Progetto #1 – Prova in Itinere: Relazione

Venezia Vincenzo
Corso di Ingegneria Informatica LM32 UNICT
Data 02-12-2025

1. Introduzione al progetto

1.1. Obiettivo del progetto

Il progetto consiste nello sviluppo di un sistema distribuito a microservizi, dockerizzato, finalizzato alla gestione di utenti e al recupero di dati sui voli aerei tramite le API di OpenSky Network. In particolare, il sistema deve supportare le seguenti funzionalità principali:

- **Gestione degli utenti**: Consente la registrazione e la cancellazione degli utenti.
- **Recupero e monitoraggio dei dati sui voli**: Recupera informazioni sui voli in partenza e in arrivo dagli aeroporti di interesse dell'utente.
- **Elaborazione dei dati**: Include la possibilità di calcolare statistiche come la media dei voli negli ultimi X giorni per ciascun aeroporto.

1.2. Architettura e componenti

l sistema è suddiviso in due microservizi principali:

- **User Manager**: Responsabile della gestione degli utenti. Permette di registrare, cancellare e monitorare gli utenti tramite un database relazionale. Comunica con il **Data Collector** tramite gRPC per verificare l'esistenza di un utente prima di elaborare o recuperare i dati sui voli.

- **Data Collector**: Si occupa di raccogliere i dati sui voli aerei da OpenSky Network e memorizzarli in un database. Gestisce anche la logica di calcolo della media dei voli negli ultimi X giorni e fornisce endpoint per recuperare informazioni sui voli in partenza e arrivo.

1.3. Scelte tecnologiche

Nel seguente progetto sono state adottate le seguenti tecnologie:

- **Docker e Docker Compose**: Per la containerizzazione e l'orchestrazione dei microservizi. Questo approccio garantisce che ogni componente del sistema venga eseguito in un ambiente isolato.

- **gRPC**: Un framework che consente la comunicazione tra i microservizi (User Manager e Data Collector).

- **Flask**: Un micro-framework Python per lo sviluppo delle API RESTful. Flask è stato scelto per la sua leggerezza e la facilità d'uso.

- **PostgreSQL**: Un database relazionale scelto per la gestione dei dati sugli utenti e sui voli.

- **OpenSky Network API**: Per ottenere i dati sui voli in tempo reale, inclusi i voli in partenza e in arrivo da aeroporti specifici.

2. Struttura e Impostazione del Progetto

2.1. Introduzione al secondo Capitolo

In questo capitolo viene presentata la struttura organizzativa del progetto, descrivendo la disposizione dei vari componenti e dei file all'interno del repository. Il sistema è basato su una **struttura a microservizi** ed è progettato per essere facilmente dockerizzabile tramite l'uso di **Docker** e **Docker Compose**. Il progetto è suddiviso in due principali microservizi: **User Manager** e **Data Collector**, ognuno dei quali ha un'implementazione indipendente.

2.2. Architettura del Sistema

L'architettura del sistema è composta da due microservizi principali:

1. **User Manager**: Gestisce la registrazione, la cancellazione e la gestione degli utenti.

2. **Data Collector**: Si occupa di raccogliere e monitorare i dati sui voli tramite l'API di OpenSky Network, e fornisce delle funzionalità aggiuntive come il calcolo della media dei voli per aeroporto.

I microservizi comunicano tra loro tramite **gRPC** e utilizzano un database per memorizzare le informazioni sugli utenti e i voli. Ogni componente è gestito come un container Docker separato.

2.3. Directory di Progetto

La struttura del progetto, incluse le varie directory, è stata organizzata come segue:

progetto_dsbd/

│ ├── docker-compose.yaml

│ ├──.gitignore

├── user_manager/

│ ├── Dockerfile

│ ├── requirements.txt

│ └── app/

│ ├── __init__.py

│ ├── database.py

│ ├── grpc_service.py

│ ├── main.py

│ ├── user_pb2_grpc.py

│ ├── user_pb2.py

│ ├── proto/

│ │ └── user.proto

├── data_collector/

│ ├── Dockerfile

│ ├── requirements.txt

│ └── app/

│ ├── __init__.py

│ ├── database.py

│ ├── grpc_service.py

│ ├── main.py

│ ├── scheduler.py

│ ├── opensky.py

│ ├── user_pb2_grpc.py

│ ├── user_pb2.py

│ ├── proto/

│ │ └── user.proto

2.4. Comunicazione fra Servizi

La comunicazione tra **User Manager** e **Data Collector** avviene tramite **gRPC**. Quando il **Data Collector** ha bisogno di verificare se un utente esiste (ad esempio, quando un aeroporto viene aggiunto o rimosso), effettua una richiesta gRPC al **User Manager**, che risponde con l'esito della verifica.

