati univocamente da un indirizzo e-mail e associati ad attributi descrittivi di base (es. nome, data di creazione);
- il **catalogo degli aeroporti**, identificati da un codice univoco (ad esempio ICAO) e arricchiti da metadati (nome, città, paese, fuso orario);
- le **relazioni di interesse** tra utenti e aeroporti, con la proprietà che per ogni coppia *(utente, aeroporto)* possa esistere **al più un interesse** attivo;
- le **registrazioni di volo** (*flight records*), che rappresentano gli eventi di arrivo/partenza rilevati per un dato aeroporto e in un dato intervallo temporale, con attributi quali:
    - identificativo esterno del volo;
    - numero del volo (quando presente);
    - direzione (*ARRIVAL* / *DEPARTURE*);
    - orari pianificati ed effettivi;
    - eventuali informazioni di stato e ritardo;
    - timestamp di raccolta da parte dell'applicazione.

A questi requisiti si aggiungono vincoli informativi trasversali:

- necessità di **ricercare e ordinare** i voli per intervalli temporali (es. ultimo volo per actual time, conteggio dei voli in una finestra [*from*, *to*]);
- necessità di **conteggiare** i voli in insiemi filtrati per aeroporto, direzione e intervallo di tempo in vista di statistiche (es. media dei voli per giorno);
- necessità di preservare **idempotenza e consistenza**:
    - evitando duplicazioni di utenti con la stessa e-mail;
    - evitando duplicazioni di interessi per la stessa coppia *(utente, aeroporto)*;
    - garantendo coerenza tra interessi registrati e utenti realmente esistenti.

Su questa base è stato definito uno schema dati relazionale che implementa in modo esplicito entità e relazioni necessarie al funzionamento del sistema.

4.2. SCHEMA LOGICO DEI DATABASE

Lo schema logico complessivo è articolato su due database logici distinti:

- **User DB**, dedicato alle informazioni sugli utenti;
- **Data DB**, dedicato agli aeroporti, agli interessi e ai flight records.

La scelta relazionale consente di esprimere in modo naturale le **relazioni uno-a-molti** e **molti-a-molti** presenti nel dominio, sfruttando chiavi primarie, chiavi esterne e vincoli di unicità.

4.2.1. DIAGRAMMA ER – SCHEMA LOGICO DEI DATABASE (User DB e Data DB)

In questa sezione viene inserita l'immagine dell'"**diagramma ER – schema logico dei database (User DB e Data DB)**", che rappresenta la struttura complessiva dei dati gestiti dal sistema e le relazioni tra le principali entità applicative, suddivise nei due domini **User DB** e **Data DB**.

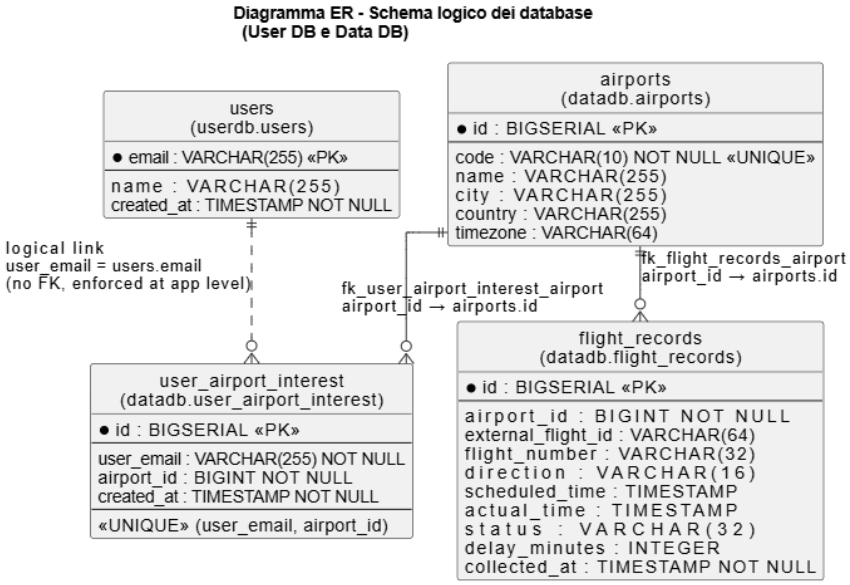
Lo schema evidenzia:

- nel **User DB**:
  - l'entità `users`, con chiave primaria sull'attributo `email`;
- nel **Data DB**:
  - l'entità `airports`, con chiave primaria `id` e vincolo di unicità su `code`;
  - l'entità `user_airport_interest`, con chiave primaria `id`, attributo `user_email` (collegamento logico all'utente) e chiave esterna `airport_id` verso `airports`;
  - l'entità `flight_records`, con chiave primaria `id` e chiave esterna `airport_id` verso `airports`.

Le relazioni principali modellate sono:

- `airports` **1–N** `user_airport_interest`: un aeroporto può essere collegato a molti interessi;
- `airports` **1–N** `flight_records`: un aeroporto può avere molti flight records associati;
- **collegamento logico tra** `users` **e** `user_airport_interest` tramite l'attributo `user_email`, gestito a livello applicativo anziché come vincolo di chiave esterna cross-database.

Il diagramma mette inoltre in evidenza il vincolo di unicità sulla coppia `(user_email, airport_id)` nella tabella degli interessi, fondamentale per garantire la semantica *at-most-once* nella registrazione degli interessi utente–aeroporto.



4.2.2. USER DB: MODELLAZIONE DELL'ENTITA' UTENTE

Il **User DB** è concepito per mantenere in modo minimale ma completo le informazioni relative agli utenti del sistema. La struttura principale è costituita dalla tabella:

- `users (userdb.users)`

con i seguenti attributi significativi:

- `email : VARCHAR(255)` – **chiave primaria (**`PK`**)** e identificatore univoco dell'utente;
- `name : VARCHAR(255)` – nome dell'utente, utilizzato a fini descrittivi;
- `created_at : TIMESTAMP NOT NULL` – timestamp di creazione dell'utente.

L'uso dell'e-mail come chiave primaria ha le seguenti implicazioni:

- consente di identificare in modo immediato l'utente a partire dal dato che è utilizzato come input in quasi tutte le interazioni (registro interessi, validazione gRPC);
- permette di evitare un ulteriore identifier surrogato (ad esempio un ID numerico), riducendo il numero di join necessari in molte query applicative.

La tabella è gestita tramite migrazioni **Flyway**, che garantiscono l'allineamento coerente dello schema in tutti gli ambienti, e viene acceduta unicamente dal microservizio *User Manager* attraverso il layer JPA.

4.2.3. DATA DB: AEREPORTI, INTERESSI UTENTE E REGISTRAZIONI DI VOLO

Il **Data DB** è il database logico dedicato al dominio:

- *aeroporti*,
- *interessi utente–aeroporto*,
- *flight records*.

Le tabelle principali sono:

- `airports (datadb.airports)`
    - `id : BIGSERIAL` – **chiave primaria (**`PK`**)**;
    - `code : VARCHAR(10) NOT NULL` – codice univoco dell'aeroporto, con vincolo **UNIQUE**;
    - `name : VARCHAR(255)` – nome descrittivo dell'aeroporto;
    - `city : VARCHAR(255)` – città in cui si trova l'aeroporto;
    - `country : VARCHAR(255)` – paese;
    - `timezone : VARCHAR(64)` – fuso orario associato.

La presenza del vincolo di unicità su `code` consente di utilizzare tale campo come chiave naturale in tutte le interazioni esterne (API REST, configurazioni), mantenendo `id` come identificatore interno efficiente per join e chiavi esterne.

- `user_airport_interest (datadb.user_airport_interest)`
    - `id : BIGSERIAL` – **chiave primaria (**`PK`**)**;
    - `user_email : VARCHAR(255) NOT NULL` – e-mail dell'utente, collegamento logico verso la tabella `users` del User DB;
    - `airport_id : BIGINT NOT NULL` – chiave esterna verso airports.id;
    - `created_at : TIMESTAMP NOT NULL` – timestamp di creazione dell'interesse;
    - vincolo **UNIQUE** sulla coppia `(user_email, airport_id)`.

Questo modello implementa a livello logico la relazione **molti-a-molti** tra *utente* e *aeroporto* come entità associativa autonoma. Il vincolo di unicità garantisce che per ogni coppia *(utente, aeroporto)* esista al più un record di interesse, fondamento per le politiche di idempotenza nella registrazione degli interessi.

- `flight_records (datadb.flight_records)`
    - `id : BIGSERIAL` – **chiave primaria (**`PK`**)**;
    - `airport_id : BIGINT NOT NULL` – chiave esterna verso airports.id;
    - `external_flight_id : VARCHAR(64)` – identificativo esterno del volo (derivato dai dati OpenSky, ad esempio combinazione di `icao24`, `firstSeen`, `lastSeen`);
    - `flight_number : VARCHAR(32)` – numero di volo, se disponibile;
    - `direction : VARCHAR(16)` – direzione del volo, tipicamente *ARRIVAL* o *DEPARTURE*;
    - `scheduled_time : TIMESTAMP` – orario pianificato, se presente nei dati sorgente;

- `actual_time : TIMESTAMP` – orario effettivo dell'evento (arrivo/partenza) utilizzato come riferimento temporale principale;
- status : VARCHAR(32) – eventuale stato del volo;
- delay_minutes : INTEGER – eventuale ritardo stimato in minuti;
- collected_at : TIMESTAMP NOT NULL – timestamp della raccolta da parte del sistema.

Questa struttura consente di:

- memorizzare **più registrazioni storiche** per ciascun aeroporto;
- esprimere query efficienti per:
  - ultime occorrenze (ordinamento per `actual_time` e limit);
  - conteggi su intervalli temporali (`actual_time BETWEEN from AND to`);
  - estrazione di serie di voli filtrati per aeroporto e direzione.

Il legame tra `flight_records` e `airports` è realizzato tramite chiave esterna `airport_id`, garantendo integrità referenziale a livello di database per tutti i dati di volo memorizzati.

4.3. MAPPATURA ORM (JPA) E CONVENZIONI ADOTTATE

L'accesso ai dati è mediato da **Spring Data JPA**, che fornisce una mappatura oggetto–relazionale tra il modello di dominio Java e lo schema relazionale dei database.

Per ciascuna tabella principale è definita un'entità JPA:

- `User` per la tabella `users`;
- `Airport` per la tabella `airports`;
- `UserAirportInterest` per la tabella `user_airport_interest`;
- `FlightRecord` per la tabella `flight_records`.

Le convenzioni adottate includono:

- Annotazione delle classi di dominio con `@Entity` e `@Table`, specificando il nome effettivo della tabella e, ove necessario, eventuali indici o vincoli;
- Utilizzo di annotazioni `@Id` e `@GeneratedValue` (per le chiavi numeriche) oppure di campi naturali (ad esempio l'e-mail per `User`) come chiave primaria;
- Mappatura delle relazioni:
  - associazione *molti-a-uno* (`@ManyToOne`) tra `UserAirportInterest` e `Airport`, realizzata tramite `airport_id`;
  - associazione *molti-a-uno* (`@ManyToOne`) tra `FlightRecord` e `Airport`;
- Rappresentazione delle enumerazioni (ad esempio la **direzione del volo**) con tipi Java `enum`, mappati su colonne `VARCHAR` tramite

`@Enumerated(EnumType.STRING)` al fine di garantire leggibilità e stabilità dei valori memorizzati.

Sul piano dei repository, sono state definite interfacce che estendono `JpaRepository` o simili, con metodi che seguono le convenzioni di naming di Spring Data:

- per l'entità `User`:
  - `findById(email)`, `existsById(email)`, `save(user)`;
- per l'entità `Airport`:
  - `findByCode(code)`, `findAll()`;
- per l'entità `UserAirportInterest`:
  - `findByUserEmailAndAirport(userEmail, airport)`,
  - metodi per elencare gli interessi di un utente o gli aeroporti distinti con interessi attivi;
- per l'entità `FlightRecord`:
  - `findFirstByAirportAndDirectionOrderByActualTimeDesc(...)` per ottenere l'ultimo volo;
  - `countByAirportAndDirectionAndActualTimeBetween(...)` per i conteggi su intervalli;
  - `findByAirportAndDirectionAndActualTimeBetweenOrderByActualTimeDesc(...)` per ottenere liste ordinate di voli.

L'utilizzo di **metodi derivati** dal nome, tipici di Spring Data, permette di ridurre la quantità di query esplicite, concentrando la logica di business nei service e mantenendo una chiara separazione tra accesso dati e logica applicativa.

Per le operazioni di sola lettura (interrogazione dei voli, conteggi statistici) vengono utilizzate annotazioni come `@Transactional(readOnly = true)`, a indicare la natura non mutativa delle transazioni e a consentire eventuali ottimizzazioni da parte del framework e del database.

4.4. COERENZA TRA DOMINI APPLICATIVI

La presenza di due domini distinti – *utente* e *dati di volo* – richiede meccanismi espliciti per garantire la **coerenza applicativa** tra i dati memorizzati nei rispettivi database, in assenza di vincoli di chiave esterna inter-database.

La coerenza è ottenuta tramite:

- un **collegamento logico** tra utenti e interessi, basato sull'e-mail;
- **policy di validazione applicativa** che impediscono la creazione di interessi per utenti inesistenti;
- **politiche di idempotenza** per evitare duplicazioni e garantire comportamenti deterministici.

4.4.1. COLLEGAMENTO LOGICO TRA UTENTI E INTERISSI (user_email)

La tabella `user_airport_interest` nel Data DB utilizza l'attributo `user_email` come riferimento all'utente corrispondente presente nel User DB. Non viene definito un vincolo di chiave esterna a livello di database, poiché:

- i due domini sono gestiti da microservizi distinti, con responsabilità separate;
- i database logici, pur condividendo la stessa istanza PostgreSQL, sono concepiti come *bounded context* separati.

Il collegamento tra `user_email` e `users.email` è quindi gestito **a livello applicativo**:

- prima di creare un interesse, il Data Collector invoca il servizio gRPC esposto dal User Manager per verificare l'esistenza dell'utente;
- solo in caso di risposta positiva viene permessa la persistenza del record in `user_airport_interest`.

In questo modo, la coerenza tra dominii viene garantita non tramite vincoli SQL cross-database, ma tramite il rispetto rigoroso dei contratti di servizio tra i microservizi.

4.4.2. STRATEGIE PER GARANTIRE CONSISTENZA APPLICATIVA

La consistenza applicativa è ottenuta mediante un insieme di strategie complementari:

- **Validazione preventiva via gRPC**
  La creazione di un record in `user_airport_interest` è subordinata alla conferma, da parte del User Manager, che l'e-mail fornita corrisponda a un utente registrato. Questo meccanismo evita l'introduzione di interessi orfani (cioè, associati a utenti inesistenti).
- **Controllo di idempotenza sugli interessi**
  Il vincolo di unicità su `(user_email, airport_id)` e la logica applicativa nel service del Data Collector garantiscono che:
  - se un interesse esiste già, una nuova richiesta con gli stessi parametri non produce un duplicato;
  - le operazioni possono essere ripetute dal client senza effetti collaterali indesiderati, caratteristica fondamentale in scenari distribuiti dove le richieste possono essere ritentate.
- **Controllo di idempotenza sulla registrazione utente**
  Nel dominio utente, la registrazione verifica la presenza di un record con la stessa e-mail prima di procedere all'inserimento, garantendo una semantica *at-most-once* anche per la creazione dell'utente.
- **Vincoli relazionali interni al Data DB**
  Le chiavi esterne da `user_airport_interest` e `flight_records` verso `airports` assicurano che:
  - non possano esistere interessi o flight records associati ad aeroporti inconsistente;

    - la cancellazione di un aeroporto richieda una gestione esplicita delle entità collegate, prevenendo inconsistenze nel Data DB.

- **Gestione esplicita dei casi di errore**
  Il sistema distingue tra:
    - errori di **dominio** (utente inesistente, aeroporto non trovato, volo non presente), che impediscono la creazione o lettura di dati incoerenti;
    - errori di **validazione dei parametri** (direzioni non ammesse, intervalli temporali non validi), che vengono intercettati a livello di API e non raggiungono il livello di persistenza.

Queste strategie, combinate con l'uso disciplinato dei servizi di validazione e dei vincoli di database, permettono di mantenere un elevato livello di coerenza tra i due domini applicativi, nonostante la scelta di una separazione netta a livello di microservizi e di database.

