Distributed Systems & Big Data

Relazione Homework 1

Descrizione applicazione

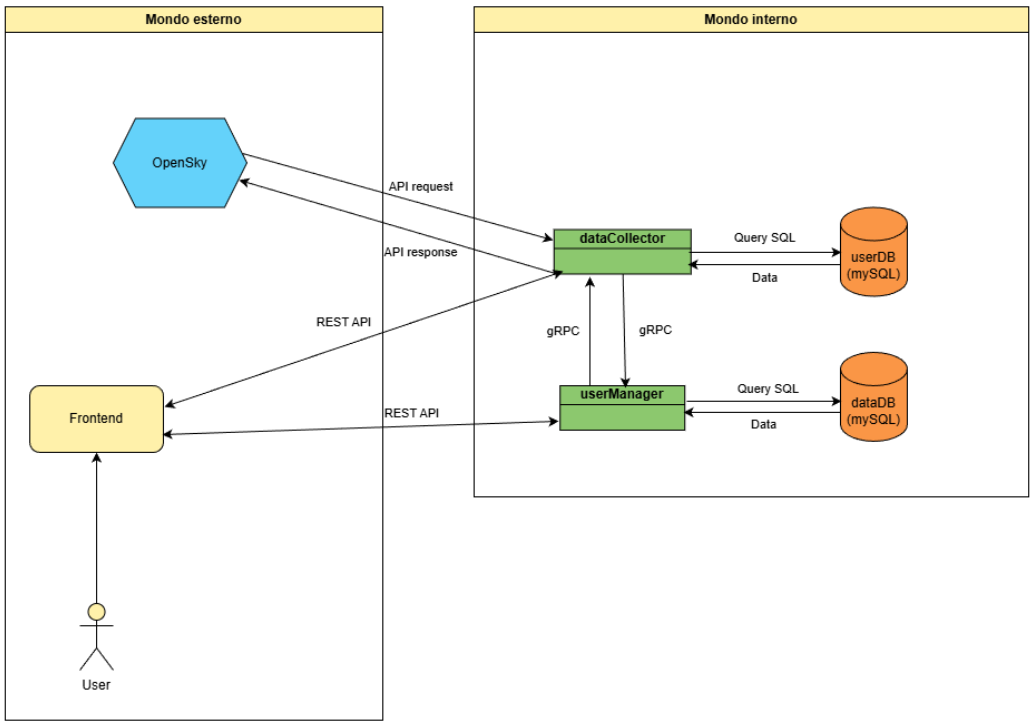
L'applicazione si presenta come un sistema distribuito composto da due servizi principali interni – User Manager e Data Collector – e da uno esterno, OpenSky.
Gli utenti possono registrarsi all'applicazione per cercare gli aeroporti di loro interesse e visualizzare i voli più recenti ad essi associati.

Le informazioni sugli aeroporti e sui voli vengono recuperate tramite le API di OpenSky e memorizzate dal Data Collector, in modo da rendere le successive consultazioni rapide e indipendenti dal servizio esterno.
Lo User Manager, invece, gestisce tutte le informazioni degli utenti registrati.

Per agevolare il testing, l'applicazione presenta un front-end composto da due pagine HMTL. La Home page consente di effettuare la registrazione, il login, e la cancellazione dell'account. Dopo l'autenticazione, l'utente viene reindirizzato alla Data Collector Page, da cui può testare tutte le funzionalità disponibili, tra cui: aggiungere o rimuovere un aeroporto dalla lista di interesse, visualizzare gli aeroporti aggiunti, consultare i prossimi voli in partenza o in arrivo dagli aeroporti di interesse, calcolare la media dei voli in partenza e in arrivo negli ultimi giorni da un dato aeroporto, e visualizzare l'ultimo volo in partenza e in arrivo da un dato aeroporto.

Il Data Collector aggiorna automaticamente i dati ogni 12 ore tramite un thread separato, così da mantenere il database locale sempre coerente con quanto esposto da OpenSky.

Schema architetturale



Database

Per affrontare questo homework abbiamo iniziato dalla progettazione dei database necessari a memorizzare le informazioni sugli utenti e sui voli.
Abbiamo scelto MySQL come tecnologia di riferimento perché open-source, affidabile e largamente diffusa.

