UNIVERSITÀ DEGLI STUDI DI CATANIA

Corso di Laurea Magistrale in Ingegneria Informatica LM-32

Relazione Homework 1

Corso di Distributed Systems and Big Data

Anno Accademico 2025-2026

Studente:
Riccardo Maria Villaggio
Matricola: 1000084767

Indice

1 Introduzione

Il presente documento illustra le scelte progettuali e implementative adottate nello sviluppo di un sistema a microservizi distribuito dockerizzato per il monitoraggio del traffico aereo sfruttando le API di OpenSky Network. Il progetto è stato realizzato nell'ambito del corso di Distributed Systems and Big Data, con l'obiettivo di mettere in pratica i principi fondamentali della progettazione di sistemi distribuiti moderni, ponendo particolare attenzione alla comunicazione tra servizi, alla containerizzazione Docker e all'implementazione di politiche di consegna dei messaggi, quali At-Most-Once.
In conformità alle specifiche di progetto, il sistema è stato decomposto in quattro componenti principali, ciascuno incapsulato in un proprio container Docker. L'architettura adottata rispetta fedelmente i requisiti richiesti, con l'aggiunta di alcune funzionalità supplementari che arricchiscono il sistema senza alterarne la struttura fondamentale.
L'applicazione sviluppata si compone di due microservizi principali, ciascuno con responsabilità ben definite e database dedicato, che comunicano tra loro mediante il framework gRPC. Il sistema consente agli utenti di registrarsi, specificare gli aeroporti di interesse e consultare i dati relativi ai voli in partenza e in arrivo, raccolti automaticamente tramite le API pubbliche di OpenSky Network.

1.1 Obiettivi del Sistema

Il sistema è stato progettato per soddisfare i seguenti requisiti funzionali e non funzionali:

- **Gestione degli utenti**: il microservizio User Manager implementa le operazioni di registrazione e cancellazione degli utenti, garantendo la validazione dei dati in ingresso e la protezione delle informazioni sensibili (attualmente l'IBAN) mediante crittografia. È inoltre possibile recuperare i dati di un singolo utente o di tutti gli utenti registrati, e verificare l'esistenza di un utente tramite la sua email.

- **Tracciamento degli interessi utente**: ogni utente registrato può esprimere interesse verso uno o più aeroporti, identificati mediante codice ICAO. Il sistema mantiene questa associazione e la utilizza per determinare quali dati raccogliere e quali rendere accessibili al singolo utente.

- **Raccolta periodica e automatica dei dati**: un componente scheduler integrato nel Data Collector interroga ogni 12 ore l'API di OpenSky Network per recuperare le informazioni sui voli relativi agli aeroporti di interesse delle ultime 24 ore, memorizzandole in un database dedicato.

- **Fornitura di statistiche e analisi**: il sistema espone endpoint REST che consentono di interrogare i dati raccolti, ottenendo informazioni quali l'ultimo volo registrato (in partenza e/o in arrivo), la media giornaliera dei voli e le compagnie aeree più frequenti da un dato aeroporto di interesse.

- **Garanzia politica At-Most-Once**: l'operazione di registrazione utente implementa la politica At-Most-Once, assicurando che una richiesta venga processata al massimo una volta anche in presenza di retry da parte del client.

- **Isolamento e sicurezza**: l'architettura prevede la separazione delle reti Docker, garantendo che ogni microservizio possa accedere esclusivamente al proprio database.

1.2 Tecnologie Utilizzate

Lo stack tecnologico adottato comprende:

- **Python 3.11** con framework **Flask 3.0.0** per le API REST
- **Flask-SQLAlchemy** per l'Object-Relational Mapping
- **gRPC** con Protocol Buffers per la comunicazione inter-service
- **MySQL 8.0** come sistema di gestione database relazionale
- **Docker** e **Docker Compose** per la containerizzazione
- **APScheduler** per la schedulazione dei job periodici
- **Cryptography (Fernet)** per la cifratura dei dati sensibili
- **hashlib** per il calcolo degli hash SHA-256
- **Threading** per l'esecuzione di operazioni in background (ad esempio i server gRPC)
- **ThreadPoolExecutor** per la gestione di più chiamate contemporanee all'API esterna
- **Lock** per la sincronizzazione dell'accesso al refresh token OAuth2

2 Architettura del Sistema

L'architettura del sistema segue il paradigma dei microservizi e delle specifiche di progetto, con una chiara separazione delle responsabilità tra le singole componenti. Questa scelta progettuale consente di ottenere vantaggi significativi in termini di scalabilità, manutenibilità e resilienza del sistema complessivo, utile anche in vista di sviluppi futuri.

2.1 Diagramma Architetturale

La Figura 1 illustra la struttura complessiva del sistema, evidenziando i quattro container Docker, le tre reti isolate e i flussi di comunicazione tra i componenti.

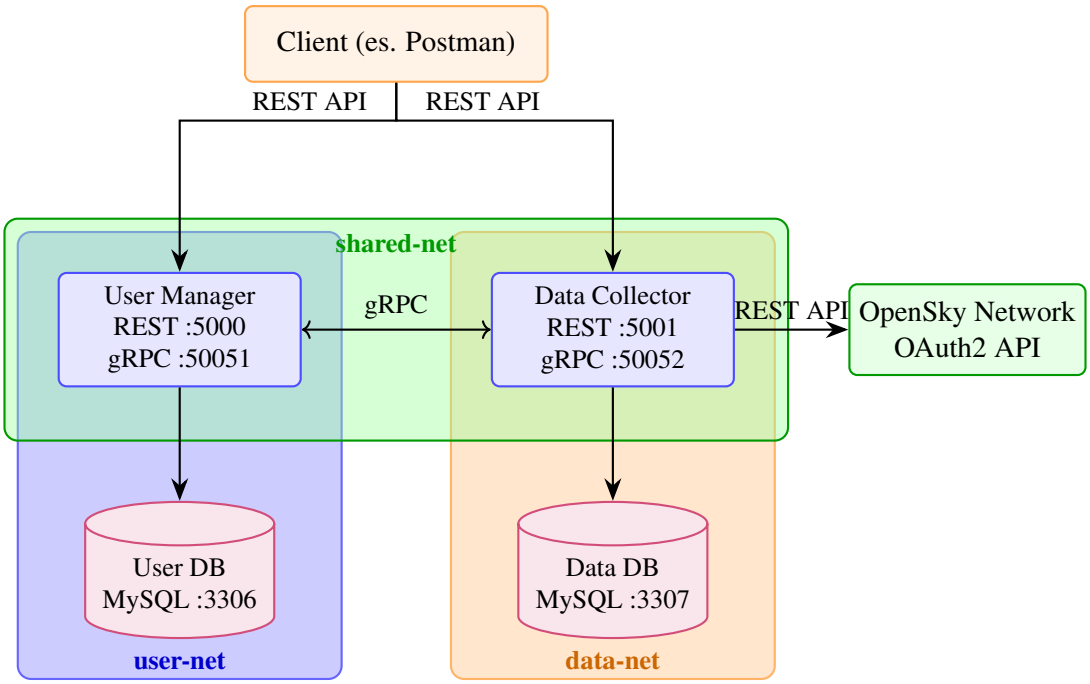

Figura 1: Architettura del sistema

2.2 Descrizione dei Componenti

2.2.1 User Manager Microservice

Il microservizio User Manager costituisce il punto di ingresso per tutte le operazioni relative alla gestione degli utenti. Le sue responsabilità principali includono:

- Esposizione di API REST (porta 5000) per le operazioni sugli utenti (registrazione, recupero del profilo, cancellazione, verifica esistenza).
- Implementazione di un server gRPC (porta 50051) che espone i metodi `VerifyUser` e `GetUser`.
- Implementazione di un client gRPC per richiedere al Data Collector la cancellazione degli interessi associati a un utente in fase di eliminazione.
- Gestione di una Request Cache per l'implementazione della politica At-Most-Once.
- Crittografia dei dati sensibili (IBAN) mediante l'algoritmo di cifratura Fernet.
- Esecuzione di un thread in background per la pulizia periodica della cache delle richieste.

2.2.2 Data Collector Microservice

Il microservizio Data Collector è responsabile della raccolta e gestione dei dati relativi ai voli aerei. Le sue funzionalità comprendono:

- Gestione degli interessi utente (creazione e/o cancellazione dell'associazione utente-aeroporto).
- Verifica dell'esistenza degli utenti mediante chiamate gRPC al User Manager prima di elaborare una qualsiasi richiesta tramite le API che offre.

- Esposizione di API REST (porta 5001) per la consultazione dei dati e delle statistiche sui voli aerei salvati nel Data DB.
- Esposizione di un server gRPC (porta 50052) per ricevere le richieste di cancellazione degli interessi al momento della cancellazione di un utente.
- Esecuzione periodica (ogni 12 ore) di job di raccolta dati tramite APScheduler.
- Utilizzo delle API di OpenSky Network con autenticazione OAuth2.
- Memorizzazione ottimizzata dei dati mediante operazioni di bulk upsert.
- Meccanismo di garbage collection automatica dei dati relativi ad aeroporti non più monitorati (poiché nessun utente ne è più interessato).