5. DETTAGLIO DEI MICROSERVIZI E DELLE COMUNICAZIONI

5.1. USER MANAGER SERVICE

Il **User Manager Service** realizza il *bounded context* del dominio utente. Espone API REST per la gestione del ciclo di vita degli utenti e un servizio gRPC dedicato alla validazione della loro esistenza, utilizzato dagli altri microservizi come fonte autorevole. Tutta la logica relativa alla creazione, lettura e cancellazione degli utenti, nonché alla verifica dell'unicità dell'e-mail, è incapsulata in questo servizio, che opera in modo esclusivo sul proprio database logico (*User DB*).

5.1.1. STRUTTURA DEI PACKAGE E COMPONENTI PRINCIPALI

La struttura interna del User Manager segue una suddivisione per layer coerente con le best practice dei microservizi basati su Spring Boot. A livello logico, si possono distinguere i seguenti gruppi di package e componenti:

- **Layer di esposizione REST (`controller`)**
  Contiene il *REST controller* principale, ad esempio `UserController`, responsabile di:
  - mappare le richieste HTTP sugli endpoint `/api/users`;
  - ricevere e validare i DTO di input (es. `UserRegistrationRequest`, `UserResponse`);
  - delegare la logica di business al layer di servizio;
  - comporre le risposte HTTP (codice di stato e payload JSON).
- **Layer di servizio (`service`)**
  Contiene `UserService`, che incapsula la logica applicativa del dominio utente:
  - applicazione della politica di **idempotenza** in fase di registrazione;
  - orchestrazione delle operazioni di lettura e cancellazione;
  - fornitura di metodi dedicati alla verifica dell'esistenza dell'utente, utilizzati sia dal controller REST sia dal servizio gRPC.
- **Layer di accesso dati (`repository`)**
  Implementato tramite `UserRepository`, basato su Spring Data JPA, che:
  - mappa l'entità `User` sulla tabella `users` del *User DB*;
  - espone metodi come `existsById(email)`, `findById(email)`, `save(user)`, aderenti alle convenzioni JPA;
  - delega a JPA la generazione delle query SQL, mantenendo il codice focalizzato sulla logica applicativa.
- **Modello di dominio (entity / model)**
  Comprende l'entità `User`, annotata con `@Entity` e `@Table`, con:
  - campo `email` come chiave primaria;

    - campi descrittivi come `name` e `createdAt`.

- **Servizio gRPC (`grpc` o equivalente)**
  Include l'implementazione del servizio, ad esempio `UserValidationGrpcService`, che:
    - realizza le interfacce generate dal file `.proto`;
    - espone un metodo di validazione (ad esempio `checkUserExists` o equivalente), delegando al `UserService` la verifica sul repository.
- **Gestione delle eccezioni (`exception`)**
  Contiene un componente di tipo `RestExceptionHandler` che:
    - intercetta eccezioni applicative (utente non trovato, violazioni di validazione, ecc.);
    - le traduce in risposte HTTP standardizzate con un payload di errore coerente.

La configurazione generale (connessione al database, porte esposte per REST e gRPC, properties applicative) è centralizzata in classi e file di configurazione dedicati (`application.yml` / `application.properties` e relative classi `@Configuration`).

5.1.2. GESTIONE DELLA REGISTRAZIONE UTENTE E POLITICA AT-MOST-ONCE

La **registrazione utente** è un'operazione critica, per la quale è stata adottata una politica di tipo **at-most-once**. L'obiettivo è garantire che, a parità di e-mail, l'utente venga creato *al più una volta*, anche in presenza di richieste ripetute o ritentate dal client.

Il flusso logico è il seguente:

1. Il `UserController` riceve una richiesta `POST /api/users` con payload contenente `email` e `name`.
2. Le annotazioni di validazione (`@Valid` e vincoli sui campi) verificano che:
    - l'e-mail sia presente e formalmente valida;
    - il nome utente rispetti eventuali vincoli sintattici.
3. Il `UserService` esegue una **verifica preventiva** attraverso `userRepository.existsById(email)`:
    - se l'utente esiste già:
        - la registrazione viene considerata **idempotente**;
        - il servizio può restituire i dati dell'utente esistente, tipicamente con un codice di stato che indica esito positivo senza nuova creazione (es. `200 OK`).
    - se l'utente non esiste:
        - viene creato un nuovo oggetto `User` con `createdAt` impostato al timestamp corrente;

- viene invocato `userRepository.save(user)` per persistere il record;
- la risposta HTTP riflette l'avvenuta creazione (es. `201 Created`).

Eventuali anomalie (ad esempio violazioni di vincoli o errori di validazione) sono intercettate dal `RestExceptionHandler`, che le converte in errori strutturati (es. `400 Bad Request` con dettagli sugli errori di campo).

Questa impostazione garantisce:

- l'**unicità dell'e-mail** a livello di database (tramite chiave primaria);
- la tolleranza rispetto a richieste ripetute, evitando la creazione di duplicati;
- una semantica chiara per il client, che ottiene sempre una rappresentazione consistente dell'utente associato a quell'indirizzo e-mail.

5.1.3. ESPOSIZIONE DEL SERVIZIO GRPC DI VALIDAZIONE UTENTE

Oltre alle API REST, il User Manager espone un **servizio gRPC** dedicato alla validazione dell'esistenza degli utenti. Ciò consente agli altri microservizi di:

- delegare al User Manager la **verifica dell'identità**;
- evitare accessi diretti al *User DB*;
- basare le proprie decisioni su una fonte autorevole e centralizzata.

Gli elementi tecnici principali sono:

- **Definizione del contratto (`.proto`)**
  Il file `.proto` definisce:
  - il servizio, ad esempio `UserValidationService`;
  - i messaggi `UserValidationRequest` (contenente l'e-mail) e `UserValidationResponse` (contenente un boolean `exists` e, opzionalmente, ulteriori campi informativi).
- **Implementazione server (UserValidationGrpcService)**
  Il servizio gRPC server-side:
  - riceve la richiesta dal client (es. Data Collector);
  - invoca il `UserService` per verificare l'esistenza dell'utente tramite il `UserRepository`;
  - costruisce la risposta gRPC valorizzando il campo `exists` in base all'esito della verifica.
- **Integrazione con il sistema**
  L'endpoint gRPC viene esposto su una porta dedicata, configurata in `application.yml` e resa raggiungibile all'interno della rete Docker tramite hostname del container `user-manager-service`. Il Data Collector utilizza un client gRPC generato a partire dallo stesso `.proto`, garantendo compatibilità strutturale e semantica.

Questo servizio funge da **punto di coesione** tra domini: ogni volta che un altro microservizio ha necessità di verificare che una certa e-mail corrisponda a un utente registrato, la richiesta viene inoltrata al User Manager, che rimane l'unico *owner* dei dati utente.

5.2. DATA COLLECTOR SERVICE

Il **Data Collector Service** costituisce il fulcro del dominio aeroporti–interessi–voli. Si occupa di gestire gli interessi utente verso gli aeroporti, di orchestrare la raccolta periodica dei dati di volo da OpenSky e di fornire API REST per l'interrogazione dei flight records. È progettato per operare in modalità mista **event-driven temporale** (scheduler) e **request-driven** (chiamate REST del client).

5.2.1. STRUTTURA DEI PACKAGE E COMPONENTI PRINCIPALI

Anche il Data Collector segue una chiara separazione per layer e responsabilità. La struttura logica include:

- **Controller REST**
  - `UserInterestController`
    Gestisce le operazioni su /api/interests:
    - registrazione di un interesse (POST);
    - rimozione di un interesse (DELETE);
    - interrogazione degli interessi di un utente (GET).
  - `FlightQueryController`
    Gestisce le operazioni di interrogazione voli su `/api/flights`, `/api/flights/last`, `/api/flights/average`.
- **Service layer**
  - `UserInterestService`
    Contiene la logica per:
    - validare la richiesta di creazione degli interessi;
    - invocare il servizio gRPC del User Manager per verificare l'esistenza dell'utente;
    - evitare duplicati grazie al controllo preventivo (query + vincoli di unicità) su `user_airport_interest`.
  - `FlightCollectionService`
    Responsabile della logica di raccolta dei voli per singolo aeroporto o per tutti gli aeroporti di interesse:
    - interrogazione di OpenSky tramite `OpenSkyClient`;
    - trasformazione dei dati esterni in `FlightRecord`;
    - salvataggio in batch tramite `FlightRecordRepository`.

- `FlightQueryService`
  Implementa la logica di interrogazione dei voli:
  - ricerca dell'ultimo volo per aeroporto/direzione;
  - calcolo della media dei voli in un intervallo;
  - estrazione di liste di voli filtrate e ordinate.

- **Scheduler (`FlightCollectionScheduler`)**
  Configurato mediante annotazioni `@Scheduled`, richiama periodicamente metodi di `FlightCollectionService` per avviare la raccolta dei dati sui voli per tutti gli aeroporti con interessi attivi.
- **Repository JPA**
  - `AirportRepository` per l'accesso alla tabella `airports`;
  - `UserAirportInterestRepository` per la tabella `user_airport_interest`, con metodi dedicati (es. `findDistinctAirportsOfInterest`, `findByUserEmailAndAirport`);
  - `FlightRecordRepository` per la tabella `flight_records`, con metodi di conteggio e selezione basati su intervalli temporali e direzione.
- **Client esterni**
  - `UserValidationGrpcClient`, che incapsula la comunicazione gRPC verso il User Manager per la validazione dell'utente;
  - `OpenSkyClient`, che gestisce l'autenticazione OAuth2 e le chiamate REST verso le API di OpenSky.
- **Gestione delle eccezioni (`RestExceptionHandler`)**
  Centralizza il trattamento delle eccezioni applicative (utente inesistente, aeroporto non trovato, volo non presente, parametri non validi) e le traduce in risposte HTTP standardizzate.

Questa organizzazione consente di mantenere il **core della logica di business** concentrato nei service, mentre controller, repository, client esterni e scheduler svolgono ruoli ben definiti di orchestrazione, accesso al dato e integrazione.

5.2.2. GESTIONE DEGLI INTERESSI UTENTE VERSO GLI AEROPORTI

La gestione degli **interessi utente–aeroporto** rappresenta il punto di contatto tra il dominio utente e il dominio aeroporti–voli. Il flusso tipico per la registrazione di un interesse è strutturato come segue:

1. Il `UserInterestController` riceve una richiesta `POST /api/interests` contenente:
   - `userEmail`: e-mail dell'utente;
   - `airportCode`: codice dell'aeroporto di interesse.

2. Vengono eseguite verifiche preliminari:
    - validazione sintattica dei parametri;
    - normalizzazione di eventuali valori (es. maiuscole/minuscole per il codice aeroporto).
3. Il `UserInterestService` invoca il `UserValidationGrpcClient` che, tramite gRPC, contatta il servizio `UserValidationService` del User Manager:
    - se il servizio indica che l'utente **non esiste**, viene sollevata un'eccezione di dominio (`UserNotFoundException` o equivalente) tradotta in `404 Not Found`;
    - se l'utente esiste, si procede oltre.
4. Il servizio interroga l'`AirportRepository` per verificare l'esistenza dell'aeroporto corrispondente al code fornito:
    - se l'aeroporto non è presente nel catalogo, viene sollevata un'eccezione (`AirportNotFoundException`) tradotta in `404 Not Found`;
    - in caso contrario, si ottiene l'entità Airport.
5. Il `UserAirportInterestRepository` viene utilizzato per verificare se esista già un record per la coppia `(userEmail, airport)`:
    - se l'interesse esiste già:
        - la richiesta viene trattata in modo *idempotente*;
        - si restituisce al client una rappresentazione dell'interesse esistente (es. `200 OK`);
    - se l'interesse non esiste:
        - viene creato un nuovo `UserAirportInterest` con `createdAt` impostato al timestamp corrente;
        - viene eseguita l'operazione di persistenza (`save`), nel rispetto del vincolo di unicità `(user_email, airport_id)` definito a livello di database;
        - si restituisce al client una risposta che indica l'avvenuta creazione dell'interesse (es. `201 Created`).

La rimozione di un interesse (`DELETE /api/interests`) segue un flusso simile, ma si conclude con la cancellazione del record se presente, oppure con un errore `404` in caso di interesse inesistente, a seconda della scelta di design adottata per semantica dell'operazione.

5.2.3. SCHEDULER PER LA RACCOLTA PERIODICA DEI VOLI

La **raccolta dei voli** è gestita da uno scheduler interno al Data Collector, che consente al sistema di operare in modalità *time-driven* indipendentemente dalle richieste del client esterno.

Le caratteristiche principali sono:

- **Trigger periodico**
  Una classe `FlightCollectionScheduler` definisce uno o più metodi annotati con `@Scheduled` (es. `fixedRate` o `cron`), che:
  - calcolano l'intervallo temporale di interesse per la raccolta (es. ultima ora o ultime 24 ore);
  - invocano il `FlightCollectionService` per orchestrare la raccolta sui vari aeroporti.
- **Selezione degli aeroporti rilevanti**
  Il `FlightCollectionService` utilizza il `UserAirportInterestRepository` per ottenere l'elenco degli aeroporti per cui esiste almeno un interesse utente:
  - la query tipica utilizza una selezione dei `DISTINCT airport` dalla tabella `user_airport_interest`;
  - in assenza di interessi, il servizio non effettua chiamate verso OpenSky, riducendo il carico inutile.
- **Raccolta per singolo aeroporto**
  Per ciascun aeroporto di interesse:
  - viene invocato l'`OpenSkyClient` per recuperare:
    - i voli in arrivo (`arrivals`) nell'intervallo [*begin*, *end*];
    - i voli in partenza (`departures`) nello stesso intervallo;
  - le risposte vengono mappate in liste di `FlightRecord`:
    - impostando `airport` come FK verso l'entità `Airport`;
    - derivando `externalFlightId`, `flightNumber`, `direction`, `actualTime`, e altri campi;
    - impostando `collectedAt` al momento della raccolta.
- **Persistenza in batch**
  Le liste di `FlightRecord` vengono salvate tramite `FlightRecordRepository`, preferibilmente attraverso operazioni batch (es. `saveAll`), riducendo il numero di round trip verso il database.

- **Gestione dei casi particolari**
  Lo scheduler e il servizio di raccolta gestiscono esplicitamente:
    - il caso in cui OpenSky non restituisca voli per un dato intervallo (liste vuote, eventuali codici HTTP specifici come 404 interpretati come nessun volo);
    - eventuali errori di rete o problemi temporanei di disponibilità del servizio esterno, che possono essere gestiti con log dedicati e, se previsto, politiche di *retry* o backoff (estendibili in fasi successive).

Questo meccanismo rende il sistema capace di mantenere **aggiornato il Data DB** in modo regolare, senza richiedere interventi manuali o trigger esterni.

5.2.4. CLIENT OpenSky: AUTENTICAZIONE Oauth2 E CHIAMATE REST

Il componente `OpenSkyClient` incapsula l'intera logica di integrazione con il servizio esterno **OpenSky Network**. Le responsabilità principali sono:

- **Gestione dell'autenticazione OAuth2**
    - Lettura delle credenziali (`client_id`, `client_secret`, endpoint di token) dalla configurazione applicativa o dalle variabili d'ambiente;
    - invio di una richiesta al token endpoint OAuth2 di OpenSky, utilizzando il grant type *client credentials*;
    - parsing della risposta per ottenere l'`access_token` e il relativo `expires_in`;
    - memorizzazione del token e della sua scadenza in memoria locale (cache) in modo da:
        - riutilizzare il token finché valido;
        - evitare richieste ridondanti al token endpoint.
- **Costruzione delle richieste REST**
    - Costruzione degli URL verso gli endpoint di:
        - **arrivals** (voli in arrivo su un aeroporto in [*begin*, *end*]);
        - **departures** (voli in partenza da un aeroporto in [*begin*, *end*]);
    - aggiunta del header `Authorization: Bearer <token>` alle richieste HTTP;
    - utilizzo di un client HTTP (ad esempio `RestTemplate` o `WebClient`) per eseguire le chiamate.
- **Mapping delle risposte**
    - Conversione del payload JSON in oggetti DTO, ad esempio `OpenSkyFlightDto`, che rappresentano il formato delle risposte di OpenSky;
    - gestione delle casistiche:
        - risposta con lista di voli;
        - risposta vuota;

        - codici HTTP specifici (es. 404 utilizzato per indicare l'assenza di voli nell'intervallo).

- **Integrazione con il servizio di raccolta**
  I metodi `getArrivals(airportCode, begin, end)` e `getDepartures(airportCode, begin, end)` espongono al `FlightCollectionService` un'interfaccia astratta, indipendente dai dettagli del protocollo HTTP e della struttura JSON. Il servizio può così concentrarsi sul mapping verso le entità `FlightRecord` e sulla logica di persistenza.