Entrambi i microservizi sono containerizzati utilizzando **Docker** e orchestrati tramite **Docker Compose**; il file *docker-compose.yaml* definisce i servizi, le reti e i volumi necessari per eseguire i container.

2.5. Configurazione del Database

Ogni microservizio si connette a un database (PostgreSQL) per memorizzare e recuperare i dati, dove i database sono stati implementati a loro volta come container separati:

- **User Manager** utilizza il database per memorizzare informazioni sugli utenti e le richieste processate.
- **Data Collector** utilizza un database per memorizzare i dati sui voli e sugli aeroporti.

2.6. File di Configurazione

I file di configurazione usati per il corretto sviluppo del progetto sono:

- **requirements.txt**: Ogni microservizio ha il proprio file requirements.txt per gestire le dipendenze Python. Ad esempio, per il **User Manager**, potrebbero esserci librerie come Flask, psycopg2 (per PostgreSQL) e grpcio. Il **Data Collector** potrebbe avere anche dipendenze per **requests** (per interagire con OpenSky) e **apscheduler** (per la pianificazione delle operazioni).
- **Dockerfile**: Ogni microservizio ha un proprio **Dockerfile** per creare l'immagine Docker. I Dockerfile includono l'installazione delle dipendenze e la configurazione dell'ambiente di esecuzione.
- **docker-compose.yaml**: Il file docker-compose.yaml è essenziale per avviare e gestire i microservizi in modo coordinato. Con Docker Compose, è possibile definire e configurare l'infrastruttura di contenitori per l'intero sistema, descrivendo come i vari servizi (User Manager, Data Collector, e i relativi database) interagiscono tra loro.

  Il file **docker-compose.yaml** definisce i seguenti aspetti principali:

- **Servizi**:
    1. **User Manager**: Viene configurato per costruire l'immagine dalla directory user_manager e mappa le porte necessarie. Inoltre, dipende dal database userdb, che deve essere avviato prima del servizio.
    2. **Data Collector**: Viene configurato in modo simile, ma dipende sia dal database datadb che dal user_manager (per la comunicazione tramite gRPC).

3. **User DB** e **Data DB**: Entrambi sono configurati per utilizzare l'immagine ufficiale di PostgreSQL. Ogni servizio di database ha una sua configurazione (utente, password e database) e l'accesso è limitato alle rispettive applicazioni (user_manager per userdb e data_collector per datadb).

- **Volumi**:

  I volumi Docker sono definiti per persistere i dati dei database anche in caso di riavvio dei container. In questo caso, userdb e datadb vengono utilizzati per archiviare i dati persistenti all'interno dei database PostgreSQL.

2.7. Avvio del progetto

Per avviare il progetto con docker-compose, basta eseguire il comando nella root del progetto: `docker-compose up –build`

3. Microservizi: Endpoint e Implementazione API

Di seguito si riportano i microservizi, di cui si è mantenuta l'architettura proposta nell'assegnazione del progetto.

3.1. Servizio UserManager

Il microservizio **User Manager** è responsabile della gestione degli utenti. Le sue funzionalità principali sono:

- Registrazione di nuovi utenti.
- Cancellazione di utenti esistenti.
- Verifica dello stato delle richieste per implementare la politica **at-most-once**.
- Comunicazione con altri microservizi tramite **gRPC** per la verifica dell'esistenza di un utente.

Il servizio è sviluppato in **Python** con **Flask** per le API REST e utilizza **PostgreSQL** come database relazionale per la persistenza dei dati.

3.1.1 At-Most-One-Policy

Il microservizio implementa la politica **at-most-once** per evitare di eseguire più volte la stessa operazione in caso di retry da parte del client.
Il meccanismo funziona nel seguente modo:

1. **Richieste identificate da request_id**: Ogni richiesta inviata al server include un identificativo univoco request_id.

2. **Log delle richieste processate**: Il database mantiene una tabella processed_requests in cui viene memorizzato ogni request_id già processato.

3. **Verifica prima dell'esecuzione**: Prima di registrare o cancellare un utente, il servizio controlla se il request_id esiste già nel log tramite la funzione:

```python
# -------- AT-MOST-ONCE ----------
def get_request_log(conn, request_id):
    cur = conn.cursor()
    cur.execute("SELECT request_id FROM processed_requests WHERE request_id = %s", (request_id,))
    row = cur.fetchone()
    cur.close()
    return row is not None
```

4. **Esecuzione condizionata:** Se il `request_id` è presente, la richiesta viene considerata già processata e il server risponde con `{"status": "already_processed"}`.

   Altrimenti, l'operazione viene eseguita normalmente e il `request_id` viene inserito nella tabella `processed_requests` per registrare l'avvenuto processing.

3.1.2. API

1.Registrazione utente (POST /register)

- **Funzione**: registra un nuovo utente se non esiste già, altrimenti restituisce le informazioni esistenti.

- **Flusso interno**:

  1. Controllo request_id per policy at-most-once.

  2. Verifica se l'utente esiste nel database.

  3. Inserimento nel database se non esiste.

4. Registrazione del request_id nella tabella processed_requests.

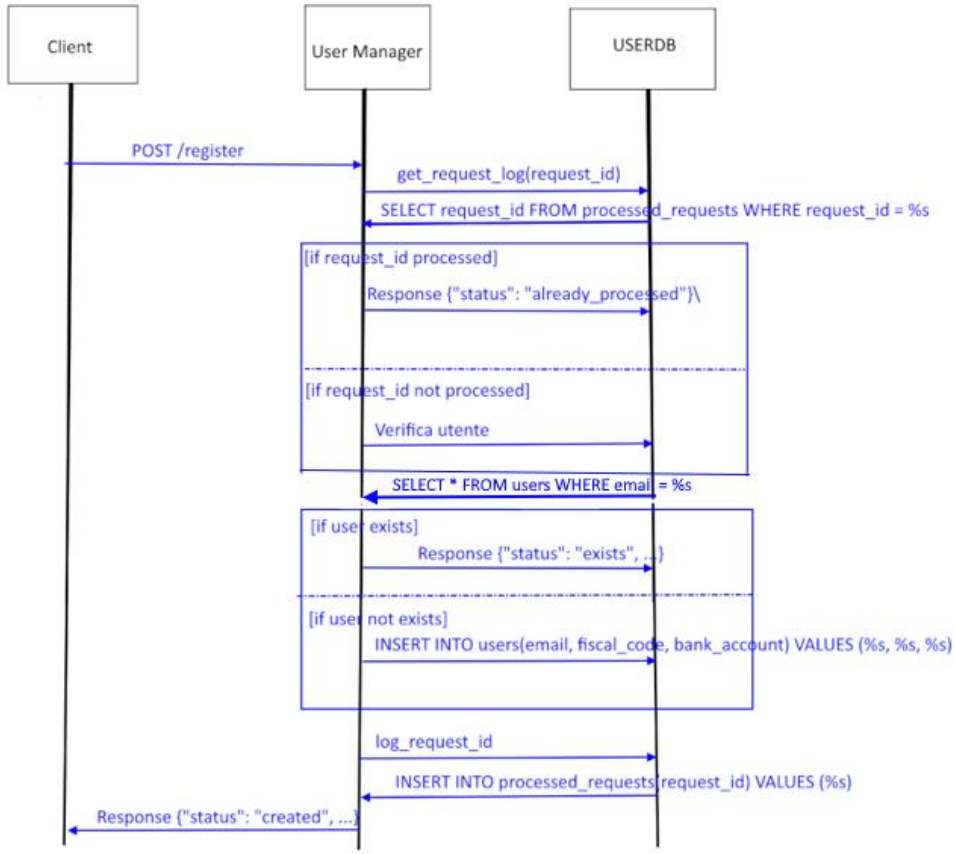
- **Response**:
    - Creazione: {"status": "created", ...}
    - Esistente: {"status": "exists", ...}
    - Richiesta già processata: {"status": "already_processed"}

2.Cancellazione utente (DELETE /user/<email>)

- **Funzione**: cancella un utente esistente.
- **Flusso interno**:
    1. Controllo request_id per policy at-most-once.
    2. Verifica se l'utente esiste.
    3. Cancellazione dal database.
    4. Registrazione del request_id nella tabella processed_requests.
- **Response**:
    - Cancellazione riuscita: {"status": "deleted", "email": ...}
    - Utente non trovato: {"status": "not_found", "email": ...}
    - Richiesta già processata: {"status": "already_processed"}

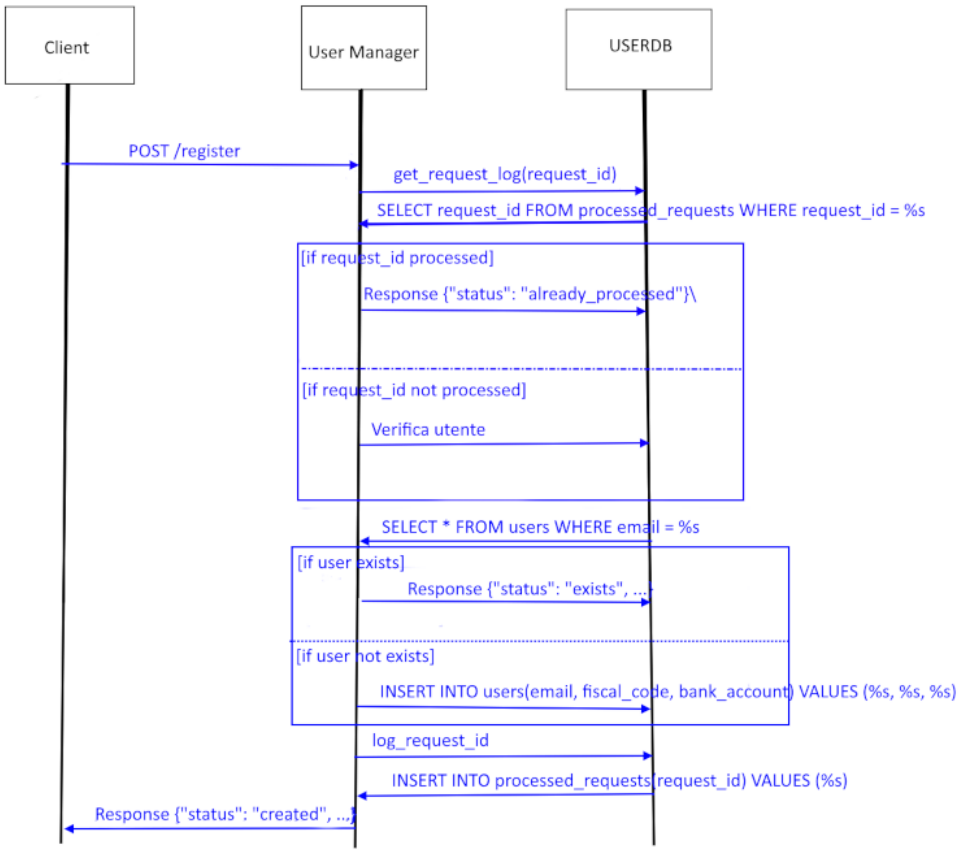
3.1.3. Diagrammi di Sequenza API

1) Diagramma di sequenza POST/register



.

2) Diagramma di sequenza DELETE/user/[email]



3.2. Servizio DataCollector

Il microservizio **Data Collector** si occupa della gestione dei dati sui voli e degli aeroporti associati agli utenti. Le funzionalità principali sono:

- Gestione aeroporti per utente: aggiunta, rimozione e lista degli aeroporti di interesse.
- Recupero dei voli più recenti per un aeroporto specifico.
- Calcolo della media dei voli negli ultimi X giorni.
- Comunicazione con **User Manager** tramite gRPC per verificare l'esistenza di un utente.
- Scheduler interno (tramite APScheduler) per eventuali task periodici di raccolta dati dai voli.

Il servizio è sviluppato in **Python** con **Flask** e utilizza PostgreSQL per persistere dati su aeroporti e voli

3.2.1. Comunicazion con UserManager (gRPC)

Ogni operazione che richiede un utente valido (es. aggiunta o rimozione aeroporto) prima verifica l'esistenza dell'utente tramite gRPC:

```
if not grpc_client.check_user(email):
    return {"error": f"User {email} does not exist"}, 404
```

Questo garantisce che il Data Collector non gestisca dati di utenti inesistenti

3.2.2. API

1.Aggiunta aeroporto (POST /add_airport)

- **Request**:

```
{
  "email": "user@example.com",
  "airport_code": "JFK"
}
```

- **Flusso interno**:

  1. Verifica che email e airport_code siano presenti.
  2. Controllo tramite gRPC che l'utente esista.
  3. Inserimento nel DB della coppia (email, airport_code).

- **Response**:

```
{ "status": "added" }
```

2. Rimozione aeroporto (DELETE /remove_airport)

- **Request**:

```
{
  "email": "user@example.com",
  "airport_code": "JFK"
```

}

- **Flusso interno**:

    1. Verifica parametri.
    2. Controllo tramite gRPC che l'utente esista.
    3. Cancellazione dal DB della coppia (email, airport_code).

- **Response**:

```
{ "status": "removed" }
```

3. Lista aeroporti utente (GET /airports/<email>)

- **Flusso interno**:

    1. Verifica che l'utente esista tramite gRPC.
    2. Lettura dal DB di tutti gli aeroporti associati all'utente.

- **Response**:

```
{
    "email": "user@example.com",
    "airports": ["JFK", "LAX"]
}
```

4. Ultimo volo per aeroporto (GET /last_flight/<airport_code>)

- **Query params**: email=<utente>

- **Flusso interno**:

    1. Verifica che l'email sia fornita.

    2. Query al DB per il volo più recente per (airport_code, email).

- **Response (successo)**:


```
{
  "last_flight":
   {
     "flight_code": "AB123",
      "airport_code": "JFK",
       "created_at": "2025-12-01T15:30:00Z"
   }
}
```


- **Response (errore)**:


```
{ "error": "No flights found for this airport and user." }
```


5. Media voli negli ultimi X giorni (GET /avg/<airport_code>/<int:days>)

- **Query params**: email=<utente>

- **Flusso interno**:

    1. Verifica email.
    2. Query al DB per contare i voli dell'utente negli ultimi X giorni per quell'aeroporto.
    3. Calcolo della media dividendo il count per i giorni.

- **Response (successo)**:

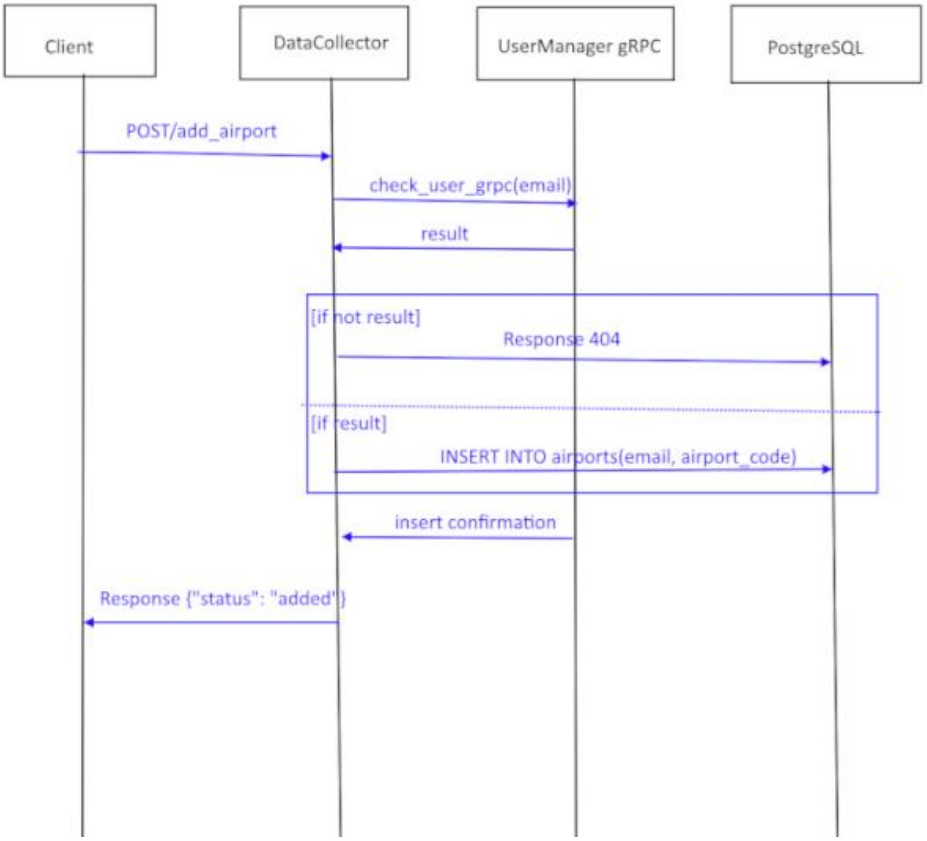
```
{
    "airport": "JFK",
    "days": 7,
     "average_flights": 3
}
```

- **Response (errore)**:

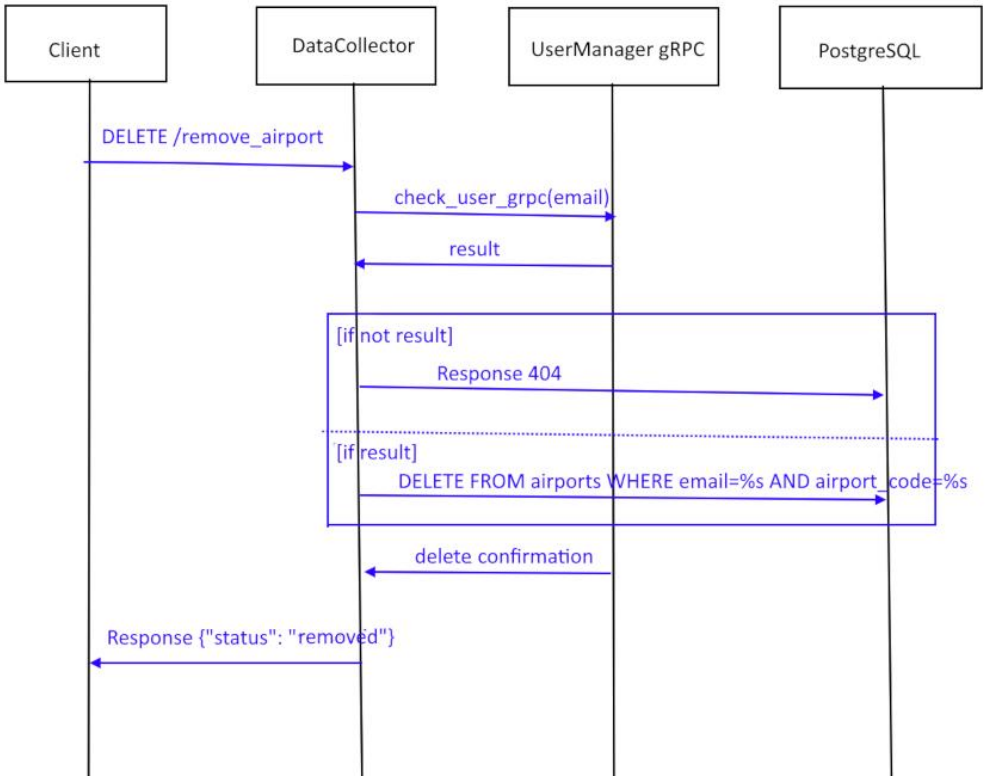
```
{ "error": "No flights found for this airport and user." }
```

3.2.3. Diagrammi di Sequenza API

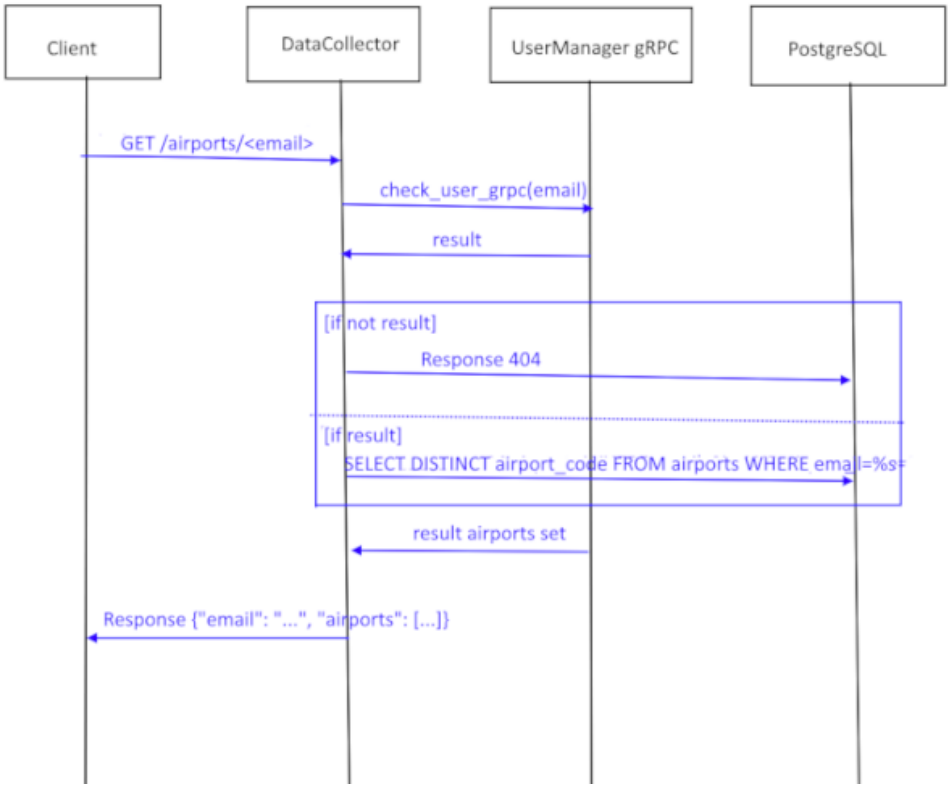
1) POST/add_airport



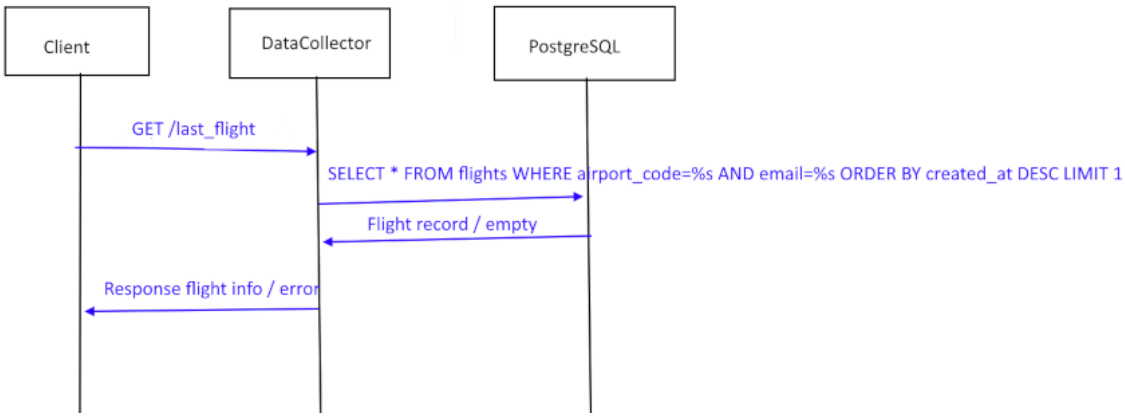
2) DELETE/remove_airport



3) GET /airports/<email>



4)GET/last_flight/[airport_code]



5) GET /avg/[airport_code]/[int:days]