2.3 Topologia delle Reti Docker

Una delle scelte architetturali fondamentali riguarda l'isolamento di rete tra i componenti del sistema, in modo tale da garantire che nessun microservizio possa accedere direttamente al database gestito da un altro componente. Sono state, dunque, definite tre reti Docker distinte:

Tabella 1: Configurazione delle reti Docker

Network	Servizi Connessi	Finalità
user-net	user-manager, user-db	Isolamento database utenti
data-net	data-collector, data-db	Isolamento database voli
shared-net	user-manager, data-collector	Comunicazione gRPC

3 Scelte Progettuali e Motivazioni

In questa sezione vengono illustrate le principali scelte progettuali adottate durante lo sviluppo del sistema, con particolare attenzione alle motivazioni tecniche che le hanno guidate.

3.1 Adozione di MySQL 8.0

La scelta di MySQL 8.0 come sistema di gestione database è stata motivata dalle seguenti considerazioni:

- **Struttura dati relazionale**: i dati trattati dal sistema (utenti, interessi, voli) presentano una struttura intrinsecamente di tipo relazionale, che dunque si presta bene all'utilizzo di un RDBMS.
- **Conformità ACID**: requisito fondamentale per operazioni critiche come la registrazione utente.
- **Supporto per vincoli UNIQUE**: essenziale per l'implementazione della politica At-Most-Once.
- **Clausola ON DUPLICATE KEY UPDATE**: consente di ottimizzare significativamente il processo di inserimento massivo dei dati di volo.

La configurazione Docker prevede:

- **User DB**: Mappato sulla porta esterna **3306**.
- **Data DB**: Mappato sulla porta esterna **3307**.

3.2 Comunicazione Eterogenea: REST e gRPC

Il sistema adotta una strategia di comunicazione ibrida, combinando API REST per l'interazione con i client esterni e gRPC per la comunicazione interna tra microservizi. Le motivazioni alla base di questa scelta sono riassunte nella Tabella 2.

Tabella 2: Protocolli di comunicazione adottati

Interazione	Protocollo	Motivazione Architetturale
Client → Microservizi	REST	Adozione dello standard de facto per API pubbliche, garantendo: disaccoppiamento, facilità di debug/testing e interoperabilità con client eterogenei.
Data Collector → User Manager	gRPC	Necessità di verificare l'esistenza dell'utente.
User Manager → Data Collector	gRPC	Necessità di coordinamento della transazione distribuita per la cancellazione utente-interessi (pattern *All-or-Nothing*).

3.3 Definizione dei File Protocol Buffers

Per la definizione delle interfacce gRPC, si è scelto di utilizzare due file `.proto` separati, uno per ciascun servizio, garantendo una chiara separazione dei contratti. I file definiti sono i seguenti:

```
service UserService {
  rpc VerifyUser(VerifyUserRequest) returns (VerifyUserResponse);

  // Metodo predisposto per estensioni future
  rpc GetUser(GetUserRequest) returns (GetUserResponse);
}
```

Listing 1: File user_service.proto - Servizio User Manager

```
service DataCollectorService {
  rpc DeleteInterests(DeleteInterestsRequest) returns (
    DeleteInterestsResponse);
}
```

Listing 2: File data_collector_service.proto - Servizio Data Collector

È opportuno notare la presenza del metodo `GetUser` nel servizio `UserService`. Sebbene tale metodo sia stato definito in fase di progettazione per consentire il recupero strutturato dei dati utente via gRPC, nell'attuale iterazione del sistema non viene utilizzato dal Data Collector. La sua definizione è stata tuttavia mantenuta all'interno dell'Interface Definition Language (IDL) in ottica di estensibilità futura (design for extensibility), permettendo di implementare funzionalità più complesse di scambio dati senza richiedere modifiche.

3.4 Crittografia IBAN con Fernet

L'IBAN rappresenta un dato sensibile che richiede protezione. È stata implementata una strategia di encryption-at-rest utilizzando l'algoritmo Fernet:

```
class User(db.Model):
    _iban = db.Column('iban', db.String(500), nullable=True)
    iban_hash = db.Column(db.String(64), unique=True)

    @property
    def iban(self):
        key = os.getenv('ENCRYPTION_KEY').encode()
        f = Fernet(key)
        return f.decrypt(self._iban.encode()).decode()

    @iban.setter
    def iban(self, value):
        self.iban_hash = hashlib.sha256(value.encode()).hexdigest()
        key = os.getenv('ENCRYPTION_KEY').encode()
        f = Fernet(key)
        self._iban = f.encrypt(value.encode()).decode()
```

Listing 3: Implementazione della crittografia dell'IBAN nel model User

4 Implementazione della Politica At-Most-Once

La politica *At-Most-Once* rappresenta una delle garanzie semantiche fondamentali nei sistemi distribuiti, assicurando che una determinata operazione venga eseguita al massimo una volta, indipendentemente dal numero di tentativi di invio effettuati dal client.

4.1 Motivazione della Scelta Progettuale

La necessità di tale politica deriva dall'intrinseca inaffidabilità delle comunicazioni di rete. In scenari di timeout o perdita del messaggio di risposta, il client non può determinare se la sua richiesta sia stata processata o meno. In questo contesto di incertezza, affidarsi esclusivamente ai vincoli di integrità del database (come l'unicità dell'email o del codice fiscale) risulta insufficiente. Tale approccio, infatti, non permetterebbe di distinguere tra:

- Un **errore reale**, causato da dati effettivamente duplicati inseriti da un utente diverso
- Un **retry automatico**, generato dal client a seguito di instabilità della rete o timeout

Senza un meccanismo di gestione lato messaging, entrambi gli scenari produrrebbero un errore 409 Conflict, rendendo impossibile per il client capire se la propria operazione originale sia andata a buon fine.

4.2 Soluzione Implementata: Request Cache Persistente

L'implementazione si basa su una cache persistente delle richieste, memorizzata nel database e svuotata periodicamente da un thread dedicato. Ogni richiesta viene identificata univocamente attraverso una chiave specifica, che garantisce l'idempotenza durante il retry del client, calcolata come segue:

```
# Identificazione del client mediante hash dell'indirizzo IP
client_id = hashlib.sha256(request.remote_addr.encode()).hexdigest()

# Chiave di idempotenza = Hash(client_id + request_id)
idempotency_key = hashlib.sha256(
    f"{client_id}:{request_id}".encode()
).hexdigest()
```

Listing 4: Calcolo della chiave di idempotenza (ID univoco della richiesta)

4.2.1 Gestione del Request ID

Il sistema supporta due modalità per l'identificazione della richiesta:

1. **Modalità esplicita (consigliata)**: il client fornisce un UUID v4 tramite l'header HTTP `X-Request-ID` o nel campo `request_id` del corpo JSON. Questa modalità offre il massimo controllo al client sulla semantica di retry: utilizzando lo stesso UUID, il client dichiara esplicitamente che si tratta di un nuovo tentativo della stessa operazione.

2. **Modalità implicita (fallback)**: in assenza di un Request ID esplicito, il sistema calcola un hash SHA-256 dell'intero payload JSON. Questa modalità è sicura per la registrazione utente poiché, per definizione di dominio, creare lo stesso utente due volte con dati identici è impossibile; pertanto, ricevere un payload identico implica necessariamente un retry della stessa richiesta.

4.3 Scenari di Funzionamento

La Tabella 3 riassume i diversi scenari gestiti dal sistema e le relative risposte.

Tabella 3: Scenari di gestione At-Most-Once

Scenario	Condizione	Risposta
Prima richiesta (successo)	Cache miss, dati validi	201 Created + salvataggio in cache
Retry con stesso ID	Cache hit (ID già presente)	201 Created (risposta dalla cache)
Dati duplicati, ID diverso	Cache miss, vincolo violato	409 Conflict (email/CF/IBAN esistente)
Dati non validi	Cache miss, validazione fallita	400 Bad Request

4.3.1 Scenario 1: Prima Richiesta con Successo

Quando il server riceve una richiesta per la prima volta (cache miss), verifica la validità dei dati, inserisce l'utente nel database e memorizza la risposta nella Request Cache insieme alla chiave di idempotenza. La risposta 201 Created viene quindi inviata al client.

4.3.2 Scenario 2: Retry con lo Stesso Request ID

Se il server riceve una richiesta con un identificativo già presente nella cache, riconosce immediatamente che si tratta di un duplicato, dovuto ad esempio a una ritrasmissione automatica per scadenza del timeout nel codice del client. In questo caso, il server restituisce immediatamente l'esito positivo dell'operazione precedente senza rieseguirla, garantendo che l'utente non venga creato due volte.

4.3.3 Scenario 3: Dati Duplicati con Request ID Diverso

Questo scenario, seppur raro in condizioni normali, si verifica quando un client tenta di registrare un utente con dati già esistenti nel sistema (stessa email, codice fiscale o IBAN) ma utilizzando un nuovo Request ID. In questo caso, non trattandosi di un retry ma di una nuova richiesta con dati in conflitto, il sistema restituisce correttamente un errore 409 Conflict, permettendo al client di comprendere che l'operazione non può essere completata per ragioni di business logic.