L'astrazione fornita da `OpenSkyClient` consente di limitare l'impatto di eventuali cambiamenti futuri del servizio esterno (variazioni di endpoint, parametri o formato delle risposte) a un'unica porzione di codice, preservando la stabilità del resto del sistema.

5.3. PATTERN DI COMUNICAZIONE ADOTTATI

L'architettura adotta una combinazione di **comunicazioni sincrone** (REST e gRPC) e **comunicazioni schedulate**, coerente con la natura del dominio: da un lato interazioni *request–response* con client esterni e microservizi, dall'altro processi periodici di raccolta dati.

5.3.1. COMUNICAZIONI SINCRONE REST (CLIENT ↔ MICROSERVIZI)

Le comunicazioni tra **client esterno** e microservizi (User Manager, Data Collector) avvengono tramite API **REST/HTTP**, seguendo le linee guida di un design *resource-oriented*:

- utilizzo esplicito dei verbi HTTP (`GET`, `POST`, `DELETE`) per esprimere il tipo di operazione;
- utilizzo di path semantici per identificare le risorse (`/api/users`, `/api/interests`, `/api/flights`);
- scambio di payload in formato **JSON**, con:
    - DTO di input ben definiti per le richieste;
    - DTO di output coerenti per le risposte.

I codici di stato HTTP vengono utilizzati in modo semantico:

- `200 OK` per operazioni di lettura o di creazione/aggiornamento idempotenti;
- `201 Created` per creazioni di nuove risorse;
- `400 Bad Request` per errori di validazione dei parametri o del payload;
- `404 Not Found` per risorse inesistenti (utente, aeroporto, volo);
- altri codici (es. `409 Conflict`) possono essere utilizzati in caso di specifiche condizioni di concorrenza o violazione di vincoli.

Il pattern complessivo è quello di una **comunicazione sincrona a richiesta**, dove il client ottiene un feedback immediato sul risultato dell'operazione, con errori strutturati gestiti dal `RestExceptionHandler`.

5.3.2. COMUNICAZIONI SINCRONE gRPC (DATA COLLECTOR ↔ USER MANAGER)

La comunicazione tra **Data Collector** e **User Manager** per la validazione dell'utente utilizza gRPC, un framework RPC ad alte prestazioni basato su HTTP/2 e Protobuf. Le sue caratteristiche principali nel sistema sono:

- **Contratto di servizio rigoroso**
  Il file `.proto` definisce in modo formale:
  - il servizio `UserValidationService`;
  - i messaggi di richiesta e risposta.

  Da questo contratto vengono generati automaticamente:
  - lo *stub* server-side (implementato dal User Manager);
  - lo *stub* client-side (utilizzato dal Data Collector).
- **Chiamata sincrona a bassa latenza**
  Quando il Data Collector deve verificare l'esistenza di un utente:
  - invoca il `UserValidationGrpcClient`;
  - il client gRPC effettua una chiamata sincrona al servizio remoto, passando l'e-mail;
  - il server risponde con un boolean (`exists`) o con informazioni più ricche, consentendo al Data Collector di decidere se procedere o bloccare l'operazione.
- **Isolamento dei domini**
  Il Data Collector non conosce dettagli del modello dati interno del User Manager (struttura della tabella `users`, schema del *User DB*). L'unico contratto condiviso è il servizio gRPC, che funge da *boundary* tra i due domini.

Questo pattern garantisce una comunicazione interna **fortemente tipizzata**, efficiente e separata dalle API REST pubbliche, preservando i principi di *loose coupling* tra microservizi.

5.3.3. COMUNICAZIONI SCHEDULATE (SCHEDULER → FLIGHT COLLECTION SERVICE)

Oltre alle interazioni sincrone, il sistema adotta una forma di comunicazione **time-triggered**, realizzata tramite uno scheduler interno al Data Collector. In questo caso:

- non vi è un attore esterno che effettua una richiesta;
- il trigger è rappresentato dallo **scorrere del tempo** (esecuzione ogni N minuti).

Il flusso è:

1. Lo scheduler (`FlightCollectionScheduler`) viene attivato da Spring secondo la configurazione (`@Scheduled`).
2. Lo scheduler invoca metodi del `FlightCollectionService`, passando:

    - gli estremi temporali dell'intervallo su cui effettuare la raccolta;
    - eventuali parametri di configurazione (finestra temporale, filtri specifici).
3. Il `FlightCollectionService` esegue a sua volta:
    - query sul *Data DB* per identificare gli aeroporti di interesse;
    - chiamate a `OpenSkyClient` per l'accesso al servizio esterno;
    - scritture su `FlightRecordRepository` per la persistenza.

Questa forma di comunicazione interna può essere vista come una **cooperazione intra-microservizio** guidata dal tempo, che realizza una pipeline *pull-based* rispetto alla sorgente esterna (OpenSky) e *push-based* rispetto al database locale.

5.4. POLITICHE DI CONSISTENZA E IDEMPOTENZA

Le politiche di **consistenza** e **idempotenza** adottate nei microservizi sono fondamentali per garantire un comportamento deterministico e robusto in presenza di richieste ripetute, possibili ritardi di rete e interazioni tra domini distincti.

5.4.1. REGISTRAZIONE UTENTE (USER MANAGER, AT-MOST-ONCE)

Come già evidenziato nella descrizione del User Manager, la registrazione utente implementa una politica di tipo **at-most-once**:

- l'e-mail è definita come **chiave primaria** e rappresenta il vincolo di unicità a livello di database;
- il `UserService` esegue un controllo preventivo tramite `existsById(email)` prima di creare un nuovo record;
- se l'utente esiste già:
    - l'operazione non produce effetti collaterali;
    - viene restituita una risposta consistente con lo stato già presente;
- se l'utente non esiste:
    - viene creato un nuovo record `User`;
    - viene salvato in modo transazionale in `users`.

Questa impostazione consente al client di **ritentare** una richiesta di registrazione senza rischiare la creazione di utenti duplicati. Il sistema garantisce che per ciascun indirizzo e-mail ci sia *al più un utente*, e che l'eventuale ripetizione della richiesta porti sempre a una rappresentazione coerente di quello stesso utente.

5.4.2. REGISTRAZIONE DEGLI INTERESSI (EVITARE DUPLICATI UTENTE–AEROPORTO)

Nel dominio degli interessi utente–aeroporto, le politiche di consistenza e idempotenza sono realizzate mediante la combinazione di:

- **vincoli di unicità a livello di database**
  La tabella `user_airport_interest` definisce un vincolo **UNIQUE** sulla coppia `(user_email, airport_id)`:
  - questo impedisce, a livello relazionale, l'inserimento di due interessi con la stessa combinazione di utente e aeroporto;
  - il vincolo è coerente con il modello concettuale, in cui l'interesse è un'entità che rappresenta l'esistenza o meno di un legame utente–aeroporto, e non un contatore o uno storico di richieste.
- **controlli applicativi nel** `UserInterestService`
  Prima di eseguire l'operazione di salvataggio:
  - il servizio invoca `findByUserEmailAndAirport(userEmail, airport)` o metodo equivalente;
  - se l'interesse esiste già:
    - la richiesta viene trattata come *idempotente*;
    - l'operazione non crea nuovi record;
    - viene restituita al client la rappresentazione dell'interesse esistente;
  - se l'interesse non esiste:
    - viene creato un nuovo `UserAirportInterest`;
    - viene invocato il salvataggio (che, in presenza di concorrenza, è ulteriormente protetto dal vincolo di unicità del database).
- **validazione cross-dominio via gRPC**
  La creazione di un interesse è condizionata all'esistenza dell'utente nel dominio del User Manager:
  - il Data Collector invia una richiesta al `UserValidationService` via gRPC;
  - solo se la risposta indica `exists = true`, il sistema permette la creazione dell'interesse;
  - in caso contrario, viene sollevata un'eccezione e l'operazione viene abortita.

Queste misure garantiscono che:

- **non vengano creati interessi per utenti inesistenti**;
- **non vengano introdotti duplicati** per la stessa coppia *(utente, aeroporto)*, anche in caso di richieste ripetute o concorrenti;

- il comportamento del sistema rimanga prevedibile e coerente in scenari distribuiti, in cui il client potrebbe non conoscere con certezza l’esito dell’ultima richiesta e scegliere di ritentare l’operazione.

6. DETTAGLIO DELLE INTERAZIONI (DIAGRAMMI DI SEQUENZA)

6.1. INTERAZIONE PER LA REGISTRAZIONE UTENTE

L'interazione per la **registrazione utente** descrive il flusso completo che parte dalla richiesta del client esterno e attraversa i componenti del microservizio *User Manager* fino alla persistenza nel *User DB*. Il diagramma di sequenza mette in evidenza il ruolo dei diversi layer (controller, service, repository) e la modalità con cui viene implementata la politica *at-most-once* sulla creazione dell'utente, evitando duplicati in presenza di richieste ripetute.

6.1.1. DIAGRAMMA DI SEQUENZA – REGISTRAZIONE UTENTE (USER MANAGER, POLITICA AT-MOST-ONCE)

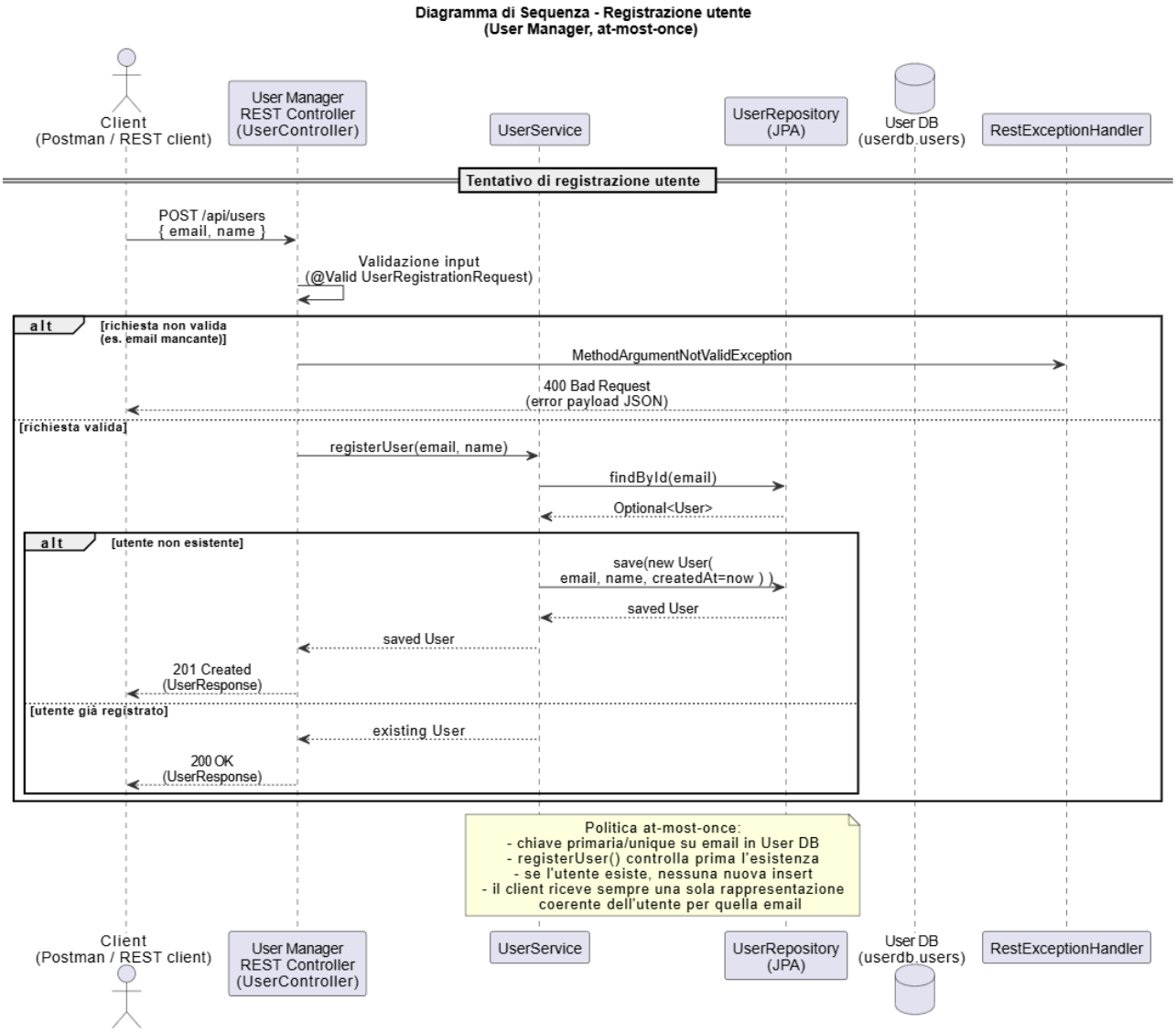
In questa sezione viene inserita l'immagine del "**diagramma di sequenza – registrazione utente (User Manager, politica at-most-once)**", che rappresenta gli scambi tra i seguenti partecipanti:

- **Client esterno** (ad esempio Postman o altra applicazione REST);
- **UserController** (layer REST del microservizio User Manager);
- **UserService** (layer applicativo del dominio utente);
- **UserRepository** (layer di accesso dati JPA);
- **User DB** (database logico degli utenti).

Il diagramma evidenzia le seguenti fasi principali:

1. Il *Client* invia una richiesta `POST /api/users` con il payload contenente `email` e `name`.
2. Lo `UserController` riceve la richiesta, applica la **validazione dei dati** in ingresso (annotazioni di validazione, vincoli sui campi) e, se la richiesta è valida, invoca il `UserService`.
3. Il `UserService` interroga il `UserRepository` (`existsById(email)` o equivalente) per verificare se l'utente con quella e-mail esista già nel *User DB*.
4. In caso di **utente già esistente**, il servizio evita la creazione di un nuovo record e restituisce al controller l'entità utente già presente, mantenendo la semantica di *idempotenza*.
5. In caso di **utente non esistente**, il `UserService` costruisce un nuovo oggetto `User`, imposta il campo `createdAt` e invia una richiesta di salvataggio al `UserRepository` (`save(user)`), che a sua volta persiste il record nel *User DB*.
6. Il `UserController` traduce l'esito dell'operazione in una risposta HTTP:
    - ad esempio, `201 Created` per una nuova creazione;
    - oppure `200 OK` per la ripetizione di una richiesta idempotente che trova un utente già registrato.

Eventuali errori di validazione o violazioni di regole di dominio sono intercettati da un componente di *exception handling* e restituiti al client con codici HTTP e payload strutturati.



6.1.2. ANALISI DEL FLUSSO E RESPONSABILITÀ DEI COMPONENTI

Il flusso di registrazione utente mette in evidenza:

- la **separazione chiara dei ruoli**:
    - il controller gestisce esclusivamente la traduzione tra protocollo HTTP e dominio applicativo;
    - il service incapsula la logica di business (idempotenza, controllo di esistenza, costruzione del modello utente);
    - il repository si occupa dell'accesso transazionale al database;
- l'adozione di una **semantica at-most-once**:
    - l'utente è identificato univocamente dalla propria e-mail;

    - la verifica di esistenza prima della creazione evita la generazione di record duplicati in caso di retry;
- la gestione strutturata degli errori:
    - richieste non valide (es. e-mail malformata) producono errori `400 Bad Request`;
    - eventuali condizioni di concorrenza vengono mitigate dal vincolo di chiave primaria sul campo `email`.

L'interazione, così modellata, costituisce il *building block* su cui si basano tutte le successive verifiche di esistenza dell'utente, incluse quelle eseguite via gRPC da altri microservizi.

6.2. INTERAZIONE PER LA REGITRAZIONE DEGLI INTERESSI

L'interazione per la **registrazione di un interesse utente–aeroporto** coinvolge in modo congiunto il microservizio *Data Collector* e il microservizio *User Manager*. Il diagramma di sequenza descrive come il Data Collector:

- riceve la richiesta del client;
- valida i parametri;
- invoca il servizio gRPC del User Manager per verificare l'esistenza dell'utente;
- interroga il proprio database per verificare l'esistenza dell'aeroporto e di eventuali interessi già registrati;
- persiste o meno un nuovo record nel *Data DB* sulla base di tali verifiche.

