

PROGETTO 1 - Distributed Systems and Big Data

UniCT - Anno Accademico 2025/2026

Gioele Messina - Luca Camiolo

1. DESCRIZIONE APPLICAZIONE

Il progetto implementa un'architettura per la gestione di utenti e di informazioni riguardanti voli aerei mediante un server User Manager, due database MySQL ed un Data Collector, il quale si occupa anche di gestire la comunicazione openSky Network sfruttando le API che quest'ultimo mette a disposizione.

Il nostro lavoro implementa dei meccanismi che mirano a ridurre il carico sul sistema, in quanto se più utenti esprimono il medesimo interesse verso uno specifico aeroporto verrà effettuata una singola richiesta a openSky per ottenere le informazioni riguardo i voli per quello specifico aeroporto, ottimizzando così l'uso del database in quanto non conterrà alcuna informazione duplicata. È stato implementato anche il meccanismo di eliminazione di tutti gli interessi di un utente, memorizzati nel data_db, quando l'utente elimina il proprio account. È stato implementato anche un meccanismo di sicurezza che consente al Data Collector, comunicando con lo User Manager tramite canale grpc, di verificare se le richieste che riceve sono effettuate da utenti registrati al servizio, ed in caso contrario negandone l'esecuzione.

Lo User Manager implementa la politica "at-most-once" per la funzionalità di iscrizione degli utenti, garantendo che una stessa richiesta venga processata al massimo una volta sola. Inoltre, lo User Manager si occupa di gestire i dati utente tramite il database user_db, consentendo agli utenti di poter eliminare il proprio account.

Il Data Collector opera sia ciclicamente, per aggiornare i dati relativi ai voli riguardanti gli aeroporti di interesse per gli utenti, sia non appena verrà espresso, da un utente, un aeroporto di interesse che non era stato indicato. Inoltre, si occuperà di fornire i dati archiviati nel database non appena un utente richiederà informazioni riguardo un dato aeroporto.

2. ARCHITETTURA DEL SISTEMA

Per la realizzazione della nostra applicazione distribuita abbiamo utilizzato quattro container: due per i database (user_db e data_db), uno per il microservizio UserManager e uno per il microservizio DataCollector. Con questa architettura è possibile implementare e scalare in modo indipendente i microservizi, i quali per comunicare tra loro utilizzano un canale gRPC sulla porta 50051.

Entrambi i microservizi utilizzano un database, in particolare lo UserManager memorizza nello user_db, all'interno della tabella "users", i dati relativi agli utenti registrati, come: nome, cognome, e-mail, eliminando i dati nel caso in cui l'utente decida di eliminare il proprio account, e nella tabella "requestID", memorizza la coppia (id,esito_risposta), informazioni

usate per implementare la politica at-most-once, mentre il Data Collector memorizza nel data_db, in particolare nella tabella "user_interest", gli interessi indicati dagli utenti, memorizzando la coppia (email,airport_code) e nella tabella "flights" tutte le informazioni dei voli, ottenute usando le API di openSky, relative agli aeroporti memorizzati nella tabella "user_interest" come aeroporto di partenza, di arrivo, orario di partenza, di arrivo, icao del volo.

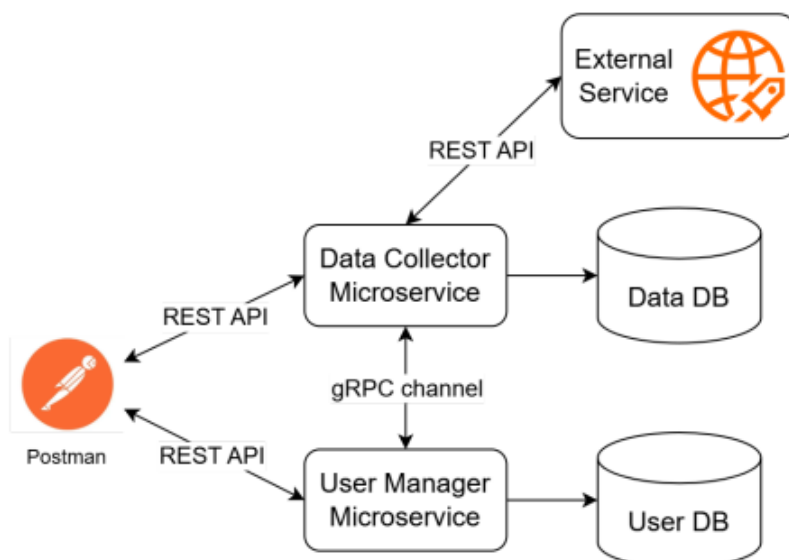
Lo User Manager è il microservizio il cui scopo è quello di:

- gestire la registrazione degli utenti;
- eliminare gli account quando l'utente lo richiede, andando a richiedere al Data Collector, tramite canale grpc, di eliminare tutti gli interessi a cui era sottoscritto quel dato utente.

Il Data Collector è il microservizio che si occupa di:

- recuperare ciclicamente le informazioni dei voli relativi agli aeroporti di interesse degli utenti da OpenSky;
- recuperare da OpenSky le informazioni dei voli quando viene sottoscritto da un utente un interesse relativo ad un aeroporto ancora mai monitorato;
- fornire tutte le informazioni relative ai voli di un dato aeroporto indicato dall'utente;
- fornire all'utente le informazioni relative all'ultimo volo in partenza e/o arrivo da un dato aeroporto;
- fornire all'utente la media dei voli in partenza e/o arrivo da un dato aeroporto nell'arco di tempo indicato dall'utente

Di seguito è riportato lo schema che mostra l'architettura del sistema:



2.1. MICROSERVIZI E LORO INTERAZIONI

- **User Manager:**

Scopo:

Comunica con lo user_db per leggere e aggiornare i dati relativi agli utenti.

Struttura file:

- app.py: Punto di ingresso principale dell'applicazione, comprendente API verso l'utente per esporre le funzionalità richieste dal progetto. Nello specifico fornisce le seguenti funzionalità all'utente:
 - ❖ /register: permette la registrazione dell'utente inserendo email (campo unique), nome e cognome.
 - ❖ /delete: permette la rimozione di tutte le informazioni dell'utente associate all'e-mail specificata.
- user_logic.py: contiene la logica applicativa.
- gRPC_Logic.py: Il file definisce il client e il server gRPC per verificare l'email degli utenti e richiedere al Data Collector l'eliminazione di tutti gli interessi di un dato utente.
- database.py: Modello di creazione del database user_db con le relative tabelle.

Canale gRPC

- service: UserService
- metodo: VerifyUser (UserRequest) returns (UserResponse);
 - metodo che permette di verificare se una data e-mail è presente all'interno dello user_db. Questo metodo viene anche richiamato dal Data Collector ogni volta che deve verificare se un utente è correttamente registrato

- **Data Collector:**

Scopo:

Raccoglie le informazioni sui voli, relative agli aeroporti di interesse sottoscritti dagli utenti, memorizzandole nel database data_db, e fornendo endpoint API per consentire agli utenti di sottoscrivere a degli aeroporti, consentendo anche di recuperare informazioni relative ai voli. Per il recupero periodico delle informazioni (ogni 12 ore) viene utilizzato un processo in background (scheduler) che si occupa di eseguire la funzione del recupero dei dati utilizzando le API di OpenSky.

Struttura file:

- app.py: Punto di ingresso principale dell'applicazione Data Collector, comprendente API verso l'utente per esporre le funzionalità richieste dal progetto e implementa il canale grpc. Nello specifico fornisce le seguenti funzionalità all'utente:

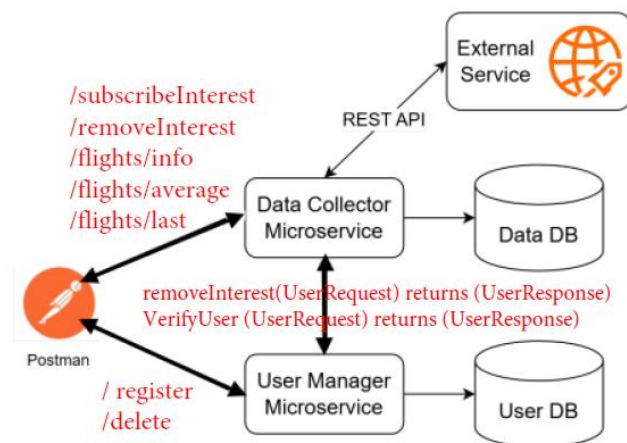
- ❖ `/subscribeInterest`: permette all'utente di sottoscrivere ad uno o più aeroporti specificando l'e-mail e gli ICAO degli aeroporti di suo interesse. Il Data Collector prima di registrare tali informazioni chiede allo User Manager, tramite canale gRPC se l'e-mail presente nell'header della richiesta è stata registrata. In caso affermativo registrerà le informazioni. Nel caso in cui l'utente abbia indicato un ICAO che non era stato indicato da nessun altro utente, e quindi non sono presenti informazioni riguardo i voli per quell'aeroporto, un thread si occuperà di interrogare tramite API OpenSky in modo da recuperare le informazioni.
 - ❖ `/removeInterest`: permette la rimozione dell'interesse indicato dall'utente. Il Data collector quando riceve una richiesta di rimozione di uno o più interessi da parte di un utente, prima di tutto chiede allo User Manager, tramite canale gRPC, se l'utente è registrato, in caso affermativo verifica se l'utente era sottoscritto a quegli interessi, e solo in questo caso procederà all'eliminazione.
 - ❖ `/flights/info`: permette all'utente di ottenere tutte le informazioni sui voli relative ad un dato aeroporto. L'utente nella richiesta indica l'icao dell'aeroporto per cui vuole ottenere le informazioni (iaco). Il data Collector, una volta verificato che l'utente è sottoscritto, recupererà dal database tutte le informazioni relative a quell'aeroporto, mostrandole all'utente
 - ❖ `/flights/average`: permette all'utente di ottenere la media dei voli partiti/atterrati in un dato aeroporto negli ultimi X giorni. In particolare, l'utente nella richiesta indica l'email, l' ICAO dell'aeroporto di cui vuole ottenere l'informazione, il numero di giorni per cui vuole calcolare la media, ed il type, il quale può essere "arrival" o "departure" (qualunque altro valore verrà considerato errato e la richiesta non viene gestita). Il Data Collector, una volta verificata l'e-mail tramite canale gRPC, calolerà la media dei voli in partenza dal dato aeroporto indicato se l'utente ha indicato come type "departure", in arrivo se ha indicato "arrival".
 - ❖ `/flights/last`: permette all'utente di ottenere l'ultimo volo partito/ in arrivo per un dato aeroporto. In particolare, l'utente nella richiesta deve indicare l'icao dell'aeroporto (airport) a cui è interessato ed il type, il quale può essere o "arrival" o "departure" (qualunque altro valore verrà considerato errato e la richiesta non viene gestita)
Il Data collector dopo aver verificato che l'e-mail dell'utente è registrata, mostrerà l'ultimo volo partito dall'aeroporto indicato dall'utente se il type indicato è "departure" o l'ultimo volo arrivato all'aeroporto indicato se type è "arrival"
- `dataCollector_logic.py`: contiene la logica applicativa
 - `gRPC_Logic.py`: Il file definisce il client e il server gRPC per rimuovere tutti gli interessi di un dato utente e comunicare con lo User Manager per richiedere la verifica dell'email.
 - `OpenSky.py`: contiene la logica per recuperare il token e

- database.py: Modello di creazione del database user_db con le relative tabelle.

Canale gRPC

- service: DataCollectorService
- metodo: removeInterest(UserRequest) returns (UserResponse)
 - Questo metodo consente l'eliminazione di tutti gli interessi associati ad un dato utente (e-mail) nel data_db nella tabella user_interest. Questo metodo viene richiamato dallo User Manager ogni volta che un utente richiede l'eliminazione del proprio account.

Schema di interazione



3. SCHEMA DEI DATABASE

I database utilizzati sono 2, lo user_db ed il data_db:

user_db:

Lo user_db è il database utilizzato dal microservizio User Manager attraverso la porta 3311 per memorizzare le informazioni degli utenti e delle richieste gestite, in particolare contiene 2 tabelle:

- **users:** vengono memorizzati i dati utente, in particolare troviamo tre campi:
 - Email (VARCHAR) (PRIMARY KEY): non è possibile memorizzare due righe che abbiano la stessa e-mail
 - Nome (VARCHAR)
 - Cognome (VARCHAR)
- **requestID:** vengono memorizzati gli id delle richieste di registrazione effettuate dagli utenti con il relativo esito dell'operazione. In particolare, troviamo due campi:
 - id (VARCHAR): memorizza l'id della richiesta
 - esito_richiesta: memorizza l'esito di elaborazione della richiesta

Questa tabella è stata introdotta per implementare la politica at-most-once riguardo la registrazione degli utenti.

AT-MOST-ONCE

La politica at-most-once implementata permette di impedire l'elaborazione di richieste di registrazione degli utenti duplicate, garantendo quindi che ciascuna richiesta "/register", effettuata allo User Manager, venga elaborata al massimo una volta. Per implementare questa politica è stata introdotta la tabella requestID dove lo User Manager andrà a memorizzare tutti gli id univoci delle richieste di registrazione che riceve. Quando arriva una richiesta "/register", lo User Manager innanzitutto verifica che quella richiesta non sia stata già ricevuta ed elaborata, e per fare questo controlla se l'id della richiesta ricevuta è presente in tabella:

- Se l'id non è presente in tabella allora significa che è una nuova richiesta, quindi verrà elaborata e l'esito dell'elaborazione, insieme all'id della richiesta, verranno memorizzati nella tabella requestID.
- Se l'id risulta presente nella tabella significa che l'utente per qualche motivo ha ritrasmesso la stessa richiesta (ad es non ha ricevuto la risposta e quindi allo scadere di un ipotetico timer ritrasmette), quindi in questo caso la richiesta non verrà elaborata e lo User Manager inoltrerà semplicemente l'esito della richiesta elaborata inizialmente, la quale era stata salvata nella tabella requestID.

Se la richiesta effettuata dall'utente per qualche motivo non presenta un id, la richiesta verrà automaticamente scartata senza essere elaborata.

data_db:

Il data_db è il database utilizzato dal microservizio Data Collector attraverso la porta 3307 per memorizzare gli interessi degli utenti e tutte le informazioni dei voli relativi agli aeroporti di interesse agli utenti, in particolare contiene 2 tabelle:

- User_interest: vengono memorizzati gli interessi degli utenti, in particolare contiene due campi:
 - Email (VARCHAR) : contiene l'email dell'utente
 - airport_code (VARCHAR): contiene l'icao a cui è interessato l'utente

Email e airport_code sono UNIQUE

- flights: vengono memorizzate tutte le informazioni relative ai voli recuperate tramite l'api di OpenSky, in particolare abbiamo i seguenti campi:
 - icao (VARCHAR): contiene l'icao del volo
 - departure_airport(VARCHAR): contiene l'icao dell'aeroporto di partenza
 - arrival_airport(VARCHAR): contiene l'icao dell'aeroporto di arrivo
 - departure_time (BIGINT): contiene la data e l'orario di partenza in formato UNIX
 - arrival_time (BIGINT): contiene la data e l'orario di arrivo in formato UNIX

icao e departure_time sono UNIQUE, questo permette di non avere informazioni duplicate relative allo stesso volo

4. SCELTE DI IMPLEMENTAZIONE

Framework: Per lo sviluppo dei microservizi è stato adottato Flask, un framework web Python leggero che permette di costruire API in modo rapido e senza vincoli strutturali.

containerizzazione: i microservizi e i database, vengono containerizzati tramite Docker. L'utilizzo di Docker Compose consente di orchestrare facilmente più container, definendo reti, dipendenze e configurazioni in un singolo file facilitando lo sviluppo. L'uso dei container garantisce che l'applicazione funzioni allo stesso modo su diversi tipi di macchine, riducendo i problemi legati alle differenti configurazioni delle macchine, rendendo più semplice scalare il sistema.