4.4 Diagramma di Sequenza: Registrazione Utente

La Figura 2 illustra il flusso completo della registrazione utente con gestione At-Most-Once, includendo tutti i possibili esiti. Nel diagramma si suppone che il client invii un header `X-Request-ID` con un UUID v4, inoltre le query al database sono state semplificate per chiarezza e leggibilità, e la tabella della Request Cache è rappresentata esplicitamente (e non come parte del User DB) per rendere più evidente il funzionamento del meccanismo At-Most-Once.

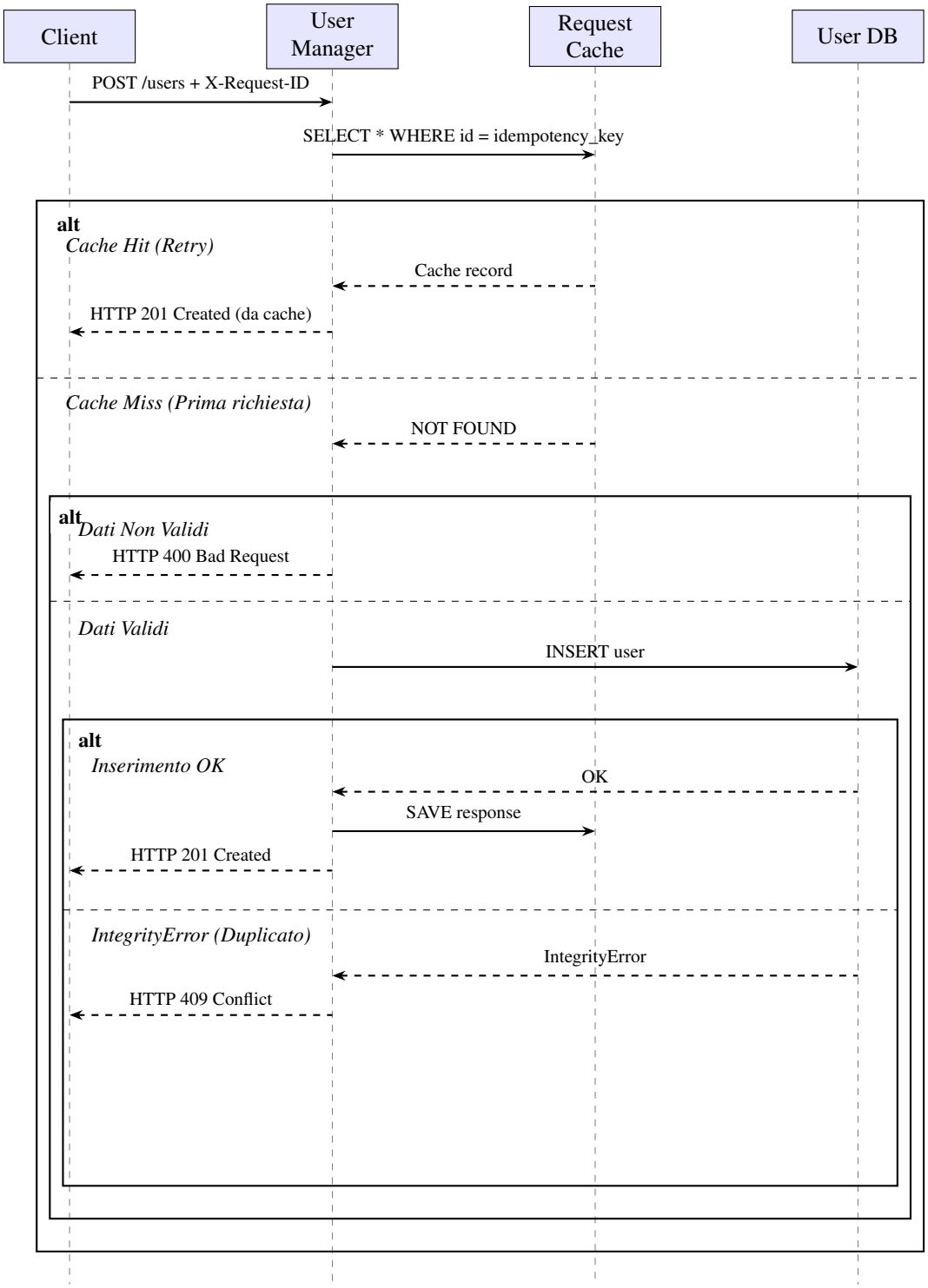

Figura 2: Diagramma di sequenza: Registrazione utente con At-Most-Once

4.5 Cache Cleaner Thread

Per evitare una crescita illimitata della Request Cache, un thread in background esegue periodicamente la pulizia delle entry scadute. Il Time-To-Live (TTL) è configurato a 5 minuti: dopo tale intervallo, un eventuale retry viene trattato come una nuova richiesta indipendente, poiché si assume che il client abbia già ricevuto o gestito l'esito della richiesta originale.

```
def clean_request_cache():
    while True:
        with app.app_context():
            expiration_time = datetime.now(timezone.utc) - timedelta(
   minutes=5)
            deleted = db.session.execute(
                db.delete(RequestCache).where(
                    RequestCache.created_at < expiration_time
                )
            )
            db.session.commit()
        time.sleep(300)  # Esecuzione ogni 5 minuti
```

Listing 5: Implementazione del Cache Cleaner

5 Diagrammi di Sequenza degli Endpoint

In questa sezione vengono presentati i diagrammi di sequenza per tutti gli endpoint esposti dal sistema, inclusi i casi alternativi (i più rilevanti da analizzare) e le eventuali query ai database coinvolti (per esigenze di leggibilità, le query sono state semplificate).

5.1 User Manager: GET /users/{email}

La Figura 3 illustra il recupero di un singolo utente.

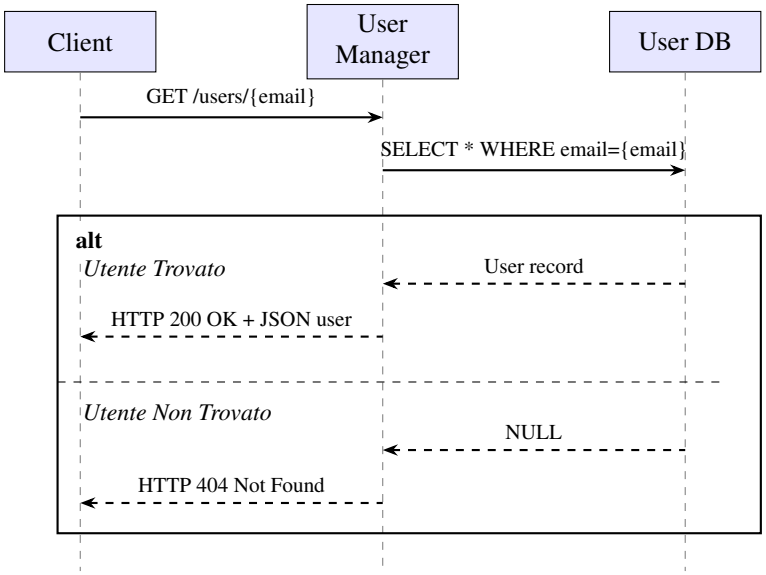

Figura 3: Diagramma di sequenza: GET /users/{email}

5.2 User Manager: GET /users

La Figura 4 illustra il recupero di tutti gli utenti registrati.

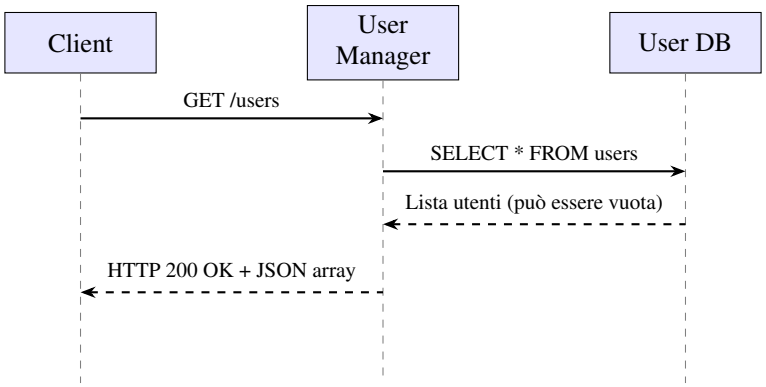

Figura 4: Diagramma di sequenza: GET /users

5.3 User Manager: GET /users/verify/{email}

Questo endpoint molto semplice consente ai client (o ad altri servizi) di verificare rapidamente l'esistenza di un utente senza recuperarne l'intero profilo, utile per controlli durante il normale ciclo di vita di funzionamento del sistema. La Figura 5 mostra il flusso di verifica.