6.2.1. DIAGRAMMA DI SEQUENZA – REGISTRAZIONE INTERESSE AEROPORTO CON VALIDAZIONE VIA gRPC

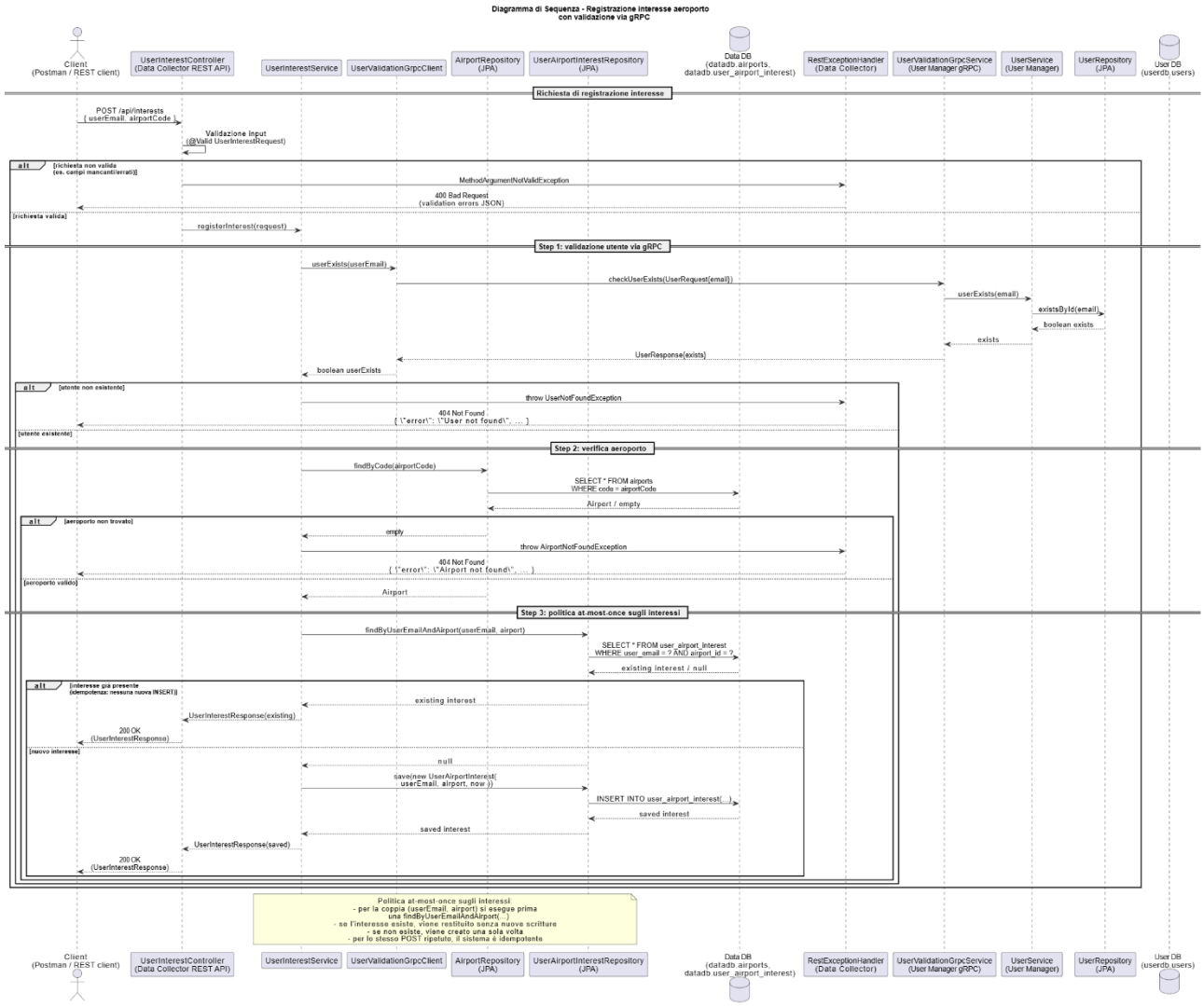
In questa sezione viene inserita l'immagine del "**diagramma di sequenza – Registrazione interesse aeroporto con validazione via gRPC**", che tipicamente include i seguenti partecipanti:

- **Client esterno**;
- **UserInterestController** (REST controller del Data Collector);
- **UserInterestService** (service del Data Collector responsabile degli interessi);
- **UserValidationGrpcClient** (client gRPC verso il User Manager);
- **UserValidationService (gRPC)** nel microservizio User Manager;
- **AirportRepository** e **UserAirportInterestRepository**;
- **Data DB**.

Le fasi principali mostrate nel diagramma sono:

1. Il *Client* invia una richiesta `POST /api/interests` con `userEmail` e `airportCode`.

2. Lo `UserInterestController` esegue la **validazione del payload** e inoltra la richiesta al `UserInterestService`.
3. Il `UserInterestService` usa il `UserValidationGrpcClient` per inviare una richiesta gRPC al `UserValidationService` del User Manager, passando l'e-mail dell'utente.
4. Il `UserValidationService` interroga il proprio `UserService/UserRepository`:
    - se l'utente **non esiste**, il servizio gRPC risponde con `exists = false`;
    - se l'utente **esiste**, il servizio gRPC risponde con `exists = true`.
5. In caso di risposta negativa (`exists = false`), il `UserInterestService` solleva un'eccezione di dominio (utente inesistente), che viene poi mappata in una risposta `404 Not Found` o equivalente verso il client. Nessuna operazione viene eseguita sul Data DB.
6. In caso di risposta positiva (`exists = true`), il `UserInterestService` interroga:
    - l'`AirportRepository` per verificare l'esistenza dell'aeroporto corrispondente al codice;
    - il `UserAirportInterestRepository` per verificare se esista già un record per la coppia (`userEmail, airport`).
7. Se l'aeroporto non è presente, viene sollevata un'eccezione (`AirportNotFound`) che produce un `404 Not Found`.
8. Se l'aeroporto è valido ma l'interesse esiste già, il servizio tratta la richiesta in modo *idempotente* e restituisce al client l'interesse esistente.
9. Se l'interesse non esiste, viene creato un nuovo `UserAirportInterest` e inviato al `UserAirportInterestRepository` per il salvataggio nel *Data DB*; l'operazione rispetta il vincolo di unicità (`user_email, airport_id`).



6.2.2. ANALISI DEL FLUSSO E RUOLO DELLA VALIDAZIONE gRPC

L'analisi del flusso mette in evidenza alcuni aspetti cruciali:

- il **ruolo centrale del User Manager** come fonte autorevole per l'esistenza dell'utente:
    - il Data Collector non accede mai al *User DB*;
    - tutte le verifiche avvengono via gRPC, nel rispetto dei confini tra domini;
- la **catena di validazioni** prima della persistenza:
    - verifica dell'utente via gRPC;
    - verifica dell'esistenza dell'aeroporto nel catalogo;
    - verifica dell'assenza di un interesse duplicato;
- l'implementazione di una semantica **idempotente**:
    - richieste ripetute con gli stessi parametri non producono interessi duplicati;

    - il comportamento dell'API è deterministico anche in presenza di retry;
- la **separazione delle responsabilità**:
    - il Data Collector è responsabile del dominio interessi e aeroporto;
    - il User Manager è responsabile del dominio utente e della validazione.

La combinazione tra controllo applicativo e vincoli a livello di database consente di ottenere una gestione robusta e coerente delle associazioni utente–aeroporto.

6.3. INTERAZIONE PER LA RACCOLTA PERIODICA DEI VOLI

L'interazione per la **raccolta periodica dei voli** è guidata da uno scheduler interno al Data Collector e comprende l'intero ciclo:

- identificazione degli aeroporti per cui esistono interessi attivi;
- interrogazione del servizio esterno OpenSky;
- trasformazione delle risposte in entità interne `FlightRecord`;
- persistenza dei dati nel *Data DB*.

6.3.1. DIAGRAMMA DI SEQUENZA – RACCOLTA PERIODICA DEI VOLI (SCHEDULER + OpenSky)

In questa sezione viene inserita l'immagine del "**diagramma di sequenza – raccolta periodica dei voli (Scheduler + OpenSky)**", che mostra gli scambi tra:

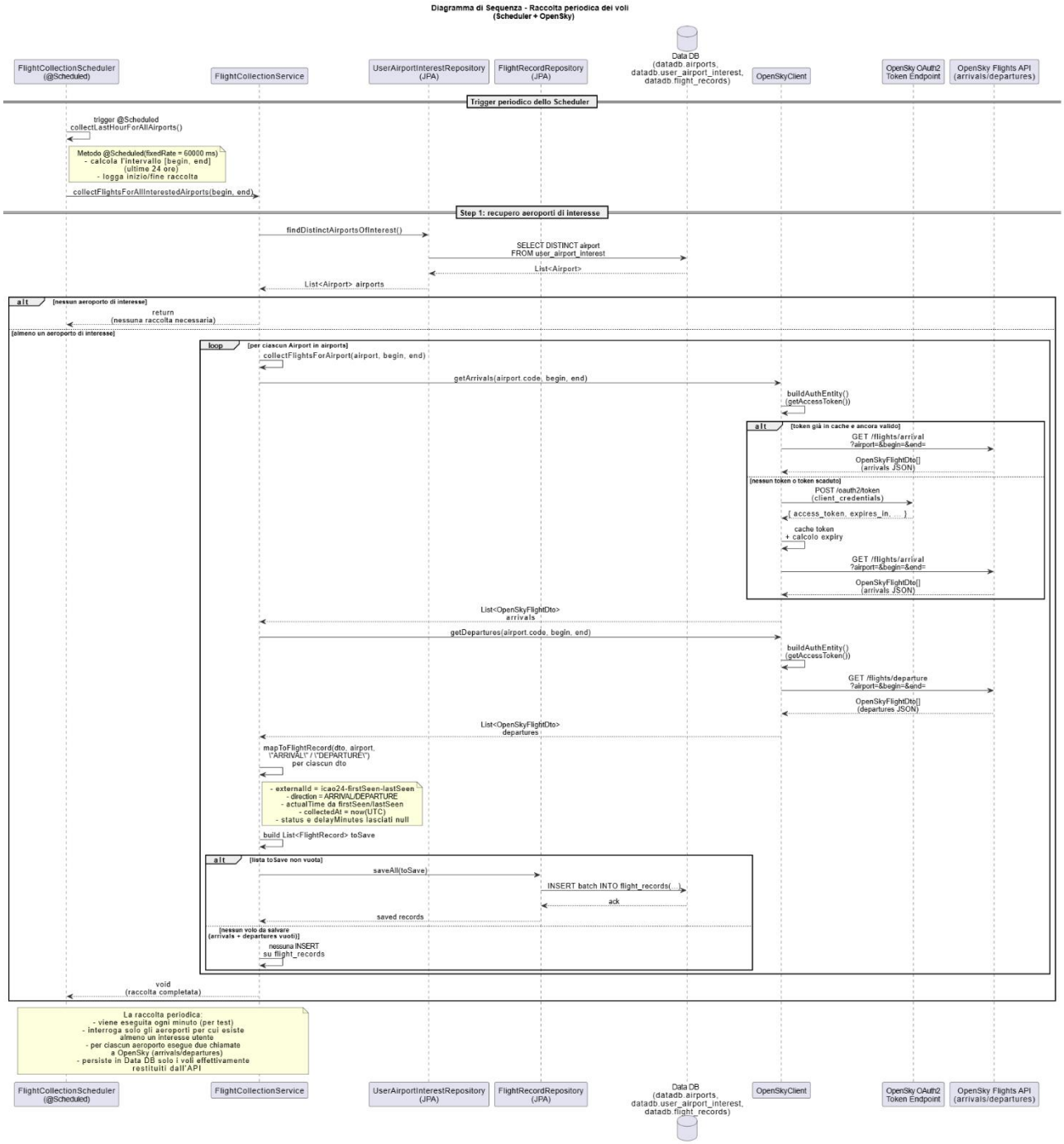
- **Scheduler** (es. `FlightCollectionScheduler`);
- **FlightCollectionService**;
- **UserAirportInterestRepository** (o un repository/service che fornisce gli aeroporti di interesse);
- **Airport**/relativo repository;
- **OpenSkyClient**;
- **Servizio esterno OpenSky Network**;
- **FlightRecordRepository**;
- **Data DB**.

Il diagramma tipicamente presenta il seguente flusso:

1. Lo **Scheduler** attiva periodicamente un metodo di `FlightCollectionService`, passando l'intervallo temporale [*begin*, *end*] per la raccolta.
2. Il `FlightCollectionService` interroga il `UserAirportInterestRepository` per ottenere la lista degli aeroporti distinti per cui esistono interessi (`findDistinctAirportsOfInterest` o equivalente).
3. Per ciascun aeroporto della lista, il `FlightCollectionService`:

- invoca `OpenSkyClient.getArrivals(airportCode, begin, end)` per ottenere i voli in arrivo;
- invoca `OpenSkyClient.getDepartures(airportCode, begin, end)` per ottenere i voli in partenza.

4. L'`OpenSkyClient`:
   - verifica la presenza e validità del token OAuth2, richiedendone uno nuovo se necessario;
   - effettua le chiamate HTTP verso gli endpoint di OpenSky;
   - riceve le risposte (lista di voli, nessun volo, eventuali errori) e le mappa in DTO interni.
5. Il `FlightCollectionService` trasforma i DTO di OpenSky in enti `FlightRecord`:
   - associa ciascun record all'entità `Airport` corrispondente;
   - imposta la `direction` (*ARRIVAL* o *DEPARTURE*);
   - valorizza `actualTime`, `externalFlightId`, `flightNumber`, `collectedAt` e gli altri campi rilevanti.
6. Il `FlightRecordRepository` persiste i nuovi record nel *Data DB*, preferibilmente in modalità batch (`saveAll`).
7. In caso di errori nella comunicazione con OpenSky (es. rete non disponibile, risposte inattese), il `FlightCollectionService` registra log specifici; eventuali strategie di retry o fallback possono essere gestite a questo livello.



6.3.2. ANALISI DEL FLUSSO DI INGESTIONE DATI E PERSISTENZA NEL DATA DB

L'analisi dell'interazione evidenzia:

- un flusso di tipo **pull** verso la sorgente esterna:
  - il sistema decide *quando* interrogare OpenSky, sulla base dello scheduler;
  - la frequenza di raccolta è configurabile e può essere adattata alle esigenze del contesto operativo;

- l'uso di una **selezione guidata dagli interessi**:
    - vengono interrogati solo gli aeroporti per i quali esistono interessi utente;
    - si evita di raccogliere dati superflui per aeroporti che non sono di alcun interesse per il sistema;
- la distinzione tra **logica di integrazione** (gestita da `OpenSkyClient`) e **logica di dominio** (gestita da `FlightCollectionService`):
    - la prima si occupa di token, HTTP, mapping JSON;
    - la seconda si occupa di trasformare i dati esterni in `FlightRecord` e di istruirne la persistenza;
- l'accumulo di **dati storici** nel *Data DB*:
    - ogni esecuzione della raccolta inserisce nuovi record;
    - le query successive (per ultimo volo, media, elenco) operano su questo storico, filtrando per tempi, direzione e aeroporto;
- la gestione esplicita dei casi di **assenza di dati**:
    - una risposta vuota da OpenSky non è trattata come errore applicativo;
    - rappresenta semplicemente l'assenza di voli nella finestra considerata.

L'interazione definisce una pipeline di ingestione che mantiene il database allineato con lo stato del traffico aereo per gli aeroporti di interesse, compatibilmente con la finestra temporale e la frequenza di raccolta impostate.

6.4. INTERAZIONE PER L'INTERROGAZIONE DEI DATI DI VOLO

L'interazione per l'**interrogazione dei dati di volo** descrive i flussi scatenati dalle richieste del client verso il Data Collector per ottenere:

- l'ultimo volo registrato per aeroporto e direzione;
- la media dei voli su una finestra temporale in giorni;
- l'elenco dei voli registrati in un intervallo temporale specifico.

Il diagramma di sequenza rappresenta un pattern comune, declinato nelle diverse varianti di endpoint.

6.4.1. DIAGRAMMA DI SEQUENZA – INTERROGAZIONE DEI DATI DI VOLO (CLIENT → DATA COLLECTOR)

In questa sezione viene inserita l'immagine del "**diagramma di sequenza – Interrogazione dei dati di volo (Client → Data Collector)**", che mostra gli scambi tra:

- **Client esterno**;
- **FlightQueryController**;
- **FlightQueryService**;

- **FlightRecordRepository** (ed eventualmente `AirportRepository`);
- **Data DB**.