Canale gRPC per la comunicazione tra microservizi: Per lo scambio di informazioni tra i vari servizi è stato utilizzato gRPC, un protocollo che utilizza i file .proto per definire contratti precisi tra client e server. Questa tecnologia assicura un trasferimento dati molto più rapido rispetto alle tradizionali API REST basate su JSON, grazie alla serializzazione compatta fornita dal Protocol Buffers.

Il protocol Buffer, rispetto ad altri formati testuali, ottimizza al massimo la codifica dei messaggi in modo da aumentare l'efficienza, riducendo la quantità di informazioni necessarie per rappresentare lo stesso contenuto informativo, in quanto utilizza numeri identificativi (tag) per identificare i nomi di campo a differenza di formati testuali come Json, i quali utilizzano i nomi di campo in chiaro, risultando autoesplicativi ma richiedendo molti più byte. In questo modo il numero di byte trasmessi nella rete è molto minore e questo porta ad una riduzione dello spazio richiesto e della banda richiesta.

MySQL: Il sistema utilizza MySQL come motore di persistenza dei dati. La scelta è motivata dalla sua stabilità, diffusione e capacità di gestire agevolmente informazioni strutturate. Ogni microservizio dispone del proprio database, eseguito in un container dedicato, mantenendo così isolati i dati e semplificando sia la manutenzione sia la gestione dello schema. MySQL consente di gestire relazioni tra entità, vincoli di integrità e query complesse in maniera semplice ma efficiente.

5. CONFIGURAZIONE, ESECUZIONE E TEST

Requisiti

- Docker
- Docker Compose

Prima di eseguire l'applicazione, è necessario configurare le credenziali dell'API di OpenSky, per farlo è necessario aprire il .env e sostituire nel campo OPENSKEY_CLIENT_ID e OPENSKEY_CLIENT_SECRET le credenziali effettive di OpenSky Network, ottenute a seguito della registrazione al seguente link [OpenSky](#).

Per eseguire l'applicazione usare docker Compose eseguendo il comando da Terminal: docker-compose up --build.

TEST DA HTTP

Per testare l'applicazione è possibile usare il file `api_request.http` dove sono implementate tutte le chiamate che è possibile effettuare.

TEST DA POSTMAN

È possibile testare l'applicazione utilizzando anche postman effettuando le richieste allo User Manager all'indirizzo `127.0.0.1:5001/nome_api`

(es:

URL: `27.0.0.1:5001/register`

METODO: POST

Passare nel body tutte le informazioni necessarie per fare la richiesta:

```
{  
  "id": "{{ $guid }}",  
  "nome": "nome_utente",  
  "cognome": "cognome_utente",  
  "email": "email_utente"  
}
```

TEST POLITICA AT-MOST-ONCE

Per testare la politica at-most-once effettuare una richiesta legittima di registrazione (tramite http o Postman) facendo generare l'id della richiesta in modo automatico (in Postman usare `"id": "{{ $guid }}"`). Una volta che l'utente è stato registrato ripetere la richiesta, ma questa volta forzando nell'id della richiesta l'id della richiesta precedente, visibile nella tabella `requestID` (in Postman ad es: `"id": "id_richiesta_precedente"`). Una volta mandata questa richiesta lo User Manager troverà l'id della richiesta nella tabella che contiene le richieste gestite e quindi non elaborerà la richiesta, restituendo all'utente l'esito dell'elaborazione della richiesta precedente.

Se si effettua una nuova richiesta di iscrizione ma con la stessa email (id richiesta differente, su Postman riutilizzare `"id": "{{ $guid }}"`), lo User Manager accetterà la richiesta in quanto non era stata precedentemente elaborata (id della richiesta non presente nella tabella `requestID`), ma grazie alla verifica dell'e-mail che viene fatta prima di registrare un utente (controlla se l'email è presente nella tabella `users`), vedrà che l'email è già presente e quindi non permetterà la registrazione.