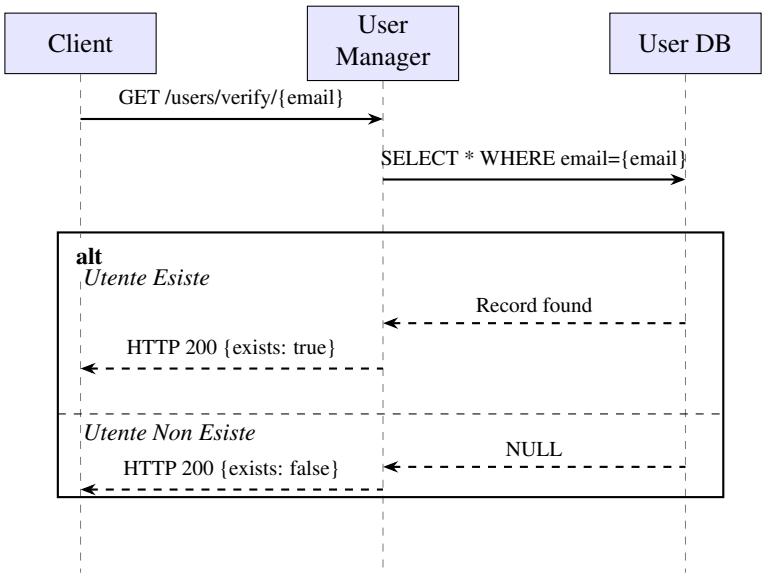

Figura 5: Diagramma di sequenza: GET /users/verify/{email}

5.4 User Manager: DELETE /users/{email}

La cancellazione di un utente richiede il coordinamento tra User Manager e Data Collector per garantire che anche gli eventuali interessi associati vengano rimossi. La Figura 6 illustra questo flusso transazionale distribuito.

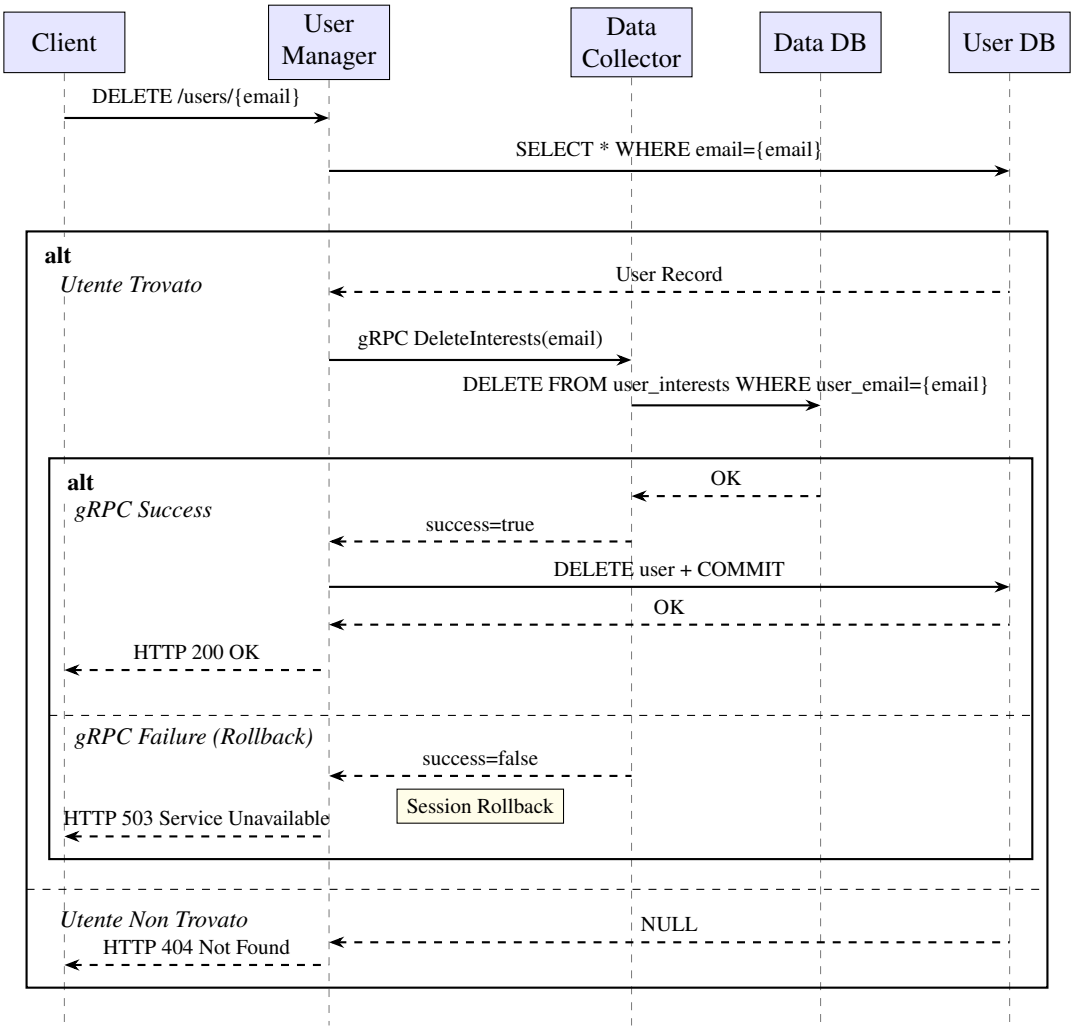

Figura 6: Diagramma di sequenza: DELETE /users/{email}

5.5 Data Collector: POST /interests

L'aggiunta di un nuovo aeroporto di interesse richiede una validazione preventiva dell'identità dell'utente presso User Manager. La Figura 7 mostra come il Data Collector verifichi prima l'esistenza dell'utente via gRPC e successivamente controlli localmente che l'interesse non sia già stato registrato.

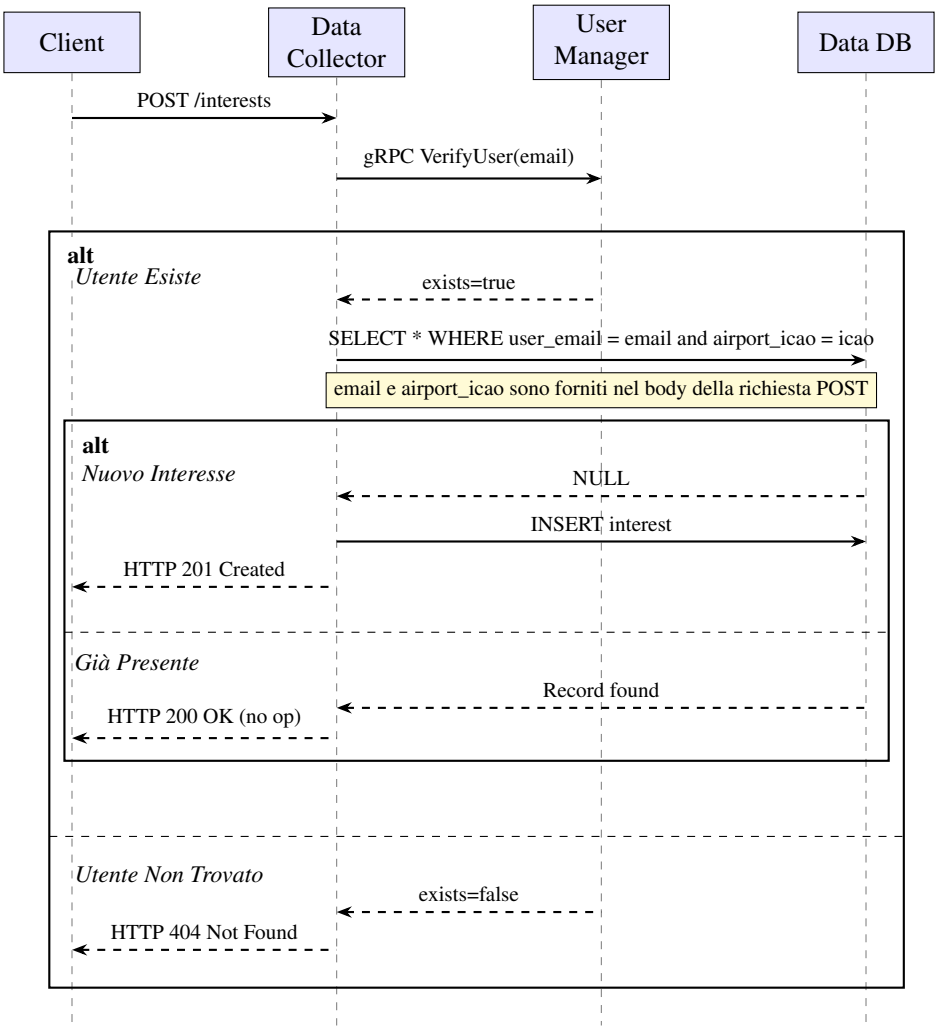

Figura 7: Diagramma di sequenza: POST /interests

5.6 Data Collector: GET /interests/{email}

Per recuperare la lista degli interessi di un utente, il Data Collector deve prima validare l'identità dell'utente tramite il servizio User Manager. La Figura 8 illustra come la verifica gRPC preceda l'interrogazione locale del database degli interessi.