Il diagramma illustra un flusso generico, adattabile ai tre casi principali:

1. **Richiesta dell'ultimo volo (**`GET /api/flights/last`**)**
    - Il client invia una richiesta con parametri quali `airportCode` e `direction`.
    - Il `FlightQueryController` valida i parametri e invoca il `FlightQueryService`.
    - Il servizio:
        - risolve l'aeroporto a partire da `airportCode` (`AirportRepository.findByCode`);
        - esegue una query sul FlightRecordRepository (es. findFirstByAirportAndDirectionOrderByActualTimeDesc);
    - Se viene trovato un record, lo restituisce come risposta JSON;
    - Se nessun volo è presente per quei criteri, viene sollevata un'eccezione di dominio (es. `FlightNotFound`) mappata in `404 Not Found`.
2. **Richiesta della media dei voli (**`GET /api/flights/average`**)**
    - Il client fornisce parametri come `airportCode`, `direction` e `days`.
    - Il `FlightQueryController` controlla che `days` sia un valore positivo e calcola l'intervallo temporale [*now* - *days*, *now*].
    - Il `FlightQueryService`:
        - risolve l'aeroporto;
        - usa il `FlightRecordRepository` per **contare** i voli con `actualTime` nell'intervallo (`countByAirportAndDirectionAndActualTimeBetween`).
    - Il servizio calcola:
        - il numero totale di voli;
        - la media giornaliera (totale / days);
    - I valori vengono restituiti al client in un oggetto JSON contenente i campi di riepilogo.
3. **Richiesta dell'elenco dei voli (**`GET /api/flights`**)**
    - Il client può specificare parametri `airportCode`, `direction`, `from`, `to`; in assenza di parametri temporali, viene utilizzata una finestra di default.
    - Dopo la validazione dei parametri (inclusa la verifica che `from <= to`), il `FlightQueryService`:

        - risolve l’aeroporto;
        - esegue una query per ottenere tutti i `FlightRecord` corrispondenti (`findByAirportAndDirectionAndActualTimeBetweenOrderByActualTimeDesc`).
    - I record vengono mappati in DTO di risposta contenenti informazioni come flightNumber, actualTime, status, delayMinutes e restituiti in una lista JSON.

In tutti i casi, eventuali errori sui parametri (direzione non ammessa, intervalli non validi) sono intercettati a livello di controller o service e tradotti in risposte `400 Bad Request` con dettagli espliciti.

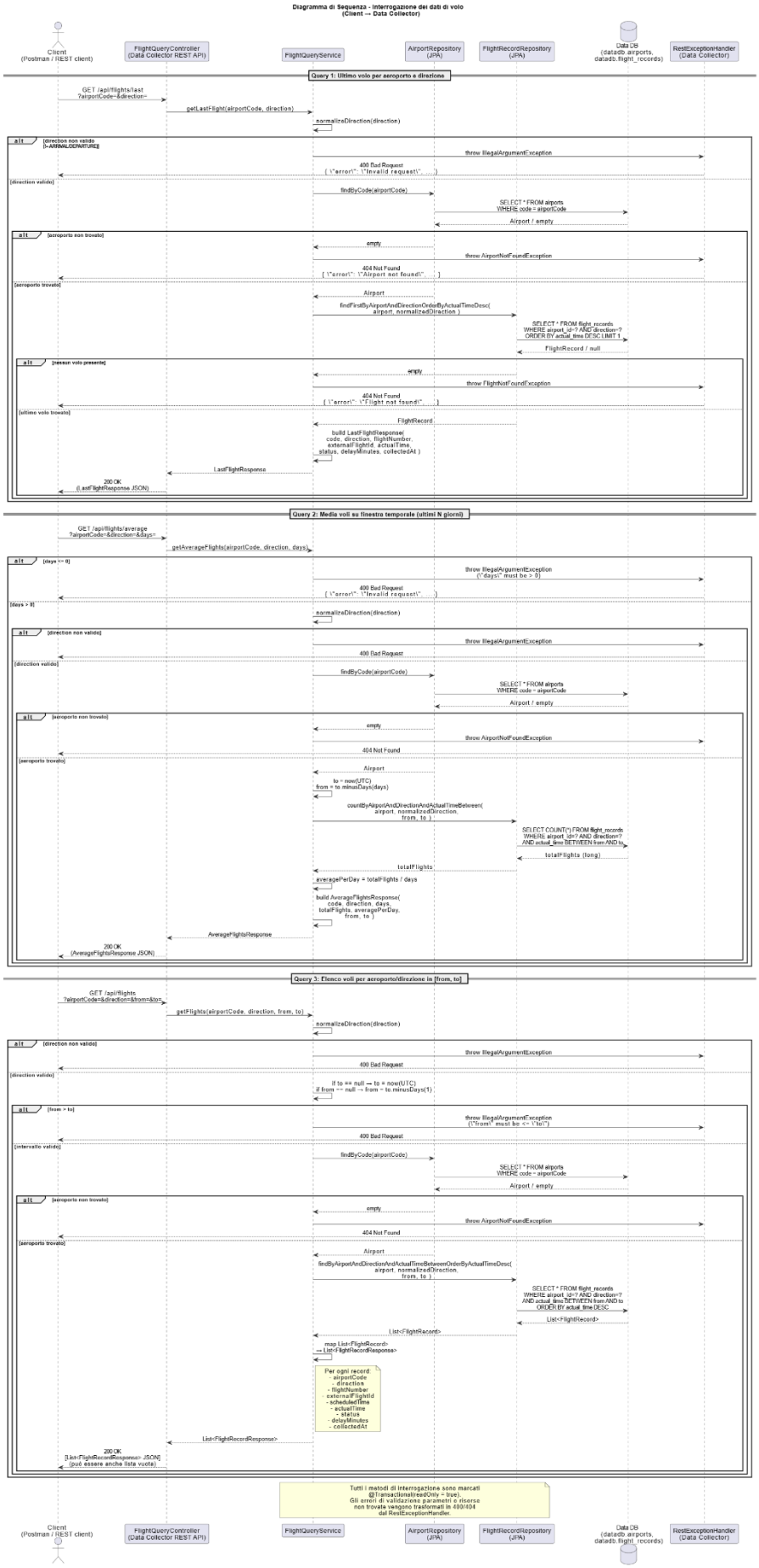


Diagramma di Sequenza - Interrogazione dei dati di volo
(Client → Data Collector)
Client
(Postman / REST client)
FlightQueryController
(Data Collector REST API)
FlightQueryService
AirportRepository
(JPA)
FlightRecordRepository
(JPA)
Data DB
(datadb.airports,
datadb.flight_records)
RestExceptionHandler
(Data Collector)
Query 1: Ultimo volo per aeroporto e direzione
GET /api/flights/last
?airportCode=&direction=
getLastFlight(airportCode, direction)
normalizeDirection(direction)
alt
[direction non valido
(!= ARRIVAL/DEPARTURE)]
throw IllegalArgumentException
400 Bad Request
{ \"error\": \"Invalid request\" }
[direction valida]
findByCode(airportCode)
SELECT * FROM airports
WHERE code = airportCode
Airport / empty
alt
[aeroporto non trovato]
empty
throw AirportNotFoundException
404 Not Found
{ \"error\": \"Airport not found\" }
[aeroporto trovato]
Airport
findFirstByAirportAndDirectionOrderByActualTimeDesc(
airport, normalizedDirection)
SELECT * FROM flight_records
WHERE airport_id=? AND direction=?
ORDER BY actual_time DESC LIMIT 1
FlightRecord / null
alt
[nessun volo presente]
empty
throw FlightNotFoundException
404 Not Found
{ \"error\": \"Flight not found\" }
[ultimo volo trovato]
FlightRecord
build LastFlightResponse(
code, direction, flightNumber,
externalFlightId, actualTime,
status, delayMinutes, collectedAt)
LastFlightResponse
200 OK
(LastFlightResponse JSON)
Query 2: Media voli su finestra temporale (ultimi N giorni)
GET /api/flights/average
?airportCode=&direction=&days=
getAverageFlights(airportCode, direction, days)
alt
[days <= 0]
throw IllegalArgumentException
(\"days\" must be > 0)
400 Bad Request
{ \"error\": \"Invalid request\" }
[days > 0]
normalizeDirection(direction)
alt
[direction non valida]
throw IllegalArgumentException
400 Bad Request
[direction valida]
findByCode(airportCode)
SELECT * FROM airports
WHERE code = airportCode
Airport / empty
alt
[aeroporto non trovato]
empty
throw AirportNotFoundException
404 Not Found
[aeroporto trovato]
Airport
to = now(UTC)
from = to.minusDays(days)
countByAirportAndDirectionAndActualTimeBetween(
airport, normalizedDirection,
from, to)
SELECT COUNT(*) FROM flight_records
WHERE airport_id=? AND direction=?
AND actual_time BETWEEN from AND to
totalFlights (long)
totalFlights
averagePerDay = totalFlights / days
build AverageFlightsResponse(
code, direction, days,
totalFlights, averagePerDay,
from, to)
AverageFlightsResponse
200 OK
(AverageFlightsResponse JSON)
Query 3: Elenco voli per aeroporto/direzione in [from, to]
GET /api/flights
?airportCode=&direction=&from=&to=
getFlights(airportCode, direction, from, to)
normalizeDirection(direction)
alt
[direction non valida]
throw IllegalArgumentException
400 Bad Request
[direction valida]
if to == null → to = now(UTC)
if from == null → from = to.minusDays(1)
alt
[from > to]
throw IllegalArgumentException
(\"from\" must be <= \"to\")
400 Bad Request
[intervallo valido]
findByCode(airportCode)
SELECT * FROM airports
WHERE code = airportCode
Airport / empty
alt
[aeroporto non trovato]
empty
throw AirportNotFoundException
404 Not Found
[aeroporto trovato]
Airport
findByAirportAndDirectionAndActualTimeBetweenOrderByActualTimeDesc(
airport, normalizedDirection,
from, to)
SELECT * FROM flight_records
WHERE airport_id=? AND direction=?
AND actual_time BETWEEN from AND to
ORDER BY actual_time DESC
List<FlightRecord>
List<FlightRacord>
map List<FlightRecord>
→ List<FlightRacordResponse>
Per ogni record:
- airportCode
- direction
- flightNumber
- externalFlightId
- scheduledTime
- actualTime
- status
- delayMinutes
- collectedAt
List<FlightRecordResponse>
200 OK
[List<FlightRecordResponse> JSON]
(può essere anche lista vuota)
Tutti i metodi di interrogazione sono marcati
@Transactional(readOnly = true).
Gli errori di validazione parametri e risorse
non trovate vengono trasformati in 400/404
dal RestExceptionHandler.

6.4.2. ANALISI DEGLI ENDPOINT DI LETTURA E DEI CASI DI ERRORE

L'analisi degli endpoint di lettura mette in evidenza:

- un pattern comune di **validazione → risoluzione aeroporto → query → mapping risposta**:
    - la validazione previene l'esecuzione di query su input non coerenti;
    - la risoluzione dell'aeroporto funge da primo filtro: se l'aeroporto non esiste, la richiesta viene respinta con un errore dedicato;
    - le query sul `FlightRecordRepository` sono ottimizzate per tipo di richiesta (max per ultimo volo, count per media, range per elenco);
- una gestione coerente dei **casi di assenza di dati**:
    - assenza di voli in un intervallo può portare a:
        - restituzione di una lista vuota (per l'elenco);
        - sollevamento di un'eccezione `FlightNotFound` per l'ultimo volo, che segnala al client che non esistono ancora dati per quella combinazione di parametri;
- la distinzione tra **errori di dominio** ed **errori di input**:
    - input formalmente errati (date invertite, giorni non positivi, direzioni non valide) sono trattati come `400 Bad Request`;
    - risorse mancanti (aeroporto non presente, nessun volo trovato) sono trattate come `404 Not Found`.

Questa impostazione rende le API di lettura prevedibili per il client e consente una gestione chiara dei casi limite e di errore.

7. API ESPOSTE E CONTRATTI DI SERVIZIO

7.1. CONVENZIONI GENERALI

L'esposizione delle API segue principi di **chiarezza**, **coerenza semantica** e **separazione dei domini**.
Le interfacce REST costituiscono il *boundary* verso il client esterno, mentre l'interfaccia gRPC costituisce il contratto di comunicazione *service-to-service* tra i microservizi.

Le convenzioni adottate riguardano:

- strutturazione dei path (`/api/users`, `/api/interests`, `/api/flights`);
- uso consistente dei metodi HTTP;
- formato dei dati (JSON per le API REST, Protobuf per gRPC);
- semantica dei codici HTTP;
- modello standardizzato di risposta di errore.

7.1.1. BASE URL, FORMATO DATI (JSON), CONVENZIONI SUI CODICI HTTP

Le API REST sono esposte su due microservizi distinti:

- **User Manager Service**
  - Base URL tipica: `http://<host-user-manager>:<port>/api/users`
  - Esempio in ambiente locale: `http://localhost:8080/api/users`
- **Data Collector Service**
  - Base URL tipica per interessi: `http://<host-data-collector>:<port>/api/interests`
  - Base URL tipica per voli: `http://<host-data-collector>:<port>/api/flights`
  - Esempio in ambiente locale: `http://localhost:8081/api/interests`, `http://localhost:8081/api/flights`

Formato dati REST

- **Request body**: JSON per tutte le operazioni che richiedono un payload (principalmente `POST`).
- **Response body**: JSON per tutte le risposte di successo (200, 201, 4xx gestiti), con strutture coerenti per ciascuna famiglia di endpoint.
- **Charset**: UTF-8.

Codici HTTP

Le principali convenzioni sui codici di stato sono:

- `200 OK`
  Operazione eseguita con successo, tipicamente per:
  - lettura di risorse (`GET`);
  - operazioni idempotenti di creazione quando la risorsa era già presente (registrazione utente/ interesse già esistente).
- `201 Created`
  Creazione di una nuova risorsa (utente, interesse) quando l'operazione produce effettivamente un nuovo record nel database.
- `204 No Content`
  Operazioni di cancellazione eseguite con successo (`DELETE`) che non richiedono un payload di risposta.
- `400 Bad Request`
  Errori di **validazione input** o parametri non coerenti:
  - campi mancanti o non validi nel body;
  - valori non ammessi per parametri query (direzione, intervalli temporali, numero di giorni).
- `404 Not Found`
  Risorse non trovate:
  - utente inesistente;
  - aeroporto assente dal catalogo;
  - nessun volo trovato quando è richiesto un singolo record (ultimo volo).
- `409 Conflict` (eventuale)
  Conflitti di stato, ad esempio tentativi di creazione in violazione di vincoli, se non assorbiti dalla logica di idempotenza.
- `5xx`
  Errori interni non gestiti esplicitamente o malfunzionamenti infrastrutturali/di integrazione, da evitare per quanto possibile tramite gestione controllata delle eccezioni.

7.1.2. MODELLO DI ERRORE E STRUTTURA DELLE RISPOSTE DI ERRORE

Gli errori applicativi sono gestiti da un **handler centralizzato** che converte eccezioni di dominio e di validazione in una risposta JSON strutturata.
Un possibile modello di errore è il seguente:

```
{
  "timestamp": "2025-11-30T15:42:10Z",
  "status": 404,
```

```
  "error": "Not Found",
  "errorCode": "USER_NOT_FOUND",
  "message": "User with email 'alice@example.com' not found",
  "path": "/api/users/alice@example.com"
}
```

Campi principali:

- `timestamp` – istante in cui l'errore è stato generato;
- `status` – codice HTTP numerico;
- `error` – breve descrizione testuale associata al codice HTTP;
- `errorCode` – codice di errore applicativo (es. `USER_NOT_FOUND`, `AIRPORT_NOT_FOUND`, `INVALID_PARAMETERS`);
- `message` – descrizione leggibile dell'errore, pensata per l'operatore o per il client;
- `path` – path della richiesta che ha generato l'errore.

Per gli errori di **validazione del payload** (es. campi mancanti o formati non validi), il payload può includere un campo aggiuntivo:

```
{
  "timestamp": "...",
  "status": 400,
  "error": "Bad Request",
  "errorCode": "VALIDATION_ERROR",
  "message": "One or more fields are invalid",
  "fieldErrors": [
    { "field": "email", "message": "must be a valid email address" },
    { "field": "days", "message": "must be greater than 0" }
  ],
  "path": "/api/flights/average"
}
```

Questo modello consente di gestire in modo coerente:

- errori di dominio (risorse non trovate, stato non coerente);
- errori di validazione;
- errori di formato/parametri.

7.2. API DEL MICROSERVIZIO USER MANAGER

Il microservizio **User Manager** espone un set di API REST per la gestione del ciclo di vita degli utenti. Tutte le operazioni utilizzano l'e-mail come identificatore univoco dell'utente.

7.2.1. POST /api/users – REGISTRAZIONE DI UN NUOVO UTENTE

Descrizione
Registra un nuovo utente a partire da e-mail e nome.

L’operazione è **idempotente** rispetto all’e-mail: più richieste con la stessa e-mail non creano duplicati.

- **Metodo**: `POST`
- **URL**: `/api/users`
- **Body (JSON)**, esempio:

```
{
  "email": "alice@example.com",
  "name": "Alice Doe"
}
```

- **Vincoli principali**:
  - `email` obbligatoria, formato e-mail valido;
  - `name` obbligatorio, lunghezza entro limiti definiti.

Risposta – caso utente creato

- **Status**: `201 Created`
- **Body (JSON)**:

```
{
  "email": "alice@example.com",
  "name": "Alice Doe",
  "createdAt": "2025-11-30T15:40:00Z"
}
```

Risposta – caso utente già esistente (idempotenza)

- **Status**: `200 OK`
- **Body (JSON)**: rappresentazione dell’utente già presente, con gli stessi campi del caso precedente.

Errori principali

- `400 Bad Request` – payload non valido (e-mail malformata, campi mancanti).
- `409 Conflict` – eventuale conflitto non assorbito dall’idempotenza (ad esempio se, per motivi esterni, il vincolo di unicità venisse violato).

7.2.2. GET /api/users/{email} – RECUPERO DI UN UTENTE

Descrizione
Recupera i dati di un utente a partire dall’indirizzo e-mail.

- **Metodo**: `GET`
- **URL**: `/api/users/{email}`

- **Path variable**:
  - `email` – indirizzo e-mail dell'utente.

Risposta – successo

- **Status**: `200 OK`
- **Body (JSON)**:

```
{
  "email": "alice@example.com",
  "name": "Alice Doe",
  "createdAt": "2025-11-30T15:40:00Z"
}
```

Errori principali

- `404 Not Found` – se non esiste alcun utente con l'e-mail specificata.
- `400 Bad Request` – se il formato dell'e-mail nel path non è valido.

7.2.3. DELETE /api/users/{email} – CANCELLAZIONE DI UN UTENTE

Descrizione
Cancella l'utente identificato dall'e-mail.
L'operazione rimuove il record dal *User DB*; eventuali interessi associati nel *Data DB* devono essere gestiti applicativamente o mantenuti come *dati orfani* a seconda delle scelte progettuali adottate.

- **Metodo**: `DELETE`
- **URL**: `/api/users/{email}`

Risposta – successo

- **Status**: `204 No Content`
- **Body**: vuoto.

Errori principali

- `404 Not Found` – se l'utente non esiste.
- `400 Bad Request` – se il formato dell'e-mail è invalido.

7.2.4. API DEL MICROSERVIZIO DATA COLLECTOR – GESTIONE INTERESSI

Le API di gestione degli interessi permettono di configurare la relazione tra utenti (identificati da e-mail) e aeroporti (identificati da codice). La creazione di un interesse è subordinata alla validazione dell'utente via gRPC presso il User Manager.

7.2.5. POST /api/interests – REGISTRAZIONE DI UN INTERESSE

Descrizione
Registra l'interesse di un utente verso un aeroporto.
L'operazione è **idempotente** rispetto alla coppia *(userEmail, airportCode)*.

- **Metodo**: `POST`
- **URL**: `/api/interests`
- **Body (JSON)**, esempio:

```
{
  "userEmail": "alice@example.com",
  "airportCode": "LIRF"
}
```

Flusso logico

1. Validazione sintattica del payload.
2. Validazione dell'utente via gRPC verso il User Manager.
3. Verifica dell'esistenza dell'aeroporto nel catalogo.
4. Verifica dell'esistenza di un interesse con la stessa coppia *(userEmail, airport)*.
5. Creazione del record se non esistente, altrimenti risposta idempotente.

Risposta – interesse creato

- **Status**: `201 Created`
- **Body (JSON)**:

```
{
  "id": 123,
  "userEmail": "alice@example.com",
  "airportCode": "LIRF",
  "createdAt": "2025-11-30T15:45:00Z"
}
```

Risposta – interesse già esistente

- **Status**: `200 OK`
- **Body (JSON)**: rappresentazione dell'interesse già presente, con gli stessi campi.

Errori principali

- `404 Not Found` – utente inesistente (validazione gRPC negativa) oppure aeroporto non presente nel catalogo.
- `400 Bad Request` – parametri non validi (e-mail malformata, codice aeroporto vuoto o non conforme).
- `409 Conflict` – eventuale violazione di vincolo di unicità non assorbita dalla logica applicativa.

7.2.6. DELETE /api/interests – RIMOZIONE DI UN INTERESSE

Descrizione
Rimuove l'interesse di un utente verso un aeroporto.

La semantica può essere *idempotente*: la rimozione di un interesse inesistente può essere considerata un'operazione riuscita o segnalata con `404`, in funzione delle scelte implementative.

- **Metodo**: `DELETE`
- **URL**: `/api/interests`
- **Query parameters** (o body JSON, a seconda dell'implementazione; qui si assume query):
  - `userEmail` – e-mail dell'utente;
  - `airportCode` – codice dell'aeroporto.
- Esempio di richiesta:
  `/api/interests?userEmail=alice@example.com&airportCode=LIRF`

Risposta – successo

- **Status**: `204 No Content`
- **Body**: vuoto.

Errori principali

- `404 Not Found` – se non esiste alcun interesse per la coppia *(utente, aeroporto)*, nel caso in cui si scelga di segnalare esplicitamente l'assenza.
- `400 Bad Request` – parametri mancanti o non validi.

7.2.7. GET /api/interests – ELENCO INTERESSI PER UTENTE

Descrizione
Restituisce l'elenco degli aeroporti di interesse associati a un determinato utente.

- **Metodo**: `GET`
- **URL**: `/api/interests`
- **Query parameters**:
  - `userEmail` – e-mail dell'utente (obbligatorio).
- Esempio di richiesta:
  `/api/interests?userEmail=alice@example.com`

Risposta – successo

- **Status**: `200 OK`
- **Body (JSON)**, esempio:

```
[
  {
    "id": 123,
    "userEmail": "alice@example.com",
```

```
    "airportCode": "LIRF",
    "createdAt": "2025-11-30T15:45:00Z"
  },
  {
    "id": 124,
    "userEmail": "alice@example.com",
    "airportCode": "EGLL",
    "createdAt": "2025-11-30T16:10:00Z"
  }
]
```

Se l'utente non ha interessi registrati, il servizio può restituire una **lista vuota** con `200 OK`.

Errori principali

- `404 Not Found` – opzionale, se si decide di segnalare esplicitamente il caso di utente inesistente.
- `400 Bad Request` – e-mail non valida o mancante.

7.3. API DEL MICROSERVIZIO DATA COLLECTOR – INTERROGAZIONE VOLI

Le API di interrogazione voli espongono le informazioni aggregate e di dettaglio ottenute dalla raccolta periodica su OpenSky, filtrabili per aeroporto, direzione e intervalli temporali.

7.3.1. GET /api/flights/last – ULTIMO VOLO PER AEROPORTO/DIREZIONE

Descrizione
Restituisce l'**ultimo volo registrato** per un determinato aeroporto e una specifica direzione (*ARRIVAL* o *DEPARTURE*), in base al campo `actualTime`.

- **Metodo**: `GET`
- **URL**: `/api/flights/last`
- **Query parameters**:
  - `airportCode` – codice dell'aeroporto (obbligatorio);
  - `direction` – direzione del volo, valori ammessi ad esempio: `ARRIVAL`, `DEPARTURE` (obbligatorio).
- Esempio di richiesta:
  `/api/flights/last?airportCode=LIRF&direction=ARRIVAL`

Risposta – successo

- **Status**: `200 OK`
- **Body (JSON)**, esempio:

```
{
  "airportCode": "LIRF",
  "direction": "ARRIVAL",
  "flightNumber": "AZ123",
```

```
  "externalFlightId": "icao24-firstSeen-lastSeen",
  "actualTime": "2025-11-30T15:30:00Z",
  "scheduledTime": "2025-11-30T15:20:00Z",
  "status": "LANDED",
  "delayMinutes": 10,
  "collectedAt": "2025-11-30T15:35:00Z"
}
```

Errori principali

- `404 Not Found` – nessun volo registrato per quella combinazione di aeroporto e direzione.
- `404 Not Found` – aeroporto non presente nel catalogo.
- `400 Bad Request` – direzione non ammessa o parametri mancanti/non validi.

7.3.2. GET /api/flights/average – MEDIA VOLI SU FINESTRA TEMPORALE

Descrizione
Restituisce la **media giornaliera** dei voli per un determinato aeroporto e una direzione, calcolata su una finestra temporale di *N* giorni passati.

- **Metodo**: `GET`
- **URL**: `/api/flights/average`
- **Query parameters**:
  - `airportCode` – codice dell'aeroporto (obbligatorio);
  - `direction` – `ARRIVAL` o `DEPARTURE` (obbligatorio);
  - `days` – numero di giorni da considerare a ritroso rispetto all'istante corrente (obbligatorio, intero > 0).
- Esempio:
  `/api/flights/average?airportCode=LIRF&direction=DEPARTURE&days=7`

Elaborazione

- Il servizio calcola l'intervallo [*now* − *days*, *now*];
- esegue una query di conteggio dei flight records dell'aeroporto/direzione in tale intervallo;
- calcola la media giornaliera come `totalFlights / days`.

Risposta – successo

- **Status**: `200 OK`
- **Body (JSON)**, esempio:

```
{
  "airportCode": "LIRF",
```

```
  "direction": "DEPARTURE",
  "days": 7,
  "totalFlights": 210,
  "averagePerDay": 30.0,
  "from": "2025-11-23T00:00:00Z",
  "to": "2025-11-30T00:00:00Z"
}
```

Errori principali

- `400 Bad Request` – `days <= 0`, direzione non valida, parametri mancanti, intervallo mal formato.
- `404 Not Found` – aeroporto non presente nel catalogo.

7.3.3. GET /api/flights – ELENCO VOLI IN UN INTERVALLO TEMPORALE

Descrizione
Restituisce l'**elenco dei voli** per un determinato aeroporto e direzione, filtrato su un intervallo temporale [*from*, *to*] rispetto al campo `actualTime`.
In assenza di parametri temporali, può essere utilizzata una finestra di default (ad esempio ultime N ore).

- **Metodo**: `GET`
- **URL**: `/api/flights`
- **Query parameters**:
  - `airportCode` – codice dell'aeroporto (obbligatorio);
  - `direction` – `ARRIVAL` o `DEPARTURE` (obbligatorio);
  - `from` – istante di inizio intervallo, in formato ISO-8601 (opzionale);
  - `to` – istante di fine intervallo, in formato ISO-8601 (opzionale).
- Esempio:
  `/api/flights?airportCode=LIRF&direction=ARRIVAL&from=2025-11-29T00:00:00Z&to=2025-11-30T00:00:00Z`

Risposta – successo

- **Status**: `200 OK`
- **Body (JSON)**, esempio:

```
[
  {
    "airportCode": "LIRF",
    "direction": "ARRIVAL",
    "flightNumber": "AZ101",
    "externalFlightId": "icao24-1",
    "actualTime": "2025-11-29T08:15:00Z",
```

```
    "scheduledTime": "2025-11-29T08:05:00Z",
    "status": "LANDED",
    "delayMinutes": 10,
    "collectedAt": "2025-11-29T08:20:00Z"
  },
  {
    "airportCode": "LIRF",
    "direction": "ARRIVAL",
    "flightNumber": "AZ205",
    "externalFlightId": "icao24-2",
    "actualTime": "2025-11-29T09:45:00Z",
    "scheduledTime": "2025-11-29T09:30:00Z",
    "status": "LANDED",
    "delayMinutes": 15,
    "collectedAt": "2025-11-29T09:50:00Z"
  }
]
```


Se non sono presenti voli nell'intervallo, viene restituita una **lista vuota** con `200 OK`.

Errori principali

- `400 Bad Request` – parametri temporali non validi (formato errato, `from > to`), direzione non ammessa.
- `404 Not Found` – aeroporto non presente nel catalogo.

7.4. INTERFACCIA gRPC DI VALIDAZIONE UTENTE

L'interfaccia gRPC di validazione utente fornisce un **contratto di servizio interno** tra il microservizio Data Collector e il microservizio User Manager.
Questo canale è dedicato esclusivamente alla verifica dell'esistenza di utenti a partire dall'e-mail e non ha esposizione diretta verso client esterni.

7.4.1. SERVIZIO UserValidationService E MESSAGGI (request/response)

L'interfaccia è definita in un file `.proto` che descrive il servizio `UserValidationService` e i relativi messaggi. Un esempio di definizione è il seguente (a livello concettuale):

```
service UserValidationService {
  rpc CheckUserExists (UserValidationRequest) returns
(UserValidationResponse);
}

message UserValidationRequest {
  string email = 1;
}

message UserValidationResponse {
  bool exists = 1;
}
```

Elementi principali:

- **Servizio**: `UserValidationService`
  Espone il metodo `CheckUserExists`.
- **Request**: `UserValidationRequest`
  - Campo `email` – indirizzo e-mail dell'utente da verificare.
- **Response**: `UserValidationResponse`
  - Campo `exists` – valore booleano che indica se l'utente è presente o meno nel dominio del User Manager.

Sulla base di questa definizione, vengono generati automaticamente:

- lo *stub server* per il User Manager (implementato in `UserValidationGrpcService` o equivalente);
- lo *stub client* per il Data Collector (`UserValidationGrpcClient`).

7.4.2. CONTRATTO E SEMANTICA DELLA CHIAMATA DI VALIDAZIONE

La semantica di `CheckUserExists` è estremamente **semplice e deterministica**, con l'obiettivo di ridurre al minimo l'accoppiamento tra microservizi:

- **Precondizioni**:
  - il chiamante (Data Collector) deve fornire un indirizzo e-mail non vuoto e formalmente valido;
  - la chiamata viene effettuata in modalità sincrona, tipicamente in occasione della registrazione di un interesse.
- **Comportamento atteso**:
  - se nel *User DB* esiste un record utente con quella e-mail:
    - il servizio restituisce `exists = true`;
  - se nessun utente corrisponde a quell'e-mail:
    - il servizio restituisce `exists = false`.
- **Errori gRPC**:
  - errori legati al trasporto (connessione non disponibile, timeout) vengono segnalati come *status* gRPC (`UNAVAILABLE`, `DEADLINE_EXCEEDED`, ecc.);
  - il Data Collector deve gestire tali errori in modo esplicito, tipicamente mappandoli in eccezioni applicative e restituendo al client REST un errore 5xx o un messaggio che indichi l'indisponibilità temporanea del servizio di validazione.
- **Idempotenza e coerenza**:
  - la chiamata è **side-effect free**: non modifica lo stato del User Manager;

- è intrinsecamente idempotente: chiamate ripetute con la stessa e-mail producono sempre la stessa risposta fintanto che lo stato del dominio utente non cambia.

Questo contratto consente al Data Collector di prendere decisioni coerenti (creare o meno un interesse, rifiutare richieste con utenti inesistenti) senza dover conoscere o accedere direttamente allo schema dati del User Manager, preservando l'indipendenza dei domini applicativi.

8. ASPETTI NON FUNZIONALI E QUALITA' DEL SOFTWARE

8.1. GESTIONE DELLE ECCEZIONI E MAPPING VERSO HTTP (RestExceptionHandler)

La gestione delle eccezioni è centralizzata tramite un componente dedicato di tipo `RestExceptionHandler`, presente in entrambi i microservizi. L'obiettivo è garantire che:

- le eccezioni tecniche e di dominio non trapelino verso l'esterno come stacktrace o messaggi opachi;
- ogni errore venga rappresentato tramite una risposta HTTP **strutturata**, coerente e facilmente interpretabile dal client;
- la semantica dei codici HTTP sia uniforme in tutti i punti di esposizione delle API REST.

La strategia adottata prevede:

- **Gestione delle eccezioni di validazione**
  Le eccezioni generate a seguito di violazioni dei vincoli sui DTO in ingresso (es. `MethodArgumentNotValidException`, `ConstraintViolationException`) vengono intercettate e trasformate in risposte con:
  - codice `400 Bad Request`;
  - body JSON contenente:
    - un codice di errore applicativo (es. `VALIDATION_ERROR`);
    - un messaggio generale;
    - una lista di errori di campo (`fieldErrors`) con coppie *campo–messaggio* che specificano la natura della violazione.
- **Gestione delle eccezioni di dominio**
  Eccezioni come `UserNotFoundException`, `AirportNotFoundException`, `FlightNotFoundException`, `InterestNotFoundException` (o equivalenti) rappresentano condizioni di **mancata corrispondenza tra la richiesta e lo stato del dominio**.
  Il `RestExceptionHandler` le mappa tipicamente in:
  - `404 Not Found`, per risorse inesistenti;
  - con un payload JSON che riporta:
    - `errorCode` specifici (es. `USER_NOT_FOUND`, `AIRPORT_NOT_FOUND`);
    - un messaggio descrittivo contestualizzato.
- **Gestione dei parametri non coerenti**
  Errori dovuti a parametri query non validi (es. `days <= 0`, intervalli temporali con from > to, direzioni non ammesse) generano eccezioni applicative o di validazione che vengono mappate in `400 Bad Request`, con un messaggio che descrive la condizione non soddisfatta.