Il primo database, User DB, contiene due tabelle: users, che memorizza i dati degli utenti registrati; request_log, utilizzata per registrare le richieste di registrazione e cancellazione e per implementare la politica at-most-once.
La tabella users include:

- email come chiave primaria;
- username, soggetto a vincolo unique;
- password, salvata tramite hashing bcrypt.

La tabella request_log contiene:

- request_id come chiave primaria, generata tramite hashing SHA-256 di email, username, operation e timestamp, così da garantirne l'unicità;
- operation, che distingue tra registrazione e cancellazione;
- timestamp della richiesta;
- message, ovvero la risposta associata alla richiesta, necessario per restituire sempre lo stesso esito in caso di duplicati;
- status_code, che identifica il risultato dell'operazione.

Il secondo database, Data DB, gestisce le informazioni sugli aeroporti di interesse e sui voli.
La tabella interests associa ogni utente agli aeroporti seguiti e contiene:

- email dell'utente;
- airport_code (codice ICAO).

  La chiave primaria è composta dai due campi precedenti.

La tabella flights raccoglie le informazioni provenienti da OpenSky e comprende:

- un id autoincrementale;
- icao24, identificativo esadecimale dell'aeromobile;
- callsign del volo;
- departure_airport e arrival_airport;
- departure_time e arrival_time;
- la data dell'ultimo aggiornamento.

Per strutturare i database abbiamo creato una cartella DB all'interno del progetto, nella quale sono presenti i due file .sql che definiscono le tabelle. Questi file non vengono eseguiti direttamente da quella cartella: durante la build del container MySQL vengono copiati in automatico nella directory docker-entrypoint-initdb.d del container.

È proprio questa directory interna a MySQL che consente l'inizializzazione automatica: al primo avvio del container, MySQL esegue tutti i file .sql presenti lì, creando così tabelle e strutture dati senza interventi manuali.

Pattern di comunicazione

L'architettura utilizza API REST e gRPC rispettivamente: per tutte le interazioni con il frontend e con i servizi esterni, e per la comunicazione interna tra microservizi.

All'esterno dei microservizi, la comunicazione tra Front-end, User Manager, Data Collector e OpenSky avviene tramite API REST, basate sul tradizionale pattern request/response su HTTP. Questo modello è semplice da implementare, indipendente dalla piattaforma e adatto a interazioni sincrone in cui il client richiede un'operazione e attende la risposta del server.
Il Front-end usa le API REST per accedere alle funzionalità offerte dai due servizi interni, mentre il Data Collector sfrutta lo stesso meccanismo per interrogare l'API pubblica di OpenSky e ottenere informazioni aggiornate su aeroporti e voli.
L'uso di REST garantisce un'interfaccia facilmente testabile e compatibile anche con strumenti esterni come Postman.

All'interno dell'ecosistema applicativo, invece, i microservizi comunicano tramite gRPC, un framework open-source per implementare Remote Procedure Calls in modo efficiente, multipiattaforma e scalabile.
Con gRPC è stato possibile definire User Manager e Data Collector, specificando: il formato dei messaggi di richiesta e risposta, e i metodi che possono essere invocati da remoto, insieme ai loro parametri e ai tipi di ritorno.
Per fare questo, gRPC utilizza i Protocol Buffers, che rappresentano il meccanismo utilizzato per la serializzazione di dati strutturati, a partire dai file .proto (che descriveremo fra breve), nei quali sono definiti i messaggi sotto forma di record di coppie nome-valore, e i metodi.
Grazie a gRPC le comunicazioni sono rapide e poco costose, il che è particolarmente utile per scambi frequenti e strutturati, come quelli tra User Manager e Data Collector.

File .proto e altri file di configurazione

Dopo aver predisposto i database, abbiamo definito i file .proto necessari alla comunicazione tra i microservizi tramite gRPC. Sono stati creati due file distinti: uno per lo User Manager e uno per il Data Collector, ciascuno progettato in base alle funzionalità del rispettivo servizio.

