

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.

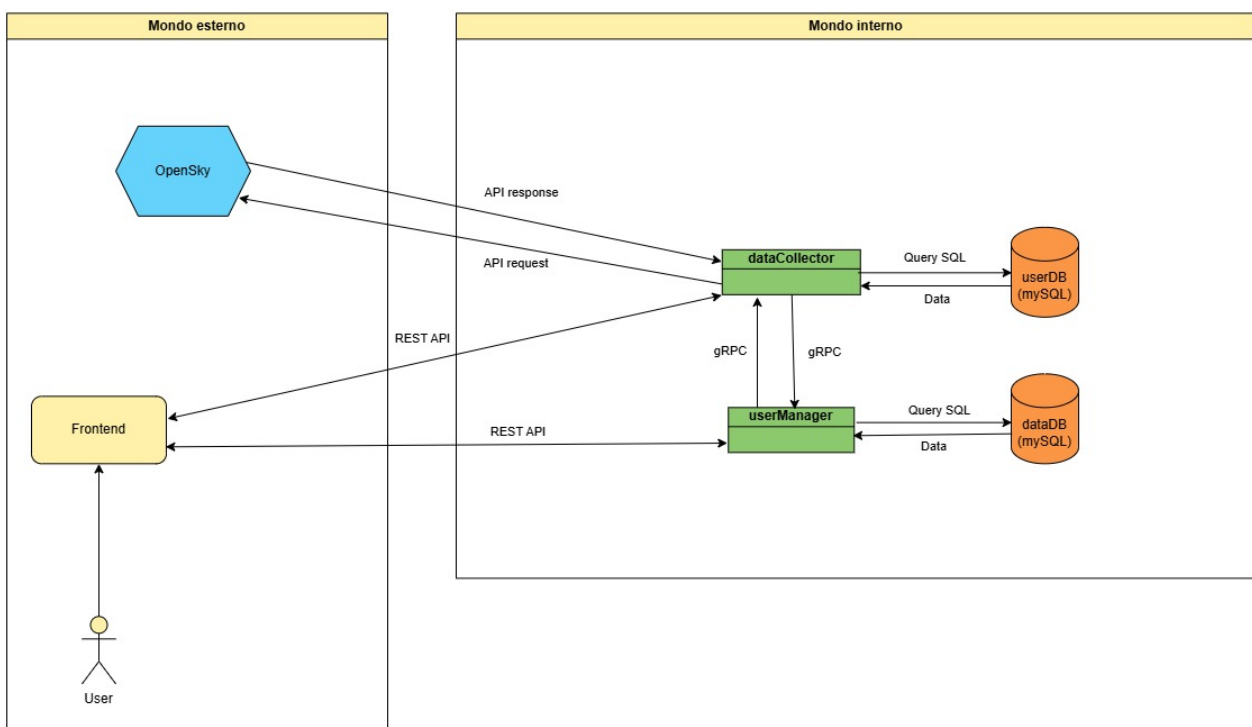
Le informazioni sugli aeroporti e sui voli vengono recuperate tramite le API di OpenSky e memorizzate dal Data Collector, in modo da rendere le successive consultazioni rapide e indipendenti dal servizio esterno.

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 HTML. 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, multiplatforma 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 espongono 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 "saltare" 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:

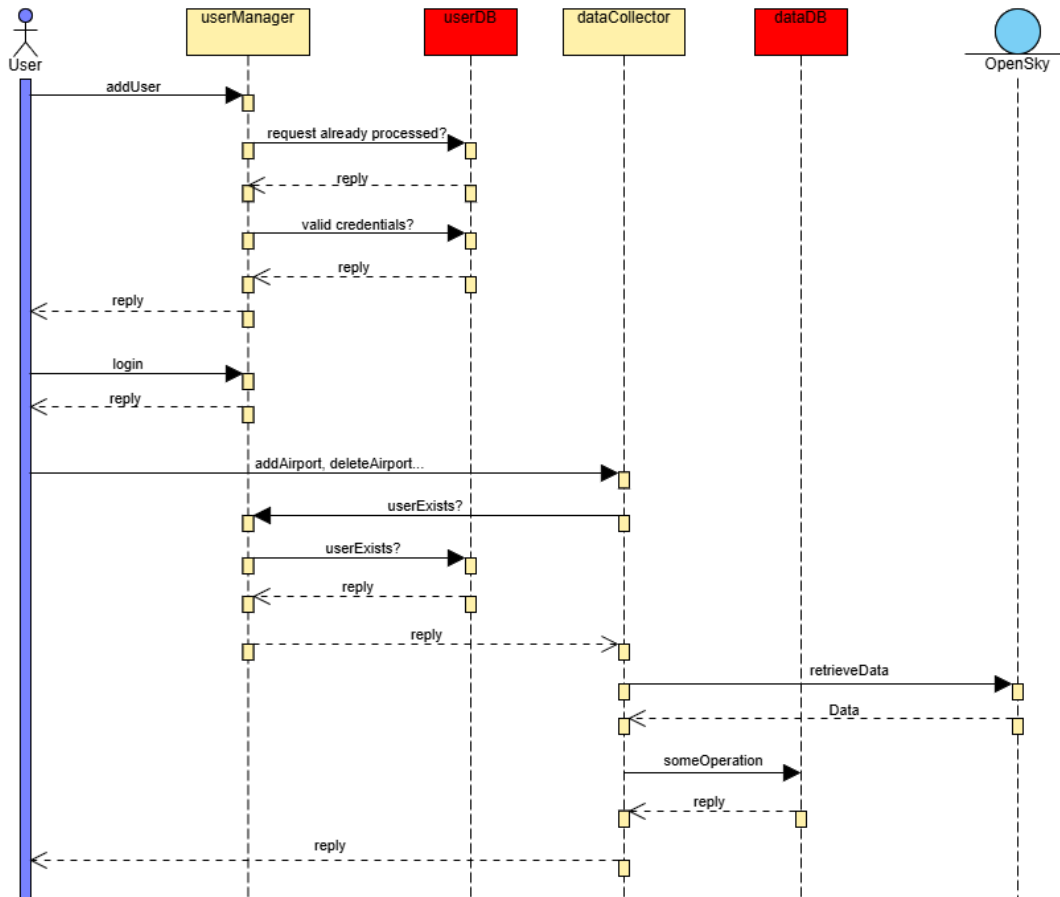
- POST /addUser: registra un nuovo utente nel sistema.
- POST /deleteUser: rimuove un utente già esistente.
- POST /loginUser: permette a un utente di autenticarsi e ottenere accesso ai servizi offerti dal DataCollector.
- GET /exists: verifica che un utente sia registrato.

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:



Iterazione 2

Nuove funzionalità

Il sistema sviluppato è stato esteso introducendo nuove funzionalità che ne ampliano le capacità di monitoraggio, notifica e robustezza architetturale.

La principale estensione riguarda la possibilità per l'utente di configurare, per ciascun aeroporto di interesse, due parametri di soglia opzionali: high-value e low-value.

A seguito di ogni aggiornamento dei dati, il sistema verifica per ciascun utente e per ciascun aeroporto monitorato se la somma dei voli in arrivo e in partenza supera la soglia superiore (condizione Above-high) oppure scende al di sotto di quella inferiore (condizione Below-low). Nel caso in cui una delle due condizioni si verifichi, l'utente viene notificato tramite email.

Per supportare questo meccanismo in modo asincrono e scalabile, è stato integrato un message broker basato su Kafka, utilizzato per la comunicazione tra il Data Collector e due nuovi componenti introdotti nel sistema: AlertSystem e AlertNotifierSystem.

In particolare, il Data Collector pubblica un messaggio sul broker al termine della fase di aggiornamento dei voli, includendo le informazioni relative al numero di voli registrati per ciascun aeroporto di interesse.

L'AlertSystem, in qualità di consumer, elabora tali messaggi verificando per ogni profilo utente il rispetto delle soglie configurate e, se necessario, pubblica un nuovo messaggio contenente il dettaglio della condizione rilevata.

L'AlertNotifierSystem, infine, consuma questi messaggi e si occupa dell'invio delle notifiche email agli utenti interessati.

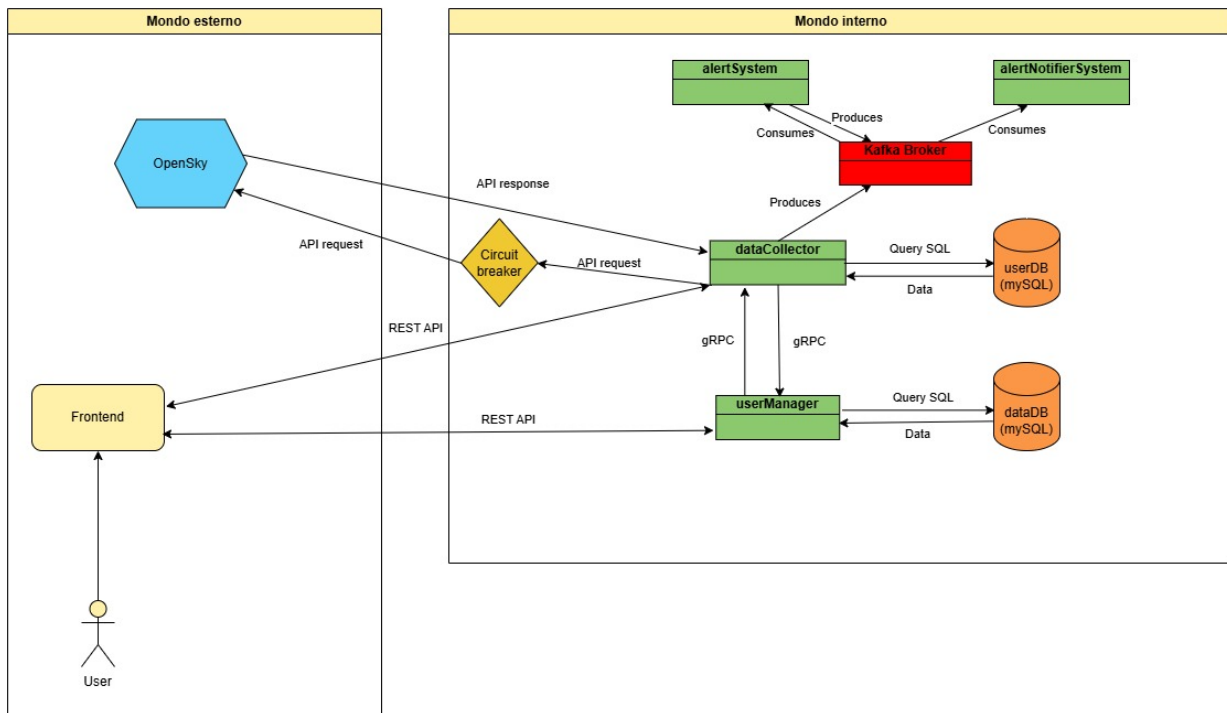
Un'ulteriore estensione riguarda l'introduzione di un API Gateway, realizzato tramite NGINX, che funge da punto di ingresso unico per il sistema.

Questa soluzione consente di centralizzare l'accesso ai servizi, migliorando la gestione delle richieste e facilitando eventuali evoluzioni future dell'architettura.

Infine, è stato integrato un Circuit Breaker a protezione di tutte le operazioni verso il servizio esterno OpenSky Network.

Questo meccanismo aumenta la resilienza del sistema, evitando interrogazioni ripetute verso un servizio non disponibile e riducendo l'impatto di fault esterni sulle componenti interne.

Schema Architeturale Aggiornato



Database Aggiornato

L'unica modifica apportata ai database riguarda il Data DB.

In particolare, la tabella interests è stata estesa con i campi high_value e low_value per supportare i nuovi parametri di soglia introdotti dal sistema.

Essi rappresentano rispettivamente la soglia superiore e inferiore sul numero totale di voli associati a un aeroporto di interesse di un utente.

È stato inoltre introdotto un vincolo di consistenza che impone, nel caso in cui entrambi i valori siano presenti, che high_value sia maggiore di low_value.

La tabella aggiornata risulta quindi composta dai seguenti campi:

- email;
- airport_code;
- high_value e low_value.

Circuit Breaker

Il Circuit Breaker è un componente utilizzato per controllare e proteggere le chiamate verso servizi remoti, interrompendo automaticamente le richieste quando viene rilevato un numero eccessivo di fallimenti consecutivi.

L'adozione di questo meccanismo offre diversi vantaggi all'interno del sistema, tra cui:

- si evitano interrogazioni ripetute verso un servizio non disponibile,
- si impedisce che il failure di un servizio ne comprometta altri (guasti in cascata),
- si riduce lo spreco di risorse computazionali su operazioni destinate a fallire,
- si riducono i tempi d'attesa restituendo rapidamente un errore controllato.

Nel sistema sviluppato, il Circuit Breaker è impiegato per proteggere tutte le chiamate verso il servizio esterno OpenSky Network, riducendone l'impatto complessivo in caso di indisponibilità prolungata.

L'implementazione è stata realizzata tramite un modulo Python dedicato, non containerizzato, collocato all'interno della directory del Data Collector.

Tale modulo definisce una classe che mantiene uno stato interno e gestisce il flusso delle richieste sulla base di alcuni parametri fondamentali: un contatore dei fallimenti, una soglia di errore (threshold), e un timer di recupero (recovery timeout).

Lo stato può assumere, in base alla disponibilità del servizio remoto, tre diversi valori:

- **Closed (Chiuso):** OpenSky è disponibile, e tutte le richieste gli vengono inoltrate normalmente. In questo stato viene monitorato il numero di fallimenti consecutivi incrementando il contatore: al superamento della soglia, il Circuit Breaker transita allo stato Open.
- **Open (Aperto):** OpenSky è indisponibile, e tutte le richieste vengono immediatamente bloccate restituendo un errore. Quando il sistema entra in questo stato, viene avviato il timer di recupero, al termine del quale lo stato passa a Half-Open.
- **Half-Open (Semichiuso):** viene consentita una singola richiesta di test per verificare se il servizio sia tornato operativo. In caso di successo il Circuit Breaker ritorna nello stato Closed, mentre in caso di fallimento viene nuovamente impostato su Open.

I valori dei parametri della classe sono stati scelti in modo coerente con le caratteristiche dell'applicazione e del servizio OpenSky:

- **failure_threshold:** numero massimo di fallimenti consecutivi prima di passare allo stato Open.
È stato impostato a 5, un valore pragmatico per API pubbliche come OpenSky, dove sono frequenti errori transitori. In questo modo distinguiamo meglio un guasto persistente dal "rumore" di rete, evitando aperture troppo aggressive ma senza ritardare eccessivamente la protezione;
- **recovery_timeout:** tempo di attesa, espresso in secondi, prima di passare dallo stato Open allo stato Half-Open.
Il valore scelto è 60 secondi, bilanciando reattività e prudenza nel tentativo di ripristino del servizio. Un minuto è, infatti, abbastanza lungo da lasciare tempo al servizio di riprendersi da malfunzionamenti temporanei ed evitare retry troppo frequenti, ma abbastanza corto da non penalizzare troppo il flusso dati;
- **expected_exception:** tipologia di eccezioni monitorate.
È stato utilizzato il tipo generico Exception per catturare in modo uniforme tutte le anomalie operative delle chiamate HTTP, mantenendo il breaker semplice e robusto.

Il DataCollector effettua il wrapping delle chiamate alle API di OpenSky tramite un'istanza del Circuit Breaker. In questo modo, ogni fallimento o successo aggiornano lo stato locale del breaker secondo le logiche descritte.

È anche prevista una modalità di test, impostando `FORCE_OPENSKY_ERRORS=true` per il container `data-collector` nel `compose`, che forza errori per verificare il comportamento.

Infine, le già discusse credenziali di OpenSky sono state spostate dal file `credentials.json` (rimosso) a un file `.env`, anch'esso non tracciato nel repository grazie alla voce presente nel `.gitignore` per permettere una configurazione sicura dei dati privati dell'utente.

Broker Kafka

Apache Kafka è un sistema di messaggistica distribuito che consente di pubblicare e sottoscrivere a flussi di messaggi organizzati in topic. È progettato per funzionare come un cluster scalabile, una piattaforma centrale in grado di scalare elasticamente per gestire più flussi di dati in modo efficiente.

Kafka non è soltanto un sistema di messaging, ma anche un sistema di archiviazione persistente: i messaggi vengono salvati su disco, replicati tra più broker e mantenuti per il tempo necessario. Questo garantisce durabilità dei dati e affidabilità nella consegna, anche in presenza di fault temporanei.

Un broker Kafka è un singolo server che riceve i messaggi dai producer, assegna loro un offset progressivo e li memorizza su disco. Allo stesso tempo, serve i consumer, rispondendo alle richieste di lettura e inviando i messaggi salvati.

Nel sistema sviluppato, Kafka è stato introdotto per implementare un meccanismo asincrono di notifica, che consente agli utenti di essere informati quando un aeroporto di loro interesse registra un numero di voli superiore o inferiore alle soglie configurate.

Il meccanismo di notifica è basato su due topic Kafka distinti:

- `to-alert-system`
- `to-notifier`

Il `DataCollector` agisce come producer del topic `to-alert-system`. Al termine della fase di aggiornamento dei dati provenienti da OpenSky, pubblica un messaggio che segnala il completamento dell'aggiornamento e contiene le informazioni raccolte per ciascun aeroporto di interesse. Il messaggio pubblicato include:

- codice dell'aeroporto;
- numero di voli in arrivo;
- numero di voli in partenza;
- timestamp.

Il consumer del topic `to-alert-system` è il nuovo componente `AlertSystem` che, per ogni messaggio ricevuto:

1. calcola la somma dei voli in arrivo e in partenza per l'aeroporto indicato;
2. verifica, per tutti gli utenti che hanno quell'aeroporto tra i propri interessi, se il valore supera la soglia `high_value` o scende sotto la soglia `low_value` (se presenti nel DB);
3. in caso di condizione verificata, pubblica un messaggio nel topic `to-notifier`.

Il messaggio prodotto da AlertSystem contiene:

- email dell'utente;
- codice dell'aeroporto;
- condizione rilevata (above_high oppure below_low);
- timestamp.

I messaggi pubblicati nel topic to-notifier vengono consumati da un altro nuovo componente: l'AlertNotifierSystem. Alla ricezione di ciascun messaggio, il servizio invia una notifica all'utente tramite email.

Per ogni condizione verificata viene inviata una singola email all'utente, contenente le informazioni relative all'aeroporto e al tipo di superamento della soglia.

Le email vengono inviate via SMTP con STARTTLS: la connessione è cifrata con TLS sulla porta 587 e utilizza credenziali e mittente letti dalle variabili d'ambiente (.env), configurabili seguendo le indicazioni in .env.example. Se l'SMTP non è impostato oppure DISABLE_EMAIL=true, il servizio effettua un "mock send": registra l'invio nei log senza spedire realmente, così l'intero flusso resta testabile in sicurezza.

Un aspetto rilevante riguarda i parametri di configurazione utilizzati per l'inizializzazione dei producer Kafka, sia nel DataCollector sia in AlertSystem (lato producer).

I parametri principali sono:

- acks=all: il broker risponde positivamente solo quando tutte le repliche hanno scritto il messaggio, garantendo la massima affidabilità;
- max.in.flight.requests.per.connection=5: consente fino a 5 richieste non ancora confermate sulla stessa connessione;
- retries=5: numero massimo di tentativi di ritrasmissione in caso di errore;
- linger.ms=10: tempo di attesa (in ms) prima dell'invio di un batch di messaggi.

La possibilità di mantenere più richieste in-flight è coerente con la natura dell'applicazione, in cui l'ordinamento globale dei messaggi non è critico. Poiché sono consentite fino a 5 richieste parallele, anche il numero di retry è stato impostato allo stesso valore, in modo da aumentare la probabilità di consegna in presenza di fault temporanei.

Per quanto riguarda i consumer, invece, i parametri impostati per AlertSystem e AlertNotifierSystem sono i seguenti:

- auto.offset.reset=latest: se il consumer non dispone di un offset salvato, inizierà a consumare i messaggi dalla fine del topic. Questa scelta evita il consumo di messaggi storici (potenzialmente vecchi di molte ore), che potrebbe generare un numero elevato di notifiche indesiderate e causare un sovraccarico del sistema di alert;
- enable.auto.commit=true: abilita il commit automatico periodico degli offset, semplificando la gestione e risultando adeguato in uno scenario in cui una duplicazione occasionale delle notifiche è tollerabile.

I topic `to-alert-system` e `to-notifier` vengono creati implicitamente dal broker Kafka, senza una configurazione esplicita del numero di partizioni nel file di `compose`. Di conseguenza, viene utilizzato il valore di default del broker, tipicamente una sola partizione.

Questa scelta comporta le seguenti implicazioni:

- per ogni topic, un solo consumer alla volta può leggere i messaggi all'interno dello stesso consumer group;
- l'ordinamento dei messaggi è totale;
- non è presente parallelismo intra-topic e la scalabilità è limitata;
- il modello risultante è assimilabile a una comunicazione point-to-point (un producer – un consumer).

L'invio dei messaggi Kafka avviene in modo asincrono, utilizzando una callback di conferma. In particolare, il `DataCollector` e l'`AlertSystem` utilizzano la callback `_delivery_report` viene invocata quando il broker risponde, consentendo di registrare il successo o il fallimento dell'invio senza bloccare l'applicazione sul singolo messaggio.

L'integrazione di Kafka introduce diversi benefici architetturali nel sistema realizzato:

- forte disaccoppiamento: il `DataCollector` pubblica eventi senza conoscere i consumer → `AlertSystem` e `AlertNotifierSystem` possono evolvere in modo indipendente;
- affidabilità e recupero: grazie alla persistenza dei topic e ad `acks=all`, i messaggi non vengono persi durante picchi di carico o fault temporanei;
- Buffering e backpressure: Kafka assorbe i burst di pubblicazioni (per aeroporto) e consente ai consumer di processare al ritmo disponibile, evitando blocchi tra servizi.
- Scalabilità futura: è possibile aumentare le partizioni e aggiungere istanze dei consumer per processare in parallelo.

API Gateway

In un'architettura a microservizi, i client non dovrebbero interagire direttamente con i singoli servizi. Farlo causa problemi critici:

- Ogni servizio deve essere chiamato direttamente, aumentano la latenza;
- I client devono conoscere indirizzi IP e porte dei servizi, che sono dinamici per natura (mancanza di Location Transparency).
- I client sono costretti a implementare protocolli di backend specifici;
- Modifiche architetturali richiedono aggiornamenti alle implementazioni lato client (mancanza di Distribution transparency).

Per risolvere questi problemi, si è deciso di implementare nel sistema un API Gateway, che agisce come unico punto di ingresso per tutte le richieste dai client.

Questa soluzione consente di nascondere la complessità interna e disaccoppiare il client dall'implementazione backend, offrendo Distribution Transparency, Location Transparency e Replication Transparency.

L'API Gateway è stato realizzato integrando nel sistema un container NGINX.

NGINX è ottimale in quanto è un web server open-source ad alte prestazioni, in grado di gestire migliaia di connessioni concorrenti grazie a un'architettura asincrona e non bloccante. Questo componente opera come reverse proxy e web server, con le seguenti responsabilità:

- ascoltare richieste HTTP sulla porta 80;
- porsi frontalmente rispetto a tutti i microservizi;
- instradare le richieste verso il microservizio corretto in base al path richiesto;
- servire i contenuti statici dell'interfaccia web.

L'API Gateway è collegato alla rete interna `app_net`, all'interno della quale sono presenti anche i microservizi applicativi.

La logica di instradamento è definita nel file `routing.conf`, che separa chiaramente le rotte statiche dell'interfaccia web dalle rotte API dei microservizi.

I contenuti statici serviti direttamente da NGINX includono:

- `/` – la homepage dell'applicazione;
- `/registration` – la pagina di registrazione dell'utente;
- `/login` – la pagina di login;
- `/dashboard` – il "cuore" dell'applicazione, da dove possono venire usate tutte le funzionalità previste.

Per quanto riguarda le API:

- le richieste con path `/api/users/**` vengono inoltrate al microservizio `UserManager` sulla porta 8081;
- le richieste con path `/api/**` e `/flights/**` vengono inoltrate al microservizio `DataCollector` sulla porta 8082.

I nomi dei servizi sono risolti tramite il DNS interno di Docker, sfruttando la rete `app_net`, così il client ha un unico endpoint pubblico e non accede direttamente alle porte interne.

Per illustrare il funzionamento dell'API Gateway, si supponga di ricevere una richiesta `http://localhost/api/users/login`, il flusso è il seguente:

- NGINX riceve la richiesta sulla porta 80;
- il path `/api/users` viene riconosciuto e associato al microservizio `UserManager`;
- la richiesta viene inoltrata a `user-manager:8081`;
- lo `UserManager` elabora la richiesta e produce la risposta;
- NGINX restituisce la risposta al client.

In questo modo, si ottiene un gateway semplice, che centralizza il routing, proteggendo la topologia interna, e servendo l'interfaccia web dallo stesso punto di ingresso.

Docker Compose Aggiornato

A seguito dell'introduzione dei nuovi requisiti, il file `docker-compose.yml` è stato esteso includendo ulteriori container che implementano i nuovi meccanismi.

È stato innanzitutto aggiunto il container `kafka`, configurato in modalità `KRaft single-node` (quindi senza l'uso di `ZooKeeper`), ed è collegato alla rete `app_net` per consentire la comunicazione con i microservizi applicativi. Il broker espone la porta 9092 verso l'host per debugging, mentre i container interni comunicano tramite il listener interno sulla porta 29092. Anche `Kafka` è associato a un volume persistente (`kafka_data`) per garantire la durabilità dei messaggi e dei log.

Il container `data-collector` è stato aggiornato per integrare `Kafka` come producer. Oltre alle variabili d'ambiente già presenti per la connessione al DB e allo `User Manager`, sono stati aggiunti i parametri relativi al broker `Kafka` e ai topic utilizzati (`to-alert-system` e `to-notifier`). Il servizio resta connesso sia alla rete `app_net` sia alla rete `data_db_net`, mantenendo l'isolamento del database.

È stato introdotto il container `alert-system` come consumer del topic `to-alert-system` e come producer del topic `to-notifier`. Questo servizio è responsabile della logica di verifica delle soglie configurate dagli utenti (`high-value` e `low-value`). Per svolgere tale funzione, è connesso sia alla rete `app_net`, per comunicare con `Kafka`, sia alla rete `data_db_net`, per accedere in modo sicuro al `Data DB` e recuperare le informazioni sugli interessi degli utenti. L'uso di un consumer group dedicato consente future estensioni in termini di scalabilità.

Un ulteriore container aggiunto è `alert-notifier`, che consuma i messaggi pubblicati sul topic `to-notifier` e, per ogni condizione rilevata, invia una email all'utente interessato. Le credenziali SMTP e i parametri sensibili non sono hardcoded, ma caricati tramite un file `.env`, migliorando la sicurezza e la portabilità del sistema. Anche questo container è collegato esclusivamente alla rete `app_net`, poiché non necessita di accesso diretto al database.

Infine, è stato introdotto il container `api-gateway`, basato su `NGINX`. Il gateway espone la porta 80 verso l'esterno e instrada le richieste HTTP verso i microservizi appropriati (`User Manager`, `Data Collector` e componenti di supporto), sfruttando il DNS interno di Docker sulla rete `app_net`.

REST API Aggiornate

Il `Data Collector` espone le seguenti nuove API:

- `POST /list_airports_with_thresholds`: restituisce la lista degli aeroporti di interesse dell'utente con le soglie configurate (campi `high_value` e `low_value`).
- `POST /update_thresholds`: imposta o azzerà le soglie per un aeroporto seguito dall'utente; i campi vuoti vengono interpretati come `null` per rimuovere la soglia.

Schema di Comunicazione Dettagliato

Viene presentato uno schema di comunicazione completo che illustra l'intero flusso del sistema: dall'aggiornamento dei dati sui voli, alla pubblicazione dei messaggi sui topic Kafka, fino alla trasmissione della notifica email all'utente finale.