- **Gestione delle eccezioni tecniche**
  Eccezioni non previste (errori di runtime, problemi di serializzazione JSON, errori inattesi nei client esterni) vengono intercettate da un **catch-all handler** che produce:
  - una risposta `500 Internal Server Error`;
  - un payload di errore generico, privo di dettagli tecnici, per evitare leak informativi;
  - log a livello server sufficientemente dettagliati per l'analisi post-mortem.

Il modello di risposta d'errore è progettato per essere **omogeneo** tra User Manager e Data Collector, in modo che un eventuale client integratore possa gestire in maniera uniforme tutti i casi di errore, indipendentemente dal microservizio interrogato.

8.2. ROBUSTEZZA NELLE INTERAZIONI CON SERVIZI ESTERNI (OpenSky down, 404, ecc.)

Le interazioni con il servizio esterno **OpenSky Network** introducono un elemento di incertezza dovuto alla natura distribuita e alle possibili condizioni di malfunzionamento o indisponibilità. Per questo motivo sono state adottate alcune scelte orientate alla **robustezza** e alla gestione controllata degli errori.

Gli aspetti principali sono:

- **Gestione delle condizioni di "nessun dato disponibile"**
  OpenSky può restituire:
  - risposte vuote (nessun volo nell'intervallo richiesto);
  - codici HTTP specifici (ad esempio 404 interpretato come *nessun volo disponibile*).
    In questi casi:
  - l'`OpenSkyClient` traduce tali condizioni in liste vuote di DTO;
  - il `FlightCollectionService` interpreta correttamente queste situazioni come *assenza di voli*, non come errore;
  - nessuna eccezione viene propagata verso l'alto, e il *Data DB* non viene modificato per quell'intervallo.
- **Gestione degli errori di rete e indisponibilità del servizio**
  In presenza di:
  - timeout;
  - connessioni rifiutate;
  - errori DNS o indisponibilità temporanea del servizio OpenSky;
    l'`OpenSkyClient` genera eccezioni che vengono intercettate a livello di `FlightCollectionService`.

La strategia adottata prevede:

  - log dettagliati dell'errore con informazioni sull'aeroporto coinvolto, sull'intervallo temporale e sulla natura dell'errore;
  - nessuna propagazione di eccezioni verso l'esterno, poiché si tratta di un processo schedulato, non richiesto dal client;
  - possibilità di introdurre in futuro meccanismi di **retry** con backoff esponenziale o di **circuit breaker**, senza modificare l'interfaccia pubblica delle API.

- **Isolamento dell'errore per singolo aeroporto**
  Un errore nella raccolta voli per un determinato aeroporto non deve compromettere la raccolta per gli altri aeroporti di interesse:
  - il `FlightCollectionService` itera sugli aeroporti e gestisce in modo indipendente eventuali errori per ciascuno di essi;
  - l'interruzione del flusso per un aeroporto non blocca il ciclo complessivo.
- **Protezione dall'invalidità del token OAuth2**
  Il token viene:
  - richiesto tramite grant *client credentials*;
  - memorizzato insieme alla scadenza prevista;
    in caso di:
  - token scaduto;
  - risposta che indica un problema di autenticazione;
    l'`OpenSkyClient` può invalidare la cache del token e ripetere la procedura di autenticazione, minimizzando l'impatto sul flusso di raccolta.

Queste scelte garantiscono che l'indisponibilità temporanea di OpenSky o l'assenza di dati non compromettano la stabilità del sistema, limitandosi a ridurre la quantità di dati aggiornati disponibili per le interrogazioni successive.

8.3. SCALABILITA' E ISOLAMENTO DEI DOMINI (USER MANAGER VS DATA COLLECTOR)

L'architettura è progettata per garantire **scalabilità** e **isolamento dei domini** attraverso:

- un modello a **microservizi indipendenti**;
- una separazione logica dei database;
- l'uso di protocolli e contratti ben definiti (REST e gRPC).

Dal punto di vista della scalabilità:

- **User Manager**
  - gestisce operazioni relativamente leggere, centrali sulla registrazione e validazione degli utenti;

    - può essere scalato orizzontalmente aumentando il numero di istanze del microservizio dietro un bilanciatore HTTP/gRPC, mantenendo un unico *User DB* centralizzato;
    - essendo sostanzialmente *stateless* dal punto di vista applicativo (stato persistente nel database), non richiede sincronizzazione di stato tra istanze.
- **Data Collector**
    - è il servizio maggiormente esposto a carichi variabili:
        - richieste REST del client per interrogazioni sui voli;
        - carico aggiuntivo dovuto allo scheduler per la raccolta periodica;
    - può essere scalato selettivamente in base alle esigenze del dominio voli, indipendentemente dal User Manager;
    - la separazione del *Data DB* consente di ottimizzare parametri e risorse del database orientati alla lettura intensiva e alle query su grandi volumi di dati storici.

Dal punto di vista dell'isolamento tra domini:

- il **dominio utente** (User Manager + User DB) e il **dominio voli/interessi** (Data Collector + Data DB) sono trattati come *bounded context* separati;
- la comunicazione tra domini avviene esclusivamente tramite:
    - **REST** verso l'esterno;
    - **gRPC** per la validazione dell'utente, con un contratto dedicato;
- l'assenza di join diretti cross-database preserva:
    - l'indipendenza dei modelli dati;
    - la possibilità di evolvere i domini (schema, tecnologia, deployment) in modo disaccoppiato.

Questa impostazione consente di intervenire in modo mirato sul dimensionamento delle risorse e sull'evoluzione architetturale, senza effetti collaterali sugli altri componenti.

8.4. MANUTENIBILITÀ E ORGANIZZAZIONE DEL CODICE (LAYERING, SEPARAZIONE DELLE RESPONSABILITÀ)

La manutenibilità del sistema è supportata da una **organizzazione del codice rigorosa**, basata su:

- suddivisione in microservizi con repository e codebase distinte;
- adozione di un'architettura a layer ben definiti all'interno di ciascun servizio;
- uso coerente di pattern e convenzioni di naming.

Gli elementi principali sono:

- **Layer di presentazione (REST controller)**
  - Contiene esclusivamente:
    - mapping degli endpoint;
    - validazione degli input;
    - traduzione tra DTO e dominio;
  - evita di incorporare logica di business complessa, che rimane confinata nei service.
- **Layer di business (service)**
  - Incapsula la logica di dominio:
    - regole di idempotenza;
    - orchestrazione tra repository, client esterni, scheduler;
    - decisioni legate a vincoli e semantiche applicative;
  - facilita il testing unitario grazie a dipendenze iniettate e a un perimetro ben definito.
- **Layer di accesso dati (repository + entity)**
  - Implementato tramite Spring Data JPA:
    - riduce il codice boilerplate per query e operazioni CRUD;
    - favorisce la leggibilità attraverso metodi derivati (`findBy...`, `countBy...`);
  - le entità JPA sono concentrate in package dedicati, separati dai DTO esposti verso l'esterno.
- **Client esterni e gRPC**
  - I client HTTP (`OpenSkyClient`) e gRPC (`UserValidationGrpcClient`) sono incapsulati in componenti dedicati:
    - nascondono la complessità dei protocolli sottostanti;
    - permettono di sostituire o estendere la logica di integrazione senza toccare il resto del codice.
- **Scheduler**
  - Posizionato in un componente specializzato (`FlightCollectionScheduler`);
  - richiama esclusivamente metodi di servizio, senza introdurre logica di dominio all'interno del scheduler stesso.

- **Gestione delle eccezioni**
    - Accentrata in un unico componente per microservizio (`RestExceptionHandler`);
    - riduce duplicazioni e assicura coerenza nella gestione degli errori.

Questa organizzazione:

- facilita l'**estensione** delle funzionalità (ad esempio l'aggiunta di nuovi endpoint o nuove query sui voli) intervenendo in punti circoscritti;
- semplifica l'**attività di refactoring**;
- rende più agevole l'adozione di pratiche di testing automatico (unit test, integration test) grazie a confini chiari tra componenti.

8.5. CONSIDERAZIONI SU SICUREZZA E GESTIONE DELLE CREDENZIALI (TOKEN OpenSky, VARIABILI D'AMBIENTE, ECC.)

La sicurezza è considerata sia a livello di **gestione delle credenziali** sia a livello di esposizione delle interfacce.

Gli aspetti principali sono:

- **Gestione delle credenziali e dei segreti**
    - Le credenziali del database (utente, password), così come le credenziali per l'accesso a OpenSky (`client_id`, `client_secret`), non sono codificate nel codice sorgente;
    - vengono fornite ai microservizi tramite:
        - **variabili d'ambiente**;
        - file di configurazione esterni (`application.yml` / `application.properties`) gestiti a livello di deployment;
    - ciò permette di:
        - mantenere il repository sorgente privo di segreti;
        - differenziare agevolmente le configurazioni tra ambienti (sviluppo, test, produzione).
- **Gestione del token OAuth2 di OpenSky**
    - Il token ottenuto tramite il flusso *client credentials*:
        - viene memorizzato in memoria applicativa insieme alla scadenza;
        - non viene scritto nei log;
        - viene rigenerato solo quando necessario, riducendo l'esposizione delle credenziali di base.

    - L'header `Authorization` con il *bearer token* è gestito esclusivamente all'interno dell'`OpenSkyClient`, evitando che altri componenti del sistema debbano conoscere o manipolare direttamente tali informazioni.

- **Isolamento dei database**
    - La separazione tra User DB e Data DB consente di applicare, se necessario, **politiche di accesso differenziate**:
        - permessi più restrittivi per il database utenti;
        - permessi specifici per il dominio voli.
    - I microservizi accedono esclusivamente al proprio database logico, secondo il principio del *least privilege*.
- **Esposizione delle API**
    - Gli endpoint REST sono progettati per non esporre dettagli interni, stacktrace o informazioni sensibili in caso di errore;
    - i payload di errore sono limitati a:
        - codici di errore;
        - messaggi descrittivi di alto livello.
- **Configurabilità e hardening futuro**
    - L'uso di Docker e Docker Compose facilita l'integrazione con meccanismi di sicurezza a livello infrastrutturale:
        - reti interne Docker non esposte all'esterno;
        - possibilità di inserire *reverse proxy* o API Gateway con TLS, autenticazione e rate limiting;
    - la separazione delle responsabilità nei microservizi permette di aggiungere:
        - autenticazione/autorizzazione sulle API REST (ad esempio OAuth2, JWT);
        - logging sicuro e tracciabilità (correlation ID, audit trail) senza modificare la logica di dominio.

Queste considerazioni contribuiscono a costruire una base solida per l'evoluzione del sistema verso scenari operativi con requisiti di sicurezza più stringenti, mantenendo al contempo la chiarezza e la pulizia della codebase.

9. POSSIBILI ESTENSIONI ED EVOLUZIONI DEL SISTEMA

9.1. ESTENSIONI FUNZIONALI E ANALITICHE

Le funzionalità attualmente esposte coprono il nucleo essenziale del dominio: gestione degli utenti, configurazione degli interessi utente–aeroporto, raccolta periodica dei voli da OpenSky e interrogazione dei flight records tramite endpoint mirati. Su questa base è possibile sviluppare un ventaglio di **estensioni funzionali** che rendano il sistema più espressivo, analitico e orientato a casi d'uso avanzati.

Una prima direttrice riguarda l'arricchimento delle **API di interrogazione**:

- introduzione di query orientate alle **compagnie aeree** (laddove i dati OpenSky o sorgenti aggiuntive consentano di identificare il vettore), con endpoint dedicati che permettano di analizzare il traffico per compagnia, rotta o fascia oraria;
- estensione delle API per supportare interrogazioni **multi-airport**, ad esempio con parametri che permettano di confrontare in un'unica risposta il traffico di più aeroporti di interesse;
- supporto a **pattern temporali più articolati**, come:
    - distribuzioni orarie dei voli su una giornata o su più giorni;
    - confronti tra giorni feriali e weekend;
    - analisi mensili o stagionali del traffico.

In parallelo, le interrogazioni esistenti possono essere arricchite con **filtri e opzioni avanzate**:

- filtri combinati su `airportCode`, `direction`, `flightNumber`, `status`, intervalli di `delayMinutes`;
- parametri standardizzati di **paginazione e ordinamento** (`page`, `size`, `sort`) per le liste di voli, con sorting configurabile su campi quali `actualTime`, `delayMinutes`, `flightNumber`;
- endpoint dedicati a **ricerche puntuali**, come il recupero dello storico di un determinato volo in un intervallo definito.

Sul fronte delle **aggregazioni**, il sistema può esporre nuove API che restituiscano indicatori sintetici, tra cui:

- ritardo medio per aeroporto, direzione, fascia oraria o compagnia;
- breakdown per **stato operativo** del volo (atterrato, cancellato, dirottato, in volo);
- indicatori di *on-time performance* per aeroporto o per insiemi di aeroporti.

Queste estensioni funzionali si integrano con la roadmap che prevede, in una fase successiva, l'introduzione di pattern come **CQRS** e **outbox**: le API analitiche potranno essere servite da viste denormalizzate o da proiezioni dedicate, alimentate in modo asincrono e ottimizzate per la lettura massiva, senza impattare sul modello transazionale

attuale. In tal modo, la parte di lettura potrà scalare ed evolvere indipendentemente dai workflow di scrittura (registrazione utenti, interessi, ingestione voli).

9.2. ESTENSIONI ARCHITETTURALI E INFRASTRUTTURALI

L'architettura corrente è basata su due microservizi sincroni (**User Manager** e **Data Collector**), che comunicano via REST e gRPC e sono orchestrati in ambiente Docker Compose. Su queste basi il sistema può evolvere introducendo, in modo graduale, componenti e pattern avanzati orientati a una maggiore **resilienza**, **osservabilità** e **flessibilità architetturale**.

Un asse centrale di evoluzione è la transizione verso un paradigma **event-driven**, mediante l'introduzione di un **message broker** (ad esempio Kafka o tecnologie analoghe):

- il User Manager potrebbe emettere eventi di dominio quali `UserRegistered` o `UserDeleted`, pubblicati su topic dedicati;
- il Data Collector potrebbe emettere eventi `InterestCreated`, `InterestDeleted`, `FlightsCollected` quando vengono inseriti batch significativi di flight records;
- altri microservizi futuri (analytics, notifiche, compliance, data export) potrebbero sottoscriversi a questi topic e reagire in maniera asincrona, senza introdurre accoppiamenti diretti ai servizi esistenti.

In questo contesto, l'adozione del **pattern outbox** diventa cruciale per garantire l'**affidabilità** della pubblicazione degli eventi:

- le operazioni transazionali sui database (ad esempio la creazione di un interesse o l'inserimento di un batch di `FlightRecord`) verrebbero accompagnate dalla registrazione di eventi in una tabella di *outbox* nello stesso database;
- un *outbox worker* dedicato leggerebbe tali record e li pubblicherà sul broker, assicurando che non vi siano discrepanze tra lo stato persistito e gli eventi emessi, anche in presenza di failure parziali.

In scenari più complessi di coordinamento tra microservizi, l'introduzione di un **saga orchestrator** consentirebbe di modellare **transazioni distribuite** basate su sequenze di step e azioni compensative:

- una saga potrebbe coordinare, ad esempio, la creazione di un "profilo utente esteso" che coinvolga User Manager, Data Collector e servizi futuri (es. billing, notifiche), garantendo che in caso di fallimento di uno step vengano eseguite le operazioni di rollback logico sugli step precedenti;
- l'orchestratore della saga potrebbe essere implementato come microservizio dedicato, o come componente integrato in un *workflow engine* esterno.

Dal punto di vista dell'**esposizione delle API**, l'introduzione di un **API Gateway** costituisce un'evoluzione naturale:

- aggregazione degli endpoint di User Manager e Data Collector dietro un'unica *facciata* di ingresso;

- gestione centralizzata di:
    - autenticazione e autorizzazione (OAuth2, JWT, API key);
    - rate limiting, throttling e protezione da abusi;
    - trasformazioni di payload, versioning delle API, routing condizionale;
- possibilità di esporre API *composite* che orchestrano chiamate a più microservizi, semplificando l'esperienza del client.

Sul piano **infrastrutturale**, il passaggio da Docker Compose a una piattaforma di **container orchestration** (ad esempio Kubernetes) consentirebbe:

- auto-scaling dei microservizi in base a metriche di carico;
- deployment rolling, canary e blue–green;
- configurazione centralizzata (ConfigMap, Secret), gestione dei secret e delle credenziali a livello cluster.

In parallelo, l'introduzione di una **stack di observability** (metriche, log strutturati, tracing distribuito) permetterebbe di monitorare il sistema in modo molto più fine:

- esportazione di **metriche applicative** e infrastrutturali (latenza media delle API, throughput delle raccolte da OpenSky, tassi di errore gRPC, lag sui topic del broker);
- centralizzazione dei log con campi strutturati (correlation ID, userEmail, airportCode) per facilitare l'analisi;
- **tracing distribuito** per seguire l'intero ciclo di vita di una richiesta, dal gateway al microservizio interno, fino al database e al broker.

Infine, l'introduzione di **CQRS** consentirebbe di separare a livello architetturale il modello di scrittura (command) dal modello di lettura (query):

- i comandi (registrazione utente, registrazione interessi, ingestione voli) verrebbero gestiti dai microservizi transazionali attuali;
- le query analitiche e di reporting potrebbero essere servite da proiezioni e viste ottimizzate, alimentate dagli eventi tramite broker e outbox.

9.3. INTEGRAZIONE CON UN FRONT-END DEDICATO O DASHBOARD DI MONITORAGGIO

Le API REST attualmente esposte costituiscono una base stabile per lo sviluppo di un **front-end dedicato** e di una **dashboard di monitoraggio** orientata agli operatori.

Dal punto di vista dell'utente finale, un front-end web o mobile potrebbe offrire:

- una **gestione interattiva degli interessi**:
    - interfacce per aggiungere o rimuovere aeroporti di interesse;
    - visualizzazione immediata degli aeroporti sottoscritti e dei relativi indicatori di traffico;

- viste **sintetiche e analitiche** sui voli:
    - elenco dei voli più recenti per gli aeroporti preferiti, con evidenza di stato operativo e ritardo;
    - grafici temporali per monitorare l'andamento del traffico in determinate fasce orarie o periodi;
- funzioni di **notifica** future:
    - avvisi su variazioni significative del traffico;
    - notifiche su pattern anomali (ritardi persistenti, cancellazioni elevate).

La roadmap architetturale che prevede API Gateway, CQRS ed eventi su broker facilita l'integrazione:

- il front-end potrebbe dialogare unicamente con l'**API Gateway**, delegando ad esso il routing verso i microservizi sottostanti;
- le **proiezioni di lettura** (view model CQRS) possono essere esposte da servizi dedicati alle query, ottimizzati per le esigenze del front-end (risposte leggere, pre-aggregate, con formati specifici per i widget grafici).

Parallelamente, una **dashboard di monitoraggio tecnico–operativo** potrebbe integrare:

- metriche esposte dai microservizi e raccolte da un sistema di monitoring (ad esempio Prometheus, con visualizzazione tramite Grafana);
- indicatori di funzionamento dello scheduler (ultimo run, durata, volumi raccolti, errori verso OpenSky);
- statistiche sul throughput dei topic del **message broker**, eventuali lag dei consumer, numero di eventi in outbox in attesa di pubblicazione;
- visualizzazioni dei **traces distribuiti**, per analizzare le latenze end-to-end.

Questa integrazione consente di mantenere separati i due piani di osservazione:

- il piano **business**, focalizzato su aeroporti, voli e interessi utente;
- il piano **operativo e infrastrutturale**, focalizzato sull'affidabilità della pipeline di ingestione, sulla salute dei microservizi e sulla qualità del servizio erogato.

9.4. UTILIZZO DEL SISTEMA IN CONTESTI REALI O SCENARI DI BIG DATA PIÙ COMPLESSI

L'utilizzo in contesti reali, con volumi di traffico elevati e orizzonti temporali di retention estesi, richiede una serie di evoluzioni mirate alla **scalabilità**, alla **gestione dei dati** su larga scala e alla **integrazione con ecosistemi di big data**.

Sul piano del **data management**, l'aumento del numero di aeroporti monitorati, della frequenza di raccolta e della durata dello storico di `FlightRecord` rende necessari:

- indici mirati su campi come `airport_id, direction`, `actual_time`, `status`, per evitare degradi prestazionali delle query;

- **partizionamento** delle tabelle, ad esempio:
    - per intervallo temporale (partizioni mensili o settimanali);
    - per aeroporto o cluster di aeroporti, in modo da distribuire carico e storage;
- politiche esplicite di **retention**:
    - mantenimento di un orizzonte “online” ridotto nel database operativo;
    - archiviazione dei dati più vecchi in tabelle separate, in un data warehouse o in un data lake.

L’introduzione di **message broker, CQRS e outbox** abilita scenari più ricchi di big data:

- un **servizio di proiezione** (read side CQRS) potrebbe consumare eventi `FlightsCollected` e mantenere viste aggregate appositamente progettate per analytics, indipendenti dal modello transazionale;
- pipeline **ETL/ELT** potrebbero consumare gli stessi eventi dal broker o attingere dagli archivi per alimentare:
    - un **data warehouse** relazionale;
    - un **data lake** su storage distribuito, integrabile con strumenti di calcolo massivo (Spark, Flink, ecc.).

In scenari con requisiti di **near real-time analytics**, i flight records potrebbero entrare in una pipeline di **stream processing**:

- il Data Collector pubblica gli eventi dei voli raccolti su un topic Kafka;
- uno o più job di streaming elaborano i flussi per:
    - calcolare aggregazioni sliding (medie mobili, conteggi per finestra temporale);
    - identificare pattern anomali (picchi di ritardi, congestione, cancellazioni anomale);
    - alimentare dashboard real-time.

Dal punto di vista dell’**operatività su larga scala**, l’evoluzione verso:

- deployment su Kubernetes;
- scaling orizzontale del Data Collector e dei servizi di query;
- distribuzione dei database (replica, sharding, o uso di database specializzati per time series)

permette di sostenere carichi elevati, riducendo al contempo il rischio di single point of failure.

In questo scenario, i pattern **saga** e **orchestrazione** possono diventare rilevanti per coordinare processi complessi multi-servizio (ad esempio workflow che combinano gestione utenti, policy di accesso ai dati, provisioning di viste analitiche o configurazione di

regole di alert). L'orchestratore di saga, integrato con il broker e con il sistema di observability, può fornire tracciabilità fine e gestione controllata degli errori in presence di flussi lunghi e articolati.

L'insieme di queste evoluzioni consente al sistema di passare da un asset centrato sul **monitoraggio focalizzato di aeroporti di interesse** a una piattaforma in grado di supportare **analisi su larga scala**, integrazione con ecosistemi di dati eterogenei e requisiti operativi tipici di contesti reali caratterizzati da alti volumi e alta variabilità del traffico.

10. CONCLUSIONI

10.1. SINTESI DELLE SCELTE PROGETTUALI ADOTTATE

L'architettura realizzata si fonda sulla **separazione netta dei domini applicativi** in due microservizi distinti:

- **User Manager**, responsabile del *bounded context* utente (registrazione, gestione e validazione dell'identità), con un proprio database logico (*User DB*);
- **Data Collector**, responsabile del dominio aeroporti–interessi–voli, con un proprio database logico (*Data DB*) e l'integrazione con il servizio esterno **OpenSky Network**.

Questa separazione è rafforzata dall'uso di **contratti di comunicazione espliciti**:

- interfacce **REST/HTTP** per l'esposizione delle funzionalità verso il client esterno;
- interfaccia **gRPC** dedicata alla validazione utente (*UserValidationService*), che consente al Data Collector di verificare l'esistenza degli utenti senza accedere direttamente al *User DB*.

La modellazione dei dati è basata su uno schema relazionale **pulito e aderente al dominio**:

- nel *User DB*, l'entità `users` utilizza l'e-mail come **chiave primaria**, coerente con l'uso dell'e-mail come identificatore globale dell'utente;
- nel *Data DB*, le entità `airports`, `user_airport_interest` e `flight_records` modellano rispettivamente il catalogo degli aeroporti, la relazione di interesse utente–aeroporto e lo storico dei voli, con vincoli di unicità e chiavi esterne che preservano l'integrità interna del dominio.

Le scelte tecnologiche si inseriscono in un contesto consolidato:

- **Spring Boot** per entrambi i microservizi, con un'architettura a layer (controller, service, repository);
- **Spring Data JPA** per l'accesso ai dati, che riduce il codice boilerplate e rende leggibili le query attraverso metodi derivati;
- **PostgreSQL** come motore relazionale condiviso, suddiviso in due schemi logici separati;
- **Flyway** per la gestione delle migrazioni di schema;
- **Docker e Docker Compose** per il packaging, l'orchestrazione locale e la riproducibilità degli ambienti.

Sul piano della logica applicativa, sono state adottate esplicitamente politiche di **idempotenza** e **consistenza applicativa**:

- registrazione utente con semantica *at-most-once*;
- vincolo di unicità sulla coppia *(user_email, airport_id)* per gli interessi;

- validazione cross-dominio degli utenti via gRPC prima della creazione degli interessi.

La raccolta dei dati da OpenSky è gestita tramite uno **scheduler interno** al Data Collector, che interroga il servizio esterno per intervalli temporali configurabili e popola il *Data DB* con i `FlightRecord`. Le API di interrogazione (`/api/flights/last`, `/api/flights/average`, `/api/flights`) forniscono viste sintetiche e di dettaglio sui voli, filtrabili per aeroporto, direzione e intervalli temporali.

Nel complesso, il sistema adotta un **modello microservizi ben strutturato**, con separazione dei domini, contratti chiari, gestione rigorosa dei dati e un set di API che copre i principali scenari d'uso legati al monitoraggio del traffico aereo su aeroporti di interesse.

10.2. VALUTAZIONE CRITICA DELL'ARCHITETTURA IMPLEMENTATA

L'architettura implementata presenta diversi punti di forza sul piano **strutturale**, **manutenibile** e **operativo**, insieme ad alcune aree che possono essere oggetto di ulteriori miglioramenti.

Dal punto di vista dei punti di forza, si evidenziano:

- una **chiara separazione delle responsabilità**:
    - il User Manager governa completamente il ciclo di vita degli utenti;
    - il Data Collector concentra in sé la gestione degli interessi, la raccolta e l'interrogazione dei dati di volo;
- una **modellazione dei domini coerente con i principi DDD-lite**:
    - i due bounded context non condividono tabelle o schema;
    - lo scambio di informazioni avviene tramite contratti espliciti (REST e gRPC), evitando accoppiamenti stretti a livello di database;
- un **layering interno pulito**:
    - controller REST responsabili solo di esposizione e validazione;
    - service centrati sulla logica di dominio;
    - repository dedicati all'accesso ai dati;
    - client esterni e scheduler incapsulati in componenti separati;
- una gestione **robusta e uniforme delle eccezioni**, con mapping verso HTTP tramite `RestExceptionHandler` e payload di errore strutturati;
- un approccio consapevole alla **consistenza e all'idempotenza**, che riduce i rischi di stato incoerente in presenza di retry o richieste duplicate.

Sul piano critico, emergono alcuni **trade-off** e possibili limiti:

- la comunicazione tra microservizi è attualmente **interamente sincrona** (gRPC per la validazione utente), con un legame diretto tra la disponibilità del User Manager e la capacità del Data Collector di registrare interessi:
    - questo introduce un potenziale punto di failure;
    - in contesti con requisiti più stringenti di disponibilità, potrebbero essere opportuni pattern asincroni o meccanismi di *fallback*;
- l'uso di un'unica istanza PostgreSQL, seppur con due schemi logici separati, rende i due domini **non completamente indipendenti a livello infrastrutturale**:
    - in scenari fortemente scalati, potrebbe essere necessario isolare anche fisicamente i database o adottare soluzioni multi–database/multi–cluster;
- l'assenza di funzionalità trasversali come **autenticazione/autorizzazione sulle API REST**, **rate limiting** o **quote per client**:
    - non incide sulla correttezza logica del sistema;
    - rappresenta però un limite in scenari esposti a utenti multipli o non fidati;
- la raccolta periodica tramite scheduler interno è **semplice ed efficace** ma poco flessibile rispetto a scenari di:
    - ingestione a frequenza dinamica;
    - orchestrazioni complesse basate su eventi o su carico effettivo.

Dal punto di vista delle **prestazioni e della scalabilità**, l'architettura è adeguata per un volume di dati medio e per un numero controllato di aeroporti di interesse. All'aumentare della frequenza di scraping, del numero di aeroporti monitorati e dell'orizzonte temporale di retention, sarà necessario:

- ottimizzare indici e query;
- considerare partizionamenti delle tabelle dei `FlightRecord`;
- valutare soluzioni di caching o di pre-aggregazione per le statistiche più richieste.

Nel complesso, l'architettura mostra un **buon equilibrio** tra semplicità, chiarezza dei confini e preparazione all'evoluzione futura, pur lasciando spazio a potenziamenti in ambiti di disponibilità, sicurezza, scalabilità estrema e integrazione asincrona.

10.3. POSSIBILI SVILUPPI FUTURI E RIFLESSIONI FINALI

L'architettura attuale fornisce una base sufficientemente solida da poter essere **estesa ed evoluta** lungo diverse direttrici, sia funzionali sia architetturali, senza stravolgere i principi alla base del sistema (separazione dei domini, microservizi autonomi, contratti ben definiti, modellazione relazionale coerente).

Un primo asse di evoluzione riguarda il **rafforzamento del livello funzionale e analitico**, facendo leva sulle strutture già presenti:

- introduzione di **API analitiche dedicate**, supportate da viste o proiezioni ottimizzate per la lettura (es. statistiche di ritardo, indicatori di performance per aeroporto, confronti tra gruppi di aeroporti);
- adozione del **pattern CQRS** per separare in modo netto il modello di scrittura (registrazione utenti, interessi, ingestione voli) dal modello di lettura, servito da read model specifici per reporting e dashboard;
- utilizzo di **outbox** per propagare in modo affidabile gli eventi di dominio dal perimetro transazionale ai componenti di lettura o ai sistemi analitici, mantenendo allineato lo stato tra database e flussi di eventi.

Un secondo asse potenziale è quello **architetturale–infrastrutturale**, orientato a incrementare disaccoppiamento, resilienza e osservabilità:

- introduzione di un **message broker** come backbone event-driven, su cui pubblicare eventi quali `UserRegistered`, `InterestCreated`, `FlightsCollected`, permettendo ad altri servizi di reagire in maniera asincrona;
- definizione di **saga** per modellare processi distribuiti complessi, con step multipli e azioni compensative, ad esempio per orchestrare workflow che coinvolgano più servizi (gestione utenti, interessi, eventuali moduli di billing o notifiche);
- collocazione di un **API Gateway** come punto di ingresso unico verso il sistema, incaricato di:
    - gestire autenticazione e autorizzazione;
    - applicare politiche di rate limiting e protezione;
    - centralizzare logging, routing e versioning delle API;
- consolidamento di una **stack di observability** completa (metriche, log strutturati, tracing distribuito) per ottenere visibilità end-to-end sulle chiamate, sulle interazioni fra microservizi e sul comportamento della pipeline di ingestione.

Sul piano dei dati e dei carichi, l'evoluzione verso contesti con volumi maggiori e requisiti tipici di **big data** può poggiare su ulteriori sviluppi:

- ottimizzazione del livello di persistenza tramite indici dedicati, partizionamento delle tabelle dei flight records e politiche di retention differenziate tra dati "caldi" e dati di archivio;
- integrazione con **piattaforme analitiche** (data warehouse, data lake, motori di streaming) alimentate da eventi o da esportazioni periodiche, così da abilitare analisi storiche profonde, near real-time analytics e, se necessario, modelli predittivi;
- possibile utilizzo di motori specializzati (time-series database, sistemi di stream processing) per gestire in modo efficiente flussi ad alto throughput e query temporali complesse.

In prospettiva, queste direttrici lasciano spazio alla costruzione di un ecosistema più ampio attorno al nucleo esistente, in cui orchestrazione, messaggistica, osservabilità avanzata, API gateway, CQRS, outbox e saga possano essere introdotti gradualmente per rispondere a esigenze crescenti di scalabilità, affidabilità e capacità analitica, mantenendo come riferimento i principi di chiarezza architetturale e di separazione dei domini già alla base del sistema.