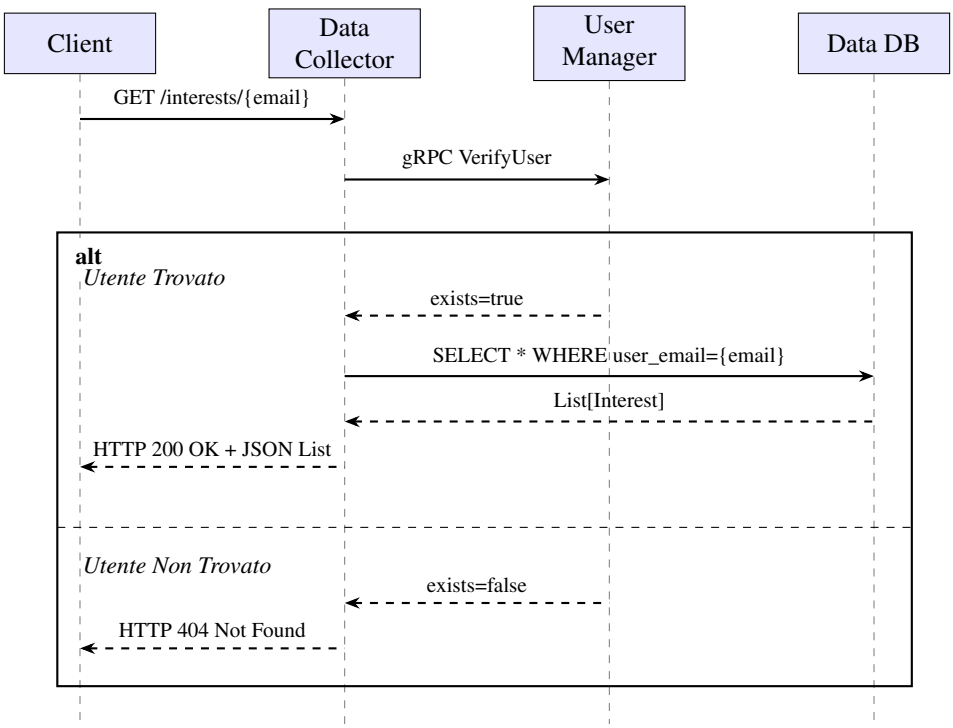

Figura 8: Diagramma di sequenza: GET /interests/{email}

5.7 Data Collector: DELETE /interests

La Figura 9 illustra la rimozione di un interesse di un utente.

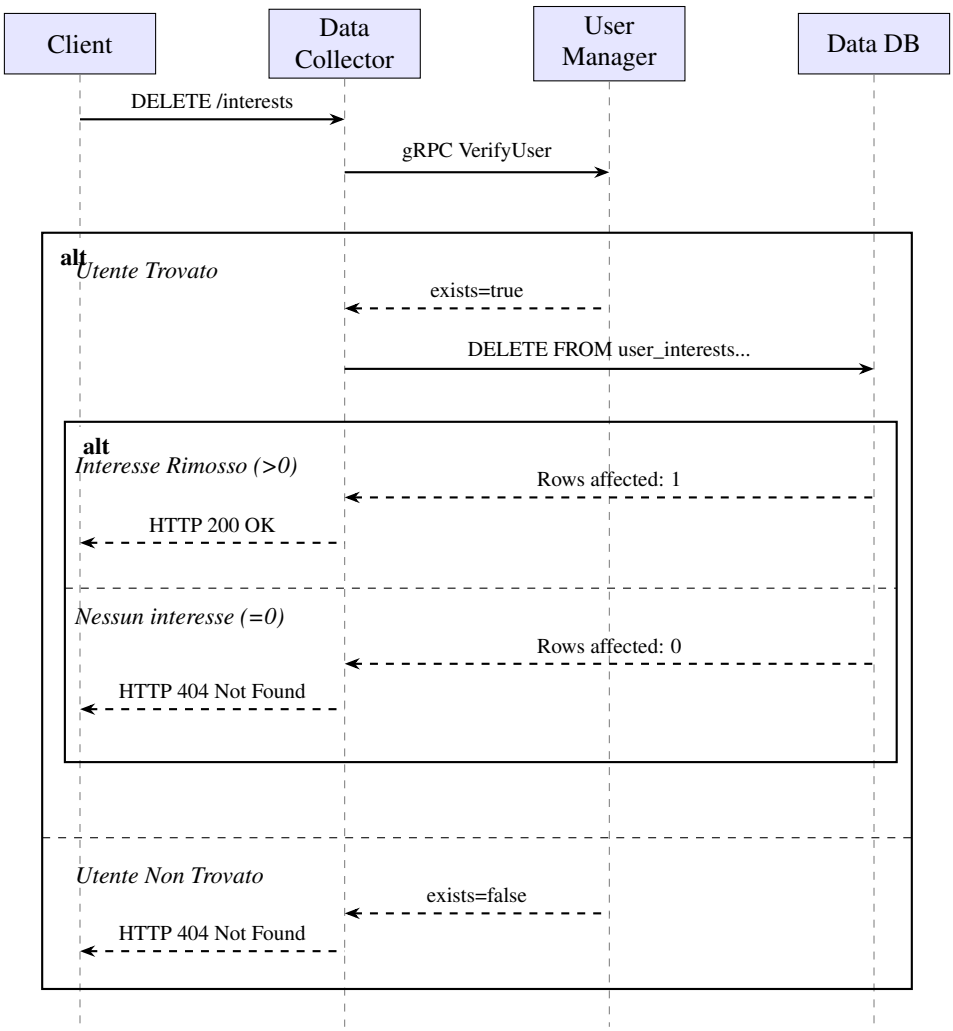

Figura 9: Diagramma di sequenza: DELETE /interests

5.8 Data Collector: GET /flights/icao (Ricerca Generale)

Questo endpoint permette di consultare lo storico dei voli applicando filtri specifici come intervalli temporali e tipologia di volo (partenza/arrivo). Come mostrato in Figura 10, il sistema verifica prima l'utente via gRPC, poi controlla che l'aeroporto sia tra gli interessi dell'utente, e infine esegue la query filtrata.

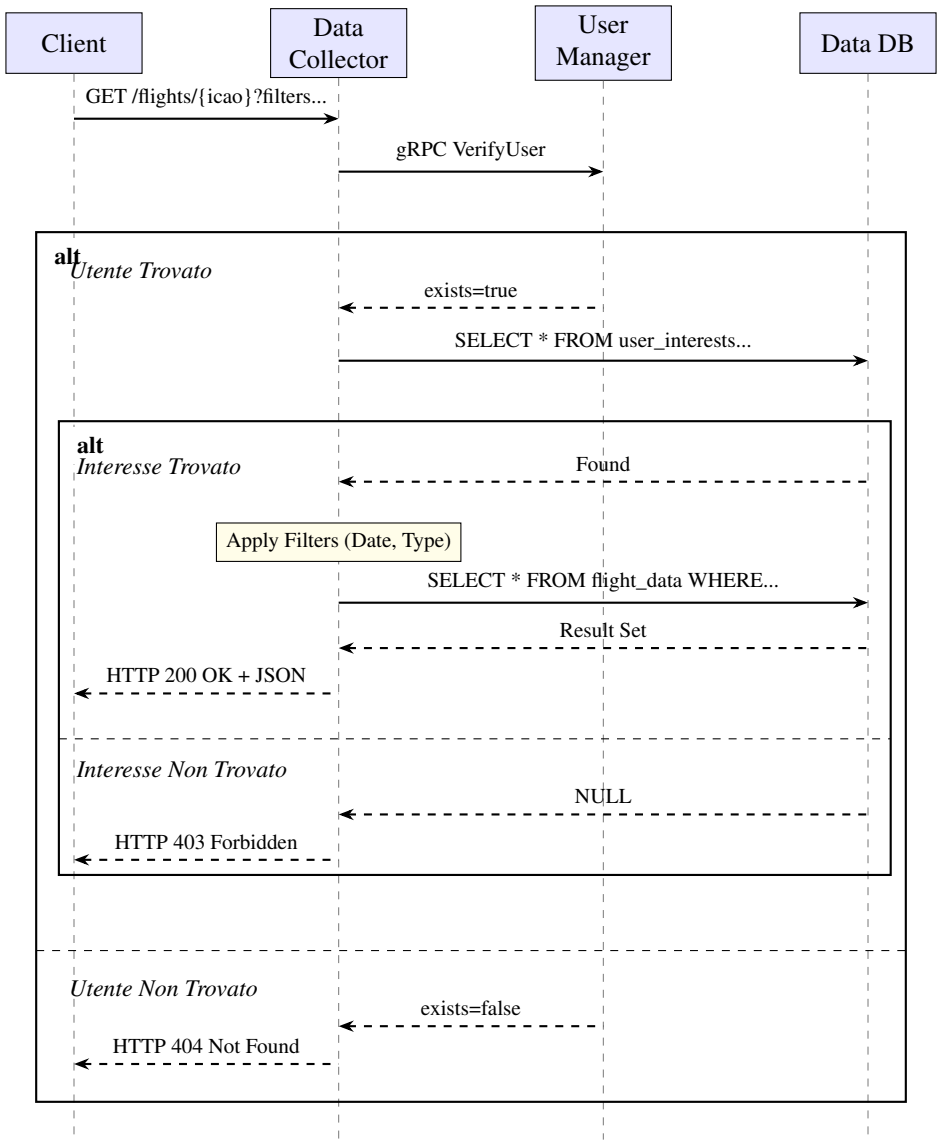

Figura 10: Diagramma di sequenza: GET /flights/{icao}

5.9 Data Collector: GET /flights/{icao}/latest

Questo endpoint permette di ottenere rapidamente i dati del volo più recente registrato per un determinato aeroporto, distinguendo opzionalmente tra arrivi e partenze. Come illustrato in Figura 11, il sistema esegue la verifica utente via gRPC e controlla i permessi di accesso ai dati dell'aeroporto prima di restituire il risultato o un errore 404 se nessun volo è disponibile.

