Corso di Distributed System and Big Data
A.A. 2025-2026

Homework Progetto#1

Relazione

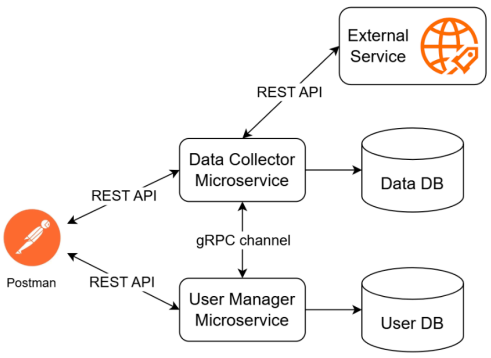
Giorgia Maria Musumeci - 1000061185

INTRODUZIONE

L'homework consiste nella progettazione e sviluppo di un sistema distribuito a microservizi per il monitoraggio del traffico aereo. L'obiettivo principale è permettere agli utenti di registrarsi al servizio, indicare aeroporti di interesse e consultare statistiche sui voli, basandosi sui dati reali forniti dall'API esterna OpenSky Network.

Il sistema è stato implementato con Docker, utilizzando container per garantire il disaccoppiamento tra i componenti.

DIAGRAMMA ARCHITETTURALE

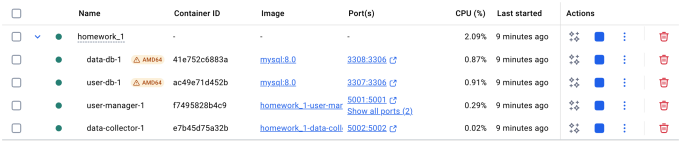


Come mostrato in figura, il sistema si compone dei seguenti elementi:

1. *User Manager Service:* microservizio responsabile della gestione degli utenti (registrazione e cancellazione). Espone un'interfaccia REST verso l'esterno e un server gRPC verso l'interno.
2. *Data Collector Service:* microservizio "core" che gestisce la logica di business. Si occupa di raccogliere le preferenze degli utenti, verificare la loro esistenza tramite gRPC, e scaricare ciclicamente i dati da OpenSky Network tramite uno scheduler interno.
3. *User DB:* database relazionale (MySQL) dedicato esclusivamente ai dati anagrafici degli utenti.
4. *Data DB*: database relazionale (MySQL) dedicato alla memorizzazione del interessi e dello storico dei voli scaricati.

DETTAGLIO DEI CONTAINER

Il sistema è stato implementato utilizzando 4 container Docker distinti, ognuno con una responsabilità specifica, per garantire l'isolamento dei processi e la scalabilità indipendente dei componenti.
Tutti i container comunicano tra di loro tramite una rete bridge interna dedicata. I database non sono accessibili direttamente dall'esterno, ma solo dai microservizi e dall'host locale per scopi di manutenzione, aumentando la sicurezza dell'architettura.



SCELTE PROGETTUALI

È stato scelto Python con framework Flask per la sua versatilità nella creazione rapida di servizi RESTful e per l'ampia disponibilità di librerie per la gestione delle reti e dei dati.

Per garantire il disaccoppiamento è stato scelto di utilizzare un database MySQL per ogni microservizio. Questo garantisce che un eventuale carico eccessivo sui dati dei voli non impatti le prestazioni della gestione utenti.

L'uso di *Docker* assicura che l'ambiente di esecuzione sia isolato e riproducibile su qualsiasi macchina, risolvendo le dipendenze software.

Nella registrazione utenti è stata implementato la logica *At-Most-Once*, basata su token di idempotenza. Il client invia, insieme ai dati, un identificativo univoco della richiesta (request_id). Lo *User Manager* verifica in una tabella di log dedicata (request_log) se quell'ID è già stato processato. Se l'ID è presente, il server restituisce la risposta salvata in precedenza senza rieseguire l'inserimento. Se l'ID è nuovo, il server procede con la creazione dell'utente. Questo meccanismo garantisce che l'operazione di registrazione venga eseguita esattamente una volta (o nessuna), anche in caso di ritrasmissioni multiple dovute a errori di rete.

COMUNICAZIONE E INTERAZIONE

Il sistema utilizza un approccio di comunicazione ibrida per ottimizzare le prestazioni:

- *REST (HTTP/JSON):* utilizzato per l'interazione tra Client esterno (Postman) e i microservizi.
- *gRPC*: utilizzato per la comunicazione interna sincrona tra Data Collector e User Manager.

IMPLEMENTAZIONE DELLA POLITICA "At-Most-Once"

Per soddisfare il requisito nella registrazione utenti, è stata implementata una politica *At-Most-Once*.
In caso di errore di rete, un client potrebbe inviare nuovamente la stessa richiesta di registrazione, rischiando di creare duplicati o errori non gestiti.

La soluzione implementata consiste che il client invia, insieme ai dati, un identificatore univoco della richiesta (*request_id*), lo user Manager mantiene una tabella dedicata *request_log* nel database. All'arrivo di una richiesta, il servizio verifica se quel *request_id* è già stato processato; se è stato già processato restituisce immediatamente la risposta salvata in precedenza, senza rieseguire l'inserimento, altrimenti esegue l'inserimento dell'utente e salva l'esito nel log.
Questa operazione garantisce che l'utente venga creato al massimo una volta, indipendentemente dal numero di tentativi del client.

LISTA DELLE API IMPLEMENTATE

Il sistema espone le seguenti interfacce RESTful:

- Microservizio User Manager (Porta 5001)
  POST/register: Registra un nuovo utente. Richiede email, first_name, last_name, request_id.
- Microservizio Data Collector (Porta 5002)
  POST/add_interest: Aggiunge un aeroporto da monitorare. Richiedere email, airport_code.
  GET/stats/last/<code>: restituisce i dati dell'ultimo volo registrato per un dato aeroporto.
  GET/stats/average/<code>/<days>: Calcola e restituisce la media dei voli in arrivo negli ultimi X giorni.

Il sistema è stato validato tramite test Postman.

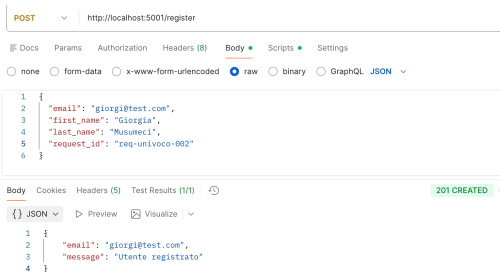
TEST E VALIDAZIONE DEL SISTEMA

La validazione del sistema è stata effettuata tramite test eseguiti con Postman, simulando il comportamento di un client.

Di seguito troviamo tre scenari di test principali.

1. Test di Registrazione Utente

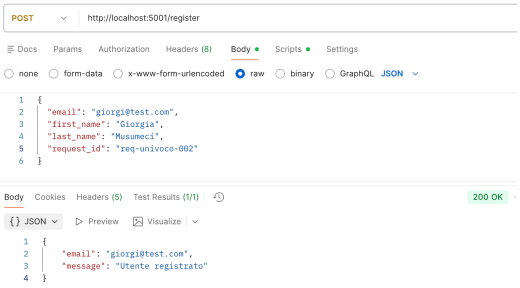
Verifica la corretta esposizione dell'API REST dello User Manager e la persistenza dei dati nel database user_db.
È stata inviata una richiesta POST all'endpoint /register contenente i dati anagrafici di un nuovo utente e un request_id univoco. Il server ha risposto con codice *201 Created*, confermando l'avvenuto inserimento nel database.



Test di Registrazione Utente

2. Test della politica At-Most-Once

Dimostra la robustezza del sistema in caso di richieste duplicate. È stata inviata nuovamente la stessa identica richiesta del test precedente, mantenendo invariato il request_id. Il sistema non ha generato errori di chiave duplicata nel database. Al contrario, ha riconosciuto l'ID della richiesta tramite la tabella di log interna e ha restituito la risposta memorizzata precedente.

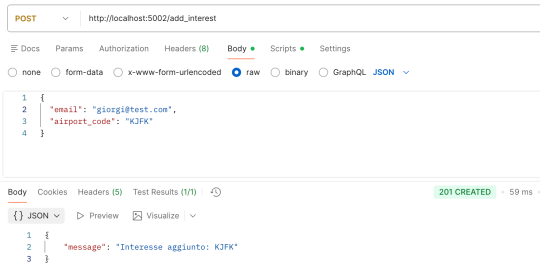


Test At Most Once

Questo conferma che l'operazione di registrazione viene eseguita al massimo una volta, garantendo la consistenza dei dati.

3. Test di comunicazione gRPC

Verifica la comunicazione distribuita tra i microservizi. È stata inviata una richiesta POST all'endpoint /add_interest del Data Collector, specificando l'email dell'utente appena creato e un codice aeroportuale. Per soddisfare la richiesta, il Data Collector ha dovuto sospendere l'elaborazione e contattare lo user Manager tramite protocollo gRPC per verificare l'esistenza dell'email. Il server ha risposto con *201 Created*, questo dimostra che il canale gRPC tra i container è attivo e funzionante.

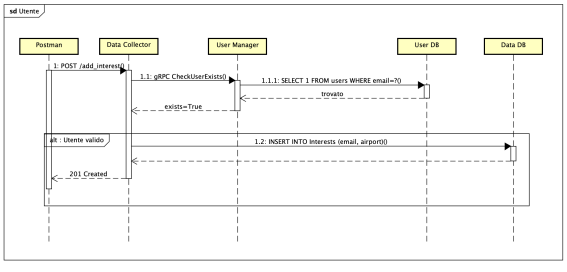


Test di comunicazione gRPC

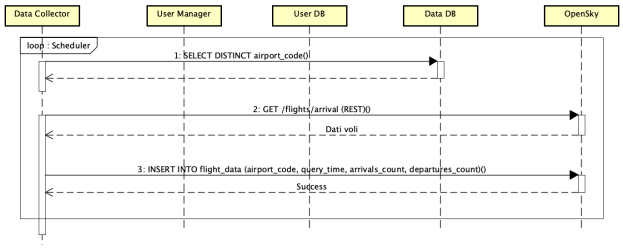
I test effettuati confermano che l'architettura a microservizi è stata implementata correttamente.

INTERAZIONI

Di seguito è riportato il diagramma di sequenza che mostra due flussi principali: interazione con l'utente (aggiunta dell'interesse e controllo gRPC) e Scheduler.



In questo diagramma si mostra che il Data Collector riceve una richiesta REST e fa una chiamata sincrona gRPC verso lo User Manager per validare l'identità tramite query sul User DB, prima di persistere l'interesse nel Data DB.



In questo diagramma lo Scheduler interno attiva ciclicamente il processo di aggiornamento, interrogando l'API esterna di OpenSky e salvando i risultati nel Data DB per le future analisi statistiche.

CONCLUSIONI

Il progetto ha raggiunto pienamente gli obiettivi prefissati, realizzando un'architettura a microservizi distribuita, scalabile e robusta. L'implementazione ha rispettato i requisiti, dimostrando la corretta applicazione dei pattern architetturali tipici dei sistemi distribuiti.