Il file dataCollector.proto, invece, contiene una gamma più ampia di messaggi e metodi, poiché il Data Collector gestisce operazioni eterogenee. Qui sono specificati servizi come AddAirport, RemoveAirport, ListAirports, RefreshFlights, ListFlights, AverageFlightsPerDay e LastFlights, ciascuno con il proprio formato di richiesta e risposta.

Tutti i servizi definiti nei due file .proto sono di tipo Unary RPC, cioè prevedono una singola richiesta e una singola risposta, analogamente a una normale funzione remota.

User Manager e Data Collector implementano le operazioni definite nei file .proto.
Inoltre, includono anche del codice Flask, per gestire le API REST esposte verso il browser, instradando le URL verso le funzioni appropriate.

La cartella del Data Collector contiene un file aggiuntivo, entrypoint.py, assente nello User Manager. Questo script ha un ruolo molto semplice: carica le credenziali OpenSky necessarie per accedere alle API e avvia l'esecuzione di dataCollector.py.
Questa soluzione è stata adottata per evitare di inserire nel repository GitHub credenziali sensibili come client ID e client secret.
Ogni sviluppatore può quindi collocare localmente il proprio file credentials.json, non tracciato nel repository grazie alla voce presente nel .gitignore.

Implementazione della Politica At-Most Once

La politica at-most-once garantisce che ogni messaggio venga elaborato non più di una volta.
Essa è stata implementata nella fase di registrazione e cancellazione di un utente.
Nel primo caso, ciò significa che l'invio di email, username e password allo User Manager viene processato zero o una volta, ma mai più.

Questa garanzia è assicurata dal campo request_id nella tabella request_log, generato in maniera univoca a partire da email, username, tipo di operazione e timestamp.
Prima di elaborare una nuova richiesta, lo User Manager verifica se l'id è già presente nel log: se sì, restituisce la stessa risposta memorizzata in precedenza; se no, processa la richiesta normalmente e registra l'esito insieme all'id nel log.

In questo modo, eventuali richieste duplicate — ad esempio dovute alla perdita della risposta originale — non provocano effetti collaterali come inserimenti o cancellazioni ripetute.
La memorizzazione dell'esito garantisce anche l'idempotenza dell'operazione: inviare più volte la stessa richiesta produce lo stesso risultato della prima esecuzione, senza errori.

Docker-Compose

Il progetto è strutturato con quattro container definiti nel file docker-compose.yml: userdb, datadb, user-manager e data-collector.

I due container dei database (userdb e datadb) sono configurati in modo da garantire isolamento e persistenza dei dati.
Inizialmente entrambi esponevano una porta verso l'esterno (3307 per userdb e 3308 per datadb), ma per motivi di sicurezza abbiamo rimosso la mappatura delle porte, così che i DB siano accessibili solo dai container autorizzati tramite reti interne dedicate. In particolare:

- userdb comunica esclusivamente tramite la rete user_db_net, cui è connesso solo il container user-manager.
- datadb comunica attraverso la rete data_db_net, cui è connesso solo il data-collector.

Entrambi i database sono associati a volumi persistenti, in modo da salvare i dati anche quando i container vengono ricreati.

Inoltre, all'interno dei container vengono montati file di inizializzazione .sql nella directory docker-entrypoint-initdb.d per creare automaticamente le tabelle al primo avvio.

Il container user-manager rappresenta il microservizio che gestisce le informazioni sugli utenti. Esso contiene le credenziali necessarie per connettersi al database userdb e comunica tramite gRPC e REST.
Il container espone due porte locali: 50051 per gRPC e 8081 per REST.
Dal punto di vista delle reti, user-manager è connesso sia ad app_net, la rete principale dell'applicazione per la comunicazione con il container data-collector, sia a user_db_net, necessaria per accedere al database in sicurezza.

Il container data-collector è il microservizio che gestisce la raccolta e l'elaborazione dei dati sui voli. Analogamente a user-manager, espone una porta REST (8082) per test e comunicazione con il front-end.
Per accedere al database datadb utilizza le credenziali specificate nelle variabili d'ambiente. Inoltre, il container monta in volume il file credentials.json contenente le credenziali per accedere alle API di OpenSky, garantendo così che i dati sensibili siano persistenti e accessibili in sola lettura (read-only).
Per quanto riguarda le reti, data-collector è connesso sia ad app_net sia a data_db_net, così da comunicare con il database e con lo user-manager.