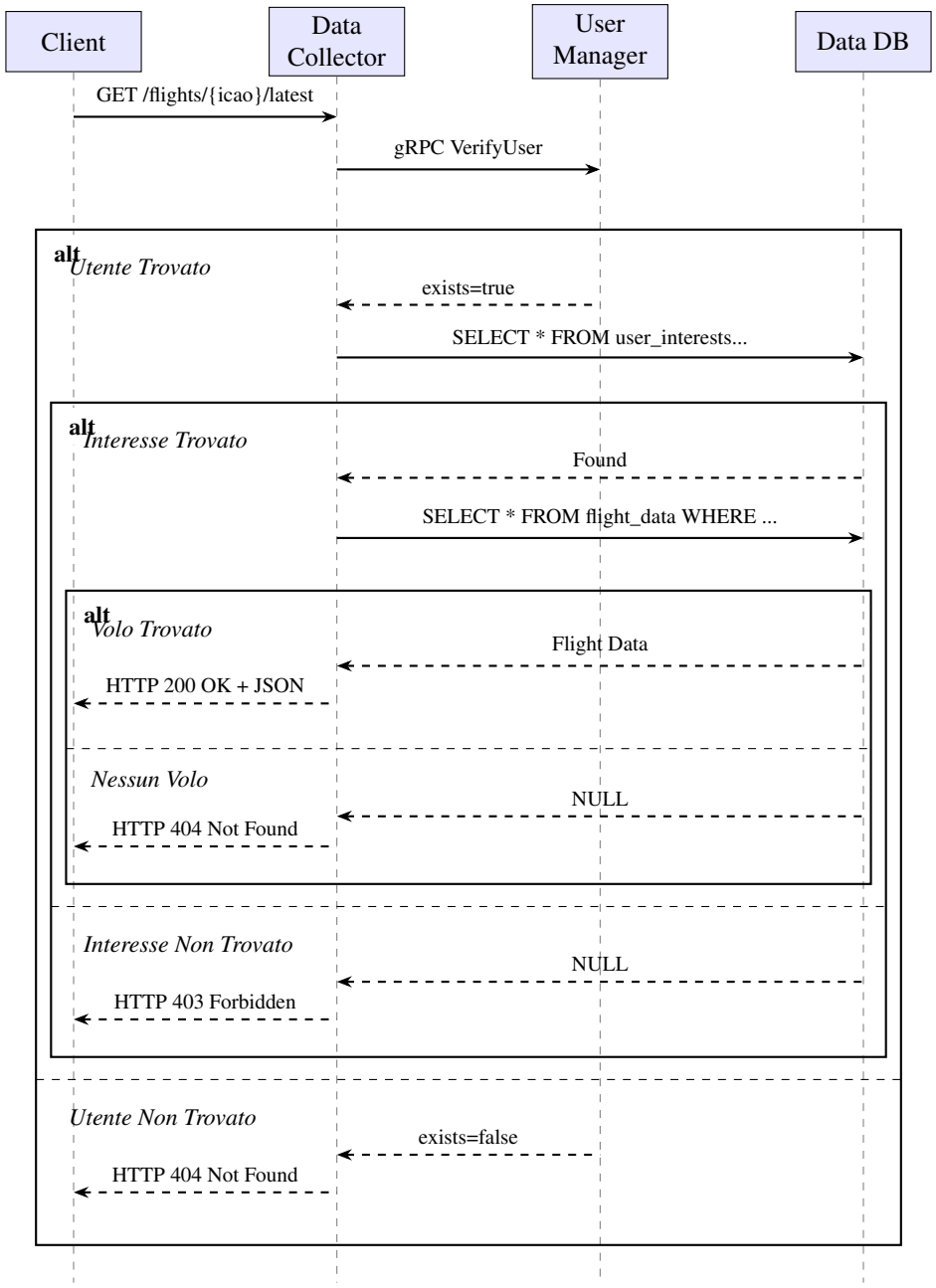

Figura 11: Diagramma di sequenza: GET /flights/{icao}/latest

5.10 Data Collector: Raccolta Periodica (Scheduler)

La Figura 12 illustra il flusso del job di raccolta dati periodico, che interroga l'API OpenSky per raccogliere i dati di tutti gli aeroporti monitorati. Tale job è eseguito dal componente *Scheduler* (un thread interno al Data Collector) che viene attivato a intervalli regolari. In particolare, al momento è stato deciso di eseguire

il job ogni 12 ore chiedendo ad OpenSky i dati delle ultime 24 ore, in modo tale da avere la certezza di non perdere dati di voli precedenti che non erano ancora stati registrati interamente (ad esempio a causa di voli in ritardo o di lunga durata). Un aspetto cruciale di questo processo è la fase di salvataggio dei dati, che non avviene singolarmente per ogni volo ma sfrutta una tecnica di **Bulk Upsert**. Questa ottimizzazione, fondamentale per le performance del sistema e per ridurre il carico sul database, viene analizzata nel dettaglio tecnico nel paragrafo 7.3.

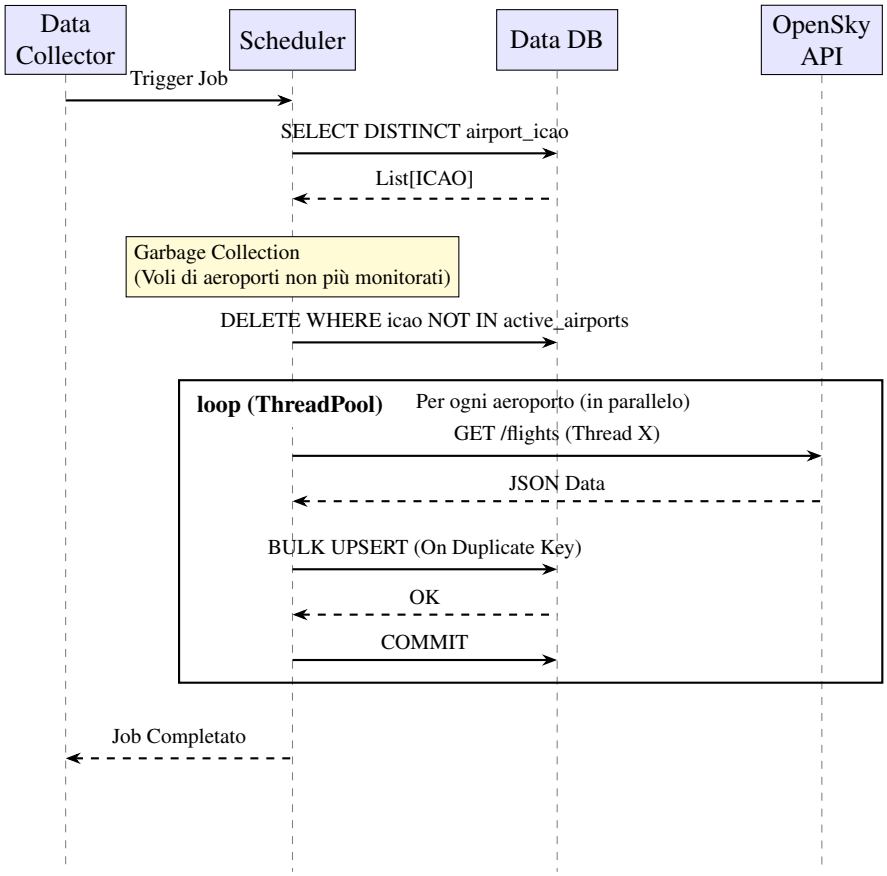

Figura 12: Diagramma di sequenza: Job di raccolta dati periodica

5.11 Data Collector: GET /flights/{icao}/stats/airlines

La Figura 13 illustra il calcolo delle statistiche sulle compagnie aeree. Per semplificare l'esempio, si assume che l'utente esista (sempre verificato via gRPC) e che abbia i permessi per accedere ai dati dell'aeroporto richiesto (ha l'interesse registrato).

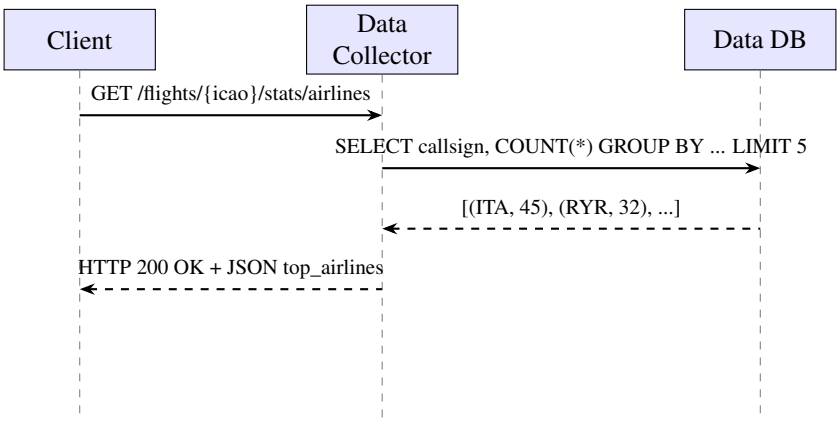

Figura 13: Diagramma di sequenza: GET /flights/{icao}/stats/airlines

5.12 Data Collector: GET /flights/{icao}/average

La Figura 14 illustra il calcolo della media giornaliera dei voli. Anche in questo caso si assume che l'utente esista e abbia i permessi per accedere ai dati dell'aeroporto richiesto. Inoltre, per questo endpoint il client può specificare sia un tipo di volo (arrivi/partenze), sia un numero di giorni su cui basare il calcolo; in assenza di questi filtri, il calcolo viene effettuato su una settimana di dati e le due medie vengono restituite separatamente.

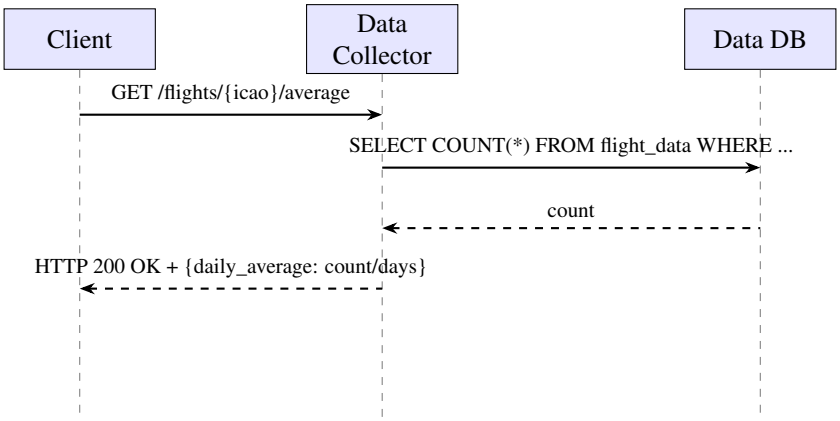

Figura 14: Diagramma di sequenza: GET /flights/{icao}/average

5.13 Data Collector: POST /collect/manual

Per scopi di test e debug, è possibile forzare l'esecuzione immediata del job di raccolta dati senza attendere l'intervallo programmato. Come illustrato in Figura 15, questa operazione viene eseguita in modalità sincrona: il client riceverà una risposta solo al termine dell'intero ciclo di aggiornamento (chiamate OpenSky e salvataggio DB).

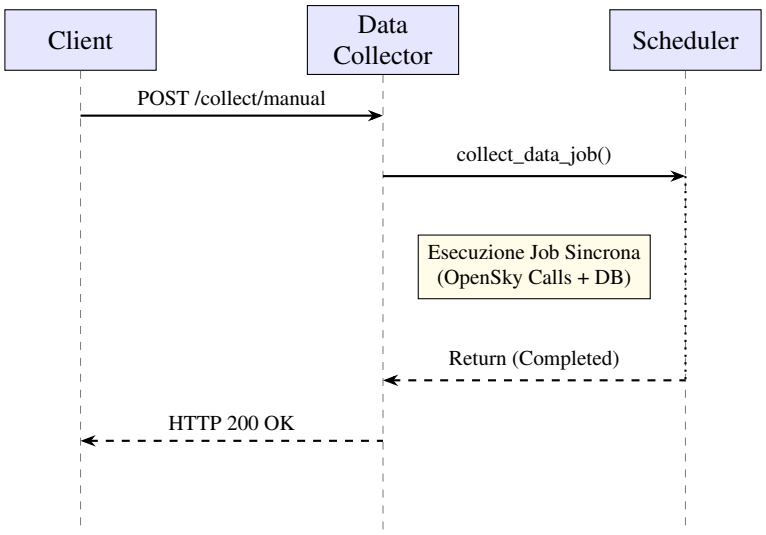

Figura 15: Diagramma di sequenza: POST /collect/manual

6 Comunicazione Inter-Service con gRPC

La comunicazione tra i microservizi è realizzata mediante gRPC, un framework RPC ad alte prestazioni sviluppato da Google. Questa sezione descrive le scelte implementative e le configurazioni adottate.

6.1 Configurazione Retry Policy

I client gRPC sono configurati con una policy di retry che gestisce automaticamente gli errori momentanei di connettività, evitando che instabilità temporanee della rete causino fallimenti delle operazioni:

```
service_config = {
    "methodConfig": [{
        "name": [{"service": "user_service.UserService"}],
        "retryPolicy": {
            "maxAttempts": 5,
            "initialBackoff": "1s",
            "maxBackoff": "5s",
            "backoffMultiplier": 2,
            "retryableStatusCodes": ["UNAVAILABLE"]
        },
        "timeout": "10s"
    }]
}

options = [('grpc.service_config', json.dumps(service_config))]
channel = grpc.insecure_channel(target, options=options)
```

Listing 6: Configurazione service config per gRPC

La configurazione prevede un massimo di 5 tentativi con backoff esponenziale: il primo retry avviene dopo 1 secondo, i successivi raddoppiano fino a un massimo di 5 secondi. Solo gli errori con status code `UNAVAILABLE` (tipici di problemi di rete temporanei) vengono ritentati automaticamente.

6.2 Digressione su transazione distribuita

La cancellazione di un utente richiede il coordinamento tra User Manager e Data Collector per garantire la consistenza dei dati. L'implementazione adotta una strategia "All or Nothing" simulando un approccio a due fasi: l'utente viene rimosso definitivamente dal database locale solo se la contestuale cancellazione degli interessi sul servizio remoto ha successo.

```python
@app.route('/users/<email>', methods=['DELETE'])
def delete_user(email):
    clean_email = email.strip().lower()
    user = db.session.get(User, clean_email)

    if not user:
        return jsonify({"error": "Utente non trovato"}), 404

    try:
        # FASE 1: Preparazione locale
        # Marchiamo l'utente per l'eliminazione nella sessione, ma NON committiamo ancora.
        db.session.delete(user)

        # FASE 2: Coordinamento Remoto
        # Verifichiamo che il Data Collector elimini i dati associati.
        grpc_success, grpc_msg = data_collector_client.delete_interests(clean_email)

        if not grpc_success:
            # Fallimento remoto -> Rollback locale
            # Annulliamo la cancellazione in stato pending.
            db.session.rollback()
            return jsonify({
                "error": "Impossibile completare: errore Data Collector",
                "details": grpc_msg
            }), 503

        # FASE 3: Commit Finale
        # Il remoto ha confermato, rendiamo permanente la modifica locale.
        db.session.commit()

        return jsonify({
            "message": "Utente eliminato con successo",
            "data_cleanup": "Completed"
        }), 200

    except Exception:
        db.session.rollback()
        raise
```

Listing 7: Coordinamento transazionale nella cancellazione utente

7 Data Collector: Ottimizzazioni

Il microservizio Data Collector implementa diverse ottimizzazioni per garantire efficienza e scalabilità nella raccolta e persistenza dei dati di volo. Questa sezione descrive le principali tecniche adottate.

7.1 Client OpenSky con OAuth2 Thread-Safe

Il client per l'API OpenSky implementa l'autenticazione OAuth2 Client Credentials con gestione thread-safe del token mediante il pattern Double-Check Locking. Questo pattern garantisce che il token venga rinnovato una sola volta anche in presenza di richieste concorrenti che rilevano simultaneamente la scadenza:

```
class OpenSkyClient:
    def __init__(self):
        self.token = None
        self.token_expiry = 0
        self._auth_lock = threading.Lock()

    def get_headers(self):
        if self._is_token_expired():
            with self._auth_lock:
                if self._is_token_expired():  # Re-check dopo acquisizione lock
                    self._perform_login()
        return {'Authorization': f'Bearer {self.token}'}
```

Listing 8: Gestione token OAuth2 thread-safe

7.2 Parallelizzazione con ThreadPoolExecutor

La raccolta dei dati da più aeroporti viene parallelizzata utilizzando un ThreadPoolExecutor con un massimo di 5 worker concorrenti. Questo approccio riduce significativamente il tempo totale di raccolta quando sono monitorati molti aeroporti:

```python
def collect_data_job(self):
    with self.app.app_context():
        airports = db.session.execute(
            select(UserInterest.airport_icao).distinct()
        ).scalars().all()

        with ThreadPoolExecutor(max_workers=5) as executor:
            future_to_icao = {
                executor.submit(self._fetch_airport_data, icao): icao
                for icao in airports
            }
            for future in as_completed(future_to_icao):
                airport_icao, flight_data = future.result()
                self._save_flights(airport_icao, flight_data)
```

Listing 9: Raccolta parallela dei dati

7.3 Bulk Upsert con ON DUPLICATE KEY UPDATE

La gestione della persistenza dei dati di volo rappresenta una delle sfide prestazionali più critiche del sistema. Dato che una singola chiamata all'API OpenSky può restituire centinaia di voli per un dato aeroporto, un approccio iterativo tradizionale (una `INSERT` per ogni volo) risulterebbe molto inefficace a causa dell'elevato numero di dati da gestire e dell'overhead transazionale.
In un approccio non ottimizzato, infatti, salvare N voli richiederebbe di comunicare N volte con il database. Inoltre, per evitare duplicati, il sistema dovrebbe prima verificare l'esistenza del record e poi decidere se inserire o aggiornare, portando la complessità delle operazioni a $O(N)$.
Per risolvere questo problema, il Data Collector implementa una strategia di Bulk Upsert (Update or Insert) sfruttando SQLAlchemy. L'algoritmo implementato nel metodo `_save_flights` riduce drasticamente il numero di interazioni con il database agendo in tre fasi:

1. **Preparazione dei dati**: I dati ricevuti dall'API vengono pre-processati e accumulati in una lista di dizionari in memoria.

2. **Preparazione della query**: Viene costruita una singola istruzione SQL `INSERT` contenente tutti i record del batch di dati corrente.

3. **Inserimento**: Viene utilizzata la clausola nativa di MySQL `ON DUPLICATE KEY UPDATE`: in questo modo, il database tenta l'inserimento e se rileva una violazione del vincolo di unicità sulla chiave composta `(icao24, first_seen, airport_icao)`, esegue automaticamente un aggiornamento dei campi specificati (es. `last_seen`) invece di generare un errore.

Questa tecnica permette di ridurre la complessità dell'operazione da $O(N)$ a $O(1)$ per ogni aeroporto elaborato, garantendo al contempo l'atomicità dell'operazione per l'intero batch.

```python
from sqlalchemy.dialects.mysql import insert

# Preparazione dati (semplificata per chiarezza)
insert_values = [{
    "airport_icao": airport_icao,
    "icao24": flight.get('icao24'),
    "first_seen": flight.get('firstSeen'),
    "last_seen": flight.get('lastSeen'),
    "callsign": flight.get('callsign'),
    "flight_type": flight_type,
    # ... altri campi (distanze, aeroporti stimati, ecc.)
    "collected_at": datetime.now()
} for flight in flight_data_list if flight.get('icao24')]

# Costruzione query
query = insert(FlightData).values(insert_values)

on_duplicate_key_query = query.on_duplicate_key_update(
    last_seen=query.inserted.last_seen,
    callsign=query.inserted.callsign,
    est_arrival_airport=query.inserted.est_arrival_airport,
    # ... aggiornamento altri campi dinamici
    collected_at=datetime.now()
)

# Esecuzione atomica
db.session.execute(on_duplicate_key_query)
db.session.commit()
```

Listing 10: Operazione di bulk upsert (estratto)

7.4 Garbage Collection dei Dati Obsoleti

All'inizio di ogni ciclo di raccolta, un processo di garbage collection rimuove automaticamente i dati relativi ad aeroporti non più monitorati da alcun utente. Questo meccanismo previene l'accumulo indefinito di dati inutilizzati:

```
active_airports = db.session.execute(
    select(UserInterest.airport_icao).distinct()
).scalars().all()

if not active_airports:
    db.session.execute(delete(FlightData))
else:
    db.session.execute(
        delete(FlightData).where(
            not_(FlightData.airport_icao.in_(active_airports))
        )
    )
db.session.commit()
```

Listing 11: Garbage collection dei dati

8 Schema del Database

Il sistema utilizza due database MySQL 8.0 completamente isolati, ciascuno accessibile esclusivamente dal proprio microservizio. Questa separazione implementa il pattern "Database per Service", garantendo autonomia e isolamento dei dati.

8.1 User Database (Porta 3306)

Il database degli utenti contiene due tabelle: `users` per i dati anagrafici e le coordinate bancarie e `request_cache` per l'implementazione della politica At-Most-Once.

Tabella 4: Struttura tabella users

Colonna	Tipo	Vincoli	Descrizione
email	VARCHAR(255)	PRIMARY KEY	Funge da identificativo univoco
nome	VARCHAR(100)	NOT NULL	Nome dell'utente
cognome	VARCHAR(100)	NOT NULL	Cognome dell'utente
codice_fiscale	VARCHAR(16)	UNIQUE, NOT NULL	Codice Fiscale italiano
iban	VARCHAR(500)	NOT NULL	IBAN cifrato (Fernet)
iban_hash	VARCHAR(64)	UNIQUE, NOT NULL	Hash SHA-256 per unicità
data_registrazione	DATETIME	DEFAULT NOW()	Timestamp di registrazione

Tabella 5: Struttura tabella request_cache

Colonna	Tipo	Vincoli	Descrizione
id	VARCHAR(64)	PRIMARY KEY	Hash chiave di idempotenza
response_body	TEXT	NOT NULL	JSON della risposta cached
response_code	INTEGER	NOT NULL	HTTP Status Code (es. 201)
created_at	DATETIME	DEFAULT NOW()	Timestamp per calcolo TTL

8.2 Data Database (Porta 3307)

Il database dei dati di volo contiene le tabelle `user_interests` e `flight_data`.

Tabella 6: Struttura tabella user_interests

Colonna	Tipo	Vincoli	Descrizione
id	INTEGER	PK, AUTO_INCREMENT	Identificativo univoco
user_email	VARCHAR(255)	NOT NULL	Email utente
airport_icao	VARCHAR(10)	NOT NULL	Codice ICAO aeroporto
created_at	DATETIME	DEFAULT NOW()	Timestamp di creazione
UNIQUE(user_email, airport_icao) - Constraint per unicità			

Tabella 7: Struttura tabella flight_data

Colonna	Tipo	Vincoli	Descrizione
id	INTEGER	PK, AUTO_INCREMENT	Identificativo univoco
airport_icao	VARCHAR(10)	NOT NULL, INDEX	Aeroporto di riferimento
icao24	VARCHAR(50)	NOT NULL	Transponder ID
first_seen	BIGINT	NOT NULL	Primo avvistamento (UNIX)
last_seen	BIGINT	NULL	Ultimo avvistamento (UNIX)
est_departure_airport	VARCHAR(10)	NULL	Aeroporto partenza stimato
est_arrival_airport	VARCHAR(10)	NULL	Aeroporto arrivo stimato
callsign	VARCHAR(20)	NULL	Identificativo volo
est_departure_airport_horiz_distance	INTEGER	NULL	Distanza orizz. partenza
est_departure_airport_vert_distance	INTEGER	NULL	Distanza vert. partenza
est_arrival_airport_horiz_distance	INTEGER	NULL	Distanza orizz. arrivo
est_arrival_airport_vert_distance	INTEGER	NULL	Distanza vert. arrivo
departure_airport_candidates_count	INTEGER	NULL	Num. candidati partenza
arrival_airport_candidates_count	INTEGER	NULL	Num. candidati arrivo
flight_type	VARCHAR(20)	NOT NULL	“departure” o “arrival”
collected_at	DATETIME	DEFAULT NOW()	Timestamp raccolta dati
UNIQUE(icao24, first_seen, airport_icao) - Chiave per Upsert			

9 Possibili Estensioni Future

Il sistema è stato progettato con l’estensibilità in mente. Alcune evoluzioni possibili includono:

- **API Gateway con Nginx**: introduzione di un reverse proxy per fornire un unico punto di accesso scalabile al sistema, centralizzando routing, load balancing e sicurezza.
- **Autenticazione JWT**: implementazione di token JWT per proteggere gli endpoint pubblici e gestire le sessioni utente.
- **Cache Redis**: introduzione di una cache tramite Redis per sostituire l’attuale sistema di caching, riducendo il carico sui database e aumentando le performance.

A Reference Endpoints e gRPC

A.1 User Manager Endpoints (Porta 5000)

Tabella 8: Endpoint REST del User Manager

Metodo	Endpoint	Descrizione
POST	/users	Registrazione utente (At-Most-Once)
GET	/users	Lista tutti gli utenti
GET	/users/{email}	Recupero singolo utente
DELETE	/users/{email}	Cancellazione con coordinamento gRPC
GET	/users/verify/{email}	Verifica esistenza utente
GET	/health	Health check del servizio

A.2 Data Collector Endpoints (Porta 5001)

Tabella 9: Endpoint REST del Data Collector

Metodo	Endpoint	Descrizione
POST	/interests	Aggiunta interesse
GET	/interests/{email}	Lista interessi di un utente
DELETE	/interests	Rimozione interesse
GET	/flights/{icao}	Lista voli per aeroporto
GET	/flights/{icao}/latest	Ultimo volo registrato
GET	/flights/{icao}/average	Media giornaliera voli
GET	/flights/{icao}/stats/airlines	Top 5 compagnie aeree
POST	/collect/manual	Trigger manuale raccolta
GET	/scheduler/status	Stato dello scheduler
GET	/health	Health check del servizio

A.3 Servizi gRPC

Tabella 10: Metodi gRPC esposti

Servizio	Porta	Metodo	Descrizione
UserService	50051	VerifyUser	Verifica esistenza utente
UserService	50051	GetUser	Recupero dati utente (per possibili estensioni future)
DataCollectorService	50052	DeleteInterests	Cancellazione interessi