Gestione delle Eccezioni

Abbiamo adottato una gestione delle eccezioni pragmatica e orientata alla continuità del servizio in entrambi i microservizi.

Nello User Manager, le operazioni di accesso al DB sono racchiuse in blocchi try/except, con rollback delle transazioni e ritorno di un codice 500 in caso di errori imprevisti.
Gestiamo, però, in modo specifico l'errore MySQL "duplicate key" (errno 1062) sull'inserimento in request_log per realizzare la semantica at-most-once e l'idempotenza: quando viene intercettato, recuperiamo e restituiamo messaggio e status originali.
La verifica delle password tramite bcrypt, intercetta eventuali eccezioni e restituisce False.
L'endpoint /exists – usato dal Data Collector per verificare l'esistenza di un utente che ha richiesto un servizio – cattura eccezioni DB e risponde con 500.
Alcuni aggiornamenti non critici alla tabella request_log sono gestiti in modalità best effort: un errore non interrompe il flusso e viene semplicemente registrato nei log.

Nel Data Collector, gli errori di rete verso lo User Manager (ad es. nella chiamata a /exists) e verso OpenSky sono gestiti con try/except, log e fallback non bloccanti: vengono restituiti messaggi di errore ma senza interrompere il servizio.
Infatti, durante l'ingest dei voli, ogni INSERT è protetto tramite try/except per "skippare" record problematici senza interrompere il batch.
Le operazioni CRUD gRPC critiche (Add/Rem. airport) catturano eccezioni DB e ritornano 500.
Gli endpoint REST, invece, applicano controlli preventivi sui parametri, e restituiscono codici 4xx per segnalare errori dell'utente, mentre lasciano propagare gli errori inattesi come 500.

I principali codici di stato usati nel progetto sono:

- 200: Operazione riuscita (es. login ok, delete ok, rimozione aeroporto riuscita…);
- 201: Risorsa creata (es. utente creato, aeroporto aggiunto…);
- 400: Richiesta non valida (es. campi mancanti in add/delete/login, /exists senza email/username);
- 401: Non autorizzato / credenziali invalide (es. login con password errata);
- 404: Non trovato (es. utente inesistente in delete);
- 409: Conflitto con risorse esistenti (es. email/username già esistenti in add, interesse già presente in add_airport);
- HTTP 500: Errore interno (es. errori DB generici, eccezioni durante inserimenti/refresh OpenSky).

REST API

Lo UserManager espone tre REST API principali, attraverso cui è possibile gestire gli utenti del sistema:

- POST /addUser: registra un nuovo utente nel sistema.
- POST /deleteUser: rimuove un utente già esistente.
- POST /loginUser: permette a un utente di autenticarsi e ottenere accesso ai servizi offerti dal DataCollector.
- GET /exists: verifica che un utente sia registrato.

Il DataCollector mette invece a disposizione una serie più ampia di endpoint, dedicati alla gestione degli aeroporti di interesse e all'analisi dei voli::

- GET /airport_suggest: fornisce suggerimenti durante la ricerca degli aeroporti da aggiungere agli interessi;
- POST /add_airport: aggiunge un aeroporto alla lista degli aeroporti seguiti dall'utente;
- POST /remove_airport: rimuove un aeroporto dagli interessi dell'utente;
- POST /list_airports: restituisce l'elenco completo degli aeroporti di interesse;
- POST /refresh: interroga le API OpenSky e aggiorna il database con i nuovi voli relativi agli aeroporti monitorati;
- POST /list_flights: restituisce tutti i voli associati a un determinato aeroporto;
- POST /debug_data_range: mostra l'intervallo temporale coperto dai voli attualmente presenti nel database.
- POST /average_flights_per_day: calcola il numero medio di voli in partenza e in arrivo da un dato aeroporto registrati negli ultimi X giorni;
- POST /last_flights: restituisce gli ultimi voli in partenza e in arrivo da un aeroporto specificato;

Schema di Comunicazione Dettagliato

Si mostra uno schema di comunicazione completo, in cui un utente si registra (con successo) all'applicazione, effettua il login, e interagisce con il data collector richiamando alcune delle funzioni offerte: